Николай Прохоренок
Владимир Дронов

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Python 3 и PyQt 6

Разработка приложений

- Типы данных Python
- Объектно-ориентированное программирование
- Работа с файлами и каталогами
- Взаимодействие с Windows
- Создание оконных программ
- Работа с базами данных
- Мультимедиа
- Запись звука, видео и фото
- Печать и экспорт в формат PDF
- Работающий пример: приложение «Судоку»

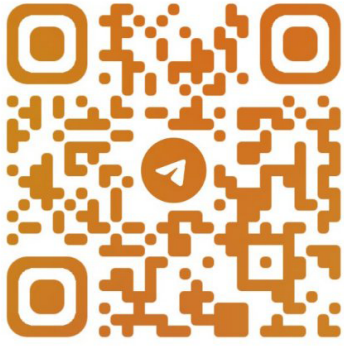
Материалы на www.bhv.ru

bhv

Николай Прохоренок
Владимир Дронов

Python 3 и PyQt 6

Разработка приложений



Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.43
ББК 32.973.26-018.1
П84

Прохоренок, Н. А.

П84 Python 3 и PyQt 6. Разработка приложений / Н. А. Прохоренок, В. А. Дронов. — СПб.: БХВ-Петербург, 2023. — 832 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-1706-5

Описан язык Python 3: типы данных, операторы, условия ветвления и выбора, циклы, регулярные выражения, функции, классы, работа с файлами и каталогами, взаимодействие с механизмами Windows, часто используемые модули стандартной библиотеки. Особое внимание уделено библиотеке PyQt, позволяющей создавать приложения с графическим интерфейсом. Описаны средства для создания и вывода окон, основных компонентов (кнопок, полей, списков, таблиц, меню, панелей инструментов и др.). Рассмотрена обработка событий и сигналов, разработка многопоточных программ, работа с базами данных, вывод графики, воспроизведение мультимедиа, запись аудио, видео и фото, печать документов, экспорт их в формат Adobe PDF и сохранения настроек программ. Дан пример полнофункционального приложения для создания и решения головоломок судоку. На сайте издательства размещен электронный архив со всеми примерами из книги.

Для программистов

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Марины Дамбиевой*
Оформление обложки	*Зои Канторович*

Подписано в печать 01.07.22.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 67,08.
Тираж 1300 экз. Заказ № 4691.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-1706-5

Оглавление

Предисловие

Python

Python — это высокоуровневый, объектно-ориентированный, тьюринг-полный, интерпретируемый язык программирования, предназначенный для решения самого широкого круга задач. С его помощью можно обрабатывать числовую и текстовую информацию, создавать изображения, работать с базами данных, разрабатывать веб-сайты и оконные программы. Python — язык кросс-платформенный, он позволяет создавать программы, которые будут работать во всех операционных системах.

Согласно официальной версии, название языка произошло вовсе не от змеи. Создатель языка, Гвидо ван Россум (Guido van Rossum), назвал свое творение в честь британского комедийного телешоу Би-би-си «Летающий цирк Монти Пайтона» (Monty Python's Flying Circus). Поэтому правильно название этого замечательного языка должно звучать как «Пайтон».

Программа на языке Python представляет собой обычный текстовый файл с расширением py (консольная программа) или pyw (программа с графическим интерфейсом). Все инструкции из этого файла выполняются интерпретатором построчно. Для ускорения работы при первом импорте модуля создается промежуточный байт-код, который сохраняется в оперативной памяти или одноименном файле с расширением pyc. Для выполнения низкоуровневых операций и задач, требующих высокой скорости работы, можно написать модуль на языке C или C++, скомпилировать его, а затем подключить к программе.

Поскольку Python, как было только что отмечено, является языком объектно-ориентированным, практически все данные в нем представляются объектами — даже значения, относящиеся к элементарным типам данных, наподобие чисел и строк, а также сами типы данных. При этом в переменной всегда сохраняется только ссылка на объект, а не сам объект. Например, можно создать функцию, сохранить ссылку на нее в переменной, а затем вызвать функцию через эту переменную. Такое обстоятельство делает язык Python идеальным инструментом для создания программ, использующих функции обратного вызова, — например, при разработке графического интерфейса.

Python — самый стильный язык программирования в мире, он не допускает двоякого написания кода. В Python отсутствуют лишние языковые конструкции, и код можно написать только одним способом. Все программисты, работающие с языком Python, должны придерживаться стандарта оформления программного кода, описанного в документе PEP-8 (**https://peps.python.org/pep-0008/**). Соответственно, более читаемого кода нет ни в одном ином языке программирования.

Поскольку программа на языке Python представляет собой текстовый файл, Python-код можно писать в любом текстовом редакторе (например, Блокноте, поставляемом в составе Windows). Однако лучше использовать какой-либо редактор, специально предназначенный для программистов: Visual Studio Code (**https://code.visualstudio.com/**), PyCharm (**https://www.jetbrains.com/ru-ru/pycharm/**), Atom (**https://atom.io/**), Sublime Text (**https://www.sublimetext.com/**), Brackets (**https://brackets.io/**), Notepad++ (**https://notepad-plus-plus.org/**) и др. Мы же в процессе изучения материала этой книги будем пользоваться утилитой IDLE, которая позволяет исполнять инструкции языка интерактивно и входит в состав стандартной поставки Python.

PyQt

Библиотека PyQt служит для создания кросс-платформенных оконных программ с графическим интерфейсом. Библиотека очень проста в использовании и идеально подходит для разработки весьма серьезных программ. Пользуясь исключительно ее средствами, мы можем выводить на экран графику практически любой сложности, работать с базами данных наиболее распространенных форматов, воспроизводить мультимедийные файлы, записывать звук и видео, делать фото, выводить документы на печать и экспортировать их в популярный формат Adobe PDF. Библиотека PyQt основана на популярном фреймворке Qt и служит своего рода связкой между ним и Python.

В самом конце книги мы с вами самостоятельно напишем на языке Python с применением библиотеки PyQt полнофункциональную программу «Судоку», предназначенную для создания и решения одноименных головоломок.

Использованное ПО

Авторы применяли при работе над книгой следующее ПО:

- Microsoft Windows 10, русская 64-разрядная редакция со всеми установленными обновлениями;
- Python — 3.10.1, 64-разрядная редакция;
- winpath — 202002.24
- pyshortcuts — 1.8.1;
- PyQt — 6.2.3;
- Qt Designer — 5.11.1;
- PyQt6-WebEngine — 6.2.1.

Типографские соглашения

В книге будут часто приводиться форматы написания различных языковых конструкций, применяемых в Python. В них использованы особые типографские соглашения, приведенные далее.

- Программный код набирается моноширинным шрифтом. Например:

```
print("Привет, мир!")
input()
```

- В угловые скобки (<>) заключаются наименования различных значений, подставляемых в исходный код (к примеру, параметров функций и методов). Например:

  ```
  type(<Значение>)
  ```

  Здесь вместо слова `<Значение>` должно быть подставлено реальное значение.

- В наименованиях параметров часто присутствуют типы значений, указываемых в качестве параметров. Например:

  ```
  setPixmap(<Изображение QPixmap>)
  ```

  Здесь в качестве параметра должно быть указано изображение, представленное объектом класса `QPixmap`. Еще пример:

  ```
  setAlignment(<Выравнивание AlignmentFlag>)
  ```

  Здесь в качестве параметра указывается элемент перечисления `AlignmentFlag`, задающий выравнивание.

- В квадратные скобки (`[]`) заключаются фрагменты кода, необязательные к указанию. Например:

  ```
  input([<Сообщение>])
  ```

  Здесь `<Сообщение>` может быть указано, а может и не указываться.

- Значения по умолчанию у необязательных параметров функций и методов ставятся после наименований параметров через знак равенства (=). Например:

  ```
  blake2s(<Значение>[, digest_size=32][, salt=b""])
  ```

  Здесь у необязательного параметра `digect_size` значение по умолчанию — 32, а у необязательного параметра `salt` — пустая последовательность байтов типа `bytes`.

- Вертикальной чертой (`|`) разделяются различные варианты языковой конструкции, из которых следует указать лишь какой-то один. Например:

  ```
  python -m compileall <Путь к файлу>|<Путь к каталогу> <Опции>
  ```

  Здесь следует поставить либо `<Путь к файлу>`, либо `<Путь к каталогу>`.

- Слишком длинные, не помещающиеся на одной строке языковые конструкции автор разрывал на несколько строк и в местах разрывов ставил знак ↳. Например:

  ```
  [[<Заполнитель>]<Выравнивание>][<Знак>][#][0][<Ширина>][,][_]↳
  [.<Точность>][<Преобразование>]
  ```

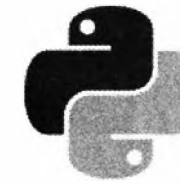
  Приведенный код здесь разбит на две строки, но должен быть набран в одну. Символ ↳ при этом нужно удалить.

Все листинги из этой книги вы найдете в электронном архиве, который можно свободно загрузить с сервера издательства «БХВ» по ссылке: **https://zip.bhv.ru/9785977517065.zip** или со страницы книги на сайте издательства **https://bhv.ru/** (см. *приложение*).

Авторы книги желают вам приятного чтения и надеются, что она станет верным спутником в вашей грядущей карьере программиста!

ЧАСТЬ I

Основы языка Python



ГЛАВА 1

Первые шаги

Прежде чем мы начнем знакомство с языком Python, хочется отметить, что книги по программированию нужно не только читать, — весьма желательно выполнять все имеющиеся в них примеры, а также экспериментировать, что-нибудь в этих примерах изменяя. Поэтому, если вы удобно устроились на диване и настроились просто читать, у вас практически нет шансов изучить язык. Чем больше вы будете делать самостоятельно, тем большему научитесь.

> ***Внимание!***
> Текущая версия Python поддерживает только Microsoft Windows версий 8.1, 10 и 11. Более старые версии этой операционной системы не поддерживаются.

1.1. Установка Python

Вначале необходимо установить на компьютер *интерпретатор* (или *исполняющую среду*) Python — программный пакет, который переводит программы, написанные на этом языке, в представление, «понятное» центральному процессору компьютера. Без интерпретатора ни одну Python-программу запустить не получится.

1. Для загрузки дистрибутива исполняющей среды заходим на страницу **https://www.python.org/downloads/** и в списке доступных версий щелкаем на гиперссылке **Python 3.10.1** (эта версия является наиболее актуальной на момент подготовки книги). На открывшейся странице находим раздел **Files** и щелкаем на гиперссылке **Windows installer (32-bit)** (32-разрядная редакция интерпретатора) или **Windows installer (64-bit)** (его 64-разрядная редакция) — в зависимости от редакции нашей операционной системы. В результате на наш компьютер будет загружен файл python-3.10.1.exe или python-3.10.1-amd64.exe соответственно. Запускаем загруженный файл двойным щелчком на нем.
2. В открывшемся окне (рис. 1.1) проверяем, установлен ли флажок **Install launcher for all users (recommended)** (Установить утилиту launcher для всех пользователей), устанавливаем флажок **Add Python 3.10 to PATH** (Добавить Python 3.10 в список путей из переменной PATH) и нажимаем кнопку **Customize installation** (Настроить установку).

   Утилита launcher, поставляемая в составе дистрибутива, предназначена для запуска программ под определенной версией Python, если на компьютере установлены несколько разных версий этого языка. К сожалению, если ее не установить, не получится ассоциировать файлы Python-программ с исполняющей средой Python, в результате чего запускать программы щелчком мыши станет невозможно.

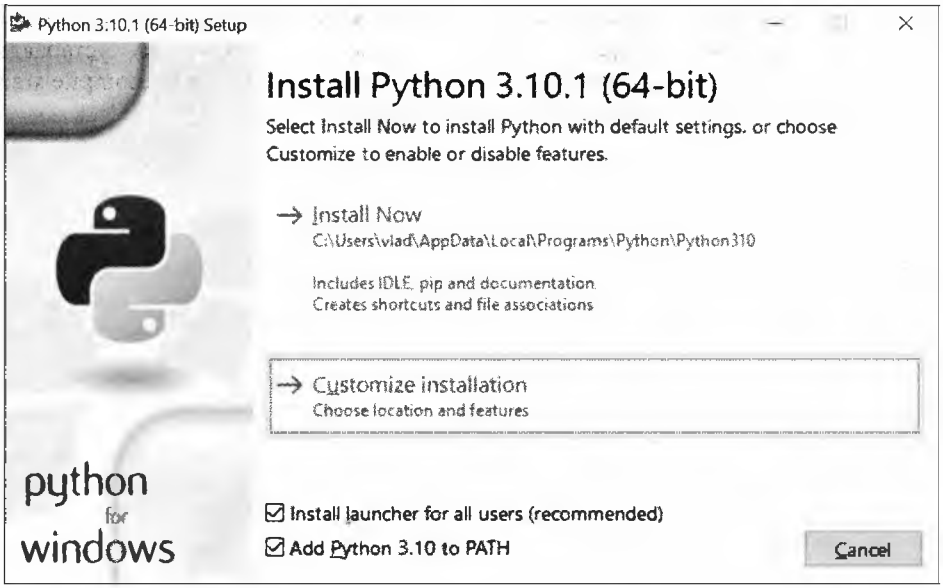


Рис. 1.1. Установка Python: шаг 1

3. В следующем диалоговом окне (рис. 1.2) предлагается выбрать устанавливаемые компоненты. Оставляем установленными все флажки, представляющие эти компоненты, и нажимаем кнопку **Next**.

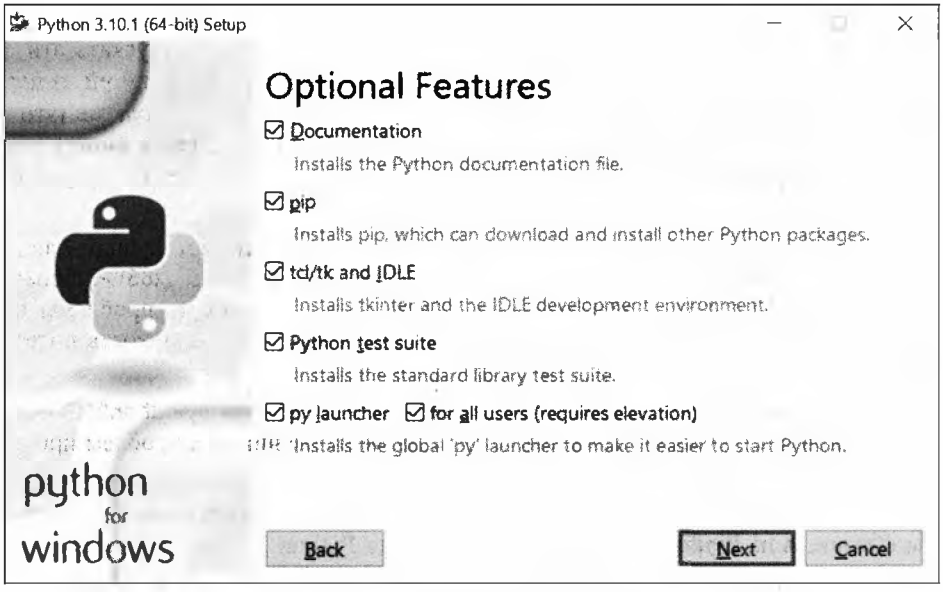


Рис. 1.2. Установка Python: шаг 2

4. На следующем шаге (рис. 1.3) задаем дополнительные настройки и выбираем путь установки. Проверяем, установлены ли флажки **Associate files with Python (requires the py launcher)** (Ассоциировать Python-файлы с исполняющей средой), **Create shortcuts for**

installed applications (Создать ярлыки для установленных приложений), **Add Python to environment variables** (Добавить Python в переменные окружения), устанавливаем флажки **Install for all users** (Установить для всех пользователей) и **Precompile standard library** (Предварительно откомпилировать стандартную библиотеку).

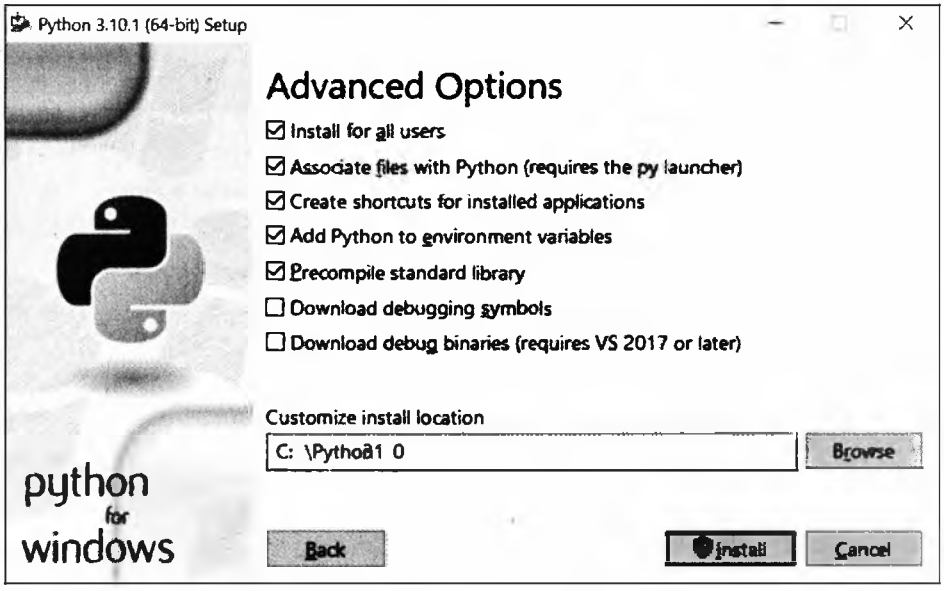


Рис. 1.3. Установка Python: шаг 3

Ассоциирование Python-файлов с исполняющей средой позволит запускать эти файлы щелчками на них. Добавление пути к интерпретатору Python в список путей из переменной `PATH` даст возможность запускать этот интерпретатор набором в *консоли* (это сокращенное название командной строки Windows, применяемое далее в книге) команды `python`. Предварительная компиляция стандартной библиотеки ускорит запуск Python-программ.

Обратите внимание на указание пути, по которому будет установлен Python. Изначально предлагается установить интерпретатор по пути c:\Program Files (x86)\Python310 или c:\Program Files\Python310. Однако в этом случае могут возникнуть проблемы при использовании дополнительных библиотек, расширяющих функциональность интерпретатора.

Поэтому авторы книги рекомендуют установить Python по пути c:\Python310 — т. е. непосредственно в корень диска (см. рис. 1.3). В этом случае никаких проблем при использовании дополнительных библиотек не возникнет.

Задав все необходимые параметры, нажимаем кнопку **Install** и положительно отвечаем на появившееся на экране предупреждение UAC.

5. После завершения установки откроется окно, показанное на рис. 1.4. Нажимаем в нем кнопку **Close** для выхода из программы установки.

В результате на компьютер будут установлены две редакции интерпретатора Python:

- python.exe — предназначена для исполнения консольных программ и задействуется при щелчке мышью на файле с расширением py.

С помощью этой редакции можно выполнять и оконные программы, однако в этом случае на экране появится окно консоли, что может обескуражить пользователя;

- pythonw.exe — служит для исполнения оконных программ и задействуется при щелчке мышью на файле с расширением pyw. Консоль при этом не выводится.

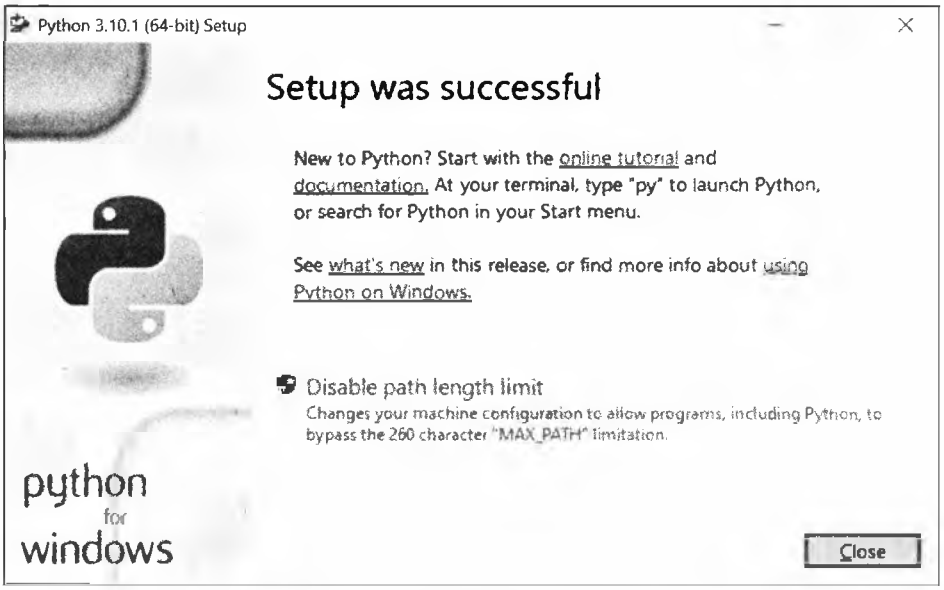


Рис. 1.4. Установка Python: шаг 4

1.2. Интерактивный режим Python. Утилита IDLE

Изучать Python удобнее всего, вводя инструкции этого языка вручную и тотчас получая результаты их выполнения. Для этого интерпретатор Python должен работать в *интерактивном режиме*.

Запустить интерпретатор Python в интерактивном режиме можно одним из следующих способов:

- выбрав в меню **Пуск | Python 3.10** пункт **Python 3.10 (32-bit)** или **Python 3.10 (64-bit)**;
- набрав в консоли `python` и нажав клавишу <Enter>;
- выполнив щелчок мышью (одинарный или двойной — в зависимости от настроек операционной системы) на файле python.exe в каталоге c:\Python310.

В результате на экране появится окно, напоминающее таковое у командной строки Windows (рис. 1.5). Символами >>> в этом окне помечается приглашение для ввода инструкций. Если после этих символов ввести, например, `2 + 2` и нажать клавишу <Enter>, то на следующей строке появится результат сложения — `4`, а затем — приглашение для ввода новой инструкции. Таким образом, интерактивный режим можно использовать и в качестве очень мощного калькулятора.

Однако значительно удобнее учить язык и выполнять вычисления, работая с утилитой IDLE, которая входит в комплект поставки Python. Эта утилита представляет собой своего рода обертку вокруг интерпретатора, работающего в интерактивном режиме, которая

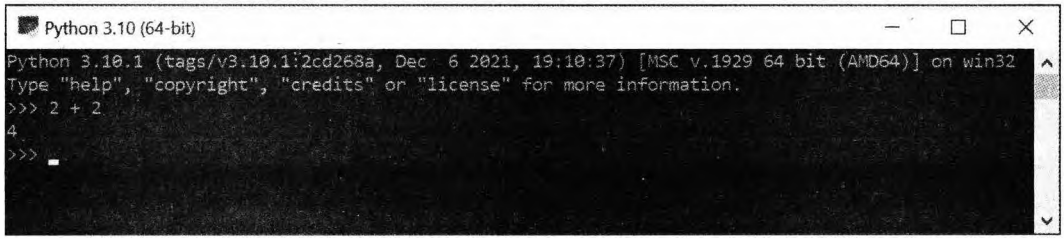


Рис. 1.5. Интерпретатор Python, запущенный в интерактивном режиме

добавляет интерпретатору расширенную функциональность (в частности, синтаксическую подсветку программного кода и вывод подсказок).

Для запуска IDLE достаточно выбрать в меню **Пуск | Python 3.10** пункт **IDLE (Python 3.10 32-bit)** или **IDLE (Python 3.10 64-bit)**. На экране появится окно **IDLE Shell** (рис. 1.6), в котором выводится интерактивный интерпретатор Python.

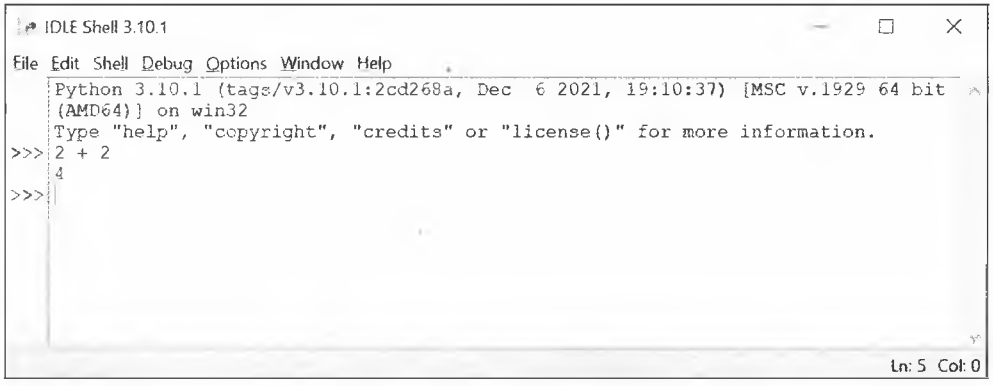


Рис. 1.6. Окно **IDLE Shell** утилиты IDLE

Отметим, что в окне **IDLE Shell** приглашение >>> отображается на свободной полосе, расположенной левее области редактирования, в которой вводятся инструкции Python и выводятся результаты. Поэтому при копировании кода в буфер обмена комбинацией клавиш <Ctrl>+<C> приглашение в него не попадает.

Кроме того, IDLE позволяет работать с Python-программами, хранящимися в файлах. Содержимое таких файлов выводится в отдельных окнах. Более подробно функциональные возможности этой утилиты будут рассмотрены позже.

> ***Внимание!***
>
> В дальнейшем для написания и выполнения Python-программ мы будем использовать утилиту IDLE.

1.3. Введение в Python-программирование

Изучение языков программирования принято начинать с программы, выводящей надпись «Привет, мир!» Не будем нарушать традиции и продемонстрируем, как это будет выглядеть на Python (листинг 1.1).

Листинг 1.1. Первая программа на Python

```
# Выводим надпись с помощью функции print()
print("Привет, мир!")
```

Эта программа состоит из двух *инструкций* (или *выражений*) — предписаний, указывающих Python выполнить какое-либо законченное действие и записываемых на отдельных строках. Первая инструкция представляет собой комментарий — пометку, написанную самим программистом с целью объяснить себе или коллегам, что делает тот или иной код. Вторая инструкция выводит в консоли строку «Привет, мир!»

Вывод значения во второй инструкции выполняется посредством функции `print()`. *Функцией* называется языковая конструкция, выполняющая над переданными ей значениями (*параметрами*) какое-либо относительно сложное действие (в нашем случае — вывод в консоль). Параметры записываются внутри круглых скобок и отделяются друг от друга запятыми. В листинге 1.1 функции `print()` передается всего один параметр — выводимое значение.

Запускаем интерпретатор Python в интерактивном режиме (как это сделать, было описано в *разд. 1.2*). Правее приглашения >>> вводим сначала первую строку из листинга 1.1, а затем вторую. После ввода каждой строки нажимаем клавишу <Enter>. В третьей строке сразу отобразится результат, а далее — приглашение для ввода новой команды. Мы увидим следующее (напоминаем, что символы >>> — это приглашение, их вводить не нужно):

```
>>> # Выводим надпись с помощью функции print()
>>> print("Привет, мир!")
    Привет, мир!
>>>
```

> ***Внимание!***
>
> В дальнейшем в подобных фрагментах кода инструкции, вводимые вручную, будут помечаться расположенными левее символами приглашения >>>, а результаты, выведенные интерпретатором, показываться без этих символов.

Для создания файла с Python-программой в меню **File** окна **IDLE Shell** выбираем пункт **New File** или нажимаем комбинацию клавиш <Ctrl>+<N>. В открывшемся окне набираем код из листинга 1.1, а затем сохраняем его в каком-либо каталоге в файле с именем hello_world.py, выбрав пункт меню **File | Save** (или нажав комбинацию клавиш <Ctrl>+<S>).

Кстати, файл с Python-кодом в терминологии этого языка называется *модулем*.

Запустить написанную программу на выполнение можно, выбрав в окне с кодом этой программы пункт меню **Run | Run Module** или нажав клавишу <F5>. Результат выполнения программы будет отображен в окне **IDLE Shell**.

Запустить Python-программу вне среды IDLE можно двумя способами:

- набрав имя хранящего ее файла вместе с расширением (например: `hello_world.py`) в консоли и нажав клавишу <Enter>.

  Результат выполнения будет выведен там же, в консоли;

- выполнив щелчок мышью (двойной или одинарный — в зависимости от настроек системы) на значке файла с этой программой.

  В этом случае после вывода результата окно консоли сразу закроется. Чтобы предотвратить его закрытие, необходимо добавить в конец кода программы вызов функции

input(), которая станет ожидать нажатия клавиши <Enter> и не позволит окну сразу закрыться. С учетом сказанного наша первая программа будет выглядеть так, как показано в листинге 1.2.

Листинг 1.2. Программа для запуска с помощью щелчка мыши

```
print("Привет, мир!")      # Выводим строку
input()                    # Ожидаем нажатия клавиши <Enter>
```

Отметим, что функция input() в листинге 1.2 вызывается без параметров. В таком случае после имени функции ставятся пустые круглые скобки.

> ***Примечание***
>
> Если до выполнения функции input() в коде возникнет ошибка, то сообщение о ней будет выведено в консоли, но последняя после этого сразу закроется, и вы не сможете прочитать сообщение об ошибке. Попав в подобную ситуацию, запустите программу из консоли или утилиты IDLE, и вы сможете прочитать это сообщение.

Чтобы открыть Python-файл для редактирования, запустим IDLE, выберем пункт меню **File | Open** (комбинация клавиш <Ctrl>+<O>) и выберем нужный файл в появившемся на экране диалоговом окне открытия файла. Файл будет открыт в новом окне утилиты IDLE.

1.4. Принципы написания Python-программ

- Программа, написанная на Python, представляет собой обычный текстовый файл с расширением py (будет исполнен обычным интерпретатором python.exe) или pyw (будет исполнен «оконным» интерпретатором pythonw.exe).

  Для написания Python-программ, как уже ранее отмечалось, можно использовать, помимо IDLE, любой подходящий текстовый редактор: стандартный Блокнот, Notepad++ (**https://notepad-plus-plus.org/**), Visual Studio Code (**https://code.visualstudio.com/**) и др. Также существует ряд специализированных текстовых редакторов, предназначенных именно для Python-программистов, в частности PyCharm[1] (**https://www.jetbrains.com/ru-ru/pycharm/**).
- Инструкции, записанные в коде программы, выполняются последовательно, в порядке сверху вниз.
- Каждая инструкция Python должна располагаться на отдельной строке. Признаком конца инструкции является перевод строки. Пример программы из двух выражений приведен в листинге 1.3.

Листинг 1.3. Программа, выполняющая арифметические вычисления

```
x = 15 + 20 + 30
print(x)
```

Первое выражение складывает числа 15, 20 и 30 и заносит сумму в переменную с именем x (*переменная* — ячейка памяти, имеющая уникальное имя, способная хранить ка-

[1] Доступны как базовая бесплатная, так и расширенная платная редакции этого редактора.

кое-либо значение — например, число, и создаваемая при присваивании ей значения). Второе выражение выводит содержимое этой переменной — число 65 — в консоли.

В первом выражении используются два оператора. *Оператором* называется языковая конструкция, выполняющая над переданными ему значениями какое-либо элементарное действие (например, сложение чисел или присваивание результата указанной переменной). Оператор сложения обозначается привычным символом +, а оператор присваивания — символом =.

В Windows перевод строки формируется комбинацией символов `\r` (перевод каретки) и `\n` (перевод строки), в UNIX — одним символом `\n`.

Если загрузить файл Python-программы по протоколу FTP в бинарном режиме, то символ `\r` вызовет фатальную ошибку. По этой причине файлы по протоколу FTP следует загружать только в текстовом (ASCII) режиме — тогда символ `\r` будет удален автоматически.

- Если строка с инструкцией получилась слишком длинной, инструкцию можно разделить на несколько строк одним из следующих способов:
  - поставить в месте разрыва строк символ \, сразу после которого вставить перевод строки (как обычно, нажатием клавиши <Enter>):

```
x = 15 + 20 + \
    30
print(x)
```

    Как правило, вторая и все последующие строки, содержащие длинное выражение, набираются с отступами слева — для улучшения читаемости кода. Следует только формировать эти отступы *исключительно пробелами* и делать *одинаковой длины* (содержащими одинаковое количество пробелов).

    При вводе подобного рода многострочной инструкции в интерпретаторе, работающем в интерактивном режиме, строки с продолжением инструкции помечаются расположенным слева приглашением в виде трех точек (..., а не >>>, как обычно). Кроме того, интерпретатор сам выводит последующие строки с отступами. Чтобы завершить ввод многострочной инструкции, следует, введя ее последнюю строку, нажать клавишу <Enter> (при этом после многострочной инструкции будет создана пустая строка с приглашением из трех точек). Пример:

```
>>> x = 15 + 20 + \   # Вводим первую строку многострочной инструкции
...     30             # Вводим вторую, последнюю, строку
...           # Завершаем ввод многострочной инструкции, нажав <Enter>
>>> print(x)          # Вводим следующую инструкцию
```

  - заключить выражение в круглые скобки, внутри которых вставить нужное количество переводов строки:

```
x = (15 +
     20 +
     30)
print(x)
```

  - определение списка и словаря (описываются в *главах 8* и *9*), при оформлении которых используются квадратные и фигурные скобки соответственно, также можно разбить на несколько строк:

пример определения списка:

```
arr = [15, 20,
       30]
print(arr)
```

пример определения словаря:

```
arr = {"x": 15, "y": 20,
       "z": 30}
print(arr)
```

- Короткие инструкции можно записать в одной строке, разделив их символами точки с запятой (`;`):

```
x = 15 + 20 + 30; print(x)
```

После точки с запятой не возбраняется ставить пробел — для улучшения читаемости кода.

- *Блоки* (наборы из произвольного количества инструкций, входящие в состав более сложных инструкций) формируются *отступами слева*. Такие отступы должны формироваться *исключительно пробелами* и быть *одинаковой длины* (т. е. содержать одинаковое количество пробелов). Символы табуляции в отступах не допускаются и при выполнении программы вызывают возникновение ошибки.

Листинг 1.4 показывает код программы, выводящей последовательно числа от 1 до 10, которые разделяются тремя дефисами.

Листинг 1.4. Пример блока

```
for i in range(1, 11):
    print(i)
    print("---")
```

Первая строка содержит сложную инструкцию — цикл (описывается в *главе 4*), который выполняется 10 раз. При каждом исполнении он заносит в переменную `i` очередное число в диапазоне от 1 до 10 и выполняет блок, входящий в его состав. Этот блок состоит из двух инструкций, записанных во второй и третьих строках: первая инструкция выводит в консоли число из переменной `i`, а вторая — строку из трех дефисов. Обе инструкции, входящие в блок, имеют одинаковый отступ слева, содержащий 4 пробела.

Числа, перебираемые циклом (и, соответственно, количество выполнений, или *итераций*, цикла), указываются в функции `range()` (первая инструкция в листинге 1.4). Первый параметр задает начальное число перебираемого диапазона, второй — конечное число плюс 1.

Примечание

4 пробела — общепринятая величина отступа в Python.

При вводе инструкции, содержащей блок, в интерпретаторе, который работает в интерактивном режиме, интерпретатор предваряет выражения, входящие в блок, приглашением в виде трех точек (...) и сам вставляет отступы. Чтобы завершить ввод блока, следует, занеся последнее входящее в него выражение, нажать клавишу <Enter> (при этом в набранном коде появится пустая строка с приглашением в виде трех точек). Пример:

```
>>> for i in range(1, 11):
...     print(i)
...     print("---")
...
>>> # Следующая инструкция
```

- Если блок содержит одну короткую инструкцию, и сам блок, и сложную инструкцию, в состав которой он входит, можно записать в одну строку. При этом интерпретатор посчитает, что ввод блока продолжится после нажатия клавиши <Enter>, и выведет приглашение в виде трех точек. Чтобы завершить ввод инструкции, следует снова нажать <Enter>. Пример программы, выводящей в консоли числа от 1 до 10:

```
>>> for i in range(1, 11): print(i)
...
... Вывод пропущен ...
```

1.4.1. Комментарии и строки документирования

Комментарий — это произвольное пояснение, вставленное в код программы, предназначенное исключительно программисту и полностью игнорируемое интерпретатором. Внутри комментария может располагаться любой текст.

Комментарий в языке Python начинается с символа # и заканчивается концом строки:

```
# Выводим числа от 1 до 10
for i in range(1, 11): print(i)
```

Комментарий может располагаться после собственно инструкции:

```
print("Все, числа закончились") # Сообщаем об окончании чисел
```

Если символ # разместить перед инструкцией, то она не будет выполнена (*закомментированная инструкция*):

```
# print("Привет, мир!") Эта инструкция выполнена не будет
```

Если требуется разместить комментарий из нескольких строк, перед каждой из них придется ставить символ #:

```
# Это наша первая программа!
# Она выводит числа от 1 до 10
# Да, не впечатляет, но для начала неплохо...
for i in range(1, 11): print(i)
```

Строки документирования Python обычно применяются для написания инструкций к программам и модулям (подробности — в *главе 6*). Однако их можно использовать и для комментирования кода.

Строка документирования заключается в утроенные кавычки или апострофы:

```
"""
Это наша первая программа!
Она выводит числа от 1 до 10
Да, не впечатляет, но для начала неплохо...
"""
for i in range(1, 11): print(i)
```

1.4.2. Кодировки, поддерживаемые Python

Код Python-программы, написанный в IDLE, по умолчанию сохраняется в кодировке UTF-8 без BOM[1].

Если программу следует сохранить в какой-либо другой кодировке (что может пригодиться, например, при переписывании старого Python-кода), в первой строке ее кода следует указать кодировку с помощью инструкции формата:

```
# -*- coding: <Обозначение кодировки> -*-
```

Например, кодировка Windows-1251 указывается инструкцией:

```
# -*- coding: cp1251 -*-
```

Встретив такую инструкцию в коде программы, IDLE при сохранении файла самостоятельно переведет программу в кодировку с заданным обозначением. При использовании других редакторов следует перевести программу в указанную кодировку вручную.

Получить полный список поддерживаемых Python кодировок и их обозначения позволяет программа, приведенная в листинге 1.5.

Листинг 1.5. Вывод списка поддерживаемых кодировок

```
import encodings.aliases
arr = encodings.aliases.aliases
keys = list( arr.keys() )
keys.sort()
for key in keys: print("%s => %s" % (key, arr[key]))
```

1.4.3. Подготовка Python-программ для выполнения в UNIX

Если программа предназначена для исполнения в операционных системах семейства UNIX, в первой строке кода программы необходимо указать путь к интерпретатору Python:

```
#!/usr/bin/python
```

В некоторых UNIX-системах путь к интерпретатору выглядит по-другому:

```
#!/usr/local/bin/python
```

Иногда можно не указывать точный путь к интерпретатору, а передать название языка программе `env`:

```
#!/usr/bin/env python
```

В этом случае программа `env` произведет поиск интерпретатора Python в соответствии с настройками путей поиска.

Если программа, исполняемая в UNIX, сохранена в кодировке, отличной от UTF-8, обозначение кодировки указывается во второй строке ее кода:

```
#!/usr/bin/python
# -*- coding: cp1251 -*-
```

[1] *BOM* (Byte Order Mark) — метка порядка байтов. Указывает порядок, в котором записываются байты, кодирующие символы в UTF-8.

Также следует разрешить выполнять Python-программу, указав у ее файла права 755 (`-rwxr-xr-x`).

1.5. Дополнительные возможности IDLE

Поскольку далее для работы мы будем использовать утилиту IDLE, рассмотрим предлагаемые ею дополнительные возможности.

Обычно после ввода инструкции и нажатия клавиши <Enter> введенная инструкция выполняется, и на следующей строке выводится ее результат (при условии, что инструкция возвращает результат). После чего появляется приглашение для ввода новой инструкции.

При вводе сложной инструкции, содержащей блок, после нажатия клавиши <Enter> интерпретатор, работающий в интерактивном режиме, автоматически создаст отступ и будет ожидать ввода блока. Чтобы сообщить интерпретатору об окончании ввода инструкции, необходимо нажать <Enter> дважды:

```
>>> for n in range(1, 3):
...     print(n)
...
1
2
>>>
```

Если ввести какое-либо значение — например, строку или число, и нажать <Enter>, это значение появится в следующей строке:

```
>>> "Привет, мир!"
'Привет, мир!'
>>>
```

Обратите внимание на то, что строки выводятся в апострофах. Этого не произойдет, если выводить строку с помощью функции `print()`:

```
>>> print("Привет, мир!")
Привет, мир!
>>>
```

Как говорилось в *разд. 1.2*, окно **IDLE Shell** можно использовать для изучения языка, а также в качестве многофункционального калькулятора (здесь `*` — это оператор умножения):

```
>>> 12 * 32 + 54
438
>>>
```

Результат вычисления последней инструкции сохраняется в переменной _ (одно подчеркивание). Это позволяет производить дальнейшие расчеты без ввода предыдущего результата — вместо него достаточно ввести символ подчеркивания. Пример (здесь / — оператор деления):

```
>>> 125 * 3       # Умножение
375
>>> _ + 50        # Сложение. Эквивалентно 375 + 50
425
```

```
>>> _ / 5          # Деление. Эквивалентно 425 / 5
    85.0
>>>
```

При вводе команды можно воспользоваться комбинацией клавиш <Ctrl>+<Пробел>. В результате будет отображен список, из которого можно выбрать нужную языковую конструкцию. Если при открытом списке начать вводить буквы, то показываться будут языковые конструкции, начинающиеся с этих букв. Выбирать конструкцию необходимо с помощью клавиш <↑> и <↓>. После выбора не следует нажимать клавишу <Enter>, иначе это приведет к выполнению инструкции, — просто вводите инструкцию дальше, а список закроется. Такой же список будет автоматически появляться (с некоторой задержкой) при обращении к атрибутам объекта или модуля после ввода точки. Для автоматического завершения языковой конструкции после ввода первых букв можно воспользоваться комбинацией клавиш <Alt>+</>. При каждом последующем нажатии этой комбинации будет вставляться следующая конструкция. Эти две комбинации клавиш очень удобны, если вы забыли, как пишется слово, или хотите, чтобы редактор закончил его за вас.

Иногда возникает необходимость выполнить ранее введенную инструкцию повторно. Для таких случаев IDLE предоставляет комбинации клавиш <Alt>+<N> (вставка первой введенной инструкции) и <Alt>+<P> (вставка последней инструкции). Каждое последующее нажатие этих клавиш будет вставлять следующую (или предыдущую) инструкцию. Для еще более быстрого повторного ввода инструкции следует предварительно ввести ее первые буквы — в этом случае перебираться будут только инструкции, начинающиеся с этих букв.

1.6. Вывод данных

Вывести заданные значения можно с помощью функции `print()`:

```
print([<Значения через запятую>][, sep=' '][, end='\n'][, file=sys.stdout][,
                                                        flush=False])
```

Указанные значения при необходимости преобразуются в строки и посылаются в стандартный поток вывода `stdout`. С помощью параметра `file` можно перенаправить вывод в другое место — например, в файл. При этом, если параметр `flush` имеет значение `False`, выводимые значения будут записаны в файл принудительно. Перенаправление вывода мы подробно рассмотрим при изучении файлов.

Если выполняется вывод одного значения, автоматически добавляется символ перевода строки:

```
print("Строка 1")
print("Строка 2")
```

Результат:

```
Строка 1
Строка 2
```

Можно вывести несколько значений в одну строку, указав их в вызове функции `print()` отдельными параметрами, разделенными запятыми:

```
print("Строка 1", "Строка 2")
```

Результат:

```
Строка 1 Строка 2
```

Как видно из примера, между выводимыми значениями автоматически вставляется пробел. С помощью параметра sep можно указать другой разделяющий символ. Например, выведем строки без пробела между ними, указав в качестве разделителя пустую строку:

```
print("Строка1", "Строка2", sep="")
```

Результат:

```
Строка 1Строка 2
```

Ряд функций, встроенных в Python, поддерживают так называемые *именованные параметры*. В число таких параметров и входит sep. Обратим внимание, как задается его значение в приведенном примере.

После вывода нескольких значений в одном вызове функции print() в конце добавляется символ перевода строки. Если необходимо произвести дальнейший вывод на той же строке, то в именованном параметре end следует указать другой символ:

```
print("Строка 1", "Строка 2", end=" ")
print("Строка 3")
# Выведет: Строка 1 Строка 2 Строка 3
```

Вызов функции print() без параметров выводит пустую строку:

```
for n in range(1, 5):
    print(n, end=" ")
print()
print("Это текст на новой строке")
```

Результат выполнения:

```
1 2 3 4
Это текст на новой строке
```

Здесь мы использовали цикл, выполняющийся четыре раза. На каждой итерации он присваивает переменной n число от 1 до 4 и выполняет блок, содержащий вызов функции print(), которая выводит число из переменной n. Не забываем вставить в выражение, входящее в блок, отступ слева.

Как только цикл выполнится четыре раза, будут исполнены два следующих за ним выражения. В них отступы слева указывать не следует, поскольку эти выражения не должны входить в состав блока. Первое выражение выведет пустую строку, второе — надпись «Это текст на новой строке».

Если необходимо вывести большой блок текста, его следует разместить между утроенными кавычками или утроенными апострофами. В этом случае текст сохраняет свое форматирование:

```
print("""Строка 1
Строка 2
Строка 3""")
```

В результате выполнения этого примера мы получим три строки:

```
Строка 1
Строка 2
Строка 3
```

Вместо функции `print()` можно использовать метод `write()` объекта `stdout` из модуля `sys` (методы очень похожи на функции и в подробностях, вместе с объектами, будут рассмотрены позже):

```
import sys                          # Подключаем модуль sys, чтобы использовать
                                    # содержащийся в нем объект stdout
sys.stdout.write("Строка")
```

Как мы помним из *разд. 1.3*, модуль — это просто файл с Python-кодом. Однако модуль `sys` поставляется в составе Python и входит в *стандартную библиотеку* (набор модулей, содержащих полезные функции, объекты и др.) этого языка. Подключив этот модуль, мы можем использовать созданные в нем функции и объекты.

В первой строке с помощью оператора `import` подключаем модуль `sys`, в котором объявлен объект `stdout`. Далее с помощью метода `write()` этого объекта выводим строку. Следует заметить, что метод не вставляет символ перевода строки, поэтому при необходимости следует добавить его самим с помощью символа `\n`:

```
import sys
sys.stdout.write("Строка 1\n")
sys.stdout.write("Строка 2")
```

Метод `write()` возвращает *результат* — значение, полученное в процессе выполненных методом вычислений. Таковым результатом является количество символов в выведенной строке. Его можно присвоить какой-либо переменной и использовать в дальнейшем:

```
import sys
cnt = sys.stdout.write("Привет, Python\n")
print("Символов в выведенной строке: ", cnt)
```

Результат:

```
Привет, Python
Символов в выведенной строке:  15
```

1.7. Ввод данных

Для ввода данных в Python предназначена функция `input()`, которая получает данные со стандартного потока ввода `stdin`. Функция имеет следующий формат:

```
input([<Сообщение>])
```

Введенное значение она возвращает в качестве результата. Этот результат следует присвоить какой-либо переменной посредством оператора =.

Для примера переделаем нашу первую программу так, чтобы она здоровалась не со всем миром, а только с нами (листинг 1.6).

Листинг 1.6. Пример использования функции `input()`

```
name = input("Введите ваше имя: ")
print("Привет,", name)
input("Нажмите <Enter> для закрытия окна")
```

В первой инструкции значение, введенное пользователем и полученное функцией `input()`, присваивается переменной `name`.

Сохраняем программу в файле test.py и запускаем на выполнение с помощью двойного щелчка. Откроется черное окно, в котором мы увидим надпись: **Введите ваше имя:**. Вводим свое имя — например, `Николай`, и нажимаем клавишу <Enter>. В результате будет выведено приветствие: **Привет, Николай**.

При использовании функции `input()` следует учитывать, что при достижении конца файла или при нажатии комбинации клавиш <Ctrl>+<Z>, а затем клавиши <Enter> генерируется исключение `EOFError`. Если не предусмотреть обработку исключения, то программа аварийно завершится. Обработать исключение можно следующим образом:

```
try:
    s = input("Введите данные: ")
    print(s)
except EOFError:
    print("Обработали исключение EOFError")
```

Если внутри блока `try` возникнет исключение `EOFError`, то управление будет передано в блок `except`. После исполнения инструкций в блоке `except` программа нормально продолжит работу.

Передать данные можно в консоли, указав их после имени файла программы. Такие данные доступны через список `argv` из модуля `sys`. Первый элемент списка `argv` будет содержать имя файла запущенной программы, а последующие элементы — переданные данные. Для примера создадим файл test2.py в каталоге C:\book. Содержимое файла приведено в листинге 1.7.

Листинг 1.7. Получение данных из консоли

```
import sys
arr = sys.argv[:]
for n in arr:
    print(n)
```

Теперь запустим программу на выполнение из консоли, передав ей данные. Запустим консоль, для чего выберем в меню **Пуск** пункт **Выполнить**, в открывшемся окне наберем команду `cmd` и нажмем кнопку **OK**. Откроется черное окно с приглашением для ввода команд. Перейдем в каталог C:\book, набрав команду:

```
cd C:\book
```

В консоли должно появиться приглашение:

```
C:\book>
```

Для запуска нашей программы вводим команду:

```
test2.py uhud opk987
```

В этой команде мы передаем имя файла (`test2.py`) и некоторые данные (`-uNik` и `-p123`). Результат выполнения программы будет выглядеть так:

```
test2.py
uhud
opk987
```

1.8. Утилита *pip*: установка дополнительных библиотек

Интерпретатор Python поставляется с богатой стандартной библиотекой, реализующей, в частности, работу с файлами, шифрование, архивирование, обмен данными по сети и пр. Однако такие операции, как обработка графики, взаимодействие с базами данных SQLite, MySQL и многое другое, она не поддерживает, и для их выполнения нам придется устанавливать дополнительные библиотеки.

Для установки дополнительных библиотек в Python достаточно воспользоваться имеющейся в комплекте поставки Python утилитой pip, которая самостоятельно найдет запрошенную библиотеку в интернет-хранилище (*репозитории*) PyPI (Python Package Index, реестр пакетов Python), загрузит дистрибутивный пакет с редакцией библиотеки, совместимый с установленной версией Python, и установит ее. Если инсталлируемая библиотека требует для работы другие библиотеки, они также будут установлены.

Помимо этого, утилита pip позволяет просмотреть список уже установленных библиотек, получить сведения о выбранной библиотеке и удалить ненужную библиотеку.

Для использования утилиты pip в консоли следует набрать команду следующего формата:

```
pip <Команда и ее опции> <Универсальные опции>
```

Параметр `<Команда и ее опции>` указывает, что должна сделать утилита: установить библиотеку, вывести список библиотек, предоставить сведения об указанной библиотеке или удалить ее. Параметр `<Универсальные опции>` задают дополнительные настройки для самой утилиты и действуют на все поддерживаемые ею команды.

Далее приведен сокращенный список поддерживаемых утилитой pip команд вместе с их собственными опциями, а также наиболее востребованных универсальных опций. Полный список всех команд pip можно получить, воспользовавшись командой `help` и опцией `-h`.

Итак, утилита pip поддерживает следующие наиболее полезные нам команды:

- `install` — установка указанной библиотеки. Формат этой команды таков:

  ```
  pip install [<Опции>] <Название библиотеки>
  ```

  Если в параметре `<Опции>` не указана ни одна из них (см. далее), утилита просто загрузит и установит библиотеку с указанным названием. Если такая библиотека уже установлена, ничего не произойдет. Вот пример установки библиотеки Pillow:

  ```
  pip install pillow
  ```

  Доступные опции:

  - `-U` (или `--upgrade`) — обновление библиотеки с заданным названием до актуальной версии, имеющейся в репозитории PyPI. Обновляем библиотеку Pillow:

    ```
    pip install -U pillow
    ```

  - `--force-reinstall` — выполняет полную переустановку заданной библиотеки. Обычно используется вместе с опцией `-U`;
  - `--compile` — после установки библиотеки откомпилировать ее код. Позволяет ускорить запуск программ, использующих эту библиотеку.

  Если указано лишь название библиотеки, будет установлена наиболее актуальная версия этой библиотеки.

Если нужно установить определенную версию библиотеки, в качестве параметра <Название библиотеки> также можно использовать конструкцию такого формата (кавычки обязательны):

```
"<Название библиотеки><Оператор сравнения><Номер версии библиотеки>"
```

Номер версии устанавливаемой библиотеки задается в формате:

```
<Номер старшей версии>.<Номер младшей версии>[.<Номер модификации>]
```

Все номера указываются в виде целых чисел.

Пример установки библиотеки Pillow версии 8.4.0:

```
pip install "pillow==8.4.0"
```

Если номер модификации не указан, будет установлена версия с наиболее актуальной модификацией, имеющаяся в репозитории. Пример:

```
pip install "pillow==8.4"
```

Вместо любого из номеров, указываемых в составе версии, можно записать символ подстановки *. В таком случае будет установлена версия библиотеки, в которой соответствующий номер будет максимальным из присутствующих в репозитории. Пример установки библиотеки Pillow со старшей версией 7 и наиболее актуальной младшей:

```
pip install "pillow==7.*"
```

Поддерживаются следующие операторы сравнения:

- == — равно;
- < — меньше;
- > — больше;
- <= — не больше (меньше или равно);
- >= — не меньше (больше или равно);
- ~= — совместимо (будет описан позже).

Пример установки библиотеки Pillow версии 5.0.0 или более новой:

```
pip install "pillow>=5.0.0"
```

После названия библиотеки можно указать несколько конструкций формата:

```
<Оператор сравнения><Номер версии>
```

разделив их запятыми (после которых можно поставить пробелы). Каждая из этих конструкций задаст одно условие, которому должна удовлетворять устанавливаемая версия библиотеки. Будет установлена версия, удовлетворяющая *всем* указанным условиям. Пример установки библиотеки Pillow версии не ниже 8.4.0 и меньше 8.6.0:

```
pip install "pillow>=8.4.0, <8.6.0"
```

Что касается оператора ~= (совместимо), то при его использовании будет установлена либо указанная версия библиотеки, либо, при отсутствии таковой:

- если модификация указана — библиотека с заданными номерами старшей, младшей версий и указанной или, если таковая отсутствует, наиболее актуальной модификацией из имеющихся в репозитории;

- если модификация не указана — библиотека с заданным номером старшей версии и указанными или, если таковые отсутствуют, наиболее актуальными младшей версией и модификацией.

Примеры:

```
pip install "pillow~=8.4.0"
pip install "django~=3.1"
```

- `list` — вывод списка установленных библиотек и их версий. Формат команды:

```
pip list [<Опции>]
```

Пример:

```
pip list
```

У авторов было выведено:

```
Package    Version
---------- -------
Pillow     8.4.0
pip        21.2.4
setuptools 58.1.0
```

Единственная доступная здесь опция: `--format=<Формат вывода>`, задающая формат вывода. В качестве параметра `<Формат вывода>` можно указать `columns` (вывод в виде таблицы, как было показано в приведенном примере, — это формат по умолчанию), `freeze` (вывод в виде перечня) или `json` (вывод в формате JSON). Вот пример вывода списка установленных библиотек, оформленного в виде перечня:

```
pip list --format=freeze
```

У авторов было выведено:

```
Pillow==8.4.0
pip==21.2.4
setuptools==58.1.0
```

Вывод в формате JSON:

```
pip list --format=json
```

У авторов получилось:

```
[{"name": "Pillow", "version": "8.4.0"},
 {"name": "pip", "version": "21.2.4"},
 {"name": "setuptools", "version": "58.1.0"}]
```

> ***Примечание***
>
> Изначально в комплекте поставки Python уже присутствуют две библиотеки такого рода: pip, реализующая функциональность одноименной утилиты, и setuptools, предоставляющая специфические инструменты для установки дополнительных библиотек.

- `show` — вывод сведений об указанной библиотеке. Формат команды:

```
pip show [<Опции>] <Название библиотеки>
```

Выводятся название библиотеки, ее версия, краткое описание, интернет-адрес «домашнего» сайта, имя разработчика, его адрес электронной почты, название лицензии, по ко-

торой распространяется библиотека, путь, по которому она установлена, список библиотек, требующихся ей для работы, и список библиотек, которым она требуется для работы (если таковые есть). Для примера посмотрим сведения о Pillow:

```
pip show pillow
```

Вывод:

```
Name: Pillow
Version: 8.4.0
Summary: Python Imaging Library (Fork)
Home-page: https://python-pillow.org
Author: Alex Clark (PIL Fork Author)
Author-email: aclark@python-pillow.org
License: HPND
Location: c:\python310\lib\site-packages
Requires:
Required-by:
```

Единственная доступная опция: `-f` (или `--files`), которая указывает утилите pip дополнительно вывести список всех файлов, составляющих библиотеку. Вот пример вывода сведений о библиотеке Pillow, включая перечень составляющих ее файлов:

```
pip show -f pillow
```

- `uninstall` — удаление указанной библиотеки. Формат команды:

  ```
  pip uninstall [<Опции>] <Название библиотеки>
  ```

  Сначала будет выведен перечень каталогов, в которых хранятся файлы удаляемой библиотеки, и вопрос, действительно ли пользователь хочет удалить ее. Чтобы подтвердить удаление, нужно ввести букву `y`, чтобы отменить его — `n`, после чего в любом случае нажать клавишу <Enter>. Вот пример удаления библиотеки Pillow:

  ```
  pip uninstall pillow
  ```

  Из всех доступных опций для нас будет полезна только `-y` (или `--yes`), подавляющая вывод вопроса на удаление библиотеки. Пример:

  ```
  pip uninstall -y pillow
  ```

- `help` — вывод справочных сведений об утилите pip, поддерживаемых ею командах и опциях. Формат команды:

  ```
  pip help [<Команда>]
  ```

  - Если <Команда> не указана — будет выведен список всех поддерживаемых утилитой pip команд и универсальных опций:

    ```
    pip help
    ```

    Того же результата можно достичь, просто запустив в консоли утилиту pip безо всяких параметров.

  - Если <Команда> указана — будет выведена справочная информация об этой команде и всех ее опциях, а также перечень универсальных опций. В качестве примера выведем описание команды `install`:

    ```
    pip help install
    ```

Универсальные опции, поддерживаемые pip:

- `--proxy` — задает прокси-сервер, через который будет выполняться доступ к Интернету. Формат использования:

  ```
  --proxy=[<Имя пользователя>:<Пароль пользователя>@]<Интернет-адрес>:
  <Номер порта>
  ```

  Пример:

  ```
  pip install pillow --proxy=user123:pAsSwOrD@192.168.1.1:3128
  ```

- `--no-cache-dir` — отключает кеширование загруженных из репозитория библиотек на локальном диске. Работает только с командой `install`.

- `-v` (или `--verbose`) — вывод более подробных сведений о выполняемых утилитой pip действиях. Опция может быть указана один раз, дважды или трижды, тем самым задавая вывод все более и более подробных сведений:

  ```
  pip show pillow -v
  pip show pillow -v -v
  pip install pillow -v -v -v
  ```

  Дает эффект не со всеми командами pip.

- `-q` (или `--quiet`) — вывод менее подробных сведений о выполняемых утилитой pip действиях. Также может быть указана один раз, дважды или трижды, тем самым задавая вывод все менее и менее подробных сведений:

  ```
  pip uninstall pillow -q
  pip install pillow -q -q
  pip install pillow -q -q -q
  ```

  Дает эффект не со всеми командами pip.

- `-h` (или `--help`) — вывод справочных сведений о заданной команде pip, всех ее опциях и универсальных опциях pip (т. е. дает эффект, аналогичный отдаче описанной ранее команды `help` с указанием команды, для которой нужно вывести справку). Для примера выведем сведения о команде `uninstall`:

  ```
  pip uninstall -h
  ```

Все дополнительные библиотеки устанавливаются в каталог <Каталог установки Python>\Lib\site-packages.

В составе некоторых библиотек присутствуют программы, запускаемые из консоли. Такие программы устанавливаются в каталог <Каталог установки Python>\Scripts. Путь к этому каталогу при установке Python добавляется в список путей из системной переменной `PATH`, поэтому любую из записанных в этом каталоге программ можно запустить, просто набрав в консоли ее имя.

Если Python был установлен в каталог c:\Program Files (x86) или c:\Program Files, дополнительные библиотеки будут устанавливаться по пути c:\Users\<Имя учетной записи>\AppData\Roaming\Python\Python<Номер версии>\site-packages, а программы, входящие в состав библиотек, — по пути c:\Users\<Имя учетной записи>\AppData\Roaming\Python\Python<Номер версии>\Scripts. Путь к каталогу программ не добавляется в список путей из переменной `PATH`, и нам его придется добавить туда самостоятельно (или для запуска любой программы набирать полный путь к ней).

1.9. Доступ к документации

В составе Python поставляется документация по этому языку в формате CHM. Чтобы открыть ее, в меню **Пуск | Python 3.10** нужно выбрать пункт **Python 3.10 Manuals (32-bit)** или **Python 3.10 Manuals (64-bit)**.

Если в меню **Пуск | Python 3.10** выбрать пункт **Python 3.10 Module Docs (32-bit)** или **Python 3.10 Module Docs (64-bit)**, запустится сервер документов pydoc (рис. 1.7). Это написанный на самом Python веб-сервер, выводящий в веб-браузере документацию по модулям стандартной библиотеки Python.

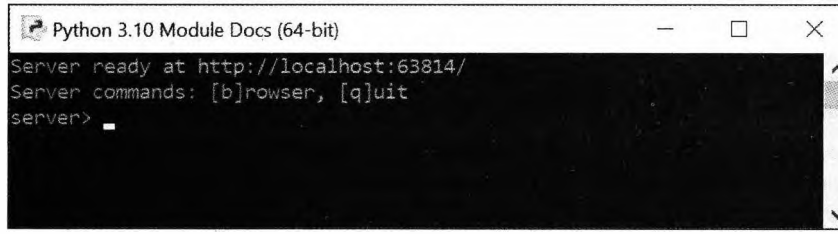


Рис. 1.7. Окно pydoc

Сразу после запуска pydoc откроется веб-браузер, в котором будет выведен список всех модулей стандартной библиотеки. Щелкнув на имени модуля, мы откроем страницу с описанием всех классов, функций и констант, объявленных в этом модуле.

Чтобы завершить работу pydoc, следует переключиться в его окно (см. рис. 1.7), ввести в нем команду `q` (от quit, выйти) и нажать клавишу <Enter> — окно при этом автоматически закроется. А введенная там команда `b` (от browser, браузер) повторно выведет в браузере страницу со списком модулей.

Документацию по различным языковым конструкциям Python можно получить, запустив интерпретатор в интерактивном режиме (см. *разд. 1.2*) и вызвав функцию `help()`. В качестве примера отобразим документацию по встроенной функции `input()`:

```
>>> help(input)
```

Результат выполнения:

```
Help on built-in function input in module builtins:

input(prompt=None, /)
  Read a string from standard input. The trailing newline is stripped.

  The prompt string, if given, is printed to standard output without a
  trailing newline before reading input.

  If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
  On *nix systems, readline is used if available.
```

С помощью функции `help()` можно получить документацию не только по конкретной функции, но и по всему модулю сразу. Для этого необходимо предварительно подключить нужный модуль. Например, подключим модуль `builtins`, содержащий определения всех встроенных функций и классов, а затем выведем документацию по нему:

```
>>> import builtins
>>> help(builtins)
```

В *разд. 1.4.1* говорилось, что для комментирования кода часто используются строки документирования. Такая строка сохраняется в атрибуте `__doc__`. Функция `help()` при составлении документации получает информацию из этого атрибута.

В качестве примера создадим файл **test3.py**, содержимое которого показано в листинге 1.8.

Листинг 1.8. Тестовый модуль test3.py

```
""" Это описание нашего модуля """
def func():
  """ Это описание функции"""
  pass
```

Теперь подключим этот модуль и выведем содержимое строк документирования, записанных в нем. Все эти действия выполняет код из листинга 1.9.

Листинг 1.9. Вывод строк документирования посредством функции help()

```
import test3            # Подключаем файл test3.py
help(test3)
input()
```

Запустим эту программу в окне **IDLE Shell** или щелчком мыши и получим результат:

```
Help on module test3:

NAME
    test3 - Это описание нашего модуля

FUNCTIONS
    func()
        Это описание функции

FILE
    d:\data\документы\работа\книги\python 3 и pyqt 6 разработка
    приложений\ftp\1\test3.py
```

Теперь получим содержимое строк документирования с помощью атрибута `__doc__`. Как это делается, показывает код из листинга 1.10.

Листинг 1.10. Вывод строк документирования посредством атрибута __doc__

```
import test3            # Подключаем файл test3.py
print(test3.__doc__)
print(test3.func.__doc__)
input()
```

Результат выполнения:

```
Это описание нашего модуля
Это описание функции
```

Атрибут __doc__ можно использовать вместо функции help(). В качестве примера получим документацию по функции input():

```
>>> print(input.__doc__)
```

Результат выполнения:

```
Read a string from standard input. The trailing newline is stripped.

The prompt string, if given, is printed to standard output without a
trailing newline before reading input.

If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
On *nix systems, readline is used if available.
```

Получить список всех идентификаторов внутри модуля позволяет функция dir(). Пример ее использования показывает код из листинга 1.11.

Листинг 1.11. Получение списка идентификаторов

```
import test3            # Подключаем файл test3.py
print(dir(test3))
input()
```

Результат выполнения:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'func']
```

Теперь получим список всех встроенных идентификаторов:

```
>>> import builtins
>>> print(dir(builtins))
```

Будучи вызванной без параметров, функция dir() возвращает список идентификаторов из текущего модуля:

```
>>> print(dir())
```

Результат выполнения:

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__']
```

1.10. Компиляция Python-файлов

При запуске любой Python-программы, сохраненной в файле, интерпретатор предварительно выполняет ее *компиляцию* — преобразование кода программы в особое компактное внутреннее представление. Откомпилированный код сохраняется в оперативной памяти и запускается на исполнение. Код, прошедший компиляцию, занимает меньше места и быстрее выполняется.

К сожалению, после завершения выполнения программы откомпилированный код удаляется из памяти. И при повторном запуске этой же программы ее компиляция выполняется заново, что замедляет запуск.

Однако можно выполнить компиляцию указанного модуля (и даже всех модулей, хранящихся в указанном каталоге) с сохранением откомпилированного кода в файле. При запуске такого файла хранящаяся в нем программа запустится очень быстро, поскольку интерпретатору не придется выполнять предварительную компиляцию кода.

Для компиляции Python-модулей следует набрать в консоли команду:

```
python -m compileall <Путь к файлу>|<Путь к каталогу> <Опции>
```

Можно указать либо путь к файлу модуля, который следует откомпилировать:

```
python -m compileall c:\work\python\program1.py
```

либо путь к каталогу — в этом случае будут откомпилированы все модули, хранящиеся как в каталоге с указанным путем, так и во вложенных в него каталогах:

```
python -m compileall c:\work\python\big_program
```

Откомпилированные модули сохранятся в каталоге __pycache__, который создается в каталоге с исходными Python-модулями, в файлах с именами формата:

<Имя исходного модуля>.cpython-<Номер версии Python>.pyc

Например, откомпилированный код модуля hello_world.py (см. листинг 1.1) будет сохранен в файле __pycache__\hello_world.cpython-310.pyc.

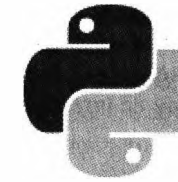
Расширение pyc также ассоциируется с исполняющей средой Python, поэтому файлы с таким расширением можно запускать простым щелчком мышью. А при распространении Python-программы можно передать конечным пользователям только откомпилированные файлы, содержащие ее код.

При повторной компиляции будут откомпилированы только те модули, чье содержимое изменилось после предыдущей компиляции. Python отслеживает изменение содержимого модулей по дате и времени последнего изменения файлов, в которых они хранятся.

Команда компиляции поддерживает следующие полезные опции:

- `-l` — компилировать только модули, непосредственно находящиеся в каталоге с указанным путем. Модули во вложенных каталогах компилироваться не будут;
- `-f` — принудительно перекомпилировать все модули в указанном каталоге, даже не изменившиеся после последней компиляции.

Следует отметить, что при подключении какого-либо модуля с целью использовать созданные в нем функции и объекты он *всегда* компилируется в сохранением результирующего кода в файл. Это делается для ускорения запуска программ, использующих этот же модуль.



ГЛАВА 2

Переменные и типы данных

2.1. Переменные

Переменная — это ячейка в оперативной памяти компьютера, предназначенная для хранения какого-либо значения и имеющая уникальное имя. Обращение к переменной с целью извлечь хранящееся в ней значение или занести в нее другое значение выполняется по имени этой переменной.

Чтобы создать переменную в языке Python, достаточно лишь занести в нее какое-либо значение. Впоследствии в эту же переменную можно заносить любые другие значения. Если занести в переменную другое значение, хранившееся в ней ранее будет потеряно.

Операция занесения значения в переменную называется *присваиванием*. Она выполняется посредством *оператора присваивания* = (знак равенства), слева от которого ставится имя нужной переменной, а справа — присваиваемое значение. Примеры:

```
>>> # Присваиваем переменной language_name строковое значение 'Python'
>>> language_name = 'Python'
>>> # Присваиваем переменной n число 123
>>> n = 123
```

Чтобы извлечь из переменной хранящееся в ней значение для использования его в вычислениях, следует просто указать имя этой переменной:

```
>>> # Выводим результат сложения числа из переменной n и 321
>>> print(n + 321)
    444
```

Имя переменной должно состоять из букв (можно использовать буквы любых алфавитов, но традиционно применяется лишь латиница), цифр и знаков подчеркивания, причем имя переменной не может начинаться с цифры. Кроме того, следует избегать указания символа подчеркивания в начале имени, поскольку идентификаторам с таким символом определено специальное назначение (подробности будут приведены в *главе 12*).

В качестве имен переменной нельзя использовать *ключевые слова* — слова, посредством которых в Python составляются всевозможные языковые конструкции. Получить список всех ключевых слов позволяет такой код:

```
>>> import keyword
>>> keyword.kwlist
    ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
     'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally',
```

```
'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Помимо ключевых слов, следует избегать совпадений со встроенными идентификаторами (именами функций, классов и модулей, встроенных в язык). Получить полный список встроенных идентификаторов позволяет следующий код:

```
>>> import builtins
>>> print(dir(builtins))
```

Правильные имена переменных: `x`, `y1`, `strName`, `str_name`.

Неправильные имена переменных: `1y`, `!important_value!`, `global`.

И последнее: Python принимает во внимание регистр букв, используемых в именах переменных. Например, `x` и `X` — разные переменные:

```
>>> x = 10; X = 20
>>> print(x, X)
    10 20
```

2.2. Типы данных. Понятие объекта и ссылки

Тип данных — это разновидность значения, обрабатываемого Python, и набор действий, которые могут быть выполнены с этим значением. Поддерживаются следующие типы данных:

- `int` — целое число. Размер числа ограничен лишь объемом оперативной памяти. Пример (функция `type()` позволяет определить тип значения):

  ```
  >>> type(2147483647), type(999999999999999999999999)
      (<class 'int'>, <class 'int'>)
  ```

- `float` — дробное, или *вещественное*, число:

  ```
  >>> type(5.1), type(8.5e-3)
      (<class 'float'>, <class 'float'>)
  ```

- `bool` — *логическая величина*. Поддерживаются значения `True` («истина») и `False` («ложь»). Пример:

  ```
  >>> type(True), type(False)
      (<class 'bool'>, <class 'bool'>)
  ```

- `str` — *строка* (последовательность произвольных символов):

  ```
  >>> type("Строка")
      <class 'str'>
  ```

- `list` — *список* (набор пронумерованных значений, называемых *элементами*, которые могут быть произвольными, списки можно изменять, добавляя, изменяя и удаляя находящиеся в них значения):

  ```
  >>> type( [1, 2, 3] )
      <class 'list'>
  ```

  Как говорилось ранее, списки можно изменять:

  ```
  >>> l = [1, 2, 3]        # Создаем список из трех чисел
  >>> print(l)
      [1, 2, 3]
  ```

```
>>> l[1] = 40           # Изменяем второй элемент
>>>                     # Нумерация элементов в списке начинается с 0
>>> print(l)
    [1, 40, 3]
```

- `tuple` — *кортеж* (аналогичен списку, но не может быть изменен):

```
>>> type( (1, 2, 3) )
    <class 'tuple'>
```

Попытка изменить кортеж вызовет программную ошибку:

```
>>> t = (1, 2, 3)       # Создаем кортеж из трех чисел
>>> t[2] = 60           # Пытаемся заменить третий элемент и получаем ошибку
    Traceback (most recent call last):
      File "<pyshell#30>", line 1, in <module>
        t[2] = 60
    TypeError: 'tuple' object does not support item assignment
```

- `range` — *диапазон* (последовательность целых чисел с заданными начальным, конечным значениями и разницей между отдельными числами):

```
>>> type( range(1, 10) )
    <class 'range'>
```

- `set` — *множество* (аналогично списку, но хранит лишь уникальные значения):

```
>>> type( {"a", "b", "c"} )
    <class 'set'>
```

- `frozenset` — неизменяемое множество:

```
>>> type(frozenset(["a", "b", "c"]))
    <class 'frozenset'>
```

- `bytes` — неизменяемая последовательность байтов:

```
>>> type(bytes("Строка", "utf-8"))
    <class 'bytes'>
```

- `bytearray` — изменяемая последовательность байтов:

```
>>> type(bytearray("Строка", "utf-8"))
    <class 'bytearray'>
```

Типы `list`, `tuple`, `range`, `set`, `frozenset`, `bytes` и `bytearray` носят общее название *последовательностей*, поскольку содержащиеся в них элементы можно перебрать последовательно, один за другим (например, в цикле), и выполнить над ними какие-либо действия. К последовательностям относятся и строки — отдельные символы в них также можно перебрать;

- `dict` — *словари* (наборы значений, каждое из которых связано с уникальным ключом, по которому это значение можно извлечь):

```
>>> type( {"x": 5, "y": 20} )
    <class 'dict'>
```

Тип `dict` называется *отображением*, поскольку в нем каждый ключ отображается на связанное с ним значение;

- `NoneType` — «пустой» тип. Поддерживается единственное значение `None`, обозначающий отсутствие любого «значащего» значения:

```
>>> type(None)
    <class 'NoneType'>
```

- `complex` — комплексные числа:

```
>>> type(2+2j)
    <class 'complex'>
```

- `function` — функции:

```
>>> def func(): pass
...
>>> type(func)
    <class 'function'>
```

- `module` — модули:

```
>>> import sys
>>> type(sys)
    <class 'module'>
```

- `type` — классы и типы данных:

```
>>> class C: pass
...
>>> type(C)
    <class 'type'>
>>> type(type(""))
    <class 'type'>
```

Значение каждого типа хранится в оперативной памяти в виде довольно сложной структуры, называемой *объектом*. Помимо собственно значения, в объекте содержатся всевозможные данные, описывающие это значение (например, у строкового значения также хранится количество символов, содержащихся в строке).

А теперь — важный момент! При присваивании значения, представленного объектом, какой-либо переменной в последнюю заносится не само это значение, а ссылка на него (*ссылку* можно рассматривать как особый компактный указатель на определенный объект). Поэтому, если, например, список присвоить одной переменной, а потом значение этой переменной присвоить другой переменной, обе переменные будут содержать ссылки на один и тот же список. Пример, иллюстрирующий сказанное:

```
>>> a = [1, 2, 3]           # Присваиваем список переменной a
>>> b = a                   # Присваиваем значение переменной a (список)
>>>                         # переменной b
>>> print(a, b)
    [1, 2, 3] [1, 2, 3]
>>> a[1] = 40               # Изменяем второй элемент списка из переменной a
>>>                         # Как видим, обе переменные ссылаются
>>> print(a, b)             # на один и тот же список
    [1, 40, 3] [1, 40, 3]
```

Типы делятся на изменяемые и неизменяемые. Значения *изменяемых* типов, к которым относятся `list`, `set`, `bytearray` и `dict`, можно изменять (так, можно изменить значение эле-

мента списка или байт из последовательности). Значения *неизменяемых* типов изменить нельзя (например, невозможно изменить значение элемента кортежа и символ в строке).

К неизменяемым типам относятся также числа и логические величины. То есть, если присвоить число переменной, а потом присвоить этой же переменной другое число, интерпретатор не станет заменять значение, хранящееся в существующем числовом объекте, а создаст новый числовой объект, который и занесет в переменную. Старый объект при этом будет удален из памяти.

2.3. Присваивание значений переменным

Присваивание — это занесение в переменную какого-либо значения (при этом значение, хранившееся в переменной ранее, теряется). Оно выполняется с помощью *оператора присваивания* = (знак равенства). Имя переменной, которой присваивается значение, ставится слева от оператора, а само присваиваемое значение — справа. Примеры:

```
>>> x = 7            # Присваивание целого числа (тип int)
>>> y = 7.8          # Присваивание вещественного числа (тип float)
>>> s1 = "Строка"    # Присваивание строки (тип string)
>>> s2 = 'Строка'    # Также присваивание строки
>>> b = True         # Присваивание логической величины True (тип bool)
```

В одной строке можно присвоить значение сразу нескольким переменным (*групповое присваивание*):

```
>>> x = y = 10       # Переменным x и y присвоено число 10
>>> print(x, y)
    10 10
```

Не забываем, что при этом все переменные получат ссылку на один и тот же объект. Поэтому групповое присваивание не рекомендуется использовать для значений изменяемых типов. Пример:

```
>>> x = y = [1, 2]          # Якобы создали два списка
>>> print(x, y)
    [1, 2] [1, 2]
>>> y[1] = 100              # Изменяем второй элемент списка
>>> print(x, y)
    [1, 100] [1, 100]
```

Если требуется сохранить в разных переменных разные списки с одинаковым составом значений, придется выполнить раздельное присваивание:

```
>>> x = [1, 2]
>>> y = [1, 2]
>>> y[1] = 100              # Изменяем второй элемент
>>> print(x, y)
    [1, 2] [1, 100]
```

Проверить, ссылаются ли две переменные на один и тот же объект, позволяет оператор `is`. Если переменные ссылаются на один и тот же объект, оператор `is` возвращает значение `True`, в противном случае — `False`. Пример:

```
>>> x = y = [1, 2]          # Один объект
>>> x is y
    True
```

```
>>> x = [1, 2]              # Разные объекты
>>> y = [1, 2]              # Разные объекты
>>> x is y
    False
```

Следует заметить, что в целях повышения производительности интерпретатор производит кеширование малых целых чисел и небольших строк. Это означает, что если ста переменным присвоено число 2, то, скорее всего, в этих переменных будет сохранена ссылка на один и тот же объект. Пример:

```
>>> x = 2; y = 2; z = 2
>>> x is y, y is z
    (True, True)
```

Посмотреть количество ссылок на указанный объект позволяет функция `getrefcount(<Объект>)` из модуля `sys`:

```
>>> import sys              # Подключаем модуль sys
>>> sys.getrefcount(2)
    304
```

Когда число ссылок на объект становится равно нулю, объект автоматически удаляется из оперативной памяти. Исключением являются объекты, которые подлежат кешированию.

Помимо группового, Python поддерживает *позиционное присваивание*. В этом случае переменные записываются через запятую слева от оператора =, а присваивание им значения — через запятую справа. Пример:

```
>>> x, y, z = 1, 2, 3
>>> print(x, y, z)
    1 2 3
```

С помощью позиционного присваивания можно поменять значения двух переменных местами:

```
>>> x, y = 1, 2
>>> x, y
    (1, 2)
>>> x, y = y, x
>>> x, y
    (2, 1)
```

По обе стороны оператора = могут быть указаны последовательности, к каковым относятся строки, списки, кортежи, диапазоны, типы `bytes` и `bytearray`:

```
>>> x, y, z = "123"              # Строка
>>> x, y, z
    ('1', '2', '3')
>>> x, y, z = [1, 2, 3]          # Список
>>> x, y, z
    (1, 2, 3)
>>> x, y, z = (1, 2, 3)          # Кортеж
>>> x, y, z
    (1, 2, 3)
```

```
>>> [x, y, z] = (1, 2, 3)          # Список слева, кортеж справа
>>> x, y, z
    (1, 2, 3)
```

Обратите внимание на то, что количество переменных и значений справа и слева от оператора = должно совпадать, иначе будет выведено сообщение об ошибке:

```
>>> x, y, z = (1, 2, 3, 4)
    Traceback (most recent call last):
      File "<pyshell#130>", line 1, in <module>
        x, y, z = (1, 2, 3, 4)
    ValueError: too many values to unpack (expected 3)
```

Python при несоответствии количества переменных и значений справа и слева от оператора = позволяет сохранить в одной из переменных список, состоящий из «лишних» значений. Для этого перед именем нужной переменной указывается звездочка (*). Пример:

```
>>> x, y, *z = (1, 2, 3, 4)
>>> x, y, z
    (1, 2, [3, 4])
>>> x, *y, z = (1, 2, 3, 4)
>>> x, y, z
    (1, [2, 3], 4)
>>> *x, y, z = (1, 2, 3, 4)
>>> x, y, z
    ([1, 2], 3, 4)
>>> x, y, *z = (1, 2, 3)
>>> x, y, z
    (1, 2, [3])
>>> x, y, *z = (1, 2)
>>> x, y, z
    (1, 2, [])
```

Как видно из примера, переменная, перед которой указана звездочка, всегда получает в качестве значения список. Если для этой переменной не хватило значений, то ей присваивается пустой список. Следует помнить, что звездочку можно указать только перед одной переменной, в противном случае возникнет неоднозначность и интерпретатор выведет сообщение об ошибке. Пример:

```
>>> *x, y, *z = (1, 2, 3, 4)
    SyntaxError: two starred expressions in assignment
```

2.4. Проверка типа данных

Выяснить тип данных, к которому относится заданное значение, позволяет функция `type(<Значение>)`. В качестве результата она возвращает ссылку на объект типа. Пример:

```
>>> type(a)
    <class 'int'>
```

Проверить, принадлежит ли какое-либо значение указанному типу данных, можно следующими способами:

- сравнить значение, возвращаемое функцией `type()`, с нужным типом данных:

```
>>> x = 10
>>> if type(x) == int:
        print("Это целое число (тип int)")
```

- использовать функцию `isinstance(<Значение>, <Тип>)`:

```
>>> s = "Строка"
>>> if isinstance(s, str):
        print("Это строка (тип str)")
```

2.5. Преобразование типов данных

Над значением, относящимся к определенному типу, можно производить лишь операции, допустимые для этого типа. Например, можно складывать друг с другом числа, но строку сложить с числом нельзя — это приведет к выводу сообщения об ошибке:

```
>>> 2 + "25"
    Traceback (most recent call last):
      File "<pyshell#0>", line 1, in <module>
        2 + "25"
    TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Для преобразования значения из одного типа данных в другой предназначены следующие функции:

- `bool([<Значение>])` — преобразует заданное значение в логический тип данных:

```
>>> bool(0), bool(1), bool(""), bool("Строка"), bool([1, 2]), bool([])
    (False, True, False, True, True, False)
```

- `int([<Значение>[, <Система счисления>]])` — преобразует заданное значение в число. Во втором параметре можно указать систему счисления (по умолчанию 10). Примеры:

```
>>> int(7.5), int("71")
    (7, 71)
>>> int("71", 10), int("71", 8), int("0o71", 8), int("A", 16)
    (71, 57, 57, 10)
```

Если преобразование невозможно, то генерируется исключение:

```
>>> int("71s")
    Traceback (most recent call last):
      File "<pyshell#9>", line 1, in <module>
        int("71s")
    ValueError: invalid literal for int() with base 10: '71s'
```

- `float([<Число или строка>])` — преобразует указанное целое число или строку в вещественное число:

```
>>> float(7), float("7.1")
    (7.0, 7.1)
>>> float("Infinity"), float("-inf")
    (inf, -inf)
>>> float("Infinity") + float("-inf")
    nan
```

- `str([<Значение>])` — преобразует указанное значение в строку:

```
>>> str(125), str([1, 2, 3])
    ('125', '[1, 2, 3]')
>>> str((1, 2, 3)), str({"x": 5, "y": 10})
    ('(1, 2, 3)', "{'y': 10, 'x': 5}")
>>> str(bytes("строка", "utf-8"))
    "b'\\xd1\\x81\\xd1\\x82\\xd1\\x80\\xd0\\xbe\\xd0\\xba\\xd0\\xb0'"
>>> str(bytearray("строка", "utf-8"))
    "bytearray(b'\\xd1\\x81\\xd1\\x82\\xd1\\x80\\xd0\\xbe\\xd0↵
    \\xba\\xd0\\xb0')"
```

- `str(<Значение>[, <Кодировка>[, <Обработка ошибок>]])` — преобразует заданное значение типа `bytes` или `bytearray` в строку в указанной кодировке. В третьем параметре можно указать значение `"strict"` (при ошибке возбуждается исключение `UnicodeDecodeError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется символом, имеющим код `\uFFFD`) или `"ignore"` (неизвестные символы игнорируются). Примеры:

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
    ('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
    Traceback (most recent call last):
      File "<pyshell#16>", line 1, in <module>
        str(obj1, "ascii", "strict")
    UnicodeDecodeError: 'ascii' codec can't decode byte ↵
    0xd1 in position 0: ordinal not in range(128)
>>> str(obj1, "ascii", "ignore")
    '1'
```

- `bytes(<Строка>, <Кодировка>[, <Обработка ошибок>])` — преобразует указанную строку в объект типа `bytes` в заданной кодировке. В третьем параметре могут быть заданы значения `"strict"` (по умолчанию), `"replace"` или `"ignore"`. Примеры:

```
>>> bytes("строка", "cp1251")
    b'\xf1\xf2\xf0\xee\xea\xe0'
>>> bytes("строка123", "ascii", "ignore")
    b'123'
```

- `bytes(<Последовательность>)` — преобразует указанную последовательность целых чисел от 0 до 255 в объект типа `bytes`. Если какое-либо число из последовательности не попадает в диапазон, возбуждается исключение `ValueError`. Пример:

```
>>> b = bytes([225, 226, 224, 174, 170, 160])
>>> b
    b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
    'строка'
```

- `bytearray(<Строка>, <Кодировка>[, <Обработка ошибок>])` — преобразует заданную строку в объект типа `bytearray` в указанной кодировке. В третьем параметре могут быть указаны значения `"strict"` (по умолчанию), `"replace"` или `"ignore"`. Пример:

```
>>> bytearray("строка", "cp1251")
    bytearray(b'\xf1\xf2\xf0\xee\xea\xe0')
```

- `bytearray(<Последовательность>)` — преобразует заданную последовательность целых чисел от 0 до 255 в объект типа `bytearray`. Если какое-либо число из последовательности не попадает в диапазон, возбуждается исключение `ValueError`. Пример:

```
>>> b = bytearray([225, 226, 224, 174, 170, 160])
>>> b
    bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
    'строка'
```

- `list(<Последовательность>)` — преобразует элементы заданной последовательности в список:

```
>>> list("12345")              # Преобразование строки
    ['1', '2', '3', '4', '5']
>>> list((1, 2, 3, 4, 5))      # Преобразование кортежа
    [1, 2, 3, 4, 5]
```

- `tuple(<Последовательность>)` — преобразует элементы заданной последовательности в кортеж:

```
>>> tuple("123456")            # Преобразование строки
    ('1', '2', '3', '4', '5', '6')
>>> tuple([1, 2, 3, 4, 5])     # Преобразование списка
    (1, 2, 3, 4, 5)
```

В качестве примера рассмотрим возможность сложения двух чисел, введенных пользователем (листинг 2.1).

Листинг 2.1. Получение данных от пользователя

```
x = input("x = ")               # Вводим 5
y = input("y = ")               # Вводим 12
print(x + y)
input()
```

Результатом выполнения этой программы будет не число, а строка `"512"`. Как видим, функция `input()` возвращает результат в виде строки. Чтобы просуммировать два числа, необходимо преобразовать полученные строки в числа (листинг 2.2).

Листинг 2.2. Преобразование строки в число

```
x = int(input("x = "))          # Вводим 5
y = int(input("y = "))          # Вводим 12
print(x + y)
input()
```

В этом случае мы получим число 17, как и должно быть. Однако если пользователь вместо числа введет строку, то программа завершится с фатальной ошибкой. Как обработать ошибку, мы разберемся по мере изучения языка.

2.6. Удаление переменных

Удалить переменную можно с помощью инструкции следующего формата:

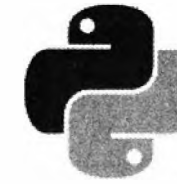
```
del <Переменная 1>[, . . ., <Переменная N>]
```

Пример удаления одной переменной:

```
>>> x = 10
>>> x
    10
>>> del x
>>> x
    Traceback (most recent call last):
      File "<pyshell#1>", line 1, in <module>
        del x; x
    NameError: name 'x' is not defined
```

Пример удаления нескольких переменных:

```
>>> x, y = 10, 20
>>> del x, y
```



ГЛАВА 3

Операторы

Оператор — языковая конструкция, выполняющая над переданными ему значениями (*операндами*) какое-либо элементарное действие (например, математическое сложение). Набор операторов, поддерживаемых Python, определен разработчиками этого языка и не может быть расширен (в отличие от набора поддерживаемых функций, определение которых описывается в *главе 11*).

3.1. Математические операторы

Производить операции над числами позволяют математические операторы:

- `+` — сложение:

```
>>> 10 + 5              # Целые числа
    15
>>> 12.4 + 5.2          # Вещественные числа
    17.6
>>> 10 + 12.4           # Целые и вещественные числа
    22.4
```

- `-` — вычитание:

```
>>> 10 - 5              # Целые числа
    5
>>> 12.4 - 5.2          # Вещественные числа
    7.2
>>> 12 - 5.2            # Целые и вещественные числа
    6.8
```

- `*` — умножение:

```
>>> 10 * 5              # Целые числа
    50
>>> 12.4 * 5.2          # Вещественные числа
    64.48
>>> 10 * 5.2            # Целые и вещественные числа
    52.0
```

- `/` — деление. Результатом деления всегда является вещественное число, даже если производится деление целых чисел. Примеры:

```
>>> 10 / 5              # Деление целых чисел без остатка
    2.0
>>> 10 / 3              # Деление целых чисел с остатком
    3.3333333333333335
>>> 10.0 / 5.0          # Деление вещественных чисел
    2.0
>>> 10.0 / 3.0          # Деление вещественных чисел
    3.3333333333333335
>>> 10 / 5.0            # Деление целого числа на вещественное
    2.0
>>> 10.0 / 5            # Деление вещественного числа на целое
    2.0
```

- // — деление с округлением вниз. Остаток всегда отбрасывается. Примеры:

```
>>> 10 // 5          # Деление целых чисел без остатка
    2
>>> 10 // 3          # Деление целых чисел с остатком
    3
>>> 10.0 // 5.0      # Деление вещественных чисел
    2.0
>>> 10.0 // 3.0      # Деление вещественных чисел
    3.0
>>> 10 // 5.0        # Деление целого числа на вещественное
    2.0
>>> 10 // 3.0        # Деление целого числа на вещественное
    3.0
>>> 10.0 // 5        # Деление вещественного числа на целое
    2.0
>>> 10.0 // 3        # Деление вещественного числа на целое
    3.0
```

- % — остаток от деления:

```
>>> 10 % 5           # Деление целых чисел без остатка
    0
>>> 10 % 3           # Деление целых чисел с остатком
    1
>>> 10.0 % 5.0       # Операция над вещественными числами
    0.0
>>> 10.0 % 3.0       # Операция над вещественными числами
    1.0
>>> 10 % 5.0  # Операция над целыми и вещественными числами
    0.0
>>> 10 % 3.0  # Операция над целыми и вещественными числами
    1.0
>>> 10.0 % 5  # Операция над целыми и вещественными числами
    0.0
>>> 10.0 % 3  # Операция над целыми и вещественными числами
    1.0
```

- ** — возведение в степень. Основание указывается первым операндом, показатель — вторым. Примеры:

```
>>> 10 ** 2, 10.0 ** 2
    (100, 100.0)
```

- унарный минус (-) — изменяет знак числа на противоположный:

```
>>> -10, -10.0, -(-10), -(-10.0)
    (-10, -10.0, 10, 10.0)
```

- унарный плюс (+) — ничего не делает с числом[1]:

```
>>> +10, +10.0)
    (10, 10.0)
```

Как видно из приведенных примеров, операции над числами разных типов возвращают число, имеющее более сложный из типов, участвующих в операции. Целые числа имеют самый простой тип, далее идут вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое число и вещественное, то целое число будет автоматически преобразовано в вещественное число, затем будет произведена операция над вещественными числами, а результатом станет вещественное число.

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей операции может показаться странным:

```
>>> 0.3 - 0.1 - 0.1 - 0.1
    -2.7755575615628914e-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другой результат. Если необходимо производить операции с фиксированной точностью, следует использовать модуль `decimal`. Пример:

```
>>> from decimal import Decimal
>>> Decimal("0.3") - Decimal("0.1") - Decimal("0.1") - Decimal("0.1")
    Decimal('0.0')
```

3.2. Двоичные операторы

Двоичные операторы предназначены для манипуляции отдельными битами:

- `~` — двоичная инверсия. Значение каждого бита заменяется на противоположное:

```
>>> x = 100                 # 01100100
>>> x = ~x                  # 10011011
```

- `&` — двоичное И:

```
>>> x = 100                 # 01100100
>>> y = 75                  # 01001011
>>> z = x & y               # 01000000
```

- `|` — двоичное ИЛИ:

```
>>> x = 100                 # 01100100
>>> y = 75                  # 01001011
>>> z = x | y               # 01101111
```

[1] Не совсем понятно, зачем нужен этот оператор…

- ^ — двоичное исключающее ИЛИ:

```
>>> x = 100                    # 01100100
>>> y = 250                    # 11111010
>>> z = x ^ y                  # 10011110
```

- << — сдвиг влево — сдвигает двоичное представление числа, заданного первым операндом, влево на количество разрядов, указанное вторым операндом, и заполняет разряды справа нулями:

```
>>> x = 100                    # 01100100
>>> y = x << 1                 # 11001000
>>> z = y << 1                 # 10010000
>>> k = z << 2                 # 01000000
```

- >> — сдвиг вправо — сдвигает двоичное представление числа, заданного первым операндом, вправо на количество разрядов, указанное вторым операндом, и заполняет разряды слева нулями, если число положительное, и единицами, если число отрицательное:

```
>>> x = 100                    # 01100100
>>> y = x >> 1                 # 00110010
>>> z = y >> 1                 # 00011001
>>> k = z >> 2                 # 00000110
>>> x = -127                   # 10000001
>>> y = x >> 1                 # 11000000
>>> z = y >> 2                 # 11110000
>>> k = z << 1                 # 11100000
>>> m = k >> 1                 # 11110000
```

3.3. Операторы для работы с последовательностями

Для работы с последовательностями предназначены следующие операторы:

- + — *конкатенация* (объединение):

```
>>> print("Строка1" + "Строка2")    # Конкатенация строк
    Строка1Строка2
>>> [1, 2, 3] + [4, 5, 6]          # Списки
    [1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)          # Кортежи
    (1, 2, 3, 4, 5, 6)
```

- * — повторение. Повторяет последовательность из первого операнда количество раз, указанное вторым операндом. Примеры:

```
>>> "s" * 20                       # Строки
    'ssssssssssssssssssss'
>>> [1, 2] * 3                     # Списки
    [1, 2, 1, 2, 1, 2]
>>> (1, 2) * 3                     # Кортежи
    (1, 2, 1, 2, 1, 2)
```

- `in` — проверка на вхождение. Если значение, заданное первым операндом, входит в последовательность, указанную вторым операндом, то возвращается логическое значение `True`, в противном случае — `False`. Примеры:

```
>>> "Строка" in "Строка для поиска"    # Строки
    True
>>> "Буква" in "Строка для поиска"     # Строки
    False
>>> 2 in [1, 2, 3], 4 in [1, 2, 3]    # Списки
    (True, False)
>>> 2 in (1, 2, 3), 6 in (1, 2, 3)    # Кортежи
    (True, False)
```

- `not in` — проверка на невхождение. Если значение, заданное первым операндом, *не* входит в последовательность, указанную вторым операндом, то возвращается значение `True`, в противном случае — `False`. Примеры:

```
>>> "Строка" not in "Строка для поиска"       # Строки
    False
>>> "Буква" not in "Строка для поиска"        # Строки
    True
>>> 2 not in [1, 2, 3], 4 not in [1, 2, 3]    # Списки
    (False, True)
>>> 2 not in (1, 2, 3), 6 not in (1, 2, 3)    # Кортежи
    (False, True)
```

3.4. Операторы присваивания

Операторы присваивания предназначены для сохранения значения в переменной:

- `=` — простое присваивание:

```
>>> x = 5
>>> x
    5
```

- `+=` — присваивание со сложением — увеличивает значение переменной на величину из второго операнда:

```
>>> x = 5; x += 10            # Эквивалентно x = x + 10
>>> x
    15
```

Будучи примененным к последовательностям, оператор `+=` производит конкатенацию:

```
>>> s = "Python"; s += " 3.10"
>>> print(s)
    Python 3.10
```

- `-=` — присваивание с вычитанием — уменьшает значение переменной на величину из второго операнда:

```
>>> x = 10; x -= 5            # Эквивалентно x = x - 5
>>> x
    5
```

- `*=` — присваивание с умножением — умножает значение переменной на величину из второго операнда:

```
>>> x = 10; x *= 5            # Эквивалентно x = x * 5
>>> x
    50
```

Будучи примененным к последовательности, оператор `*=` производит повторение:

```
>>> s = "*"; s *= 20
>>> s
    '********************'
```

- `/=` — присваивание с делением — делит значение переменной на величину из второго операнда:

```
>>> x = 10; x /= 3            # Эквивалентно x = x / 3
>>> x
    3.3333333333333335
>>> y = 10.0; y /= 3.0        # Эквивалентно y = y / 3.0
>>> y
    3.3333333333333335
```

- `//=` — присваивание с делением и округлением вниз — делит значение переменной на величину из второго операнда и округляет результат вниз:

```
>>> x = 10; x //= 3           # Эквивалентно x = x // 3
>>> x
    3
>>> y = 10.0; y //= 3.0       # Эквивалентно y = y // 3.0
>>> y
    3.0
```

- `%=` — присваивание с делением по модулю — делит по модулю значение переменной на величину из второго операнда:

```
>>> x = 10; x %= 2            # Эквивалентно x = x % 2
>>> x
    0
>>> y = 10; y %= 3            # Эквивалентно y = y % 3
>>> y
    1
```

- `**=` — присваивание с возведением в степень — возводит значение переменной в степень, заданную вторым операндом:

```
>>> x = 10; x **= 2           # Эквивалентно x = x ** 2
>>> x
    100
```

- `:=` (начиная с Python 3.8) — присваивание в составе сложного выражения (рассмотрен в *главе 4*).

3.5. Пустой оператор

Пустой оператор `pass` ничего не делает:

```
>>> pass                          # Ничего не делаем
>>> pass                          # Снова ничего не делаем
>>> print("Что-то сделали")
    Что-то сделали
```

Обычно пустой оператор используется, чтобы временно определить «пустую», ничего не делающую функцию или «пустой» класс (определения функций и классов описываются в *главах 11* и *13*):

```
def empty_function(): pass
. . .
class empty_class: pass
```

Впоследствии такие «пустые» функции и классы наполняются необходимым содержимым.

3.6. Приоритет операторов

Приоритет операторов — это очередность, в которой выполняются различные операторы Python. Например, оператор умножения будет выполнен раньше оператора сложения, поскольку приоритет первого выше, чем у второго.

Рассмотрим следующее выражение:

```
x = 5 + 10 * 3 / 2
```

Последовательность его вычисления будет такой:

1. Число 10 будет умножено на 3, поскольку приоритет оператора умножения выше приоритета оператора сложения.
2. Полученное значение будет поделено на 2, поскольку приоритет оператора деления равен приоритету оператора умножения (а операторы с равными приоритетами выполняются слева направо), но выше, чем у оператора сложения.
3. К полученному значению будет прибавлено число 5.
4. Значение будет присвоено переменной `x`, поскольку оператор присваивания `=` имеет наименьший приоритет.

```
>>> x = 5 + 10 * 3 / 2
>>> x
    20.0
```

С помощью скобок можно изменить последовательность вычисления выражения

```
x = (5 + 10) * 3 / 2
```

Теперь порядок вычислений станет иным:

1. К числу 5 будет прибавлено 10.
2. Полученное значение будет умножено на 3.
3. Полученное значение будет поделено на 2.
4. Значение будет присвоено переменной `x`.

```
>>> x = (5 + 10) * 3 / 2
>>> x
    22.5
```

Перечислим операторы в порядке убывания приоритета:

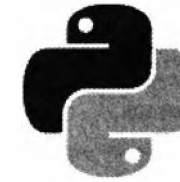
1. `-x`, `+x`, `~x`, `**` — унарный минус, унарный плюс, двоичная инверсия, возведение в степень. Если унарные операторы расположены слева от оператора `**`, то возведение в степень имеет больший приоритет, а если справа — то меньший. Например, выражение

   ```
   -10 ** -2
   ```

   эквивалентно следующей расстановке скобок:

   ```
   -(10 ** (-2))
   ```

2. `*`, `%`, `/`, `//` — умножение (повторение), остаток от деления, деление, деление с округлением вниз.
3. `+`, `–` — сложение (конкатенация), вычитание.
4. `<<`, `>>` — двоичные сдвиги.
5. `&` — двоичное И.
6. `^` — двоичное исключающее ИЛИ.
7. `|` — двоичное ИЛИ.
8. `=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=` — присваивание.



ГЛАВА 4

Инструкции ветвления, выбора и циклы

Инструкция ветвления позволяет при выполнении какого-либо условия исполнить один блок кода, а при невыполнении условия — другой. *Инструкция выбора*, если указанная переменная хранит одно значение, исполняет первый блок кода, если переменная хранит второе значение — второй блок кода, если третье — третий блок кода и т. д. Два вида *циклов* выполняют указанный блок кода либо столько раз, сколько элементов содержится в заданной последовательности, при этом перебирая ее элементы, либо пока выполняется заданное условие.

В качестве *условий* в подобного рода инструкциях применяются выражения, в качестве результата выдающие логическую величину `True` («истина», условие выполняется, или истинно) или `False` («ложь», условие не выполняется, или ложно).

Также в качестве условия можно использовать выражение, выдающее результат любого другого типа, который в этом случае будет автоматически приведен к логическому типу:

- в значение `True` будут преобразованы:
  - любое число, не равное нулю:

```
>>> bool(1), bool(20), bool(-20)
    (True, True, True)
>>> bool(1.0), bool(0.1), bool(-20.0)
    (True, True, True)
```

  - не пустой объект:

```
>>> bool("0"), bool([0, None]), bool((None,)), bool({"x": 5})
    (True, True, True, True)
```

- в значение `False` будут преобразованы:
  - число, равное нулю:

```
>>> bool(0), bool(0.0)
    (False, False)
```

  - пустой объект:

```
>>> bool(""), bool([]), bool(())
    (False, False, False)
```

  - значение `None`:

```
>>> bool(None)
    False
```

4.1. Операторы сравнения

Операторы сравнения применяются в условиях, чтобы сравнить значения указанных операндов:

- `==` — *равно* — значения обоих операндов должны быть равны друг другу:

```
>>> 1 == 1, 1 == 5
    (True, False)
```

- `!=` — *не равно* — значения обоих операндов должны быть не равны друг другу:

```
>>> 1 != 5, 1 != 1
    (True, False)
```

- `<` — *меньше* — первый операнд должен быть меньше второго:

```
>>> 1 < 5, 1 < 0
    (True, False)
```

- `>` — *больше* — первый операнд должен быть больше второго:

```
>>> 1 > 0, 1 > 5
    (True, False)
```

- `<=` — *меньше или равно* — первый операнд должен быть меньше или равен второму:

```
>>> 1 <= 5, 1 <= 0, 1 <= 1
    (True, False, True)
```

- `>=` — *больше или равно* — первый операнд должен быть больше или равен второму:

```
>>> 1 >= 0, 1 >= 5, 1 >= 1
    (True, False, True)
```

- `in` — *вхождение* — первый операнд должен входить в последовательность из второго операнда:

```
>>> "Строка" in "Строка для поиска" # Строки
    True
>>> 2 in [1, 2, 3], 4 in [1, 2, 3]  # Списки
    (True, False)
>>> 2 in (1, 2, 3), 4 in (1, 2, 3)  # Кортежи
    (True, False)
```

Оператор `in` также можно использовать для проверки существования ключа в словаре:

```
>>> "x" in {"x": 1, "y": 2}, "z" in {"x": 1, "y": 2}
    (True, False)
```

- `not in` — *невхождение* — первый операнд *не* должен входить в последовательность из второго операнда:

```
>>> "Строка" not in "Строка для поиска" # Строки
    False
>>> 2 not in [1, 2, 3], 4 not in [1, 2, 3]  # Списки
    (False, True)
>>> 2 not in (1, 2, 3), 4 not in (1, 2, 3)  # Кортежи
    (False, True)
```

- `is` — переменные, заданные операндами, должны ссылаться на один и тот же объект:

```
>>> x = y = [1, 2]
>>> x is y
    True
>>> x = [1, 2]; y = [1, 2]
>>> x is y
    False
```

- `is not` — переменные, заданные операндами, должны ссылаться на разные объекты:

```
>>> x = y = [1, 2]
>>> x is not y
    False
>>> x = [1, 2]; y = [1, 2]
>>> x is not y
    True
```

Значение логического выражения можно инвертировать (изменить на противоположное) с помощью оператора логического НЕ — `not`:

```
>>> x = 1; y = 1
>>> x == y
    True
>>> not (x == y), not x == y
    (False, False)
```

Круглые скобки здесь можно не указывать, поскольку оператор `not` имеет более низкий приоритет, чем операторы сравнения.

В логическом выражении можно указывать сразу несколько условий:

```
>>> x = 10
>>> 1 < x < 20, 11 < x < 20
    (True, False)
```

Несколько логических выражений можно объединить в одно большое с помощью следующих операторов:

- `and` — *логическое И* — если оба операнда равны `True`, возвращается `True`, в противном случае — `False`:

```
>>> 1 < 5 and 2 < 5              # True and True == True
    True
>>> 1 < 5 and 2 > 5              # True and False == False
    False
>>> 1 > 5 and 2 < 5              # False and True == False
    False
```

  Если значение первого операнда не относится к логическому типу, интерпретатор преобразует его в логический тип. Если после преобразования получается `False`, то возвращается значение первого операнда, в противном случае — значение второго операнда. Пример:

```
>>> 10 and 20, 0 and 20, 10 and 0
    (20, 0, 0)
```

- `or` — *логическое ИЛИ* — если хотя бы один операнд равен `True`, возвращается `True`, в противном случае — `False`:

```
>>> 1 < 5 or 2 < 5              # True or True == True
    True
>>> 1 < 5 or 2 > 5              # True or False == True
    True
>>> 1 > 5 or 2 < 5              # False or True == True
    True
>>> 1 > 5 or 2 > 5              # False or False == False
    False
```

Если значение первого операнда не относится к логическому типу, интерпретатор преобразует его в логический тип. Если после преобразования получается `False`, то возвращается значение второго операнда, в противном случае — значение первого операнда. Пример:

```
>>> 10 or 20, 0 or 20, 10 or 0
    (10, 20, 10)
>>> 0 or "" or None or [] or "s"
    's'
```

Перечислим операторы сравнения в порядке убывания приоритета:

1. `<`, `>`, `<=`, `>=`, `==`, `!=`, `<>`, `is`, `is not`, `in`, `not in`.
2. `not` — логическое отрицание.
3. `and` — логическое И.
4. `or` — логическое ИЛИ.

4.2. Инструкция ветвления

Инструкция ветвления при выполнении одного заданного условия исполняет один блок кода, при выполнении другого условия — другой, при выполнении третьего — третий, а при невыполнении всех условий — четвертый. Оно записывается в следующем формате:

```
if <Условие 1>:
    <Блок, исполняемый, если выполняется условие 1>
[elif <Условие 2>:
    <Блок, исполняемый, если выполняется условие 2>
elif <Условие 3>:
    <Блок, исполняемый, если выполняется условие 3>
. . .
elif <Условие n>:
    <Блок, исполняемый, если выполняется условие n>
]
[else:
    <Блок, исполняемый, если ни одно условие не выполняется>
]
```

Условий и соответствующих им блоков может быть произвольное количество.

Выражения, входящие в блок, должны выделяться отступом слева, содержащим одинаковое количество пробелов (обычно четыре). Концом блока является инструкция, перед которой расположено меньшее количество пробелов.

Напишем программу, которая проверяет, является ли введенное пользователем число четным, и выводит соответствующее сообщение (листинг 4.1).

Листинг 4.1. Проверка числа на четность

```
x = int(input("Введите число: "))
if x % 2 == 0:
    print(x, " - четное число")
else:
    print(x, " - нечетное число")
input()
```

Если блок состоит из одной инструкции, эту инструкцию можно разместить на одной строке с языковой конструкцией `if`, `elif` или `else`:

```
x = int(input("Введите число: "))
if x % 2 == 0: print(x, " - четное число")
else: print(x, " - нечетное число")
```

В таком случае концом блока является конец строки. Это означает, что инструкции, входящие в блок, можно разместить на одной строке, разделяя их точкой с запятой. Пример:

```
x = int(input("Введите число: "))
if x % 2 == 0: print(x, end=" "); print("- четное число")
else: print(x, end=" "); print("- нечетное число")
```

> ***Совет***
>
> Код, подобный приведенному в этом примере, плохо читается. Поэтому всегда размещайте все инструкции на отдельных строках.

Как говорилось ранее, выражение ветвления может содержать произвольное количество условий и соответствующих им блоков. Рассмотрим это на примере (листинг 4.2).

Листинг 4.2. Проверка нескольких условий

```
print("""Какой операционной системой вы пользуетесь?
1 — Windows 11
2 — Windows 10
3 — Windows 8.1
4 — Windows 8
5 — Windows 7
6 — Другая""")
os = input("Введите число, соответствующее ответу: ")
if   os == "1":
    print("Вы выбрали: Windows 11")
elif os == "2":
    print("Вы выбрали: Windows 10")
elif os == "3":
    print("Вы выбрали: Windows 8.1")
```

```
elif os == "4":
    print("Вы выбрали: Windows 8")
elif os == "5":
    print("Вы выбрали: Windows 7")
elif os == "6":
    print("Вы выбрали: другая")
elif not os:
    print("Вы не ввели число")
else:
    print("Мы не смогли определить вашу операционную систему")
input()
```

Обратите внимание, что условие в последней конструкции `elif` не содержит операторов сравнения:

```
elif not os:
```

Такая запись эквивалентна следующей:

```
elif os == "":
```

Пустая строка всегда преобразуется в `False`, и, чтобы условие в этом случае считалось выполненным, нам следует инвертировать полученный результат оператором `not`.

Одну инструкцию ветвления можно вложить в другую. В этом случае отступ у блоков вложенной инструкции должен быть в два раза больше (листинг 4.3).

Листинг 4.3. Вложенные инструкции

```
print("""Какой операционной системой вы пользуетесь?
1 — Windows 11
2 — Windows 10
3 — Windows 8.1
4 — Windows 8
5 — Windows 7
6 — Другая""")
os = input("Введите число, соответствующее ответу: ")
if os != "":
    if   os == "1":
        print("Вы выбрали: Windows 11")
    elif os == "2":
        print("Вы выбрали: Windows 10")
    elif os == "3":
        print("Вы выбрали: Windows 8.1")
    elif os == "4":
        print("Вы выбрали: Windows 8")
    elif os == "5":
        print("Вы выбрали: Windows 7")
    elif os == "6":
        print("Вы выбрали: другая")
    else:
        print("Мы не смогли определить вашу операционную систему")
```

```
else:
    print("Вы не ввели число")
input()
```

Инструкция ветвления может быть записана в альтернативном формате:

```
<Если истина> if <Условие> else <Если ложь>
```

Если заданное условие выполняется, инструкция возвращает значение `<Если истина>`, в противном случае — значение `<Если ложь>`. Возвращенное значение можно вывести в консоль, присвоить какой-либо переменной или использовать в последующих вычислениях. Пример:

```
>>> print("Да" if 10 % 2 == 0 else "Нет")
    Да
>>> s = "Да" if 10 % 2 == 0 else "Нет"
>>> s
    'Да'
>>> s = "Да" if 11 % 2 == 0 else "Нет"
>>> s
    'Нет'
```

4.3. Инструкция выбора

Инструкция выбора, если проверяемое значение совпадает с образцом 1, исполняет блок 1, если совпадает с образцом 2 — блок 2 и т. д. Если же проверяемое значение не совпадает ни с одним из указанных образцов, будет выполнен `<Блок _>` (если он присутствует, если же этот блок отсутствует, ничего не произойдет). Поддержка этой инструкции появилась в Python 3.10.

Формат записи инструкции выбора:

```
match <Проверяемое значение>:
    case <Образец 1>:
        <Блок 1>
    case <Образец 2>:
        <Блок 2>
    . . .
    case <Образец n>:
        <Блок n>
    [case _:
        <Блок _>]
```

Проверяемое значение и образцы могут принадлежать к любым типам. Языковых конструкций `case`, содержащих образцы и соответствующие им блоки, может быть произвольное количество.

В качестве примера перепишем программу, чей код приведен в листинге 4.2, с применением инструкции выбора (листинг 4.4).

Листинг 4.4. Инструкция выбора

```
print("""Какой операционной системой вы пользуетесь?
1 — Windows 11
2 — Windows 10
```

```
3 — Windows 8.1
4 — Windows 8
5 — Windows 7
6 — Другая""")
os = input("Введите число, соответствующее ответу: ")
match os:
    case "1":
        print("Вы выбрали: Windows 11")
    case "2":
        print("Вы выбрали: Windows 10")
    case "3":
        print("Вы выбрали: Windows 8.1")
    case "4":
        print("Вы выбрали: Windows 8")
    case "5":
        print("Вы выбрали: Windows 7")
    case "6":
        print("Вы выбрали: другая")
    case "":
        print("Вы не ввели число")
    case _:
        print("Мы не смогли определить вашу операционную систему")
input()
```

Сразу уясним, что исполняется только один блок — соответствующий первому совпавшему образцу. После этого обработка инструкции прекращается, и начинают исполняться инструкции, следующие за ней. Так что при выполнении кода:

```
os = "1"
match os:
    case "1":
        print("Вы выбрали: Windows 11")
    case "1":
        print("Отличный выбор!")
    case "2":
        print("Вы выбрали: Windows 10")
```

в консоли появится лишь строка:

```
Вы выбрали: Windows 11
```

Рассмотрим особенности обработки инструкцией выбора проверяемых значений разных типов:

- числа, строки и логические величины — образцы указываются в виде соответственно чисел, строк и логических величин (пример см. в листинге 4.4);
- последовательности — образец указывается в виде кортежа (также можно указать список). При этом образец может быть задан:
  - с явным перечислением всех значений — проверяемая последовательность должна содержать *только* заданные в образце значения, находящиеся в заданных позициях:

```
match seq:
    case (1, 2, 3):
        print("Последовательность 1: 1, 2, 3")
```

```
    case (1, 2, 3, 4):
        print("Последовательность 2: 1, 2, 3, 4")
    case (3, 2, 1):
        print("Последовательность 3: 3, 2, 1")
    case _:
        print("Другая последовательность")
```

Результаты:

```
seq = (1, 2, 3)        =>  Последовательность 1: 1, 2, 3
seq = (1, 2, 3, 4)     =>  Последовательность 2: 1, 2, 3, 4
seq = (3, 2, 1)        =>  Последовательность 3: 3, 2, 1
seq = (1, 2, 3, 4, 5)  =>  Другая последовательность
seq = (3, 2)           =>  Другая последовательность
```

- с явным указанием части значений (вместо остальных явно заданных значений ставятся какие-либо переменные) — проверяемая последовательность должна содержать, по крайней мере, заданные значения, находящиеся в заданных позициях. Остальные значения будут присвоены указанным в образце переменным, которые станут доступны в блоке. Пример:

```
match seq:
    case (1, 2, 3):
        print("Последовательность 1: 1, 2, 3")
    case (1, 2, 3, m):
        print("Последовательность 2: 1, 2, 3, " + str(m))
    case (n, 2, 1):
        print("Последовательность 3: " + str(n) + ", 2, 1")
    case _:
        print("Другая последовательность")
```

Результаты:

```
seq = (1, 2, 3)        =>  Последовательность 1: 1, 2, 3
seq = (1, 2, 3, 4)     =>  Последовательность 2: 1, 2, 3, 4
seq = (1, 2, 3, 56)    =>  Последовательность 2: 1, 2, 3, 56
seq = (3, 2, 1)        =>  Последовательность 3: 3, 2, 1
seq = (800, 2, 1)      =>  Последовательность 3: 800, 2, 1
seq = (1, 2, 3, 4, 5)  =>  Другая последовательность
```

Также можно указать в образце переменную, предварив ее символом звездочки (*). В эту переменную будет занесен список со всеми оставшимися элементами проверяемой последовательности. Пример:

```
match seq:
    case (1, 2, 3, *a):
        print("Последовательность 1: 1, 2, 3, " + str(a))
    case (*b, 2, 1):
        print("Последовательность 2: " + str(b) + ", 2, 1")
    case _:
        print("Другая последовательность")
```

Результаты:

```
seq = (1, 2, 3, 4)     =>  Последовательность 1: 1, 2, 3, [4]
seq = (1, 2, 3, 4, 5)  =>  Последовательность 1: 1, 2, 3, [4, 5]
```

```
seq = (1, 2, 3)          =>  Последовательность 1: 1, 2, 3, []
seq = (5, 4, 3, 2, 1)    =>  Последовательность 2: [5, 4, 3], 2, 1
seq = (3, 2)             =>  Другая последовательность
```

В последнем случае вместо переменной можно указать символ подчеркивания — тогда оставшиеся значения проверяемой последовательности не будут сохранены ни в какой переменной:

```
match seq:
    case (1, 2, 3, *_):
        print("Последовательность 1: 1, 2, 3 и, возможно, " +
              "какие-либо другие элементы")
```

- отображения — образец указывается в виде словаря. Образец можно задать:
  - с явным указанием всех значений — проверяемое отображение должно содержать, по крайней мере, заданные в образце значения под указанными ключами:

```
match mpn:
    case {"a": 1, "b": 2}:
        print("Отображение 1: a - 1, b - 2")
    case {"a": 1, "c": 3 }:
        print("Отображение 2: a - 1, c - 3")
    case _:
        print("Другое отображение")
```

Результаты:

```
mpn = {"a": 1, "b": 2}          => Отображение 1: a - 1, b - 2
mpn = {"a": 1, "b": 2, "c": 3}  => Отображение 1: a - 1, b - 2
mpn = {"a": 1, "c": 3}          => Отображение 2: a - 1, c - 3
mpn = {"e": 11, "f": -32}       => Другое отображение
```

  - с явным указанием части значений (у остальных после ключей ставятся переменные) — проверяемое отображение должно содержать, по крайней мере, заданные значения и какие-либо значения, указанные под ключами, в которых поставлены переменные. Значения из ключей с переменными будут присвоены этим переменным, которые станут доступными в блоке. Пример:

```
match mpn:
    case {"a": 1, "b": 2, "d": d}:
        print("Отображение 1: a - 1, b - 2, d - " + str(d))
    case {"a": 1, "b": 2}:
        print("Отображение 2: a - 1, b - 2")
```

Результаты:

```
mpn = {"a": 1, "b": 2, "d": 3}  => Отображение 1: a - 1, b - 2, d - 3
mpn = {"a": 1, "b": 2, "d": 25} => Отображение 1: a - 1, b - 2, d - 25
mpn = {"a": 1, "b": 2}          => Отображение 2: a - 1, b - 2
mpn = {"a": 1, "b": 2, "c": 3}  => Отображение 2: a - 1, b - 2
```

После перечисления значений в образце можно поставить переменную, предварив ее двойным символом звездочки (**). В эту переменную будет занесен словарь со всеми оставшимися элементами проверяемого отображения. Пример:

```
match mpn:
    case {"a": 1, "b": 2, **d}:
        print("Отображение 1: a - 1, b - 2, " + str(d))
    case {"a": 1, "d": 4}:
        print("Отображение 2: a - 1, d - 4")
```

Результаты:

```
mpn = {"a": 1, "b": 2, "c": 3}
                    => Отображение 1: a - 1, b - 2, {'c': 3}
mpn = {"a": 1, "b": 2, "c": 3, "d": 4}
                    => Отображение 1: a - 1, b - 2, {'c': 3, 'd': 4}
mpn = {"a": 1, "b": 2}          => Отображение 1: a - 1, b - 2, {}
mpn = {"a": 1, "d": 4}          => Отображение 2: a - 1, d - 4
```

- объекты классов (описаны в *главе 13*) — можно проверять:
  - принадлежность проверяемого объекта указанному классу — образец записывается в формате `<Имя нужного класса>()`:

```
import datetime
match val:
    case str():
        print("Строка")
    case int():
        print("Целое число")
    case datetime.datetime():
        print("Временная отметка")
```

Результаты:

```
val = "Python"                          => Строка
val = 589                               => Целое число
val = datetime.datetime(2020, 12, 28)   => Временная отметка
```

  - дополнительно — наличие в атрибутах проверяемого объекта указанных значений — образец записывается в формате:

```
<Имя нужного класса>(<Атрибут 1>=<Значение атрибута 1>,
                     <Атрибут 2>=<Значение атрибута 2>,
                     . . .,
                     <Атрибут n>=<Значение атрибута n>)
```

Пример:

```
import datetime
match val:
    case str():
        print("Строка")
    case datetime.datetime(year=2020, month=12, day=28):
        print("28 декабря 2020 года")
    case datetime.datetime(year=2021, month=1, day=1):
        print("1 января 2021 года")
    case datetime.datetime():
        print("Другие дата и время")
```

Результаты:

```
val = "Python"                          => Строка
val = datetime.datetime(2020, 12, 28)   => 28 декабря 2020 года
val = datetime.datetime(2021, 1, 1)     => 1 января 2021 года
val = datetime.datetime(2021, 1, 2)     => Другие дата и время
```

В конструкции <Атрибут>=<Значение атрибута> вместо значения атрибута можно задать переменную. Эта переменная получит значение, хранящееся в соответствующем атрибуте проверяемого объекта класса, и будет доступна в блоке. Пример:

```
import datetime
match val:
    case datetime.datetime(year=2021, month=1, day=1):
        print("1 января 2021 года")
    case datetime.datetime(year=2021, month=2, day=d):
        print(str(d) + " февраля 2021 года")
```

Результаты:

```
val = datetime.datetime(2021, 1, 1)     => 1 января 2021 года
val = datetime.datetime(2021, 2, 20)    => 20 февраля 2021 года
```

В качестве образца можно указать набор значений, разделенных символами вертикальной черты (|). Соответствующий блок будет выполнен, если проверяемое значение совпадает с любым из перечисленных образцов. Пример:

```
match n:
    case "Python" | "JavaScript" | "PHP" | "Ruby":
        print("Язык программирования")
    case "Django" | "Express" | "Laravel" | "Rails":
        print("Веб-фреймворк")
```

Результаты:

```
n = "Python"                => Язык программирования
n = "Laravel"               => Веб-фреймворк
n = "Express"               => Веб-фреймворк
n = "Ruby"                  => Язык программирования
```

В языковой конструкции case можно дополнительно указать конструкцию if следующего формата:

```
case <Образец> if <Условие>
```

В этом случае проверяемое значение будет считаться совпавшим с указанным образцом только в том случае, если заданное условие в качестве результата выдаст True. Пример (функция len() возвращает количество значений в последовательности):

```
match seq:
    case (1, 2, 3, *a) if (len(seq) > 4):
        print("Более 4 значений: 1, 2, 3, " + str(a))
    case (1, 2, 3, *a):
        print("Не менее 4 значений: 1, 2, 3, " + str(a))
```

Результаты:

```
seq = (1, 2, 3, 4, 5)       =>  Более 4 значений: 1, 2, 3, [4, 5]
seq = (1, 2, 3, 4)          =>  Не менее 4 значений: 1, 2, 3, [4]
```

Инструкция выбора несколько нагляднее инструкции ветвления с множественными языковыми конструкциями `elif` (см. *разд. 4.2*) и применяется в тех же случаях, что и последняя.

4.4. Цикл перебора последовательности

Цикл перебора последовательности выполняет указанный блок, называемый *телом цикла*, столько раз, сколько значений присутствует в заданной последовательности. Перед каждым выполнением блока (*итерацией*) он присваивает очередное значение из последовательности указанной переменной, которая становится доступной в теле цикла. Формат записи цикла перебора последовательности:

```
for <Переменная> in <Последовательность>:
    <Тело цикла>
else:
    <Блок else>
```

`<Блок else>` выполняется, если указанная последовательность пуста (не содержит ни одного элемента), и исполнение цикла не было прервано оператором `break` (описан в *разд. 4.8*).

Цикл такого типа может обрабатывать любые последовательности: списки, кортежи, диапазоны, строки и др.

Пример перебора символов в строке приведен в листинге 4.5.

Листинг 4.5. Перебор строки

```
for s in "str":
    print(s, end=" ")
else:
    print("\nЦикл выполнен")
```

Результат выполнения:

```
s t r
Цикл выполнен
```

Перебор элементов списка и кортежа с построчным выводом (листинг 4.6).

Листинг 4.6. Перебор списка и кортежа

```
for x in [10, 200, 3000]:
    print(x)
for y in (3000, 200, 10):
    print(y)
```

Перебор чисел из диапазона между 5 и 15:

```
>>> for n in range(5, 15): print(n, end=" ")
...
    5 6 7 8 9 10 11 12 13 14
```

Перебор списка кортежей:

```
>>> arr = [(1, 2), (3, 4)]          # Список кортежей
>>> for a, b in arr:
...     print(a, b)
...
1 2
3 4
```

4.5. Цикл с условием

Выполнение тела *цикла с условием* `while` продолжается до тех пор, пока выполняется заданное условие. Цикл записывается в следующем формате:

```
while <Условие>:
    <Тело цикла>
[else:
    <Блок else>
]
```

`<Блок else>` будет выполнен, если указанное условие изначально не выполняется и исполнение цикла не было прервано оператором `break` (описан в *разд. 4.8*).

В теле цикла с условием следует произвести какие-либо действия, которые в определенный момент сделают так, чтобы указанное условие перестало выполняться и цикл завершился. Например, условие может проверять, не превысило ли значение какой-либо переменной указанную величину, — в таком случае в блоке следует увеличивать значение этой переменной на 1.

В качестве примера выведем все числа от 1 до 100, используя цикл `while` (листинг 4.7).

Листинг 4.7. Вывод чисел от 1 до 100

```
i = 1
while i < 101:
    print(i)
    i += 1
```

> ***Внимание!***
>
> Если в теле цикла с условием не предпринимается никаких действий, способных прервать цикл, он будет выполняться бесконечно (*бесконечный цикл*).
>
> Прервать выполнение бесконечного цикла, выводящего данные, можно нажатием комбинации клавиш <Ctrl>+<C>. В результате генерируется исключение `KeyboardInterrupt` (исключения рассматриваются в *главе 14*), и выполнение программы останавливается.

Выведем все числа от 100 до 1 (листинг 4.8).

Листинг 4.8. Вывод чисел от 100 до 1

```
i = 100
while i:
    print(i)
    i -= 1
```

Здесь условие не содержит операторов сравнения. На каждой итерации цикла мы вычитаем единицу из значения переменной `i`. Как только значение станет равным 0, цикл завершится, поскольку число 0 при преобразовании в логический тип даст значение `False`.

С помощью цикла с условием можно перебирать элементы последовательностей. Однако такой цикл работает медленнее цикла перебора последовательности (см. *разд. 4.4*). В качестве примера умножим каждый элемент списка на 2 (листинг 4.9).

Листинг 4.9. Перебор элементов списка

```
arr = [1, 2, 3]
i, count = 0, len(arr)
while i < count:
    arr[i] *= 2
    i += 1
print(arr)              # Результат выполнения: [2, 4, 6]
```

4.6. Оператор *continue*: переход на следующую итерацию цикла

Оператор `continue` позволяет перейти к следующей итерации цикла до завершения выполнения текущей итерации. В качестве примера выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно (листинг 4.10).

Листинг 4.10. Оператор `continue`

```
for i in range(1, 101):
    if 4 < i < 11:
        continue            # Переходим на следующую итерацию цикла
    print(i)
```

4.7. Оператор *break*: прерывание цикла

Оператор `break` прерывает выполнение цикла досрочно, после чего начнут исполняться инструкции, следующие за прерванным циклом. Для примера выведем все числа от 1 до 100 еще одним способом (листинг 4.11).

Листинг 4.11. Оператор `break`

```
i = 1
while True:
    if i > 100: break       # Прерываем цикл
    print(i)
    i += 1
```

Поскольку в условии указано значение `True`, цикл станет выполняться бесконечно. Однако оператор `break` прерывает цикл, как только он будет выполнен 100 раз.

Цикл с условием совместно с оператором `break` удобно использовать для получения не определенного заранее количества данных от пользователя. В качестве примера просуммируем произвольное количество чисел (листинг 4.12).

Листинг 4.12. Суммирование не определенного заранее количества чисел

```
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = input("Введите число: ")
    if x == "stop":
        break                       # Выход из цикла
    x = int(x)                      # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода трех чисел и получения суммы выглядит так (значения, введенные пользователем, здесь выделены полужирным шрифтом):

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число: 20
Введите число: 30
Введите число: stop
Сумма чисел равна: 60
```

4.8. Оператор присваивания в составе инструкции

Оператор присваивания в составе инструкции := полностью аналогичен оператору присваивания =, за тем исключением, что он выдает результат — само присвоенное значение. Это позволяет использовать его в условиях инструкций ветвления, выбора и циклов. Поддержка такого оператора появилась в Python 3.8.

Использование оператора присваивания в составе инструкции лучше всего показать на примере. Рассмотрим код, получающий количество символов в строке из переменной `s` (с помощью функции `len()`) и выводящий это количество, если оно не равно 0:

```
l = len(s)
if l > 0:
    print("Длина строки: " + str(l))
```

Этот код можно сократить, выполнив вычисление длины строки и присваивание ее переменной `n` непосредственно в условии инструкции ветвления — посредством оператора :=:

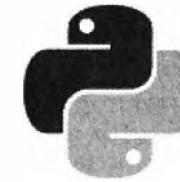
```
if (l := len(s)) > 0:
    print("Длина строки: " + str(l))
```

Выражение, включающее оператор присваивания в составе инструкции, следует взять в круглые скобки, поскольку этот оператор имеет крайне низкий приоритет — даже ниже, чем у операторов сравнения.

Использовав в условии цикла оператор присваивания в составе инструкции, мы можем несколько сократить код программы из листинга 4.12 (листинг 4.13).

Листинг 4.13. Оператор присваивания в составе инструкции

```
print("Введите слово 'stop' для получения результата")
summa = 0
while (x := input("Введите число: ")) != "stop":
    x = int(x)
    summa += x
print("Сумма чисел равна:", summa)
input()
```



ГЛАВА 5

Числа

Язык Python поддерживает следующие числовые типы данных:

- `int` — целые числа. Размер числа ограничен лишь объемом оперативной памяти;
- `float` — вещественные числа;
- `complex` — комплексные числа.

Операции над числами разных типов возвращают число, имеющее более сложный тип из типов, участвующих в операции. Целые числа имеют самый простой тип, далее идут вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое число и вещественное, то целое число будет автоматически преобразовано в вещественное число, а затем произведена операция над вещественными числами. Результатом этой операции будет вещественное число.

5.1. Запись чисел

Целые числа в десятичной системе счисления записываются как есть:

```
>>> x = 0; y = 10; z = -80
>>> x, y, z
    (0, 10, -80)
```

Можно записать целое число в двоичной, восьмеричной или шестнадцатеричной форме. Такие числа будут автоматически преобразованы в десятичные целые числа.

- Двоичные числа начинаются с комбинации символов `0b` (или `0B`) и содержат цифры 0 или 1:

```
>>> 0b11111111, 0b101101
    (255, 45)
```

- Восьмеричные числа начинаются с нуля и следующей за ним латинской буквы `o` (регистр не имеет значения) и содержат цифры от 0 до 7:

```
>>> 0o7, 0o12, 0o777, 0O7, 0O12, 0O777
    (7, 10, 511, 7, 10, 511)
```

- Шестнадцатеричные числа начинаются с комбинации символов `0x` (или `0X`) и могут содержать цифры от 0 до 9 и буквы от `A` до `F` (регистр не имеет значения):

```
>>> 0X9,0xA, 0x10, 0xFFF, 0xfff
    (9, 10, 16, 4095, 4095)
```

Длинные целые числа для улучшения читаемости кода можно разбивать на группы цифр, вставляя символы подчеркивания:

```
>>> 1_000_000
    1000000
>>> 0b1111_1111
    255
>>> 0o2_777
    1535
>>> 0xab_cd
    43981
```

Вещественное число может содержать точку и (или) быть представлено в экспоненциальной форме с буквой `E` (регистр не имеет значения). Начальный 0 можно не указывать. Примеры:

```
>>> 10., .14, 3.14, 11E20, 2.5e-12
    (10.0, 0.14, 3.14, 1.1e+21, 2.5e-12)
```

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей операции может показаться странным:

```
>>> 0.3 - 0.1 - 0.1 - 0.1
    -2.7755575615628914e-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другое значение. Если необходимо производить операции с фиксированной точностью, то следует использовать модуль `decimal`:

```
>>> from decimal import Decimal
>>> Decimal("0.3") - Decimal("0.1") - Decimal("0.1") - Decimal("0.1")
    Decimal('0.0')
```

Кроме того, можно использовать дроби, поддержка которых реализована в модуле `fractions`. При создании дроби можно указать два целых числа (числитель и знаменатель), одно вещественное число или строку с вещественным числом.

Для примера создадим несколько дробей. Вот так формируется дробь $^4/_5$:

```
>>> from fractions import Fraction
>>> Fraction(4, 5)
    Fraction(4, 5)
```

А вот так — дробь $^1/_2$, причем можно сделать это тремя способами:

```
>>> Fraction(1, 2)
    Fraction(1, 2)
>>> Fraction(0.5)
    Fraction(1, 2)
>>> Fraction("0.5")
    Fraction(1, 2)
```

Над дробями можно производить арифметические операции, как и над обычными числами:

```
>>> Fraction(9, 5) - Fraction(2, 3)
    Fraction(17, 15)
```

```
>>> Fraction("0.3") - Fraction("0.1") - Fraction("0.1") - Fraction("0.1")
    Fraction(0, 1)
>>> float(Fraction(0, 1))
    0.0
```

Комплексные числа записываются в формате:

```
<Вещественная часть>+<Мнимая часть>J
```

Здесь буква `J` может стоять в любом регистре. Примеры комплексных чисел:

```
>>> 2+5J, 8j
    ((2+5j), 8j)
```

Подробное рассмотрение модулей `decimal` и `fractions`, а также комплексных чисел выходит за рамки нашей книги. За информацией обращайтесь к документации по Python.

5.2. Обработка чисел

Для работы с числами предназначены следующие функции:

- `int([<Значение>[, <Система счисления>]])` — преобразует заданное значение в целое число. Во втором параметре можно указать систему счисления преобразуемого числа (по умолчанию: 10). Пример:

```
>>> int(7.5), int("71", 10), int("0o71", 8), int("0xA", 16)
    (7, 71, 57, 10)
>>> int(), int("0b11111111", 2)
    (0, 255)
```

- `float([<Число или строка>])` — преобразует заданное целое число или строку в вещественное число:

```
>>> float(7), float("7.1"), float("12.")
    (7.0, 7.1, 12.0)
>>> float("inf"), float("-Infinity"), float("nan")
    (inf, -inf, nan)
>>> float()
    0.0
```

- `bin(<Число>)` — преобразует заданное десятичное число в двоичное и возвращает его в виде строки:

```
>>> bin(255), bin(1), bin(-45)
    ('0b11111111', '0b1', '-0b101101')
```

- `oct(<Число>)` — преобразует заданное десятичное число в восьмеричное и возвращает его в виде строки:

```
>>> oct(7), oct(8), oct(64)
    ('0o7', '0o10', '0o100')
```

- `hex(<Число>)` — преобразует заданное десятичное число в шестнадцатеричное и возвращает его в виде строки:

```
>>> hex(10), hex(16), hex(255)
    ('0xa', '0x10', '0xff')
```

- `round(<Число>[, <Количество знаков после точки>])` — округляет заданное число и возвращает результат. Числа с дробной частью, меньшей `0.5`, округляет до ближайшего меньшего целого, числа с дробной частью, большей `0.5`, — до ближайшего большего целого, числа с дробной частью, равной `0.5`, — до ближайшего четного числа. Примеры:

```
>>> round(0.49), round(0.50), round(0.51)
    (0, 0, 1)
>>> round(1.49), round(1.50), round(1.51)
    (1, 2, 2)
>>> round(2.49), round(2.50), round(2.51)
    (2, 2, 3)
>>> round(3.49), round(3.50), round(3.51)
    (3, 4, 4)
```

  Во втором параметре можно указать желаемое количество знаков после запятой (по умолчанию 0 — т. е. число будет округлено до целого):

```
>>> round(1.524, 2), round(1.525, 2), round(1.5555, 3)
    (1.52, 1.52, 1.556)
```

- `abs(<Число>)` — возвращает абсолютное значение заданного числа:

```
>>> abs(-10), abs(10), abs(-12.5)
    (10, 10, 12.5)
```

- `pow(<Основание>, <Показатель>[, <Делитель>])` — возводит `<Основание>` в степень, заданную `<Показателем>`, и возвращает результат:

```
>>> pow(10, 2), 10 ** 2, pow(3, 3), 3 ** 3
    (100, 100, 27, 27)
```

  Если указан третий параметр, возвращается остаток от деления полученного результата на значение этого параметра:

```
>>> pow(10, 2, 2), (10 ** 2) % 2, pow(3, 3, 2), (3 ** 3) % 2
    (0, 0, 1, 1)
```

- `max()` — возвращает максимальное число. Поддерживает два формата вызова:

```
max(<Число 1>, <Число 2>, . . ., <Число n>[, key=<Функция, выдающая число>])
max(<Последовательность>[, key=<Функция, выдающая число>][,
                         default=<Значение по умолчанию>])
```

  Первый формат возвращает максимальное число из указанных:

```
>>> max(1, 2, 3), max(3, 2, 3, 1), max(1, 1.0), max(1.0, 1)
    (3, 3, 1, 1.0)
```

  Вместо чисел можно указать значения другого типа. Однако тогда в именованном параметре `key` следует задать функцию, которая будет получать с единственным параметром очередное значение и возвращать его же, преобразованное в числовой тип. Результат в этом случае будет возвращаться в изначальном виде, безо всяких преобразований. Пример с числами, указанными в виде строк (для преобразования используется функция `str()`):

```
>>> max("1", "435", "65", "-57", key=str)
    '65'
```

Второй формат возвращает максимальное число из содержащихся в указанной последовательности (например, в списке):

```
>>> max([1, 2, 3]), max([3, 2, 3, 1]), max([1, 1.0]), max([1.0, 1])
    (3, 3, 1, 1.0)
>>> seq = ["1", "435", "65", "-57"]
>>> max(seq, key=str)
    '65'
```

Если указанная последовательность пуста, генерируется исключение `ValueError`, которое приведет к аварийному завершению программы (если его не обработать, о чем рассказывается в *главе 14*):

```
>>> max([])
    Traceback (most recent call last):
      File "<pyshell#19>", line 1, in <module>
        max([])
    ValueError: max() arg is an empty sequence
```

Однако можно указать в именованном параметре `default` значение, которое будет возвращаться функцией при передаче ей пустой последовательности (исключение в этом случае не генерируется):

```
>>> max([], default=0)
    0
```

- `min()` — возвращает минимальное число. Оба поддерживаемых формата вызова аналогичны таковым у функции `max()`. Примеры:

```
>>> min(1, 2, 3), min(3, 2, 3, 1), min(1, 1.0), min(1.0, 1)
    (1, 1, 1, 1.0)
>>> min("1", "435", "65", "-57", key=str)
    '-57'
>>> min([1, 2, 3]), min([3, 2, 3, 1]), min([1, 1.0]), min([1.0, 1])
    (1, 1, 1, 1.0)
>>> seq = ["1", "435", "65", "-57"]
>>> min(seq, key=str)
    '-57'
```

- `sum(<Последовательность>[, <Начальное значение>])` — возвращает сумму значений элементов заданной последовательности (списка, кортежа и пр.) плюс `<Начальное значение>`. Если второй параметр не указан, начальное значение принимается равным 0. Если последовательность пуста, возвращается значение второго параметра. Пример:

```
>>> sum((10, 20, 30, 40)), sum([10, 20, 30, 40])
    (100, 100)
>>> sum([10, 20, 30, 40], 2), sum([], 2)
    (102, 2)
```

- `divmod(x, y)` — возвращает кортеж из двух значений `(x // y, x % y)`:

```
>>> divmod(13, 2)              # 13 == 6 * 2 + 1
    (6, 1)
>>> 13 // 2, 13 % 2
    (6, 1)
>>> divmod(13.5, 2.0)          # 13.5 == 6.0 * 2.0 + 1.5
    (6.0, 1.5)
```

```
>>> 13.5 // 2.0, 13.5 % 2.0
    (6.0, 1.5)
```

В *главе 2* говорилось, что в Python любое значение представляется в виде объекта — сложной структуры, хранящей, помимо собственно значения, всевозможные связанные с ней данные. Эти данные хранятся в *атрибутах* — переменных, принадлежащих объекту. Объект также содержит *методы* — принадлежащие ему функции, которые позволяют манипулировать содержащимся в нем значением и получать какие-либо сведения о нем.

Все объекты, хранящие значения одного типа, создаются на основе одного *класса* — своего рода образца, определяющего функциональность типа, набор содержащихся в нем атрибутов и методов. Имя класса совпадает с именем соответствующего ему типа данных (так, класс `int` служит для представления целых чисел, а класс `float` — вещественных чисел).

Класс `float`, представляющий вещественные числа, поддерживает следующие полезные методы:

- `is_integer()` — возвращает `True`, если *текущий объект* (объект, у которого вызывается метод) вещественного числа не содержит дробной части — т. е. фактически представляет собой целое число:

  ```
  >>> (2.0).is_integer()
      True
  >>> (2.3).is_integer()
      False
  ```

  Обратим внимание, как вызывается метод: его вызов ставится после объекта и отделяется от него точкой;

- `as_integer_ratio()` — возвращает кортеж из двух целых чисел, представляющих собой числитель и знаменатель дроби, которая соответствует текущему числу:

  ```
  >>> (0.5).as_integer_ratio()
      (1, 2)
  >>> (2.3).as_integer_ratio()
      (2589569785738035, 1125899906842624)
  ```

Начиная с Python 3.8, класс `int`, представляющий целые числа, также поддерживает метод `as_integer_ratio()`, только знаменатель возвращаемой им дроби всегда равен 1:

```
>>> (2).as_integer_ratio()
    (2, 1)
>>> (57).as_integer_ratio()
    (57, 1)
```

5.3. Математические функции

Модуль `math` предоставляет дополнительные функции для работы с числами, а также стандартные константы. Прежде чем использовать этот модуль, необходимо подключить его с помощью инструкции:

```
import math
```

> ***Примечание***
>
> Для работы с комплексными числами необходимо использовать модуль `cmath`.

Модуль `math` предоставляет стандартные константы, хранящиеся в следующих переменных:

- `pi` — число π:

```
>>> import math
>>> math.pi
    3.141592653589793
```

- `e` — константа *e*:

```
>>> math.e
    2.718281828459045
```

- `tau` — число τ, равное 2π:

```
>>> math.tau
   6.283185307179586
```

Также модуль `math` поддерживает следующие основные функции:

- `sin(<Угол>)`, `cos(<Угол>)`, `tan(<Угол>)` — синус, косинус и тангенс. Угол указывается в радианах;
- `asin(<Значение>)`, `acos(<Значение>)`, `atan(<Значение>)` — арксинус, арккосинус и арктангенс. Результат возвращается в радианах;
- `degrees(<Угол в радианах>)` — угол в градусах:

```
>>> math.degrees(math.pi)
    180.0
```

- `radians(<Угол в градусах>)` — угол в радианах:

```
>>> math.radians(180.0)
    3.141592653589793
```

- `exp(<Число>)` — экспонента от заданного числа;
- `log(<Число>[, <База>])` — логарифм заданного числа по указанной базе. Если база не указана, вычисляется натуральный логарифм (по базе *e*);
- `log10(<Число>)` — десятичный логарифм заданного числа;
- `log2(<Число>)` — логарифм по базе 2 от заданного числа;
- `sqrt(<Число>)` — квадратный корень от заданного числа:

```
>>> math.sqrt(100), math.sqrt(25)
    (10.0, 5.0)
```

- `ceil(<Значение>)` — заданное значение, округленное до ближайшего большего целого:

```
>>> math.ceil(5.49), math.ceil(5.50), math.ceil(5.51)
    (6, 6, 6)
```

- `floor()` — заданное значение, округленное до ближайшего меньшего целого:

```
>>> math.floor(5.49), math.floor(5.50), math.floor(5.51)
    (5, 5, 5)
```

- `pow(<Основание>, <Показатель>)` — указанное основание, возведенное в степень, которая задана `<Показателем>`. В отличие от оператора `**`, всегда возвращает результат в виде вещественного числа. Пример:

```
>>> math.pow(10, 2), 10 ** 2, math.pow(3, 3), 3 ** 3
    (100.0, 100, 27.0, 27)
```

- `fabs(<Число>)` — абсолютное значение заданного числа. В отличие от функции `abs()`, всегда возвращает результат в виде вещественного числа. Пример:

```
>>> math.fabs(10), math.fabs(-10), math.fabs(-12.5)
    (10.0, 10.0, 12.5)
```

- `fmod(<Делимое>, <Делитель>)` — остаток от деления. В отличие от оператора `%`, всегда возвращает результат в виде вещественного числа. Пример:

```
>>> math.fmod(10, 5), 10 % 5, math.fmod(10, 3), 10 % 3
    (0.0, 0, 1.0, 1)
```

- `factorial(<Число>)` — факториал заданного числа. Начиная с Python 3.8, принимает только целые числа. Пример:

```
>>> math.factorial(5), math.factorial(6)
    (120, 720)
```

- `fsum(<Последовательность>)` — возвращает точную сумму чисел из заданной последовательности:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
    0.9999999999999999
>>> math.fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
    1.0
```

> ***Примечание***
>
> В этом разделе мы рассмотрели только основные функции, поддерживаемые модулем `math`. Чтобы получить полный список функций, обращайтесь к документации по Python.

5.4. Генерирование случайных чисел

Модуль `random` предоставляет инструменты для генерирования случайных чисел. Сначала необходимо подключить его с помощью инструкции:

```
import random
```

Основные функции, содержащиеся в этом модуле:

- `random()` — возвращает псевдослучайное число от `0.0` до `1.0`:

```
>>> import random
>>> random.random()
    0.9753144027290991
>>> random.random()
    0.5468390487484339
>>> random.random()
    0.13015058054767736
```

- `seed([<База>][, version=2])` — настраивает генератор случайных чисел на новую последовательность. Если первый параметр не указан, в качестве базы для случайных чисел будет использовано системное время. При одинаковых значениях базы генерируется одинаковая последовательность чисел. Пример:

```
>>> random.seed(10)
>>> random.random()
    0.5714025946899135
>>> random.seed(10)
>>> random.random()
    0.5714025946899135
```

- `uniform(<Начало>, <Конец>)` — возвращает псевдослучайное вещественное число в диапазоне от параметра `<Начало>` до параметра `<Конец>`:

```
>>> random.uniform(0, 10)
    9.965569925394552
>>> random.uniform(0, 10)
    0.4455638245043303
```

- `randint(<Начало>, <Конец>)` — возвращает псевдослучайное целое число в диапазоне от параметра `<Начало>` до параметра `<Конец>`:

```
>>> random.randint(0, 10)
    4
>>> random.randint(0, 10)
    10
```

- `randrange([<Начало>, ]<Конец>[, <Шаг>])` — возвращает случайный элемент из создаваемого «за кадром» диапазона:

```
>>> random.randrange(10)
    5
>>> random.randrange(0, 10)
    2
>>> random.randrange(0, 10, 2)
    6
```

- `choice(<Последовательность>)` — возвращает случайный элемент из заданной последовательности:

```
>>> random.choice("string")          # Строка
    'i'
>>> random.choice(["s", "t", "r"])   # Список
    'r'
>>> random.choice(("s", "t", "r"))   # Кортеж
    't'
```

- `shuffle(<Список>)` — перемешивает элементы заданного списка случайным образом. Результата не возвращает. Пример:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.shuffle(arr)
>>> arr
    [8, 6, 9, 5, 3, 7, 2, 4, 10, 1]
```

Ранее эта функция принимала два параметра, но, начиная с Python 3.9, второй параметр объявлен нерекомендованным к использованию и подлежащим удалению в Python 3.11;

- `sample()` — возвращает список из указанного количества элементов, которые будут выбраны случайным образом из заданной последовательности:

```
sample(<Последовательность>, <Количество элементов>[,
                              counts=<Количество повторений элементов>])
>>> random.sample("string", 2)
    ['i', 'r']
>>> random.sample("string", 6)
    ['i', 'r', 'n', 'g', 't', 's']
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 2)
    [7, 10]
>>> arr # Сам список не изменяется
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample((1, 2, 3, 4, 5, 6, 7), 3)
    [6, 3, 5]
>>> random.sample(range(300), 5)
    [126, 194, 272, 46, 71]
```

Начиная с Python 3.9, поддерживается именованный параметр `counts`, в котором можно указать последовательность, содержащую значения количества повторений элементов из последовательности, заданной первым параметром. Первый элемент последовательности из параметра `counts` укажет количество повторений первого элемента из первой последовательности, второй элемент — количество повторений второго элемента и т. д. Например, следующие два вызова функции `sample()` эквивалентны:

```
random.sample("sssttrrrrriinggg", 6)
random.sample("string", 6, counts=[3, 2, 5, 2, 1, 3])
```

Также, начиная с Python 3.9, указывать множества в первом параметре функции `sample()` не допускается — эта возможность объявлена устаревшей и подлежащей удалению в будущих версиях языка.

Напишем функцию-генератор паролей произвольной длины (листинг 5.1). Для этого добавим в список `arr` все разрешенные символы, а далее в цикле будем получать случайный элемент с помощью функции `choice()`. По умолчанию будет выдаваться пароль из 8 символов.

Листинг 5.1. Генератор паролей

```
import random
def passw_generator(count_char=8):
    arr = ['a','b','c','d','e','f','g','h','i','j','k','l','m',
    'n','o','p','q','r','s','t','u','v','w','x','y','z',
    'A','B','C','D','E','F','G','H','I','J','K','L',
    'M','N','O','P','Q','R','S','T','U','V', 'W',
    'X','Y','Z','1','2','3','4','5','6','7','8','9','0']
    passw = []
    for i in range(count_char):
        passw.append(random.choice(arr))
    return "".join(passw)
```

Испытаем эту функцию в действии:

```
print(passw_generator(10))  # Выведет что-то вроде ngODHE8J8x
print(passw_generator())    # Выведет что-то вроде ZxcpkF5O
```

ГЛАВА 6

Строки и двоичные данные

Строка — это неизменяемая последовательность произвольных символов в кодировке Unicode. Длина строки ограничена лишь объемом оперативной памяти компьютера.

Двоичные данные — это последовательность байтов (чисел от 0 до 255), которая может быть как неизменяемой, так и изменяемой. Длина такой последовательности также ограничена лишь объемом оперативной памяти. В виде двоичных данных может быть сохранена информация любого рода: строка, графическое изображение, архив и др.

6.1. Создание строк

Создать строку можно следующими способами:

- записав составляющие строку символы между одинарными или двойными кавычками:

```
>>> 'строка', "строка"
    ('строка', 'строка')
```

Строки, созданные с применением одинарных и двойных кавычек, ничем не отличаются друг от друга.

В строках, созданных одинарными кавычками, можно размещать двойные кавычки, а в строках, созданных двойными кавычками, — одинарные кавычки:

```
>>> 'Группа "Кино"', "О'Брайен"
    ('Группа "Кино"', "О'Брайен")
```

Попытка вставить одинарную кавычку в строку, созданную одинарными кавычками, или двойную кавычку в строку в двойных кавычках приведет к ошибке:

```
>>> "Группа "Кино""
    SyntaxError: invalid syntax
```

В строках можно указать *специальные символы* — символы, обрабатываемые особым образом. Обозначение специального символа начинается со знака обратного слеша (\). Например, специальный символ `\n` обозначает разрыв строки, специальный символ `\'` — одинарную кавычку, символ `\"` — двойную кавычку, а `\\` — обратный слеш. Более подробно специальные символы будут описаны в *разд. 6.1.1*. Примеры:

```
>>> print("Строка1\\nСтрока2")
Строка1\nСтрока2
```

```
>>> "Группа \"Кино\"", 'О\'Брайен'
    ('Группа "Кино"', "О'Брайен")
```

Специальный символ \\ обычно применяется для вставки символа обратного слеша в конец строки:

```
>>> print("string\\")
    string\
```

Если же использовать единичный символ обратного слеша, интерпретатор посчитает его и следующую за ним кавычку специальным символом и выведет сообщение о синтаксической ошибке:

```
>>> print("string\")
    SyntaxError: unterminated string literal (detected at line 1)
```

Нельзя разбивать строковый объект, созданный с помощью кавычек, на несколько строк — это вызовет синтаксическую ошибку:

```
>>> "string
    SyntaxError: unterminated string literal (detected at line 1)
```

Чтобы расположить строковое значение на нескольких строках, следует либо перед символом перевода строки указать символ \, либо поместить отдельные части строки внутри круглых скобок, либо выполнить конкатенацию строк также внутри круглых скобок:

```
>>> "string1\
...   string2"              # После \ не должно быть никаких символов
    'string1string2'
>>> ("string1"
...   "string2")            # Неявная конкатенация строк
    'string1string2'
>>> ("string1" +
...   "string2")            # Явная конкатенация строк
    'string1string2'
```

- указав строку между утроенными одинарными или двойными кавычками. Такие строковые объекты могут размещаться на нескольких строках, содержать одинарные и двойные кавычки. Примеры:

```
>>> print('''Строка1
... Строка2''')
    Строка1
    Строка2
>>> print("""Строка1
... Строка2""")
    Строка1
    Строка2
```

- с помощью функции `str([<Значение>[, <Кодировка>[, <Обработка ошибок>]]])`. Если указан только первый параметр, функция возвращает строковое представление указанного значения. Если параметры не указаны вообще, возвращается пустая строка. Примеры:

```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
    ('', '[1, 2]', '(3, 4)', "{'x': 1}")
```

При попытке преобразовать двоичные данные типа `byte` или `bytearray` (описаны в *разд. 6.11* и *6.12*) в строку будет выдано строковое представление двоичных данных:

```
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0")
    "b'\\xf1\\xf2\\xf0\\xee\\xea\\xe0'"
```

Чтобы получить строку, следует указать кодировку:

```
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0", "cp1251")
    'строка'
```

В третьем параметре могут быть указаны значения `"strict"` (при ошибке возбуждается исключение `UnicodeDecodeError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется символом с кодом `\uFFFD`) или `"ignore"` (неизвестные символы игнорируются):

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
    ('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
    Traceback (most recent call last):
      File "<pyshell#16>", line 1, in <module>
        str(obj1, "ascii", "strict")
    UnicodeDecodeError: 'ascii' codec can't decode byte
    0xd1 in position 0: ordinal not in range(128)
>>> str(obj1, "ascii", "ignore")
    '1'
```

Если строка не присваивается переменной, то она считается *строкой документирования*. Такая строка сохраняется в атрибуте `__doc__` того объекта, в котором расположена. В качестве примера создадим функцию со строкой документирования, а затем выведем содержимое строки:

```
>>> def test():
...     """Это описание функции"""
...     pass
...
>>> print(test.__doc__)
    Это описание функции
```

6.1.1. Специальные символы

Специальные символы, как говорилось ранее, имеют особое значение и обрабатываются особым образом. Python поддерживает следующие специальные символы:

- `\n` — перевод строки;
- `\r` — возврат каретки;
- `\t` — знак табуляции;
- `\v` — вертикальная табуляция;
- `\a` — звонок;
- `\b` — забой;

- \f — перевод формата;
- \0 — нулевой символ;
- \" — двойная кавычка;
- \' — одинарная кавычка (апостроф);
- \\ — обратный слеш;
- \<N> — символ с восьмеричным кодом <N>. Например, \74 соответствует символу <;
- \x<N> — символ с шестнадцатеричным кодом <N>. Например, \x6a соответствует символу j;
- \u<nnnn> — символ с 16-битным Unicode-кодом <nnnn>. Например, \u043a соответствует русской букве к;
- \U<nnnnnnnn> — символ с 32-битным Unicode-кодом <nnnnnnnn>;
- \N{<name>} — символ с Unicode-именем <name>. Например, \N{Registered Sign} соответствует знаку зарегистрированной торговой марки ®.

Комбинация обратного слеша с любым другим символом выводится как есть:

```
>>> print("Этот символ \не специальный")
    Этот символ \не специальный
```

Тем не менее для вставки обратного слеша лучше использовать специальный символ \\:

```
>>> print("Этот символ \\не специальный")
    Этот символ \не специальный
```

6.1.2. Необрабатываемые строки

В *необрабатываемых строках* специальные символы не обрабатываются, а выводятся как есть. Чтобы превратить строку в необрабатываемую, достаточно предварить ее модификатором r или R. Примеры:

```
>>> print("Строка1\nСтрока2")
    Строка1
    Строка2
>>> print(r"Строка1\nСтрока2")
    Строка1\nСтрока2
>>> print(R"Строка1\nСтрока2")
    Строка1\nСтрока2
>>> print(r"""Строка1\nСтрока2""")
    Строка1\nСтрока2
```

Необрабатываемые строки удобно использовать в шаблонах регулярных выражений и при записи файловых путей:

```
>>> print(r"C:\Python310\lib\site-packages")
    C:\Python310\lib\site-packages
```

В обычных строках все обратные слеши придется заменять на специальные символы \\:

```
>>> print("C:\\Python310\\lib\\site-packages")
    C:\Python310\lib\site-packages
```

Если в конце необрабатываемой строки должен располагаться слеш, следует использовать специальный символ \\. Однако этот символ будет добавлен в строку. Пример:

```
>>> print(r"C:\Python310\lib\site-packages\")
    SyntaxError: unterminated string literal (detected at line 1)
>>> print(r"C:\Python310\lib\site-packages\\")
    C:\Python310\lib\site-packages\\
```

Избавиться от лишнего слеша можно, использовав конкатенацию строк, обычные строки или удалив слеш явно:

```
>>> print(r"C:\Python310\lib\site-packages" + "\\") # Конкатенация
    C:\Python310\lib\site-packages\
>>> print("C:\\Python310\\lib\\site-packages\\")    # Обычная строка
    C:\Python310\lib\site-packages\
>>> print(r"C:\Python310\lib\site-packages\\"[:-1]) # Удаление слеша
    C:\Python310\lib\site-packages\
```

6.2. Операции над строками

Строки относятся к последовательностям и, соответственно, поддерживают все операции, выполняемые над последовательностями.

Можно извлечь любой символ строки, указав индекс этого символа в квадратных скобках. Нумерация символов начинается с нуля. Пример:

```
>>> s = "Python"
>>> s[0], s[1], s[2], s[3], s[4], s[5]
    ('P', 'y', 't', 'h', 'o', 'n')
```

При обращении к символу с несуществующим индексом возбуждается исключение IndexError:

```
>>> s = "Python"
>>> s[10]
    Traceback (most recent call last):
      File "<pyshell#90>", line 1, in <module>
        s[10]
    IndexError: string index out of range
```

Можно указать отрицательный индекс — он будет отсчитываться от конца строки:

```
>>> s = "Python"
>>> s[-1], s[-4]
    ('n', 't')
```

Так как строки относятся к неизменяемым типам данных, то изменить символ с указанным индексом нельзя:

```
>>> s = "Python"
>>> s[0] = "J"                          # Изменить строку нельзя
    Traceback (most recent call last):
      File "<pyshell#94>", line 1, in <module>
        s[0] = "J"                          # Изменить строку нельзя
    TypeError: 'str' object does not support item assignment
```

Чтобы выполнить изменение, можно воспользоваться операцией извлечения *среза* — фрагмента последовательности, также представляющего собой последовательность. Формат записи среза:

```
[<Начало>:<Конец>:<Шаг>]
```

Все параметры здесь не являются обязательными. Если параметр `<Начало>` не указан, то используется значение 0. Если параметр `<Конец>` не указан, то возвращается фрагмент до конца строки. Символ с индексом, указанным в этом параметре, не входит в возвращаемый фрагмент. Если параметр `<Шаг>` не указан, то используется значение 1. В качестве значения параметров можно указать отрицательные значения.

Примеры:

- получение копии строки:

```
>>> s = "Python"
>>> s[:]        # Возвращается фрагмент от позиции 0 до конца строки
    'Python'
```

- вывод символов в обратном порядке:

```
>>> s[::-1]     # Указываем отрицательное значение в параметре <Шаг>
    'nohtyP'
```

- замена первого символа в строке:

```
>>> "J" + s[1:] # Извлекаем фрагмент от символа 1 до конца строки
    'Jython'
```

- удаление последнего символа:

```
>>> s[:-1]      # Возвращается фрагмент от 0 до len(s)-1
    'Pytho'
```

- получение первого символа в строке:

```
>>> s[0:1]      # Символ с индексом 1 не входит в диапазон
    'P'
```

- получение последнего символа:

```
>>> s[-1:]      # Получаем фрагмент от len(s)-1 до конца строки
    'n'
```

- вывод символов с индексами 2, 3 и 4:

```
>>> s[2:5]      # Возвращаются символы с индексами 2, 3 и 4
    'tho'
```

Узнать *длину* строки (количество символов в ней) позволяет функция `len()`:

```
>>> len("Python"), len("\r\n\t"), len(r"\r\n\t")
    (6, 3, 6)
```

Можно перебрать все символы в строке с помощью цикла перебора последовательности (см. *разд. 4.4*):

```
>>> s = "Python"
>>> for i in s: print(i, end=" ")
    P y t h o n
```

Выполнить конкатенацию строк позволяет оператор +:

```
>>> print("Строка1" + "Строка2")
    Строка1Строка2
>>> s = "Строка1"
>>> print(s + "Строка2")
    Строка1Строка2
```

Кроме того, можно выполнить *неявную конкатенацию* строк, записав их через пробел:

```
>>> print("Строка1" "Строка2")
    Строка1Строка2
```

Однако выполнить неявную конкатенацию переменной и строки нельзя — это вызовет ошибку:

```
>>> print(s "Строка2")              # Ошибка
    SyntaxError: invalid syntax
```

При необходимости объединить строку со значением другого типа (например, с числом) следует произвести явное преобразование типов с помощью функции `str()`:

```
>>> "string" + str(10)
    'string10'
```

Еще строки поддерживают операцию повторения, проверки на вхождение и невхождение. Повторить строку указанное количество раз можно с помощью оператора *, выполнить проверку на вхождение фрагмента в строку позволяет оператор `in`, а проверить на невхождение — оператор `not in`:

```
>>> "-" * 20
    '--------------------'
>>> "yt" in "Python"               # Найдено
    True
>>> "yt" in "Perl"                 # Не найдено
    False
>>> "PHP" not in "Python"          # Не найдено
    True
```

6.3. Форматирование строк

При форматировании указанная строка объединяется со значениями любых других типов. Форматирование выполняется быстрее конкатенации.

Операция форматирования записывается в следующем формате:

```
<Строка специального формата> % <Значения>
```

Внутри параметра `<Строка специального формата>` могут быть указаны спецификаторы, имеющие следующий синтаксис:

```
%[(<Ключ>)][<Флаг>][<Ширина>][.<Точность>]<Тип преобразования>
```

Количество спецификаторов внутри строки должно быть равно количеству элементов в параметре `<Значения>`. Если спецификатор один, то параметр `<Значения>` может содержать одно значение, в противном случае следует задать кортеж. Примеры:

```
>>> "%s" % 10                          # Один элемент
    '10'
>>> "%s – %s – %s" % (10, 20, 30)      # Несколько элементов
    '10 – 20 – 30'
```

Параметры внутри спецификатора имеют следующий смысл:

- <Ключ> — ключ словаря. Если задан ключ, то в параметре <Значения> необходимо указать словарь, а не кортеж:

```
>>> "%(name)s – %(year)s" % {"year": 1978, "name": "Nik"}
    'Nik – 1978'
```

- <Флаг> — флаг преобразования. Может содержать следующие значения:
  - # — у восьмеричных чисел добавляет в начало комбинацию символов 0o, у шестнадцатеричных — комбинацию символов 0x (если используется тип x) или 0X (если используется тип X), у вещественных чисел предписывает всегда выводить дробную точку, даже если в параметре <Точность> задано значение 0:

```
>>> print("%#o %#o %#o" % (0o77, 0xff, 10))
    0o77 0o377 0o12
>>> print("%#x %#x %#x" % (0o77, 0xff, 10))
    0x3f 0xff 0xa
>>> print("%#X %#X %#X" % (0o77, 0xff, 10))
    0X3F 0XFF 0XA
>>> print("%#.0F %.0F" % (300, 300))
    300. 300
```

  - 0 — вывод ведущих нулей у чисел:

```
>>> "'%d' – '%05d'" % (3, 3) # 5 – ширина поля
    "'3' – '00003'"
```

  - - — выравнивание по левой границе области (по умолчанию используется выравнивание по правой границе). Если флаг указан одновременно с флагом 0, то действие последнего будет отменено. Примеры:

```
>>> "'%5d' - '%-5d'" % (3, 3) # 5 – ширина поля
    "'    3' - '3    '"
>>> "'%05d' - '%-05d'" % (3, 3)
    "'00003' - '3    '"
```

  - пробел — вывод пробела перед положительным числом и минуса — перед отрицательным:

```
>>> "'% d' - '% d'" % (-3, 3)
    "'-3' - ' 3'"
```

  - + — вывод знака как у отрицательных, так и у положительных чисел. Если флаг + указан одновременно с флагом пробел, то действие последнего будет отменено. Пример:

```
>>> "'%+d' - '%+d'" % (-3, 3)
    "'-3' - '+3'"
```

- <Ширина> — минимальная ширина поля. Если строка не помещается в указанную ширину, значение игнорируется, и строка выводится полностью. Примеры:

```
>>> "'%10d' - '%-10d'" % (3, 3)
    "'         3' - '3         '"
>>> "'%3s''%10s'" % ("string", "string")
    "'string''    string'"
```

Вместо значения ширины можно указать символ «*». В этом случае ширину следует задать внутри кортежа. Пример:

```
>>> "'%*s''%10s'" % (10, "string", "str")
    "'    string''       str'"
```

- <Точность> — количество знаков после точки у вещественных чисел. Перед этим параметром обязательно должна стоять точка. Пример:

```
>>> import math
>>> "%s %f %.2f" % (math.pi, math.pi, math.pi)
    '3.141592653589793 3.141593 3.14'
```

Вместо значения точности можно указать символ «*». В этом случае точность следует задать внутри кортежа. Пример:

```
>>> "'%*.*f'" % (8, 5, math.pi)
    "' 3.14159'"
```

- <Тип преобразования>:
  - s — преобразование выводимого значения в строку с помощью функции str():

```
>>> print("%s" % ("Обычная строка"))
    Обычная строка
>>> print("%s %s %s" % (10, 10.52, [1, 2, 3]))
    10 10.52 [1, 2, 3]
```

  - r — преобразование выводимого значения в строку с помощью функции repr():

```
>>> print("%r" % ("Обычная строка"))
    'Обычная строка'
```

  - a — преобразование выводимого значения в строку вызовом функции ascii():

```
>>> print("%a" % ("строка"))
    '\u0441\u0442\u0440\u043e\u043a\u0430'
```

  - c — вывод символа с указанным кодом. Выведем числовое значение и соответствующий ему символ:

```
>>> for i in range(33, 127): print("%s => %c" % (i, i))
```

  - d и i — вывод целой части заданного числа:

```
>>> print("%d %d %d" % (10, 25.6, -80))
    10 25 -80
>>> print("%i %i %i" % (10, 25.6, -80))
    10 25 -80
```

  - o — вывод заданного целого числа в восьмеричном представлении:

```
>>> print("%o %o" % (0o77, 10))
    77 12
```

```
>>> print("%#o %#o" % (0o77, 10))
    0o77 0o12
```

При попытке вывести таким образом вещественное число возникнет ошибка;

- `x` — вывод заданного целого числа в шестнадцатеричном представлении в нижнем регистре (при попытке вывести вещественное число возникнет ошибка):

```
>>> print("%x %x" % (0xff, 10))
    ff a
>>> print("%#x %#x" % (0xff, 10))
    0xff 0xa
```

- `X` — вывод заданного целого числа в шестнадцатеричном представлении в верхнем регистре (при попытке вывести вещественное число возникнет ошибка):

```
>>> print("%X %X" % (0xff, 10))
    FF A A
>>> print("%#X %#X" % (0xff, 10))
    0XFF 0XA
```

- `f` и `F` — вывод заданного вещественного числа в десятичном представлении:

```
>>> print("%f %f %f" % (300, 18.65781452, -12.5))
    300.000000 18.657815 -12.500000
>>> print("%F %F %F" % (300, 18.65781452, -12.5))
    300.000000 18.657815 -12.500000
>>> print("%#.0F %.0F" % (300, 300))
    300. 300
```

- `e` — вывод заданного вещественного числа в экспоненциальной форме (буква `e` выводится в нижнем регистре):

```
>>> print("%e %e" % (3000, 18657.81452))
    3.000000e+03 1.865781e+04
```

- `E` — вывод заданного вещественного числа в экспоненциальной форме (буква `E` выводится в верхнем регистре):

```
>>> print("%E %E" % (3000, 18657.81452))
    3.000000E+03 1.865781E+04
```

- `g` — эквивалентно `f` или `e` (выбирается более короткая запись числа):

```
>>> print("%g %g %g" % (0.086578, 0.000086578, 1.865E-005))
    0.086578 8.6578e-05 1.865e-05
```

- `G` — эквивалентно `f` или `E` (выбирается более короткая запись числа):

```
>>> print("%G %G %G" % (0.086578, 0.000086578, 1.865E-005))
    0.086578 8.6578E-05 1.865E-05
```

Если внутри строки необходимо использовать символ процента, этот символ следует удвоить, иначе будет выведено сообщение об ошибке:

```
>>> print("% %s" % ("- это символ процента")) # Ошибка
    Traceback (most recent call last):
      File "<pyshell#55>", line 1, in <module>
        print("% %s" % ("- это символ процента"))
    TypeError: not all arguments converted during string formatting
```

```
>>> print("%% %s" % ("- это символ процента")) # Нормально
    % - это символ процента
```

Форматирование строк очень удобно использовать при передаче данных в шаблон веб-страницы. Для этого заполняем словарь данными и указываем его справа от символа %, а сам шаблон — слева. Продемонстрируем это на примере (листинг 6.1).

Листинг 6.1. Форматирование строк

```
html = """<html>
<head><title>%(title)s</title>
</head>
<body>
<h1>%(h1)s</h1>
<div>%(content)s</div>
</body>
</html>"""
arr = {"title":    "Это название документа",
       "h1":       "Это заголовок первого уровня",
       "content":  "Это основное содержание страницы"}
print(html % arr) # Подставляем значения и выводим шаблон
input()
```

Результат выполнения:

```
<html>
<head><title>Это название документа</title>
</head>
<body>
<h1>Это заголовок первого уровня</h1>
<div>Это основное содержание страницы</div>
</body>
</html>
```

Для форматирования строк также можно использовать следующие методы, поддерживаемые объектом строки:

- expandtabs([<Ширина поля>]) — заменяет каждый символ табуляции в текущей строке пробелами так, чтобы общая ширина фрагмента вместе с текстом, расположенным перед символом табуляции, была равна указанной величине. Если параметр не указан, то ширина поля предполагается равной 8 символам. Пример:

  ```
  >>> s = "1\t12\t123\t"
  >>> "'%s'" % s.expandtabs(4)
      "'1   12  123 '"
  ```

  В этом примере ширина задана равной четырем символам. Поэтому во фрагменте 1\t табуляция будет заменена тремя пробелами, во фрагменте 12\t — двумя пробелами, а во фрагменте 123\t — одним пробелом. Во всех трех фрагментах ширина будет равна четырем символам.

  Если перед символом табуляции нет текста или количество символов перед табуляцией равно указанной в вызове метода ширине, то табуляция заменяется указанным количеством пробелов:

```
>>> s = "\t"
>>> "'%s' — '%s'" % (s.expandtabs(), s.expandtabs(4))
    "'        ' — '    '"
>>> s = "1234\t"
>>> "'%s'" % s.expandtabs(4)
    "'1234    '"
```

Если количество символов перед табуляцией больше ширины, то табуляция заменяется пробелами таким образом, чтобы ширина фрагмента вместе с текстом делилась без остатка на указанную ширину:

```
>>> s = "12345\t123456\t1234567\t1234567890\t"
>>> "'%s'" % s.expandtabs(4)
    "'12345   123456  1234567 1234567890  '"
```

Таким образом, если количество символов перед табуляцией больше 4, но менее 8, то фрагмент дополняется пробелами до 8 символов. Если количество символов больше 8, но менее 12, то фрагмент дополняется пробелами до 12 символов и т. д. Все это справедливо при указании в качестве параметра числа 4;

- `center(<Ширина>[, <Символ>])` — выравнивает текущую строку по центру внутри поля указанной ширины с добавлением слева и справа символов из второго параметра (если он не указан, будут добавлены пробелы):

```
>>> s = "str"
>>> s.center(15), s.center(11, "-")
    ('      str      ', '----str----')
```

Теперь произведем выравнивание трех фрагментов шириной 15 символов: первого — по правому краю, второго — по левому, а третьего — по центру:

```
>>> s = "str"
>>> "'%15s' '%-15s' '%s'" % (s, s, s.center(15))
    "'            str' 'str            ' '      str      '"
```

Если количество символов в текущей строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью:

```
>>> s = "string"
>>> s.center(6), s.center(5)
    ('string', 'string')
```

- `ljust(<Ширина>[, <Символ>])` — выравнивает текущую строку по левому краю внутри поля указанной ширины с добавлением справа символов из второго параметра (если он не указан, будут добавлены пробелы). Если количество символов в текущей строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Примеры:

```
>>> s = "string"
>>> s.ljust(15), s.ljust(15, "-")
    ('string         ', 'string---------')
>>> s.ljust(6), s.ljust(5)
    ('string', 'string')
```

- `rjust(<Ширина>[, <Символ>])` — выравнивает текущую строку по правому краю внутри поля указанной ширины с добавлением слева символов из второго параметра (если он не

указан, будут добавлены пробелы). Если количество символов в текущей строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Пример:

```
>>> s = "string"
>>> s.rjust(15), s.rjust(15, "-")
    ('         string', '---------string')
>>> s.rjust(6), s.rjust(5)
    ('string', 'string')
```

- `zfill(<Ширина>)` — выравнивает текущую строку по правому краю внутри поля указанной ширины с добавлением слева нулей. Если количество символов в текущей строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Примеры:

```
>>> "5".zfill(20), "123456".zfill(5)
    ('00000000000000000005', '123456')
```

6.4. Метод *format()*

Для форматирования строк можно использовать метод `format()`. Он имеет следующий синтаксис вызова:

```
<Строка специального формата>.format(*args, **kwargs)
```

В качестве результата возвращается отформатированная строка.

В параметре `<Строка специального формата>` внутри символов фигурных скобок `{` и `}` указываются спецификаторы, имеющие следующий синтаксис:

```
{[<Поле>][!<Функция>][:<Формат>]}
```

Все символы, расположенные вне фигурных скобок, выводятся без преобразований. Если внутри строки необходимо использовать символы `{` и `}`, то эти символы следует удвоить, иначе возбуждается исключение `ValueError`, например:

```
>>> print("Символы {{ и }} — {0}".format("специальные"))
    Символы { и } — специальные
```

- `<Поле>` — параметр метода `format()`, в котором указано выводимое значение, в виде его порядкового номера (нумерация начинается с нуля) или имени. Допустимо комбинировать позиционные и именованные параметры, при этом именованные параметры следует указать последними. Примеры:

```
>>> "{0} - {1} - {2}".format(10, 12.3, "string")          # Индексы
    '10 - 12.3 - string'
>>> arr = [10, 12.3, "string"]
>>> "{0} - {1} - {2}".format(*arr)                        # Индексы
    '10 - 12.3 - string'
>>> "{model} - {color}".format(color="red", model="BMW") # Имена
    'BMW - red'
>>> d = {"color": "red", "model": "BMW"}
>>> "{model} - {color}".format(**d)                       # Имена
'BMW - red'
```

```
>>> "{color} - {0}".format(2015, color="red")          # Комбинация
'red - 2015'
```

В вызове метода `format()` можно указать список, словарь или объект. Для доступа к элементам списка по индексу внутри строки формата применяются квадратные скобки, а для доступа к элементам словаря или атрибутам объекта — точечная нотация. Пример:

```
>>> arr = [10, [12.3, "string"]]
>>> "{0[0]} - {0[1][0]} - {0[1][1]}".format(arr)      # Индексы
    '10 - 12.3 - string'
>>> "{arr[0]} - {arr[1][1]}".format(arr=arr)          # Индексы
    '10 - string'
>>> class Car: color, model = "red", "BMW"
...
>>> car = Car()
>>> "{0.model} - {0.color}".format(car)               # Атрибуты объекта
    'BMW - red'
```

Существует также краткая форма записи, при которой <Поле> не указывается. В этом случае скобки без заданного индекса нумеруются слева направо, начиная с нуля. Пример:

```
>>> "{} - {} - {} - {n}".format(1, 2, 3, n=4) # "{0} - {1} - {2} - {n}"
    '1 - 2 - 3 - 4'
>>> "{} - {} - {n} - {}".format(1, 2, 3, n=4) # "{0} - {1} - {n} - {2}"
    '1 - 2 - 4 - 3'
```

- <Функция> — обозначение функции, обрабатывающей значение перед вставкой в строку. Если указано обозначение `s`, то значение обрабатывается функцией `str()`, если `r`, то функцией `repr()`, а если `a`, то функцией `ascii()`. Если параметр не указан, для преобразования значения используется функция `str()`. Пример:

```
>>> print("{0!s}".format("строка"))                  # str()
    строка
>>> print("{0!r}".format("строка"))                  # repr()
    'строка'
>>> print("{0!a}".format("строка"))                  # ascii()
    '\u0441\u0442\u0440\u043e\u043a\u0430'
```

- <Формат> — должен иметь следующий синтаксис:

```
[[<Заполнитель>]<Выравнивание>][<Знак>][#][0][<Ширина>][,][_]↵
[.<Точность>][<Преобразование>]
```

  - <Ширина> — минимальная ширина поля. Если выводимое значение не помещается в указанную ширину, то ширина игнорируется и значение выводится полностью. Пример:

```
>>> "'{0:10}' '{1:3}'".format(3, "string")
"'         3' 'string'"
```

    Ширину поля можно передать в качестве параметра в методе `format()`. В этом случае вместо значения ширины внутри фигурных скобок указывается индекс соответствующего параметра. Пример:

```
>>> "'{0:{1}}'".format(3, 10) # 10 — это ширина поля
"'         3'"
```

- <Выравнивание>:
  - < — по левому краю;
  - > — по правому краю (поведение по умолчанию);
  - ^ — по центру поля.

  Пример:

```
>>> "'{0:<10}' '{1:>10}' '{2:^10}'".format(3, 3, 3)
    "'3          '          3' '    3     '"
```

  - = — знак числа выравнивается по левому краю, а число — по правому:

```
>>> "'{0:=10}' '{1:=10}'".format(-3, 3)
    "'-         3' '          3'"
```

    Как видно из приведенного примера, пространство между знаком и числом по умолчанию заполняется пробелами, а знак у положительного числа не указывается. Чтобы вместо пробелов пространство заполнялось нулями, необходимо указать нуль перед шириной поля, например:

```
>>> "'{0:=010}' '{1:=010}'".format(-3, 3)
    "'-000000003' '0000000003'"
```

    Начиная с Python 3.10, такое же поведение характерно и при выводе строк:

```
>>> "'{0:10}'".format("string")
    "'string    '"
>>> "'{0:010}'".format("string")
    "'string0000'"
```

- <Заполнитель> — символ, которым будет заполняться свободное пространство в поле (по умолчанию пробел):

```
>>> "'{0:0=10}' '{1:0=10}'".format(-3, 3)
    "'-000000003' '0000000003'"
>>> "'{0:*<10}' '{1:+>10}' '{2:.^10}'".format(3, 3, 3)
    "'3*********' '+++++++++3' '....3.....'"
```

- <Знак> — управляет выводом знака числа:
  - + — вывод знака как у отрицательных, так и у положительных чисел;
  - - — вывод знака только у отрицательных чисел (значение по умолчанию);
  - пробел — вывод пробела у положительных чисел и минуса у отрицательных.

  Примеры:

```
>>> "'{0:+}' '{1:+}' '{0:-}' '{1:-}'".format(3, -3)
    "'+3' '-3' '3' '-3'"
>>> "'{0: }' '{1: }'".format(3, -3)          # Пробел
    "' 3' '-3'"
```

- <Преобразование> — у целых чисел:
  - b — в двоичную систему счисления:

```
>>> "'{0:b}' '{0:#b}'".format(3)
    "'11' '0b11'"
```

- c — заданного числа в соответствующий символ:

```
>>> "'{0:c}'".format(167)
    "'§'"
```

- d — в десятичную систему счисления;
- n — аналогично опции d, но учитывает настройки локали. Например, выведем большое число с разделением тысячных разрядов пробелом (точнее, символом с кодом \xa0, который при выводе преобразуется в пробел):

```
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, 'Russian_Russia.1251')
    'Russian_Russia.1251'
>>> print("{0:n}".format(100000000).replace("\uffa0", " "))
    100 000 000
```

Также можно разделить тысячные разряды запятой, указав ее в строке формата:

```
>>> print("{0:,d}".format(100000000))
    100,000,000
```

или символами подчеркивания — таким же образом:

```
>>> print("{0:_d}".format(100000000))
    100_000_000
```

- o — в восьмеричную систему счисления:

```
>>> "'{0:d}' '{0:o}' '{0:#o}'".format(511)
    "'511' '777' '0o777'"
```

- x — в шестнадцатеричную систему счисления в нижнем регистре:

```
>>> "'{0:x}' '{0:#x}'".format(255)
    "'ff' '0xff'"
```

- X — в шестнадцатеричную систему счисления в верхнем регистре:

```
>>> "'{0:X}' '{0:#X}'".format(255)
    "'FF' '0XFF'"
```

- <Преобразование> — у вещественных чисел:
  - f и F — в десятичную систему счисления:

```
>>> "'{0:f}' '{1:f}' '{2:f}'".format(30, 18.6578145, -2.5)
    "'30.000000' '18.657815' '-2.500000'"
```

По умолчанию выводимое число имеет шесть знаков после запятой. Задать другое количество знаков после запятой можно в параметре <Точность>, например:

```
>>> "'{0:.7f}' '{1:.2f}'".format(18.6578145, -2.5)
    "'18.6578145' '-2.50'"
```

  - e — в экспоненциальную форму (буква e в нижнем регистре):

```
>>> "'{0:e}' '{1:e}'".format(3000, 18657.81452)
    "'3.000000e+03' '1.865781e+04'"
```

  - E — в экспоненциальную форму (буква E в верхнем регистре):

```
>>> "'{0:E}' '{1:E}'".format(3000, 18657.81452)
    "'3.000000E+03' '1.865781E+04'"
```

Здесь количество знаков после запятой по умолчанию также равно шести, но можно указать другое количество знаков в параметре <Точность>:

```
>>> "'{0:.2e}' '{1:.2E}'".format(3000, 18657.81452)
    "'3.00e+03' '1.87E+04'"
```

- `g` — эквивалентно `f` или `e` (выбирается более короткая запись числа):

```
>>> "'{0:g}' '{1:g}'".format(0.086578, 0.000086578)
    "'0.086578' '8.6578e-05'"
```

- `n` — аналогично опции `g`, но учитывает настройки локали;
- `G` — эквивалентно `f` или `E` (выбирается более короткая запись числа):

```
>>> "'{0:G}' '{1:G}'".format(0.086578, 0.000086578)
    "'0.086578' '8.6578E-05'"
```

- `%` — умножает число на 100 и добавляет символ процента в конец. Значение отображается в соответствии с опцией `f`. Пример:

```
>>> "'{0:%}' '{1:.4%}'".format(0.086578, 0.000086578)
    "'8.657800%' '0.0087%'"
```

6.4.1. Форматируемые строки

Форматируемая строка — это более компактная и удобная альтернатива методу `format()`.

Форматируемая строка предваряется буквой `f` или `F`. В нужных местах такой строки записываются команды на вставку в эти места значений, хранящихся в переменных, — точно так же, как и в строках специального формата, описанных ранее. Такие команды имеют следующий синтаксис:

```
{[<Переменная>][!<Функция>][:<Формат>]}
```

Параметр <Переменная> задает имя переменной, из которой будет извлечено вставляемое в строку значение. Вместо имени переменной можно записать выражение, вычисляющее значение, которое нужно вывести. Параметры <Функция> и <Формат> имеют то же назначение и записываются так же, как и в случае метода `format()`. Примеры:

```
>>> a = 10; b = 12.3; s = "string"
>>> f"{a} - {b} - {s}"                  # Простой вывод чисел и строк
    '10 - 12.3 - string'
>>> f"{a} - {b:5.2f} - {s}"             # Вывод с форматированием
    '10 - 12.30 - string'
>>> d = 3
>>> f"{a} - {b:5.{d}f} - {s}"           # В опциях можно использовать
>>>                                     # значения из переменных
    '10 - 12.300 - string'
>>> arr = [3, 4]
>>> f"{arr[0]} - {arr[1]}"              # Вывод элементов массива
    '3 - 4'
>>> f"{arr[0]} - {arr[1] * 2}"          # Вывод результатов вычисления выражений
    '3 - 8'
```

Начиная с Python 3.8, в форматируемой строке после переменной можно поставить символ равенства (=) — и в строку будет подставлено имя этой переменной, знак равенства и значение переменной:

```
>>> b = 12.3
>>> f"{b=}", f"{b = }"
    ('b=12.3', 'b = 12.3')
>>> f"Значение переменной {b = :5.2f}"
    'Значение переменной b = 12.30'
```

Этот программный инструмент может быть применен для вывода значений переменных при отладке программ.

6.5. Функции и методы для работы со строками

Основные функции для работы со строками:

- `str([<Значение>])` — преобразует заданное значение в строку, которую и возвращает. Если параметр не указан, возвращается пустая строка. Используется функцией `print()` для вывода объектов. Примеры:

```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
    ('', '[1, 2]', '(3, 4)', "{'x': 1}")
>>> print("строка1\nстрока2")
    строка1
    строка2
```

- `repr(<Значение>)` — возвращает строковое представление заданного значения. Используется при выводе данных интерпретатором, работающим в интерактивном режиме. Примеры:

```
>>> repr("Строка"), repr([1, 2, 3]), repr({"x": 5})
    ("'Строка'", '[1, 2, 3]', "{'x': 5}")
>>> repr("строка1\nстрока2")
    "'строка1\\nстрока2'"
```

- `ascii(<Значение>)` — возвращает строковое представление заданного значения, содержащее только символы из кодировки ASCII:

```
>>> ascii([1, 2, 3]), ascii({"x": 5})
    ('[1, 2, 3]', "{'x': 5}")
>>> ascii("строка")
    "'\\u0441\\u0442\\u0440\\u043e\\u043a\\u0430'"
```

- `len(<Строка>)` — возвращает длину заданной строки в символах:

```
>>> len("Python"), len("\r\n\t"), len(r"\r\n\t")
    (6, 3, 6)
>>> len("строка")
    6
```

Основные методы, поддерживаемые строками:

- `strip([<Символы>])` — удаляет указанные в параметре символы в начале и в конце текущей строки. Если параметр не задан, удаляются пробельные символы: пробел, перевод строки (`\n`), возврат каретки (`\r`), горизонтальная (`\t`) и вертикальная (`\v`) табуляция. Примеры:

```
>>> s1, s2 = "     str\n\r\v\t", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.strip(), s2.strip("tsr"))
    "'str — 'ok'"
```

- `lstrip([<Символы>])` — удаляет заданные символы в начале текущей строки (если параметр не указан, удаляет пробелы):

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.lstrip(), s2.lstrip("tsr"))
    "'str     ' — 'okstrstrstr'"
```

- `rstrip([<Символы>])` — удаляет заданные символы в конце текущей строки (если параметр не указан, удаляет пробелы):

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.rstrip(), s2.rstrip("tsr"))
    "'     str' — 'strstrstrok'"
```

- `split([<Разделитель>[, <Лимит>]])` — разделяет текущую строку на подстроки по указанному разделителю и добавляет эти подстроки в список, который возвращает в качестве результата. Если первый параметр не указан или имеет значение `None`, то в качестве разделителя используется символ пробела. Во втором параметре можно задать количество подстрок в результирующем списке, если он не указан или равен `-1`, в список попадут все подстроки. Если подстрок больше указанного количества, то список будет содержать еще один элемент — с остатком строки. Примеры:

```
>>> s = "word1 word2 word3"
>>> s.split(), s.split(None, 1)
    (['word1', 'word2', 'word3'], ['word1', 'word2 word3'])
>>> s = "word1\nword2\nword3"
>>> s.split("\n")
    ['word1', 'word2', 'word3']
```

Если в текущей строке содержатся несколько пробелов подряд и разделитель не указан, то пустые элементы не будут добавлены в список:

```
>>> s = "word1          word2 word3     "
>>> s.split()
    ['word1', 'word2', 'word3']
```

При использовании другого разделителя могут возникнуть пустые элементы:

```
>>> s = ",,word1,,word2,,word3,,"
>>> s.split(",")
    ['', '', 'word1', '', 'word2', '', 'word3', '', '']
>>> "1,,2,,3".split(",")
    ['1', '', '2', '', '3']
```

Если разделитель в текущей строке не найден, то список будет состоять из одного элемента, представляющего текущую строку:

```
>>> "word1 word2 word3".split("\n")
    ['word1 word2 word3']
```

- `rsplit([<Разделитель>[, <Лимит>]])` — аналогичен методу `split()`, но поиск символа-разделителя производится не слева направо, а наоборот — справа налево:

```
>>> s = "word1 word2 word3"
>>> s.rsplit(), s.rsplit(None, 1)
    (['word1', 'word2', 'word3'], ['word1 word2', 'word3'])
```

- `splitlines([False])` — разделяет текущую строку на подстроки по символу перевода строки (`\n`) и добавляет их в список. Символы новой строки включаются в результат, только если необязательный параметр имеет значение `True`. Если разделитель в текущей строке не найден, список будет содержать только один элемент — саму текущую строку. Примеры:

```
>>> "word1\nword2\nword3".splitlines()
    ['word1', 'word2', 'word3']
>>> "word1\nword2\nword3".splitlines(True)
    ['word1\n', 'word2\n', 'word3']
>>> "word1 word2 word3".splitlines()
    ['word1 word2 word3']
```

- `partition(<Разделитель>)` — находит первое вхождение указанного символа-разделителя в текущей строке и возвращает кортеж из трех элементов: фрагмента, расположенного перед разделителем, разделителя и фрагмента, расположенного после разделителя. Поиск производится слева направо. Если символ-разделитель не найден, то первый элемент кортежа будет содержать текущую строку, а остальные элементы останутся пустыми. Примеры:

```
>>> "word1 word2 word3".partition(" ")
    ('word1', ' ', 'word2 word3')
>>> "word1 word2 word3".partition("\n")
    ('word1 word2 word3', '', '')
```

- `rpartition(<Разделитель>)` — аналогичен методу `partition()`, но поиск символа-разделителя производится не слева направо, а наоборот — справа налево. Если символ-разделитель не найден, то первые два элемента кортежа окажутся пустыми, а третий элемент будет содержать текущую строку. Примеры:

```
>>> "word1 word2 word3".rpartition(" ")
    ('word1 word2', ' ', 'word3')
>>> "word1 word2 word3".rpartition("\n")
    ('', '', 'word1 word2 word3')
```

- `join()` — преобразует последовательность в строку. Элементы добавляются через указанный разделитель. Формат метода:

```
<Разделитель>.join(<Последовательность>)
```

В качестве примера преобразуем список и кортеж в строку:

```
>>> " => ".join(["word1", "word2", "word3"])
    'word1 => word2 => word3'
>>> " ".join(("word1", "word2", "word3"))
    'word1 word2 word3'
```

Элементы последовательностей должны быть строками, иначе возбуждается исключение `TypeError`:

```
>>> " ".join(("word1", "word2", 5))
    Traceback (most recent call last):
```

```
  File "<pyshell#48>", line 1, in <module>
    " ".join(("word1", "word2", 5))
TypeError: sequence item 2: expected str instance, int found
```

- `removeprefix(<Префикс>)` (начиная с Python 3.9) — удаляет у текущей строки указанный префикс и возвращает результат. Если текущая строка не содержит такого префикса, возвращает текущую строку. Пример:

```
>>> "ActivePython".removeprefix("Active")
    'Python'
>>> "CPython".removeprefix("Active")
    'CPython'
```

- `removesuffix(<Префикс>)` (начиная с Python 3.9) — удаляет у текущей строки указанный суффикс и возвращает результат. Если текущая строка не содержит такого суффикса, возвращает текущую строку. Пример:

```
>>> "program.py".removesuffix(".py")
    'program'
>>> "program.js".removesuffix(".py")
    'program.js'
```

Строки относятся к неизменяемым типам данных, поэтому, если попытаться изменить какой-либо символ в строке, возникнет ошибка. Однако можно преобразовать строку в список с помощью функции `list()`, изменить нужные символы по их индексам, а затем вызовом метода `join()` превратить список обратно в строку. Пример:

```
>>> s = "Python"
>>> arr = list(s); arr          # Преобразуем строку в список
    ['P', 'y', 't', 'h', 'o', 'n']
>>> arr[0] = "J"; arr           # Изменяем элемент по индексу
    ['J', 'y', 't', 'h', 'o', 'n']
>>> s = "".join(arr); s         # Преобразуем список в строку
    'Jython'
```

Если строка содержит лишь символы из кодировки ASCII, ее можно преобразовать в последовательность байтов `bytearray`:

```
>>> s = "Python"
>>> b = bytearray(s, "cp1251"); b
    bytearray(b'Python')
>>> b[0] = ord("J"); b
    bytearray(b'Jython')
>>> s = b.decode("cp1251"); s
    'Jython'
```

6.6. Настройка локали

Для установки *локали* (совокупности языковых настроек системы) служит функция `setlocale()` из модуля `locale`. Прежде чем использовать функцию, необходимо подключить модуль с помощью выражения

```
import locale
```

Функция `setlocale()` имеет следующий формат вызова:

```
setlocale(<Категория>[, <Локаль>]);
```

Параметр <Категория> может принимать следующие значения:

- `locale.LC_ALL` — устанавливает локаль для всех режимов;
- `locale.LC_COLLATE` — для сравнения строк;
- `locale.LC_CTYPE` — для перевода символов в нижний или верхний регистр;
- `locale.LC_MONETARY` — для отображения денежных единиц;
- `locale.LC_NUMERIC` — для форматирования чисел;
- `locale.LC_TIME` — для форматирования значений даты и времени.

Получить текущую локаль позволяет функция `getlocale([<Категория>])`. Если категория не указана, будет выдана локаль для всех категорий.

В качестве примера настроим локаль вначале на кодировку Windows-1251, потом на кодировку UTF-8, затем на кодировку по умолчанию, после чего выведем текущую локаль для всех категорий и только для `locale.LC_COLLATE`:

```
>>> import locale
>>> # Для кодировки windows-1251
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
    'Russian_Russia.1251'
>>> # Устанавливаем локаль по умолчанию
>>> locale.setlocale(locale.LC_ALL, "")
    'Russian_Russia.1251'
>>> # Получаем текущее значение локали для всех категорий
>>> locale.getlocale()
    ('Russian_Russia', '1251')
>>> # Получаем текущее значение категории locale.LC_COLLATE
>>> locale.getlocale(locale.LC_COLLATE)
    ('Russian_Russia', '1251')
```

Получить настройки локали позволяет функция `localeconv()` из модуля `locale`. Функция возвращает словарь с настройками. Результат ее выполнения для локали `Russian_Russia.1251` выглядит следующим образом:

```
>>> locale.localeconv()
    {'int_curr_symbol': 'RUB', 'currency_symbol': ' ',
     'mon_decimal_point': ',', 'mon_thousands_sep': '\xa0',
     'mon_grouping': [3, 0], 'positive_sign': '', 'negative_sign': '-',
     'int_frac_digits': 2, 'frac_digits': 2, 'p_cs_precedes': 0,
     'p_sep_by_space': 1, 'n_cs_precedes': 0, 'n_sep_by_space': 1,
     'p_sign_posn': 1, 'n_sign_posn': 1, 'decimal_point': ',',
     'thousands_sep': '\xa0', 'grouping': [3, 0]}
```

6.7. Изменение регистра символов

Для изменения регистра символов в строке предназначены следующие методы:

- `upper()` — приводит все символы в текущей строке к верхнему регистру:

```
>>> print("строка".upper())
    СТРОКА
```

- `lower()` — приводит все символы в текущей строке к нижнему регистру:

```
>>> print("СТРОКА".lower())
    строка
```

- `swapcase()` — заменяет в текущей строке все строчные символы соответствующими прописными буквами, а все прописные символы — строчными:

```
>>> print("СТРОКА строка".swapcase())
    строка СТРОКА
```

- `capitalize()` — делает первую букву текущей строки прописной:

```
>>> print("строка строка".capitalize())
    Строка строка
```

- `title()` — делает первую букву каждого слова в текущей строке прописной:

```
>>> s = "первая буква каждого слова станет прописной"
>>> print(s.title())
    Первая Буква Каждого Слова Станет Прописной
```

- `casefold()` — то же самое, что и `lower()`, но дополнительно преобразует все символы с диакритическими знаками и лигатуры в буквы стандартной латиницы. Обычно применяется для сравнения строк:

```
>>> "Python".casefold() == "python".casefold()
    True
>>> "grosse".casefold() == "große".casefold()
    True
```

6.8. Функции для работы с символами

Для работы с отдельными символами предназначены следующие функции:

- `chr(<Код символа>)` — возвращает символ по указанному коду:

```
>>> print(chr(177))
    ±
```

- `ord(<Символ>)` — возвращает код указанного символа:

```
>>> print(ord("±"))
    177
```

6.9. Поиск и замена в строке

Для поиска и замены в строке используются следующие методы:

- `find()` — ищет в текущей строке заданную подстроку. Возвращает номер позиции, с которой начинается вхождение подстроки в строку. Если подстрока не найдена, возвращает значение `-1`. Регистр символов учитывается. Формат метода:

```
<Строка>.find(<Подстрока>[, <Начало>[, <Конец>]])
```

  Если параметр `<Начало>` не указан, то поиск будет осуществляться с начала текущей строки. Если параметры `<Начало>` и `<Конец>` указаны, то извлекается срез строки:

```
<Строка>[<Начало>:<Конец>]
```

и поиск подстроки выполняется с начала полученного фрагмента:

```
>>> s = "пример пример Пример"
>>> s.find("при"), s.find("При"), s.find("тест")
   (0, 14, -1)
>>> s.find("при", 9), s.find("при", 0, 6), s.find("при", 7, 12)
   (-1, 0, 7)
```

- `index()` — аналогичен методу `find()`, но если подстрока не найдена, возбуждает исключение `ValueError`:

```
>>> s = "пример пример Пример"
>>> s.index("при"), s.index("при", 7, 12), s.index("При", 1)
    (0, 7, 14)
>>> s.index("тест")
    Traceback (most recent call last):
      File "<pyshell#24>", line 1, in <module>
        s.index("тест")
    ValueError: substring not found
```

- `rfind()` — ищет в текущей строке заданную подстроку. Возвращает позицию последнего вхождения подстроки в строку. Если подстрока не найдена, возвращается значение `-1`. Регистр символов учитывается. Формат метода:

```
<Строка>.rfind(<Подстрока>[, <Начало>[, <Конец>]])
```

Если параметр `<Начало>` не указан, то поиск будет производиться с конца текущей строки. Если параметры `<Начало>` и `<Конец>` указаны, то извлекается срез строки, и поиск подстроки производится с конца полученного фрагмента. Пример:

```
>>> s = "пример пример Пример Пример"
>>> s.rfind("при"), s.rfind("При"), s.rfind("тест")
    (7, 21, -1)
>>> s.rfind("При"), s.find("При", 12, 18)
    (21, 14)
```

- `rindex()` — аналогичен методу `rfind()`, но если подстрока не найдена, возбуждает исключение `ValueError`:

```
>>> s = "пример пример Пример Пример"
>>> s.rindex("при"), s.rindex("При"), s.rindex("при", 0, 6)
    (7, 21, 0)
>>> s.rindex("тест")
    Traceback (most recent call last):
      File "<pyshell#30>", line 1, in <module>
        s.rindex("тест")
    ValueError: substring not found
```

- `count()` — возвращает число вхождений заданной подстроки в текущую строку. Регистр символов учитывается. Формат метода:

```
<Строка>.count(<Подстрока>[, <Начало>[, <Конец>]])
```

Если параметр `<Начало>` не указан, то подсчет подстрок будет производиться во всей текущей строке. Если параметры `<Начало>` и `<Конец>` указаны, то извлекается срез строки и подсчет подстрок производится в полученном фрагменте. Примеры:

```
>>> s = "пример пример Пример Пример"
>>> s.count("при"), s.count("при", 6), s.count("При")
    (2, 1, 2)
>>> s.count("тест")
    0
```

- `startswith()` — проверяет, начинается ли текущая строка с указанной подстроки. Если начинается, возвращается значение `True`, в противном случае — `False`. Регистр символов учитывается. Формат метода:

```
<Строка>.startswith(<Подстрока>[, <Начало>[, <Конец>]])
```

Если параметр `<Начало>` не указан, сравнение будет производиться с началом текущей строки. Если параметры `<Начало>` и `<Конец>` указаны, то извлекается срез строки и сравнение производится с началом полученного фрагмента. Примеры:

```
>>> s = "пример пример Пример Пример"
>>> s.startswith("при"), s.startswith("При")
    (True, False)
>>> s.startswith("при", 6), s.startswith("При", 14)
    (False, True)
```

Параметр `<Подстрока>` может быть кортежем:

```
>>> s = "пример пример Пример Пример"
>>> s.startswith(("при", "При"))
    True
```

- `endswith()` — проверяет, заканчивается ли текущая строка указанной подстрокой. Если заканчивается, то возвращается значение `True`, в противном случае — `False`. Регистр символов учитывается. Формат метода:

```
<Строка>.endswith(<Подстрока>[, <Начало>[, <Конец>]])
```

Если параметр `<Начало>` не указан, то сравнение будет производиться с концом текущей строки. Если параметры `<Начало>` и `<Конец>` указаны, то извлекается срез строки и сравнение производится с концом полученного фрагмента. Примеры:

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith("ока"), s.endswith("ОКА")
    (False, True)
>>> s.endswith("ока", 0, 9)
    True
```

Параметр `<Подстрока>` может быть кортежем:

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith(("ока", "ОКА"))
    True
```

- `replace()` — производит замену всех вхождений заданной подстроки в текущей строке на другую подстроку и возвращает результат в виде новой строки. Регистр символов учитывается. Формат метода:

```
<Строка>.replace(<Заменяемая подстрока>, <Заменяющая подстрока>[,
                 <Максимальное количество замен>])
```

Если максимальное количество замен не указано, будет выполнена замена всех найденных подстрок. Примеры:

```
>>> s = "Привет, Петя"
>>> print(s.replace("Петя", "Вася"))
    Привет, Вася
>>> print(s.replace("петя", "вася")) # Зависит от регистра
    Привет, Петя
>>> s = "strstrstrstrstr"
>>> s.replace("str", ""), s.replace("str", "", 3)
    ('', 'strstr')
```

- `translate(<Таблица символов>)` — заменяет символы в текущей строке. Параметр `<Таблица символов>` должен быть словарем, ключами которого являются Unicode-коды заменяемых символов, а значениями — Unicode-коды заменяющих символов. Если вместо кода заменяющего символа указать `None`, то соответствующий заменяемый символ будет удален. Для примера удалим букву `П`, а также изменим регистр всех букв `р`:

```
>>> s = "Пример"
>>> d = {ord("П"): None, ord("р"): ord("Р")}
>>> d
    {1088: 1056, 1055: None}
>>> s.translate(d)
    'РимеР'
```

Упростить создание таблицы символов позволяет статический метод `maketrans()`. Формат метода:

```
str.maketrans(<X>[, <Y>[, <Z>]])
```

Если указан только первый параметр, то он должен быть словарем:

```
>>> t = str.maketrans({"а": "А", "о": "О", "с": None})
>>> t
    {1072: 'А', 1089: None, 1086: 'О'}
>>> "строка".translate(t)
    'трОкА'
```

Если указаны два первых параметра, то они должны быть строками одинаковой длины. В результате будет создан словарь с ключами из строки `<X>` и значениями из строки `<Y>`, расположенными в той же позиции. Изменим регистр некоторых символов:

```
>>> t = str.maketrans("абвгдежзи", "АБВГДЕЖЗИ")
>>> t
    {1072: 1040, 1073: 1041, 1074: 1042, 1075: 1043, 1076: 1044,
     1077: 1045, 1078: 1046, 1079: 1047, 1080: 1048}
>>> "абвгдежзи".translate(t)
    'АБВГДЕЖЗИ'
```

В третьем параметре можно дополнительно указать строку из символов, которым будет сопоставлено значение `None`. После выполнения метода `translate()` эти символы будут удалены из строки. Заменим все цифры на 0, а некоторые буквы удалим из строки:

```
>>> t = str.maketrans("123456789", "0" * 9, "str")
>>> t
    {116: None, 115: None, 114: None, 49: 48, 50: 48, 51: 48,
     52: 48, 53: 48, 54: 48, 55: 48, 56: 48, 57: 48}
>>> "str123456789str".translate(t)
    '000000000'
```

6.10. Проверка содержимого строки

Для проверки содержимого строки предназначены следующие методы:

- `isalnum()` — возвращает `True`, если текущая строка содержит только буквы и (или) цифры, в противном случае — `False`. Если строка пустая, возвращается `False`. Примеры:

```
>>> "0123".isalnum(), "123abc".isalnum(), "abc123".isalnum()
    (True, True, True)
>>> "строка".isalnum()
    True
>>> "".isalnum(), "123 abc".isalnum(), "abc, 123.".isalnum()
    (False, False, False)
```

- `isalpha()` — возвращает `True`, если текущая строка содержит только буквы, в противном случае — `False`. Если строка пустая, возвращается `False`. Примеры:

```
>>> "string".isalpha(), "строка".isalpha(), "".isalpha()
    (True, True, False)
>>> "123abc".isalpha(), "str str".isalpha(), "st,st".isalpha()
    (False, False, False)
```

- `isascii()` (начиная с Python 3.7) — возвращает `True`, если текущая строка содержит лишь символы из кодировки ASCII, в противном случае — `False`. Если строка пустая, возвращается `True`. Примеры:

```
>>> "123abc".isascii(), "строка123".isascii(), "".isascii()
    (True, False, True)
```

- `isdigit()` — возвращает `True`, если текущая строка содержит только обычные цифры, в противном случае — `False`. Если строка пустая, возвращается `False`. Примеры:

```
>>> "0123".isdigit(), "123abc".isdigit(), "abc123".isdigit()
    (True, False, False)
```

- `isdecimal()` — возвращает `True`, если текущая строка содержит только символы цифр (обычных, надстрочных и подстрочных), в противном случае — `False`. Если строка пустая, возвращается `False`. Примеры:

```
>>> "123".isdecimal(), "123стр".isdecimal()
    (True, False)
```

- `isnumeric()` — возвращает `True`, если текущая строка содержит только символы чисел (обычные и римские цифры, дробные числа), в противном случае — `False`. Если строка пустая, возвращается `False`. Примеры:

```
>>> "\u2155".isnumeric(), "\u2155".isdigit()
    (True, False)
>>> print("\u2155") # Выведет символ "1/5"
```

- `isupper()` — возвращает `True`, если текущая строка содержит буквы только верхнего регистра, в противном случае — `False`. Если строка пустая или не содержит букв, возвращается `False`. Примеры:

```
>>> "STRING".isupper(), "СТРОКА".isupper(), "".isupper()
    (True, True, False)
```

```
>>> "STRINGl".isupper(), "СТРОКА, 123".isupper(), "123".isupper()
    (True, True, False)
>>> "string".isupper(), "STRing".isupper()
    (False, False)
```

- `islower()` — возвращает `True`, если текущая строка содержит буквы только нижнего регистра, в противном случае — `False`. Если строка пустая или не содержит букв, возвращается `False`. Примеры:

```
>>> "srting".islower(), "строка".islower(), "".islower()
    (True, True, False)
>>> "stringl".islower(), "str, 123".islower(), "123".islower()
    (True, True, False)
>>> "STRING".islower(), "Строка".islower()
    (False, False)
```

- `istitle()` — возвращает `True`, если все слова в текущей строке начинаются с заглавной буквы, в противном случае — `False`. Если строка пустая или не содержит букв, возвращается `False`. Примеры:

```
>>> "Str Str".istitle(), "Стр Стр".istitle()
    (True, True)
>>> "Str Str 123".istitle(), "Стр Стр 123".istitle()
    (True, True)
>>> "Str str".istitle(), "Стр стр".istitle()
    (False, False)
>>> "".istitle(), "123".istitle()
    (False, False)
```

- `isprintable()` — возвращает `True`, если текущая строка содержит только печатаемые символы, в противном случае — `False`. Пробел относится к печатаемым символам. Если строка пустая, возвращается `True`. Примеры:

```
>>> "123".isprintable()
    True
>>> "PHP Python".isprintable()
    True
>>> "\n".isprintable()
    False
```

- `isspace()` — возвращает `True`, если текущая строка содержит только пробельные символы, в противном случае — `False`. Если строка пустая, возвращается `False`. Пример:

```
>>> "".isspace(), " \n\r\t".isspace(), "str str".isspace()
    (False, True, False)
```

- `isidentifier()` — возвращает `True`, если текущая строка представляет собой допустимое с точки зрения Python имя переменной, функции или класса, в противном случае — `False`:

```
>>> "s".isidentifier()
    True
>>> "func".isidentifier()
    True
```

```
>>> "123func".isidentifier()
    False
```

Следует иметь в виду, что метод `isidentifier()` лишь проверяет, удовлетворяет ли заданное имя правилам языка. Он не проверяет, совпадает ли это имя с каким-либо ключевым словом Python. Для выполнения такой проверки надлежит применять функцию `iskeyword()` из модуля `keyword`, которая возвращает `True`, если переданная ей строка совпадает с одним из ключевых слов, например:

```
>>> keyword.iskeyword("else")
    True
>>> keyword.iskeyword("elsewhere")
    False
```

Переделаем нашу программу суммирования произвольного количества целых чисел, введенных пользователем (см. листинг 4.13), таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой. Кроме того, предусмотрим возможность ввода отрицательных целых чисел (листинг 6.2).

Листинг 6.2. Суммирование неопределенного количества чисел

```
print("Введите слово 'stop' для получения результата")
summa = 0
while (x := input("Введите число: ")) != "stop":
    if x == "":
        print("Вы не ввели значение!")
        continue
    if x[0] == "-": # Если первым символом является минус
        if not x[1:].isdigit(): # Если фрагмент не состоит из цифр
            print("Необходимо ввести число, а не строку!")
            continue
    else: # Если минуса нет, то проверяем всю строку
        if not x.isdigit(): # Если строка не состоит из цифр
           print("Необходимо ввести число, а не строку!")
           continue
    x = int(x) # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода значений и получения результата выглядит так (значения, введенные пользователем, выделены полужирным шрифтом):

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число:
Вы не ввели значение!
Введите число: str
Необходимо ввести число, а не строку!
Введите число: -5
Введите число: -str
Необходимо ввести число, а не строку!
```

```
Введите число: stop
Сумма чисел равна: 5
```

6.11. Двоичные данные типа *bytes*

Тип данных `bytes` представляет неизменяемую последовательность байтов (чисел от 0 до 255).

Создать значение типа `bytes` можно следующими способами:

- с помощью функции `bytes([<Строка>, <Кодировка>[, <Обработка ошибок>]])`. Если параметры не указаны, то возвращается пустая последовательность байтов. Чтобы преобразовать заданную строку в значение типа `bytes`, необходимо передать минимум два первых параметра. Если указан только первый параметр, то возбуждается исключение `TypeError`. Примеры:

```
>>> bytes()
    b''
>>> bytes("строка", "cp1251")
    b'\xfl\xf2\xf0\xee\xea\xe0'
>>> bytes("строка")
    Traceback (most recent call last):
      File "<pyshell#33>", line 1, in <module>
        bytes("строка")
    TypeError: string argument without an encoding
```

  В третьем параметре могут быть указаны значения `"strict"` (при ошибке возбуждается исключение `UnicodeEncodeError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется знаком вопроса) или `"ignore"` (неизвестные символы игнорируются):

```
>>> bytes("string\uFFFD", "cp1251", "strict")
    Traceback (most recent call last):
      File "<pyshell#35>", line 1, in <module>
        bytes("string\uFFFD", "cp1251", "strict")
      File "C:\Python310\lib\encodings\cp1251.py", line 12, in encode
        return codecs.charmap_encode(input,errors,encoding_table)
    UnicodeEncodeError: 'charmap' codec can't encode character
    '\ufffd' in position 6: character maps to <undefined>
>>> bytes("string\uFFFD", "cp1251", "replace")
    b'string?'
>>> bytes("string\uFFFD", "cp1251", "ignore")
    b'string'
```

- с помощью строкового метода `encode([encoding="utf-8"][, errors="strict"])`. В параметре `encoding` задается кодировка (по умолчанию UTF-8). В параметре `errors` могут быть указаны значения `"strict"` (значение по умолчанию), `"replace"`, `"ignore"`, `"xmlcharrefreplace"` или `"backslashreplace"`. Примеры:

```
>>> "строка".encode()
    b'\xd1\x81\xdl\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
>>> "строка".encode(encoding="cp1251")
    b'\xfl\xf2\xf0\xee\xea\xe0'
```

```
>>> "строка\uFFFD".encode(encoding="cp1251",
...                       errors="xmlcharrefreplace")
    b'\xf1\xf2\xf0\xee\xea\xe0&#65533;'
>>> "строка\uFFFD".encode(encoding="cp1251",
...                       errors="backslashreplace")
    b'\xf1\xf2\xf0\xee\xea\xe0\\ufffd'
```

- указав букву `b` (регистр не имеет значения) перед строкой в апострофах, кавычках, тройных апострофах или тройных кавычках. В строке могут присутствовать только символы из кодировки ASCII, все остальные символы должны быть представлены специальными последовательностями. Примеры:

```
>>> b"string", b'string', b"""string""", b'''string'''
    (b'string', b'string', b'string', b'string')
>>> b"строка"
    SyntaxError: bytes can only contain ASCII literal characters.
>>> b"\xf1\xf2\xf0\xee\xea\xe0"
    b'\xf1\xf2\xf0\xee\xea\xe0'
```

- с помощью функции `bytes(<Последовательность>)`, которая преобразует заданную последовательность целых чисел от 0 до 255 в объект типа `bytes`. Если какое-либо число из последовательности не попадает в диапазон, возбуждается исключение `ValueError`. Примеры:

```
>>> b = bytes([225, 226, 224, 174, 170, 160])
>>> b
    b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
    'строка'
```

- с помощью функции `bytes(<Количество>)`, которая возвращает последовательность из заданного количества нулевых элементов:

```
>>> bytes(10)
    b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

- с помощью метода `bytes.fromhex(<Строка>)`. Заданная строка в этом случае должна содержать только шестнадцатеричные числа. Начиная с Python 3.7, числа в строке можно разделять пробелами, которые будут проигнорированы. Пример:

```
>>> b = bytes.fromhex(" e1 e2e0ae aaa0 ")
>>> b
    b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
    'строка'
```

Как и все последовательности, значения типа `bytes` поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повторение и проверку на вхождение:

```
>>> b = bytes("string", "cp1251")
>>> b
    b'string'
>>> b[0]                          # Обращение по индексу
    115
>>> b[1:3]                        # Получение среза
    b'tr'
```

```
>>> b + b"123"                    # Конкатенация
    b'string123'
>>> b * 3                         # Повторение
    b'stringstringstring'
>>> 115 in b, b"tr" in b, b"as" in b
    (True, True, False)
```

Как видно из примера, при выводе значения целиком, а также при извлечении среза производится попытка отображения символов. Однако доступ по индексу возвращает целое число, а не символ. Если преобразовать объект в список, то мы получим последовательность целых чисел:

```
>>> list(bytes("string", "cp1251"))
    [115, 116, 114, 105, 110, 103]
```

Тип `bytes` относится к неизменяемым типам. Это означает, что можно получить значение по индексу, но изменить его нельзя:

```
>>> b = bytes("string", "cp1251")
>>> b[0] = 168
    Traceback (most recent call last):
      File "<pyshell#76>", line 1, in <module>
        b[0] = 168
    TypeError: 'bytes' object does not support item assignment
```

Объекты типа `bytes` поддерживают большинство строковых методов, рассмотренных в предыдущих разделах, за исключением `encode()`, `isidentifier()`, `isprintable()`, `isnumeric()`, `isdecimal()`, `format_map()`, `format()` и операции форматирования. При использовании этих методов следует учитывать, что в параметрах нужно указывать объекты типа `bytes`, а не строки, например:

```
>>> b = bytes("string", "cp1251")
>>> b.replace(b"s", b"S")
    b'String'
```

Смешивать строки и значения типа `bytes` в выражениях нельзя. Необходимо явно преобразовать объекты к одному типу, и лишь затем производить операцию, например:

```
>>> b"string" + "string"
    Traceback (most recent call last):
      File "<pyshell#79>", line 1, in <module>
        b"string" + "string"
    TypeError: can't concat bytes to str
>>> b"string" + "string".encode("ascii")
    b'stringstring'
```

Значение типа `bytes` может содержать как однобайтовые, так и многобайтовые символы. При использовании многобайтовых символов некоторые функции могут работать не так, как предполагалось, — например, функция `len()` вернет количество байтов, а не символов:

```
>>> len("строка")
    6
>>> len(bytes("строка", "cp1251"))
    6
>>> len(bytes("строка", "utf-8"))
    12
```

Преобразовать значение типа bytes в строку позволяет метод decode(). Метод имеет следующий формат:

```
decode([encoding="utf-8"][, errors="strict"])
```

Параметр encoding задает кодировку символов (по умолчанию UTF-8) в текущей последовательности, а параметр errors — способ обработки ошибок при преобразовании: "strict" (значение по умолчанию), "replace" или "ignore". Пример преобразования:

```
>>> b = bytes("строка", "cp1251")
>>> b.decode(encoding="cp1251")
    'строка'
```

Для преобразования также можно воспользоваться функцией str():

```
>>> b = bytes("строка", "cp1251")
>>> str(b, "cp1251")
    'строка'
```

Чтобы изменить кодировку данных, сначала следует преобразовать значение типа bytes в строку, а затем произвести обратное преобразование, указав нужную кодировку. Преобразуем данные из кодировки Windows-1251 в кодировку KOI8-R, а затем обратно:

```
>>> w = bytes("Строка", "cp1251") # Данные в кодировке windows-1251
>>> k = w.decode("cp1251").encode("koi8-r")
>>> k, str(k, "koi8-r")           # Данные в кодировке KOI8-R
    (b'\xf3\xd4\xd2\xcf\xcb\xc1', 'Строка')
>>> w = k.decode("koi8-r").encode("cp1251")
>>> w, str(w, "cp1251")           # Данные в кодировке windows-1251
    (b'\xd1\xf2\xf0\xee\xea\xe0', 'Строка')
```

Последовательности типа bytes можно форматировать с применением описанного в *разд. 6.3* оператора %:

```
>>> b"%i - %i - %f" % (10, 20, 30)
    b'10 - 20 - 30.000000'
```

Однако тип преобразования s (т. е. вывод в виде Unicode-строки) в этом случае не поддерживается, и его использование приведет к возбуждению исключения TypeError:

```
>>> b"%s - %s - %s" % (10, 20, 30)
    Traceback (most recent call last):
      File "<pyshell#50>", line 1, in <module>
        b"%s - %s - %s" % (10, 20, 30)
    TypeError: %b requires a bytes-like object, or an object that implements
    __bytes__, not 'int'
```

Метод hex() возвращает строку с шестнадцатеричным представлением текущей последовательности типа bytes. Формат метода:

```
hex([<Разделитель>[, <Число байтов в группе>]])
```

Поддержка параметров у этого метода появилась в Python 3.8. Параметр <Разделитель> задает разделитель, который будет вставляться между отдельными группами байтов в выдаваемой строке (если он не указан, никакой разделитель не вставляется). Если параметру <Число байтов в группе> присвоено положительное число, отсчет байтов ведется справа, если отрицательное — слева, если параметр вообще не задан, число байтов в группе принимается равным 1. Примеры:

```
>>> b"string".hex()
    '737472696e67'
>>> b"string".hex('_'), b"string".hex('_', 4), b"string".hex('_', -4)
    ('73_74_72_69_6e_67', '7374_72696e67', '73747269_6e67')
```

6.12. Двоичные данные типа *bytearray*

Тип данных `bytearray` аналогичен типу `bytes`, только является изменяемым.

Создать значение типа `bytearray` можно следующими способами:

- с помощью функции `bytearray([<Строка>, <Кодировка>[, <Обработка ошибок>]])`. Если параметры не указаны, то возвращается пустая последовательность байтов. Чтобы преобразовать заданную строку в значение типа `bytearray`, необходимо передать как минимум два первых параметра. Если указан только первый параметр, то возбуждается исключение `TypeError`. Примеры:

```
>>> bytearray()
    bytearray(b'')
>>> bytearray("строка", "cp1251")
    bytearray(b'\xf1\xf2\xf0\xee\xea\xe0')
>>> bytearray("строка")
    Traceback (most recent call last):
      File "<pyshell#2>", line 1, in <module>
        bytearray("строка")
    TypeError: string argument without an encoding
```

  В третьем параметре могут быть указаны значения `"strict"` (при ошибке возбуждается исключение `UnicodeEncodeError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется знаком вопроса) или `"ignore"` (неизвестные символы игнорируются):

```
>>> bytearray("string\uFFFD", "cp1251", "strict")
    Traceback (most recent call last):
      File "<pyshell#5>", line 1, in <module>
        bytearray("string\uFFFD", "cp1251", "strict")
      File "C:\Python310\lib\encodings\cp1251.py", line 12, in encode
        return codecs.charmap_encode(input,errors,encoding_table)
    UnicodeEncodeError: 'charmap' codec can't encode character
    '\ufffd' in position 6: character maps to <undefined>
>>> bytearray("string\uFFFD", "cp1251", "replace")
    bytearray(b'string?')
>>> bytearray("string\uFFFD", "cp1251", "ignore")
    bytearray(b'string')
```

- с помощью функции `bytearray(<Последовательность>)`, которая преобразует заданную последовательность целых чисел от 0 до 255 в значение типа `bytearray`. Если какое-либо число из последовательности не попадает в диапазон, возбуждается исключение `ValueError`. Примеры:

```
>>> b = bytearray([225, 226, 224, 174, 170, 160])
>>> b
    bytearray(b'\xel\xe2\xe0\xae\xaa\xa0')
```

```
>>> str(b, "cp866")
    'строка'
```

- с помощью функции `bytearray(<Количество>)`, которая вернет последовательность из указанного количества нулевых элементов:

```
>>> bytearray(5)
    bytearray(b'\x00\x00\x00\x00\x00')
```

- с помощью метода `bytearray.fromhex(<Строка>)`. Указанная строка должна содержать только шестнадцатеричные числа и пробелы, которые будут проигнорированы. Примеры:

```
>>> b = bytearray.fromhex(" e1 e2e0ae aaa0 ")
>>> b
    bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
    'строка'
```

Любой элемент в последовательности типа `bytearray` можно изменить. При этом важно помнить, что присваиваемое ему новое значение должно быть целым числом в диапазоне от 0 до 255. Примеры:

```
>>> b = bytearray("Python", "ascii")
>>> b[0]                              # Можем получить значение
    80
>>> b[0] = b"J"[0]                    # Можем изменить значение
>>> b
    bytearray(b'Jython')
```

Для изменения значения типа `bytearray` также можно использовать следующие методы:

- `append(<Число>)` — добавляет заданное число в конец текущей последовательности:

```
>>> b = bytearray("string", "ascii")
>>> b.append(b"1"[0]); b
    bytearray(b'string1')
```

- `extend(<Последовательность>)` — добавляет элементы заданной последовательности в конец текущей:

```
>>> b = bytearray("string", "ascii")
>>> b.extend(b"123"); b
    bytearray(b'string123')
```

Добавить несколько элементов также можно с помощью операторов + и +=:

```
>>> b = bytearray("string", "ascii")
>>> b + b"123"                  # Возвращает новый объект
    bytearray(b'string123')
>>> b += b"456"; b              # Изменяет текущий объект
    bytearray(b'string456')
```

Кроме того, можно воспользоваться операцией присваивания значения срезу:

```
>>> b = bytearray("string", "ascii")
>>> b[len(b):] = b"123"         # Добавляем элементы в последовательность
```

```
>>> b
    bytearray(b'string123')
```

- `insert(<Индекс>, <Число>)` — добавляет заданное число в текущую последовательность по указанному индексу. Остальные элементы смещаются. Добавим элемент в начало последовательности:

```
>>> b = bytearray("string", "ascii")
>>> b.insert(0, b"1"[0]); b
    bytearray(b'1string')
```

Чтобы добавить несколько элементов, можно воспользоваться операцией присваивания значения срезу. Добавим несколько элементов в начало объекта:

```
>>> b = bytearray("string", "ascii")
>>> b[:0] = b"123"; b
    bytearray(b'123string')
```

- `pop([<Индекс>])` — удаляет из текущей последовательности элемент, расположенный по указанному индексу, и возвращает его. Если индекс не указан, удаляет и возвращает последний элемент. Примеры:

```
>>> b = bytearray("string", "ascii")
>>> b.pop()                # Удаляем последний элемент
    103
>>> b
    bytearray(b'strin')
>>> b.pop(0)               # Удаляем первый элемент
    115
>>> b
    bytearray(b'trin')
```

Удалить элемент последовательности также позволяет оператор `del`:

```
>>> b = bytearray("string", "ascii")
>>> del b[5]              # Удаляем последний элемент
>>> b
    bytearray(b'strin')
>>> del b[:2]             # Удаляем первый и второй элементы
>>> b
    bytearray(b'rin')
```

- `remove(<Число>)` — удаляет из текущей последовательности первый встреченный элемент, хранящий указанное число. Если элемент не найден, возбуждается исключение `ValueError`. Пример:

```
>>> b = bytearray("strstr", "ascii")
>>> b.remove(b"s"[0])     # Удаляет только первый элемент
>>> b
    bytearray(b'trstr')
```

- `reverse()` — изменяет порядок следования элементов текущей последовательности на противоположный:

```
>>> b = bytearray("string", "ascii")
>>> b.reverse(); b
    bytearray(b'gnirts')
```

Преобразовать значение типа bytearray в строку позволяет метод decode(). Метод имеет следующий формат:

```
decode([encoding="utf-8"][, errors="strict"])
```

Параметр encoding задает кодировку символов (по умолчанию UTF-8) в текущей последовательности, а параметр errors — способ обработки ошибок при преобразовании: "strict" (значение по умолчанию), "replace" или "ignore". Пример:

```
>>> b = bytearray("строка", "cp1251")
>>> b.decode(encoding="cp1251"), b.decode("cp1251")
    ('строка', 'строка')
```

Для преобразования также можно воспользоваться функцией str():

```
>>> b = bytearray("строка", "cp1251")
>>> str(b, "cp1251")
    'строка'
```

6.13. Сериализация и десериализация значений

Сериализация — это преобразование какого-либо значения в последовательность байтов, обычно типа bytes (что может понадобиться, например, для записи значения в файл). *Десериализация* — это обратное преобразование значения из последовательности байтов в исходный вид.

Инструменты для сериализации и десериализации находятся в модуле pickle, который предварительно следует подключить с помощью инструкции:

```
import pickle
```

Для преобразования предназначены две функции:

- dumps(<Значение>[, protocol=None]) — производит сериализацию заданного значения и возвращает результирующую последовательность байтов. Формат, в котором представляется сериализованное значение, зависит от указанного во втором параметре протокола, который задается в виде числа от 0 до значения константы pickle.HIGHEST_PROTOCOL. Если второй параметр не указан, будет использован протокол 4 (константа pickle.DEFAULT_PROTOCOL, до Python 3.8 этот параметр имел значение по умолчанию 3). Пример преобразования списка и кортежа:

```
>>> import pickle
>>> obj1 = [1, 2, 3, 4, 5]      # Список
>>> obj2 = (6, 7, 8, 9, 10)     # Кортеж
>>> pickle.dumps(obj1)
    b'\x80\x04\x95\x0f\x00\x00\x00\x00\x00\x00\x00]\x94(K\x01K\x02K\↵
    x03K\x04K\x05e.'
>>> pickle.dumps(obj2)
    b'\x80\x04\x95\x0e\x00\x00\x00\x00\x00\x00\x00(K\x06K\x07K\x08K\↵
    tK\nt\x94.'
```

- loads(<Последовательность байтов>) — производит десериализацию значения из заданной последовательность байтов. Пример восстановления списка и кортежа:

```
>>> pickle.loads(b'\x80\x04\x95\x0f\x00\x00\x00\x00\x00\x00\x00]\x94(K\↵
x01K\x02K\x03K\x04K\x05e.')
    [1, 2, 3, 4, 5]
```

```
>>> pickle.loads(b'\x80\x04\x95\x0e\x00\x00\x00\x00\x00\x00\x00(K\x06K\↵
x07K\x08K\tK\nt\x94.')
    (6, 7, 8, 9, 10)
```

6.14. Хеширование значений

Для вычисления хешей на основе последовательностей байтов предназначен модуль `hashlib`. Предварительно следует подключить его с помощью инструкции:

```
import hashlib
```

Модуль предоставляет следующие функции: `md5()`, `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()` и `shake_256()`. В качестве необязательного параметра функциям можно передать последовательность байтов, на основе которой следует вычислить хеш:

```
>>> import hashlib
>>> h1 = hashlib.sha512(b"password")
```

Указать хешируемую последовательность байтов также можно с помощью метода `update()`. В этом случае заданная последовательность присоединяется к уже имеющейся. Пример:

```
>>> h2 = hashlib.sha512(b"pass")
>>> h2.update(b"word")
```

Получить хеш в виде последовательности байтов типа `bytes` и строки позволяют методы соответственно `digest()` и `hexdigest()`:

```
>>> d1 = h1.digest()
>>> d1
 b'\xb1\t\xf3\xbb\xbc$N\xb8$A\x91~\xd0ma\x8b\x90\x08\xdd\t\xb3\xbe\↵
  xfd\x1b^\x079Lpj\x8b\xb9\x80\xb1\xd7x^Yv\xec\x04\x9bF\xdf_\x13&\xafZ.\↵
  xa6\xd1\x03\xfd\x07\xc9S\x85\xff\xab\x0c\xac\xbc\x86'
>>> h1.hexdigest()
 'b109f3bbbc244eb82441917ed06d618b9008dd09b3befd1b5e07394c706a8bb980b1d7↵
  785e5976ec049b46df5f1326af5a2ea6d103fd07c95385ffab0cacbc86'
>>> d1 == h2.digest()    # Имитируем сверку хеша пароля, введенного
>>>                      # пользователем, с хешем сохраненного пароля
    True
>>>                      # Пользователь ввел верный пароль!
```

Свойство `digest_size` хранит длину сгенерированного хеша в байтах:

```
>>> h1.digest_size
    64
```

Поддерживаются еще две функции, вычисляющие более устойчивые к взлому хеши:

- `blake2s(<Значение>[, digest_size=32][, salt=b""])` — хеширует заданное значение по алгоритму BLAKE2s, оптимизированному для 32-разрядных систем. Второй параметр задает требуемый размер хеша в виде числа от 1 до 32, третий — «соль» в виде последовательности типа `bytes`, которая может иметь в длину не более 8 байтов. Возвращает хеш в виде последовательности типа `bytes`. Пример:

```
>>> h = hashlib.blake2s(b"string", digest_size=16, salt=b"saltsalt")
>>> h.digest()
    b'\x961\xe0\xfa\xb4\xe7Bw\x11\xf7D\xc2\xa4\xcf\x06\xf7'
```

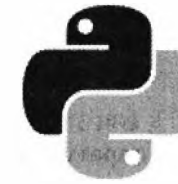
- `blake2b(<Значение>[, digest_size=64][, salt=b""])` — хеширует заданное значение по алгоритму BLAKE2b, оптимизированному для 64-разрядных систем. Второй параметр задает требуемый размер хеша в виде числа от 1 до 64, третий — «соль» в виде последовательности типа `bytes`, которая может иметь в длину не более 16 байтов. Возвращает хеш в виде последовательности типа `bytes`. Пример:

```
>>> h = hashlib.blake2b(b"string", digest_size=48, salt=b"saltsaltsalt")
>>> h.digest()
    b'\x0e\xcf\xb9\xc8G;q\xbaV\xbdV\x16\xd4@/J\x97W\x0c\xc4\xc5{\xd4\xb6\↳
      x12\x01z\x9f\xdd\xf6\xf1\x03o\x97&v\xfd\xa6\x90\x81\xc4T\xb8z\xaf\↳
      xc3\x9a\xd9'
```

Примечание

Функции `blake2b()` и `blake2s()` поддерживают большое количество параметров, которые применяются только в специфических случаях. Полное описание этих функций можно найти в документации по Python.

Функция хеширования `pbkdf2_hmac()`, поддерживавшаяся в предыдущих версиях Python, начиная с Python 3.10, объявлена устаревшей и не рекомендуется к использованию.



ГЛАВА 7

Регулярные выражения

Регулярное выражение — это шаблон, применяемый для поиска совпадающих с ним фрагментов в строках. Регулярное выражение состоит как из обычных знаков, так и всевозможных специальных символов: метасимволов, квантификаторов и др.

В Python использовать регулярные выражения позволяет модуль `re`, который необходимо предварительно подключить с помощью инструкции:

```
import re
```

7.1. Синтаксис регулярных выражений

Создать откомпилированное и готовое к использованию регулярное выражение позволяет функция `compile()`. Формат ее вызова:

```
re.compile(<Шаблон регулярного выражения>[, <Флаги>])
```

Шаблон создаваемого регулярного выражения указывается в виде строки или последовательности байтов типа `bytes` или `bytearray`. Флаги управляют особенностями обработки регулярного выражения. Можно указать как один флаг, так и их комбинацию через оператор `|`. Все поддерживаемые флаги также объявлены в модуле `re`:

- `I` или `IGNORECASE` — регистр символов не учитывается:

```
>>> import re
>>> p = re.compile(r"^[а-яе]+$", re.I | re.U)
>>> print("Найдено" if p.search("АБВГДЕЕ") else "Нет")
    Найдено
>>> p = re.compile(r"^[а-яе]+$", re.U)
>>> print("Найдено" if p.search("АБВГДЕЕ") else "Нет")
    Нет
```

- `M` или `MULTILINE` — метасимвол `^` обозначает начало каждой подстроки в строке, в которой выполняется поиск (сразу после символа перевода строки `\n`, который используется для разбиения строки на подстроки), а символ `$` — конец каждой подстроки (непосредственно перед символом перевода строки). Без этого флага метасимвол `^` обозначает начало самой строки, в которой выполняется поиск, а символ `$` — ее конец. Примеры:

```
>>> p = re.compile(r"^.*$", re.M)
>>> p.findall("Python\nDjango\nSQLite")
    ['Python', 'Django', 'SQLite']
```

```
>>> p = re.compile(r"^.*$")
>>> p.findall("Python\nDjango\nSQLite")
    []
```

- S или DOTALL — метасимвол «точка» соответствует любому символу, включая перевод строки (\n). Без этого флага метасимвол «точка» будет соответствовать любому символу, исключая перевод строки. Примеры:

```
>>> p = re.compile(r"^.*$")
>>> p.findall("Python\nDjango\nSQLite")
    []
>>> p = re.compile(r"^.*$", re.S)
>>> p.findall("Python\nDjango\nSQLite")
    ['Python\nDjango\nSQLite']
```

- X или VERBOSE — пробелы, символы перевода строки и комментарии, поставленные в шаблоне регулярного выражения, игнорируются. Этот флаг позволяет форматировать шаблоны регулярных выражений для наилучшей читабельности. Примеры:

```
>>> p = re.compile(r"""^ # Привязка к началу строки
...     [0-9]+ # Строка должна содержать одну цифру (или более)
...     $      # Привязка к концу строки
...     """, re.X | re.S)
>>> print("Найдено" if p.search("1234567890") else "Нет")
    Найдено
>>> print("Найдено" if p.search("abcd123") else "Нет")
    Нет
```

- A или ASCII — классы \w, \W, \b, \B, \d, \D, \s и \S соответствуют символам в кодировке ASCII. Без этого флага указанные классы соответствуют Unicode-символам.

> ***Примечание***
>
> Флаги U и UNICODE, включающие режим соответствия классов \w, \W, \b, \B, \d, \D, \s и \S Unicode-символам, сохранены в Python лишь для совместимости с ранними версиями этого языка и никакого влияния на обработку регулярных выражений не оказывают.

- L или LOCALE — учитываются настройки текущей локали. Флаг принимается во внимание, только если шаблон регулярного выражения задан в виде последовательности байтов типа bytes или bytearray.

Шаблоны регулярных выражений удобно записывать в виде необрабатываемых строк (содержащих модификатор r):

```
p = re.compile(r"^\w+$")
```

В обычных строках все обратные слеши придется заменять специальными символами \\:

```
p = re.compile("^\\w+$")
```

В шаблоне регулярного выражения символы ., ^, $, *, +, ?, {, [,], \, |, (и) являются *специальными* и обрабатываются особым образом. Если же эти символы должны трактоваться как есть, их следует предварить обратными слешами. Некоторые специальные символы теряют свое особое значение, если их разместить внутри квадратных скобок, — в этом случае экранировать их не нужно. Например, как уже было отмечено ранее, метасимвол «точка» по умолчанию соответствует любому символу, кроме символа перевода строки. Если необхо-

димо найти именно точку, то перед точкой нужно указать символ \ или разместить точку внутри квадратных скобок: [.]. Продемонстрируем это на примере проверки правильности введенной даты (листинг 7.1).

Листинг 7.1. Проверка правильности ввода даты

```
import re          # Подключаем модуль
d = "29,12.2009"  # Вместо точки указана запятая
p = re.compile(r"^[0-3][0-9].[01][0-9].[12][09][0-9][0-9]$")
# Символ "\" не указан перед точкой
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Так как точка означает любой символ,
# выведет: Дата введена правильно

p = re.compile(r"^[0-3][0-9]\.[01][0-9]\.[12][09][0-9][0-9]$")
# Символ "\" указан перед точкой
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Так как перед точкой указан символ "\",
# выведет: Дата введена неправильно

p = re.compile(r"^[0-3][0-9][.][01][0-9][.][12][09][0-9][0-9]$")
# Точка внутри квадратных скобок
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Выведет: Дата введена неправильно
input()
```

Метасимвол — это специальный символ, обозначающий привязку к началу или концу строки:

- ^ — к началу строки или подстроки. Поведение зависит от флагов M (или MULTILINE) и S (или DOTALL);
- $ — к концу строки или подстроки. Поведение зависит от флагов M (или MULTILINE) и S (или DOTALL);
- \A — к началу строки (не зависит от флагов);
- \Z — к концу строки (не зависит от флагов).

Если указан флаг M (или MULTILINE), то при поиске в строке, состоящей из нескольких подстрок, разделенных символом \n, метасимвол ^ обозначает привязку к началу каждой такой подстроки (сразу после символа \n), а метасимвол $ — к концу каждой подстроки (непосредственно перед символом \n):

```
>>> p = re.compile(r"^.+$")        # Точка не соответствует \n
>>> p.findall("str1\nstr2\nstr3") # Ничего не найдено
    []
>>> p = re.compile(r"^.+$", re.S) # Теперь точка соответствует \n
>>> p.findall("str1\nstr2\nstr3") # Строка полностью соответствует
    ['str1\nstr2\nstr3']
>>> p = re.compile(r"^.+$", re.M) # Многострочный режим
>>> p.findall("str1\nstr2\nstr3") # Получили каждую подстроку
    ['str1', 'str2', 'str3']
```

Привязку к началу и концу строки следует использовать, если строка должна полностью соответствовать регулярному выражению. Например, для проверки, содержит ли строка число (листинг 7.2).

Листинг 7.2. Проверка наличия целого числа в строке

```
import re
p = re.compile(r"^[0-9]+$", re.S)
if p.search("245"):
    print("Число")              # Выведет: Число
else:
    print("Не число")
if p.search("Строка245"):
    print("Число")
else:
    print("Не число")           # Выведет: Не число
input()
```

Если убрать привязку к началу и концу строки, то любая строка, содержащая хотя бы одну цифру, будет распознана как «Число» (листинг 7.3).

Листинг 7.3. Отсутствие к началу или концу строки

```
import re
p = re.compile(r"[0-9]+", re.S)
if p.search("Строка245"):
    print("Число")              # Выведет: Число
else:
    print("Не число")
input()
```

Можно указать привязку только к началу или только к концу строки (листинг 7.4).

Листинг 7.4. Привязка к началу и концу строки

```
import re
p = re.compile(r"[0-9]+$", re.S)
if p.search("Строка245"):
    print("Есть число в конце строки")
else:
    print("Нет числа в конце строки")
```

```
# Выведет: Есть число в конце строки
p = re.compile(r"^[0-9]+", re.S)
if p.search("Строка245"):
    print("Есть число в начале строки")
else:
    print("Нет числа в начале строки")
# Выведет: Нет числа в начале строки
input()
```

Также поддерживаются два метасимвола, указывающие привязку к началу или концу слова:

- `\b` — к началу слова (началом слова считается пробел или любой символ, не являющийся буквой, цифрой или знаком подчеркивания);
- `\B` — к позиции, не являющейся началом слова.

Рассмотрим несколько примеров:

```
>>> p = re.compile(r"\bpython\b")
>>> print("Найдено" if p.search("python") else "Нет")
    Найдено
>>> print("Найдено" if p.search("pythonware") else "Нет")
    Нет
>>> p = re.compile(r"\Bth\B")
>>> print("Найдено" if p.search("python") else "Нет")
    Найдено
>>> print("Найдено" if p.search("this") else "Нет")
    Нет
```

В квадратных скобках `[]` можно перечислить символы, которые могут встречаться на этом месте в строке, или указать диапазон таких символов через дефис:

- `[09]` — соответствует числу 0 или 9;
- `[0-9]` — соответствует любому числу от 0 до 9;
- `[абв]` — соответствует буквам «а», «б» и «в»;
- `[а-г]` — соответствует буквам «а», «б», «в» и «г»;
- `[а-яё]` — соответствует любой букве от «а» до «я»;
- `[АБВ]` — соответствует буквам «А», «Б» и «В»;
- `[А-ЯЁ]` — соответствует любой букве от «А» до «Я»;
- `[а-яА-ЯеЁ]` — соответствует любой русской букве в любом регистре;
- `[0-9а-яА-ЯеЁa-zA-Z]` — любая цифра и любая буква независимо от регистра и языка.

> ***Внимание!***
>
> Буква «ё» не входит в диапазон `[а-я]`, а буква «Ё» — в диапазон `[А-Я]`.

Значение в скобках инвертируется, если после первой скобки вставить символ `^`, что позволяет указать символы, которых не должно быть на этом месте в строке:

- `[^09]` — не цифра 0 или 9;
- `[^0-9]` — не цифра от 0 до 9;
- `[^а-яА-ЯеЁa-zA-Z]` — не буква.

Как вы уже знаете, точка теряет свое специальное значение, если ее заключить в квадратные скобки. Кроме того, внутри квадратных скобок могут встретиться символы, которые имеют специальное значение (например, ^ и -). Символ ^ теряет свое специальное значение, если он не расположен сразу после открывающей квадратной скобки. Чтобы отменить специальное значение символа -, его необходимо указать после всех символов, перед закрывающей квадратной скобкой или сразу после открывающей квадратной скобки. Все специальные символы можно сделать обычными, если перед ними указать символ \.

Метасимвол | позволяет сделать выбор между альтернативными значениями. Выражение n|m соответствует одному из символов: n или m, например:

```
>>> p = re.compile(r"красн((ая)|(ое))")
>>> print("Найдено" if p.search("красная") else "Нет")
    Найдено
>>> print("Найдено" if p.search("красное") else "Нет")
    Найдено
>>> print("Найдено" if p.search("красный") else "Нет")
    Нет
```

Класс символов — это специальный символ, обозначающий какой-либо символ из определенного набора, символ с указанным Unicode-кодом или именем:

- \d — любая цифра. При указании флага A (ASCII) эквивалентен [0-9];
- \w — любая буква, цифра или символ подчеркивания. При указании флага A (ASCII) эквивалентен [a-zA-Z0-9_];
- \s — любой пробельный символ. При указании флага A (ASCII) эквивалентен [\t\n\r\f\v];
- \D — любой символ, не являющийся цифрой. При указании флага A (ASCII) эквивалентен [^0-9];
- \W — любой символ, не являющийся буквой, цифрой или подчеркиванием. При указании флага A (ASCII) эквивалентен [^a-zA-Z0-9_];
- \S — любой символ, не являющийся пробельным. При указании флага A (ASCII) эквивалентен [^\t\n\r\f\v].

> ***Примечание***
>
> Упомянутые здесь классы трактуются довольно широко. Так, класс \d соответствует не только десятичным цифрам, но и другим цифрам из кодировки Unicode (например, дробям), класс \w включает буквы не только латиницы, но и других алфавитов, а класс \s охватывает также неразрывные пробелы. Поэтому, если необходимо явно указать набор требуемых символов, лучше привести эти символы внутри квадратных скобок, а не использовать классы.

- \u<nnnn> — символ с 16-битным Unicode-кодом <nnnn>. Например, \u043a соответствует русской букве к;
- \U<nnnnnnnn> — символ с 32-битным Unicode-кодом <nnnnnnnn>;
- \N{<name>} (начиная с Python 3.8) — символ с Unicode-именем <name>. Например, \N{Registered Sign} соответствует знаку зарегистрированной торговой марки ®.

Квантификатор — специальный символ, задающий количество экземпляров указанного символа, которое должно присутствовать в строке:

- {n} — строго n экземпляров. Например, шаблон r"^[0-9]{2}$" соответствует двум вхождениям любой цифры;

- `{n,}` — `n` или более вхождений символа в строку. Например, шаблон `r"^[0-9]{2,}$"` соответствует двум и более экземплярам любой цифры;
- `{n,m}` — не менее `n` и не более `m` экземплярам. Значения количества указываются через запятую без пробела. Например, шаблон `r"^[0-9]{2,4}$"` соответствует 2–4 экземплярам любой цифры;
- `*` — ноль или больше экземпляров. Эквивалентно комбинации `{0,}`;
- `+` — один или больше экземпляров. Эквивалентно комбинации `{1,}`;
- `?` — ноль или один экземпляр. Эквивалентно комбинации `{0,1}`.

Все квантификаторы являются «жадными». При поиске соответствия ищется самая длинная подстрока, соответствующая шаблону, и не учитываются более короткие соответствия. Для примера получим содержимое всех тегов `<b>` вместе с самими тегами:

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>.*</b>", re.S)
>>> p.findall(s)
    ['<b>Text1</b>Text2<b>Text3</b>']
```

Вместо желаемого результата мы получили всю строку. Чтобы ограничить «жадность», необходимо после квантификатора указать символ `?`, например:

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>.*?</b>", re.S)
>>> p.findall(s)
    ['<b>Text1</b>', '<b>Text3</b>']
```

Этот код вывел то, что мы искали. Если необходимо получить содержимое без тегов, то нужный фрагмент внутри шаблона следует разместить внутри круглых скобок:

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>(.*?)</b>", re.S)
>>> p.findall(s)
    ['Text1', 'Text3']
```

Круглые скобки используются для *групп* внутри регулярных выражений. Фрагменты, соответствующие группам, будут запоминаться и станут доступны в результатах поиска (это называется *захватом фрагмента*).

Чтобы избежать захвата фрагмента, после открывающей круглой скобки в группе следует разместить символы `?:` (вопросительный знак и двоеточие):

```
>>> s = "test text"
>>> p = re.compile(r"([a-z]+((st)|(xt)))", re.S)
>>> p.findall(s)
    [('test', 'st', 'st', ''), ('text', 'xt', '', 'xt')]
>>> p = re.compile(r"([a-z]+(?:(?:st)|(?:xt)))", re.S)
>>> p.findall(s)
    ['test', 'text']
```

В первом примере мы получили список с двумя элементами. Каждый элемент списка является кортежем, содержащим четыре элемента. Все эти элементы соответствуют группам, заключенным в шаблоне в круглые скобки: первый элемент соответствует первому фрагменту, второй — второму и т. д. Три последних элемента кортежа являются лишними. Чтобы они не включались в результаты поиска, во втором примере мы добавили символы `?:`

после каждой открывающей круглой скобки. В результате список состоит только из фрагментов, полностью соответствующих регулярному выражению.

К найденному фрагменту можно обратиться с помощью механизма *обратных ссылок*. Для этого в шаблоне регулярного выражения следует указать обратный слеш и порядковый номер нужной группы (начиная с 1) (например, `\1`). Для примера получим текст между одинаковыми парными тегами:

```
>>> s = "<b>Text1</b>Text2<I>Text3</I><b>Text4</b>"
>>> p = re.compile(r"<([a-z]+)>(.*?)</\1>", re.S | re.I)
>>> p.findall(s)
    [('b', 'Text1'), ('I', 'Text3'), ('b', 'Text4')]
```

Группам можно дать имена, создав тем самым *именованные группы*. Для этого после открывающей круглой скобки в группе следует указать комбинацию символов `?P<Имя группы>`. Для примера разберем e-mail на составные части:

```
>>> email = "test@mail.ru"
>>> p = re.compile(r"""(?P<name>[a-z0-9_.-]+) # Название ящика
...     @                                     # Символ "@"
...     (?P<host>(?:[a-z0-9-]+\.)+[a-z]{2,6}) # Домен
...     """, re.I | re.VERBOSE)
>>> r = p.search(email)
>>> r.group("name")                          # Название ящика
    'test'
>>> r.group("host")                          # Домен
    'mail.ru'
```

Чтобы внутри шаблона обратиться к фрагментам из именованных групп, используется следующий синтаксис: `(?P=name)`. Для примера получим текст между одинаковыми парными тегами:

```
>>> s = "<b>Text1</b>Text2<I>Text3</I>"
>>> p = re.compile(r"<(?P<tag>[a-z]+)>(.*?)</(?P=tag)>", re.S | re.I)
>>> p.findall(s)
    [('b', 'Text1'), ('I', 'Text3')]
```

Кроме того, внутри круглых скобок могут быть расположены следующие конструкции:

- `(?#<Комментарий>)` — комментарий. Текст внутри круглых скобок игнорируется;
- `(?=...)` — положительный просмотр вперед. Выведем все слова, после которых расположена запятая:

```
>>> s = "textl, text2, text3 text4"
>>> p = re.compile(r"\w+(?=[,])", re.S | re.I)
>>> p.findall(s)
    ['textl', 'text2']
```

- `(?!...)` — отрицательный просмотр вперед. Выведем все слова, после которых нет запятой:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"[a-z]+[0-9](?![,])", re.S | re.I)
>>> p.findall(s)
    ['text3', 'text4']
```

- `(?<=...)` — положительный просмотр назад. Выведем все слова, перед которыми расположена запятая с пробелом:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"(?<=[,][ ])[a-z]+[0-9]", re.S | re.I)
>>> p.findall(s)
    ['text2', 'text3']
```

- `(?<!...)` — отрицательный просмотр назад. Выведем все слова, перед которыми расположен пробел, но перед пробелом нет запятой:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"(?<![,]) ([a-z]+[0-9])", re.S | re.I)
>>> p.findall(s)
    ['text4']
```

- `(?(<Номер>|<Имя>)<Шаблон 1>|<Шаблон 2>)` — если группа с заданным номером или именем найдена, то должно выполняться условие из параметра `<Шаблон 1>`, в противном случае — условие из параметра `<Шаблон 2>`. Выведем все слова, которые расположены внутри апострофов. Если перед словом нет апострофа, то в конце слова должна быть запятая:

```
>>> s = "text1 'text2' 'text3 text4, text5"
>>> p = re.compile(r"(')?([a-z]+[0-9])(?(1)'|,)", re.S | re.I)
>>> p.findall(s)
    [("'", 'text2'), ('', 'text4')]
```

- `(?aiLmsux)` — задает флаги у регулярного выражения. Буквы `a`, `i`, `L`, `m`, `s`, `u` и `x` имеют такое же назначение, что и одноименные флаги в функции `compile()`.

  Перед буквами `i`, `m`, `s` и `x` можно указать дефис — в этом случае соответствующий флаг, заданный ранее при создании регулярного выражения или в предыдущей подобной языковой конструкции, перестанет действовать.

Предположим, что необходимо получить все слова, расположенные после дефиса, причем перед дефисом и после слов должны следовать пробельные символы:

```
>>> s = "-word1 -word2 -word3 -word4 -word5"
>>> re.findall(r"\s\-([a-z0-9]+)\s", s, re.S | re.I)
    ['word2', 'word4']
```

Мы получили только два слова вместо пяти. Первое и последнее слова не попали в результат, поскольку расположены в начале и в конце строки. Чтобы эти слова попали в результат, необходимо добавить альтернативный выбор `(^|\s)` — для начала строки и `(\s|$)` — для конца строки. Чтобы найденные выражения внутри круглых скобок не попали в результат, следует добавить символы `?:` после открывающей скобки, например:

```
>>> re.findall(r"(?:^|\s)\-([a-z0-9]+)(?:\s|$)", s, re.S | re.I)
    ['word1', 'word3', 'word5']
```

В этом случае в результат не попали слова `word2` и `word4`. Чтобы понять причину, рассмотрим поиск по шагам. Первое слово успешно попадает в результат, т. к. перед дефисом расположено начало строки и после слова есть пробел. После поиска указатель перемещается, и строка для дальнейшего поиска примет следующий вид:

```
"-word1 <Указатель>-word2 -word3 -word4 -word5"
```

Обратите внимание, что перед фрагментом -word2 больше нет пробела и дефис не расположен в начале строки. Поэтому следующим совпадением окажется слово word3, и указатель снова будет перемещен:

```
"-word1 -word2 -word3 <Указатель>-word4 -word5"
```

Опять перед фрагментом -word4 нет пробела, и дефис не расположен в начале строки. Поэтому следующим совпадением окажется слово word5, и поиск будет завершен. Таким образом, слова word2 и word4 не попадают в результат, поскольку пробел до фрагмента уже был использован в предыдущем поиске. Чтобы этого избежать, следует воспользоваться положительным просмотром вперед (?=...), например:

```
>>> re.findall(r"(?:^|\s)\-([a-z0-9]+)(?=\s|$)", s, re.S | re.I)
    ['word1', 'word2', 'word3', 'word4', 'word5']
```

Теперь все слова успешно попали в список совпадений.

7.2. Поиск первого совпадения с шаблоном

Для поиска первого совпадения с шаблоном предназначены следующие методы, поддерживаемые объектом регулярного выражения:

- match() — ищет фрагмент, соответствующий текущему регулярному выражению, в начале заданной строки. Формат метода:

  ```
  match(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствие найдено, возвращается объект класса Match, в противном случае — значение None:

  ```
  >>> import re
  >>> p = re.compile(r"[0-9]+")
  >>> print("Найдено" if p.match("str123") else "Нет")
      Нет
  >>> print("Найдено" if p.match("str123", 3) else "Нет")
      Найдено
  >>> print("Найдено" if p.match("123str") else "Нет")
      Найдено
  ```

  Вместо метода match() можно воспользоваться функцией match(). Формат функции:

  ```
  re.match(<Шаблон>, <Строка>[, <Флаги>])
  ```

  В параметре <Шаблон> указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре <Флаги> можно указать флаги, используемые в функции compile(). Если соответствие найдено, то возвращается объект Match, в противном случае — значение None:

  ```
  >>> p = r"[0-9]+"
  >>> print("Найдено" if re.match(p, "str123") else "Нет")
      Нет
  >>> print("Найдено" if re.match(p, "123str") else "Нет")
      Найдено
  >>> p = re.compile(r"[0-9]+")
  >>> print("Найдено" if re.match(p, "123str") else "Нет")
      Найдено
  ```

- `search()` — ищет фрагмент, соответствующий текущему регулярному выражению, во всей заданной строке. Формат метода:

```
search(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Если соответствие найдено, возвращается объект `Match`, в противном случае — значение `None`:

```
>>> p = re.compile(r"[0-9]+")
>>> print("Найдено" if p.search("str123") else "Нет")
    Найдено
>>> print("Найдено" if p.search("123str") else "Нет")
    Найдено
>>> print("Найдено" if p.search("123str", 3) else "Нет")
    Нет
```

Вместо метода `search()` можно воспользоваться функцией `search()`. Формат функции:

```
re.search(<Шаблон>, <Строка>[, <Флаги>])
```

В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Флаги>` можно указать флаги, используемые в функции `compile()`. Если соответствие найдено, возвращается объект `Match`, в противном случае — значение `None`. Примеры:

```
>>> p = r"[0-9]+"
>>> print("Найдено" if re.search(p, "str123") else "Нет")
    Найдено
>>> p = re.compile(r"[0-9]+")
>>> print("Найдено" if re.search(p, "str123") else "Нет")
    Найдено
```

- `fullmatch()` — проверяет, соответствует ли заданная строка текущему регулярному выражению целиком. Формат метода:

```
fullmatch(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Если соответствие найдено, то возвращается объект `Match`, в противном случае — значение `None`:

```
>>> p = re.compile("[Pp]ython")
>>> print("Найдено" if p.fullmatch("Python") else "Нет")
    Найдено
>>> print("Найдено" if p.fullmatch("py") else "Нет")
    Нет
>>> print("Найдено" if p.fullmatch("PythonWare") else "Нет")
    Нет
>>> print("Найдено" if p.fullmatch("PythonWare", 0, 6) else "Нет")
    Найдено
```

Вместо метода `fullmatch()` можно воспользоваться функцией `fullmatch()`. Формат функции:

```
re.fullmatch(<Шаблон>, <Строка>[, <Флаги>])
```

В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Флаги>` можно указать флаги, используемые

в функции compile(). Если строка полностью совпадает с шаблоном, возвращается объект Match, в противном случае — значение None. Примеры:

```
>>> p = "[Pp]ython"
>>> print("Найдено" if re.fullmatch(p, "Python") else "Нет")
    Найдено
>>> print("Найдено" if re.fullmatch(p, "py") else "Нет")
    Нет
```

В качестве примера переделаем программу суммирования произвольного количества целых чисел, введенных пользователем, из листинга 4.12 таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой. Предусмотрим также возможность ввода отрицательных целых чисел (листинг 7.5).

Листинг 7.5. Суммирование произвольного количества чисел

```
import re
print("Введите слово 'stop' для получения результата")
summa = 0
p = re.compile(r"^[-]?[0-9]+$", re.S)
while (x := input("Введите число: ")) != "stop":
    if not p.search(x):
        print("Необходимо ввести число, а не строку!")
        continue # Переходим на следующую итерацию цикла
    x = int(x)   # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Объект класса Match, возвращаемый методами (функциями) match(), search() и fullmatch(), имеет следующие атрибуты и методы:

- re — ссылка на сам шаблон регулярного выражения, поддерживающий следующие атрибуты:
  - groups — количество групп в шаблоне;
  - groupindex — словарь с именами групп и их номерами;
  - pattern — исходная строка с шаблоном регулярного выражения;
  - flags — комбинация флагов, заданных при создании регулярного выражения в функции compile(), и флагов, указанных в самом регулярном выражении, в конструкции (?aiLmsux);
- string — значение параметра <Строка> в методах (функциях) match(), search() и fullmatch();
- pos — значение параметра <Начальная позиция> в методах match(), search() и fullmatch();
- endpos — значение параметра <Конечная позиция> в методах match(), search() и fullmatch();
- lastindex — номер последней группы или значение None, если поиск завершился неудачей;

- `lastgroup` — имя последней группы или значение `None`, если эта группа не имеет имени или поиск завершился неудачей:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m
    <re.Match object; span=(0, 12), match='123456string'>
>>> m.re.groups, m.re.groupindex
    (2, mappingproxy({'num': 1, 'str': 2}))
>>> m.re.groups, m.re.groupindex["num"]
    (2, 1)
>>> p.groups, p.groupindex
    (2, mappingproxy({'num': 1, 'str': 2}))
>>> p.groups, p.groupindex["str"]
    (2, 2)
>>> m.string
    '123456string 67890text'
>>> m.lastindex, m.lastgroup
    (2, 'str')
>>> m.pos, m.endpos
    (0, 22)
```

- `group([<Номер 1>|<Имя 1>[, . . ., <Номер n>|<Имя n>]])` — возвращает фрагменты, соответствующие группам. Если параметры не заданы или указано значение 0, возвращается фрагмент, полностью соответствующий шаблону. Если указан номер или имя группы, возвращается фрагмент, совпадающий с этой группой. Можно указать несколько номеров или имен групп — в этом случае возвращается кортеж, содержащий фрагменты, что соответствует группам. Если нет группы с указанным номером или именем, то возбуждается исключение `IndexError`. Примеры:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m.group(), m.group(0) # Полное соответствие шаблону
    ('123456string', '123456string')
>>> m.group(1), m.group(2)          # Обращение по индексу
    ('123456', 'string')
>>> m.group("num"), m.group("str") # Обращение по имени
    ('123456', 'string')
>>> m.group(1, 2), m.group("num", "str") # Несколько параметров
    (('123456', 'string'), ('123456', 'string'))
```

- `groupdict([<Значение по умолчанию>])` — возвращает словарь, содержащий фрагменты из именованных групп. Можно указать значение по умолчанию, которое будет выводиться вместо `None` для групп, не имеющих совпадений. Примеры:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
>>> m = p.search("123456")
>>> m.groupdict()
    {'num': '123456', 'str': None}
>>> m.groupdict("")
    {'num': '123456', 'str': ''}
```

- `groups([<Значение по умолчанию>])` — возвращает кортеж, содержащий фрагменты из всех групп. Можно указать значение по умолчанию, которое будет выводиться вместо `None` для групп, не имеющих совпадений. Примеры:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
>>> m = p.search("123456")
>>> m.groups()
    ('123456', None)
>>> m.groups("")
    ('123456', '')
```

- `start([<Номер или имя группы>])` — возвращает индекс начала фрагмента, соответствующего заданной группе. Если параметр не указан, возвращает индекс начала полного соответствия с шаблоном. Если соответствия нет, возвращается значение `-1`;
- `end([<Номер или имя группы>])` — возвращает индекс конца фрагмента, соответствующего заданной группе. Если параметр не указан, возвращает индекс конца полного соответствия с шаблоном. Если соответствия нет, возвращается значение `-1`;
- `span([<Номер или имя группы>])` — возвращает кортеж, содержащий начальный и конечный индексы фрагмента, который соответствует заданной группе. Если параметр не указан, возвращает индексы начала и конца полного соответствия с шаблоном. Если соответствия нет, возвращается значение `(-1, -1)`. Примеры:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> s = "str123456str"
>>> m = p.search(s)
>>> m.start(), m.end(), m.span()
    (3, 12, (3, 12))
>>> m.start(1), m.end(1), m.start("num"), m.end("num")
    (3, 9, 3, 9)
>>> m.start(2), m.end(2), m.start("str"), m.end("str")
    (9, 12, 9, 12)
>>> m.span(1), m.span("num"), m.span(2), m.span("str")
    ((3, 9), (3, 9), (9, 12), (9, 12))
>>> s[m.start(1):m.end(1)], s[m.start(2):m.end(2)]
    ('123456', 'str')
```

- `expand(<Шаблон>)` — производит замену в строке согласно заданному шаблону. Внутри последнего можно использовать обратные ссылки: `\<Номер группы>`, `\g<Номер группы>` и `\g<Имя группы>`. Содержимое не совпавших групп будет заменено на пустые строки. Для примера поменяем два тега местами:

```
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> m = p.search("<br><hr>")
>>> m.expand(r"<\2><\1>")              # \<Номер>
    '<hr><br>'
>>> m.expand(r"<\g<2>><\g<1>>")        # \g<Номер>
    '<hr><br>'
>>> m.expand(r"<\g<tag2>><\g<tag1>>") # \g<Имя>
    '<hr><br>'
```

В качестве примера использования метода `search()` проверим на соответствие шаблону введенный пользователем адрес электронной почты (листинг 7.6).

Листинг 7.6. Проверка e-mail на соответствие шаблону

```
import re
email = input("Введите e-mail: ")
pe = r"^([a-z0-9_.-]+)@(([a-z0-9-]+\.)+[a-z]{2,6})$"
p = re.compile(pe, re.I | re.S)
m = p.search(email)
if not m:
    print("E-mail не соответствует шаблону")
else:
    print("E-mail", m.group(0), "соответствует шаблону")
    print("ящик:", m.group(1), "домен:", m.group(2))
input()
```

Результат выполнения (введенное пользователем значение выделено полужирным шрифтом):

```
Введите e-mail: user@mail.ru
E-mail user@mail.ru соответствует шаблону
ящик: user домен: mail.ru
```

7.3. Поиск всех совпадений с шаблоном

Для поиска всех совпадений с шаблоном предназначены следующие методы, поддерживаемые объектом регулярного выражения:

- `findall()` — ищет в заданной строке все совпадения с текущим регулярным выражением. Формат метода:

  ```
  findall(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствия найдены, возвращается список с фрагментами, в противном случае — пустой список. Если внутри шаблона есть более одной группы, то каждый элемент списка будет кортежем, а не строкой. Примеры:

  ```
  >>> import re
  >>> p = re.compile(r"[0-9]+")
  >>> p.findall("2018, 2019, 2020, 2021, 2022")
      ['2018', '2019', '2020', '2021', '2022']
  >>> p = re.compile(r"[a-z]+")
  >>> p.findall("2018, 2019, 2020, 2021, 2022")
      []
  >>> p = re.compile(r"(([0-9]{3})-([0-9]{2})-([0-9]{2}))")
  >>> p.findall("322-77-20, 528-22-98")
      [('322-77-20', '322', '77', '20'),
       ('528-22-98', '528', '22', '98')]
  ```

  Вместо метода `findall()` можно воспользоваться функцией `findall()`. Формат функции:

  ```
  re.findall(<Шаблон>, <Строка>[, <Флаги>])
  ```

  В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Флаги>` можно указать флаги, используемые в функции `compile()`. Примеры:

```
>>> re.findall(r"[0-9]+", "1 2 3 4 5 6")
    ['1', '2', '3', '4', '5', '6']
>>> p = re.compile(r"[0-9]+")
>>> re.findall(p, "1 2 3 4 5 6")
    ['1', '2', '3', '4', '5', '6']
```

- `finditer()` — аналогичен методу `findall()`, но возвращает итератор, который на каждой итерации выдает объект класса `Match`. Формат метода:

  ```
  finditer(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Пример:

  ```
  >>> p = re.compile(r"[0-9]+")
  >>> for m in p.finditer("2018, 2019, 2020, 2021, 2022"):
  ...     print(m.group(0), "start:", m.start(), "end:", m.end())
  ...
      2018 start: 0 end: 4
      2019 start: 6 end: 10
      2020 start: 12 end: 16
      2021 start: 18 end: 22
      2022 start: 24 end: 28
  ```

  Вместо метода `finditer()` можно воспользоваться функцией `finditer()`. Ее формат:

  ```
  re.finditer(<Шаблон>, <Строка>[, <Флаги>])
  ```

  В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Флаги>` можно указать флаги, используемые в функции `compile()`. Получим содержимое между тегами:

  ```
  >>> p = re.compile(r"<b>(.+?)</b>", re.I | re.S)
  >>> s = "<b>Text1</b>Text2<b>Text3</b>"
  >>> for m in re.finditer(p, s):
  ...     print(m.group(1))
  ...
      Text1
      Text3
  ```

7.4. Замена в строке

Метод `sub()`, поддерживаемый регулярным выражением, ищет в указанной строке все совпадения с текущим регулярным выражением и заменяет их фрагментом, соответствующим заданному регулярному выражению. Содержимое несовпавших групп будет заменено на пустые строки. Если совпадения не найдены, возвращается исходная строка. Метод имеет следующий формат:

```
sub(<Регулярное выражение, задающее замену>, <Строка>[,
    <Максимальное количество замен>])
```

Внутри регулярного выражения, задающего замену, можно использовать обратные ссылки `\<Номер группы>`, `\g<Номер группы>` и `\g<Имя группы>`. Для примера поменяем два тега местами:

```
>>> import re
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> p.sub(r"<\2><\1>", "Теги <br><hr>")              # \<Номер>
    'Теги <hr><br>'
>>> p.sub(r"<\g<2>><\g<1>>", "Теги <br><hr>")       # \g<Номер>
    'Теги <hr><br>'
>>> p.sub(r"<\g<tag2>><\g<tag1>>", "Теги <br><hr>") # \g<Имя>
    'Теги <hr><br>'
```

В первом параметре можно указать ссылку на функцию. В эту функцию будет передаваться объект класса `Match`, соответствующий найденному фрагменту. Результат, возвращаемый этой функцией, послужит заменяющим фрагментом. Для примера найдем все числа в строке и прибавим к ним 10 (листинг 7.7).

Листинг 7.7. Поиск чисел в строке

```
import re
def repl(m):
    """ Функция для замены. m — объект Match """
    x = int(m.group(0))
    x += 10
    return "{0}".format(x)

p = re.compile(r"[0-9]+")
# Заменяем все вхождения
print(p.sub(repl, "2019, 2020, 2021, 2022"))
# Заменяем только первые два вхождения
print(p.sub(repl, "2019, 2020, 2021, 2022", 2))
input()
```

Результат выполнения:

```
2029, 2030, 2031, 2032
2029, 2030, 2021, 2022
```

Вместо метода `sub()` можно воспользоваться функцией `sub()`. Формат функции:

```
re.sub(<Шаблон>, <Регулярное выражение, задающее замену>,
       <Строка>[, <Максимальное количество замен>[, flags=0]])
```

В качестве параметров `<Шаблон>` и `<Регулярное выражение, задающее замену>` можно указать строки с регулярными выражениями или скомпилированные регулярные выражения. В параметре `flags` можно указать флаги, используемые в функции `compile()`. Для примера поменяем два тега местами, а также изменим регистр букв (листинг 7.8).

Листинг 7.8. Перестановка тегов с изменением регистра букв

```
import re
def repl(m):
    """ Функция для замены. m — объект Match """
    tag1 = m.group("tag1").upper()
    tag2 = m.group("tag2").upper()
    return "<{0}><{1}>".format(tag2, tag1)
```

```
p = r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>"
print(re.sub(p, repl, "<br><hr>"))
input()
```

Результат выполнения:

```
<HR><BR>
```

Метод `subn()` аналогичен методу `sub()`, но возвращает кортеж из двух элементов: измененной строки и количества произведенных замен. Метод имеет следующий формат:

```
subn(<Регулярное выражение, задающее замену>, <Строка>[,
     <Максимальное количество замен>])
```

Заменим все числа в строке на `0`:

```
>>> p = re.compile(r"[0-9]+")
>>> p.subn("0", "2019, 2020, 2021, 2022")
    ('0, 0, 0, 0', 4)
```

Вместо метода `subn()` можно воспользоваться функцией `subn()`. Формат функции:

```
re.subn(<Шаблон>, <Регулярное выражение, задающее замену>,
        <Строка>[, <Максимальное количество замен>[, flags=0]])
```

В качестве параметров `<Шаблон>` и `<Регулярное выражение, задающее замену>` можно указать строки с регулярными выражениями или скомпилированные регулярные выражения. В параметре `flags` можно указать флаги, используемые в функции `compile()`. Примеры:

```
>>> p = r"201[89]"
>>> re.subn(p, "2022", "2019, 2020, 2021, 2018")
    ('2022, 2020, 2021, 2022', 2)
```

Для выполнения замен также можно использовать метод `expand(<Шаблон>)`, поддерживаемый объектом `Match`. Внутри указанного шаблона можно использовать обратные ссылки: `\<Номер группы>`, `\g<Номер группы>` и `\g<Имя группы>`:

```
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> m = p.search("<br><hr>")
>>> m.expand(r"<\2><\1>")              # \<Номер>
    '<hr><br>'
>>> m.expand(r"<\g<2>><\g<1>>")        # \g<Номер>
    '<hr><br>'
>>> m.expand(r"<\g<tag2>><\g<tag1>>") # \g<Имя>
    '<hr><br>'
```

7.5. Прочие функции и методы

Метод `split(<Строка>[, <Лимит>])` разбивает заданную строку по шаблону текущего регулярного выражения и возвращает список подстрок. Если во втором параметре задано число, то в списке окажется указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать еще один элемент — с остатком строки. Примеры:

```
>>> import re
>>> p = re.compile(r"[\s,.]+")
```

```
>>> p.split("word1, word2\nword3\r\nword4.word5")
    ['word1', 'word2', 'word3', 'word4', 'word5']
>>> p.split("word1, word2\nword3\r\nword4.word5", 2)
    ['word1', 'word2', 'word3\r\nword4.word5']
```

Если разделитель в строке не найден, список будет состоять только из одного элемента, содержащего исходную строку:

```
>>> p = re.compile(r"[0-9]+")
>>> p.split("word, word\nword")
    ['word, word\nword']
```

Вместо метода `split()` можно воспользоваться функцией `split()`. Формат функции:

```
re.split(<Шаблон>, <Строка>[, <Лимит>[, flags=0]])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. В параметре `flags` можно указать флаги, используемые в функции `compile()`. Примеры:

```
>>> p = re.compile(r"[\s,.]+")
>>> re.split(p, "word1, word2\nword3")
    ['word1', 'word2', 'word3']
>>> re.split(r"[\s,.]+", "word1, word2\nword3")
    ['word1', 'word2', 'word3']
```

Функция `escape(<Строка>)` экранирует все специальные символы в заданной строке, после чего ее можно безопасно использовать в составе регулярного выражения:

```
>>> print(re.escape(r"[]().*"))
    \[\]\(\)\.\*
```

Функция `purge()` очищает кеш, в котором хранятся промежуточные данные, используемые при выполнении регулярных выражений. Ее рекомендуется вызывать после обработки большого количества регулярных выражений. Результата эта функция не возвращает. Пример:

```
>>> re.purge()
```

ГЛАВА 8

Списки, кортежи, множества и диапазоны

Список, кортеж, множество и диапазон — это упорядоченные наборы значений, иначе говоря, *последовательности*. Значение, входящее в последовательность, называется *элементом* и может быть любого типа: числом, строкой, логической величиной и даже другой последовательностью. Количество элементов в последовательности называется *размером*.

Большинство последовательностей (в частности, список, кортеж и диапазон) являются *пронумерованными*, поскольку в них за каждым элементом закреплен порядковый номер — *индекс*. Зная его, можно извлечь из списка или кортежа соответствующий этому индексу элемент. Нумерация элементов в пронумерованной последовательности начинается с 0.

Часть последовательностей (например, множество) относятся к группе *непронумерованных*, т. к. у их элементов нет индексов. Соответственно, обратиться к элементу множества по индексу невозможно.

8.1. Создание списков

Список — это изменяемая пронумерованная последовательность.

Создать список можно следующими способами:

- с помощью функции `list([<Последовательность>])`. Функция возвращает список, созданный на основе указанной последовательности. Если последовательность не указана, выдается пустой список. Примеры:

```
>>> list()                    # Создаем пустой список
    []
>>> list("String")            # Преобразуем строку в список
    ['S', 't', 'r', 'i', 'n', 'g']
>>> list((1, 2, 3, 4, 5))     # Преобразуем кортеж в список
    [1, 2, 3, 4, 5]
```

- приведя элементы списка внутри квадратных скобок через запятую:

```
>>> arr = [1, "str", 3, "4"]
>>> arr
    [1, 'str', 3, '4']
```

- заполнив список поэлементно с помощью метода `append()`:

```
>>> arr = []            # Создаем пустой список
>>> arr.append(1)       # Добавляем элемент1 (индекс 0)
```

```
>>> arr.append("str")  # Добавляем элемент2 (индекс 1)
>>> arr
    [1, 'str']
```

Не забываем, что при присваивании объекта (например, списка) какой-либо переменной в последнюю заносится не сам объект, а ссылка на него (подробности — в *разд. 2.2* и *2.3*). В результате получатся две переменные, указывающие на один и тот же объект. Пример:

```
>>> x = y = [1, 2]        # Якобы создали два списка
>>> x, y
    ([1, 2], [1, 2])
>>> y[1] = 100            # Изменяем второй элемент списка из второй переменной
>>> x, y                  # Изменилось значение сразу в двух переменных
    ([1, 100], [1, 100])
```

Но что же делать, если необходимо создать копию списка? Первый способ заключается в применении операции извлечения среза, второй — в использовании функции `list()`, а третий — в вызове метода `copy()`. Примеры:

```
>>> x = [1, 2, 3, 4, 5] # Создали список
>>> # Создаем копию списка
>>> y = list(x) # или с помощью среза: y = x[:]
>>>             # или вызовом метода copy(): y = x.copy()
>>> y
    [1, 2, 3, 4, 5]
>>> x is y      # Оператор is показывает, что это разные объекты
    False
>>> y[1] = 100  # Изменяем второй элемент
>>> x, y        # Изменился только список в переменной y
    ([1, 2, 3, 4, 5], [1, 100, 3, 4, 5])
```

Однако, если в числе элементов копируемого списка имеются другие списки (*вложенные*), будут скопированы не сами эти списки, а ссылки на них. Как говорят в таких случаях программисты, будет создана *поверхностная копия*. Рассмотрим пример:

```
>>> x = [1, [2, 3, 4, 5]] # Создали вложенный список
>>> y = list(x)           # Якобы сделали копию списка
>>> x is y                # Разные объекты
    False
>>> y[1][1] = 100         # Изменяем элемент вложенного списка
>>> x, y                  # Изменение затронуло переменную x!!!
    ([1, [2, 100, 4, 5]], [1, [2, 100, 4, 5]])
```

Чтобы получить *полную копию* списка вместе с вложенными списками, следует воспользоваться функцией `deepcopy()` из модуля `copy`:

```
>>> import copy             # Подключаем модуль copy
>>> x = [1, [2, 3, 4, 5]]
>>> y = copy.deepcopy(x)    # Делаем полную копию списка
>>> y[1][1] = 100           # Изменяем элемент вложенного списка
>>> x, y                    # Изменился только список в переменной y
    ([1, [2, 3, 4, 5]], [1, [2, 100, 4, 5]])
```

Функция `deepcopy()` создает копию каждого объекта, при этом сохраняя внутреннюю структуру списка. Иными словами, если в списке существуют два элемента, ссылающиеся

на один объект, то будет создана копия объекта и элементы будут ссылаться на этот новый объект, а не на разные объекты. Пример:

```
>>> import copy
>>> x = [1, 2]
>>> y = [x, x]           # Два элемента ссылаются на один список
>>> y
    [[1, 2], [1, 2]]
>>> z = copy.deepcopy(y) # Сделали копию списка
>>> z[0] is x, z[1] is x, z[0] is z[1]
    (False, False, True)
>>> z[0][0] = 300        # Изменили один элемент вложенного списка
>>> z                    # Значение изменилось сразу в двух вложенных списках!
    [[300, 2], [300, 2]]
>>> x                    # Начальный список не изменился
    [1, 2]
```

8.2. Операции над списками

Чтобы получить элемент списка по его индексу, следует поставить после списка квадратные скобки и указать индекс в них:

```
>>> arr = [1, "str", 3.2, "4"]
>>> arr[0], arr[1], arr[2], arr[3]
    (1, 'str', 3.2, '4')
```

С помощью позиционного присваивания можно присвоить значения элементов списка каким-либо переменным. Количество элементов справа и слева от оператора = должно совпадать, иначе будет выведено сообщение об ошибке. Примеры:

```
>>> x, y, z = [1, 2, 3] # Позиционное присваивание
>>> x, y, z
    (1, 2, 3)
>>> x, y = [1, 2, 3]    # Количество элементов должно совпадать
    Traceback (most recent call last):
      File "<pyshell#86>", line 1, in <module>
        x, y = [1, 2, 3]
    ValueError: too many values to unpack (expected 2)
```

Перед одной из переменных слева от оператора = можно указать звездочку — и тогда в этой переменной будет сохранен список из «лишних» элементов. Если таких элементов нет, список будет пустым. Примеры:

```
>>> x, y, *z = [1, 2, 3]; x, y, z
    (1, 2, [3])
>>> x, y, *z = [1, 2, 3, 4, 5]; x, y, z
    (1, 2, [3, 4, 5])
>>> x, y, *z = [1, 2]; x, y, z
    (1, 2, [])
>>> *x, y, z = [1, 2]; x, y, z
    ([], 1, 2)
```

```
>>> x, *y, z = [1, 2, 3, 4, 5]; x, y, z
    (1, [2, 3, 4], 5)
>>> *z, = [1, 2, 3, 4, 5]; z
    [1, 2, 3, 4, 5]
```

В качестве индекса можно указать отрицательное значение. Такой индекс будет отсчитываться от конца списка. Пример:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[-1], arr[-3]
    (5, 3)
```

Так как списки относятся к изменяемым типам данных, мы можем изменить элемент по его индексу:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[0] = 600              # Изменение элемента по индексу
>>> arr
    [600, 2, 3, 4, 5]
```

Если элемент с указанным индексом отсутствует в списке, возбуждается исключение `IndexError`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[5]                # Обращение к несуществующему элементу
    Traceback (most recent call last):
      File "<pyshell#99>", line 1, in <module>
        arr[5]                # Обращение к несуществующему элементу
    IndexError: list index out of range
```

Получить размер списка (количество элементов в нем) позволяет функция `len()`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> len(arr)              # Получаем количество элементов
    5
>>> arr[len(arr)-1]       # Получаем последний элемент
    5
```

Срез — это фрагмент списка, который сам является списком и содержит элементы, индексы которых располагаются между указанными начальным, конечным индексами и отстоят друг от друга на заданный шаг. Формат операции извлечения среза:

```
[<Начало>:<Конец>:<Шаг>]
```

Все параметры не являются обязательными. Если параметр `<Начало>` не указан, используется значение 0. Если параметр `<Конец>` не указан, возвращается фрагмент до конца списка. Элемент с индексом, указанным в параметре `<Конец>`, не входит в возвращаемый срез. Если параметр `<Шаг>` не указан, используется значение 1. В качестве значения параметров можно указать отрицательные значения.

Рассмотрим несколько примеров:

- получение поверхностной копии списка:

```
>>> arr = [1, 2, 3, 4, 5]
>>> m = arr[:]; m # Создаем поверхностную копию и выводим значения
    [1, 2, 3, 4, 5]
```

```
>>> m is arr      # Оператор is показывает, что это разные объекты
    False
```

- вывод элементов списка в обратном порядке:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[::-1]             # Шаг -1
    [5, 4, 3, 2, 1]
```

- вывод списка без первого и последнего элементов:

```
>>> arr[1:]               # Без первого элемента
    [2, 3, 4, 5]
>>> arr[:-1]              # Без последнего элемента
    [1, 2, 3, 4]
```

- извлечение среза, содержащего первые два элемента списка:

```
>>> arr[0:2]              # Символ с индексом 2 не входит в диапазон
    [1, 2]
```

- получение среза, содержащего последний элемент списка:

```
>>> arr[-1:]              # Последний элемент списка
    [5]
```

- получение среза с элементами от второго до четвертого включительно:

```
>>> arr[1:4] # Возвращаются элементы с индексами 1, 2 и 3
    [2, 3, 4]
```

С помощью среза можно изменить фрагмент списка. Если срезу присвоить пустой список, то элементы, попавшие в срез, будут удалены. Примеры:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[1:3] = [6, 7] # Изменяем значения элементов с индексами 1 и 2
>>> arr
    [1, 6, 7, 4, 5]
>>> arr[1:3] = []     # Удаляем элементы с индексами 1 и 2
>>> arr
    [1, 4, 5]
```

Объединить два списка в один список позволяет оператор +. Результатом объединения будет новый список. Пример:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = [6, 7, 8, 9]
>>> arr3 = arr1 + arr2
>>> arr3
    [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Вместо оператора + можно использовать оператор +=. Следует учитывать, что в этом случае элементы добавляются в текущий список. Пример:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr += [6, 7, 8, 9]
>>> arr
    [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Еще списки поддерживают операции повторения (оператор *), проверки на вхождение (оператор `in`) и на невхождение (оператор `not in`):

```
>>> [1, 2, 3] * 3                                  # Повторение
    [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5]    # Проверка на вхождение
    (True, False)
>>> 2 not in [1, 2, 3, 4], 6 not in [1, 2, 3, 4] # Проверка на невхождение
    (False, True)
```

8.3. Многомерные списки

Многомерным называется список, содержащий в числе своих элементов вложенные списки (или иные последовательности). Создать многомерный список можно, например, так:

```
>>> arr = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

Как вы уже знаете, выражение внутри скобок может располагаться на нескольких строках. Следовательно, предыдущий пример можно записать иначе:

```
>>> arr = [
...             [1, 2, 3],
...             [4, 5, 6],
...             [7, 8, 9]
...         ]
```

Чтобы получить значение элемента из вложенного списка, следует указать два индекса:

```
>>> arr[1][1]
    5
```

Вложенные списки также могут содержать вложенные списки. В этом случае для доступа к элементам указывается несколько индексов подряд, например:

```
>>> arr = [ [1, ["a", "b"], 3], [4, 5, 6], [7, 8, 9] ]
>>> arr[0][1][0]
    'a'
>>> arr = [ [1, { "a": 10, "b": ["s", 5] } ] ]
>>> arr[0][1]["b"][0]
    's'
```

8.4. Перебор списков

Последовательно перебрать все элементы списка можно с помощью цикла перебора последовательности (см. *разд. 4.4*):

```
>>> arr = [1, 2, 3, 4, 5]
>>> for i in arr: print(i, end=" ")
...
    1 2 3 4 5
```

Значение очередного элемента списка заносится в переменную, указанную в цикле. Его можно изменить в теле цикла, но если оно относится к неизменяемому типу данных (например, числовому или строковому), это не отразится на исходном списке. Пример:

```
>>> arr = [1, 2, 3, 4]        # Элементы имеют неизменяемый тип (число)
>>> for i in arr: i += 10
...
```

```
>>> arr                          # Список не изменился
    [1, 2, 3, 4]
>>> arr = [ [1, 2], [3, 4] ] # Элементы имеют изменяемый тип (список)
>>> for i in arr: i[0] += 10
...
>>> arr                          # Список изменился
    [[11, 2], [13, 4]]
```

Однако изменить список в цикле все же можно, если для генерирования индексов его элементов воспользоваться функцией `range()`, возвращающей диапазон. Функция имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение диапазона. Если он не указан, используется значение 0. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый диапазон. Если параметр `<Шаг>` не указан, используется значение 1. Для примера умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)                      # Результат выполнения: [2, 4, 6, 8]
```

Также можно воспользоваться функцией `enumerate(<Список>[, start=0])`, которая на каждой итерации цикла `for` возвращает кортеж из индекса и значения очередного элемента указанного списка. Параметр `start` задает индекс элемента, с которого начнется перебор списка. Умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i, elem in enumerate(arr):
    arr[i] *= 2
print(arr)                      # Результат выполнения: [2, 4, 6, 8]
```

Перебрать список можно и с помощью цикла с условием, но нужно помнить, что он выполняется медленнее цикла перебора последовательности. Для примера умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
i, c = 0, len(arr)
while i < c:
    arr[i] *= 2
    i += 1
print(arr)                      # Результат выполнения: [2, 4, 6, 8]
```

8.5. Генераторы списков и выражения-генераторы

Генератор списка заполняет создаваемый список элементами, полученными в результате каких-либо вычислений, которые выполняются над элементами другой последовательности. Формат записи генератора списка:

```
[ <Выражение> for <Переменная> in <Исходная последовательность> ]
```

Генератор списка перебирает заданную исходную последовательность, на каждой итерации занося значение ее очередного элемента в указанную переменную, и выполняет заданное выражение, проводящее необходимые вычисления над значением этого элемента (оно доступно из указанной переменной). Результаты вычислений, проведенных над всеми элементами исходной последовательности, сводятся во вновь созданный список, который и выдается в качестве результата.

Генератор цикла можно рассматривать как более компактную и производительную разновидность цикла перебора последовательности.

Для примера вернемся в *разд. 8.4* и найдем код, изменяющий элементы списка:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)                    # Результат выполнения: [2, 4, 6, 8]
```

Использовав генератор списка, можно заметно сократить объем этого кода (и ускорить его выполнение):

```
arr = [1, 2, 3, 4]
arr = [ i * 2 for i in arr ]
print(arr)                    # Результат выполнения: [2, 4, 6, 8]
```

Генераторы списков могут быть вложены друг в друга и (или) содержать оператор ветвления. Для примера получим четные элементы списка и умножим их на 10:

```
arr = [1, 2, 3, 4]
arr = [ i * 10 for i in arr if i % 2 == 0 ]
print(arr)                    # Результат выполнения: [20, 40]
```

Этот код эквивалентен коду:

```
arr = []
for i in [1, 2, 3, 4]:
    if i % 2 == 0:            # Если число четное
        arr.append(i * 10)    # Добавляем элемент
print(arr)                    # Результат выполнения: [20, 40]
```

Теперь получим четные элементы вложенного списка и умножим их на 10:

```
arr = [[1, 2], [3, 4], [5, 6]]
arr = [ j * 10 for i in arr for j in i if j % 2 == 0 ]
print(arr)                    # Результат выполнения: [20, 40, 60]
```

Этот код эквивалентен коду:

```
arr = []
for i in [[1, 2], [3, 4], [5, 6]]:
    for j in i:
        if j % 2 == 0:            # Если число четное
            arr.append(j * 10)    # Добавляем элемент
print(arr)                        # Результат выполнения: [20, 40, 60]
```

Выражение-генератор аналогично генератору списка, только выдает не список, а *итератор* — особый объект, обладающий функциональностью последовательности. В частности, итератор можно перебрать, воспользовавшись циклом перебора последовательности. Формат записи выражения-генератора:

```
( <Выражение> for <Переменная> in <Исходная последовательность> )
```

Для примера вычислим посредством выражения-генератора квадратные корни чисел от 1 до 5 и выведем их на экран в цикле перебора:

```
>>> import math
>>> expgen = ( math.sqrt(n) for n in range(1, 6) )
>>> for p in expgen: print(p)
...
    1.0
    1.4142135623730951
    1.7320508075688772
    2.0
    2.23606797749979
```

8.6. Функции *map()*, *zip()*, *filter()* и *reduce()*

Функция `map()` применяет заданную функцию к каждому элементу указанной последовательности и сводит возвращенные ей результаты в новую последовательность, которую и возвращает в качестве результата. Она имеет такой формат:

```
map(<Функция>, <Последовательность 1>[, . . ., <Последовательность N>])
```

Задаваемая функция должна принимать единственный параметр — очередной элемент указанной последовательности. Еще она должна возвращать результат (обычно получаемый в результате каких-либо вычислений с применением полученного элемента).

Функция `map()` в качестве результата возвращает итератор (объект с функциональностью последовательности, обычно ограниченной). Чтобы получить список, возвращенный итератор необходимо передать в функцию `list()`.

Для примера прибавим к каждому элементу списка число 10 (листинг 8.1).

Листинг 8.1. Функция `map()`

```
def func(elem):
    """ Увеличение значения каждого элемента списка """
    return elem + 10 # Возвращаем новое значение

arr = [1, 2, 3, 4, 5]
print( list( map(func, arr) ) )
# Результат выполнения: [11, 12, 13, 14, 15]
```

Функции `map()` можно передать несколько последовательностей. В этом случае в заданной функции будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковом смещении. Просуммируем элементы трех списков (листинг 8.2).

Листинг 8.2. Суммирование элементов трех списков

```
def func(e1, e2, e3):
    """ Суммирование элементов трех разных списков """
    return e1 + e2 + e3 # Возвращаем новое значение

arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
```

```
arr3 = [100, 200, 300, 400, 500]
print( list( map(func, arr1, arr2, arr3) ) )
# Результат выполнения: [111, 222, 333, 444, 555]
```

Если размеры заданных последовательностей различаются, за основу выбирается последовательность с минимальным количеством элементов (листинг 8.3).

Листинг 8.3. Суммирование элементов трех списков разного размера

```
def func(e1, e2, e3):
    """ Суммирование элементов трех разных списков """
    return e1 + e2 + e3

arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20]
arr3 = [100, 200, 300, 400, 500]
print( list( map(func, arr1, arr2, arr3) ) )
# Результат выполнения: [111, 222]
```

Функция `zip()` возвращает итератор, при переборе последовательно выдающий кортежи, содержащие элементы указанных последовательностей, расположенные на одинаковом смещении. Формат функции:

```
zip(<Последовательность 1>[, . . ., <Последовательность N>][, strict=False])
```

Пример:

```
>>> zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
    <zip object at 0x00FCAC88>
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))
    [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Если размеры заданных последовательностей различаются и параметру `strict` (поддерживается, начиная с Python 3.10) дано значение `False`, в результат попадут только элементы, которые существуют во всех последовательностях на одинаковом смещении:

```
>>> list(zip([1, 2, 3], [4, 6], [7, 8, 9, 10]))
    [(1, 4, 7), (2, 6, 8)]
```

Если же параметру `strict` дано значение `True`, при передаче функции `zip()` последовательностей разного размера будет сгенерировано исключение `ValueError`:

```
>>> list(zip([1, 2, 3], [4, 6], [7, 8, 9, 10], strict=True))
    Traceback (most recent call last):
      File "<pyshell#8>", line 1, in <module>
        list(zip([1, 2, 3], [4, 6], [7, 8, 9, 10], strict=True))
    ValueError: zip() argument 2 is shorter than argument 1
```

Переделаем программу суммирования элементов трех списков из листинга 8.3, использовав функцию `zip()` (листинг 8.4).

Листинг 8.4. Суммирование элементов трех списков с помощью функции `zip()`

```
arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
```

```
arr3 = [100, 200, 300, 400, 500]
arr = [x + y + z for (x, y, z) in zip(arr1, arr2, arr3)]
print(arr)
# Результат выполнения: [111, 222, 333, 444, 555]
```

Функция `filter()` выполняет проверку элементов последовательности на соответствие заданным критериям. Формат функции:

```
filter(<Функция>, <Последовательность>)
```

Указанная функция должна в качестве параметра принимать очередной элемент заданной последовательности и возвращать `True` или `False`. В результат будут добавлены только те элементы, для которых функция вернет `True`. Если вместо функции указать значение `None`, очередной элемент будет проверяться на соответствие значению `True`.

Функция `filter()` возвращает итератор. Чтобы получить список, этот итератор необходимо передать в функцию `list()`. Примеры:

```
>>> filter(None, [1, 0, None, [], 2])
    <filter object at 0x00FD58B0>
>>> list(filter(None, [1, 0, None, [], 2]))
    [1, 2]
```

Аналогичный код с использованием генераторов списков выглядит так:

```
>>> [ i for i in [1, 0, None, [], 2] if i ]
    [1, 2]
```

Для примера удалим все отрицательные значения из списка (листинг 8.5).

Листинг 8.5. Пример использования функции `filter()`

```
def func(elem):
    return elem >= 0

arr = [-1, 2, -3, 4, 0, -20, 10]
arr = list(filter(func, arr))
print(arr)                              # Результат: [2, 4, 0, 10]
# Использование генераторов списков
arr = [-1, 2, -3, 4, 0, -20, 10]
arr = [ i for i in arr if func(i) ]
print(arr)                              # Результат: [2, 4, 0, 10]
```

Функция `reduce()` из модуля `functools` применяет указанную функцию к парам элементов заданной последовательности и накапливает результат. Функция имеет следующий формат:

```
reduce(<Функция>, <Последовательность>[, <Начальное значение>])
```

В заданную функцию передаются два параметра: результат предыдущих вычислений и значение очередного элемента указанной последовательности. При обработке первого элемента последовательности функции передаются:

- если начальное значение задано — начальное значение и первый элемент последовательности;
- если начальное значение не задано — первый и второй элементы последовательности;

Получим сумму всех элементов списка (листинг 8.6).

Листинг 8.6. Пример использования функции reduce()

```
from functools import reduce

def func(x, y):
    print("({0}, {1})".format(x, y), end=" ")
    return x + y

arr = [1, 2, 3, 4, 5]
summa = reduce(func, arr)
# Последовательность: (1, 2) (3, 3) (6, 4) (10, 5)
print(summa) # Результат выполнения: 15
summa = reduce(func, arr, 10)
# Последовательность: (10, 1) (11, 2) (13, 3) (16, 4) (20, 5)
print(summa) # Результат выполнения: 25
summa = reduce(func, [], 10)
print(summa) # Результат выполнения: 10
```

8.7. Добавление и удаление элементов списка

Для добавления и удаления элементов списка используются следующие методы:

- `append(<Значение>)` — добавляет значение в качестве элемента в конец текущего списка. Результата не возвращает. Примеры:

```
>>> arr = [1, 2, 3]
>>> arr.append(4); arr          # Добавляем число
    [1, 2, 3, 4]
>>> arr.append([5, 6]); arr     # Добавляем список
    [1, 2, 3, 4, [5, 6]]
>>> arr.append((7, 8)); arr     # Добавляем кортеж
    [1, 2, 3, 4, [5, 6], (7, 8)]
```

- `extend(<Последовательность>)` — добавляет элементы заданной последовательности в конец текущего списка. Результата не возвращает. Примеры:

```
>>> arr = [1, 2, 3]
>>> arr.extend([4, 5, 6])       # Добавляем список
>>> arr.extend((7, 8, 9))       # Добавляем кортеж
>>> arr.extend("abc")           # Добавляем буквы из строки
>>> arr
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 'a', 'b', 'c']
```

Добавить несколько элементов можно посредством конкатенации или оператора +=:

```
>>> arr = [1, 2, 3]
>>> arr + [4, 5, 6]             # Возвращает новый список
    [1, 2, 3, 4, 5, 6]
>>> arr += [4, 5, 6]            # Изменяет текущий список
>>> arr
    [1, 2, 3, 4, 5, 6]
```

Кроме того, можно воспользоваться операцией присваивания значения срезу:

```
>>> arr = [1, 2, 3]
>>> arr[len(arr):] = [4, 5, 6]  # Изменяет текущий список
>>> arr
    [1, 2, 3, 4, 5, 6]
```

- `insert(<Индекс>, <Значение>)` — добавляет заданное значение в текущий список по указанному индексу. Остальные элементы смещаются. Результата не возвращает. Примеры:

```
>>> arr = [1, 2, 3]
>>> arr.insert(0, 0); arr   # Вставляем 0 в начало списка
    [0, 1, 2, 3]
>>> arr.insert(-1, 20); arr # Можно указать отрицательные числа
    [0, 1, 2, 20, 3]
>>> arr.insert(2, 100); arr # Вставляем 100 в позицию 2
    [0, 1, 100, 2, 20, 3]
>>> arr.insert(10, [4, 5]); arr # Добавляем список
    [0, 1, 100, 2, 20, 3, [4, 5]]
```

Метод `insert()` позволяет добавить только один элемент. Чтобы добавить несколько элементов, можно воспользоваться операцией присваивания значения срезу. Добавим несколько элементов в начало списка:

```
>>> arr = [1, 2, 3]
>>> arr[:0] = [-2, -1, 0]
>>> arr
    [-2, -1, 0, 1, 2, 3]
```

- `pop([<Индекс>])` — удаляет из текущего списка элемент, расположенный по указанному индексу, и возвращает его. Если индекс не указан, то удаляет и возвращает последний элемент списка. Если элемента с указанным индексом нет или список пустой, возбуждается исключение `IndexError`. Примеры:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.pop()        # Удаляем последний элемент списка
    5
>>> arr              # Список изменился
    [1, 2, 3, 4]
>>> arr.pop(0)       # Удаляем первый элемент списка
    1
>>> arr              # Список изменился
    [2, 3, 4]
```

Удалить элемент списка позволяет также оператор `del`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> del arr[4]; arr  # Удаляем последний элемент списка
    [1, 2, 3, 4]
>>> del arr[:2]; arr # Удаляем первый и второй элементы
    [3, 4]
```

- `remove(<Значение>)` — удаляет из текущего списка первый элемент, содержащий указанное значение. Если элемент не найден, возбуждается исключение `ValueError`. Метод ничего не возвращает. Примеры:

```
>>> arr = [1, 2, 3, 1, 1]
>>> arr.remove(1)    # Удаляет только первый элемент
>>> arr
    [2, 3, 1, 1]
>>> arr.remove(5)    # Такого элемента нет
    Traceback (most recent call last):
      File "<pyshell#3>", line 1, in <module>
        arr.remove(5)    # Такого элемента нет
    ValueError: list.remove(x): x not in list
```

- `clear()` — удаляет все элементы текущего списка, очищая его. Результата не возвращает. Пример:

```
>>> arr = [1, 2, 3, 1, 1]
>>> arr.clear()
>>> arr
    []
```

Если необходимо удалить все повторяющиеся элементы списка, то можно преобразовать список во множество, а затем множество обратно преобразовать в список. Обратите внимание на то, что список должен содержать только неизменяемые объекты (например, числа, строки или кортежи). В противном случае возбуждается исключение `TypeError`. Пример:

```
>>> arr = [1, 2, 3, 1, 1, 2, 2, 3, 3]
>>> s = set(arr)          # Преобразуем список во множество
>>> s
    {1, 2, 3}
>>> arr = list(s)         # Преобразуем множество в список
>>> arr                   # Все повторы были удалены
    [1, 2, 3]
```

8.8. Поиск элемента в списке и получение сведений об элементах списка

Выполнить проверку на вхождение элемента в список позволяет оператор `in`: если элемент входит в список, то возвращается значение `True`, в противном случае — `False`. Аналогичный оператор `not in` выполняет проверку на невхождение элемента в список: если элемент отсутствует в списке, возвращается `True`, в противном случае — `False`. Примеры:

```
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5]   # Проверка на вхождение
    (True, False)
>>> 2 not in [1, 2, 3, 4], 6 not in [1, 2, 3, 4] # Проверка на невхождение
    (False, True)
```

Чтобы узнать индекс элемента с заданным значением в текущем списке, следует воспользоваться методом `index()`. Формат метода:

```
index(<Значение>[, <Начало>[, <Конец>]])
```

В параметре `<Начало>` указывается индекс элемента, с которого начнется поиск. Если он не задан, поиск начнется с начала списка. В параметре `<Конец>` можно задать индекс элемента, на котором закончится поиск. Если этот параметр не указан, поиск будет вестись до конца списка.

Метод `index()` возвращает индекс элемента, имеющего указанное значение. Если значение не входит в список, то возбуждается исключение `ValueError`. Примеры:

```
>>> arr = [1, 2, 1, 2, 1]
>>> arr.index(1), arr.index(2)
    (0, 1)
>>> arr.index(1, 1), arr.index(1, 3, 5)
    (2, 4)
>>> arr.index(3)
    Traceback (most recent call last):
      File "<pyshell#16>", line 1, in <module>
        arr.index(3)
    ValueError: 3 is not in list
```

Узнать количество элементов списка с указанным значением позволяет метод `count(<Значение>)`:

```
>>> arr = [1, 2, 1, 2, 1]
>>> arr.count(1), arr.count(2)
    (3, 2)
>>> arr.count(3)                    # Элемент не входит в список
    0
```

С помощью функций `max()` и `min()`, описанных в *разд. 5.2*, можно узнать максимальное и минимальное значения из всех, что входят в список, соответственно:

```
>>> arr = [1, 2, 3, 4, 5]
>>> max(arr), min(arr)
    (5, 1)
```

Функция `any(<Последовательность>)` возвращает значение `True`, если в заданной последовательности существует хотя бы один элемент, который в логическом контексте возвращает значение `True`. Если последовательность не содержит элементов, возвращается значение `False`. Пример:

```
>>> any([0, None]), any([0, None, 1]), any([])
    (False, True, False)
```

Функция `all(<Последовательность>)` возвращает значение `True`, если все элементы заданной последовательности в логическом контексте возвращают `True`, или последовательность не содержит элементов:

```
>>> all([0, None]), all([0, None, 1]), all([]), all(["str", 10])
    (False, False, True, True)
```

8.9. Переворачивание и перемешивание списка

Метод `reverse()` изменяет порядок следования элементов текущего списка на противоположный. Результата не возвращает. Пример:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.reverse()                # Изменяется текущий список
>>> arr
    [5, 4, 3, 2, 1]
```

Если необходимо изменить порядок следования и получить новый список, следует воспользоваться функцией `reversed(<Последовательность>)`. Она возвращает итератор, который можно преобразовать в список с помощью функции `list()` или генератора списков. Примеры:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> reversed(arr)
    <list_reverseiterator object at 0x00FD5150>
>>> list(reversed(arr))              # Использование функции list()
    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> for i in reversed(arr): print(i, end=" ") # Вывод с помощью цикла
...
    10 9 8 7 6 5 4 3 2 1
>>> [i for i in reversed(arr)] # Использование генератора списков
    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Функция `shuffle(<Список>)` из модуля `random` перемешивает элементы списка случайным образом. Результата не возвращает. Примеры:

```
>>> import random          # Подключаем модуль random
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.shuffle(arr)    # Перемешиваем список случайным образом
>>> arr
    [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
```

> ***Примечание***
>
> Второй параметр функции `shuffle()` в Python 3.9 объявлен устаревшим и нерекомендованным к применению. Его поддержка будет удалена в Python 3.11.

8.10. Выбор элементов списка случайным образом

Выбрать элементы из списка случайным образом позволяют следующие функции из модуля `random`:

- `choice(<Последовательность>)` — возвращает случайный элемент из заданной последовательности, которая может быть любого типа (строкой, списком, кортежем и др.):

```
>>> import random # Подключаем модуль random
>>> random.choice(["s", "t", "r"]) # Список
    's'
```

- `sample()` — возвращает список из указанного количества элементов заданной последовательности, выбранных случайным образом. Формат вызова:

```
sample(<Последовательность>, <Количество элементов>[,
                             counts=<Количество повторений элементов>])
```

Примеры:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 2)
    [7, 10]
```

```
>>> arr                          # Сам список не изменяется
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Начиная с Python 3.9, поддерживается именованный параметр `counts`, в котором можно указать последовательность, содержащую значения количества повторений элементов из последовательности, заданной первым параметром. Первый элемент последовательности из параметра `counts` укажет количество повторений первого элемента из первой последовательности, второй элемент — количество повторений второго элемента и т. д. Например, следующие два вызова функции `sample()` эквивалентны:

```
random.sample([1, 1, 1, 2, 2, 3, 3, 3, 3], 5)
random.sample([1, 2, 3], 5, counts=[3, 2, 4])
```

Также, начиная с Python 3.9, указывать множества в первом параметре функции `sample()` не допускается — эта возможность объявлена устаревшей и подлежащей удалению в будущих версиях языка;

- `choices()` — возвращает список из указанного количества элементов заданной последовательности, выбранных случайным образом на основе заданных весовых коэффициентов. Формат вызова:

```
choices(<Последовательность>[, <Весовые коэффициенты>][, k=1])
```

Количество элементов в результирующем списке указывается в параметре `k`.

Весовые коэффициенты задаются в виде последовательности чисел. Чем больше весовой коэффициент, тем выше вероятность, что соответствующий элемент последовательности из первого параметра попадет в результирующий список. Однако следует учесть, что элементы с высокими коэффициентами могут быть включены в результат по нескольку раз. Пример:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> weights = [10, 20, 10, 40, 30, 50, 100, 40, 10, 20]
>>> random.choices(arr, weights, k=4)
    [8, 7, 5, 7]
```

Если весовые коэффициенты не указаны, вероятность попадания каждого элемента заданной последовательности в результирующий список одинакова:

```
>>> random.choices(arr, k=4)
    [7, 5, 6, 3]
```

8.11. Сортировка списка

Отсортировать текущий список позволяет метод `sort()`. Он имеет следующий формат:

```
sort([key=None][, reverse=False])
```

Метод ничего не возвращает. Отсортируем список по возрастанию значений:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort()                          # Изменяет текущий список
>>> arr
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Чтобы отсортировать список по убыванию, следует в параметре `reverse` указать значение `True`:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort(reverse=True)                # Сортировка по убыванию
>>> arr
    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

В параметре `key` можно указать функцию, выполняющую какое-либо действие над каждым элементом списка. В качестве единственного параметра она должна принимать очередной элемент списка и возвращать результат действий над ним. Этот результат будет участвовать в процессе сортировки, но значения самих элементов списка не изменятся.

Стандартная сортировка зависит от регистра символов (листинг 8.7).

Листинг 8.7. Стандартная сортировка

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort()
for i in arr:
    print(i, end=" ")
# Результат выполнения: Единица2 Единый единица1
```

Выполнив пример из листинга 8.7, мы получили неправильный результат сортировки, ведь `Единый` и `Единица2` больше `единица1`. Чтобы регистр символов не учитывался, в параметре `key` мы укажем функцию для изменения регистра символов (листинг 8.8).

Листинг 8.8. Пользовательская сортировка

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=str.lower)                # Указываем метод lower()
for i in arr:
    print(i, end=" ")
# Результат выполнения: единица1 Единица2 Единый
```

Метод `sort()` сортирует сам текущий список и не возвращает результата. Если необходимо получить отсортированный список, а текущий оставить без изменений, следует воспользоваться функцией `sorted()`. Функция имеет следующий формат:

```
sorted(<Последовательность>[, key=None][, reverse=False])
```

В первом параметре указывается список, который необходимо отсортировать. Остальные параметры эквивалентны параметрам метода `sort()`. Пример:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> sorted(arr)              # Возвращает новый список!
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> sorted(arr, reverse=True) # Возвращает новый список!
    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> arr = ["единица1", "Единый", "Единица2"]
>>> sorted(arr, key=str.lower)
    ['единица1', 'Единица2', 'Единый']
```

8.12. Заполнение списка числами

Создать список, содержащий диапазон чисел, можно, вызвав функцию `range()` и преобразовав возвращенный ею диапазон в список вызовом функции `list()`. Функция `range()` имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение, а если он не указан, используется значение 0. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый диапазон. Если параметр `<Шаг>` не указан, используется значение 1. В качестве примера заполним список числами от 0 до 10:

```
>>> list(range(11))
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Создадим список, состоящий из диапазона чисел от 1 до 15:

```
>>> list(range(1, 16))
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Изменим порядок следования чисел на противоположный:

```
>>> list(range(15, 0, -1))
    [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Список со случайными числами (или случайными элементами из другого списка) можно получить вызовом функции `sample()` (см. *разд. 8.10*):

```
>>> import random
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 3)
    [1, 9, 5]
>>> random.sample(range(300), 5)
    [259, 294, 142, 292, 245]
```

8.13. Преобразование списка в строку

Преобразовать список в строку позволяет метод `join()`. Элементы добавляются в формируемую строку через указанный разделитель. Формат метода:

```
<Разделитель>.join(<Последовательность>)
```

Пример:

```
>>> arr = ["word1", "word2", "word3"]
>>> " – ".join(arr)
    'word1 – word2 – word3'
```

Все элементы списка должны быть строками, иначе генерируется исключение `TypeError`:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> " – ".join(arr)
    Traceback (most recent call last):
      File "<pyshell#69>", line 1, in <module>
        " – ".join(arr)
    TypeError: sequence item 3: expected str instance, int found
```

Избежать этого исключения можно с помощью выражения-генератора, внутри которого текущий элемент списка преобразуется в строку с помощью функции `str()`:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> " – ".join( ( str(i) for i in arr ) )
    'word1 – word2 – word3 – 2'
```

8.14. Кортежи

Кортеж — это неизменяемая пронумерованная последовательность, фактически — неизменяемый список.

Создать кортеж можно следующими способами:

- с помощью функции `tuple([<Последовательность>])`. Функция возвращает кортеж, составленный из элементов заданной последовательности. Если параметр не указан, выдается пустой кортеж. Примеры:

```
>>> tuple()                 # Создаем пустой кортеж
    ()
>>> tuple("String")         # Преобразуем строку в кортеж
    ('S', 't', 'r', 'i', 'n', 'g')
>>> tuple([1, 2, 3, 4, 5]) # Преобразуем список в кортеж
    (1, 2, 3, 4, 5)
```

- приведя элементы создаваемого кортежа через запятую внутри круглых скобок (или без скобок):

```
>>> t1 = ()                  # Создаем пустой кортеж
>>> t2 = (5,)                # Создаем кортеж из одного элемента
>>> t3 = (1, "str", (3, 4)) # Кортеж из трех элементов
>>> t4 = 1, "str", (3, 4)   # Кортеж из трех элементов
>>> t1, t2, t3, t4
    ((), (5,), (1, 'str', (3, 4)), (1, 'str', (3, 4)))
```

Обратите особое внимание на вторую строку примера. Чтобы создать кортеж из одного элемента, необходимо в конце указать запятую. Именно запятые формируют кортеж, а не круглые скобки. Если внутри круглых скобок нет запятых, будет создан объект другого типа:

```
>>> t = (5); type(t)       # Получили число, а не кортеж!
    <class 'int'>
>>> t = ("str"); type(t)   # Получили строку, а не кортеж!
    <class 'str'>
```

Четвертая строка в исходном примере также доказывает, что не скобки формируют кортеж, а запятые. Помните, что для создания кортежа необходимо указать запятые.

Как и списки, кортежи поддерживают обращение к элементу по индексу, получение среза, объединение (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`) и невхождение (оператор `not in`):

```
>>> t = (1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> t[0]                    # Получаем значение первого элемента кортежа
    1
```

```
>>> t[::-1]                    # Изменяем порядок следования элементов
    (9, 8, 7, 6, 5, 4, 3, 2, 1)
>>> t[2:5]                     # Получаем срез
    (3, 4, 5)
>>> 8 in t, 0 in t             # Проверка на вхождение
    (True, False)
>>> (1, 2, 3) * 3              # Повторение
    (1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> (1, 2, 3) + (4, 5, 6)      # Конкатенация
    (1, 2, 3, 4, 5, 6)
>>> 8 not in t, 0 not in t     # Проверка на вхождение
    (False, True)
```

Изменять элементы кортежа нельзя:

```
>>> t = (1, 2, 3)          # Создаем кортеж
>>> t[0] = 50              # Изменить элемент по индексу нельзя!
    Traceback (most recent call last):
      File "<pyshell#95>", line 1, in <module>
        t[0] = 50             # Изменить элемент по индексу нельзя!
    TypeError: 'tuple' object does not support item assignment
```

Кортежи поддерживают уже знакомые нам по спискам функции `len()`, `min()`, `max()`, методы `index()` и `count()`:

```
>>> t = (1, 2, 3)          # Создаем кортеж
>>> len(t)                 # Получаем размер
    3
>>> t = (1, 2, 1, 2, 1)
>>> t.index(1), t.index(2) # Ищем элементы в кортеже
    (0, 1)
```

Будучи неизменяемыми, кортежи занимают меньше оперативной памяти и обрабатываются быстрее, чем изменяемые списки. Поэтому по возможности рекомендуется предпочитать кортежи спискам.

8.15. Множества, изменяемые и неизменяемые

Множество — это изменяемая непронумерованная последовательность, хранящая только уникальные значения (повторяющиеся значения при попытке добавить их в множество отбрасываются).

Создать множество из элементов указанной последовательности можно с помощью функции `set(<Последовательность>)`. Если параметр не указан, создается пустое множество. Примеры:

```
>>> s = set()                          # Создаем пустое множество
>>> s
    set([])
>>> set("string")                      # Преобразуем строку в множество
    set(['g', 'i', 'n', 's', 'r', 't'])
>>> set([1, 2, 3, 4, 5])               # Преобразуем список в множество
    set([1, 2, 3, 4, 5])
```

```
>>> set((1, 2, 3, 4, 5))              # Преобразуем кортеж в множество
    set([1, 2, 3, 4, 5])
>>> set([1, 2, 3, 1, 2, 3])           # Остаются только уникальные элементы
    set([1, 2, 3])
```

Перебрать элементы множества позволяет цикл перебора последовательности (см. *разд. 4.4*):

```
>>> for i in set([1, 2, 3]): print i
    1 2 3
```

Получить размер множества позволяет функция `len()`:

```
>>> len(set([1, 2, 3]))
    3
```

Поскольку множество является непронумерованной последовательностью, получить значение его элемента по индексу, изменить значение элемента по индексу и извлечь срез у множества нельзя — это приведет к генерированию исключения `TypeError`:

```
>>> s = set([1, 2, 3])
>>> s[1]
    Traceback (most recent call last):
      File "<pyshell#51>", line 1, in <module>
        s[1:2]
    TypeError: 'set' object is not subscriptable
```

Для работы с множествами предназначены следующие операторы и соответствующие им методы, поддерживаемые объектом множества:

- `|` и `union()` — объединяют два множества:

```
>>> s = set([1, 2, 3])
>>> s.union(set([4, 5, 6])), s | set([4, 5, 6])
    (set([1, 2, 3, 4, 5, 6]), set([1, 2, 3, 4, 5, 6]))
>>> set([1, 2, 3]) | set([2, 3, 4])
    set([1, 2, 3, 4])
```

- `a |= b` и `a.update(b)` — добавляют элементы множества `b` во множество `a`:

```
>>> s = set([1, 2, 3])
>>> s.update(set([4, 5, 6]))
>>> s
    set([1, 2, 3, 4, 5, 6])
>>> s |= set([7, 8, 9])
>>> s
    set([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- `-` и `difference()` — вычисляют разницу множеств:

```
>>> set([1, 2, 3]) - set([1, 2, 4])
    set([3])
>>> s = set([1, 2, 3])
>>> s.difference(set([1, 2, 4]))
    set([3])
```

- `a -= b` и `a.difference_update(b)` — удаляют из множества `a` элементы, которые существуют в обоих исходных множествах:

```
>>> s = set([1, 2, 3])
>>> s.difference_update(set([1, 2, 4]))
>>> s
    set([3])
>>> s -= set([3, 4, 5])
>>> s
    set([])
```

- `&` и `intersection()` — возвращают множество с элементами, которые существуют в обоих исходных множествах:

```
>>> set([1, 2, 3]) & set([1, 2, 4])
    set([1, 2])
>>> s = set([1, 2, 3])
>>> s.intersection(set([1, 2, 4]))
    set([1, 2])
```

- `a &= b` и `a.intersection_update(b)` — оставляют во множестве `a` элементы, которые существуют в обоих исходных множествах:

```
>>> s = set([1, 2, 3])
>>> s.intersection_update(set([1, 2, 4]))
>>> s
    set([1, 2])
>>> s &= set([1, 6, 7])
>>> s
    set([1])
```

- `^` и `symmetric_difference()` — возвращают множество с элементами, которые существуют только в одном из исходных множеств:

```
>>> s = set([1, 2, 3])
>>> s ^ set([1, 2, 4]), s.symmetric_difference(set([1, 2, 4]))
    (set([3, 4]), set([3, 4]))
>>> s ^ set([1, 2, 3]), s.symmetric_difference(set([1, 2, 3]))
    (set([]), set([]))
>>> s ^ set([4, 5, 6]), s.symmetric_difference(set([4, 5, 6]))
    (set([1, 2, 3, 4, 5, 6]), set([1, 2, 3, 4, 5, 6]))
```

- `a ^= b` и `a.symmetric_difference_update(b)` — оставляют во множестве `a` элементы, присутствующие только в одном из исходных множеств:

```
>>> s = set([1, 2, 3])
>>> s.symmetric_difference_update(set([1, 2, 4]))
>>> s
    set([3, 4])
>>> s ^= set([3, 5, 6])
>>> s
    set([4, 5, 6])
```

Операторы сравнения множеств:

- `in` — проверка наличия элемента во множестве:

```
>>> s = set([1, 2, 3, 4, 5])
>>> 1 in s, 12 in s
    (True, False)
```

- `not in` — проверка отсутствия элемента во множестве:

```
>>> s = set([1, 2, 3, 4, 5])
>>> 1 in s, 12 in s
    (False, True)
```

- `==` — проверка на равенство множеств:

```
>>> set([1, 2, 3]) == set([1, 2, 3])
    True
>>> set([1, 2, 3]) == set([3, 2, 1])
    True
>>> set([1, 2, 3]) == set([1, 2, 3, 4])
    False
```

- `a <= b` и `a.issubset(b)` — проверяют, входят ли все элементы множества `a` во множество `b`:

```
>>> s = set([1, 2, 3])
>>> s <= set([1, 2]), s <= set([1, 2, 3, 4])
    (False, True)
>>> s.issubset(set([1, 2])), s.issubset(set([1, 2, 3, 4]))
    (False, True)
```

- `a < b` — проверяет, входят ли все элементы множества `a` во множество `b`, причем множество `a` не должно быть равно множеству `b`:

```
>>> s = set([1, 2, 3])
>>> s < set([1, 2, 3]), s < set([1, 2, 3, 4])
    (False, True)
```

- `a >= b` и `a.issuperset(b)` — проверяют, входят ли все элементы множества `b` во множество `a`:

```
>>> s = set([1, 2, 3])
>>> s >= set([1, 2]), s >= set([1, 2, 3, 4])
    (True, False)
>>> s.issuperset(set([1, 2])), s.issuperset(set([1, 2, 3, 4]))
    (True, False)
```

- `a > b` — проверяет, входят ли все элементы множества `b` во множество `a`, причем множество `a` не должно быть равно множеству `b`:

```
>>> s = set([1, 2, 3])
>>> s > set([1, 2]), s > set([1, 2, 3])
    (True, False)
```

- `a.isdisjoint(b)` — проверяет, являются ли множества `a` и `b` полностью разными, т. е. не содержащими ни одного совпадающего элемента:

```
>>> s = set([1, 2, 3])
>>> s.isdisjoint(set([4, 5, 6]))
    True
>>> s.isdisjoint(set([1, 3, 5]))
    False
```

Для работы с множествами предназначены следующие методы:

- `copy()` — возвращает копию текущего множества:

```
>>> s = set([1, 2, 3])
>>> c = s; s is c # С помощью оператора = создать копию нельзя!
    True
>>> c = s.copy()  # Создаем копию множества
>>> c
    set([1, 2, 3])
>>> s is c        # Теперь это разные объекты
    False
```

- `add(<Элемент>)` — добавляет `<Элемент>` в текущее множество:

```
>>> s = set([1, 2, 3])
>>> s.add(4); s
    set([1, 2, 3, 4])
```

- `remove(<Элемент>)` — удаляет `<Элемент>` из текущего множества. Если элемент не найден, то возбуждается исключение `KeyError`. Примеры:

```
>>> s = set([1, 2, 3])
>>> s.remove(3); s            # Элемент существует
    set([1, 2])
>>> s.remove(5)               # Элемент НЕ существует
    Traceback (most recent call last):
      File "<pyshell#78>", line 1, in <module>
        s.remove(5)           # Элемент НЕ существует
    KeyError: 5
```

- `discard(<Элемент>)` — удаляет `<Элемент>` из текущего множества, если он там присутствует. Если указанный элемент не существует, ничего не делает. Примеры:

```
>>> s = set([1, 2, 3])
>>> s.discard(3); s       # Элемент существует
    set([1, 2])
>>> s.discard(5); s       # Элемент НЕ существует
    set([1, 2])
```

- `pop()` — удаляет произвольный элемент из текущего множества и возвращает его. Если множество пустое, возбуждается исключение `KeyError`. Примеры:

```
>>> s = set([1, 2])
>>> s.pop(), s
    (1, set([2]))
>>> s.pop(), s
    (2, set([]))
>>> s.pop() # Если нет элементов, то будет ошибка
    Traceback (most recent call last):
      File "<pyshell#89>", line 1, in <module>
        s.pop() # Если нет элементов, то будет ошибка
    KeyError: 'pop from an empty set'
```

- `clear()` — удаляет все элементы из текущего множества:

```
>>> s = set([1, 2, 3])
>>> s.clear(); s
    set([])
```

Помимо генераторов списков, Python поддерживает аналогичные им *генераторы множеств*. Формат записи генератора множества:

```
{ <Выражение> for <Переменная> in <Исходная последовательность> }
```

В результирующее множество попадут только уникальные элементы. Пример:

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3]}
    {1, 2, 3}
```

Пример создания множества, содержащего лишь четные числа:

```
>>> {x for x in [1, 2, 1, 5, 12, 2, 3] if x % 2 == 0}
    {2, 12}
```

Неизменяемые множества аналогичны обычным, изменяемым, за тем исключением, что их содержимое невозможно изменить.

Создать неизменяемое множество можно с помощью функции `frozenset()`. Формат ее вызова аналогичен таковому у функции `set()`. Примеры:

```
>>> f = frozenset()
>>> f
    frozenset([])
>>> frozenset("string")                    # Преобразуем строку
    frozenset(['g', 'i', 'n', 's', 'r', 't'])
>>> frozenset([1, 2, 3, 4, 4])             # Преобразуем список
    frozenset([1, 2, 3, 4])
>>> frozenset((1, 2, 3, 4, 4))             # Преобразуем кортеж
    frozenset([1, 2, 3, 4])
```

Неизменяемые множества поддерживают все операторы, которые не изменяют содержимое множества, а также следующие методы: `copy()`, `difference()`, `intersection()`, `issubset()`, `issuperset()`, `symmetric_difference()` и `union()`.

8.16. Диапазоны

Диапазон — это неизменяемая пронумерованная последовательность целых чисел, расположенных между заданными начальным и конечным значениями и отличающихся друг от друга на значение указанного шага.

Для создания диапазона применяется функция `range()`:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение — если он не указан, используется значение 0. Во втором параметре указывается конечное значение. Следует заметить, что это значение не войдет в создаваемый диапазон. Если параметр <Шаг> не указан, используется значение 1. Примеры:

```
>>> r = range(1, 10)
>>> for i in r: print(i, end = " ")
...
    1 2 3 4 5 6 7 8 9
>>> r = range(10, 110, 10)
>>> for i in r: print(i, end = " ")
...
    10 20 30 40 50 60 70 80 90 100
>>> r = range(10, 1, -1)
>>> for i in r: print(i, end = " ")
...
    10 9 8 7 6 5 4 3 2
```

Преобразовать диапазон в список, кортеж, обычное или неизменяемое множество можно с помощью функции `list()`, `tuple()`, `set()` или `frozenset()` соответственно:

```
>>> list(range(1, 10))            # Преобразуем в список
    [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(1, 10))           # Преобразуем в кортеж
    (1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> set(range(1, 10))             # Преобразуем в множество
    {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Диапазоны поддерживают доступ к элементу по индексу, получение среза (в результате возвращается также диапазон), проверку на вхождение и невхождение, функции `len()`, `min()`, `max()`, методы `index()` и `count()`:

```
>>> r = range(1, 10)
>>> r[2], r[-1]
    (3, 9)
>>> r[2:4]
    range(3, 5)
>>> 2 in r, 12 in r
    (True, False)
>>> 3 not in r, 13 not in r
    (False, True)
>>> len(r), min(r), max(r)
    (9, 1, 9)
>>> r.index(4), r.count(4)
    (3, 1)
```

Поддерживается ряд операторов, позволяющих сравнить два диапазона:

- `==` — возвращает `True`, если диапазоны равны, и `False` — в противном случае. Диапазоны считаются равными, если они содержат одинаковые последовательности чисел. Примеры:

```
>>> range(1, 10) == range(1, 10, 1)
    True
>>> range(1, 10, 2) == range(1, 11, 2)
    True
>>> range(1, 10, 2) == range(1, 12, 2)
    False
```

- `!=` — возвращает `True`, если диапазоны не равны, и `False` — в противном случае:

```
>>> range(1, 10, 2) != range(1, 12, 2)
    True
>>> range(1, 10) != range(1, 10, 1)
    False
```

Также диапазоны поддерживают атрибуты `start`, `stop` и `step`, возвращающие соответственно начальную, конечную границы диапазона и его шаг:

```
>>> r = range(1, 10)
>>> r.start, r.stop, r.step
    (1, 10, 1)
```

Важнейшим преимуществом диапазонов перед другими последовательностями является их компактность — вне зависимости от количества входящих в диапазон элементов, он всегда занимает один и тот же объем оперативной памяти (поскольку фактически представляет собой функцию, при каждом вызове выдающую очередное число). Однако в диапазон могут входить лишь целые числа, последовательно стоящие друг за другом, — сформировать диапазон на основе произвольного набора чисел или значений другого типа (даже вещественных чисел) невозможно.

Диапазоны чаще всего используются для проверки вхождения значения в какой-либо интервал и для организации циклов.

8.17. Модуль *itertools*

Модуль `itertools` содержит функции, позволяющие генерировать различные последовательности на основе других последовательностей, производить фильтрацию элементов и др. Все функции возвращают итераторы. Прежде чем использовать эти функции, необходимо подключить модуль с помощью инструкции:

```
import itertools
```

8.17.1. Генерирование неопределенного количества значений

Для генерирования неопределенного количества значений предназначены следующие функции:

- `count([start=0][, step=1])` — возвращает бесконечно нарастающую последовательность значений. Начальное значение задается параметром `start`, а шаг — параметром `step`. В качестве значений параметров можно указать вещественные числа. Примеры:

```
>>> import itertools
>>> for i in itertools.count():
...     if i > 10: break
...     print(i, end=" ")
...
    0 1 2 3 4 5 6 7 8 9 10
>>> list(zip(itertools.count(), "абвгд"))
    [(0, 'а'), (1, 'б'), (2, 'в'), (3, 'г'), (4, 'д')]
```

```
>>> list(zip(itertools.count(start=2, step=2), "абвгд"))
    [(2, 'а'), (4, 'б'), (6, 'в'), (8, 'г'), (10, 'д')]
>>> for i in itertools.count(start=0.5, step=1.5):
...     if i > 10: break
...     print(i, end=" ")
...
    0.5 2.0 3.5 5.0 6.5 8.0 9.5
```

- `cycle(<Последовательность>)` — на каждой итерации возвращает очередной элемент заданной последовательности. Когда будет достигнут конец последовательности, перебор начнется сначала, и так до бесконечности. Примеры:

```
>>> n = 1
>>> for i in itertools.cycle("абв"):
...     if n > 10: break
...     print(i, end=" ")
...     n += 1
...
    а б в а б в а б в а
>>> list(zip(itertools.cycle([0, 1]), "абвгд"))
    [(0, 'а'), (1, 'б'), (0, 'в'), (1, 'г'), (0, 'д')]
```

- `repeat(<Значение>[, <Количество>])` — возвращает заданное значение указанное количество раз. Если количество не указано, значение возвращается бесконечно. Примеры:

```
>>> list(itertools.repeat(1, 10))
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> list(zip(itertools.repeat(5), "абвгд"))
    [(5, 'а'), (5, 'б'), (5, 'в'), (5, 'г'), (5, 'д')]
```

8.17.2. Генерирование комбинаций значений

Получить различные комбинации значений позволяют следующие функции:

- `combinations()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества гарантированно разных элементов заданной последовательности. Комбинации элементов в выдаваемых кортежах не повторяются. Формат функции:

  `combinations(<Последовательность>, <Количество элементов>)`

  Примеры:

```
>>> import itertools
>>> list(itertools.combinations('абвг', 2))
    [('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'в'), ('б', 'г'),
     ('в', 'г')]
>>> ["".join(i) for i in itertools.combinations('абвг', 2)]
    ['аб', 'ав', 'аг', 'бв', 'бг', 'вг']
>>> list(itertools.combinations('вгаб', 2))
    [('в', 'г'), ('в', 'а'), ('в', 'б'), ('г', 'а'), ('г', 'б'),
     ('а', 'б')]
>>> list(itertools.combinations('абвг', 3))
    [('а', 'б', 'в'), ('а', 'б', 'г'), ('а', 'в', 'г'),
     ('б', 'в', 'г')]
```

- `combinations_with_replacement()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов заданной последовательности. Выдаваемые кортежи могут содержать одинаковые элементы. Комбинации элементов в выдаваемых кортежах не повторяются. Формат функции:

```
combinations_with_replacement(<Последовательность>,
                              <Количество элементов>)
```

Примеры:

```
>>> list(itertools.combinations_with_replacement('абвг', 2))
    [('а', 'а'), ('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'б'),
     ('б', 'в'), ('б', 'г'), ('в', 'в'), ('в', 'г'), ('г', 'г')]
>>> list(itertools.combinations_with_replacement('вгаб', 2))
    [('в', 'в'), ('в', 'г'), ('в', 'а'), ('в', 'б'), ('г', 'г'),
     ('г', 'а'), ('г', 'б'), ('а', 'а'), ('а', 'б'), ('б', 'б')]
```

- `permutations()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов заданной последовательности. Комбинации элементов в выдаваемых кортежах будут повторяться. Если количество элементов не указано, то используется размер последовательности. Формат функции:

```
permutations(<Последовательность>[, <Количество элементов>])
```

Примеры:

```
>>> list(itertools.permutations('абвг', 2))
    [('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'а'), ('б', 'в'),
     ('б', 'г'), ('в', 'а'), ('в', 'б'), ('в', 'г'), ('г', 'а'),
     ('г', 'б'), ('г', 'в')]
>>> ["".join(i) for i in itertools.permutations('абвг')]
    ['абвг', 'абгв', 'авбг', 'авгб', 'агбв', 'агвб', 'бавг',
     'багв', 'бваг', 'бвга', 'бгав', 'бгва', 'вабг', 'вагб',
     'вбаг', 'вбга', 'вгаб', 'вгба', 'габв', 'гавб', 'гбав',
     'гбва', 'гваб', 'гвба']
```

- `pairwise(<Последовательность>)` (начиная с Python 3.10) — на каждой итерации возвращает кортеж, содержащий два соседних элемента указанной последовательности:

```
>>> list(itertools.pairwise("abcdef"))
    [('a', 'b'), ('b', 'c'), ('c', 'd'), ('d', 'e'), ('e', 'f')]
```

- `product()` — на каждой итерации возвращает кортеж, содержащий комбинацию из элементов одной или нескольких указанных последовательностей. Формат функции:

```
product(<Последовательность 1>[, . . ., <Последовательность N>][, repeat=1])
```

Количество элементов в выдаваемых итератором кортежах будет равно количеству заданных последовательностей. Можно указать другое количество элементов в параметре `repeat`, однако оно не должно быть меньше количества указанных последовательностей. Примеры:

```
>>> list(itertools.product('абвг'))
    [('а',), ('б',), ('в',), ('г',)]
>>> list(itertools.product('абвг', repeat=2))
    [('а', 'а'), ('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'а'),
     ('б', 'б'), ('б', 'в'), ('б', 'г'), ('в', 'а'), ('в', 'б'),
```

```
     ('в', 'в'), ('в', 'г'), ('г', 'а'), ('г', 'б'), ('г', 'в'),
     ('г', 'г')]
>>> list(itertools.product('аб', 'вг'))
    [('а', 'в'), ('а', 'г'), ('б', 'в'), ('б', 'г')]
>>> list(itertools.product('аб', 'вг', repeat=3))
    [('а', 'в', 'а', 'в', 'а', 'в'), ('а', 'в', 'а', 'в', 'а', 'г'),
     ('а', 'в', 'а', 'в', 'б', 'в'), ('а', 'в', 'а', 'в', 'б', 'г'),
     ('а', 'в', 'а', 'г', 'а', 'в'), ('а', 'в', 'а', 'г', 'а', 'г'),
     ('а', 'в', 'а', 'г', 'б', 'в'), ('а', 'в', 'а', 'г', 'б', 'г'),
    ... Часть вывода пропущена ...
     ('б', 'г', 'б', 'г', 'б', 'в'), ('б', 'г', 'б', 'г', 'б', 'г')]
```

8.17.3. Фильтрация элементов последовательности

Для фильтрации элементов последовательности предназначены следующие функции:

- `filterfalse(<Функция>, <Последовательность>)` — возвращает элементы заданной последовательности (по одному на каждой итерации), для которых функция, указанная в первом параметре, вернет значение `False`:

```
>>> import itertools
>>> def func(x): return x > 3
...
>>> list(itertools.filterfalse(func, [4, 5, 6, 0, 7, 2, 3]))
    [0, 2, 3]
>>> list(filter(func, [4, 5, 6, 0, 7, 2, 3]))
    [4, 5, 6, 7]
```

  Если вместо функции указать значение `None`, то каждый элемент последовательности будет проверен на соответствие значению `False`. Если элемент в логическом контексте возвращает значение `True`, то он не войдет в возвращаемый результат. Примеры:

```
>>> list(itertools.filterfalse(None, [0, 5, 6, 0, 7, 0, 3]))
    [0, 0, 0]
>>> list(filter(None, [0, 5, 6, 0, 7, 0, 3]))
    [5, 6, 7, 3]
```

- `dropwhile(<Функция>, <Последовательность>)` — возвращает элементы заданной последовательности (по одному на каждой итерации), начиная с элемента, для которого функция, указанная в первом параметре, вернет значение `False`:

```
>>> def func(x): return x > 3
...
>>> list(itertools.dropwhile(func, [4, 5, 6, 0, 7, 2, 3]))
    [0, 7, 2, 3]
>>> list(itertools.dropwhile(func, [4, 5, 6, 7, 8]))
    []
>>> list(itertools.dropwhile(func, [1, 2, 4, 5, 6, 7, 8]))
    [1, 2, 4, 5, 6, 7, 8]
```

- `takewhile(<Функция>, <Последовательность>)` — возвращает элементы заданной последовательности (по одному на каждой итерации), пока не встретится элемент, для которого функция, указанная в первом параметре, вернет значение `False`:

```
>>> def func(x): return x > 3
...
>>> list(itertools.takewhile(func, [4, 5, 6, 0, 7, 2, 3]))
    [4, 5, 6]
>>> list(itertools.takewhile(func, [4, 5, 6, 7, 8]))
    [4, 5, 6, 7, 8]
>>> list(itertools.takewhile(func, [1, 2, 4, 5, 6, 7, 8]))
    []
```

- `compress()` — возвращает элементы указанной фильтруемой последовательности, у которых соответствующие элементы из последовательности логических значений трактуются как `True`, пока не закончится какая-либо из указанных последовательностей. Формат функции:

```
compress(<Фильтруемая последовательность>,
         <Последовательность логических значений>)
```

Примеры:

```
>>> list(itertools.compress('абвгде', [1, 0, 0, 0, 1, 1]))
    ['а', 'д', 'е']
>>> list(itertools.compress('абвгде', [True, False, True]))
    ['а', 'в']
```

8.17.4. Прочие функции

Модуль `itertools` содержит несколько дополнительных функций:

- `islice()` — на каждой итерации возвращает очередной элемент заданной последовательности. Поддерживаются форматы:

```
islice(<Последовательность>, <Конечная граница>)
```

и

```
islice(<Последовательность>, <Начальная граница>, <Конечная граница>[, <Шаг>])
```

Параметры `<Начальная граница>` и `<Конечная граница>` задают начальный и конечный индексы выдаваемых элементов, параметр `<Шаг>` — промежуток между индексами выдаваемых элементов. Элемент с указанным конечным индексом в состав выдаваемых не войдет. Если шаг не указан, будет использовано значение 1. Примеры:

```
>>> list(itertools.islice("абвгдежз", 3))
    ['а', 'б', 'в']
>>> list(itertools.islice("абвгдежз", 3, 6))
    ['г', 'д', 'е']
>>> list(itertools.islice("абвгдежз", 3, 6, 2))
    ['г', 'е']
```

- `starmap(<Функция>, <Последовательность>)` — формирует новую последовательность на основании значений, возвращенных указанной функцией. Исходная последовательность должна содержать в качестве элементов кортежи — именно на основе их элементов указанная функция и станет вычислять значения, которые войдут в генерируемую последовательность. Примеры суммирования значений:

```
>>> import itertools
>>> def func1(x, y): return x + y
```

```
>>> list(itertools.starmap(func1, [(1, 2), (4, 5), (6, 7)]))
    [3, 9, 13]
>>> def func2(x, y, z): return x + y + z
...
>>> list(itertools.starmap(func2, [(1, 2, 3), (4, 5, 6)]))
    [6, 15]
```

- `zip_longest()` — на каждой итерации возвращает кортеж, содержащий элементы заданных последовательностей с одинаковыми индексами. Если последовательности имеют разное количество элементов, вместо отсутствующего элемента вставляется значение из параметра `fillvalue`. Формат функции:

```
zip_longest(<Последовательность 1>[, . . ., <Последовательность N>][,
                                        fillvalue=None])
```

Примеры:

```
>>> list(itertools.zip_longest([1, 2, 3], [4, 5, 6]))
    [(1, 4), (2, 5), (3, 6)]
>>> list(itertools.zip_longest([1, 2, 3], [4]))
    [(1, 4), (2, None), (3, None)]
>>> list(itertools.zip_longest([1, 2, 3], [4], fillvalue=0))
    [(1, 4), (2, 0), (3, 0)]
>>> list(zip([1, 2, 3], [4]))
    [(1, 4)]
```

- `accumulate()` — на каждой итерации возвращает результат, полученный выполнением определенного действия над текущим элементом заданной последовательности и результатом, вычисленным на предыдущей итерации. Формат функции:

```
accumulate(<Последовательность>[, <Функция>][, initial=None])
```

Необходимое действие над элементами последовательности выполняется заданной функцией, а если она не указана, выполняется сложение. Заданная функция должна принимать два параметра и возвращать результат. На первой итерации всегда возвращается первый элемент переданной последовательности. Пример:

```
>>> # Выполняем сложение
>>> list(itertools.accumulate([1, 2, 3, 4, 5, 6]))
    [1, 3, 6, 10, 15, 21]
>>> # [1, 1+2, 3+3, 6+4, 10+5, 15+6]
>>> # Выполняем умножение
>>> def func(x, y): return x * y
...
>>> list(itertools.accumulate([1, 2, 3, 4, 5, 6], func))
    [1, 2, 6, 24, 120, 720]
>>> # [1, 1*2, 2*3, 6*4, 24*5, 120*6]
```

Начиная с Python 3.8, поддерживается параметр `initial`, задающий начальное значение для выполнения действия. Если он указан, на первой итерации функция `accumulate()` выдаст значение этого параметра, на второй — результат выполнения действия со значением этого параметра и первым элементом указанной последовательности, а далее станет работать как обычно. Пример:

```
>>> list(itertools.accumulate([1, 2, 3, 4, 5, 6], initial=100))
    [100, 101, 103, 106, 110, 115, 121]
```

- `chain()` — на каждой итерации возвращает элементы сначала первой из указанных последовательностей, затем второй и т. д. Формат функции:

  ```
  chain(<Последовательность 1>[, . . ., <Последовательность N>])
  ```

  Примеры:

  ```
  >>> arr1, arr2, arr3 = [1, 2, 3], [4, 5], [6, 7, 8, 9]
  >>> list(itertools.chain(arr1, arr2, arr3))
      [1, 2, 3, 4, 5, 6, 7, 8, 9]
  >>> list(itertools.chain("abc", "defg", "hij"))
      ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
  >>> list(itertools.chain("abc", ["defg", "hij"]))
      ['a', 'b', 'c', 'defg', 'hij']
  ```

- `chain.from_iterable(<Последовательность>)` — аналогична функции `chain()`, но принимает одну последовательность, каждый элемент которой также должен представлять собой последовательность:

  ```
  >>> list(itertools.chain.from_iterable(["abc", "defg", "hij"]))
      ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
  ```

- `tee(<Последовательность>[, <Количество>])` — возвращает кортеж, содержащий несколько одинаковых итераторов для заданной последовательности. Если второй параметр не указан, то возвращается кортеж из двух итераторов. Примеры:

  ```
  >>> arr = [1, 2, 3]
  >>> itertools.tee(arr)
      (<itertools.tee object at 0x00FD8760>,
       <itertools.tee object at 0x00FD8738>)
  >>> itertools.tee(arr, 3)
      (<itertools.tee object at 0x00FD8710>,
       <itertools.tee object at 0x00FD87D8>,
       <itertools.tee object at 0x00FD87B0>)
  >>> list(itertools.tee(arr)[0])
      [1, 2, 3]
  >>> list(itertools.tee(arr)[1])
      [1, 2, 3]
  ```

- `groupby(<Последовательность>[, <Функция>])` — на каждой итерации возвращает кортеж из двух элементов: значения, возвращенного указанной функцией, и итератора, выдающего элементы указанной последовательности, для которых указанная функция вернула это значение. Можно сказать, что функция `groupby()` выполняет группировку элементов последовательности по программно сгенерированному ключу.

  Заданная функция должна в качестве параметра принимать очередной элемент последовательности и возвращать сгенерированный на его основе ключ. Задаваемая последовательность уже должна быть отсортирована с применением той же функции, что будет использована для генерирования ключа.

  Пример группировки чисел по их знаку (положительные числа попадут в первую группу, отрицательные — во вторую):

  ```
  >>> def func(x): return "pos" if x > 0 else "neg"
  ```

```
>>> arr = [2, -4, 5, -20, 7, 1, -4]
>>> arr.sort(key=func)
>>> for i, j in itertools.groupby(arr, func):
...     print(str(i) + ": " + str(list(j)))
...
neg: [-4, -20, -4]
pos: [2, 5, 7, 1]
```

Если функция не указана, в качестве ключа будет использовано значение элемента последовательности:

```
>>> for i, j in itertools.groupby(arr):
...     print(str(i) + ": " + str(list(j)), end=" ")
...
-4: [-4] -20: [-20] -4: [-4] 2: [2] 5: [5] 7: [7] 1: [1]
```

ГЛАВА 9

Словари

Словарь — это изменяемый набор значений (*элементов*), каждому из которых дана уникальная пометка, называемая *ключом*. В качестве ключа может выступать произвольное значение любого типа, но практически всегда используются строки. Зная ключ, можно извлечь из словаря соответствующий ему элемент. Количество элементов в словаре называется *размером*.

Словари, а также другие подобные им типы данных называются *отображениями*, поскольку в них каждый ключ отображается на соответствующий ему элемент.

9.1. Создание словаря

Создать словарь можно следующими способами:

- с помощью функции `dict()`. Форматы функции:

```
dict(<Ключ 1>=<Значение 1>[, . . ., <Ключ N>=<Значение N>])
dict(<Список кортежей с двумя элементами: ключом и значением>)
dict(<Список списков с двумя элементами: ключом и значением>)
dict(<Словарь>)
```

Если параметры не указаны, создается пустой словарь. Примеры:

```
>>> d = dict(); d                    # Создаем пустой словарь
    {}
>>> d = dict({"a": 1, "b": 2}); d    # Указан словарь
    {'a': 1, 'b': 2}
>>> d = dict([("a", 1), ("b", 2)]); d # Указан список кортежей
    {'a': 1, 'b': 2}
>>> d = dict([["a", 1], ["b", 2]]); d # Указан список списков
    {'a': 1, 'b': 2}
>>> d = dict(a=1, b=2); d
    {'a': 1, 'b': 2}
```

Объединить два списка в список кортежей позволяет функция `zip()`:

```
>>> k = ["a", "b"]          # Список с ключами
>>> v = [1, 2]              # Список со значениями
>>> list(zip(k, v))         # Создание списка кортежей
    [('a', 1), ('b', 2)]
```

```
>>> d = dict(zip(k, v)); d # Создание словаря
    {'a': 1, 'b': 2}
```

- указав все элементы словаря внутри фигурных скобок — это наиболее часто используемый способ. Между ключом и значением ставится двоеточие, а пары «ключ/значение» записываются через запятую. Примеры:

```
>>> d = {}; d              # Создание пустого словаря
    {}
>>> d = { "a": 1, "b": 2 }; d
    {'a': 1, 'b': 2}
```

- заполнив словарь поэлементно. В этом случае ключ указывается внутри квадратных скобок. Примеры:

```
>>> d = {}             # Создаем пустой словарь
>>> d["a"] = 1         # Добавляем элемент1 (ключ "a")
>>> d["b"] = 2         # Добавляем элемент2 (ключ "b")
>>> d
    {'a': 1, 'b': 2}
```

- с помощью метода `dict.fromkeys(<Последовательность>[, <Значение>])`. Метод создает словарь, ключами которого станут элементы заданной последовательности, а их значениями — значение из второго параметра. Если второй параметр не указан, элементы созданного словаря получат значение `None`. Примеры:

```
>>> d = dict.fromkeys(["a", "b", "c"])
>>> d
    {'a': None, 'c': None, 'b': None}
>>> d = dict.fromkeys(["a", "b", "c"], 0) # Указан список
>>> d
    {'a': 0, 'c': 0, 'b': 0}
>>> d = dict.fromkeys(("a", "b", "c"), 0) # Указан кортеж
>>> d
    {'a': 0, 'c': 0, 'b': 0}
```

Создать поверхностную копию словаря позволяет функция `dict()`:

```
>>> d1 = { "a": 1, "b": 2 } # Создаем словарь
>>> d2 = dict(d1)           # Создаем поверхностную копию
>>> d1 is d2 # Оператор показывает, что это разные объекты
    False
>>> d2["b"] = 10
>>> d1, d2   # Изменилось только значение в переменной d2
    ({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Также можно воспользоваться методом `copy()`:

```
>>> d1 = { "a": 1, "b": 2 } # Создаем словарь
>>> d2 = d1.copy()          # Создаем поверхностную копию
>>> d1 is d2 # Оператор показывает, что это разные объекты
    False
>>> d2["b"] = 10
>>> d1, d2   # Изменилось только значение в переменной d2
    ({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Создать полную копию словаря позволяет функция `deepcopy()` из модуля `copy`:

```
>>> d1 = { "a": 1, "b": [20, 30, 40] }
>>> d2 = dict(d1)          # Создаем поверхностную копию
>>> d2["b"][0] = "test"
>>> d1, d2                 # Изменились значения в двух переменных!!!
    ({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 30, 40]})
>>> import copy
>>> d3 = copy.deepcopy(d1) # Создаем полную копию
>>> d3["b"][1] = 800
>>> d1, d3                 # Изменилось значение только в переменной d3
    ({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 800, 40]})
```

Начиная с Python 3.7, элементы в словаре хранятся строго в том порядке, в котором они были добавлены в него (в предыдущих версиях языка элементы в словаре могли переупорядочиваться интерпретатором).

9.2. Операции над словарями

Обращение к элементам словаря осуществляется с помощью квадратных скобок, в которых указывается ключ. Как говорилось ранее, ключ может быть произвольным значением любого типа: строкой, целым числом, кортежем и др. Однако почти всегда используются строки.

Выведем все элементы словаря:

```
>>> d = { 1: "int", "a": "str", (1, 2): "tuple" }
>>> d[1], d["a"], d[(1, 2)]
    ('int', 'str', 'tuple')
```

Если элемент с указанным ключом отсутствует в словаре, возбуждается исключение `KeyError`:

```
>>> d = { "a": 1, "b": 2 }
>>> d["c"]             # Обращение к несуществующему элементу
    Traceback (most recent call last):
      File "<pyshell#49>", line 1, in <module>
        d["c"]          # Обращение к несуществующему элементу
    KeyError: 'c'
```

Проверить существование ключа в словаре можно с помощью оператора `in`. Если ключ найден, возвращается значение `True`, в противном случае — `False`. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d           # Ключ существует
    True
>>> "c" in d           # Ключ не существует
    False
```

Проверить, отсутствует ли какой-либо ключ в словаре, позволит оператор `not in`. Если ключ отсутствует, возвращается `True`, иначе — `False`. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> "c" not in d       # Ключ не существует
    True
```

```
>>> "a" not in d           # Ключ существует
    False
```

Метод `get(<Ключ>[, <Значение по умолчанию>])` позволяет избежать возбуждения исключения `KeyError` при отсутствии в словаре указанного ключа. Если заданный ключ в текущем словаре существует, метод возвращает соответствующее ему значение, если ключ не существует — `None` или значение из второго параметра. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.get("a"), d.get("c"), d.get("c", 800)
    (1, None, 800)
```

Кроме того, можно воспользоваться методом `setdefault(<Ключ>[, <Значение по умолчанию>])`. Если заданный ключ в текущем словаре существует, метод возвращает соответствующее ему значение, в противном случае в словаре создается новый элемент со значением из второго параметра. Если второй параметр не указан, значением нового элемента будет `None`. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> d.setdefault("a"), d.setdefault("c"), d.setdefault("d", 0)
    (1, None, 0)
>>> d
    {'a': 1, 'c': None, 'b': 2, 'd': 0}
```

Так как словари относятся к изменяемым типам данных, мы можем изменить элемент по ключу. Если элемент с указанным ключом отсутствует в словаре, он будет создан. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d["a"] = 800              # Изменение элемента по ключу
>>> d["c"] = "string"         # Будет добавлен новый элемент
>>> d
    {'a': 800, 'c': 'string', 'b': 2}
```

Получить размер словаря позволяет функция `len()`:

```
>>> d = { "a": 1, "b": 2 }
>>> len(d)
    2
```

Удалить элемент из словаря можно с помощью оператора `del`:

```
>>> d = { "a": 1, "b": 2 }
>>> del d["b"]; d   # Удаляем элемент с ключом "b" и выводим словарь
    {'a': 1}
```

Оператор `a | b` (поддерживается, начиная с Python 3.9) возвращает новый словарь, содержащий все элементы из словарей `a` и `b`. Если оба словаря содержат элементы с одинаковыми ключами, значения этих элементов будут взяты из словаря `b`. Пример:

```
>>> d1 = { "a": 1, "b": 2 }
>>> d2 = { "b": 20, "c": 3 }
>>> d1 | d2
    {'a': 1, 'b': 20, 'c': 3}
```

Оператор `a |= b` (также поддерживается, начиная с Python 3.9) дополняет словарь `a` элементами из словаря `b`. Если оба словаря содержат элементы с одинаковыми ключами, значения этих элементов будут взяты из словаря `b`. Примеры:

```
>>> d1 = { "a": 1, "b": 2 }
>>> d2 = { "b": 20, "c": 3 }
>>> d1 |= d2
>>> d2
    {'b': 20, 'c': 3}
>>> d1
    {'a': 1, 'b': 20, 'c': 3}
```

9.3. Перебор элементов словаря

Перебрать все элементы словаря можно с помощью цикла перебора последовательности (хотя словари и не являются последовательностями). Для примера выведем элементы словаря двумя способами. Первый способ использует метод `keys()`, возвращающий объект с ключами словаря. Во втором случае мы просто указываем словарь в инструкции цикла. На каждой итерации цикла будет возвращаться ключ, с помощью которого внутри цикла можно получить значение, соответствующее этому ключу (листинг 9.1).

Листинг 9.1. Перебор элементов словаря

```
d = {"z": 3, "y": 2, "x": 1 }
for key in d.keys():          # Использование метода keys()
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (z => 3) (y => 2) (x => 1)
print()                       # Вставляем символ перевода строки
for key in d:                 # Словари также поддерживают итерации
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (z => 3) (y => 2) (x => 1)
```

Начиная с Python 3.7, элементы словаря выводятся в порядке их создания (в предыдущих версиях языка они могли выводиться в другом порядке).

Чтобы вывести элементы словаря с сортировкой по ключам, следует получить список ключей и отсортировать его методом `sort()` (листинг 9.2).

Листинг 9.2. Упорядоченный вывод элементов словаря с помощью метода `sort()`

```
d = {"z": 3, "y": 2, "x": 1 }
k = list(d.keys())            # Получаем список ключей
k.sort()                      # Сортируем список ключей
for key in k:
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

Для сортировки ключей также можно воспользоваться функцией `sorted()` (листинг 9.3).

Листинг 9.3. Упорядоченный вывод элементов словаря с помощью функции `sorted()`

```
d = {"z": 3, "y": 2, "x": 1 }
for key in sorted(d.keys()):
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

Так как на каждой итерации возвращается ключ словаря, функции `sorted()` можно передать сам словарь, а не результат выполнения метода `keys()` (листинг 9.4).

Листинг 9.4. Упорядоченный вывод элементов словаря с помощью функции `sorted()`

```
d = {"z": 3, "y": 2, "x": 1 }
for key in sorted(d):
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

9.4. Методы и функции для работы со словарями

Объект словаря поддерживает следующие методы:

- `keys()` — возвращает объект `dict_keys`, содержащий все ключи текущего словаря. Этот объект поддерживает итерации, а также операции над множествами. Примеры:

```
>>> d1, d2 = { "a": 1, "b": 2 }, { "a": 3, "c": 4, "d": 5 }
>>> d1.keys(), d2.keys()              # Получаем объект dict_keys
    (dict_keys(['a', 'b']), dict_keys(['a', 'c', 'd']))
>>> list(d1.keys()), list(d2.keys()) # Получаем список ключей
    (['a', 'b'], ['a', 'c', 'd'])
>>> for k in d1.keys(): print(k, end=" ")
...
    a b
>>> d1.keys() | d2.keys()    # Объединение
    {'a', 'c', 'b', 'd'}
>>> d1.keys() - d2.keys()    # Разница
    {'b'}
>>> d2.keys() - d1.keys()    # Разница
    {'c', 'd'}
>>> d1.keys() & d2.keys()    # Одинаковые ключи
    {'a'}
>>> d1.keys() ^ d2.keys()    # Уникальные ключи
    {'c', 'b', 'd'}
```

- `values()` — возвращает объект `dict_values`, содержащий все значения текущего словаря. Этот объект поддерживает итерации. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> d.values()                  # Получаем объект dict_values
    dict_values([1, 2])
>>> list(d.values())            # Получаем список значений
    [1, 2]
>>> [ v for v in d.values() ]
    [1, 2]
```

- `items()` — возвращает объект `dict_items`, содержащий кортежи с двумя элементами: ключом и значением соответствующего элемента текущего словаря. Этот объект поддерживает итерации. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> d.items()                   # Получаем объект dict_items
    dict_items([('a', 1), ('b', 2)])
```

```
>>> list(d.items())              # Получаем список кортежей
    [('a', 1), ('b', 2)]
```

- get(<Ключ>[, <Значение по умолчанию>]) — если заданный ключ присутствует в текущем словаре, метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, возвращается None или значение из второго параметра. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.get("a"), d.get("c"), d.get("c", 800)
    (1, None, 800)
```

- setdefault(<Ключ>[, <Значение по умолчанию>]) — если заданный ключ присутствует в текущем словаре, метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, в словаре создается новый элемент со значением из второго параметра. Если второй параметр не указан, значением нового элемента будет None. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> d.setdefault("a"),d.setdefault("c"),d.setdefault("d", 0)
    (1, None, 0)
>>> d
    {'a': 1, 'c': None, 'b': 2, 'd': 0}
```

- pop(<Ключ>[, <Значение по умолчанию>]) — удаляет из текущего словаря элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, возвращается значение из второго параметра. Если ключ отсутствует, и второй параметр не указан, возбуждается исключение KeyError. Примеры:

```
>>> d = { "a": 1, "b": 2, "c": 3 }
>>> d.pop("a"), d.pop("n", 0)
    (1, 0)
>>> d.pop("n") # Ключ отсутствует, и нет второго параметра
    Traceback (most recent call last):
      File "<pyshell#40>", line 1, in <module>
        d.pop("n") # Ключ отсутствует, и нет второго параметра
    KeyError: 'n'
>>> d
    {'c': 3, 'b': 2}
```

- popitem() — удаляет из текущего словаря последний элемент и возвращает кортеж из ключа и значения удаленного элемента. Если словарь пустой, возбуждается исключение KeyError. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> d.popitem()
    ('b', 2)
>>> d.popitem()
    ('a', 1)
>>> d.popitem() # Словарь пустой. Возбуждается исключение
    Traceback (most recent call last):
      File "<pyshell#45>", line 1, in <module>
        d.popitem() # Словарь пустой. Возбуждается исключение
    KeyError: 'popitem(): dictionary is empty'
```

В версиях Python, предшествующих 3.7, метод `popitem()` удалял не последний, а произвольный элемент словаря (поскольку порядок хранения элементов в словаре мог меняться в процессе работы программы);

- `clear()` — удаляет все элементы текущего словаря и ничего не возвращает:

```
>>> d = { "a": 1, "b": 2 }
>>> d.clear()               # Удаляем все элементы
>>> d                       # Словарь теперь пустой
    {}
```

- `update()` — добавляет в текущий словарь заданные элементы. Результата не возвращает. Форматы метода:

```
update(<Ключ 1>=<Значение 1>[, . . ., <Ключ N>=<Значение N>])
update(<Словарь>)
update(<Список кортежей с двумя элементами: ключом и значением>)
update(<Список списков с двумя элементами: ключом и значением>)
```

Если элемент с указанным ключом уже присутствует в текущем словаре, то его значение будет перезаписано. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> d.update(c=3, d=4)
>>> d
    {'a': 1, 'c': 3, 'b': 2, 'd': 4}
>>> d.update({"c": 10, "d": 20})          # Словарь
>>> d                       # Значения элементов перезаписаны
    {'a': 1, 'c': 10, 'b': 2, 'd': 20}
>>> d.update([("d", 80), ("e", 6)])       # Список кортежей
>>> d
    {'a': 1, 'c': 10, 'b': 2, 'e': 6, 'd': 80}
>>> d.update([["a", "str"], ["i", "t"]])  # Список списков
>>> d
    {'a': 'str', 'c': 10, 'b': 2, 'e': 6, 'd': 80, 'i': 't'}
```

- `copy()` — создает поверхностную копию текущего словаря:

```
>>> d1 = { "a": 1, "b": [10, 20] }
>>> d2 = d1.copy() # Создаем поверхностную копию
>>> d1 is d2       # Это разные объекты
    False
>>> d2["a"] = 800  # Изменяем значение
>>> d1, d2         # Изменилось значение только в d2
    ({'a': 1, 'b': [10, 20]}, {'a': 800, 'b': [10, 20]})
>>> d2["b"][0] = "new" # Изменяем значение вложенного списка
>>> d1, d2         # Изменились значения и в d1, и в d2!!!
    ({'a': 1, 'b': ['new', 20]}, {'a': 800, 'b': ['new', 20]})
```

Чтобы создать полную копию словаря, следует воспользоваться функцией `deepcopy()` из модуля `copy`.

Функция `reversed(<Словарь>)` (поддерживается, начиная с Python 3.8) возвращает итератор, который выдает ключи заданного словаря, приведенные в обратном порядке:

```
>>> for k in reversed(d): print(k + " => " + str(d[k]), end=" ")
...
    c => 3 b => 2 a => 1
```

9.5. Генераторы словарей

Помимо генераторов списков и множеств, Python поддерживает *генераторы словарей*. Формат записи генератора словаря:

```
{ <Выражение ключа>: <Выражение значения> for
  (<Переменная ключа>, <Переменная значения>) in <Исходное отображение> }
```

В заданную переменную ключа на каждой итерации помещается ключ очередного элемента заданного исходного отображения, а в переменную значения — значение очередного элемента. Заданное выражение ключа вычисляет ключ, а заданное выражение значения — значение очередного элемента создаваемого словаря. Примеры:

```
>>> keys = ["a", "b"]          # Список с ключами
>>> values = [1, 2]            # Список со значениями
>>> {k: v for (k, v) in zip(keys, values)}
    {'a': 1, 'b': 2}
>>> {k: 0 for k in keys}
    {'a': 0, 'b': 0}
```

Пример более сложного генератора словаря, создающего из исходного словаря новый, содержащий только элементы с четными значениями:

```
>>> d = { "a": 1, "b": 2, "c": 3, "d": 4 }
>>> {k: v for (k, v) in d.items() if v % 2 == 0}
    {'b': 2, 'd': 4}
```

ГЛАВА 10

Работа с датой и временем

Python предоставляет средства для работы со значениями даты и времени, а также комбинациями даты и времени (*временны́ми отметками*). Кроме того, он содержит инструменты для вывода календаря в виде текста или в формате HTML и измерения времени выполнения определенных фрагментов программы для целей отладки.

10.1. Получение текущих даты и времени

Получить текущие дату и время позволяют следующие функции из модуля `time`:

- `time()` — возвращает количество секунд, прошедшее с начала эпохи (обычно с 1 января 1970 г.), в виде вещественного числа:

```
>>> import time              # Подключаем модуль time
>>> time.time()              # Получаем количество секунд
    1642072670.3688238
```

- `gmtime([<Количество секунд>])` — возвращает универсальное время (UTC) в виде объекта `struct_time`. Если параметр не указан, возвращается текущее время. Если параметр указан, время будет соответствующим заданному количеству секунд, прошедших с начала эпохи. Примеры:

```
>>> time.gmtime(0)                  # Начало эпохи
    time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0,
    tm_min=0, tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
>>> time.gmtime()                   # Текущая дата и время
    time.struct_time(tm_year=2022, tm_mon=1, tm_mday=13, tm_hour=11,
    tm_min=19, tm_sec=55, tm_wday=3, tm_yday=13, tm_isdst=0)
>>> time.gmtime(1642072670.0)       # Дата 13.01.2022
    time.struct_time(tm_year=2022, tm_mon=1, tm_mday=13, tm_hour=11,
    tm_min=17, tm_sec=50, tm_wday=3, tm_yday=13, tm_isdst=0)
```

Получить значение конкретного атрибута можно, указав его имя или индекс внутри объекта:

```
>>> d = time.gmtime()
>>> d.tm_year, d[0]
    (2022, 2022)
```

```
>>> tuple(d)              # Преобразование в кортеж
    (2022, 1, 13, 11, 21, 6, 3, 13, 0)
```

- `localtime([<Количество секунд>])` — возвращает местное время в виде объекта `struct_time`. Если параметр не указан, возвращается текущее время. Если параметр указан, время будет соответствующим заданному количеству секунд, прошедших с начала эпохи. Примеры:

```
>>> time.localtime()              # Текущая дата и время
    time.struct_time(tm_year=2022, tm_mon=1, tm_mday=13, tm_hour=14,
    tm_min=22, tm_sec=44, tm_wday=3, tm_yday=13, tm_isdst=0)
>>> time.localtime(1642072670.0)  # Дата 13.01.2022
    time.struct_time(tm_year=2022, tm_mon=1, tm_mday=13, tm_hour=14,
    tm_min=17, tm_sec=50, tm_wday=3, tm_yday=13, tm_isdst=0)
```

- `mktime(<Объект struct_time>)` — возвращает количество секунд, прошедших с начала эпохи, в виде вещественного числа. В качестве параметра указывается объект `struct_time` или кортеж из девяти элементов. Если указанная дата некорректна, возбуждается исключение `OverflowError`. Примеры:

```
>>> d = time.localtime(1642072670.0)
>>> time.mktime(d)
    1642072670.0
>>> tuple(time.localtime(1642072670.0))
    (2022, 1, 13, 14, 17, 50, 3, 13, 0)
>>> time.mktime((2022, 1, 13, 14, 17, 50, 3, 13, 0))
    1642072670.0
>>> time.mktime((1940, 0, 31, 5, 23, 43, 5, 31, 0))
    ... Фрагмент опущен ...
    OverflowError: mktime argument out of range
```

Объект `struct_time`, возвращаемый функциями `gmtime()` и `localtime()`, содержит следующие атрибуты (указаны тройки вида «имя атрибута — индекс — описание»):

- `tm_year` — 0 — год;
- `tm_mon` — 1 — месяц (число от 1 до 12);
- `tm_mday` — 2 — день месяца (число от 1 до 31);
- `tm_hour` — 3 — час (число от 0 до 23);
- `tm_min` — 4 — минуты (число от 0 до 59);
- `tm_sec` — 5 — секунды (число от 0 до 59, изредка до 61);
- `tm_wday` — 6 — день недели (число от 0 для понедельника до 6 для воскресенья);
- `tm_yday` — 7 — количество дней, прошедшее с начала года (число от 1 до 366);
- `tm_isdst` — 8 — флаг коррекции летнего времени (значения 0, 1 или `-1`).

Выведем текущие дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 10.1).

Листинг 10.1. Вывод текущих даты и времени

```
import time
d = [ "понедельник", "вторник", "среда", "четверг",
      "пятница", "суббота", "воскресенье" ]
```

```
m = [ "", "января", "февраля", "марта", "апреля", "мая",
      "июня", "июля", "августа", "сентября", "октября",
      "ноября", "декабря" ]
t = time.localtime() # Получаем текущее время
print( "Сегодня:\n%s %s %s %s %02d:%02d:%02d\n%02d.%02d.%02d" %
     ( d[t[6]], t[2], m[t[1]], t[0], t[3], t[4], t[5],
       t[2], t[1], t[0] ) )
input()
```

Примерный результат выполнения:

```
Сегодня:
четверг 13 января 2022 14:29:55
13.01.2022
```

10.2. Форматирование даты и времени

Форматирование даты и времени выполняют следующие функции из модуля `time`:

- `strftime(<Строка формата>[, <Объект struct_time>])` — возвращает строковое представление даты и времени в соответствии с заданной строкой формата. Если второй параметр не указан, будут выведены текущие дата и время. Если во втором параметре указан объект `struct_time` или кортеж из девяти элементов, дата будет соответствовать указанному значению. Функция зависит от настройки локали. Примеры:

```
>>> import time
>>> time.strftime("%d.%m.%Y") # Форматирование даты
    '13.01.2022'
>>> time.strftime("%H:%M:%S") # Форматирование времени
    '14:31:29'
>>> time.strftime("%d.%m.%Y", time.localtime(1639725843.0))
    '17.12.2021'
```

- `strptime(<Строка с датой и временем>[, <Строка формата>])` — разбирает указанную строку с датой и временем в соответствии с заданной строкой формата. Возвращает объект `struct_time`. Если строка не соответствует формату, возбуждается исключение `ValueError`. Если строка формата не указана, используется строка `"%a %b %d %H:%M:%S %Y"`. Функция учитывает текущую локаль. Примеры:

```
>>> time.strptime("Fri Dec 17 10:24:03 2021")
    time.struct_time(tm_year=2021, tm_mon=12, tm_mday=17, tm_hour=10,
    tm_min=24, tm_sec=3, tm_wday=4, tm_yday=351, tm_isdst=-1)
>>> time.strptime("17.12.2021", "%d.%m.%Y")
    time.struct_time(tm_year=2021, tm_mon=12, tm_mday=17, tm_hour=0,
    tm_min=0, tm_sec=0, tm_wday=4, tm_yday=351, tm_isdst=-1)
>>> time.strptime("17-12-2021", "%d.%m.%Y")
    ... Фрагмент опущен ...
    ValueError: time data '17-12-2021' does not match format '%d.%m.%Y'
```

- `asctime([<Объект struct_time>])` — возвращает строку формата `"%a %b %d %H:%M:%S %Y"`. Если параметр не указан, будут выведены текущие дата и время. Если в параметре указан объект `struct_time` или кортеж из девяти элементов, то дата и время будут соответствовать указанному значению. Примеры:

```
>>> time.asctime()                                # Текущая дата
    'Thu Jan 13 17:32:37 2022'
>>> time.asctime(time.localtime(1639725843.0))  # Дата в прошлом
    'Fri Dec 17 10:24:03 2021'
```

- `ctime([<Количество секунд>])` — аналогична `asctime()`, но в качестве параметра принимает количество секунд, прошедших с начала эпохи:

```
>>> time.ctime()                    # Текущая дата
    'Thu Jan 13 17:34:04 2022'
>>> time.ctime(1639725843.0)        # Дата в прошлом
    'Fri Dec 17 10:24:03 2021'
```

В параметре `<Строка формата>` в функциях `strftime()` и `strptime()` могут быть использованы следующие комбинации специальных символов:

- `%y` — год из двух цифр (от `"00"` до `"99"`);
- `%Y` — год из четырех цифр (например, `"2011"`);
- `%m` — номер месяца с предваряющим нулем (от `"01"` до `"12"`);
- `%b` — аббревиатура месяца в зависимости от настроек локали (например, `"янв"` для января);
- `%B` — название месяца в зависимости от настроек локали (например, `"Январь"`);
- `%d` — номер дня в месяце с предваряющим нулем (от `"01"` до `"31"`);
- `%j` — день с начала года (от `"001"` до `"366"`);
- `%U` — номер недели в году (от `"00"` до `"53"`). Неделя начинается с воскресенья. Все дни с начала года до первого воскресенья относятся к неделе с номером 0;
- `%W` — номер недели в году (от `"00"` до `"53"`). Неделя начинается с понедельника. Все дни с начала года до первого понедельника относятся к неделе с номером 0;
- `%w` — номер дня недели (`"0"` — для воскресенья, `"6"` — для субботы);
- `%a` — аббревиатура дня недели в зависимости от настроек локали (например, `"Пн"` для понедельника);
- `%A` — название дня недели в зависимости от настроек локали (например, `"понедельник"`);
- `%H` — часы в 24-часовом формате (от `"00"` до `"23"`);
- `%I` — часы в 12-часовом формате (от `"01"` до `"12"`);
- `%M` — минуты (от `"00"` до `"59"`);
- `%S` — секунды (от `"00"` до `"59"`, изредка до `"61"`);
- `%p` — эквивалент значений `AM` и `PM` в текущей локали;
- `%c` — представление даты и времени в текущей локали;
- `%x` — представление даты в текущей локали;
- `%X` — представление времени в текущей локали:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
    'Russian_Russia.1251'
>>> print(time.strftime("%x"))   # Представление даты
    13.01.2022
```

```
>>> print(time.strftime("%X"))    # Представление времени
    17:36:50
>>> print(time.strftime("%c"))    # Представление даты и времени
    13.01.2022 17:37:17
```

- `%Z` — название часового пояса или пустая строка (например, `"Московское время"`, `"UTC"`);
- `%%` — символ `"%"`.

В качестве примера выведем текущие дату и время с помощью функции `strftime()` (листинг 10.2).

Листинг 10.2. Форматирование даты и времени

```
import time
import locale
locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
s = "Сегодня:\n%A %d %b %Y %H:%M:%S\n%d.%m.%Y"
print(time.strftime(s))
input()
```

Примерный результат выполнения:

```
Сегодня:
четверг 13 янв 2022 17:37:59
13.01.2022
```

10.3. Приостановка выполнения программы

Функция `sleep(<Время в секундах>)` из модуля `time` приостанавливает выполнение программы на указанное время. В качестве параметра можно указать целое или вещественное число. Пример:

```
>>> import time
>>> time.sleep(5)              # "Засыпаем" на 5 секунд
```

10.4. Значения даты и времени

Модуль `datetime` предоставляет инструменты для работы со значениями даты и времени: выполнения арифметических операций, сравнения, вывода в различных форматах и др. Предварительно необходимо подключить модуль с помощью инструкции:

```
import datetime
```

10.4.1. Временны́е промежутки

Временно́й промежуто́к — это величина разницы между какими-либо временны́ми отметками (т. е. значениями времени и даты).

Класс `timedelta` из модуля `datetime` позволяет выполнять операции над временны́ми промежутками: складывать, вычитать, сравнивать и др. Конструктор класса имеет следующий формат:

```
timedelta([days][, seconds][, microseconds][, milliseconds][, minutes][,
                                                hours][, weeks])
```

Все параметры не являются обязательными и по умолчанию имеют значение 0. Их можно записывать как позиционные или как именованные. Первые три параметра считаются основными:

- `days` — дни (диапазон `-999999999 <= days <= 999999999`);
- `seconds` — секунды (диапазон `0 <= seconds < 3600*24`);
- `microseconds` — микросекунды (диапазон `0 <= microseconds < 1000000`).

Примеры:

```
>>> datetime.timedelta(3)
    datetime.timedelta(days=3)
>>> datetime.timedelta(0, 2, 10000)
    datetime.timedelta(seconds=2, microseconds=10000)
>>> datetime.timedelta(days=1, seconds=7200)
    datetime.timedelta(days=1, seconds=7200)
```

Все остальные параметры автоматически преобразуются в следующие значения:

- `milliseconds` — миллисекунды (одна миллисекунда преобразуется в 1000 микросекунд):

```
>>> import datetime
>>> datetime.timedelta(milliseconds=1)
    datetime.timedelta(microseconds=1000)
```

- `minutes` — минуты (одна минута преобразуется в 60 секунд):

```
>>> datetime.timedelta(minutes=1)
    datetime.timedelta(seconds=60)
```

- `hours` — часы (один час преобразуется в 3600 секунд):

```
>>> datetime.timedelta(hours=1)
    datetime.timedelta(seconds=3600)
```

- `weeks` — недели (одна неделя преобразуется в 7 дней):

```
>>> datetime.timedelta(weeks=1)
    datetime.timedelta(days=7)
```

Получить результат можно с помощью следующих атрибутов:

- `days` — дни;
- `seconds` — секунды;
- `microseconds` — микросекунды.

Пример:

```
>>> d = datetime.timedelta(hours=1, days=2, milliseconds=1)
>>> d
    datetime.timedelta(days=2, seconds=3600, microseconds=1000)
>>> d.days, d.seconds, d.microseconds
    (2, 3600, 1000)
```

Получить результат в секундах позволяет метод `total_seconds()`:

```
>>> d = datetime.timedelta(minutes=1)
>>> d.total_seconds()
    60.0
```

Над временны́ми промежутками можно производить арифметические операции `+`, `-`, `/`, `//`, `%` и `*`, использовать унарные операторы `+` и `-`, получать абсолютное значение с помощью функции `abs()`, вычислять частное и остаток от деления нацело функцией `divmod()`:

```
>>> d1 = datetime.timedelta(days=2)
>>> d2 = datetime.timedelta(days=7)
>>> d1 + d2, d2 - d1                              # Сложение и вычитание
    (datetime.timedelta(days=9), datetime.timedelta(days=5))
>>> d2 / d1                                       # Деление
    3.5
>>> d1 / 2, d2 / 2.5                              # Деление
    (datetime.timedelta(days=1), datetime.timedelta(days=2, seconds=69120))
>>> d2 // d1                                      # Деление
    3
>>> d1 // 2, d2 // 2                              # Деление
    (datetime.timedelta(days=1), datetime.timedelta(days=3, seconds=43200))
>>> d2 % d1                                       # Остаток
    datetime.timedelta(days=1)
>>> d1 * 2, d2 * 2                                # Умножение
    (datetime.timedelta(days=4), datetime.timedelta(days=14))
>>> 2 * d1, 2 * d2                                # Умножение
    (datetime.timedelta(days=4), datetime.timedelta(days=14))
>>> d3 = -d1
>>> d3, abs(d3)
    (datetime.timedelta(days=-2), datetime.timedelta(days=2))
>>> divmod(d2, d1)
    (3, datetime.timedelta(days=1))
```

Кроме того, можно использовать операторы сравнения `==`, `!=`, `<`, `<=`, `>` и `>=`:

```
>>> d1 = datetime.timedelta(days=2)
>>> d2 = datetime.timedelta(days=7)
>>> d3 = datetime.timedelta(weeks=1)
>>> d1 == d2, d2 == d3                # Проверка на равенство
    (False, True)
>>> d1 != d2, d2 != d3                # Проверка на неравенство
    (True, False)
>>> d1 < d2, d2 <= d3                 # Меньше, меньше или равно
    (True, True)
>>> d1 > d2, d2 >= d3                 # Больше, больше или равно
    (False, True)
```

Также можно получить строковое представление временно́го промежутка с помощью функций `str()` и `repr()`:

```
>>> d = datetime.timedelta(hours=25, minutes=5, seconds=27)
>>> str(d), repr(d)
    ('1 day, 1:05:27', 'datetime.timedelta(days=1, seconds=3927)')
```

Класс timedelta поддерживают следующие атрибуты:

- min — минимальное значение, которое может иметь временной промежуток;
- max — максимальное значение, которое может иметь временной промежуток;
- resolution — минимальное возможное различие между временными промежутками.

Выведем значения этих атрибутов:

```
>>> datetime.timedelta.min
    datetime.timedelta(days=-999999999)
>>> datetime.timedelta.max
    datetime.timedelta(days=999999999, seconds=86399, microseconds=999999)
>>> datetime.timedelta.resolution
    datetime.timedelta(microseconds=1)
```

10.4.2. Значения даты

Класс date из модуля datetime позволяет выполнять операции над датами. Конструктор класса имеет следующий формат:

```
date(<Год>, <Месяц>, <День>)
```

В параметрах указываются числа:

- <Год> — в диапазоне между значениями констант MINYEAR и MAXYEAR класса datetime (о нем речь пойдет позже). Выведем значения этих констант:

```
>>> import datetime
>>> datetime.MINYEAR, datetime.MAXYEAR
    (1, 9999)
```

- <Месяц> — от 1 до 12 включительно;
- <День> — от 1 до количества дней в месяце.

Если значения выходят за диапазон, возбуждается исключение ValueError. Примеры:

```
>>> datetime.date(2022, 1, 13)
    datetime.date(2022, 1, 13)
>>> datetime.date(2022, 13, 3)   # Неправильное значение месяца
    ... Фрагмент опущен ...
    ValueError: month must be in 1..12
```

Для создания даты также можно воспользоваться следующими методами класса date:

- today() — возвращает текущую дату:

```
>>> datetime.date.today()
    datetime.date(2022, 1, 13)
```

- fromtimestamp(<Количество секунд>) — возвращает дату, которая соответствует заданному количеству секунд, прошедших с начала эпохи:

```
>>> import datetime, time
>>> datetime.date.fromtimestamp(time.time())  # Текущая дата
    datetime.date(2022, 1, 13)
>>> datetime.date.fromtimestamp(1639725843.0) # Дата 17.12.2021
    datetime.date(2021, 12, 17)
```

- `fromordinal(<Количество дней с 1-го года>)` — возвращает дату, которая соответствует заданному количеству дней, прошедших с первого года. В качестве параметра указывается число от 1 до `datetime.date.max.toordinal()`:

```
>>> datetime.date.max.toordinal()
    3652059
>>> datetime.date.fromordinal(3652059)
    datetime.date(9999, 12, 31)
>>> datetime.date.fromordinal(1)
    datetime.date(1, 1, 1)
```

- `fromisoformat(<Строка с датой>)` (начиная с Python 3.7) — возвращает дату, созданную в результате преобразования заданной строки, которая указывается в формате `<ГГГГ>-<ММ>-<ДД>`:

```
>>> datetime.date.fromisoformat("2021-12-17")
    datetime.date(2021, 12, 17)
```

- `fromisocalendar()` (начиная с Python 3.8) — возвращает дату, созданную на основе заданных параметров. Формат метода:

```
fromisocalendar(<Год>, <Порядковый номер недели в году>,
                       <Порядковый номер дня в неделе>)
```

Пример:

```
>>> datetime.date.fromisocalendar(2022, 2, 4)
    datetime.date(2022, 1, 13)
```

Получить результат можно с помощью следующих атрибутов:

- `year` — год (число в диапазоне от `MINYEAR` до `MAXYEAR`);
- `month` — месяц (число от 1 до 12);
- `day` — день (число от 1 до количества дней в месяце).

Пример:

```
>>> d = datetime.date.today()
>>> d.year, d.month, d.day
    (2022, 1, 13)
```

Над экземплярами класса `date` можно производить следующие операции:

- `date2 = date1 + timedelta` — прибавляет к дате `date1` временной промежуток `timedelta`. Значения атрибутов `seconds` и `microseconds` временно́го промежутка игнорируются;
- `date2 = date1 - timedelta` — вычитает из даты `date1` временной промежуток `timedelta`. Значения атрибутов `seconds` и `microseconds` временно́го промежутка игнорируются;
- `timedelta = date1 - date2` — возвращает разницу между датами `date1` и `date2` в виде временно́го промежутка. Атрибуты `seconds` и `microseconds` последнего будут иметь значения 0.

Также можно сравнивать две даты с помощью операторов сравнения:

```
>>> d1 = datetime.date(2022, 1, 13)
>>> d2 = datetime.date(2021, 12, 31)
```

```
>>> t = datetime.timedelta(days=10)
>>> d1 + t, d1 - t              # Прибавляем и вычитаем 10 дней
    (datetime.date(2022, 1, 23), datetime.date(2022, 1, 3))
>>> d1 - d2                     # Разница между датами
    datetime.timedelta(days=13)
>>> d1 < d2, d1 > d2, d1 <= d2, d1 >= d2
    (False, True, False, True)
>>> d1 == d2, d1 != d2
    (False, True)
```

Можно получить строковое представление даты с помощью функций `str()` и `repr()`:

```
>>> d = datetime.date(2022, 1, 13)
>>> str(d), repr(d)
    ('2022-01-13', 'datetime.date(2022, 1, 13)')
```

Класс `date` поддерживает следующие методы:

- `replace([year][, month][, day])` — возвращает текущую дату с обновленными значениями года (параметр `year`), месяца (`month`) и (или) числа (`day`). Параметры можно указывать как позиционные или как именованные. Примеры:

```
>>> d = datetime.date(2021, 12, 17)
>>> d.replace(2022, 7)  # Заменяем год и месяц
    datetime.date(2022, 7, 17)
>>> d.replace(year=2021, month=2, day=28)
    datetime.date(2021, 2, 28)
>>> d.replace(day=2)    # Заменяем только день
    datetime.date(2021, 12, 2)
```

- `strftime(<Строка формата>)` — возвращает текущую дату в виде строки, отформатированной согласно заданной строке формата. В строке формата можно задавать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`. Пример:

```
>>> d = datetime.date(2021, 12, 17)
>>> d.strftime("%d.%m.%Y")
    '17.12.2021'
```

- `isoformat()` — возвращает текущую дату в виде строки формата <ГГГГ>-<ММ>-<ДД>:

```
>>> d = datetime.date(2021, 12, 17)
>>> d.isoformat()
    '2021-12-17'
```

- `ctime()` — возвращает текущую дату в виде строки формата `"%a %b %d %H:%M:%S %Y"`:

```
>>> d = datetime.date(2021, 12, 17)
>>> d.ctime()
    'Fri Dec 17 00:00:00 2021'
```

- `timetuple()` — возвращает текущую дату в виде объекта `struct_time`:

```
>>> d = datetime.date(2021, 12, 17)
>>> d.timetuple()
    time.struct_time(tm_year=2021, tm_mon=12, tm_mday=17, tm_hour=0,
    tm_min=0, tm_sec=0, tm_wday=4, tm_yday=351, tm_isdst=-1)
```

- `toordinal()` — возвращает текущую дату в виде количества дней, прошедших с 1-го года:

```
>>> d = datetime.date(2021, 12, 17)
>>> d.toordinal()
    738141
```

- `weekday()` — возвращает порядковый номер дня в неделе (0 — для понедельника, 6 — для воскресенья):

```
>>> d = datetime.date(2021, 12, 17)
>>> d.weekday() # 4 - это пятница
    4
```

- `isoweekday()` — возвращает порядковый номер дня в неделе (1 — для понедельника, 7 — для воскресенья):

```
>>> d = datetime.date(2021, 12, 17)
>>> d.isoweekday() # 5 - это пятница
    5
```

- `isocalendar()` — возвращает последовательность из трех элементов: года, номера недели в году и порядкового номера дня в неделе:

```
>>> d = datetime.date(2021, 12, 17)
>>> isoc = d.isocalendar()
>>> isoc[0], isoc[1], isoc[2]
    (2021, 50, 5)
```

Наконец, имеется поддержка следующих атрибутов:

- `min` — минимально возможное значение даты;
- `max` — максимально возможное значение даты;
- `resolution` — минимальное возможное различие между значениями даты.

Выведем значения этих атрибутов:

```
>>> datetime.date.min
    datetime.date(1, 1, 1)
>>> datetime.date.max
    datetime.date(9999, 12, 31)
>>> datetime.date.resolution
    datetime.timedelta(days=1)
```

10.4.3. Значения времени

Класс `time` из модуля `datetime` позволяет выполнять операции над значениями времени. Конструктор класса имеет следующий формат:

```
time([hour][, minute][, second][, microsecond][, tzinfo], [fold])
```

Все параметры не являются обязательными. Их можно указывать как позиционные или как именованные, за исключением параметра `fold`, который можно указать только как именованный. В параметрах можно задать следующий диапазон значений:

- `hour` — часы (число от 0 до 23);
- `minute` — минуты (число от 0 до 59);

- `second` — секунды (число от 0 до 59);
- `microsecond` — микросекунды (число от 0 до 999999);
- `tzinfo` — временна́я зона (объект класса `tzinfo` или значение `None`);
- `fold` — порядковый номер отметки времени. Значение 0 (используется по умолчанию) обозначает первую отметку, значение 1 — вторую. Предусмотрен для тех случаев, когда в рассматриваемой временно́й зоне практикуется перевод часов с зимнего на летнее время и обратно, в результате чего часы могут дважды в сутки показывать одинаковое время.

Если значения выходят за диапазон, возбуждается исключение `ValueError`. Примеры:

```
>>> import datetime
>>> datetime.time(13, 49, 24, 186500)
    datetime.time(13, 49, 24, 186500)
>>> datetime.time(hour=13, second=24, minute=49)
    datetime.time(13, 49, 24)
>>> datetime.time(25, 12, 38, 375000)
    ... Фрагмент опущен ...
    ValueError: hour must be in 0..23
```

Создать значение времени также можно с помощью метода `fromisoformat(<Строка с временем>)` класса `time` (этот метод поддерживается, начиная с Python 3.7). Строка с временем должна указываться в формате ISO 8601. Пример:

```
>>> datetime.time.fromisoformat("13:49:24.186500")
    datetime.time(13, 49, 24, 186500)
```

Получить результат можно с помощью следующих атрибутов:

- `hour` — часы (число от 0 до 23);
- `minute` — минуты (число от 0 до 59);
- `second` — секунды (число от 0 до 59);
- `microsecond` — микросекунды (число от 0 до 999999);
- `tzinfo` — временна́я зона (объект класса `tzinfo` или значение `None`);
- `fold` — порядковый номер отметки времени (число 0 или 1).

Пример:

```
>>> t = datetime.time(13, 49, 24, 186500)
>>> t.hour, t.minute, t.second, t.microsecond
    (13, 49, 24, 186500)
```

Над значениями времени нельзя выполнять арифметические операции. Можно только производить сравнения, например:

```
>>> t1 = datetime.time(13, 49, 24, 186500)
>>> t2 = datetime.time(8, 54, 35)
>>> t1 < t2, t1 > t2, t1 <= t2, t1 >= t2
    (False, True, False, True)
>>> t1 == t2, t1 != t2
    (False, True)
```

Можно получить строковое представление времени с помощью функций `str()` и `repr()`:

```
>>> t = datetime.time(13, 49, 24, 186500)
>>> str(t), repr(t)
    ('13:49:24.186500', 'datetime.time(13, 49, 24, 186500)')
```

Класс `time` поддерживает следующие методы:

- `replace()` — возвращает текущее время с обновленными значениями, указанными в параметрах. Формат метода:

  ```
  replace([hour][, minute][, second][, microsecond][, tzinfo] [, fold])
  ```

  Параметры можно указывать как позиционные или как именованные. Примеры:

  ```
  >>> t = datetime.time(13, 49, 24, 186500)
  >>> t.replace(10, 12)       # Заменяем часы и минуты
      datetime.time(10, 12, 24, 186500)
  >>> t.replace(second=43)    # Заменяем только секунды
      datetime.time(13, 49, 43, 186500)
  ```

- `isoformat([timespec="auto"])` — возвращает строку с текущим временем в формате ISO 8601. В параметре `timespec` можно указать состав выдаваемого времени, задав одно из следующих значений:
  - `"auto"` — часы, минуты, секунды и микросекунды, если количество последних не равно 0 — в противном случае микросекунды не включаются в состав времени;
  - `"hours"` — часы;
  - `"minutes"` — часы и минуты;
  - `"seconds"` — часы, минуты и секунды;
  - `"milliseconds"` — часы, минуты, секунды и миллисекунды (вычисляются путем округления микросекунд до тысячных);
  - `"microseconds"` — часы, минуты, секунды и микросекунды (выводятся всегда, даже если их количество равно 0).

  Примеры:

  ```
  >>> t = datetime.time(13, 49, 24, 186500)
  >>> t.isoformat()
      '13:49:24.186500'
  >>> t.isoformat("minutes")
      '13:49'
  ```

- `strftime(<Строка формата>)` — возвращает текущее время, отформатированное согласно заданной строке формата. В строке формата можно указывать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`. Пример:

  ```
  >>> t = datetime.time(13, 49, 24, 186500)
  >>> t.strftime("%H:%M:%S")
      '13:49:24'
  ```

Дополнительно класс `time` поддерживает такие атрибуты:

- `min` — минимально возможное значение времени;
- `max` — максимально возможное значение времени;
- `resolution` — минимальное возможное различие между значениями времени.

Вот значения этих атрибутов:

```
>>> datetime.time.min
    datetime.time(0, 0)
>>> datetime.time.max
    datetime.time(23, 59, 59, 999999)
>>> datetime.time.resolution
    datetime.timedelta(microseconds=1)
```

> ***Примечание***
>
> Класс `time` поддерживает также методы `dst()`, `utcoffset()` и `tzname()`. За подробной информацией по этим методам, а также по абстрактному классу `tzinfo` обращайтесь к документации по модулю `datetime`.

10.4.4. Временны́е отметки

Класс `datetime` из модуля `datetime` позволяет выполнять операции над временны́ми отметками (комбинациями даты и времени). Конструктор класса имеет следующий формат:

```
datetime(<Год>, <Месяц>, <День>[, hour][, minute][, second][,
                                 microsecond][, tzinfo][, fold])
```

Первые три параметра являются обязательными. Остальные параметры можно указывать как позиционные или как именованные, кроме `fold`, который задается только как именованный. В параметрах можно указать следующие значения:

- `<Год>` — число, расположенное в диапазоне между значениями из констант `MINYEAR` (1) и `MAXYEAR` (9999);
- `<Месяц>` — число от 1 до 12 включительно;
- `<День>` — число от 1 до количества дней в месяце;
- `hour` — часы (число от 0 до 23);
- `minute` — минуты (число от 0 до 59);
- `second` — секунды (число от 0 до 59);
- `microsecond` — микросекунды (число от 0 до 999999);
- `tzinfo` — временна́я зона (объект класса `tzinfo` или значение `None`);
- `fold` — порядковый номер отметки времени. Значение 0 (используется по умолчанию) обозначает первую отметку, значение 1 — вторую. Предусмотрен для тех случаев, когда в рассматриваемой временно́й зоне практикуется перевод часов с зимнего на летнее время и обратно, в результате чего часы могут дважды в сутки показывать одинаковое время.

Если значения выходят за диапазон, возбуждается исключение `ValueError`. Примеры:

```
>>> import datetime
>>> datetime.datetime(2022, 1, 14)
    datetime.datetime(2022, 1, 14, 0, 0)
>>> datetime.datetime(2022, 1, 14, hour=14, minute=22)
    datetime.datetime(2022, 1, 14, 14, 22)
>>> datetime.datetime(2022, 32, 20)
    ... Фрагмент опущен ...
    ValueError: month must be in 1..12
```

Для создания временно́й отметки также можно воспользоваться следующими методами:

- `today()` — возвращает текущие дату и время:

```
>>> datetime.datetime.today()
    datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
```

- `now([<Временна́я зона>])` — возвращает текущие дату и время. Если параметр не задан, то метод аналогичен методу `today()`. Пример:

```
>>> datetime.datetime.now()
    datetime.datetime(2022, 1, 14, 14, 26, 41, 833984)
```

- `utcnow()` — возвращает текущее универсальное время (UTC):

```
>>> datetime.datetime.utcnow()
    datetime.datetime(2022, 1, 14, 11, 27, 28, 713491)
```

- `fromtimestamp(<Количество секунд>[, <Временна́я зона>])` — возвращает временну́ю отметку, которая соответствует заданному количеству секунд, прошедших с начала эпохи:

```
>>> import datetime, time
>>> datetime.datetime.fromtimestamp(time.time())
    datetime.datetime(2022, 1, 14, 14, 28, 47, 984896)
>>> datetime.datetime.fromtimestamp(1639725843.0)
    datetime.datetime(2021, 12, 17, 10, 24, 3)
```

- `utcfromtimestamp(<Количество секунд>)` — возвращает временну́ю отметку, которая соответствует заданному количеству секунд, прошедших с начала эпохи, в универсальном времени (UTC):

```
>>> datetime.datetime.utcfromtimestamp(time.time())
    datetime.datetime(2022, 1, 14, 11, 32, 11, 356915)
>>> datetime.datetime.utcfromtimestamp(1639725843.0)
    datetime.datetime(2021, 12, 17, 7, 24, 3)
```

- `fromordinal(<Количество дней с 1-го года>)` — возвращает дату, которая соответствует заданному количеству дней, прошедших с 1-го года. В качестве параметра указывается число от 1 до `datetime.datetime.max.toordinal()`. Примеры:

```
>>> datetime.datetime.max.toordinal()
    3652059
>>> datetime.datetime.fromordinal(3652059)
    datetime.datetime(9999, 12, 31, 0, 0)
>>> datetime.datetime.fromordinal(1)
    datetime.datetime(1, 1, 1, 0, 0)
```

- `fromisoformat(<Строка с датой и временем>)` (начиная с Python 3.7) — возвращает временну́ю отметку, созданную в результате преобразования заданной строки, которая указывается в формате ISO 8601:

```
>>> datetime.datetime.fromisoformat("2021-12-17T17:24:03.190301")
    datetime.datetime(2021, 12, 17, 17, 24, 3, 190301)
```

- `fromisocalendar()` (начиная с Python 3.8) — возвращает временну́ю отметку, созданную на основе заданных параметров. Формат метода:

```
fromisocalendar(<Год>, <Порядковый номер недели в году>,
                       <Порядковый номер дня в неделе>)
```

Пример:

```
>>> datetime.datetime.fromisocalendar(2022, 2, 4)
    datetime.datetime(2022, 1, 13, 0, 0)
```

- `combine(<Дата>, <Время>[, <Временна́я зона>])` — возвращает временну́ю отметку, созданную на основе переданных ему даты и времени:

```
>>> d = datetime.date(2022, 1, 14)
>>> t = datetime.time(14, 35, 14)
>>> datetime.datetime.combine(d, t)
    datetime.datetime(2022, 1, 14, 14, 35, 14)
```

- `strptime(<Строка с датой>, <Строка формата>)` — возвращает временну́ю отметку, созданную на основе разбора заданной строки с датой в соответствии с указанной строкой формата. Если строка с датой не соответствует формату, возбуждается исключение `ValueError`. Метод учитывает текущую локаль. Примеры:

```
>>> datetime.datetime.strptime("17.12.2021", "%d.%m.%Y")
    datetime.datetime(2021, 12, 17, 0, 0)
>>> datetime.datetime.strptime("17.12.2021", "%d-%m-%Y")
    ... Фрагмент опущен ...
    ValueError: time data '17.12.2021' does not match format '%d-%m-%Y'
```

Получить результат можно с помощью следующих атрибутов:

- `year` — год (число в диапазоне от `MINYEAR` до `MAXYEAR`);
- `month` — месяц (число от 1 до 12);
- `day` — день (число от 1 до количества дней в месяце);
- `hour` — часы (число от 0 до 23);
- `minute` — минуты (число от 0 до 59);
- `second` — секунды (число от 0 до 59);
- `microsecond` — микросекунды (число от 0 до 999999);
- `tzinfo` — временна́я зона (объект класса `tzinfo` или значение `None`).
- `fold` — порядковый номер отметки времени (число 0 или 1).

Примеры:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.year, d.month, d.day
    (2022, 1, 14)
>>> d.hour, d.minute, d.second, d.microsecond
    (14, 25, 21, 998076)
```

Над временны́ми отметками можно производить следующие операции:

- `datetime2 = datetime1 + timedelta` — сложение временно́й отметки `datetime1` и временно́го промежутка `timedelta`;
- `datetime2 = datetime1 - timedelta` — вычитание из временно́й отметки `datetime1` временно́го промежутка `timedelta`;
- `timedelta = datetime1 - datetime2` — возвращает разницу между временны́ми отметками `datetime1` и `datetime2` в виде временно́го промежутка.

Также можно сравнивать две даты с помощью операторов сравнения. Примеры:

```
>>> d1 = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d2 = datetime.datetime(2021, 12, 31, 9, 5, 37, 438)
>>> t = datetime.timedelta(days=10, minutes=10)
>>> d1 + t                    # Прибавляем 10 дней и 10 минут
    datetime.datetime(2022, 1, 24, 14, 35, 21, 998076)
>>> d1 - t                    # Вычитаем 10 дней и 10 минут
    datetime.datetime(2022, 1, 4, 14, 15, 21, 998076)
>>> d1 - d2                   # Разница между датами
    datetime.timedelta(days=14, seconds=19184, microseconds=997638)
>>> d1 < d2, d1 > d2, d1 <= d2, d1 >= d2
    (False, True, False, True)
>>> d1 == d2, d1 != d2
    (False, True)
```

Можно получить строковое представление временно́й отметки с помощью функций `str()` и `repr()`:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> str(d), repr(d)
    ('2022-01-14 14:25:21.998076',
     'datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)')
```

Класс `datetime` поддерживает следующие методы:

- `date()` — возвращает текущую дату в виде объекта `date`:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.date()
    datetime.date(2022, 1, 14)
```

- `time()` — возвращает текущее время в виде объекта `time` без временно́й зоны:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.time()
    datetime.time(14, 25, 21, 998076)
```

- `timetz()` — возвращает текущее время в виде объекта `time` с временно́й зоной;
- `timestamp()` — возвращает количество секунд, прошедшее с начала эпохи (обычно с 1 января 1970 г.), в виде вещественного числа:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.timestamp()
    1642159521.998076
```

- `replace()` — возвращает текущую временну́ю отметку с обновленными значениями, взятыми из параметров. Формат метода:

```
replace([year][, month][, day][, hour][, minute][, second][,
                                microsecond][, tzinfo][, fold])
```

Параметры можно указывать как позиционные или как именованные, за исключением `fold`, который указывается только как именованный. Примеры:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.replace(2023, 12)
    datetime.datetime(2023, 12, 14, 14, 25, 21, 998076)
```

```
>>> d.replace(hour=6, minute=10, microsecond=38)
    datetime.datetime(2022, 1, 14, 6, 10, 21, 38)
```

- `timetuple()` — возвращает текущие дату и время в виде объекта `struct_time`:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.timetuple()
    time.struct_time(tm_year=2022, tm_mon=1, tm_mday=14, tm_hour=14,
    tm_min=25, tm_sec=21, tm_wday=4, tm_yday=14, tm_isdst=-1)
```

- `utctimetuple()` — возвращает текущие дату и время в виде объекта `struct_time` в универсальном времени (UTC):

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.utctimetuple()
    time.struct_time(tm_year=2022, tm_mon=1, tm_mday=14, tm_hour=14,
    tm_min=25, tm_sec=21, tm_wday=4, tm_yday=14, tm_isdst=0)
```

- `toordinal()` — возвращает количество дней, прошедшее с 1-го года:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.toordinal()
    738169
```

- `weekday()` — возвращает порядковый номер дня в неделе (0 — для понедельника, 6 — для воскресенья):

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.weekday()    # 4 — это пятница
    4
```

- `isoweekday()` — возвращает порядковый номер дня в неделе (1 — для понедельника, 7 — для воскресенья):

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.isoweekday() # 5 — это пятница
    5
```

- `isocalendar()` — возвращает последовательность из трех элементов: года, номера недели в году и порядкового номера дня в неделе:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> isoc = d.isocalendar()
>>> isoc[0], isoc[1], isoc[2]
    (2022, 2, 5)
```

- `isoformat([<Разделитель>][, ][timespec="auto"])` — возвращает текущие дату и время в формате ISO 8601. Если разделитель не указан, используется буква `T`. В параметре `timespec` можно указать состав выдаваемого времени, задав одно из следующих значений:
  - `"auto"` — часы, минуты, секунды и микросекунды, если количество последних не равно 0 — в противном случае микросекунды не включаются в состав времени;
  - `"hours"` — часы;
  - `"minutes"` — часы и минуты;

- "seconds" — часы, минуты и секунды;
- "milliseconds" — часы, минуты, секунды и миллисекунды (вычисляются путем округления микросекунд до тысячных);
- "microseconds" — часы, минуты, секунды и микросекунды (выводятся всегда, даже если их количество равно 0).

Примеры:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.isoformat()           # Разделитель не указан
    '2022-01-14T14:25:21.998076'
>>> d.isoformat(" ")        # Пробел в качестве разделителя
    '2022-01-14 14:25:21.998076'
>>> d.isoformat(timespec="minutes")
    '2022-01-14T14:25'
```

- ctime() — возвращает текущие дату и время в виде строки формата "%a %b %d %H:%M:%S %Y":

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.ctime()
    'Fri Jan 14 14:25:21 2022'
```

- strftime(<Строка формата>) — возвращает текущие дату и время в виде строки, отформатированной согласно заданной строке формата. В строке формата можно указывать комбинации специальных символов, которые используются в функции strftime() из модуля time. Пример:

```
>>> d = datetime.datetime(2022, 1, 14, 14, 25, 21, 998076)
>>> d.strftime("%d.%m.%Y %H:%M:%S")
    '14.01.2022 14:25:21'
```

Класс datetime поддерживает также следующие атрибуты:

- min — минимально возможное значение временно́й отметки;
- max — максимально возможное значение временно́й отметки;
- resolution — минимальное возможное различие между временны́ми отметками.

Значения этих атрибутов:

```
>>> datetime.datetime.min
    datetime.datetime(1, 1, 1, 0, 0)
>>> datetime.datetime.max
    datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)
>>> datetime.datetime.resolution
    datetime.timedelta(microseconds=1)
```

> ***Примечание***
>
> Класс datetime также поддерживает методы astimezone(), dst(), utcoffset() и tzname(). За подробной информацией по этим методам, а также по абстрактному классу tzinfo обращайтесь к документации по модулю datetime.

10.5. Вывод календаря

Модуль `calendar` формирует календарь в виде простого текста или HTML-кода. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import calendar
```

Класс `Calendar`, содержащийся в этом модуле, является базовым, от которого наследуют остальные классы календарей, описываемые далее. Формат конструктора:

```
Calendar([<Первый день недели>])
```

В параметре указывается число от 0 (для понедельника) до 6 (для воскресенья). Если параметр не указан, его значение принимается равным 0. Вместо чисел можно использовать значения переменных: `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` или `SUNDAY`, содержащихся в модуле `calendar`.

В качестве примера получим двумерный список всех дней в январе 2022 года, распределенных по дням недели:

```
>>> import calendar
>>> c = calendar.Calendar()
>>> c.monthdayscalendar(2022, 1)
    [[0, 0, 0, 0, 0, 1, 2], [3, 4, 5, 6, 7, 8, 9],
     [10, 11, 12, 13, 14, 15, 16], [17, 18, 19, 20, 21, 22, 23],
     [24, 25, 26, 27, 28, 29, 30], [31, 0, 0, 0, 0, 0, 0]]
```

В дальнейшем указать другой первый день недели у объекта календаря позволяет метод `setfirstweekday(<Первый день недели>)`. В качестве примера получим календарь на февраль 2022 года, где первым днем недели является воскресенье:

```
>>> c.setfirstweekday(calendar.SUNDAY)
>>> c.monthdayscalendar(2022, 2)
    [[0, 0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11, 12],
     [13, 14, 15, 16, 17, 18, 19], [20, 21, 22, 23, 24, 25, 26],
     [27, 28, 0, 0, 0, 0, 0]]
```

10.5.1. Вывод календаря в текстовом виде

Для формирования календаря в текстовом виде служат два класса из модуля `calendar`:

- `TextCalendar` — выводит названия месяцев и дней недели по-английски. Формат конструктора:

  ```
  TextCalendar([<Первый день недели>])
  ```

  Параметр указывается в таком же формате, что и у класса `Calendar` (см. *разд. 10.5*). Выведем календарь на весь 2022 год:

  ```
  >>> import calendar
  >>> c = calendar.TextCalendar()
  >>> print(c.formatyear(2022))
  ```

  Результатом станет большая строка, содержащая календарь в виде отформатированного текста;

- `LocaleTextCalendar` — выводит названия месяцев и дней недели в соответствии с указанной локалью. Формат конструктора:

  ```
  LocaleTextCalendar([<Первый день недели>[, <Название локали>]])
  ```

Выведем календарь на весь 2022 год на русском языке:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2022))
```

Классы `TextCalendar` и `LocaleTextCalendar` поддерживают следующие методы:

- `formatmonth()` — возвращает текстовый календарь на указанный месяц в заданном году. Формат метода:

```
formatmonth(<Год>, <Месяц>[, <Ширина поля с днем>[,
                            <Количество символов перевода строки>]])
```

Третий параметр задает ширину поля с днем, а четвертый параметр — количество символов перевода строки между строками. Выведем календарь на декабрь 2022 года:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatmonth(2022, 12))
      Декабрь 2022
Пн Вт Ср Чт Пт Сб Вс
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

- `prmonth()` — аналогичен методу `formatmonth()`, только сразу выводит календарь в консоль. Распечатаем календарь на декабрь 2022 года, указав ширину поля с днем, равной 4 символам:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.prmonth(2022, 12, 4)
              Декабрь 2022
  Пн    Вт    Ср    Чт    Пт    Сб    Вс
                    1     2     3     4
   5     6     7     8     9    10    11
  12    13    14    15    16    17    18
  19    20    21    22    23    24    25
  26    27    28    29    30    31
```

- `formatyear(<Год>[, w=2][, l=1][, c=6][, m=3])` — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:
  - `w` — ширина поля с днем;
  - `l` — количество символов перевода строки между строками;
  - `c` — количество пробелов между месяцами;
  - `m` — количество месяцев на строке.

Их можно указывать как позиционные или как именованные.

Сформируем календарь на 2023 год, выведем на одной строке сразу четыре месяца и установим количество пробелов между месяцами, равное 2:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2023, m=4, c=2))
```

- `pryear()` — аналогичен методу `formatyear()`, только сразу выводит календарь в консоль. Распечатаем календарь на 2023 год по два месяца на строке, расстояние между месяцами установим равным четырем символам, ширину поля с датой — равной двум символам, а строки разделим одним символом перевода строки:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.pryear(2023, 2, 1, 4, 2)
```

10.5.2. Вывод календаря в формате HTML

Для формирования календаря в формате HTML служат два класса из модуля `calendar`:

- `HTMLCalendar` — выводит названия месяцев и дней недели по-английски. Формат конструктора:

```
HTMLCalendar([<Первый день недели>])
```

Параметр указывается в таком же формате, что и у класса `Calendar` (см. *разд. 10.5*). Выведем календарь на весь 2022 год:

```
>>> import calendar
>>> c = calendar.HTMLCalendar()
>>> print(c.formatyear(2022))
```

Результатом будет большая строка с HTML-кодом календаря, отформатированного в виде таблицы;

- `LocaleHTMLCalendar` — выводит названия месяцев и дней недели в соответствии с указанной локалью. Формат конструктора:

```
LocaleHTMLCalendar([<Первый день недели>[, <Название локали>]])
```

Выведем календарь на весь 2022 год на русском языке в виде отдельной веб-страницы:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> xhtml = c.formatyearpage(2022, encoding="windows-1251")
>>> print(xhtml.decode("cp1251"))
```

Классы `HTMLCalendar` и `LocaleHTMLCalendar` поддерживают следующие методы:

- `formatmonth(<Год>, <Месяц>[, <True|False>])` — возвращает календарь на указанный месяц в заданном году в виде HTML-кода, содержащего только таблицу (тег `<table>`). Если в третьем параметре указано значение `True` (значение по умолчанию), то в заголовке таблицы после названия месяца будет указан год. Выведем календарь на январь 2022 года:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> print(c.formatmonth(2022, 1))
```

У каждой ячейки HTML-таблицы, формирующей календарь, указывается стилевой класс, соответствующий одному из дней недели. Благодаря этому к ячейкам можно привязывать разные CSS-стили. Имена этих стилевых классов доступны через атрибут `cssclasses` класса календаря:

```
>>> c = calendar.HTMLCalendar(0)
>>> print(c.cssclasses)
    ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun']
```

Начиная с Python 3.7, класс календаря поддерживает следующие атрибуты, задающие другие стилевые классы:

- `cssclass_noday` — имя стилевого класса, указываемого у пустых ячеек календаря (соответствующих дням предыдущего и следующего месяцев). По умолчанию: `"noday"`;
- `cssclasses_weekday_head` — список имен стилевых классов, указываемых у заголовочных ячеек с названиями дней недели (по умолчанию совпадает со списком из атрибута `cssclasses`);
- `cssclass_month_head` — имя стилевого класса, указываемого у заголовочной ячейки с названием месяца (по умолчанию: `"month"`);
- `cssclass_month` — имя стилевого класса, указываемого у таблицы, в которой выводится календарь за месяц (по умолчанию: `"month"`);
- `cssclass_year_head` — имя стилевого класса, указываемого у заголовочной ячейки с годом (по умолчанию: `"year"`);
- `cssclass_year` — имя стилевого класса, указываемого у таблицы, в которой выводится календарь за весь год (по умолчанию: `"year"`).

Выведем календарь на январь 2022 года, указав другие имена стилевых классов для ячеек с днями недели и заголовочной ячейки с месяцем:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> c.cssclasses = ["workday", "workday", "workday", "workday",
...                 "workday", "week-end", "week-end"]
>>> c.cssclass_month_head = "month-head"
>>> print(c.formatmonth(2022, 1, False))
```

- `formatyear(<Год>[, <Количество месяцев на строке>])` — возвращает календарь на указанный год в виде HTML-кода. Календарь будет отформатирован с помощью нескольких HTML-таблиц. Если количество месяцев в строке не задано, принимается значение 3. Выведем календарь на 2023 год так, чтобы на одной строке выводились сразу четыре месяца:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2023, 4))
```

- `formatyearpage(<Год>[, width][, css][, encoding])` — возвращает календарь на указанный год в виде отдельной веб-страницы. Параметры имеют следующее предназначение:
  - `width` — количество месяцев на строке (по умолчанию: 3);
  - `css` — название файла с таблицей стилей (по умолчанию: `"calendar.css"`);
  - `encoding` — кодировка файла, указываемая в параметре `encoding` XML-пролога и теге `<meta>` (по умолчанию не указана).

  Параметры можно указывать как позиционные или как именованные.

  Выведем календарь на 2023 год так, чтобы на одной строке выводились четыре месяца, дополнительно указав кодировку:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> xhtml = c.formatyearpage(2023, 4, encoding="windows-1251")
```

```
>>> type(xhtml) # Возвращаемая строка имеет тип данных bytes
    <class 'bytes'>
>>> print(xhtml.decode("cp1251"))
```

10.5.3. Другие полезные функции

Модуль `calendar` предоставляет еще ряд полезных функций:

- `setfirstweekday(<Первый день недели>)` — устанавливает первый день недели для календаря. В качестве параметра указывается число от 0 (для понедельника) до 6 (для воскресенья). Вместо чисел можно использовать значения переменных: `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` или `SUNDAY`. Получить текущее значение параметра можно с помощью функции `firstweekday()`. Установим воскресенье первым днем недели:

```
>>> import calendar
>>> calendar.firstweekday()      # По умолчанию 0
    0
>>> calendar.setfirstweekday(6) # Изменяем значение
>>> calendar.firstweekday()      # Проверяем установку
    6
```

- `month()` — возвращает текстовый календарь на указанный месяц в году. Формат функции:

```
month(<Год>, <Месяц>[, <Ширина поля с днем>[,
                      <Количество символов перевода строки>]])
```

Третий параметр указывает ширину поля с днем, а четвертый — количество символов перевода строки между строками. Выведем календарь на январь 2022 года:

```
>>> calendar.setfirstweekday(0)
>>> print(calendar.month(2022, 1))
        January 2022
    Mo Tu We Th Fr Sa Su
                    1  2
     3  4  5  6  7  8  9
    10 11 12 13 14 15 16
    17 18 19 20 21 22 23
    24 25 26 27 28 29 30
    31
```

- `prmonth()` — аналогична функции `month()`, только сразу выводит календарь в консоль. Выведем календарь на январь 2022 года:

```
>>> calendar.prmonth(2022, 1)
```

- `monthcalendar(<Год>, <Месяц>)` — возвращает двумерный список всех дней в указанном месяце указанного года, распределенных по дням недели. Дни, выходящие за пределы месяца, будут представлены нулями. Выведем массив для января 2022 года:

```
>>> calendar.monthcalendar(2022, 1)
    [[0, 0, 0, 0, 0, 1, 2], [3, 4, 5, 6, 7, 8, 9],
     [10, 11, 12, 13, 14, 15, 16], [17, 18, 19, 20, 21, 22, 23],
     [24, 25, 26, 27, 28, 29, 30], [31, 0, 0, 0, 0, 0, 0]]
```

- `monthrange(<Год>, <Месяц>)` — возвращает кортеж из двух элементов: номера дня недели, приходящегося на первое число указанного месяца указанного года, и количества дней в месяце:

```
>>> print(calendar.monthrange(2022, 1))
    (5, 31)
>>> # Январь 2022 года начинается с субботы (5) и включает 31 день
```

- `calendar(<Год>[, w=2][, l=1][, c=6][, m=3])` — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:
  - `w` — ширина поля с днем;
  - `l` — количество символов перевода строки между строками;
  - `c` — количество пробелов между месяцами;
  - `m` — количество месяцев на строке.

  Параметры можно указывать как позиционные или как именованные.

  Выведем календарь на 2023 год так, чтобы на одной строке выводились сразу четыре месяца, указав два пробела между месяцами:

```
>>> print(calendar.calendar(2023, m=4, c=2))
```

- `prcal()` — аналогична функции `calendar()`, только сразу выводит его календарь в консоль. Выведем календарь на 2023 год по два месяца на строке, расстояние между месяцами установим равным четырем символам, ширину поля с датой — равной двум символам, а строки разделим одним символом перевода строки:

```
>>> calendar.prcal(2023, 2, 1, 4, 2)
```

- `weekheader(<Длина>)` — возвращает строку, которая содержит аббревиатуры дней недели с учетом текущей локали, разделенные пробелами. Единственный параметр задает длину каждой аббревиатуры в символах. Примеры:

```
>>> calendar.weekheader(4)
    'Mon  Tue  Wed  Thu  Fri  Sat  Sun '
>>> calendar.weekheader(2)
    'Mo Tu We Th Fr Sa Su'
>>> import locale                  # Задаем другую локаль
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
    'Russian_Russia.1251'
>>> calendar.weekheader(2)
    'Пн Вт Ср Чт Пт Сб Вс'
```

- `isleap(<Год>)` — возвращает значение `True`, если указанный год является високосным, в противном случае — `False`:

```
>>> calendar.isleap(2022), calendar.isleap(2024)
    (False, True)
```

- `leapdays(<Год 1>, <Год 2>)` — возвращает количество високосных лет в диапазоне от `<Год 1>` до `<Год 2>` (`<Год 2>` не учитывается):

```
>>> calendar.leapdays(2021, 2024) # 2024 не учитывается
    0
>>> calendar.leapdays(2020, 2024) # 2020 — високосный год
    1
```

```
>>> calendar.leapdays(2020, 2025) # 2020 и 2024 — високосные года
    2
```

- `weekday(<Год>, <Месяц>, <День>)` — возвращает номер дня недели (0 — для понедельника, 6 — для воскресенья):

```
>>> calendar.weekday(2022, 1, 17)
    0
```

- `timegm(<Объект struct_time>)` — возвращает количество секунд, прошедших с начала эпохи. В качестве параметра указывается объект `struct_time` с датой и временем, возвращаемый функцией `gmtime()` из модуля `time`. Примеры:

```
>>> import calendar, time
>>> d = time.gmtime(1639725843.0) # Дата 17-12-2021
>>> d
    time.struct_time(tm_year=2021, tm_mon=12, tm_mday=17, tm_hour=7,
    tm_min=24, tm_sec=3, tm_wday=4, tm_yday=351, tm_isdst=0)
>>> tuple(d)
    (2021, 12, 17, 7, 24, 3, 4, 351, 0)
>>> calendar.timegm(d)
    1639725843
>>> calendar.timegm((2021, 12, 17, 7, 24, 3, 4, 351, 0))
    1639725843
```

Модуль `calendar` также поддерживает ряд полезных атрибутов:

- `day_name` — список полных названий дней недели в текущей локали:

```
>>> [i for i in calendar.day_name]
    ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
     'Saturday', 'Sunday']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
    'Russian_Russia.1251'
>>> [i for i in calendar.day_name]
    ['понедельник', 'вторник', 'среда', 'четверг', 'пятница', 'суббота',
     'воскресенье']
```

- `day_abbr` — список аббревиатур названий дней недели в текущей локали:

```
>>> [i for i in calendar.day_abbr]
    ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
    'Russian_Russia.1251'
>>> [i for i in calendar.day_abbr]
    ['Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб', 'Вс']
```

- `month_name` — список полных названий месяцев в текущей локали:

```
>>> [i for i in calendar.month_name]
    ['', 'January', 'February', 'March', 'April', 'May', 'June',
     'July', 'August', 'September', 'October', 'November', 'December']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
    'Russian_Russia.1251'
```

```
>>> [i for i in calendar.month_name]
    ['', 'Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', 'Июль',
     'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь']
```

- `month_abbr` — список аббревиатур названий месяцев в текущей локали:

```
>>> [i for i in calendar.month_abbr]
    ['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
     'Sep', 'Oct', 'Nov', 'Dec']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
    'Russian_Russia.1251'
>>> [i for i in calendar.month_abbr]
    ['', 'янв', 'фев', 'мар', 'апр', 'май', 'июн', 'июл', 'авг', 'сен',
     'окт', 'ноя', 'дек']
```

10.6. Измерение времени выполнения фрагментов кода

Модуль `timeit` предоставляет инструменты для измерения времени выполнения фрагментов кода программы с целью ее оптимизации.

Измерения производятся с помощью класса `Timer`, который сначала следует импортировать из модуля `timeit` посредством инструкции:

```
from timeit import Timer
```

Конструктор класса имеет следующий формат:

```
Timer([stmt='pass'][, setup='pass'])
```

В параметре `stmt` указывается код (в виде строки), время выполнения которого нужно измерить. Параметр `setup` задает код, выполняемый перед измерением времени выполнения кода, из параметра `stmt` (например, код из параметра `setup` может подключать необходимые модули).

Получить время выполнения можно с помощью метода `timeit([number=1000000])`. В параметре `number` указывается количество повторений.

Для примера просуммируем числа от 1 до 10 000 тремя способами и выведем время выполнения каждого способа (листинг 10.3).

Листинг 10.3. Измерение времени выполнения

```
from timeit import Timer
code1 = """\
i, j = 1, 0
while i < 10001:
    j += i
    i += 1
"""
t1 = Timer(stmt=code1)
print("while:", t1.timeit(number=10000))
code2 = """\
j = 0
```

```
for i in range(1, 10001):
    j += i
"""
t2 = Timer(stmt=code2)
print("for:", t2.timeit(number=10000))
code3 = """\
j = sum(range(1, 10001))
"""
t3 = Timer(stmt=code3)
print("sum:", t3.timeit(number=10000))
input()
```

Примерный результат выполнения (зависит от мощности компьютера):

```
while: 10.956968600000437
for: 6.401410300000862
sum: 2.3285845999998855
```

Сразу видно, что цикл перебора последовательности работает почти в два раза быстрее цикла с условием, а функция `sum()` в нашем случае вовсе вне конкуренции.

Метод `repeat([repeat=5][, number=1000000])` вызывает метод `timeit()` указанное в параметре `repeat` количество раз и возвращает список значений (до Python 3.7 этот параметр имел значение по умолчанию 3). Аргумент `number` передается в качестве параметра методу `timeit()`.

Для примера создадим список со строковыми представлениями чисел от 1 до 10 000: в первом случае для создания списка используем цикл перебора последовательности и метод `append()`, а во втором — генератор списков (листинг 10.4).

Листинг 10.4. Использование метода `repeat()`

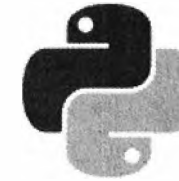
```
from timeit import Timer
code1 = """\
arr1 = []
for i in range(1, 10001):
    arr1.append(str(i))
"""
t1 = Timer(stmt=code1)
print("append:", t1.repeat(repeat=3, number=2000))
code2 = """\
arr2 = [str(i) for i in range(1, 10001)]
"""
t2 = Timer(stmt=code2)
print("генератор:", t2.repeat(repeat=3, number=2000))
input()
```

Примерный результат выполнения:

```
append: [4.658220999999685, 4.674832300000162, 4.590558100000635]
генератор: [3.93614129999969, 3.8754656000000978, 3.8798187000002144]
```

Как видно из результата, генераторы списков работают быстрее.

ГЛАВА 11



Функции

Функция — это языковая конструкция, выполняющая над переданными ей значениями (*параметрами*) какое-либо сложное действие. Так, функция `sqrt()` вычисляет квадратный корень, функция `print()` выводит заданные значения в консоль, а функция `strftime()` форматирует указанные дату и время сообразно заданному формату.

Функция может принимать произвольное количество параметров, в том числе и ни одного. Также функция может возвращать результат выполненного ею действия.

Язык Python содержит огромное количество функций, которые реализованы в нем самом и называются *встроенными*. Однако мы можем добавить к ним свои собственные, *пользовательские*, функции. Которым и посвящена эта глава.

11.1. Определение и вызов функции

Инструкция создания, или *определения*, пользовательской функции записывается в следующем формате:

```
def <Имя функции>([<Имена параметров через запятую>]):
    <Тело функции>
```

Имя функции должно быть уникальным, состоять из латинских букв, цифр и знаков подчеркивания и не начинаться с цифры. В качестве имени нельзя использовать ключевые слова Python, кроме того, следует избегать совпадений с названиями встроенных идентификаторов. Регистр символов в имени функции также имеет значение.

Имена параметров должны удовлетворять тем же требованиям, что и имена переменных (см. *разд. 2.1*). Если функция не принимает параметров, следует указать пустые круглые скобки.

Тело функции — это блок, реализующий функциональность определяемой функции. Не забываем, что инструкции, входящие в блок, должны иметь одинаковые отступы слева, выполненные пробелами.

При вызове функции создаются переменные, имеющие те же имена, что и записанные в определении функции параметры. Эти переменные будут хранить реальные значения параметров, заданные в выражении вызова функции, станут доступны лишь в теле функции (*локальные переменные*), а после завершения выполнения функции — удалены из памяти.

Пример определения функции, принимающей два параметра, делящей их друг на друга и выводящей результат в консоль:

```
>>> def division(a, b):
...     res = a / b
...     print(str(a), " / ", str(b), " = ", str(res))
...
```

Если тело функции состоит из одного выражения, его можно набрать в одной строке с языковой конструкцией `def`. Пример определения функции, не принимающей параметров и выводящей в консоль случайное число:

```
>>> def rnd(): print(random.random())
...
```

Для возврата результата из функции в ее теле записывается инструкция такого формата:

```
return <Возвращаемое значение>
```

Пример функции, принимающей два параметра, делящей их друг на друга и возвращающей в качестве результата строку с получившимся частным:

```
>>> def division2(a, b):
...     res = a / b
...     s = str(a) + " / " + str(b) + " = " + str(res)
...     return s
...
```

После выполнения инструкции, возвращающей результат, исполнение тела функции завершается:

```
>>> def func():
...     print("Текст до инструкции return")
...     return "Возвращаемый результат"
...     print("Эта инструкция никогда не будет выполнена")
...
```

Если в теле функции отсутствует инструкция, возвращающая результат, функция при вызове все равно будет неявно возвращать в качестве результата значение `None`.

Обращение к ранее определенной функции называется *вызовом*. И встроенные, и пользовательские функции вызываются одинаково — записью выражения в формате:

```
<Имя функции>([<Значения параметров через запятую>])
```

Количество параметров в выражении вызова функции должно совпадать с количеством параметров в ее определении, иначе будет выведено сообщение об ошибке. Если вызываемая функция не принимает параметров, все равно следует указать пустые круглые скобки. Примеры:

```
>>> # Вызываем функцию, принимающую параметры
>>> division(1, 2)
    1  /  2  =  0.5
>>> division(2, 1)
    2  /  1  =  2.0
>>> # Вызываем функцию, не принимающую параметров
>>> rnd()
    0.5221364008680583
```

Результат, возвращенный функцией, можно сохранить в какой-либо переменной или использовать в дальнейших вычислениях:

```
>>> n = division2(4, 6)
>>> n
    '4 / 6 = 0.6666666666666666'
>>> print("Получен результат: ", division2(7, 5))
    Получен результат:  7 / 5 = 1.4
```

Также его можно нигде не сохранять и никак не использовать — в таком случае результат будет потерян. Так поступают, если результат выполнения функции не нужен для работы. Пример:

```
division2(4, 6)                 # Результат выполнения функции потерян
```

Поскольку в теле функции для хранения полученных параметров используются специально создаваемые локальные переменные, значения параметров могут быть изменены в теле функции — и это не вызовет никаких сторонних эффектов в коде, вызвавшем функцию. Вот пример:

```
>>> def func(a):
...     print("Внутри функции: ", a)
...     a = 20                  # Изменяем значение параметра внутри функции
...     print("Внутри функции: ", a)
...
>>> b = 30                      # Создаем переменную b
>>> print("Вне функции: ", b)
    Вне функции:  30
>>> func(b)                     # Передаем функции значение из переменной b
    Внутри функции:  30
    Внутри функции:  20
>>> print("Вне функции: ", b)  # Значение переменной b не изменилось
    Вне функции:  30
```

Однако в теле функции можно изменять значения изменяемых типов (например, списков), созданных вне функции (поскольку в функцию передаются не сами эти значения, а ссылки на них):

```
>>> def func(lst): lst.append("Python"); lst.append("Django")
...
>>> platforms = ["PHP", "Laravel"]
>>> func(platforms)
>>> platforms
    ['PHP', 'Laravel', 'Python', 'Django']
```

11.1.1. Расположение определений функций

Определение функции должно быть расположено перед ее вызовом. Правильно:

```
def summa(x, y):
    return x + y
v = summa(10, 20)
```

Неправильно (возникнет ошибка: `NameError: name 'summa' is not defined`):

```
v = summa(10, 20)
def summa(x, y):
    return x + y
```

Чтобы избежать ошибки, определения функций размещают в самом начале программы после подключения модулей или в отдельном модуле (о модулях речь пойдет в *главе 12*).

Если определения функций располагаются в том же модуле, что и их вызовы, определения функций часто отделяют от остального кода — для наглядности.

С помощью инструкции ветвления можно при выполнении заданного условия определить одну функцию, а при его невыполнении — другую (листинг 11.1).

Листинг 11.1. Определение функции в зависимости от выполнения условия

```
n = input("Введите 1 для вызова первой функции: ")
if n == "1":
    def echo():
        print("Вы ввели число 1")
else:
    def echo():
        print("Альтернативная функция")

echo() # Вызываем функцию
input()
```

При вводе числа 1 мы получим сообщение `"Вы ввели число 1"`, в противном случае — `"Альтернативная функция"`.

Если определение одной и той же функции встречается в программе несколько раз, будет использоваться функция, которая была определена последней:

```
def echo():
    print("Вы ввели число 1")
def echo():
    print("Альтернативная функция")
echo() # Всегда выводит "Альтернативная функция"
```

11.1.2. Локальные и глобальные переменные

В теле функции можно создавать любые переменные. Они будут существовать только в теле функции и по завершении ее выполнения уничтожатся. Такие переменные называются *локальными*. Пример функции, в теле которой создаются локальные переменные `res` и `s`:

```
>>> def division2(a, b):
...     res = a / b
...     s = str(a) + " / " + str(b) + " = " + str(res)
...     return s
...
>>> division(4, 6)    # Вызываем функцию
    '4 / 6 = 0.6666666666666666'
>>> res              # Безуспешно пытаемся обратиться к локальной переменной
    ... Фрагмент пропущен ...
    NameError: name 'res' is not defined
```

Как отмечалось ранее, сам интерпретатор для размещения значений параметров, переданных функции при вызове, создает в ней локальные переменные с именами, совпадающими

с именами параметров, которые указаны в определении функции. Так, при вызове функции, определенной в предыдущем примере, будут созданы локальные переменные a и b.

Переменные, созданные вне любых функций, называются *глобальными*. Они доступны в теле любой функции — *но только на чтение*. Глобальную переменную можно создать как перед определением функции, так и после него. Пример:

```
>>> def func():
...     print(glob)
...
>>> glob = 10
>>> func()
    10
```

При попытке в теле функции присвоить глобальной переменной новое значение создается одноименная локальная переменная, к которой в дальнейшем и будет выполняться обращение:

```
>>> def func2():
...     glob = 20        # Будет создана локальная переменная glob
...     print(glob)
...
>>> func2()
    20
>>> glob                 # Значение глобальной переменной не изменилось
    10
```

Однако попытка в теле функции присвоить значение локальной переменной после обращения к одноименной глобальной переменной приведет к возбуждению исключения UnboundLocalError:

```
>>> def func():
...     print(glob)
...     glob = 20
...
>>> func()
    ... Фрагмент пропущен ...
    UnboundLocalError: local variable 'glob' referenced before assignment
```

Чтобы значения глобальных переменных можно было изменять внутри функции, необходимо в теле функции объявить эти переменные глобальными с помощью языковой конструкции формата:

```
global <Имена переменных через запятую>
```

Продемонстрируем это на примере (листинг 11.2).

Листинг 11.2. Использование языковой конструкции global

```
def func():
    # Объявляем переменную glob глобальной
    global glob
    glob = 25        # Изменяем значение глобальной переменной
    print("Значение glob внутри функции =", glob)
```

```
glob = 10              # Глобальная переменная
print("Значение glob вне функции =", glob)
func()                  # Вызываем функцию
print("Значение glob после функции =", glob)
```

Результат выполнения:

```
Значение glob вне функции = 10
Значение glob внутри функции = 25
Значение glob после функции = 25
```

Получить все идентификаторы и их значения позволяют следующие функции:

- `globals()` — возвращает словарь с глобальными идентификаторами;
- `locals()` — возвращает словарь с локальными идентификаторами.

Пример использования обеих этих функций показан в листинге 11.3.

Листинг 11.3. Использование функций `globals()` и `locals()`

```
def func():
    local1 = 54
    glob2 = 25
    print("Глобальные идентификаторы внутри функции")
    print(sorted(globals().keys()))
    print("Локальные идентификаторы внутри функции")
    print(sorted(locals().keys()))

glob1, glob2 = 10, 88
func()
print("Глобальные идентификаторы вне функции")
print(sorted(globals().keys()))
```

Результат выполнения:

```
Глобальные идентификаторы внутри функции
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'func', 'glob1', 'glob2']
Локальные идентификаторы внутри функции
['glob2', 'local1']
Глобальные идентификаторы вне функции
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'func', 'glob1', 'glob2']
```

- `vars([<Объект>])` — если вызывается без параметра внутри функции, возвращает словарь с локальными идентификаторами. Если вызывается без параметра вне функции, возвращает словарь с глобальными идентификаторами. При указании объекта в качестве параметра возвращает идентификаторы этого объекта (эквивалентно вызову `<Объект>.__dict__`). Пример использования этой функции можно увидеть в листинге 11.4.

Листинг 11.4. Использование функции `vars()`

```
def func():
    local1 = 54
```

```
        glob2 = 25
        print("Локальные идентификаторы внутри функции")
        print(sorted(vars().keys()))

glob1, glob2 = 10, 88
func()
print("Глобальные идентификаторы вне функции")
print(sorted(vars().keys()))
print("Указание объекта в качестве параметра")
print(sorted(vars(dict).keys()))
print("Альтернативный вызов")
print(sorted(dict.__dict__.keys()))
```

11.1.3. Позиционные и именованные параметры

Ранее отмечалось, что при вызове функции ее параметры указываются через запятую внутри круглых скобок:

```
>>> division(1, 2)
```

Такие параметры носят название *позиционных*. Их значения передаются в вызываемую функцию согласно их позициям — в том порядке, в котором параметры указаны в выражении ее вызова. Так, в случае функции `division()` значение 1 будет присвоено параметру `a` (точнее, соответствующей ему локальной переменной, созданной интерпретатором, подробности — в *разд. 11.1*), а значение 2 — параметру `b`.

Однако для вызова функции можно использовать выражение и другого формата:

```
<Имя функции>([<Параметр 1>=<Значение параметра 1>,
               <Параметр 2>=<Значение параметра 2>,
               . . .,
               <Параметр N>=<Значение параметра N>])
```

Здесь применяются *именованные* параметры, значения которых передаются вызываемой функции по их именам, например:

```
>>> division(a=1, b=2)
    1 / 2 = 0.5
```

Именованные параметры в выражении вызова функции можно располагать в произвольном порядке:

```
>>> division(b=2, a=1)
    1 / 2 = 0.5
```

При вызове функции можно использовать и позиционные, и именованные параметры. Однако именованные параметры должны располагаться после позиционных, иначе возникнет ошибка. Примеры:

```
>>> def func(a, b, c):
...     print("a =>", str(a), " b =>", str(b), " c =>", str(c))
...
>>> func(1, 2, c=3)
    a => 1  b => 2  c => 3
```

```
>>> func(1, b=2, c=3)
    a => 1  b => 2  c => 3
>>> func(1, c=2, b=3)
    a => 1  b => 3  c => 2
>>> func(a=1, 2, c=3)
    SyntaxError: positional argument follows keyword argument
```

При попытке передать функции не поддерживаемый ею именованный параметр возникнет ошибка:

```
>>> func(1, 2, 3, d=4)
    ... Фрагмент пропущен ...
    TypeError: func() got an unexpected keyword argument 'd'
```

По умолчанию любой параметр функции может быть указан и как позиционный, и как именованный (*двоякий параметр*). Однако существует возможность пометить одни параметры как задаваемые только в качестве именованных (*строго именованные*), а другие — только как позиционные (*строго позиционные* — поддерживаются, начиная с Python 3.8). Для этого языковая конструкция `def`, входящая в состав определения функции, записывается в следующем формате:

```
def <Имя функции>(<Строго позиционные параметры>, /,
                  <Двоякие параметры>, *, <Строго именованные параметры>)
```

Пример функции с двумя строго позиционными, тремя двоякими и двумя строго именованными параметрами:

```
>>> def func(a, b, /, c, d, e, *, f, g):
...     print("a =>", str(a), " b =>", str(b), end="  ")
...     print("c =>", str(c), " d =>", str(d), " e =>", str(e), end="  ")
...     print("f =>", str(f), " g =>", str(g))
...
>>> func(1, 2, 3, 4, 5, f=6, g=7)
    a => 1  b => 2  c => 3  d => 4  e => 5  f => 6  g => 7
>>> func(1, 2, 3, d=4, e=5, f=6, g=7)
    a => 1  b => 2  c => 3  d => 4  e => 5  f => 6  g => 7
```

Попытка передать строго позиционный параметр как именованный или строго именованный как позиционный приведет к ошибке:

```
>>> func(1, b=2, c=3, d=4, e=5, f=6, g=7)
    ... Фрагмент пропущен ...
    TypeError: func() got some positional-only arguments passed as keyword
    arguments: 'b'
>>> func(1, 2, 3, 4, 5, 6, g=7)
    ... Фрагмент пропущен ...
    TypeError: func() takes 5 positional arguments but 6 positional arguments
    (and 1 keyword-only argument) were given
```

Выражение вызова функции с позиционными параметрами имеет более компактную запись. Однако использование именованных параметров позволяет сделать ее нагляднее. Особенно удобно применять именованные параметры для вызова функций с большим числом необязательных параметров (о них рассказано в *разд. 11.1.4*).

11.1.4. Необязательные параметры

Есть возможность сделать некоторые (или даже все) параметры пользовательской функции необязательными к указанию (*необязательные* параметры). Если при вызове функции такой параметр не задан, он получит заданное у него значение по умолчанию.

Необязательный параметр указывается в определении функции в следующем формате:

```
<Имя параметра>=<Значение параметра по умолчанию>
```

Необязательные параметры должны следовать после обязательных к указанию, иначе возникнет ошибка. Пример функции, у которой второй параметр помечен как необязательный и имеет значение по умолчанию 2:

```
>>> def division(a, b=2):
...     print(str(a), " / ", str(b), " = ", str(a / b))
...
>>> division(5)
    5  /  2  =  2.5
>>> division(5, 3)
    5  /  3  =  1.6666666666666667
>>> division(5, b=12)
    5  /  12  =  0.4166666666666667
```

Пример функции, имеющей два необязательных параметра:

```
>>> def func(a, b=2, c=3):
...     print("a =>", str(a), " b =>", str(b), " c =>", str(c))
...
>>> func(1)
    a => 1  b => 2  c => 3
>>> func(1, 200)
    a => 1  b => 200  c => 3
```

Если при вызове функции требуется указать значение не первого из необязательных параметров, следует задать значения у всех предшествующих ему необязательных параметров (хотя бы равные указанным у них значениям по умолчанию) — в противном случае возникнет ошибка. Например, предположим, что нужно вызвать функцию `func()`, указав значение у третьего (второго необязательного) параметра:

```
>>> func(1, 2, 3456789)   # Задаем второй параметр — все работает
    a => 1  b => 2  c => 3456789
>>> func(1, , 3456789)    # Пропускаем второй параметр — получаем ошибку
    SyntaxError: invalid syntax
```

Хотя проще задать нужный параметр как именованный (если определение функции позволяет это):

```
>>> func(1, c=512)
    a => 1  b => 2  c => 512
```

Значение по умолчанию, указанное у необязательного параметра, вычисляется только один раз — при определении функции:

```
>>> n=2    # Создаем переменную для хранения значения параметра по умолчанию
>>> def division(a, b=n):
...     print(str(a), " / ", str(b), " = ", str(a / b))
...
```

```
>>> division(5)
    5 / 2 = 2.5
>>> n=3              # Изменяем значение переменной
>>> division(5)      # Значение параметра по умолчанию не изменилось
    5 / 2 = 2.5
```

Если в качестве значения по умолчанию указать значение изменяемого типа, это значение будет сохраняться между вызовами функции:

```
>>> def func(a=[]):
...     a.append(2)
...     return a
...
>>> print(func())                  # Выведет: [2]
>>> print(func())                  # Выведет: [2, 2]
>>> print(func())                  # Выведет: [2, 2, 2]
```

Как видно из примера, значения накапливаются внутри списка. Обойти эту проблему можно, например, следующим образом:

```
>>> def func(a=None):
...     # Создаем новый список, если значение равно None
...     if a is None:
...         a = []
...     a.append(2)
...     return a
...
>>> print(func())                  # Выведет: [2]
>>> print(func([1]))               # Выведет: [1, 2]
>>> print(func())                  # Выведет: [2]
```

11.1.5. Произвольное количество параметров

Наконец, можно определять функции, принимающие произвольное количество параметров. Для этого в определении функции в составе набора принимаемых параметров следует указать одну из следующих языковых конструкций или сразу обе:

- `*<Имя параметра>` — параметр с заданным именем получит в качестве значения кортеж, содержащий значения всех переданных функции *позиционных* параметров;
- `**<Имя параметра>` — параметр с заданным именем получит в качестве значения словарь, содержащий значения всех переданных функции *именованных* параметров. В качестве ключей элементов словаря будут выступать имена параметров.

Примеры:

```
>>> def func1(*pars): print(pars)
...
>>> func1(1, 2, 3, 4)
    (1, 2, 3, 4)
>>> func1(1, 2, 3, 4, 5, 6, "abc")
    (1, 2, 3, 4, 5, 6, 'abc')
>>> def func2(**pars): print(pars)
...
```

```
>>> func2(a=1, b=2, c=3)
    {'a': 1, 'b': 2, 'c': 3}
>>> func2(a=1, b=2, c=3, i=9, j="def")
    {'a': 1, 'b': 2, 'c': 3, 'i': 9, 'j': 'def'}
>>> def func3(*pars1, **pars2): print(pars1); print(pars2)
...
>>> func3(1, 2, a=3, bcd=4)
    (1, 2)
    {'a': 3, 'bcd': 4}
>>> func3(5, 6, 7)
    (5, 6, 7)
    {}
>>> func3(s="a", t="b", v="c")
    ()
    {'s': 'a', 't': 'b', 'v': 'c'}
```

Можно комбинировать упомянутые языковые конструкции с обычными параметрами, использовав следующий формат:

```
def <Имя функции>([<Строго позиционные параметры>, /,]
                  <Двоякие параметры>[,
                  *<Список позиционных параметров>][,
                  <Строго именованные параметры>][,
                  **<Словарь именованных параметров>])
```

Пример функции, принимающей один обязательный строго позиционный параметр, один обязательный двоякий, произвольное количество позиционных параметров, один обязательный строго именованный и произвольное количество именованных:

```
>>> def func4(a, /, b, *c, d, **e):
...  print(a, b, c, d, e)
...
>>> func4(1, 2, 3, 4, 5, d=6, e=7, f=8, g=9)
    1 2 (3, 4, 5) 6 {'e': 7, 'f': 8, 'g': 9}
>>> func4(1, b=2, d=6, e=7)
    1 2 () 6 {'e': 7}
>>> func4(1, 2, 3, 4, 5, e=7, f=8, g=9)
    ... Фрагмент пропущен ...
    TypeError: func4() missing 1 required keyword-only argument: 'd'
```

Пример функции, принимающей один необязательный двоякий параметр, произвольное количество позиционных и один необязательный именованный:

```
>>> def func5(a=1, *b, c=100): print(a, b, c)
...
>>> func5(1, 2, 3, 4, 5)
    1 (2, 3, 4, 5) 100
>>> func5(a=10, c=20)
    10 () 20
>>> func5()
    1 () 100
```

11.1.6. Распаковка последовательностей и отображений

Если значения параметров, которые требуется передать в функцию, содержатся в последовательности (списке или кортеже) или отображении (словаре), то в вызове функции вместо параметров можно указать одну из следующих языковых конструкций или сразу обе:

- `*<Последовательность>` — значения элементов заданной последовательности будут переданы вызываемой функции в качестве *позиционных* параметров;
- `**<Отображение>` — значения элементов заданного отображения будут переданы вызываемой функции в качестве *именованных* параметров.

Подобного рода передача в функцию параметров из последовательностей и отображений называется *распаковкой*. Ее пример приведен в листинге 11.5.

Листинг 11.5. Распаковка последовательностей и отображений

```
def summa(a, b, c):
    return a + b + c

seq1 = [1, 2, 3]
print(summa(*seq1))                 # Распаковка списка
seq2 = (2, 3)
print(summa(1, *seq2))              # Можно совмещать указание
                                    # обычных параметров и распаковку
mpn1 = {"a": 1, "b": 2, "c": 3}
print(summa(**mpn1))                # Распаковка словаря
seq3, mpn2 = (1, 2), {"c": 3}
print(summa(*seq3, **mpn2))         # Можно совмещать распаковку
                                    # списков и словарей
```

11.1.7. Функция как значение. Функции обратного вызова

Определение функции (неважно, пользовательской или встроенной) представляет собой объект, имеющий тип `function`, т. е. значение. Следовательно, функцию (точнее, ссылку на нее) можно присвоить какой-либо переменной, а позже вызвать через эту переменную. Пример показан в листинге 11.6

Листинг 11.6. Сохранение ссылки на функцию в переменной

```
def summa(x, y):
    return x + y

f = summa                           # Сохраняем ссылку в переменной f
print(f(10, 20))                    # Вызываем функцию через переменную f
```

Также можно поместить функцию в последовательность или отображение и даже передать функцию другой функции в качестве параметра (листинг 11.5). Функции, передаваемые другим функциям, носят название *функций обратного вызова*.

Листинг 11.7. Функции обратного вызова

```
def summa(x, y):
    return x + y

def func(f, a, b):
    """ Через переменную f будет доступна ссылка на
        функцию summa() """
    return f(a, b) # Вызываем функцию summa()

# Передаем ссылку на функцию в качестве параметра
print(func(summa, 10, 20))
```

Объекты функций поддерживают множество атрибутов. Например, атрибут `__name__` хранит имя функции в виде строки, атрибут `__doc__` — строку документирования и т. д. Для примера выведем имена всех атрибутов функции с помощью встроенной функции `dir()`:

```
>>> def summa(x, y):
...     """ Суммирование двух чисел """
...     return x + y
...
>>> dir(summa)
    ['__annotations__', '__builtins__', '__call__', '__class__', '__closure__',
     '__code__', '__defaults__', '__delattr__', '__dict__', '__dir__',
     '__doc__', '__eq__', '__format__', '__ge__', '__get__',
     '__getattribute__', '__globals__', '__gt__', '__hash__', '__init__',
     '__init_subclass__', '__kwdefaults__', '__le__', '__lt__', '__module__',
     '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
     '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
     '__subclasshook__']
>>> summa.__name__
    'summa'
>>> summa.__code__.co_varnames
    ('x', 'y')
>>> summa.__doc__
    ' Суммирование двух чисел '
```

11.2. Анонимные функции

Анонимная функция (или *лямбда-функция*), в отличие от обычной, не имеет имени и поэтому сразу после определения должна быть присвоена какой-либо переменной или передана какой-либо другой функции — в противном случае она будет потеряна. Формат определения анонимной функции:

```
lambda [<Параметр 1>[, . . ., <Параметр N>]]: <Возвращаемый результат>
```

Пример использования анонимных функций приведен в листинге 11.8.

Листинг 11.8. Анонимные функции

```
f1 = lambda: 10 + 20                # Функция без параметров
f2 = lambda x, y: x + y             # Функция с двумя параметрами
```

```
f3 = lambda x, y, z: x + y + z      # Функция с тремя параметрами
print(f1())                         # Выведет: 30
print(f2(5, 10))                    # Выведет: 15
print(f3(5, 10, 30))                # Выведет: 45
```

Анонимные функции могут иметь необязательные параметры (листинг 11.17).

Листинг 11.9. Анонимная функция с необязательным параметром

```
f = lambda x, y=2: x + y
print(f(5))                         # Выведет: 7
print(f(5, 6))                      # Выведет: 11
```

Чаще всего анонимные функции используют в качестве функций обратного вызова (см. *разд. 11.6.7*). Например, метод списков `sort()` позволяет указать пользовательскую функцию в параметре `key`. Отсортируем список без учета регистра символов, указав в качестве параметра анонимную функцию (листинг 11.10).

Листинг 11.10. Сортировка без учета регистра символов

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=lambda s: s.lower())
for i in arr:
    print(i, end=" ")
# Результат выполнения: единица1 Единица2 Единый
```

При указании глобальной переменной внутри анонимной функции будет сохранена ссылка на эту переменную, а не ее значение в момент определения функции:

```
>>> x = 5
>>> func = lambda: x          # Сохраняется ссылка, а не значение переменной x!!!
>>> x = 80                    # Изменили значение
>>> print(func())             # Выведет: 80, а не 5
```

Если необходимо сохранить именно текущее значение глобальной переменной, можно воспользоваться следующим способом:

```
>>> x = 5
>>> func = (lambda y: lambda: y)(x)      # Сохраняется значение переменной x
>>> x = 80                               # Изменили значение
>>> print(func())                        # Выведет: 5
```

Во второй строке кода мы определили анонимную функцию с одним параметром, возвращающую ссылку на вложенную анонимную функцию. Далее мы вызываем первую функцию с помощью круглых скобок и передаем ей значение переменной `x`. В результате сохраняется текущее значение переменной, а не ссылка на нее.

Сохранить текущее значение глобальной переменной также можно, указав эту переменную в качестве значения параметра по умолчанию:

```
>>> x = 5
>>> func = lambda x=x: x                 # Сохраняется значение переменной x
>>> x = 80                               # Изменили значение
>>> print(func())                        # Выведет: 5
```

11.3. Функции-генераторы

Функция-генератор при последовательных вызовах возвращает один за другим элементы какой-либо последовательности. Для возврата элемента в таких функциях применяется языковая конструкция формата:

```
yield <Возвращаемый элемент>
```

Напишем функцию, которая генерирует диапазон из заданного количества чисел и последовательно возвращает его элементы, возведенные в заданную степень (листинг 11.11).

Листинг 11.11. Функция-генератор

```
def func(x, y):
    for i in range(1, x+1):
        yield i ** y

for n in func(10, 2):
    print(n, end=" ")     # Выведет: 1 4 9 16 25 36 49 64 81 100
print()                   # Вставляем пустую строку
for n in func(10, 3):
    print(n, end=" ")     # Выведет: 1 8 27 64 125 216 343 512 729 1000
```

Функции-генераторы поддерживают метод `__next__()`, выдающий следующее значение из последовательности. Когда значения заканчиваются, метод генерирует исключение `StopIteration`. Метод `__next__()` автоматически вызывается в цикле перебора последовательности. Для примера перепишем предыдущую программу, использовав метод `__next__()` вместо цикла (листинг 11.12).

Листинг 11.12. Использование метода `__next__()`

```
def func(x, y):
    for i in range(1, x+1):
       yield i ** y

i = func(3, 3)
print(i.__next__())       # Выведет: 1 (1 ** 3)
print(i.__next__())       # Выведет: 8 (2 ** 3)
print(i.__next__())       # Выведет: 27 (3 ** 3)
print(i.__next__())       # Исключение StopIteration
```

Существует возможность вызвать одну функцию-генератор из другой. Для этого применяется языковая конструкция формата:

```
yield from <Вызываемая функция-генератор>
```

Предположим, у нас есть список чисел, и нам требуется получить другой список, включающий числа в диапазоне от 1 до каждого из чисел в первом списке. Чтобы создать такой список, напишем код, показанный в листинге 11.13.

Листинг 11.13. Вызов одной функции-генератора из другой (простой случай)

```
def gen(l):
    for e in l:
        yield from range(1, e + 1)

l = [5, 10]
for i in gen([5, 10]): print(i, end = " ")
```

Здесь мы в функции-генераторе `gen()` перебираем переданный ей в качестве параметра список и для каждого его элемента вызываем другую функцию-генератор. В качестве последней выступает выражение, создающее диапазон от 1 до значения очередного элемента, увеличенного на единицу (чтобы это значение вошло в диапазон). В результате на выходе мы получим вполне ожидаемый результат:

```
1 2 3 4 5 1 2 3 4 5 6 7 8 9 10
```

Усложним задачу, включив в результирующий список числа, умноженные на 2. Код, выполняющий эту задачу, показан в листинге 11.14.

Листинг 11.14. Вызов одной функции-генератора из другой (сложный случай)

```
def gen2(n):
    for e in range(1, n + 1):
        yield e * 2

def gen(l):
    for e in l:
        yield from gen2(e)

l = [5, 10]
for i in gen([5, 10]): print(i, end = " ")
```

Здесь мы вызываем из функции-генератора `gen()` функцию-генератор `gen2()`. Последняя создает диапазон, перебирает все входящие в него числа и возвращает их умноженными на 2. Результат работы приведенного в листинге кода таков:

```
2 4 6 8 10 2 4 6 8 10 12 14 16 18 20
```

Функции-генераторы пригодятся при обработке большого количества значений, поскольку не понадобится загружать весь список с исходными значениями в память.

11.4. Декораторы функций

Декоратор — это функция, изменяющая поведение другой функции (например, производящая какие-либо действия перед ее выполнением). Функция-декоратор должна принимать в качестве параметра ссылку на функцию, поведение которой необходимо изменить, и возвращать ссылку на ту же функцию или какую-либо другую.

Вызов декоратора записывается непосредственно перед определением изменяемой функции в формате:

```
@<Имя декоратора>
```

Круглые скобки у декоратора указывать не нужно.

Пример использования декоратора показан в листинге 11.15.

Листинг 11.15. Декоратор функции

```
def deco(f):                          # Функция-декоратор
    print("Вызвана функция func()")
    return f                          # Возвращаем ссылку на функцию

@deco
def func(x):
    return "x = {0}".format(x)

print(func(10))
```

Результат выполнения этого примера:

```
Вызвана функция func()
x = 10
```

Приведенный пример эквивалентен коду, показанному в листинге 11.16.

Листинг 11.16. Эквивалент использования декоратора

```
def deco(f):
    print("Вызвана функция func()")
    return f

def func(x):
    return "x = {0}".format(x)

# Вызываем функцию func() через функцию deco()
print(deco(func)(10))
```

У определения изменяемой функции можно указать сразу несколько декораторов. Для примера укажем у функции `func()` два декоратора: `deco1()` и `deco2()` (листинг 11.17).

Листинг 11.17. Указание нескольких декораторов

```
def deco1(f):
    print("Вызвана функция deco1()")
    return f

def deco2(f):
    print("Вызвана функция deco2()")
    return f

@deco1
@deco2
def func(x):
    return "x = {0}".format(x)

print(func(10))
```

Вот что мы увидим после выполнения примера:

```
Вызвана функция deco2()
Вызвана функция deco1()
x = 10
```

Использование двух декораторов эквивалентно следующему коду:

```
func = deco1(deco2(func))
```

Здесь сначала будет вызвана функция `deco2()`, а затем функция `deco1()`. Результат выполнения будет присвоен идентификатору `func`.

В качестве еще одного примера использования декораторов рассмотрим выполнение функции только при правильно введенном пароле (листинг 11.18).

Листинг 11.18. Ограничение доступа с помощью декоратора

```
passw = input("Введите пароль: ")

def test_passw(p):
    def deco(f):
        if p == "10":          # Сравниваем пароли
            return f
        else:
            return lambda: "Доступ закрыт"
    return deco                # Возвращаем функцию-декоратор

@test_passw(passw)
def func():
    return "Доступ открыт"

print(func())                  # Вызываем функцию
```

Здесь после символа `@` указана не ссылка на функцию, а выражение, возвращающее декоратор. Иными словами, декоратором здесь служит не функция `test_passw()`, а результат ее выполнения (функция `deco()`). Если введенный пароль является правильным, то выполнится функция `func()`, в противном случае будет выведена надпись `"Доступ закрыт"`, которую вернет анонимная функция.

11.5. Рекурсия

Рекурсия — это вызов функцией самой себя. Рекурсию удобно использовать для перебора объекта с неизвестной структурой или для выполнения неопределенного количества операций. Для примера рассмотрим вычисление факториала (листинг 11.19).

Листинг 11.19. Вычисление факториала

```
def factorial(n):
    if n == 0 or n == 1: return 1
    else:
        return n * factorial(n - 1)
```

```
while True:
    x = input("Введите число: ")
    if x.isdigit():             # Если строка содержит только цифры
        x = int(x)              # Преобразуем строку в число
        break                   # Выходим из цикла
    else:
        print("Вы ввели не число!")
print("Факториал числа {0} = {1}".format(x, factorial(x)))
```

Впрочем, для вычисления факториала лучше воспользоваться функцией `factorial()` из модуля `math` (см. *разд. 5.3*).

Количество вызовов функции самой себя (*проходов рекурсии*) ограничено. Узнать его можно, вызвав функцию `getrecursionlimit()` из модуля `sys`:

```
>>> import sys
>>> sys.getrecursionlimit()
    1000
```

При превышении допустимого количества проходов рекурсии будет возбуждено исключение `RecursionError`.

11.6. Вложенные функции

Вложенная функция — это функция, определенная в теле другой функции (*родительской*). Вложенная функция может создавать свои собственные локальные переменные и имеет доступ к локальным переменным, созданным в родительской функции. Рассмотрим пример из листинга 11.20.

Листинг 11.20. Вложенная функция

```
def func1(x):
    def func2():
        print(x)
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                # Выведет: 10
f2()                # Выведет: 99
```

Здесь мы определили функцию `func1()`, принимающую один параметр, а внутри нее — вложенную функцию `func2()`. Результатом выполнения функции `func1()` станет ссылка на эту вложенную функцию. Внутри функции `func2()` мы производим вывод значения переменной `x`, созданной в функции `func1()`.

Следует учитывать, что в момент определения функции сохраняются ссылки на переменные, а не их значения. Например, если после определения функции `func2()` произвести изменение переменной `x`, то будет использоваться это новое значение (листинг 11.21).

Листинг 11.21. При определении вложенной функции сохраняется ссылка на переменную

```
def func1(x):
    def func2():
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                # Выведет: 30
f2()                # Выведет: 30
```

Обратите внимание на результат выполнения — в обоих случаях мы получили значение 30. Если при определении вложенной функции необходимо сохранить именно значение переменной, следует указать его как значение по умолчанию (листинг 11.22).

Листинг 11.22. Принудительное сохранение значения переменной

```
def func1(x):
    def func2(x=x): # Сохраняем текущее значение, а не ссылку
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                # Выведет: 10
f2()                # Выведет: 99
```

Теперь попробуем из вложенной функции `func2()` изменить значение переменной `x`, созданной внутри функции `func1()`. Если внутри функции `func2()` присвоить значение переменной `x`, будет создана новая локальная переменная с таким же именем. Если внутри функции `func2()` объявить переменную как глобальную и присвоить ей значение, то изменится значение глобальной переменной, а не значение переменной `x` из функции `func1()`. Таким образом, ни один из изученных ранее способов не позволяет из вложенной функции изменить значение локальной переменной из родительской функции.

Решить эту проблему можно, объявив необходимые переменные с помощью языковой конструкции формата:

```
nonlocal <Имена переменных через запятую>
```

Пример показан в листинге 11.23.

Листинг 11.23. Языковая конструкция `nonlocal`

```
def func1(a):
    x = a
    def func2(b):
        nonlocal x    # Объявляем переменную как nonlocal
```

```
        print(x)
        x = b          # Можем изменить значение x в func1()
    return func2

f = func1(10)
f(5)                   # Выведет: 10
f(12)                  # Выведет: 5
f(3)                   # Выведет: 12
```

При использовании языковой конструкции `nonlocal` следует помнить, что переменная обязательно должна существовать внутри функции-родителя. В противном случае будет выведено сообщение об ошибке.

11.7. Аннотации функций

Аннотация — это описание назначения параметров функции, их типов и типа возвращаемого результата, которое записывается в определении функции. Аннотации имеют следующий формат:

```
def <Имя функции>(
    [<Параметр 1>[: <Описание параметра>] [= <Значение по умолчанию>]
    [, ...,
     <Параметр N>[: <Описание параметра>] [= <Значение по умолчанию>]]]
    ) -> <Описание возвращаемого результата>:
    <Тело функции>
```

В качестве описаний параметров и возвращаемого результата обычно указываются строки (хотя можно задать любое выражение языка Python, которое будет выполнено при создании функции). Пример:

```
>>> def func(a: "Выводимое значение 1", b: "Выводимое значение 2" = 3) -> None:
...     print(a, b)
```

У параметра `a` задано описание `"Выводимое значение 1"`, у параметра `b` — `"Выводимое значение 2"` (число `3` — это значение параметра `b` по умолчанию). После закрывающей круглой скобки указан тип возвращаемого функцией результата: `None` (т. е. результата функция не возвращает).

Аннотация функции сохраняется в виде словаря в атрибуте `__annotations__` ее объекта. Выведем значение этого атрибута:

```
>>> func.__annotations__
    {'a': 'Выводимое значение 1', 'b': 'Выводимое значение 2', 'return': None}
```

ГЛАВА 12

Модули, пакеты и импорт

Модуль — это любой файл с программным кодом. Любой модуль может использовать идентификаторы (переменные, функции и классы, о которых речь пойдет в *главе 13*), созданные в другом модуле, выполнив процедуру *подключения*, или *импорта*, последнего.

В составе интерпретатора Python поставляется большой набор модулей, содержащих полезные переменные, функции и классы. Эти модули называются *встроенными*, а их совокупность — *стандартной библиотекой* языка.

> ***Примечание***
>
> Модули можно создавать не только на самом Python, но и на языке C++, компилируя их в машинный код. В стандартной библиотеке содержится ряд таких модулей.

Модуль Python представляется особым объектом, содержащим ряд атрибутов. Так, атрибут `__name__` содержит имя модуля в виде строки. У модуля, непосредственно запущенного на исполнение, этот атрибут хранит строку `"__main__"`. Пример:

```
print(__name__)                    # Выведет: __main__
```

Проверить, является модуль непосредственно запущенным (*главным модулем*, или *главной программой*) или импортированным, позволяет следующий код:

```
if __name__ == "__main__":
    print("Это главная программа")
else:
    print("Импортированный модуль")
```

12.1. Импорт модуля целиком

Можно импортировать модуль целиком и использовать любые созданные в нем переменные, функции и классы. Для этого предназначена языковая конструкция следующего формата:

```
import <Модуль 1> [as <Псевдоним 1>],
       . . .,
       <Модуль N> [as <Псевдоним N>]
```

Имя модуля не должно содержать расширения и пути к файлу, а также удовлетворять всем требованиям, предъявляемым к именам переменных (см. *разд. 2.1*). Дело в том, что при им-

порте модуля интерпретатор создает переменную, которой присваивает объект импортированного модуля.

Все идентификаторы, созданные в импортированном модуле, доступны через атрибуты объекта этого модуля (атрибут — это переменная, входящая в состав объекта). Для доступа к атрибутам следует использовать такой синтаксис:

```
<Переменная с объектом модуля>.<Атрибут с нужным идентификатором>
```

Поскольку переменная с объектом модуля и атрибут отделяются друг от друга точкой, такая запись получила название *точечной нотации*.

Для примера импортируем модуль `time` и получим текущую дату вызовом функции `strftime()`, определенной в этом модуле:

```
import time
print(time.strftime("%d.%m.%Y"))
```

Подключим сразу два модуля: `time` и `math`:

```
import time, math
print(time.strftime("%d.%m.%Y"))
print(math.pi)                    # Число π
```

Функция `getattr()` позволяет получить значение атрибута с заданным в виде строки именем, принадлежащего указанному модулю. С помощью этой функции можно сформировать имя нужного атрибута программно. Формат функции:

```
getattr(<Объект модуля>, <Имя атрибута>[, <Значение по умолчанию>])
```

Если указанный атрибут не найден, возвращается заданное значение по умолчанию, а если оно не указано — возбуждается исключение `AttributeError`. Пример:

```
import math
print(getattr(math, "pi"))        # Будет выведено число π
print(getattr(math, "x", 50))     # Будет выведено 50, т. к. x не существует
```

Проверить существование атрибута с заданным именем в указанном модуле позволяет функция `hasattr(<Модуль>, <Имя атрибута>)`. Если атрибут существует, функция возвращает значение `True`, в противном случае — `False`. Напишем функцию проверки существования атрибута в модуле `math` (листинг 12.1).

Листинг 12.1. Проверка существования атрибута

```
import math
def hasattr_math(attr):
    if hasattr(math, attr):
        return "Атрибут существует"
    else:
        return "Атрибут не существует"
print(hasattr_math("pi"))        # Атрибут существует
print(hasattr_math("x"))         # Атрибут не существует
```

Если имя модуля слишком длинное и его неудобно указывать каждый раз для доступа к атрибутам, то можно дать модулю *псевдоним*. После чего для доступа к модулю следует

использовать исключительно указанный псевдоним (при попытке доступа по имени модуля возникнет ошибка). Дадим модулю `math` псевдоним `m`:

```
import math as m
print(m.pi)
```

Идентификаторы, содержащиеся в импортированном модуле, не смешиваются с идентификаторами, созданными в импортирующем модуле. Это значит, что, например, в импортируемом и импортирующем модулях могут содержаться совершенно разные переменные с одинаковым именем `x`.

Проиллюстрируем это примером. Создадим модуль test.py, в котором определим переменную `x` (листинг 12.2).

Листинг 12.2. Код модуля test.py

```
x = 50
```

В главной программе также определим переменную `x`, но с другим значением. Затем подключим модуль test.py и выведем значения переменных (листинг 12.3).

Листинг 12.3. Код главной программы

```
import test             # Подключаем файл test.py
x = 22
print(test.x)           # Значение переменной x внутри модуля
print(x)                # Значение переменной x в главной программе
input()
```

Оба модуля размещаем в одном каталоге и запускаем модуль с главной программой. Программа выведет числа 50 и 22 — значения переменных с именем `x`, хранящихся в разных модулях. Как видно из результата, одноименные переменные из разных модулей никак не конфликтуют друг с другом.

Объект каждого импортированного модуля заносится в словарь `modules` из модуля `sys`. При попытке импорта модуля сначала проверяется, есть ли этот модуль в упомянутом словаре, и если он там есть, повторный импорт выполнен не будет.

Выведем ключи словаря `modules`, предварительно отсортировав их (листинг 12.4).

Листинг 12.4. Вывод ключей словаря `modules`

```
import test, sys                # Подключаем модули test и sys
print(sorted(sys.modules.keys()))
input()
```

Инструкция импорта требует явного указания объекта модуля. Задать имя модуля в виде строки нельзя.

Чтобы подключить модуль, имя которого формируется программно, следует воспользоваться функцией `__import__(<Имя модуля>)`. Функция возвращает объект импортированного модуля. Для примера импортируем модуль test.py с помощью функции `__import__()` (листинг 12.5).

Листинг 12.5. Использование функции __import__()

```
s = "tes" + "t"              # Динамическое создание имени модуля
m = __import__(s)            # Подключение модуля test
print(m.x)                   # Вывод значения атрибута x
input()
```

Получить список всех идентификаторов, созданных в указанном модуле, позволяет функция `dir(<Объект модуля>)`. Еще можно воспользоваться словарем из .атрибута `__dict__` объекта модуля, который содержит все идентификаторы и их значения (листинг 12.6).

Листинг 12.6. Вывод списка всех идентификаторов

```
import test
print(dir(test))
print(sorted(test.__dict__.keys()))
input()
```

Примечание

При импорте модуля он всегда компилируется (о компиляции рассказывалось в *разд. 1.10*).

12.2. Импорт отдельных идентификаторов

Также можно импортировать из модуля только отдельные идентификаторы, нужные для работы. Для этого применяется языковая конструкция, записываемая в одном из следующих форматов:

```
from <Модуль> import <Идентификатор 1> [as <Псевдоним 1>],
                     . . .,
                     <Идентификатор N> [as <Псевдоним N>]
from <Модуль> import (<Идентификатор 1> [as <Псевдоним 1>],
                      . . .,
                      <Идентификатор N> [as <Псевдоним N>])
from <Модуль> import *
```

Первые два формата импортируют из указанного модуля только заданные идентификаторы. Длинным идентификаторам можно назначить псевдонимы. В качестве примера импортируем из модуля `math` константу `pi` и функцию `floor()`, дав последней псевдоним `f` (листинг 12.7).

Листинг 12.7. Импорт отдельных идентификаторов

```
from math import pi, floor as f
print(pi)                                  # Вывод числа π
# Вызываем функцию floor() через идентификатор f
print(f(5.49))                             # Выведет: 5
input()
```

Второй формат инструкции импорта позволяет разбить слишком длинный перечень импортируемых идентификаторов на несколько строк:

```
from math import (pi, floor,
                  sin, cos)
```

Третий формат импортирует из модуля все идентификаторы. Для примера импортируем все идентификаторы из модуля `math` (листинг 12.8).

Листинг 12.8. Импорт всех идентификаторов из модуля

```
from math import *     # Импортируем все идентификаторы из модуля math
print(pi)              # Выводим число π
print(floor(5.49))     # Вызываем функцию floor()
input()
```

Однако в этом случае идентификаторы, импортированные из модуля, смешаются с идентификаторами из импортирующего модуля. Например, если импортировать из модуля переменную `s`, а потом попытаться создать в импортирующем модуле переменную с таким же именем, произойдет перезапись значения импортированной переменной `s`. То же самое произойдет, если импортировать одноименную переменную из двух разных модулей, — при импорте переменной из второго модуля будет перезаписано значение переменной из первого модуля.

Для примера создадим два модуля, содержащих переменные с одинаковым именем `s` и разными значениями, и подключим их с помощью разных инструкций. Содержимое первого модуля, с именем module1.py, приведено в листинге 12.9.

Листинг 12.9. Содержимое файла module1.py

```
s = "Значение из модуля module1"
```

Содержимое второго модуля, module2.py, приведено в листинге 12.10.

Листинг 12.10. Содержимое файла module2.py

```
s = "Значение из модуля module2"
```

Код главной программы показан в листинге 12.11.

Листинг 12.11. Код главной программы

```
from module1 import *
from module2 import *
import module1, module2
print(s)                  # Выведет: "Значение из модуля module2"
print(module1.s)          # Выведет: "Значение из модуля module1"
print(module2.s)          # Выведет: "Значение из модуля module2"
input()
```

Размещаем все три модуля в одном каталоге, запускаем модуль с главной программой и смотрим на результат...

...согласно которому, переменная `s` получила значение из модуля module2.py, который был импортирован последним. Это довольно опасная ситуация, чреватая возникновением труд-

но выявляемых ошибок. Однако получить доступ к переменным s из разных модулей все-таки можно — импортировав эти модули целиком (см. *разд. 12.1*).

12.2.1. Указание идентификаторов, доступных для импорта

По умолчанию все идентификаторы, определенные в модуле, доступны для импорта в других модулях. Исключение составляют лишь идентификаторы, начинающиеся со знака подчеркивания, — их импортировать нельзя.

Однако можно явно задать перечень идентификаторов, доступных для импорта (идентификаторы, не указанные в перечне, не будут импортироваться). Для этого следует в коде модуля занести в переменную __all__ список таких идентификаторов, представленных в виде строк. Интересно, что таким образом можно сделать доступными для импорта даже идентификаторы, начинающиеся с подчеркивания.

Для примера создадим модуль module3.py с множеством переменных и разрешим другим модулям импортировать только две из них (листинг 12.12).

Листинг 12.12. Использование переменной __all__

```
x, y, z, _s = 10, 80, 22, "Строка"
__all__ = ["x", "_s"]
```

Затем напишем главную программу, которая будет его импортировать (листинг 12.13).

Листинг 12.13. Код главной программы

```
from module3 import *
print(sorted(vars().keys())) # Получаем список всех идентификаторов
input()
```

Главная программа выдаст следующий результат:

```
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', '_s', 'x']
```

Были импортированы только переменные _s и x. Если бы мы не указали идентификаторы внутри списка __all__, результат был бы другим:

```
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'x', 'y', 'z']
```

Обратите внимание на то, что переменная _s в этом случае не импортируется, поскольку ее имя начинается с символа подчеркивания.

12.2.2. Управление доступом к идентификаторам

В Python 3.7 появилась возможность управлять доступом к идентификаторам, определенным в импортируемом модуле. В частности, можно при попытке получить из импортирующего модуля доступ к какому-либо несуществующему идентификатору выдать значение другого, существующего, идентификатора или даже результат вычисления какого-либо выражения.

Чтобы реализовать управление доступом к идентификаторам, в модуле следует определить *специальную функцию* (функцию, вызываемую самим интерпретатором в определенные моменты времени) с именем `__getattr__()`. Она должна принимать в качестве единственного параметра имя идентификатора, к которому импортирующий модуль пытается получить доступ, и либо возвращать какое-либо значение, либо, если такой идентификатор не должен поддерживаться, генерировать исключение `AttributeError`.

Также в модуле можно определить специальную функцию `__dir__()`. Она не должна принимать параметров и должна возвращать последовательность с именами идентификаторов, доступных извне текущего модуля.

В качестве примера создадим модуль **module4.py** с кодом из листинга 12.14.

Листинг 12.14. Использование специальных функций `__getattr__()` и `__dir__()`

```
def __getattr__(name):
    # При попытке обратиться к переменной с выдаем значение переменной d.
    # При обращении к другим переменным генерируем исключение.
    if name == "c":
        return d
    else:
        raise AttributeError

def __dir__():
    return ("a", "b", "d")

a = 10
b = 20
d = 10000
```

Код главной программы приведен в листинге 12.15.

Листинг 12.15. Код главной программы

```
import module4
print(module4.__dir__())      # Выведет: ('a', 'b', 'd')
print(module4.a)              # Выведет: 10
print(module4.b)              # Выведет: 20
print(module4.c)              # Выведет: 10000
```

12.3. Пути поиска модулей

Получить список файловых путей, по которым интерпретатор Python будет искать импортируемые модули, можно из переменной `path`, которая создана в модуле `sys`:

```
>>> import sys
>>> sys.path
```

Список из переменной `path` содержит следующие пути:

- путь к папке с файлом главной программы или, если интерпретатор работает в интерактивном режиме, пустая строка;

- значение переменной окружения PYTHONPATH. Для создания этой переменной в меню **Пуск** выбираем пункт **Панель управления** (или **Настройка | Панель управления**). В открывшемся окне выбираем пункт **Система** и щелкаем на ссылке **Дополнительные параметры системы**. Переходим на вкладку **Дополнительно** и нажимаем кнопку **Переменные среды**. В разделе **Переменные среды пользователя** нажимаем кнопку **Создать**. В поле **Имя переменной** вводим PYTHONPATH, а в поле **Значение переменной** задаем пути к папкам с модулями через точку с запятой — например, C:\folder1; C:\folder2. Закончив, не забудем нажать кнопки **OK** обоих открытых окон. После этого изменения перезагружать компьютер не нужно, достаточно перезапустить программу;
- пути поиска стандартных модулей;
- содержимое файлов с расширением pth, расположенных в каталогах поиска стандартных модулей, — например, в каталоге C:\Python310\Lib\site-packages. Названия таких файлов могут быть произвольными, главное, чтобы они имели расширение pth. Каждый путь (абсолютный или относительный) должен быть расположен на отдельной строке.

  Для примера создайте файл mypath.pth в каталоге C:\Python310\Lib\site-packages со следующим содержимым:

```
# Это комментарий
C:\folder1
C:\folder2
```

> ***Примечание***
>
> Обратите внимание на то, что каталоги должны существовать, в противном случае они не будут добавлены в список sys.path.

При поиске модуля список sys.path просматривается от начала к концу. Поиск прекращается после первого найденного модуля. Таким образом, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, то будет использоваться модуль из папки C:\folder1, поскольку он расположен первым в списке путей поиска.

Список sys.path можно изменять программно с помощью соответствующих методов. Например, добавить каталог в конец списка можно с помощью метода append(), а в его начало — с помощью метода insert() (листинг 12.16).

Листинг 12.16. Изменение списка путей поиска модулей

```
import sys
sys.path.append(r"C:\folder1")          # Добавляем в конец списка
sys.path.insert(0, r"C:\folder2")       # Добавляем в начало списка
print(sys.path)
input()
```

В этом примере мы добавили папку C:\folder2 в начало списка. Теперь, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, будет использоваться модуль из папки C:\folder2, а не из папки C:\folder1, как в предыдущем примере.

Также можно указать полностью свои пути для поиска модулей, при этом список, хранящийся в переменной sys.path, будет проигнорирован. Для этого достаточно поместить в папку, где установлен Python, файл с именем python<первые два числа из номера версии Python>._pth (так, для Python 3.10 этот файл должен иметь имя python310._pth) или python._pth, в котором записать все нужные пути в том же формате, который используется

при создании файлов pth. Первый файл будет использоваться программами, вызывающими библиотеку времени выполнения Python, в частности **IDLE Shell**. А второй файл будет считан при запуске Python-программы щелчком мыши на ее файле.

> ***Внимание!***
>
> В файл *python<первые два числа из номера версии Python>._pth* обязательно следует включить пути для поиска модулей, составляющих стандартную библиотеку Python (их можно получить из списка, хранящегося в переменной `sys.path`). Если этого не сделать, утилита **IDLE Shell** вообще не запустится.

12.4. Перезагрузка модулей

При первой операции импорта модуль загружается в оперативную память. При последующих попытках импортировать тот же модуль ничего не произойдет. Если же требуется выполнить повторный импорт (*перезагрузку*) импортированного ранее модуля, следует воспользоваться функцией `reload(<Модуль>)` из модуля `importlib`.

В качестве примера создадим модуль test2.py, поместив его в папку C:\book (листинг 12.17).

Листинг 12.17. Содержимое модуля test2.py

```
x = 150
```

Подключим этот модуль и выведем текущее значение переменной `x`:

```
>>> import sys
>>> sys.path.append(r"C:\book") # Добавляем путь к папке с модулем
>>> import test2               # Подключаем модуль test2.py
>>> print(test2.x)             # Выводим текущее значение
    150
```

Не закрывая окна интерпретатора, изменим значение переменной `x` на 800, а затем попробуем заново импортировать модуль и вывести текущее значение переменной:

```
>>> # Изменяем значение в модуле на 800
>>> import test2
>>> print(test2.x)             # Значение не изменилось
   150
```

Как видно из примера, значение переменной `x` не изменилось. Теперь перезагрузим модуль с помощью функции `reload()`:

```
>>> from importlib import reload
>>> reload(test2)              # Перезагружаем модуль
<module 'test2' from 'C:\book\test2.py'>
>>> print(test2.x)             # Значение изменилось
    800
```

Следует учитывать, что идентификаторы, импортированные с помощью языковой конструкции `from` (см. *разд. 12.2*), перезагружены не будут. Кроме того, не перезагружаются скомпилированные модули, написанные на других языках программирования (например, на C).

12.5. Пакеты

Пакет — это обычный каталог, содержащий модули и обязательный файл инициализации __init__.py. Последний может быть пустым или содержать код, который будет выполнен при первой операции импорта любого модуля из этого пакета.

Чтобы импортировать модуль, находящийся в пакете, надо записать путь к этому модулю, разделяя имена пакетов и самого модуля точками. Например, чтобы импортировать модуль module2.py из пакета folder1\folder2, следует записать инструкцию:

```
import folder1.folder2.module2
```

Правда, переменная, создаваемая интерпретатором для хранения объекта импортированного модуля, получит имя, совпадающее с указанным путем к модулю:

```
folder1.folder2.module.func()
```

Поэтому для упрощения доступа к созданным в таком модуле идентификаторам рекомендуется задавать у модуля псевдоним:

```
import folder1.folder2.module2 as mod
mod.func()
```

или импортировать только нужные идентификаторы:

```
from folder1.folder2.module import func
func()
```

Также можно импортировать модуль из пакета, записав языковую конструкцию формата:

```
from <Путь к пакету> import <Модуль>[ as <Псевдоним>]
```

Например:

```
from folder1.folder2 import module
module.func()
```

В качестве примера создадим следующую структуру каталогов и файлов:

```
main.py                 # Файл с главной программой
folder1\                # Каталог на одном уровне вложенности с main.py
    __init__.py         # Файл инициализации
    module1.py          # Модуль folder1\module1.py
    folder2\            # Вложенный каталог
        __init__.py     # Файл инициализации
        module2.py      # Модуль folder1\folder2\module2.py
        module3.py      # Модуль folder1\folder2\module3.py
```

Содержимое файлов __init__.py приведено в листинге 12.18.

Листинг 12.18. Содержимое модулей __init__.py

```
print("__init__ из", __name__)
```

Содержимое модулей module1.py, module2.py и module3.py приведено в листинге 12.19.

Листинг 12.19. Содержимое модулей module1.py, module2.py и module3.py

```
msg = "Модуль {0}".format(__name__)
```

Теперь импортируем эти модули в главном модуле main.py и получим значение переменной msg разными способами. Содержимое модуля main.py приведено в листинге 12.20.

Листинг 12.20. Содержимое модуля main.py

```
# Доступ к модулю folder1\module1.py
import folder1.module1 as m1          # Выведет: __init__ из folder1
print(m1.msg)                         # Выведет: Модуль folder1.module1
from folder1 import module1 as m2
print(m2.msg)                         # Выведет: Модуль folder1.module1
from folder1.module1 import msg
print(msg)                            # Выведет: Модуль folder1.module1

# Доступ к модулю folder1\folder2\module2.py
import folder1.folder2.module2 as m3  # Выведет: __init__ из folder1.folder2
print(m3.msg)                         # Выведет: Модуль folder1.folder2.module2
from folder1.folder2 import module2 as m4
print(m4.msg)                         # Выведет: Модуль folder1.folder2.module2
from folder1.folder2.module2 import msg
print(msg)                            # Выведет: Модуль folder1.folder2.module2

input()
```

Инструкция импорта из пакета также позволяет импортировать сразу несколько модулей. Для этого внутри файла инициализации __init__.py в переменной `__all__` следует указать список с именами импортируемых модулей. В качестве примера изменим содержимое файла __init__.py из каталога folder1\folder2\ на следующее:

```
__all__ = ["module2", "module3"]
```

Теперь создадим модуль main2.py (листинг 12.21) и запустим его.

Листинг 12.21. Содержимое модуля main2.py

```
from folder1.folder2 import *
print(module2.msg)          # Выведет: Модуль folder1.folder2.module2
print(module3.msg)          # Выведет: Модуль folder1.folder2.module3
input()
```

Здесь после ключевого слова `from` указывается лишь путь к пакету без имени модуля. В результате будут импортированы все модули, указанные в списке `__all__`.

Чтобы импортировать модуль, расположенный в том же каталоге, что и импортирующий модуль, перед именем модуля указывается точка, например:

```
from .module import *
```

или

```
from . import module
```

Чтобы импортировать модуль, расположенный в родительском каталоге, перед именем модуля указываются две точки:

```
from ..module import *
```

или

```
from .. import module
```

Если необходимо обратиться уровнем еще выше, то указываются три точки:

```
from ...module import *
```

или

```
from ... import module
```

Для примера создадим в каталоге C:\folder1\folder2\ модуль module4.py, чей код показан в листинге 12.22.

Листинг 12.22. Содержимое модуля module4.py

```
# Импорт модуля module2.py из текущего каталога
from . import module2 as m1
var1 = "Значение из: {0}".format(m1.msg)
from .module2 import msg as m2
var2 = "Значение из: {0}".format(m2)

# Импорт модуля module1.py из родительского каталога
from .. import module1 as m3
var3 = "Значение из: {0}".format(m3.msg)
from ..module1 import msg as m4
var4 = "Значение из: {0}".format(m4)
```

Теперь создадим модуль main3.py (листинг 12.23) и запустим его на выполнение.

Листинг 12.23. Содержимое модуля main3.py

```
from folder1.folder2 import module4 as m
print(m.var1)          # Значение из: Модуль folder1.folder2.module2
print(m.var2)          # Значение из: Модуль folder1.folder2.module2
print(m.var3)          # Значение из: Модуль folder1.module1
print(m.var4)          # Значение из: Модуль folder1.module1
input()
```

Следует помнить очень важную вещь. Если при запуске Python-программы в список sys.path автоматически добавляется путь к каталогу с запущенным файлом, то при относительном импорте внутри пакета этого не происходит. Для примера изменим содержимое модуля module4.py на следующее:

```
import module2        # Ошибка! Поиск модуля по абсолютному пути
var1 = "Значение из: {0}".format(module2.msg)
var2 = var3 = var4 = 0
```

Здесь мы пытаемся импортировать из модуля module4.py модуль module2.py, находящийся в том же пакете. Однако раз путь к импортирующему модулю (и, соответственно, к содержащему его пакету) не был добавлен в его список sys.path, интерпретатор не найдет тре-

буемый модуль и выдаст ошибку. А если по одному из путей поиска модулей находится модуль с таким же именем, то он и будет импортирован и программа станет работать не так, как нужно (и, скорее всего, «вылетит» с ошибкой).

Чтобы импортировать модуль из того же пакета, следует использовать такую инструкцию:

```
from . import module2
```

или указать полный путь относительно каталога, в котором находится главный модуль:

```
import folder1.folder2.module2 as module2
```

ГЛАВА 13

Объекты и классы

Объект — это сложная структура, хранящая множество каких-либо значений и содержащая инструменты для их обработки. Отдельные значения хранятся в *атрибутах* — переменных, входящих в состав объекта, а инструменты реализованы в виде *методов* — функций, которые также принадлежат объекту.

Все значения, обрабатываемые Python, представляют собой объекты. Например, строка является объектом, содержащим методы `format()`, `strip()`, `split()` и многие другие (подробности — в *главе 6*). Значение даты также является объектом с атрибутами `year`, `month`, `day`, методами `weelday()`, `isoformat()` и др. (полное описание приведено в *разд. 10.4.2*).

Класс — это образец, на основе которого создаются объекты, относящиеся к одной разновидности. Класс определяет набор атрибутов и методов, поддерживаемых объектами соответствующей разновидности. Например, объекты строк принадлежат классу, носящему имя `str` (как и строковый тип данных), а объекты дат — классу `date`.

Стандартная библиотека Python содержит множество классов, называемых *встроенными*. Кроме того, разработчики могут создавать свои, *пользовательские* классы.

13.1. Определение классов, создание объектов и работа с ними

Инструкция создания, или *определения*, пользовательского класса записывается в следующем формате:

```
class <Имя класса>:
    [<Строка документирования>]
    <Определения методов>
```

Имя класса должно удовлетворять тем же требованиям, что и имена переменных (см. *разд. 2.1*). Дело в том, что, обработав определение класса, интерпретатор создает переменную, чье имя совпадает с именем созданного класса, и присваивает ей определение этого класса (которое также представляет собой объект).

Определения методов записываются в том же формате, что и определения функций (см. *разд. 11.1*). Первым (или единственным) параметром каждый метод должен принимать ссылку на текущий объект этого класса (чтобы в теле метода можно было обратиться к атрибутам и методам текущего объекта). Традиционно этому параметру дают имя `self`.

Атрибуты создаются в теле методов простым присваиванием им значений, для чего применяются инструкции формата:

```
<Объект>.<Атрибут> = <Значение атрибута>
```

В качестве `<Объекта>` следует использовать значение первого параметра, полученного методом (как отмечалось ранее, обычно его называют `self`).

Кроме того, внутри методов можно обращаться к созданным ранее атрибутам текущего объекта с целью получить их значения, применяя следующий синтаксис (известный под названием *точечной нотации*):

```
<Объект>.<Атрибут>
```

и вызывать методы, использовав выражение формата:

```
<Объект>.<Метод>([Параметры через запятую])
```

При вызове метода следует указать только значения второго и последующих параметров, передаваемых ему. Значение первого параметра (`self`) — ссылка на текущий объект — будет передано методу самим интерпретатором.

Определения классов должны предшествовать инструкциям, создающим на их основе объекты. Как правило, определения класса располагают в начале модуля или в отдельном модуле, отделяют от остального кода и друг от друга пустыми строками — для наглядности.

При вводе определения класса в интерпретаторе, работающем в интерактивном режиме, после ввода языковой конструкции `class` и нажатия клавиши <Enter> последующие строки станут выводиться с отступом слева, и можно будет занести строку документирования, определения атрибутов и методов. При вводе определения метода после набора языковой конструкции `def` следующие строки станут выводиться с двойным отступом слева, и можно будет занести тело метода. Завершив ввод тела очередного метода, следует уменьшить отступ нажатием клавиши <Backspace>, чтобы ввести языковую конструкцию `def` определения следующего метода. Завершив ввод всех методов, нужно нажать <Enter> дважды, чтобы закончить ввод определения класса.

Пример определения класса `Platform`, содержащего атрибуты `name` (название программной платформы) и `type` (тип платформы: клиентская или серверная, изначальное значение: `"Серверная"`), методы `setName(<Название>)` (задает новое название у текущей платформы) и `getFullName()` (возвращает строку с названием и типом текущей платформы):

```
>>> class Platform:
...     def setName(self, name):
...         self.name = name              # Создаем атрибут name
...         self.type = "Серверная"       # Создаем атрибут type
...     def getFullName(self):
...         return self.name + " (" + self.type + ")"
...
```

Определив класс, на его основе можно создать произвольное количество объектов. Для создания объекта класса примеряется следующий синтаксис:

```
<Класс>()
```

Возвращенный объект класса необходимо присвоить какой-либо переменной, атрибуту другого объекта, передать в качестве параметра функции или методу. Если этого не сделать, созданный объект будет потерян.

Для примера создадим пару объектов только что определенного класса `Platform`:

```
>>> pl1 = Platform()
>>> pl2 = Platform()
```

Для обращения к атрибутам и методам объекта применяется точечная нотация, описанная ранее. Укажем у первого из созданных объектов название платформы (вызовом метода `setName()`), а у второго — название и тип (который занесем в атрибут `type`):

```
>>> pl1.setName("Python")       # Указываем значение второго параметра метода
>>> pl2.setName("JavaScript")   # Значение первого параметра (ссылка на текущий
>>>                             # объект) будет передано методу интерпретатором
>>> pl2.type = "Клиентская"
>>> pl2.getFullName()
    'JavaScript (Клиентская)'
>>> pl1.getFullName()
    'Python (Серверная)'
```

В обычном коде можно создавать у объекта принадлежащие исключительно ему атрибуты (*атрибуты экземпляра объекта*), просто присваивая им значения. Создадим у второго объекта класса `Platform` атрибут экземпляра объекта `founded`, хранящий год создания платформы:

```
>>> pl2.founded = 1995
>>> pl2.founded
    1995
>>> pl1.founded          # У первого объекта атрибута founded нет
    ... Фрагмент пропущен ...
    AttributeError: 'Platform' object has no attribute 'founded'
```

Удалить ненужный атрибут можно, использовав языковую конструкцию `del` (см. *разд. 2.6*) и записав в ней удаляемый атрибут в формате: `<Объект>.<Атрибут>`:

```
>>> del pl1.founded
```

Для работы с атрибутами можно использовать следующие функции:

- `getattr()` — возвращает значение атрибута с заданным в виде строки именем, принадлежащего указанному объекту. Формат функции:

  ```
  getattr(<Объект>, <Имя атрибута>[, <Значение по умолчанию>])
  ```

  Если атрибут с указанным именем не найден, генерируется исключение `AttributeError`. Чтобы избежать этого, в третьем параметре можно указать значение по умолчанию, возвращаемое, если атрибут не существует;

- `setattr()` — заносит указанное значение в атрибут с указанным именем, принадлежащий заданному объекту. Формат функции:

  ```
  setattr(<Объект>, <Имя атрибута>, <Значение>)
  ```

  Вторым параметром можно передать имя несуществующего атрибута — в этом случае будет создан атрибут экземпляра объекта с таким именем;

- `delattr(<Объект>, <Имя атрибута>)` — удаляет из заданного объекта атрибут, чье имя указано в виде строки;

- `hasattr(<Объект>, <Имя атрибута>)` — возвращает `True`, если атрибут с именем, указанным в виде строки, существует в заданном объекте, и `False` — в противном случае.

Примеры использования этих функций:

```
>>> getattr(pl1, "type")
    'Серверная'
>>> getattr(pl1, "version")
    ... Фрагмент пропущен ...
    AttributeError: 'Platform' object has no attribute 'version'
>>> getattr(pl1, "version", "0.0")
    '0.0'
>>> setattr(pl1, "version", "1.0")
>>> pl1.version
    '1.0'
>>> hasattr(pl2, "founded")
    True
>>> delattr(pl2, "founded")
>>> hasattr(pl2, "founded")
    False
```

Любые атрибуты и методы, определенные в классе, доступны как изнутри этого класса, так и извне его. Такие атрибуты и методы называются *общедоступными*. Исключение составляют лишь атрибуты и методы, чьи имена начинаются с двойного подчеркивания (__) — они доступны только внутри класса (*закрытые атрибуты* и *методы*). Закрытые атрибуты и методы будут описаны далее.

Определение класса представляет собой объект типа `type` (см. *разд. 2.4*):

```
>>> type(Platform)
    <class 'type'>
>>> type(Platform) == type
    True
```

13.2. Атрибуты класса

У класса можно создать атрибуты, принадлежащие самому классу, а не создаваемым на его основе объектам (*атрибуты класса*). Для этого применяется расширенный синтаксис определения класса:

```
class <Имя класса>:
    [<Строка документирования>]
    [<Определения атрибутов класса>]
    <Определения методов>
```

Определение отдельного атрибута класса записывается в формате:

```
<Имя атрибута> = <Значение атрибута>
```

Пример класса, содержащего два атрибута класса:

```
>>> class Python:
...     type = "Язык программирования"
...     version = "3.10.1"
...
```

Получить доступ к атрибутам класса можно как через сам класс (наиболее часто используемый вариант):

```
>>> Python.type
    'Язык программирования'
>>> Python.version = "3.10.2"
>>> Python.version
    '3.10.2'
```

так и через объекты этого класса:

```
>>> p = Python()
>>> p.type
    'Язык программирования'
```

Однако если попытаться изменить значение атрибута класса через объект, в этом объекте будет создан одноименный атрибут экземпляра объекта:

```
>>> p.type = "Среда разработки"    # Якобы меняем значение атрибута класса type
>>>          # В результате будет создан одноименный атрибут экземпляра объекта
>>> p.type
    'Среда разработки'
>>> p2 = Python()                  # Создаем еще один объект класса Python
>>> p2.type                        # Проверяем значение атрибута класса type
    'Язык программирования'
```

Атрибут класса будет хранить одно и то же значение у всех объектов этого класса. Так, если в атрибуте класса сохранить список, все объекты этого класса станут ссылаться на один и тот же объект списка, например:

```
>>> class Platforms:
...     platform_list = []          # Атрибут класса, хранящий список
...
>>> pls1 = Platforms()
>>> pls2 = Platforms()
>>> # Добавляем элемент в список через первый объект
>>> pls1.platform_list.append("Python")
>>> # Добавляем элемент в список через второй объект
>>> pls2.platform_list.append("JavaScript")
>>> # И убеждаемся, что оба объекта ссылаются на один и тот же список
>>> pls1.platform_list
    ['Python', 'JavaScript']
>>> pls2.platform_list
    ['Python', 'JavaScript']
```

13.3. Конструкторы и деструкторы

В определении класса можно указать два метода, имеющих специальное назначение:

- `__init__()` — *конструктор* — автоматически вызывается при создании объекта текущего класса.

  Первым параметром конструктору, как и другим методам, передается ссылка на текущий объект. Остальными параметрами можно передать любые значения, которые могут быть занесены в атрибуты создаваемого объекта или использованы в вычислениях.

Для создания объекта класса, имеющего конструктор, применяется инструкция следующего формата:

```
<Класс>([<Значения параметров через запятую>])
```

Указанные значения параметров будут переданы конструктору;

- `__del__()` — *деструктор* — автоматически вызывается перед уничтожением объекта текущего класса, когда перестает существовать последняя ссылка на этот объект. Должен принимать один параметр — ссылку на текущий объект.

Пример класса с конструктором и деструктором показан в листинге 13.1.

Листинг 13.1. Конструктор и деструктор

```
class MyClass:
    def __init__(self, par1, par2): # Конструктор класса
        self.attr1 = par1
        self.attr2 = par2
        print("Вызван конструктор")
    def __del__(self):              # Деструктор класса
        print("Вызван деструктор")

c1 = MyClass(1, 2)      # Выведет: Вызван конструктор
del c1                  # Выведет: Вызван деструктор
c2 = MyClass("a", "b")  # Выведет: Вызван конструктор
c3 = c2                 # Создаем еще одну ссылку на объект
del c2                  # Ничего не выведет, т. к. существует еще одна ссылка
del c3                  # Выведет: Вызван деструктор
```

13.4. Наследование

Наследование — это создание одного класса (*производного*, или *подкласса*) на основе другого (*базового*, или *суперкласса*). Производный класс получает все атрибуты и методы, присутствующие в базовом классе.

Для определения производного класса применяется инструкция следующего формата:

```
class <Имя класса>(<Базовый класс>):
    <Остальная часть определения класса>
```

Если и в базовом, и в производном классе присутствуют методы с одинаковыми именами, при вызове такого метода из объекта производного класса будет вызван метод производного класса. То есть метод производного класса произведет *перекрытие* одноименного метода из базового класса. Рассмотрим пример из листинга 13.2.

Листинг 13.2. Наследование

```
class Class1:           # Базовый класс
    def func1(self):
        print("Метод func1() класса Class1")
    def func2(self):
        print("Метод func2() класса Class1")
```

```
class Class2(Class1): # Производный класс Class2 наследует класс Class1
    def func2(self):
        print("Метод func2() класса Class2 перекрыл метод класса Class1")
    def func3(self):
        print("Метод func3() класса Class2")

c1 = Class1()         # Создаем объект класса Class1
c2 = Class2()         # Создаем объект класса Class2
c2.func1()            # Выведет: Метод func1() класса Class1
c1.func2()            # Выведет: Метод func2() класса Class1
c2.func2()  # Выведет: Метод func2() класса Class2 перекрыл метод класса Class1
c2.func3()            # Выведет: Метод func3() класса Class2
```

Однако на практике чаще приходится не полностью заменять какой-либо метод базового класса в производном классе, а дополнять его функциональность — производить *переопределение метода*. В теле переопределяемого метода производного класса записывается код, реализующий дополнительную функциональность, и в нужном месте этого кода ставится инструкция вызова метода базового класса. Для вызова метода базового класса из метода производного класса можно использовать одну из следующих инструкций:

- вызов функции `super()` — с применением выражения формата:

  ```
  super([<Производный класс>, <Ссылка на текущий объект>]).↵
  <Вызов метода базового класса>
  ```

  В вызове метода базового класса первый параметр, содержащий ссылку на текущий объект, не указывается.

  У самой функции `super()` параметры указываются редко, в крайне специфических случаях;

- `<Базовый класс>.<Вызов метода базового класса>`

  Здесь в вызове метода базового класса, напротив, следует передать первый параметр, содержащий ссылку на текущий объект.

Рассмотрим пример переопределения двух методов базового класса в производном классе двумя только что описанными способами (листинг 13.3).

Листинг 13.3. Переопределение методов

```
class Class1:
    def __init__(self):
        print("Конструктор базового класса")
    def func1(self):
        print("Метод func1() класса Class1")
    def func2(self):
        print("Метод func2() класса Class1")

class Class2(Class1):
    def __init__(self):
        print("Конструктор производного класса")
        Class1.__init__(self) # Вызываем конструктор базового класса
```

```
    def funcl(self):
        print("Метод funcl() класса Class2")
        super().funcl()       # Вызываем метод базового класса
    def func2(self):
        print("Метод func2() класса Class2")
        super(Class2, self).func2()       # Вызываем метод базового класса

c = Class2()                  # Создаем объект класса Class2
c.funcl()                     # Вызываем метод funcl()
c.func2()                     # Вызываем метод func2()
```

Выведет:

```
Конструктор производного класса
Конструктор базового класса
Метод funcl() класса Class2
Метод funcl() класса Classl
Метод func2() класса Class2
Метод func2() класса Classl
```

13.4.1. Множественное наследование

При *множественном наследовании* производный класс наследует сразу от нескольких базовых классов.

Для множественного наследования применяется инструкция следующего формата:

```
class <Имя класса>(<Базовые классы через запятую>):
    <Остальная часть определения класса>
```

При обращении в производном классе к атрибуту или методу, определенному в одном из базовых классов, просмотр базовых классов будет выполняться в том порядке, в котором они указаны.

Рассмотрим множественное наследование на примере (листинг 13.4).

Листинг 13.4. Множественное наследование

```
class Classl:          # Базовый класс для класса Class2
    def funcl(self):
        print("Метод funcl() класса Classl")

class Class2(Classl): # Класс Class2 наследует класс Classl
    def func2(self):
        print("Метод func2() класса Class2")

class Class3(Classl): # Класс Class3 наследует класс Classl
    def funcl(self):
        print("Метод funcl() класса Class3")
    def func2(self):
        print("Метод func2() класса Class3")
    def func3(self):
        print("Метод func3() класса Class3")
```

```
    def func4(self):
        print("Метод func4() класса Class3")

class Class4(Class2, Class3): # Множественное наследование
    def func4(self):
        print("Метод func4() класса Class4")

c = Class4()              # Создаем объект класса Class4
c.funcl()                 # Выведет: Метод funcl() класса Class3
c.func2()                 # Выведет: Метод func2() класса Class2
c.func3()                 # Выведет: Метод func3() класса Class3
c.func4()                 # Выведет: Метод func4() класса Class4
```

Метод `funcl()` определен в двух классах: `Classl` и `Class3`. Так как вначале просматриваются все базовые классы, непосредственно указанные в определении текущего класса, метод `funcl()` будет найден в классе `Class3` (поскольку он указан в числе базовых классов в определении `Class4`), а не в классе `Classl`.

Метод `func2()` также определен в двух классах: `Class2` и `Class3`. Так как класс `Class2` стоит первым в списке базовых классов, то метод будет найден именно в нем. Чтобы наследовать метод из класса `Class3`, следует указать это явным образом. Переделаем определение класса `Class4` из предыдущего примера и наследуем метод `func2()` из класса `Class3` (листинг 13.5).

Листинг 13.5. Указание класса при наследовании метода

```
class Class4(Class2, Class3): # Множественное наследование
    # Наследуем func2() из класса Class3, а не из класса Class2
    func2 = Class3.func2
    def func4(self):
        print("Метод func4() класса Class4")
```

Вернемся к листингу 13.4. Метод `func3()` определен только в классе `Class3`, поэтому метод наследуется от этого класса. Метод `func4()`, определенный в классе `Class3`, переопределяется в производном классе.

Если искомый метод найден в производном классе, то вся иерархия наследования просматриваться не будет.

Для получения кортежа с перечнем базовых классов можно воспользоваться атрибутом `__bases__` объекта определения класса. В качестве примера выведем базовые классы для всех классов из предыдущего примера:

```
>>> print(Classl.__bases__)
>>> print(Class2.__bases__)
>>> print(Class3.__bases__)
>>> print(Class4.__bases__)
```

Выведет:

```
(<class 'object'>,)
(<class '__main__.Classl'>,)
(<class '__main__.Classl'>,)
(<class '__main__.Class2'>, <class '__main__.Class3'>)
```

Рассмотрим порядок поиска идентификаторов при сложной иерархии множественного наследования (листинг 13.6).

Листинг 13.6. Поиск идентификаторов при множественном наследовании

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class5): pass
class Class7(Class4, Class6): pass
c = Class7()
print(c.x)
```

Последовательность поиска атрибута `x` будет такой:

```
Class7 -> Class4 -> Class3 -> Class6 -> Class5 -> Class2 -> Class1
```

Получить кортеж со всей цепочкой наследования позволяет атрибут `__mro__` объекта определения класса:

```
>>> print(Class7.__mro__)
```

Результат выполнения:

```
(<class '__main__.Class7'>, <class '__main__.Class4'>,
 <class '__main__.Class3'>, <class '__main__.Class6'>,
 <class '__main__.Class5'>, <class '__main__.Class2'>,
 <class '__main__.Class1'>, <class 'object'>)
```

13.4.1.1. Примеси и их использование

Примесь — это класс, предназначенный исключительно для расширения функциональности других классов путем добавления в них атрибутов и методов с помощью множественного наследования. Примеси определяются так же, как и обычные классы (поскольку ничем от них не отличаются).

В качестве примера создадим класс-примесь `Mixin`, после чего создадим еще два класса, добавим к их функциональности ту, что определена в примеси `Mixin`, и проверим ее в действии (листинг 13.7).

Листинг 13.7. Использование примеси

```
class Mixin:                          # Определяем сам класс-примесь
    attr = 0                          # Определяем атрибут примеси
    def mixin_method(self):           # Определяем метод примеси
        print("Метод примеси")

class Class1 (Mixin):
    def method1(self):
        print("Метод класса Class1")
```

```
class Class2 (Class1, Mixin):
    def method2(self):
        print("Метод класса Class2")

c1 = Class1()
c1.method1()
c1.mixin_method()                    # Class1 поддерживает метод примеси

c2 = Class2()
c2.method1()
c2.method2()
c2.mixin_method()                    # Class2 также поддерживает метод примеси
```

Результат:

```
Метод класса Class1
Метод примеси
Метод класса Class1
Метод класса Class2
Метод примеси
```

Примеси активно применяются в различных дополнительных библиотеках — в частности, в популярном веб-фреймворке Django.

13.5. Специальные методы

Специальные методы вызываются самим интерпретатором в определенные моменты времени. Два специальных метода — конструктор и деструктор — были рассмотрены в *разд. 13.3*. Помимо них, поддерживаются еще следующие:

- `__call__()` — вызывается при вызове текущего объекта с применением того же синтаксиса, что используется при вызове функции. Формат метода:

  ```
  __call__(self[, <Параметр 1>[, . . ., <Параметр N>]])
  ```

  Пример:

  ```
  class MyClass:
      def __init__(self, m):
          self.msg = m
      def __call__(self, new_value=None):
          if new_value:
              print(new_value)
          else:
              print(self.msg)
  c1 = MyClass("Значение1") # Создание объекта
  c2 = MyClass("Значение2") # Создание объекта
  c1()                      # Выведет: Значение1
  c2()                      # Выведет: Значение1
  c2("Новое значение")      # Выведет: Новое значение
  ```

- `__getattr__(self, <Имя атрибута>)` — вызывается при обращении к несуществующему атрибуту текущего объекта:

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattr__(self, attr):
        print("Вызван метод __getattr__()")
        return 0
c = MyClass()
# Атрибут i существует
print(c.i)      # Выведет: 20. Метод __getattr__() не вызывается
# Атрибут s не существует
print(c.s)      # Выведет: Вызван метод __getattr__() 0
```

- `__getattribute__(self, <Имя атрибута>)` — вызывается при обращении к любому, даже существующему атрибуту текущего объекта. Необходимо учитывать, что использование точечной нотации (для обращения к атрибутам) внутри этого метода приведет к зацикливанию. Чтобы избежать зацикливания, следует вызвать метод `__getattribute__()` объекта `object` и внутри этого метода вернуть значение атрибута или сгенерировать исключение `AttributeError`. Пример:

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattribute__(self, attr):
        print("Вызван метод __getattribute__()")
        return object.__getattribute__(self, attr) # Только так!!!
c = MyClass()
print(c.i)      # Выведет: Вызван метод __getattribute__() 20
```

- `__setattr__(self, <Имя атрибута>, <Значение>)` — вызывается при попытке присваивания значения атрибуту текущего объекта. Если внутри метода необходимо присвоить значение атрибуту, следует использовать словарь из атрибута `__dict__`, поскольку при применении точечной нотации метод `__setattr__()` будет вызван повторно, что приведет к зацикливанию. Пример:

```
class MyClass:
    def __setattr__(self, attr, value):
        print("Вызван метод __setattr__()")
        self.__dict__[attr] = value                # Только так!!!
c = MyClass()
c.i = 10        # Выведет: Вызван метод __setattr__()
print(c.i)      # Выведет: 10
```

- `__delattr__(self, <Имя атрибута>)` — вызывается при удалении атрибута текущего объекта с помощью инструкции `del <Объект>.<Атрибут>`;
- `__len__(self)` — вызывается при использовании функции `len()` применительно к текущему объекту, а также для проверки объекта на логическое значение при отсутствии в классе метода `__bool__()`. Метод должен возвращать положительное целое число. Пример:

```
class MyClass:
    def __len__(self):
        return 50
```

```
c = MyClass()
print(len(c))              # Выведет: 50
```

- `__bool__(self)` — вызывается при использовании функции `bool()` применительно к текущему объекту. Должен возвращать логическое значение;
- `__int__(self)` — вызывается при преобразовании текущего объекта в целое число с помощью функции `int()`. Должен возвращать целое число;
- `__float__(self)` — вызывается при преобразовании текущего объекта в вещественное число с помощью функции `float()`. Должен возвращать вещественное число;
- `__complex__(self)` — вызывается при преобразовании текущего объекта в комплексное число с помощью функции `complex()`. Должен возвращать комплексное число;
- `__round__(self, n)` — вызывается при округлении текущего объекта функцией `round()`. Должен возвращать вещественное число;
- `__index__(self)` — вызывается при преобразовании текущего объекта в целое число при извлечении среза, вызове функций `bin()`, `oct()` или `hex()`. Должен возвращать целое число;
- `__str__(self)` — вызывается при использовании функций `str()` и `print()` применительно к текущему объекту. Должен возвращать строку. Если отсутствует, будет использован метод `__repr__()` (описан далее). Пример:

```
class MyClass:
    def __init__(self, m):
        self.msg = m
    def __str__(self):
        return "Вызван метод __str__() {0}".format(self.msg)
c = MyClass("Значение")
print(str(c))  # Выведет: Вызван метод __str__() Значение
print(c)       # Выведет: Вызван метод __str__() Значение
```

- `__repr__(self)` — вызывается при выводе текущего объекта интерпретатором, работающим в интерактивном режиме, и использовании функции `repr()` применительно к текущему объекту. Должен возвращать строку. Пример:

```
class MyClass:
    def __init__(self, m):
        self.msg = m
    def __repr__(self):
        return "Вызван метод __repr__() {0}".format(self.msg)
c = MyClass("Значение")
print(repr(c)) # Выведет: Вызван метод __repr__() Значение
```

- `__hash__(self)` — должен возвращать строковое представление текущего объекта, содержащее только цифры. Такое представление можно получить, использовав функцию `hash(<Значение>)`. Этот метод следует переопределить, если объекты текущего класса планируется использовать в качестве ключей словаря или внутри множества. Пример:

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __hash__(self):
        return hash(self.x)
```

```
c = MyClass(10)
d = {}
d[c] = "Значение"
print(d[c]) # Выведет: Значение
```

Классы поддерживают еще несколько специальных методов, которые применяются лишь в особых случаях и будут рассмотрены в *главе 15*.

13.6. Перегрузка операторов

Перегрузкой оператора называется изменение функциональности какого-либо из операторов, поддерживаемых Python, применительно к объектам определенного класса. Чтобы перегрузить оператор, необходимо в классе определить соответствующий специальный метод.

Для перегрузки математических операторов используются следующие методы:

- `x + y` — сложение: `x.__add__(y)`;
- `y + x` — сложение (объект класса справа): `x.__radd__(y)`;
- `x += y` — сложение и присваивание: `x.__iadd__(y)`;
- `x - y` — вычитание: `x.__sub__(y)`;
- `y - x` — вычитание (объект класса справа): `x.__rsub__(y)`;
- `x -= y` — вычитание и присваивание: `x.__isub__(y)`;
- `x * y` — умножение: `x.__mul__(y)`;
- `y * x` — умножение (объект класса справа): `x.__rmul__(y)`;
- `x *= y` — умножение и присваивание: `x.__imul__(y)`;
- `x / y` — деление: `x.__truediv__(y)`;
- `y / x` — деление (объект класса справа): `x.__rtruediv__(y)`;
- `x /= y` — деление и присваивание: `x.__itruediv__(y)`;
- `x // y` — деление с округлением вниз: `x.__floordiv__(y)`;
- `y // x` — деление с округлением вниз (объект класса справа): `x.__rfloordiv__(y)`;
- `x //= y` — деление с округлением вниз и присваивание: `x.__ifloordiv__(y)`;
- `x % y` — остаток от деления: `x.__mod__(y)`;
- `y % x` — остаток от деления (объект класса справа): `x.__rmod__(y)`;
- `x %= y` — остаток от деления и присваивание: `x.__imod__(y)`;
- `x ** y` — возведение в степень: `x.__pow__(y)`;
- `y ** x` — возведение в степень (объект класса справа): `x.__rpow__(y)`;
- `x **= y` — возведение в степень и присваивание: `x.__ipow__(y)`;
- `-x` — унарный минус: `x.__neg__()`;
- `+x` — унарный плюс: `x.__pos__()`;
- `abs(x)` — абсолютное значение: `x.__abs__()`.

Пример перегрузки математических операторов приведен в листинге 13.8.

Листинг 13.8. Перегрузка математических операторов

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __add__(self, y):            # Перегрузка оператора +
        print("Объект слева")
        return self.x + y
    def __radd__(self, y):           # Перегрузка оператора +
        print("Объект справа")
        return self.x + y
    def __iadd__(self, y):           # Перегрузка оператора +=
        print("Сложение с присваиванием")
        self.x += y
        return self
c = MyClass(50)
print(c + 10)                  # Выведет: Объект слева 60
print(20 + c)                  # Выведет: Объект справа 70
c += 30                        # Выведет: Сложение с присваиванием
print(c.x)                     # Выведет: 80
```

Специальные методы для перегрузки двоичных операторов:

- `~x` — двоичная инверсия: `x.__invert__()`;
- `x & y` — двоичное И: `x.__and__(y)`;
- `y & x` — двоичное И (объект класса справа): `x.__rand__(y)`;
- `x &= y` — двоичное И и присваивание: `x.__iand__(y)`;
- `x | y` — двоичное ИЛИ: `x.__or__(y)`;
- `y | x` — двоичное ИЛИ (объект класса справа): `x.__ror__(y)`;
- `x |= y` — двоичное ИЛИ и присваивание: `x.__ior__(y)`;
- `x ^ y` — двоичное исключающее ИЛИ: `x.__xor__(y)`;
- `y ^ x` — двоичное исключающее ИЛИ (объект класса справа): `x.__rxor__(y)`;
- `x ^= y` — двоичное исключающее ИЛИ и присваивание: `x.__ixor__(y)`;
- `x << y` — сдвиг влево: `x.__lshift__(y)`;
- `y << x` — сдвиг влево (объект класса справа): `x.__rlshift__(y)`;
- `x <<= y` — сдвиг влево и присваивание: `x.__ilshift__(y)`;
- `x >> y` — сдвиг вправо: `x.__rshift__(y)`;
- `y >> x` — сдвиг вправо (объект класса справа): `x.__rrshift__(y)`;
- `x >>= y` — сдвиг вправо и присваивание: `x.__irshift__(y)`.

Перегрузка операторов сравнения производится с помощью следующих специальных методов:

- `x == y` — равно: `x.__eq__(y)`;
- `x != y` — не равно: `x.__ne__(y)`;

- `x < y` — меньше: `x.__lt__(y)`;
- `x > y` — больше: `x.__gt__(y)`;
- `x <= y` — меньше или равно: `x.__le__(y)`;
- `x >= y` — больше или равно: `x.__ge__(y)`;
- `y in x` — проверка на вхождение: `x.__contains__(y)`.

Пример перегрузки операторов сравнения приведен в листинге 13.9.

Листинг 13.9. Перегрузка операторов сравнения

```
class MyClass:
    def __init__(self):
        self.x = 50
        self.arr = [1, 2, 3, 4, 5]
    def __eq__(self, y):           # Перегрузка оператора ==
        return self.x == y
    def __contains__(self, y):     # Перегрузка оператора in
        return y in self.arr

c = MyClass()
print("Равно" if c == 50 else "Не равно") # Выведет: Равно
print("Равно" if c == 51 else "Не равно") # Выведет: Не равно
print("Есть" if 5 in c else "Нет")        # Выведет: Есть
```

13.7. Статические методы и методы класса

Статический метод — это метод, принадлежащий классу, а не объекту. Создается он так же, как и обычный метод, только перед его определением указывается декоратор `@staticmethod`. Ссылка на текущий объект с первым параметром такому методу не передается, поэтому в теле статического метода невозможно получить доступ к обычным атрибутам и методам.

Статический метод можно вызвать как непосредственно у класса, записав инструкцию формата:

```
<Класс>.<Метод>([<Параметры через запятую>])
```

так и у любого объекта этого класса — посредством выражений формата, приведенного в *разд. 13.1*.

Пример использования статических методов показан в листинге 13.10.

Листинг 13.10. Статические методы

```
class MyClass:
    @staticmethod
    def func1(x, y):               # Статический метод
        return x + y
    def func2(self, x, y):         # Обычный метод
        return x + y
```

```
    def func3(self, x, y):
        return MyClass.func1(x, y) # Вызов статического метода из обычного

print(MyClass.func1(10, 20))       # Вызываем статический метод
c = MyClass()
print(c.func2(15, 6))              # Вызываем обычный метод
print(c.func1(50, 12))             # Вызываем статический метод через объект
print(c.func3(23, 5))              # Вызываем обычный метод,
                                   # вызывающий статический метод
```

Метод класса похож на статический, только ему первым параметром передается ссылка на текущий класс (не объект). Перед определением метода класса следует поставить декоратор `@classmethod`. Пример использования методов класса приведен в листинге 13.11.

Листинг 13.11. Методы класса

```
class MyClass:
    @classmethod
    def func(cls, x): # Метод класса
        print(cls, x)

MyClass.func(10)      # Вызываем метод класса у класса
c = MyClass()
c.func(50)            # Вызываем метод класса у объекта
```

Статические методы и методы класса часто применяются для выполнения каких-либо вычислений общего характера или для создания специфических объектов класса, содержащих определенные значения в своих атрибутах.

13.8. Абстрактные методы

Абстрактный метод предназначен не для непосредственного вызова, а для перекрытия в производных классах. В качестве тела такого метода указывается инструкция с пустым оператором `pass` (описан в *разд. 3.5*), а перед его определением ставится декоратор `@abstractmethod` из модуля `abc`. Кроме того, класс, содержащий абстрактные методы, должен быть производным от класса `ABC` из того же модуля.

При попытке создать объект производного класса, в котором не перекрыт абстрактный метод, генерируется исключение `TypeError`. Следует помнить, что это исключение генерируется только в том случае, если класс с абстрактными методами является производным от класса `ABC` (в противном случае ничего не происходит). Рассмотрим определение абстрактных методов на примере (листинг 13.12).

Листинг 13.12. Абстрактный метод

```
from abc import ABC, abstractmethod
class Class1(ABC):
    @abstractmethod
    def func(self, x):     # Абстрактный метод
        pass
```

```
class Class2(Class1):
    def func(self, x):      # Перекрываем абстрактный метод
        print(x)

class Class3(Class1):       # Класс не перекрывает абстрактный метод
    pass

c2 = Class2()
c2.func(50)                 # Выведет: 50
try:
    c3 = Class3()           # Ошибка. Метод func() не перекрыт
    c3.func(50)
except TypeError as msg:
    print(msg)              # Can't instantiate abstract class Class3
                            # with abstract methods func
```

Можно создавать абстрактные статические методы и абстрактные методы класса, для чего необходимые декораторы указываются одновременно, друг за другом (листинг 13.13).

Листинг 13.13. Абстрактный статический метод и абстрактный метод класса

```
from abc import ABC, abstractmethod
class MyClass(ABC):
    @staticmethod
    @abstractmethod
    def static_func(self, x):      # Абстрактный статический метод
        pass

    @classmethod
    @abstractmethod
    def class_func(self, x):       # Абстрактный метод класса
        pass
```

13.9. Закрытые атрибуты и методы

Закрытые атрибуты и *методы*, в отличие от общедоступных, доступны лишь внутри класса, из его собственных методов. При попытке обратиться к закрытому атрибуту или методу извне класса генерируется исключение `AttributeError`.

Превратить атрибут или метод в закрытый можно, предварив его имя двойным символом подчеркивания (__). Пример:

```
>>> class MyClass:
...     def __init__(self, a):
...         self.__a = a            # Закрытый атрибут
...     def __func(self):           # Закрытый метод
...         print(self.__a)
...     def func(self):             # Общедоступный метод
...         self.__func()           # Вызываем закрытый метод из общедоступного
...
```

```
>>> obj = MyClass(123)
>>> obj.__a            # Пытаемся получить значение закрытого атрибута — ошибка
    ... Фрагмент пропущен ...
    AttributeError: 'MyClass' object has no attribute '__a'
>>> obj.__func()       # Пытаемся вызвать закрытый метод — ошибка
    ... Фрагмент пропущен ...
    AttributeError: 'MyClass' object has no attribute '__func'. Did you mean:
    'func'?
>>> obj.func()
    123
```

Тем не менее доступ к закрытым атрибутам и методам все же можно получить извне класса, обратившись по идентификатору формата:

```
_<Имя класса>__<Имя атрибута или метода без двойного подчеркивания>
```

Например:

```
>>> obj._MyClass__a             # Получаем значение закрытого атрибута __a
    123
>>> obj._MyClass__func()        # Вызываем закрытый метод __func()
    123
```

Также можно указать набор атрибутов (но, к сожалению, не методов), которые будут доступны извне класса (остальные атрибуты автоматически станут закрытыми). Список со строковыми именами этих атрибутов следует присвоить атрибуту класса `__slots__`, а если общедоступным планируется сделать лишь один атрибут, можно присвоить строку с его именем. Пример показан в листинге 13.14.

Листинг 13.14. Использование атрибута `__slots__`

```
class MyClass:
    __slots__ = ["x", "y"]
    def __init__(self, a, b):
        self.x, self.y = a, b

c = MyClass(1, 2)
print(c.x, c.y)                 # Выведет: 1 2
c.x, c.y = 10, 20               # Изменяем значения атрибутов
print(c.x, c.y)                 # Выведет: 10 20
try:                            # Перехватываем исключения
    c.z = 50                    # Атрибут z не указан в __slots__,
                                # поэтому генерируется исключение
except AttributeError as msg:
    print(msg)                  # 'MyClass' object has no attribute 'z'
```

13.10. Свойства

Свойство — это своего рода атрибут, при обращении к которому вызывается один из трех следующих методов, определенных в классе:

- *геттер* — вызывается при попытке получить значение свойства. Должен возвращать результат, который и станет значением свойства;

Обычно геттер возвращает значение, хранящееся в каком-либо закрытом атрибуте. Однако иногда он вычисляет возвращаемое значение, основываясь на содержимом других атрибутов.

Если геттер отсутствует, свойство не будет доступно на чтение;

- *сеттер* — вызывается при попытке присвоить свойству новое значение. Должен принимать в качестве параметра новое значение и не должен возвращать результат.

  Обычно сеттер заносит новое значение свойства в специально предназначенный для его хранения закрытый атрибут. Помимо этого, он может выполнять какие-либо дополнительные действия.

  Если сеттер отсутствует, свойство не будет доступно для записи;

- *делетер* — вызывается при попытке удалить свойство.

  Обычно делетер либо удаляет закрытый атрибут, в котором хранится значение свойства, либо заносит в него значение `None`. Также он может выполнять какие-либо дополнительные действия.

  Если делетер отсутствует, свойство будет невозможно удалить.

Свойство создается вызовом функции `property()`. Формат функции:

```
property(fget=None][, ][fset=None[, ][fdel=None][, ][doc=None])
```

Параметр `fget` задает геттер, параметр `fset` — сеттер, `fdel` — делетер, `doc` — строку документирования.

В качестве результата возвращается объект, представляющий созданное свойство, который сразу следует сохранить в каком-либо атрибуте класса. Через этот атрибут и будет в дальнейшем выполняться доступ к свойству. Рассмотрим создание свойства на примере (листинг 13.15).

Листинг 13.15. Создание свойства вызовом функции `property()`

```
class MyClass:
    def get_var(self):          # Геттер
        return self.__var
    def set_var(self, value): # Сеттер
        self.__var = value
    def del_var(self):          # Делетер
        del self.__var
    v = property(fget=get_var, fset=set_var, fdel=del_var, doc="Свойство")

c = MyClass()
c.v = 35                    # Вызывается сеттер
print(c.v)                  # Вызывается геттер
del c.v                     # Вызывается делетер
```

Также поддерживается альтернативный способ создания свойств. Сначала у будущего геттера указывается декоратор `property`, после чего объект геттера получает поддержку методов `setter()` и `deleter()`. Далее у будущего сеттера следует вызвать упомянутый ранее метод `setter()` геттера в качестве декоратора, а у будущего делетера — точно таким же образом вызвать метод `deleter()` геттера. Соответствующий пример приведен в листинге 13.16.

Листинг 13.16. Создание свойства декоратором property

```
class MyClass:
    @property
    def v(self):                        # Геттер
        return self.__var
    @v.setter
    def v(self, value):                 # Сеттер
        self.__var = value
    @v.deleter
    def v(self):                        # Делетер
        del self.__var

c = MyClass()
c.v = 35                                # Вызывается сеттер
print(c.v)                              # Вызывается геттер
del c.v                                 # Вызывается делетер
```

Можно создать абстрактное свойство — в этом случае все реализующие его методы должны быть переопределены в производном классе. Абстрактное свойство создается с помощью декоратора `@abstractmethod` и класса `ABC` из модуля `abc` (см. *разд. 13.8*). Пример показан в листинге 13.17.

Листинг 13.17. Абстрактное свойство

```
from abc import ABC, abstractmethod
class MyClass(ABC):
    @property
    @abstractmethod
    def v(self):                        # Чтение
        return self.__var
    @v.setter
    @abstractmethod
    def v(self, value):                 # Запись
        self.__var = value
    @v.deleter
    @abstractmethod
    def v(self):                        # Удаление
        del self.__var
```

13.11. Декораторы классов

Помимо декораторов функций, Python поддерживает *декораторы классов* — функции, изменяющие поведение самих классов. В качестве параметра декоратор класса должен принимать ссылку на класс, поведение которого необходимо изменить, и должен возвращать ссылку на тот же класс или какой-либо другой. Пример декорирования класса показан в листинге 13.18.

Листинг 13.18. Декоратор класса

```
def deco(C):                          # Принимает объект определения класса
    print("Внутри декоратора")        # Производит какие-то действия
    return C                          # Возвращает объект определения класса

@deco
class MyClass:
    def __init__(self, value):
        self.v = value
c = MyClass(5)
print(c.v)
```

ГЛАВА 14

Исключения и их обработка

Исключение — это объект, генерируемый интерпретатором Python в случае возникновения ошибки и хранящий сведения о ней. *Обработка*, или *перехват*, исключения — это реагирование на его возникновение с целью ликвидировать последствия ошибки.

В программе могут возникать ошибки трех типов:

- *синтаксические* — ошибки в синтаксисе кода: неправильное написание языковых конструкций, идентификаторов, отсутствие закрывающей или открывающей кавычек и т. д. Как правило, интерпретатор предупреждает о наличии такой ошибки, и выполнение программы прерывается. Пример синтаксической ошибки:

```
>>> print("Нет завершающей кавычки!)
    SyntaxError: EOL while scanning string literal
```

- *логические* — ошибки в логике программы. Как правило, интерпретатор не предупреждает о наличии такой ошибки, и программа успешно выполняется, но выданный ею результат оказывается не тем, на который мы рассчитывали. Выявить и исправить такие ошибки весьма трудно;

- *ошибки времени выполнения* — ошибки, возникающие во время работы программы вследствие событий, которые не были предусмотрены программистом. Классическим примером служит деление на ноль:

```
>>> def test(x, y): return x / y
...
>>> test(4, 2)                          # Нормально
    2.0
>>> test(4, 0)                          # Ошибка
    Traceback (most recent call last):
      File "<pyshell#4>", line 1, in <module>
        test(4, 0)
      File "<pyshell#2>", line 1, in test
        def test(x, y): return x / y
    ZeroDivisionError: division by zero
```

В Python исключения также генерируются в качестве уведомлений о наступлении каких-либо событий. Например, метод `index()` генерирует исключение `ValueError`, если искомый фрагмент не входит в обрабатываемую строку:

```
>>> "Строка".index("текст")
    Traceback (most recent call last):
      File "<pyshell#5>", line 1, in <module>
        "Строка".index("текст")
    ValueError: substring not found
```

14.1. Обработчики исключений

Обработчик исключений — инструкция, предназначенная для обработки исключений. Формат этой инструкции:

```
try:
    <Блок, в котором могут возникать исключения>
except[ <Класс исключения 1>[ as <Переменная>]]:
    <Блок, выполняемый при возникновении исключений класса 1>
. . .
except[ <Класс исключения N>[ as <Переменная>]]:
    <Блок, выполняемый при возникновении исключения класса N>
[else:
    <Блок, выполняемый, если исключение не возникло>]
[finally:
    <Блок, выполняемый в любом случае>]
```

Языковых конструкций `except` может быть произвольное количество. В переменную, указанную в конструкции `except`, будет занесен объект исключения, и в соответствующем блоке можно будет извлечь из него какие-либо сведения о возникшем исключении.

Пример обработки исключения класса `ZeroDivisionError`, генерируемого при делении на ноль, приведен в листинге 14.1.

Листинг 14.1. Обработка деления на ноль

```
try:                                # Перехватываем исключения
    x = 1 / 0                       # Ошибка: деление на 0
except ZeroDivisionError:           # Указываем класс исключения
    print("Обработали деление на 0")
    x = 0
print(x)                            # Выведет: 0
```

Один обработчик исключений можно вложить в другой. Если вложенный обработчик не обработал возникшее исключение, оно всплывет во «внешний» обработчик. Если исключение в программе вообще нигде не обрабатывается, оно будет передано обработчику по умолчанию, встроенному в интерпретатор, который остановит выполнение программы и выведет стандартное сообщение об ошибке. Пример вложенных обработчиков приведен в листинге 14.2.

Листинг 14.2. Вложенные обработчики исключений

```
try:                                # Обрабатываем исключения
    try:                            # Вложенный обработчик
        x = 1 / 0                   # Ошибка: деление на 0
```

```
    except NameError:
        print("Неопределенный идентификатор")
    except IndexError:
        print("Несуществующий индекс")
    print("Выражение после вложенного обработчика")
except ZeroDivisionError:
    print("Обработка деления на 0")
    x = 0
print(x)                                # Выведет: 0
```

В этом примере во вложенном обработчике не указано исключение `ZeroDivisionError`, поэтому исключение «всплывет» в обработчик более высокого уровня.

После обработки исключения выполняются инструкции, расположенные сразу после обработчика. В нашем примере выполнится инструкция, выводящая значение переменной `x`, — `print(x)`. Инструкция `print("Выражение после вложенного обработчика")` выполнена не будет.

В языковой конструкции `except` можно указать сразу несколько исключений, записав их через запятую внутри круглых скобок (листинг 14.3).

Листинг 14.3. Обработка нескольких исключений

```
try:
    x = 1 / 0
except (NameError, IndexError, ZeroDivisionError):
    # Обработка сразу нескольких исключений
    x = 0
print(x) # Выведет: 0
```

Пример получения сведений об обрабатываемом исключении приведен в листинге 14.4.

Листинг 14.4. Получение сведений об исключении

```
try:
    x = 1 / 0                       # Ошибка деления на 0
except (NameError, IndexError, ZeroDivisionError) as err:
    print(err.__class__.__name__) # Имя класса исключения
    print(err)                      # Текст сообщения об ошибке
```

Результат выполнения:

```
ZeroDivisionError
division by zero
```

Для получения сведений об исключении можно воспользоваться функцией `exc_info()` из модуля `sys`, которая возвращает кортеж из трех элементов: класса исключения, объекта исключения и объекта с трассировочной информацией. Преобразовать эти значения в удобочитаемый вид позволяет модуль `traceback`. Пример использования функции `exc_info()` и модуля `traceback` приведен в листинге 14.5.

Листинг 14.5. Использование функции exc_info()

```
import sys, traceback
try:
    x = 1 / 0
except ZeroDivisionError:
    Type, Value, Trace = sys.exc_info()
    print("Type: ", Type)
    print("Value:", Value)
    print("Trace:", Trace)
    print("\n", "print_exception()".center(40, "-"))
    traceback.print_exception(Type, Value, Trace, limit=5,
                              file=sys.stdout)
    print("\n", "print_tb()".center(40, "-"))
    traceback.print_tb(Trace, limit=1, file=sys.stdout)
    print("\n", "format_exception()".center(40, "-"))
    print(traceback.format_exception(Type, Value, Trace, limit=5))
    print("\n", "format_exception_only()".center(40, "-"))
    print(traceback.format_exception_only(Type, Value))
```

Результат выполнения примера:

```
Type: <class 'ZeroDivisionError'>
Value: division by zero
Trace: <traceback object at 0x00000274303A7040>

-----------print_exception()------------
Traceback (most recent call last):
File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
приложений/FTP/14/14.5.py", line 3, in <module>
 x = 1 / 0
ZeroDivisionError: division by zero

---------------print_tb()---------------
File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
приложений/FTP/14/14.5.py", line 3, in <module>
 x = 1 / 0

-----------format_exception()-----------
['Traceback (most recent call last):\n',
' File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
 приложений/FTP/14/14.5.py", line 3, in <module>\n  x = 1 / 0\n',
 'ZeroDivisionError: division by zero\n']

--------format_exception_only()---------
['ZeroDivisionError: division by zero\n']
```

Если в языковой конструкции except не указан класс исключения, то записанный в ней блок будет перехватывать все исключения. На практике следует избегать пустых инструкций

except, иначе можно перехватить исключение, которое является лишь сигналом системе, а не ошибкой. Пример перехвата всех исключений приведен в листинге 14.6.

Листинг 14.6. Перехват всех исключений

```
try:
    x = 1 / 0                    # Ошибка деления на 0
except:                          # Обработка всех исключений
    x = 0
print(x)                         # Выведет: 0
```

Если в обработчике присутствует языковая конструкция else, то указанный в ней блок будет выполнен только при отсутствии ошибок. А блок в языковой конструкции finally выполнится вне зависимости от того, возникло исключение или нет. Для примера выведем последовательность выполнения этих блоков (листинг 14.7).

Листинг 14.7. Языковые конструкции else и finally

```
try:
    x = 10 / 2                   # Нет ошибки
    #x = 10 / 0                  # Ошибка деления на 0
except ZeroDivisionError:
    print("Деление на 0")
else:
    print("Блок else")
finally:
    print("Блок finally")
```

Результат выполнения при отсутствии исключения:

```
Блок else
Блок finally
```

Последовательность выполнения блоков при наличии исключения будет другой:

```
Деление на 0
Блок finally
```

Необходимо заметить, что при наличии исключения и отсутствии языковой конструкции except блок в конструкции finally будет выполнен, но исключение останется необработанным и «всплывет» в обработчик более высокого уровня. Если таковой отсутствует, исключение передается обработчику по умолчанию, который прервет выполнение программы и выведет сообщение об ошибке, например:

```
>>> try:
...     x = 10 / 0
... finally: print("Блок finally")
...
Блок finally
Traceback (most recent call last):
  File "<pyshell#17>", line 2, in <module>
    x = 10 / 0
ZeroDivisionError: division by zero
```

Переделаем нашу программу суммирования произвольного количества целых чисел, введенных пользователем (см. листинг 4.13), таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 14.8).

Листинг 14.8. Суммирование неопределенного количества чисел

```
print("Введите слово 'stop' для получения результата")
summa = 0
while (x := input("Введите число: ")) != "stop":
    try:
        x = int(x) # Преобразуем строку в число
    except ValueError:
        print("Необходимо ввести целое число!")
    else:
        summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода значений и получения результата выглядит так (значения, введенные пользователем, выделены полужирным шрифтом):

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число: str
Необходимо ввести целое число!
Введите число: -5
Введите число:
Необходимо ввести целое число!
Введите число: stop
Сумма чисел равна: 5
```

14.2. Обработчики контекстов

Контекст — это объект, который следует создать, выполнить с ним заданные действия и корректно уничтожить, вне зависимости от того, возникли исключения при выполнении действий с ним или нет. Пример контекста — объект, представляющий файл, который в любом случае должен быть закрыт после работы с ним. Класс, на основе которого создается объект-контекст, называется *менеджером контекста.*

Обработчик контекста — это выражение, создающее контекст (или сразу несколько таковых), выполняющее с ним заданные действия и самостоятельно уничтожающее его. Такое выражение записывается в любом из двух следующих форматов:

```
with <Выражение, создающее контекст 1>[ as <Переменная 1>],
     . . .,
     <Выражение, создающее контекст N>[ as <Переменная N>]]:
    <Блок, работающий с созданными контекстами>
with (<Выражение, создающее контекст 1>[ as <Переменная 1>],
      . . .,
      <Выражение, создающее контекст N>[ as <Переменная N>]]):
    <Блок, работающий с созданными контекстами>
```

Контексты, созданные заданными выражениями, заносятся в указанные переменные, и эти переменные будут доступны в блоке, который работает с созданными контекстами.

Класс, являющийся менеджером контекста, должен поддерживать два специальных метода:

- `__enter__(self)` — вызывается сразу после создания объекта текущего класса. Должен возвращать ссылку на созданный объект;
- `__exit__()` — выполняется либо по завершении исполнения блока в выражении обработчика контекста (если исключение не возникло), либо сразу после возникновения исключения. Формат метода:

```
__exit__(self, <Класс исключения>, <Объект исключения>,
                                   <Объект с трассировочной информацией>)
```

  Если исключение не возникло, последние три параметра получат значение `None`, и метод не должен возвращать результат. В противном случае метод должен вернуть значение `True`, если возникшее исключение было им обработано, или `False`, если он не обработал исключение (которое в этом случае будет передано вышестоящему обработчику).

Пример обработчика контекста приведен в листинге 14.9.

Листинг 14.9. Обработчик контекста

```
class MyClass:
    def __enter__(self):
        print("Вызван метод __enter__()")
        return self
    def __exit__(self, Type, Value, Trace):
        print("Вызван метод __exit__()")
        if Type is None:  # Если исключение не возникло
            print("Исключение не возникло")
        else:             # Если возникло исключение
            print("Value =", Value)
            return False  # False – исключение не обработано
                          # True  – исключение обработано

print("Последовательность при отсутствии исключения:")
with MyClass():
    print("Блок внутри with")
print("\nПоследовательность при наличии исключения:")
with MyClass() as obj:
    print("Блок внутри with")
    raise TypeError("Исключение TypeError")
```

Результат выполнения:

```
Последовательность при отсутствии исключения:
Вызван метод __enter__()
Блок внутри with
Вызван метод __exit__()
Исключение не возникло
```

```
Последовательность при наличии исключения:
Вызван метод __enter__()
Блок внутри with
Вызван метод __exit__()
Value = Исключение TypeError
Traceback (most recent call last):
File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
приложений/FTP/14/14.9.py", line 21, in <module>
 raise TypeError("Исключение TypeError")
TypeError: Исключение TypeError
```

Второй формат обработчиков контекстов (с круглыми скобками) поддерживается, начиная с Python 3.10, и позволяет разбивать языковую конструкцию `with` на несколько строк:

```
with (Class1() as obj1, Class2() as obj2, Class3() as obj3,
      Class4() as obj4, Class5() as obj5):
    . . .
```

Некоторые встроенные классы (например, класс файла) также являются менеджерами контекста. Они будут рассмотрены в последующих главах.

14.3. Классы встроенных исключений

Иерархия встроенных в Python классов исключений схематично выглядит так:

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
            FloatingPointError, OverflowError, ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        ImportError
            ModuleNotFoundError
        LookupError
            IndexError, KeyError
        MemoryError
        NameError
            UnboundLocalError
        OSError
            BlockingIOError
            ChildProcessError
            ConnectionError
                BrokenPipeError, ConnectionAbortedError,
                ConnectionRefusedError, ConnectionResetError
            FileExistsError
```

```
        FileNotFoundError
        InterruptedError
        IsADirectoryError
        NotADirectoryError
        PermissionError
        ProcessLookupError
        TimeoutError
    RuntimeError
        NotImplementedError, RecursionError
    SyntaxError
        IndentationError
            TabError
    SystemError
    TypeError
    ValueError
        UnicodeError
            UnicodeDecodeError, UnicodeEncodeError
            UnicodeTranslateError
    Warning
        BytesWarning, DeprecationWarning, FutureWarning, ImportWarning,
        PendingDeprecationWarning, ResourceWarning, RuntimeWarning,
        SyntaxWarning, UnicodeWarning, UserWarning
```

Можно указать базовый класс для перехвата всех исключений соответствующих производных классов. Например, указав базовый класс `ArithmeticError`, можно перехватывать исключения классов `FloatingPointError`, `OverflowError` и `ZeroDivisionError`:

```
try:
    x = 1 / 0                       # Ошибка: деление на 0
except ArithmeticError:             # Указываем базовый класс
    print("Обработали деление на 0")
```

Основные классы встроенных исключений:

- `BaseException` — базовый класс для всех прочих классов исключений;
- `Exception` — базовый класс для большинства исключений, встроенных в Python, а также пользовательских исключений;
- `AssertionError` — генерируется языковой конструкцией `assert`;
- `AttributeError` — попытка обращения к несуществующему атрибуту или методу объекта;
- `EOFError` — достигнут конец файла, из которого выполняется чтение;
- `ImportError` — невозможно выполнить импорт модуля или пакета вследствие ошибки в инструкции импорта;
- `IndentationError` — неправильно расставлены отступы в программе;
- `IndexError` — указанный индекс не существует в последовательности;
- `KeyError` — указанный ключ не существует в отображении;
- `KeyboardInterrupt` — нажата комбинация клавиш <Ctrl>+<C>;
- `MemoryError` — интерпретатору существенно не хватает оперативной памяти, и он вынужден срочно удалить неиспользуемые объекты, чтобы освободить ее;

- `ModuleNotFoundError` — импортируемый модуль или пакет не найден;
- `NameError` — попытка обращения к несуществующему идентификатору;
- `NotImplementedError` — попытка обращения к абстрактному методу;
- `OSError` — базовый класс для всех исключений, генерируемых в ответ на возникновение системных ошибок (отсутствие запрошенного файла, недостаток места на диске и пр.);
- `OverflowError` — число, получившееся в результате выполнения арифметической операции, слишком велико, чтобы Python смог его обработать;
- `RecursionError` — превышено максимальное количество проходов рекурсии;
- `RuntimeError` — неклассифицированная ошибка времени выполнения;
- `StopIteration` — генерируется методом `__next__()` как сигнал об окончании итераций;
- `SyntaxError` — синтаксическая ошибка;
- `SystemError` — ошибка в самом интерпретаторе Python;
- `TabError` — в программном коде встретился символ табуляции, использование которого для создания отступов недопустимо;
- `TypeError` — тип объекта не соответствует ожидаемому;
- `UnboundLocalError` — в теле функции создается локальная переменная после обращения к одноименной глобальной переменной;
- `UnicodeDecodeError` — ошибка преобразования последовательности байтов в Unicode-строку;
- `UnicodeEncodeError` — ошибка преобразования Unicode-строки в последовательность байтов;
- `UnicodeError` — базовый класс для исключений, генерируемых при ошибке преобразования Unicode-строк в последовательности байтов и наоборот;
- `UnicodeTranslationError` — ошибка преобразования Unicode-строки в другую кодировку;
- `ValueError` — неприемлемое значение параметра, переданного функции или методу;
- `ZeroDivisionError` — деление на ноль.

14.4. Генерирование исключений

Для программного генерирования исключений предназначена инструкция, записываемая в одном из следующих форматов:

```
raise <Объект исключения>
raise <Класс исключения>
raise <Объект или класс вторичного исключения> from ↳
<Объект первичного исключения>
raise
```

Первый формат генерирует исключение, чей объект указан в инструкции. При создании объекта исключения можно передать конструктору класса данные, которые могут быть извлечены в обработчике (см. *разд. 14.1*). Пример генерирования исключения `ValueError`:

```
>>> raise ValueError("Описание исключения")
    Traceback (most recent call last):
      File "<pyshell#0>", line 1, in <module>
        raise ValueError("Описание исключения")
    ValueError: Описание исключения
```

Пример обработки этого исключения показан в листинге 14.10.

Листинг 14.10. Принудительное генерирование исключения

```
try:
    raise ValueError("Описание исключения")
except ValueError as msg:
    print(msg) # Выведет: Описание исключения
```

Второй формат инструкции генерирует исключение на основе указанного класса, при этом объект исключения создается автоматически:

```
try:
    raise ValueError # Эквивалентно: raise ValueError()
except ValueError:
    print("Сообщение об ошибке")
```

Третий формат инструкции при возникновении одного исключения (*первичного*) генерирует другое (*вторичное*). Первичное исключение сохраняется в атрибуте __cause__ объекта вторичного исключения. Пример использования этого формата можно увидеть в листинге 14.11.

Листинг 14.11. Применение третьего формата конструкции `raise`

```
try:
    x = 1 / 0
except Exception as err:
    raise ValueError() from err
```

Результат выполнения:

```
Traceback (most recent call last):
File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
приложений/FTP/14/14.11.py", line 2, in <module>
 x = 1 / 0
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
приложений/FTP/14/14.11.py", line 4, in <module>
 raise ValueError() from err
ValueError
```

Как видно из результата, мы получили информацию не только по исключению ValueError, но и по исключению ZeroDivisionError. Следует заметить, что при отсутствии языковой конструкции from информация о первичном исключении сохраняется в объекте вторичного исключения неявным образом. Если убрать конструкцию from в предыдущем примере, мы получим следующий результат:

```
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
  приложений/FTP/14/14.11.py", line 2, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
  приложений/FTP/14/14.11.py", line 4, in <module>
    raise ValueError()
ValueError
```

Четвертый формат инструкции повторно генерирует последнее возникшее исключение и обычно применяется в коде, следующем за конструкцией except. Пример использования этого формата показан в листинге 14.12.

Листинг 14.12. Применение четвертого формата конструкции raise

```
class MyError(Exception): pass
try:
    raise MyError("Сообщение об ошибке")
except MyError as err:
    print(err)
    raise               # Повторно генерируем исключение
```

Результат выполнения:

```
Сообщение об ошибке
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
  приложений/FTP/14/14.12.py", line 3, in <module>
    raise MyError("Сообщение об ошибке")
MyError: Сообщение об ошибке
```

14.5. Пользовательские исключения

Пользовательское исключение — это исключение, созданное программистом. В качестве пользовательского исключения всегда выступает класс, производный от класса Exception. Пример определения и использования пользовательского исключения показан в листинге 14.13.

Листинг 14.13. Пользовательское исключение

```
class MyError(Exception):
    def __init__(self, value):
        self.msg = value
    def __str__(self):
        return self.msg

# Обработка пользовательского исключения
try:
    raise MyError("Описание исключения")
except MyError as err:
    print(err)        # Вызывается метод __str__()
    print(err.msg)    # Обращение к атрибуту класса
# Повторно генерируем исключение
raise MyError("Описание исключения")
```

Результат выполнения:

```
Описание исключения
Описание исключения
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 6 Разработка
  приложений/FTP/14/14.13.py", line 14, in <module>
    raise MyError("Описание исключения")
MyError: Описание исключения
```

Класс `Exception` поддерживает все необходимые методы для вывода сообщения об ошибке. Поэтому в большинстве случаев для создания пользовательского исключения достаточно определить пустой класс, производный от него (листинг 14.14).

Листинг 14.14. Упрощенный способ создания пользовательского исключения

```
class MyError(Exception): pass
try:
    raise MyError("Описание исключения")
except MyError as err:
    print(err)        # Выведет: Описание исключения
```

14.6. Проверочная инструкция

Проверочная инструкция позволяет удостовериться, что заданное условие выполняется. Если же это не так, инструкция сгенерирует исключение `AssertionError`.

Формат записи проверочной инструкции:

```
assert <Условие>[, <Значение]
```

Указанное значение будет передано конструктору класса `AssertionError` в качестве параметра.

Проверочная инструкция эквивалентна следующему коду:

```
if __debug__:
    if not <Условие>:
        raise AssertionError(<Значение>)
```

Если при запуске программы используется командный ключ `-O`, то переменная `__debug__` будет хранить значение `False`. Таким образом, если запустить программу без ключа `-O`, проверочные инструкции выполняться не будут.

Пример использования инструкции `assert` представлен в листинге 14.15.

Листинг 14.15. Проверочная инструкция

```
try:
    x = -3
    assert x >= 0, "Сообщение об ошибке"
except AssertionError as err:
    print(err) # Выведет: Сообщение об ошибке
```

ГЛАВА 15

Итераторы, контейнеры и перечисления

Python позволяет создавать классы особого назначения: итераторы, контейнеры и перечисления.

15.1. Итераторы

Итератор — это класс, который при каждом обращении выдает очередной элемент заданной последовательности. Объект такого класса можно перебрать в цикле перебора последовательности. Можно сказать, что итератор аналогичен функции-генератору (см. *разд. 11.3*), только является классом.

Чтобы превратить класс в итератор, следует переопределить в нем два специальных метода:

- `__iter__(self)` — служит признаком того, что класс является итератором. Должен возвращать текущий объект. Как правило, этот метод выполняет всевозможные предустановки.

  Если в классе определены методы `__iter__()` и `__getitem__()` (о последнем будет рассказано позже), предпочтение отдается первому методу;

- `__next__(self)` — вызывается при выполнении каждой итерации и должен возвращать очередной элемент последовательности. Если последовательность закончилась, в этом методе следует сгенерировать исключение `StopIteration`, которое сообщит вызывающему коду о завершении перебора.

Рассмотрим класс, хранящий строку и на каждой итерации возвращающий очередной ее символ, начиная с конца (листинг 15.1).

Листинг 15.1. Класс-итератор

```
class ReverseString:
    def __init__(self, s):
        self.__s = s
    def __iter__(self):
        self.__i = 0
        return self
    def __next__(self):
        if self.__i > len(self.__s) - 1:
            raise StopIteration
```

```
        else:
            a = self.__s[-self.__i - 1]
            self.__i = self.__i + 1
            return a
```

Проверим его в действии:

```
>>> s = ReverseString("Python")
>>> for a in s: print(a, end="")
...
    nohtyP
```

Также мы можем переопределить специальный метод `__len()__`, возвращающий размер последовательности, и специальные методы `__str()__` и `__repr()__`, выдающие строковое представление объекта (были рассмотрены в *разд. 13.5*). Перепишем код нашего класса-итератора, добавив в него определение методов `__len()__` и `__str()__` (в листинге 15.2 часть кода опущена).

Листинг 15.2. Расширенный класс-итератор

```
class ReverseString:
    . . .
    def __len__(self):
        return len(self.__s)
    def __str__(self):
        return self.__s[::-1]
```

Теперь мы можем получить длину последовательности, хранящейся в объекте класса `ReverseString`, и его строковое представление:

```
>>> s = ReverseString("Python")
>>> len(s)
    6
>>> str(s)
    'nohtyP'
```

15.2. Контейнеры

Контейнер — это класс с функциональностью либо пронумерованной последовательности (*контейнер-последовательность*) с произвольным доступом к любому ее элементу по его индексу, либо отображения (*контейнер-отображение*).

15.2.1. Контейнеры-последовательности

Чтобы превратить класс в контейнер-последовательность, следует переопределить в нем следующие специальные методы:

- `__getitem__(self, <Индекс>)` — вызывается при извлечении элемента последовательности с заданным индексом и должен возвращать этот элемент. Если индекс не является целым числом или срезом, метод должен генерировать исключение `TypeError`, а если такого индекса не существует, — исключение `IndexError`;

- `__setitem__(self, <Индекс>, <Значение>)` — вызывается в случае присваивания нового значения элементу последовательности с заданным индексом. Не должен возвращать результата. В случае задания индекса недопустимого типа или отсутствия такого индекса в последовательности метод должен генерировать те же исключения, что и метод `__getitem__()`;
- `__delitem__(self, <Индекс>)` — вызывается в случае удаления элемента последовательности с заданным индексом. Не должен возвращать результата. В случае задания индекса недопустимого типа или отсутствия такого индекса в последовательности метод должен генерировать те же исключения, что и метод `__getitem__()`;
- `__contains__(self, <Значение>)` — вызывается при проверке существования в последовательности элемента с заданным значением. Должен возвращать `True`, если такой элемент есть, и `False` — в противном случае.

В контейнере-последовательности можно дополнительно реализовать функциональность итератора (см. *разд. 15.1*), переопределив специальные методы `__iter__()`, `__next__()` и `__len__()`. Чаще всего так и поступают.

Мы давно знаем, что строки в Python являются неизменяемыми. Напишем класс `MutableString`, представляющий строку, которую можно изменять теми же способами, что и список (листинг 15.3).

Листинг 15.3. Класс `MutableString`

```
class MutableString:
    def __init__(self, s):
        self.__s = list(s)

    # Реализуем функциональность итератора
    def __iter__(self):
        self.__i = 0
        return self
    def __next__(self):
        if self.__i > len(self.__s) - 1:
            raise StopIteration
        else:
            a = self.__s[self.__i]
            self.__i = self.__i + 1
            return a
    def __len__(self):
        return len(self.__s)
    def __str__(self):
        return "".join(self.__s)

    # Определяем вспомогательный метод, который будет проверять
    # корректность индекса
    def __iscorrectindex(self, i):
        if type(i) == int or type(i) == slice:
            if type(i) == int and i > self.__len__() - 1:
                raise IndexError
        else:
            raise TypeError
```

```
    # Реализуем функциональность контейнера-последовательности
    def __getitem__(self, i):
        self.__iscorrectindex(i)
        return self.__s[i]
    def __setitem__(self, i, v):
        self.__iscorrectindex(i)
        self.__s[i] = v
    def __delitem__(self, i):
        self.__iscorrectindex(i)
        del self.__s[i]
    def __contains__(self, v):
        return v in self.__s
```

Проверим свеженаписанный класс в действии:

```
>>> s = MutableString("Python")
>>> print(s[-1])
    n
>>> s[0] = "J"
>>> print(s)
    Jython
>>> del s[2:4]
>>> print(s)
    Juon
```

Проверим, как наш класс обрабатывает нештатные ситуации, обратившись по несуществующему индексу:

```
>>> s[9] = "u"
    ... Фрагмент пропущен ...
    IndexError
```

15.2.2. Контейнеры-отображения

Превратить класс в контейнер-отображение можно, переопределив описанные в *разд. 15.2.1* методы: `__getitem__()`, `__setitem__()`, `__delitem__()` и `__contains__()`. При этом следует учесть, что вместо индексов здесь будут использоваться ключи произвольного типа (как правило, строкового). А при обращении по несуществующему ключу следует генерировать исключение `KeyError`.

Напишем класс `Version`, который будет хранить версию интерпретатора Python, разбитую на части: старшая версия, младшая версия и модификация. Доступ к частям версии будет осуществляться по строковым ключам (листинг 15.4). Ради простоты функциональность итератора реализовывать не станем, а также заблокируем операцию удаления элемента словаря, генерируя в методе `__delitem__()` исключение `NotImplementedError`.

Листинг 15.4. Класс Version

```
class Version:
    def __init__(self, major, minor, sub):
        self.__major = major                # Старшая версия
        self.__minor = minor                # Младшая версия
        self.__sub = sub                    # Модификация
```

```
    def __str__(self):
        return str(self.__major) + "."  + str(self.__minor) + "." + \
        str(self.__sub)

    # Реализуем функциональность отображения
    def __getitem__(self, k):
        if k == "major":
            return self.__major
        elif k == "minor":
            return self.__minor
        elif k == "sub":
            return self.__sub
        else:
            raise KeyError
    def __setitem__(self, k, v):
        if k == "major":
            self.__major = v
        elif k == "minor":
            self.__minor = v
        elif k == "sub":
            self.__sub = v
        else:
            raise KeyError
    def __delitem__(self, k):
        raise NotImplementedError
    def __contains__(self, v):
        return v == "major" or v == "minor" or v == "sub"
```

Проверим новый класс в работе:

```
>>> v = Version(3, 10, 1)
>>> v["major"]
 3
>>> v["minor"]
 10
>>> v["sub"] = 2
>>> str(v)
    '3.10.2'
>>> v["some"]
    ... Фрагмент пропущен ...
    KeyError
```

15.3. Перечисления

Перечисление — это набор каких-либо именованных значений, называемых *элементами*. Перечисление можно рассматривать как своего рода словарь, только неизменяемый.

Перечисление представляет собой класс, содержащий непосредственные определения атрибутов класса, каждый из которых представляет один из элементов. Имя атрибута класса

станет именем соответствующего элемента. Соглашения Python-программирования требуют, чтобы имена элементов перечисления были набраны в верхнем регистре.

Соответственно, для доступа к нужному элементу перечисления следует обратиться к представляющему его атрибуту класса посредством привычной точечной нотации.

Класс перечисления должен быть производным от одного из следующих четырех классов, определенных в модуле `enum`:

- `Enum` — базовый класс для создания перечислений, чьи элементы способны хранить значения произвольного типа. Элементы такого перечисления могут содержать одинаковые значения.

  Пример перечисления, содержащего элементы с названиями различных веб-фреймворков, в том числе два элемента с одинаковыми значениями:

```
>>> from enum import Enum
>>> class Frameworks(Enum):
...     LARAVEL = "Laravel"
...     DJANGO = "Django"
...     EXPRESS = "Express"
...     RAILS = "Ruby on Rails"
...     RUBY_ON_RAILS = "Ruby on Rails"
```

  Выведем значение одного из элементов перечисления:

```
>>> Frameworks.DJANGO
    <Frameworks.DJANGO: 'Django'>
```

  Присвоим значение другого элемента переменной и сравним с третьим элементом:

```
>>> fr = Frameworks.LARAVEL
>>> fr == Frameworks.RAILS
    False
```

  Элементы с одинаковыми значениями равны друг другу:

```
>>> Frameworks.RAILS == Frameworks.RUBY_ON_RAILS
    True
```

  Создать перечисление, элементы которого гарантированно содержат только уникальные значения, можно, указав у его класса декоратор `unique` из модуля `enum`:

```
>>> from enum import Enum, unique
>>> @unique
... class Frameworks2(Enum):
...     LARAVEL = "Laravel"
...     DJANGO = "Django"
...     EXPRESS = "Express"
...     RAILS = "Ruby on Rails"
```

  Пример перечисления с целыми числами:

```
>>> class Colors(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     WHITE = RED + GREEN + BLUE
...
```

```
>>> Colors.RED
    <Colors.RED: 1>
>>> Colors.WHITE
    <Colors.WHITE: 6>
```

Однако выполнять арифметические операции с элементами такого перечисления нельзя, поскольку их значения не являются целыми числами:

```
>>> Colors.GREEN + 3
    ... Фрагмент пропущен ...
    TypeError: unsupported operand type(s) for +: 'Colors' and 'int'
```

Если конкретные значения элементов целочисленного перечисления не важны, для их занесения можно использовать функцию `auto()` из модуля `enum`. Эта функция возвращает последовательно увеличивающиеся целые числа, начиная с 1. Пример:

```
>>> from enum import Enum, auto
>>> class Colors2(Enum):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     WHITE = auto()
...
>>> Colors2.RED
    <Colors2.RED: 1>
>>> Colors2.WHITE
    <Colors2.WHITE: 4>
```

- `IntEnum` — базовый класс для создания перечислений, элементы которых хранят лишь целочисленные значения. Значения элементов преобразуются в целые числа и, следовательно, могут быть использованы в арифметических операциях и выражениях сравнения с целыми числами. Пример:

```
>>> from enum import IntEnum
>>> class Colors3(IntEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     WHITE = RED + GREEN + BLUE
...
>>> Colors3.GREEN
    <Colors3.GREEN: 2>
>>> Colors3.GREEN + 3
    5
>>> Colors3.GREEN * Colors3.BLUE
    6
>>> Colors3.WHITE == 6
    True
```

- `Flag` — базовый класс для перечислений, элементы которых хранят целочисленные значения. Значения элементов могут выступать в качестве операндов для двоичных операторов (описаны в *разд. 3.2*). Функция `auto()` при использовании в таком перечислении последовательно выдает значения 1 и числа, являющиеся степенями числа 2 (2, 4, 8, 16 и т. д.). Пример:

```
>>> from enum import Flag, auto
>>> class Colors4(Flag):
...     BLACK = 0
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     WHITE = RED | GREEN | BLUE
...
>>> Colors4.BLACK
    <Colors4.BLACK: 0>
>>> Colors4.RED
    <Colors4.RED: 1>
>>> Colors4.GREEN
    <Colors4.GREEN: 2>
>>> Colors4.BLUE
    <Colors4.BLUE: 4>
>>> Colors4.WHITE
    <Colors4.WHITE: 7>
>>> Colors4.RED & Colors4.GREEN
    <Colors4.BLACK: 0>
>>> Colors4.RED | Colors4.GREEN
    <Colors4.GREEN|RED: 3>
>>> bool(Colors4.WHITE & Colors4.BLUE)
    True
```

Однако при манипуляциях с элементами такого перечисления использовать в качестве операндов двоичных операторов целые числа нельзя:

```
>>> Colors4.RED | 2
    ... Фрагмент пропущен ...
    TypeError: unsupported operand type(s) for |: 'Colors4' and 'int'
```

- `IntFlag` — аналогичен `Flag`, только значения элементов преобразуются в целые числа. Следовательно, при манипуляциях с элементами такого перечисления можно использовать в качестве операндов двоичных операторов целочисленные значения. Пример:

```
>>> from enum import IntFlag, auto
>>> class Colors5(IntFlag):
...     BLACK = 0
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     WHITE = RED | GREEN | BLUE
...
>>> Colors5.RED | 2
    <Colors5.GREEN|RED: 3>
>>> Colors5.WHITE == 7
    True
```

Все классы перечислений принадлежат типу `EnumMeta` из модуля `enum`:

```
>>> type(Colors)
    <class 'enum.EnumMeta'>
```

```
>>> from enum import EnumMeta
>>> type(Colors) == EnumMeta
    True
```

Значения элементов перечислений являются объектами их классов:

```
>>> type(Colors.RED)
    <enum 'Colors'>
>>> type(Colors.RED) == Colors
    True
```

Действия, которые можно выполнять над элементами перечислений, были описаны ранее. Помимо того, над элементами перечислений можно производить следующие операции:

- обращение к элементу в стиле словарей, используя имя элемента в качестве ключа:

```
>>> Frameworks["EXPRESS"]
    <Frameworks.EXPRESS: 'Express'>
```

- обращение к элементу по его значению, записывая его в круглых скобках после имени класса перечисления:

```
>>> Frameworks("Laravel")
    <Frameworks.LARAVEL: 'Laravel'>
```

- получение имени атрибута класса перечисления, соответствующего заданному элементу, и его значения из свойств элементов `name` и `value` соответственно:

```
>>> Frameworks.RAILS.name, Frameworks.RAILS.value
    ('RAILS', 'Ruby on Rails')
```

- проверка на вхождение или невхождение элемента в перечисление с помощью операторов `in` и `not in` соответственно:

```
>>> f = Frameworks.DJANGO
>>> f in Frameworks
    True
>>> f not in Frameworks
    False
>>> f in Colors
    False
```

Перечисление можно использовать в качестве итератора (необходимая для этого функциональность определена в базовом классе):

```
>>> list(Colors)
    [<Colors.RED: 1>, <Colors.GREEN: 2>, <Colors.BLUE: 3>, <Colors.WHITE: 6>]
>>> for f in Frameworks: print(f.value, end=" | ")
...
    Laravel | Django | Express | Ruby on Rails |
```

Помимо элементов, классы перечислений могут содержать методы, включая статические. При этом обычные методы всегда вызываются у элемента перечисления (и соответственно первым параметром любому методу передается ссылка на объект, представляющий элемент перечисления, у которого вызывается этот метод), а статические методы — у самого класса перечисления. Для примера рассмотрим код перечисления `Frameworks3` (листинг 15.5).

Листинг 15.5. Перечисление, включающее методы

```
from enum import Enum
class Frameworks3(Enum):
    LARAVEL = "Laravel"
    DJANGO = "Django"
    EXPRESS = "Express"
    RAILS = "Ruby on Rails"

    # Обычные методы. Вызываются у элемента перечисления.
    def describe(self):
        return self.name, self.value
    def __str__(self):
        return str(__class__.__name__) + "." + self.name + ": " + \
        self.value

    # Статический метод.  Вызывается у самого класса перечисления.
    @classmethod
    def getmostpopular(cls):
        return cls.LARAVEL
```

В методе `__str__()` использована встроенная переменная `__class__`, хранящая ссылку на объект определения текущего класса. Атрибут `__name__` этого объекта содержит имя класса в виде строки.

Проверим класс в работе:

```
>>> Frameworks3.DJANGO.describe()
    ('DJANGO', 'Django')
>>> str(Frameworks3.EXPRESS)
    'Frameworks3.EXPRESS: Express'
>>> Frameworks3.getmostpopular()
    <Frameworks3.LARAVEL: 'Laravel'>
```

Можно определить класс перечисления, содержащий только методы и *не включающий элементы*, а потом наследовать от него производный класс, уже содержащий элементы. Производный класс получит от базового поддержку всех его методов. Однако попытка создать производный класс на основе базового, *содержащего элементы*, приведет к ошибке.

Перечисления можно использовать как альтернативу словарей везде, где требуется хранить неизменяемый набор именованных значений. Перечисления занимают меньше оперативной памяти, чем словари, и быстрее обрабатываются.

> ***Примечание***
>
> В составе стандартной библиотеки Python присутствует модуль `struct`, позволяющий создавать нечто похожее на перечисления. Однако он не столь удобен в работе, как инструменты, предлагаемые модулем `enum`.

ГЛАВА 16

Работа с файлами и каталогами

16.1. Открытие файлов

Для открытия файла применяется функция `open()`, имеющая следующий формат вызова:

```
open(<Путь к файлу>[, mode='r'][, buffering=-1][, encoding=None][,
     errors=None][, newline=None][, closefd=True])
```

Задаваемый путь к открываемому файлу может быть абсолютным или относительным. Для разделения сегментов путей можно применять как обратные, так и прямые слеши (подробнее указание путей рассмотрено в *разд. 16.1.1*).

Параметр `mode` указывает *режим открытия файла* в виде строки (чтение, запись или чтение и запись), содержащей одно из следующих обозначений:

- `r` — чтение. Если файл не существует, генерируется исключение `FileNotFoundError`;

  Python помечает место, с которого начнется чтение из файла или запись в него, особым *указателем*. После открытия файла только на чтение этот указатель устанавливается на начало файла;
- `r+` — чтение и запись. Указатель устанавливается на начало файла. Если файл не существует, генерируется исключение `FileNotFoundError`;
- `w` — запись. Указатель устанавливается на начало файла. Если файл не существует, он будет создан, если существует — перезаписан;
- `w+` — чтение и запись. Указатель устанавливается на начало файла. Если файл не существует, он будет создан, если существует — перезаписан;
- `a` — дозапись. Указатель устанавливается на конец файла. Если файл не существует, он будет создан, если существует, данные будут добавлены в него;
- `a+` — чтение и дозапись. Указатель устанавливается на конец файла. Если файл не существует, он будет создан, если существует, данные будут добавлены в него;
- `x` — создание файла для записи. Если файл уже существует, генерируется исключение `FileExistsError`;
- `x+` — создание файла для чтения и записи. Если файл уже существует, генерируется исключение `FileExistsError`.

После собственно обозначения режима может следовать обозначение *подрежима открытия файла*, задающее формат читаемых или записываемых данных:

- `b` — *двоичный* (*бинарный*) подрежим. Читаемые и записываемые данные представляются объектами типа `bytes`;
- `t` — *текстовый* подрежим (используется по умолчанию в Windows). Читаемые и записываемые данные представляются строками.

В текстовом подрежиме автоматически выполняется обработка символов конца строки — так, в Windows при чтении вместо символов `\r\n` будет подставлен символ `\n`. Для примера создадим файл file.txt и запишем в него две строки:

```
>>> f = open(r"c:/book/file.txt", "w")  # Открываем файл на запись
>>> f.write("String1\nString2") # Записываем две строки в файл
    15
>>> f.close()                   # Закрываем файл
```

Поскольку мы указали режим `w`, то, если файл не существует, он будет создан, а если существует — перезаписан.

Теперь выведем содержимое файла в двоичном и текстовом подрежимах:

```
>>> # Двоичный подрежим (символ \r остается)
>>> with open(r"c:/book/file.txt", "rb") as f:
...     for line in f:
...         print(repr(line))
...
    b'String1\r\n'
    b'String2'
>>> # Текстовый подрежим (символ \r удаляется)
>>> with open(r"c:/book/file.txt", "r") as f:
...     for line in f:
...         print(repr(line))
...
    'String1\n'
    'String2'
```

Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только в момент закрытия файла или после вызова функции или метода `flush()`. В необязательном параметре `buffering` можно указать размер буфера. Если указано значение 0, то данные будут сразу записываться в файл (допустимо только в двоичном подрежиме). Значение 1 используется при построчной записи в файл (допустимо только в текстовом подрежиме). Другое положительное число задает примерный размер буфера, а отрицательное значение (или отсутствие значения) означает установку размера, применяемого в системе по умолчанию. По умолчанию текстовые файлы буферизуются построчно, а двоичные — частями, размер которых интерпретатор выбирает самостоятельно в диапазоне от 4096 до 8192 байтов.

При чтении файла в текстовом подрежиме производится попытка преобразовать данные в кодировку Unicode, а при записи выполняется обратная операция — строка преобразуется в последовательность байтов в определенной кодировке. По умолчанию назначается кодировка, применяемая в системе. Если преобразование невозможно, генерируется исключение. Указать кодировку, которая будет использоваться при записи и чтении файла, позволяет параметр `encoding`. Для примера запишем данные в кодировке UTF-8:

```
>>> f = open(r"c:/book/file.txt", "w", encoding="utf-8")
>>> f.write("Строка")  # Записываем строку в файл
    6
>>> f.close()          # Закрываем файл
```

При открытии этого файла для чтения следует явно указать кодировку:

```
>>> with open(r"c:/book/file.txt", "r", encoding="utf-8") as f:
...     for line in f:
...         print(line)
...
    Строка
```

При работе с текстовыми файлами в кодировках UTF-8, UTF-16 и UTF-32 следует учитывать, что в начале файла может присутствовать *метка порядка байтов* (BOM). Для кодировки UTF-8 эта метка является необязательной, и в предыдущем примере она не была добавлена в файл при записи. Чтобы BOM были добавлена, в параметре `encoding` следует указать значение `utf-8-sig`. Запишем строку в файл в кодировке UTF-8 с BOM:

```
>>> f = open(r"c:/book/file.txt", "w", encoding="utf-8-sig")
>>> f.write("Строка")  # Записываем строку в файл
    6
>>> f.close()          # Закрываем файл
```

Теперь прочитаем файл с разными значениями параметра `encoding`:

```
>>> with open(r"c:/book/file.txt", "r", encoding="utf-8") as f:
...     for line in f:
...         print(repr(line))
...
    '\ufeffСтрока'
>>> with open(r"c:/book/file.txt", "r", encoding="utf-8-sig") as f:
...     for line in f:
...         print(repr(line))
...
    'Строка'
```

Если неизвестно, есть ли BOM в файле, и необходимо получить данные без нее, то при чтении файла в кодировке UTF-8 всегда следует указывать кодировку `utf-8-sig`.

При указании кодировок `utf-16` и `utf-32` в параметре `encoding` обработка BOM производится автоматически: при записи BOM автоматически вставляется в начало файла, а при чтении она игнорируется. Запишем строку в файл, а затем прочитаем ее из файла:

```
>>> with open(r"c:/book/file.txt", "w", encoding="utf-16") as f:
...     f.write("Строка")
...
    6
>>> with open(r"c:/book/file.txt", "r", encoding="utf-16") as f:
...     for line in f:
...         print(repr(line))
...
    'Строка'
```

При указании кодировок `utf-16-le`, `utf-16-be`, `utf-32-le` и `utf-32-be` BOM необходимо добавить в начало файла самостоятельно, а при чтении удалить ее.

В параметре errors можно указать уровень обработки ошибок. Возможные значения: "strict" (при ошибке генерируется исключение ValueError — значение по умолчанию), "replace" (неизвестный символ заменяется знаком вопроса или символом с кодом \ufffd), "ignore" (неизвестные символы игнорируются), "xmlcharrefreplace" (неизвестный символ заменяется последовательностью &#xxxx;) и "backslashreplace" (неизвестный символ заменяется последовательностью \uxxxx).

Параметр newline задает режим обработки символов конца строки. Поддерживаемые им значения таковы:

- None (значение по умолчанию) — выполняется стандартная обработка символов конца строки. Например, в Windows при чтении символы \r\n преобразуются в символ \n, а при записи производится обратное преобразование;
- "" (пустая строка) — обработка не выполняется;
- "<Специальный символ>" — для обозначения конца строки используется заданный специальный символ, в качестве которого можно указать \r\n, \r и \n.

16.1.1. Указание путей к файлам и каталогам

В путях к файлам и каталогам для разделения отдельных *сегментов* (обозначений дисков, имен каталогов и файлов) можно применять как обратные, так и прямые слеши.

Символ-разделитель сегментов путей, применяемый в текущей операционной системе, хранится в переменной sep из модуля os.path. Выведем разделитель, применяемый в Windows:

```
>>> from os.path import sep
>>> sep                          # Выведет обратный слеш
    '\\'
```

При использовании обратных слешей следует помнить, что в Python они являются специальными символами (подробности — в *разд. 6.1.1*). По этой причине обратные слеши следует заменять спецсимволами \\ или вместо обычных строк использовать необрабатываемые (описаны в *разд. 6.1.1*). Примеры:

```
>>> "C:\\temp\\new\\file.txt"     # Правильно
    'C:\\temp\\new\\file.txt'
>>> r"C:\temp\new\file.txt"       # Правильно
    'C:\\temp\\new\\file.txt'
>>> "C:\temp\new\file.txt"        # Неправильно!!!
    'C:\temp\new\x0cile.txt'
```

В последнем примере интерпретатор принял последовательность символов \f за специальный символ, что привело к искажению заданного нами пути. Если такой искаженный путь передать функции open(), возникнет исключение OSError.

Кроме того, слеш, расположенный в конце строки, необходимо удваивать даже при использовании неформатированных строк:

```
>>> r"C:\temp\new\"               # Неправильно!!!
    SyntaxError: EOL while scanning string literal
>>> r"C:\temp\new\\"
    'C:\\temp\\new\\\\'
```

В первом примере последний слеш экранирует закрывающую кавычку, что приводит к синтаксической ошибке. Решить эту проблему можно, удвоив последний слеш. Однако тогда два слеша превратятся в четыре (см. второй пример). Так что в этой ситуации лучше использовать обычные строки, например:

```
>>> "C:\\temp\\new\\"            # Правильно
    'C:\\temp\\new\\'
>>> r"C:\temp\new\\"[:-1]        # Можно и удалить слеш
    'C:\\temp\\new\\'
```

Вместо *абсолютного* пути (отсчитываемого от обозначения диска) можно указать *относительный*, который отсчитывается от текущего рабочего каталога (о нем — позже). Возможны следующие варианты:

- если открываемый файл находится в текущем рабочем каталоге, можно указать только имя файла:

```
>>> open(r"file1.txt")
```

- если открываемый файл расположен в каталоге, вложенном в текущий рабочий каталог, перед именем файла через слеш указываются имена каталогов, в которые последовательно вложен файл:

```
>>> open(r"folder1/file2.txt")
>>> open(r"folder2/folder3/file3.txt")
```

- если каталог с файлом расположен ниже уровнем, перед именем файла указываются две точки и слеш (`"../"`):

```
>>> open(r"../file4.txt")
```

- если в начале пути расположен слеш, путь отсчитывается от корня текущего диска. В этом случае местоположение текущего рабочего каталога не имеет значения:

```
>>> open(r"/book/folder4/file4.txt")
>>> open(r"/book/folder5/folder6/file5.txt")
```

16.1.2. Текущий рабочий каталог

Текущий рабочий каталог — это каталог, от которого интерпретатор отсчитывает относительные пути файлов. По умолчанию текущим рабочим является следующий каталог:

- при запуске интерпретатора в интерактивном режиме — каталог, в котором установлен Python (например, C:\Python310);
- при запуске Python-файла щелчком мыши — каталог, в котором хранится запущенный файл;
- при запуске Python-файла из консоли — каталог, из которого был запущен файл. Например, если запустить консоль, выполнить в ней переход в каталог d:\work и запустить Python-файл c:\book\program.py, текущим рабочим станет каталог d:\work.

Получить путь к текущему рабочему каталогу можно вызовом функции `getcwd()` из модуля `os`:

```
>>> import os
>>> os.getcwd()
    'C:\\Python310'
```

Задать в качестве текущего рабочего каталог с указанным путем можно вызовом функции chdir(<Путь>) из модуля os:

```
>>> os.chdir(r"c:\book")
```

Указать в качестве текущего рабочего каталог, в котором хранится запущенный Python-файл, можно, вставив в код этого файла следующие инструкции:

```
import os
os.chdir(os.path.dirname(os.path.abspath(__file__)))
```

Переменная __file__ хранит в разных случаях путь к запущенному файлу или только имя файла без пути. Чтобы гарантированно получить полный путь к файлу, следует передать значение переменной в функцию abspath() из модуля os.path. Функция dirname() извлекает из переданного ей полного пути к файлу путь к хранящему его каталогу, который передается функции chdir().

16.2. Чтение и запись данных: объектные инструменты

Функция open() возвращает объект, представляющий открытый файл. Рассмотрим основные методы, поддерживаемые этим объектом:

- close() — закрывает текущий файл.

  При удалении из памяти объекта, представляющего файл, закрытие файла выполняется автоматически, поэтому в небольших программах файлы можно не закрывать явно. Тем не менее явное закрытие файла является признаком хорошего стиля программирования. Кроме того, при наличии незакрытого файла генерируется предупреждающее сообщение: "ResourceWarning: unclosed file".

  Классы объектов-файлов являются менеджерами контекста, поэтому для работы с файлами можно использовать обработчики контекста (см. *разд. 14.2*):

  ```
  with open(r"c:\book\file.txt", "w", encoding="cp1251") as f:
      f.write("Строка") # Записываем строку в файл
  # Здесь файл уже закрыт автоматически
  ```

- write(<Значение>) — записывает в текущий файл указанное значение. Если файл открыт в текстовом подрежиме, значение должно быть задано в виде строки, если в двоичном подрежиме — в виде последовательности байтов. Метод возвращает количество записанных символов или байтов. Пример:

  ```
  >>> # Текстовый подрежим
  >>> f = open(r"c:\book\file.txt", "w", encoding="cp1251")
  >>> f.write("Строка1\nСтрока2")
      15
  >>> f.close()
  >>> # Двоичный подрежим
  >>> f = open(r"c:\book\file.txt", "wb")
  >>> f.write(bytes("Строка1\nСтрока2", "cp1251"))
      15
  >>> f.write(bytearray("\nСтрока3", "cp1251"))
      8
  >>> f.close()
  ```

- `writelines(<Последовательность>)` — записывает в текущий файл заданную последовательность. Если файл открыт в текстовом подрежиме, элементы последовательности должны представлять собой строки, если в двоичном подрежиме — последовательности байтов. Пример:

```
>>> # Текстовый подрежим
>>> f = open(r"c:\book\file.txt", "w", encoding="cp1251")
>>> f.writelines(["Строка1\n", "Строка2"])
>>> f.close()
>>> # Двоичный подрежим
>>> f = open(r"c:\book\file.txt", "wb")
>>> arr = [bytes("Строка1\n", "cp1251"), bytes("Строка2", "cp1251")]
>>> f.writelines(arr)
>>> f.close()
```

- `writable()` — возвращает `True`, если в текущий файл можно писать, и `False` — в противном случае:

```
>>> f = open(r"c:\book\file.txt", "r")        # Открываем файл для чтения
>>> f.writable()
    False
>>> f = open(r"c:\book\file.txt", "w")        # Открываем файл для записи
>>> f.writable()
    True
```

- `read([<Количество>])` — считывает данные из текущего файла и возвращает их. Если файл открыт в текстовом подрежиме, выдается строка, а если в двоичном — последовательность байтов. Если параметр не указан, возвращается содержимое файла от текущей позиции указателя до конца файла. Пример:

```
>>> # Текстовый подрежим
>>> with open(r"c:\book\file.txt", "r", encoding="cp1251") as f:
...     f.read()
...
    'Строка1\nСтрока2'
>>> # Двоичный подрежим
>>> with open(r"c:\book\file.txt", "rb") as f:
...     f.read()
...
    b'\xd1\xf2\xf0\xee\xea\xe01\n\xd1\xf2\xf0\xee\xea\xe02'
```

В параметре можно указать количество символов или байтов, которое требуется прочитать из файла. Когда достигается конец файла, метод возвращает пустую строку или пустую последовательность байтов. Пример:

```
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> f.read(8)                # Считываем 8 символов
    'Строка1\n'
>>> f.read(8)                # Считываем 8 символов
    'Строка2'
>>> f.read(8)                # Достигнут конец файла
    ''
>>> f.close()
```

- `readline([<Количество>])` — считывает из текущего файла очередную строку и возвращает ее. Если файл открыт в текстовом подрежиме, выдается строка, а если в двоичном — последовательность байтов. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка — если она не завершается символом перевода строки, то таковой добавлен не будет. При достижении конца файла возвращается пустая строка или последовательность байтов. Пример:

```
>>> # Текстовый подрежим
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> f.readline(), f.readline()
    ('Строка1\n', 'Строка2')
>>> f.readline()            # Достигнут конец файла
    ''
>>> f.close()
>>> # Двоичный подрежим
>>> f = open(r"c:\book\file.txt", "rb")
>>> f.readline(), f.readline()
    (b'\xd1\xf2\xf0\xee\xea\xe01\n', b'\xd1\xf2\xf0\xee\xea\xe02')
>>> f.readline()            # Достигнут конец файла
    b''
>>> f.close()
```

В параметре можно указать количество символов или байтов, которое следует прочитать из файла. В этом случае считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), символ конца файла или из файла не будет прочитано указанное количество символов или байтов. Пример:

```
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> f.readline(2), f.readline(2)
    ('Ст', 'ро')
>>> f.readline(100) # Возвращаются остатки данных
    'ка1\n'
>>> f.close()
```

- `readlines([<Количество>])` — считывает из текущего файла все строки и сводит их в список, который и возвращает. Если файл открыт в текстовом подрежиме, возвращается список строк, а если в двоичном — список объектов типа `bytes`. Каждая строка в списке будет содержать символ перевода строки. Исключением является последняя строка — если она не завершается символом перевода строки, таковой добавлен не будет. Примеры:

```
>>> # Текстовый подрежим
>>> with open(r"c:\book\file.txt", "r", encoding="cp1251") as f:
...     f.readlines()
...
    ['Строка1\n', 'Строка2']
>>> # Двоичный подрежим
>>> with open(r"c:\book\file.txt", "rb") as f:
...     f.readlines()
...
    [b'\xd1\xf2\xf0\xee\xea\xe01\n', b'\xd1\xf2\xf0\xee\xea\xe02']
```

В параметре можно указать количество символов или байтов, которое должно быть считано из файла. Если указанное количество приходится на середину какой-либо строки, эта строка считывается полностью. Если в файле хранится меньше символов (байтов), чем указано, будет считано все содержимое файла. Примеры:

```
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> f.readlines(4)
    ['Строка1\n']
>>> f.close()
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> f.readlines(10)
    ['Строка1\n', 'Строка2']
>>> f.close()
```

- `__next__()` — при каждом вызове считывает из текущего файла одну строку и возвращает ее. Если файл открыт в текстовом подрежиме, возвращается строка, а если в двоичном — последовательность байтов. При достижении конца файла генерируется исключение `StopIteration`. Пример:

```
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> f.__next__(), f.__next__()
    ('Строка1\n', 'Строка2')
>>> f.__next__() # Достигнут конец файла
    Traceback (most recent call last):
      File "<pyshell#26>", line 1, in <module>
        f.__next__() # Достигнут конец файла
    StopIteration
>>> f.close()
```

Содержимое файла можно перебрать построчно в цикле перебора последовательности, поскольку этот цикл на каждой итерации автоматически вызывает метод `__next__()`. Для примера выведем все строки, предварительно удалив символ перевода строки:

```
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> for line in f: print(line.rstrip("\n"), end=" ")
...
    Строка1 Строка2
>>> f.close()
```

- `readable()` — возвращает `True`, если из текущего файла можно читать, и `False` — в противном случае:

```
>>> f = open(r"c:\book\file.txt", "w")        # Открываем файл для записи
>>> f.writable()
    False
>>> f = open(r"c:\book\file.txt", "r")        # Открываем файл для чтения
>>> f.writable()
    True
```

- `flush()` — принудительно записывает данные из буфера на диск;
- `fileno()` — возвращает целочисленный дескриптор текущего файла. Возвращаемое значение всегда будет больше числа 2, т. к. числа 0–2 закреплены за стандартными потоками ввода/вывода. Пример:

```
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> f.fileno()                    # Дескриптор файла
    3
>>> f.close()
```

- `truncate([<Количество>])` — обрезает или увеличивает (с заполнением свободного места нулями) текущий файл до указанного количества символов (если задан текстовый подрежим) или байтов (в случае двоичного подрежима). Метод возвращает новый размер файла. Пример:

```
>>> f = open(r"c:\book\file.txt", "r+", encoding="cp1251")
>>> f.read()
    'Строка1\nСтрока2'
>>> f.truncate(5)
    5
>>> f.close()
>>> with open(r"c:\book\file.txt", "r", encoding="cp1251") as f:
...     f.read()
...
    'Строк'
```

- `tell()` — возвращает позицию указателя относительно начала текущего файла в виде целого числа. В Windows при открытии файла в текстовом подрежиме метод считает символ `\r` как дополнительный байт, хотя этот символ удаляется из считанных строк. Пример:

```
>>> with open(r"c:\book\file.txt", "w", encoding="cp1251") as f:
...     f.write("String1\nString2")
...
    15
>>> f = open(r"c:\book\file.txt", "r", encoding="cp1251")
>>> f.tell()      # Указатель расположен в начале файла
    0
>>> f.readline()  # Перемещаем указатель
    'String1\n'
>>> f.tell()      # Возвращает 9 (8 + '\r'), а не 8!!!
    9
>>> f.close()
```

- `seek(<Смещение>[, <Позиция>])` — устанавливает указатель текущего файла в позицию, имеющую заданное `<Смещение>` относительно параметра `<Позиция>`. В качестве параметра `<Позиция>` могут быть указаны следующие атрибуты из модуля `io` или соответствующие им значения:
  - `io.SEEK_SET` или 0 — начало файла (значение по умолчанию);
  - `io.SEEK_CUR` или 1 — текущая позиция указателя. Положительное значение смещения вызывает перемещение к концу файла, отрицательное — к его началу;
  - `io.SEEK_END` или 2 — конец файла.

  Примеры:

```
>>> import io
>>> f = open(r"c:\book\file.txt", "rb")
```

```
>>> f.seek(9, io.SEEK_CUR) # 9 байтов от указателя
    9
>>> f.tell()
    9
>>> f.seek(0, io.SEEK_SET) # Перемещаем указатель в начало
    0
>>> f.tell()
    0
>>> f.seek(-9, io.SEEK_END) # -9 байтов от конца файла
    7
>>> f.tell()
    7
>>> f.close()
```

- `seekable()` — возвращает `True`, если указатель текущего файла можно переместить в другую позицию, и `False` — в противном случае:

```
>>> f = open(r"c:\book\file.txt", "r")
>>> f.seekable()
    True
```

Объекты файлов поддерживают следующие атрибуты:

- `name` — имя текущего файла;
- `mode` — режим, в котором был открыт текущий файл;
- `closed` — хранит `True`, если текущий файл уже закрыт, и `False` — в противном случае:

```
>>> f = open(r"c:\book\file.txt", "r+b")
>>> f.name, f.mode, f.closed
    ('file.txt', 'rb+', False)
>>> f.close()
>>> f.closed
    True
```

- `encoding` — название кодировки, в которой сохранен текущий файл. Доступен только в текстовом подрежиме. Пример:

```
>>> f = open(r"c:\book\file.txt", "a", encoding="cp1251")
>>> f.encoding
    'cp1251'
>>> f.close()
```

Стандартный поток вывода `stdout` также является файловым объектом. Атрибут `encoding` этого объекта всегда содержит кодировку устройства вывода, поэтому строка преобразуется в последовательность байтов в правильной кодировке. Например, при запуске программы двойным щелчком мыши атрибут `encoding` будет иметь значение `"cp866"`, а при запуске в окне IDLE Shell — значение `"utf-8"` (в версиях Python, предшествовавших 3.7, атрибут хранил значение `"cp1251"`). Пример:

```
>>> import sys
>>> sys.stdout.encoding
    'utf-8'
```

- `buffer` — хранит ссылку на объект буфера, в котором временно хранятся данные, записываемые в текущий файл. Доступен только в текстовом подрежиме. Посредством этого объекта можно записывать в файл последовательности байтов, например:

```
>>> f = open(r"c:\book\file.txt", "w", encoding="cp1251")
>>> f.buffer.write(bytes("Строка", "cp1251"))
    6
>>> f.close()
```

16.3. Чтение и запись данных: низкоуровневые инструменты

Python предоставляет ряд низкоуровневых функций, дающих больший контроль над читаемыми и записываемыми данными и реализованных в модуле `os`:

- `open(<Путь к файлу>, <Режим>[, mode=0o777])` — открывает файл с заданным путем и возвращает целочисленный *дескриптор*, с помощью которого производится дальнейшая работа с этим файлом. Если файл открыть не удалось, генерируется исключение `OSError` или одно из исключений, являющихся его подклассами (описаны в *разд. 16.11*). В параметре `<Режим>` в операционной системе Windows могут быть указаны следующие флаги (или их комбинация через символ `|`):
  - `os.O_RDONLY` — чтение;
  - `os.O_WRONLY` — запись;
  - `os.O_RDWR` — чтение и запись;
  - `os.O_APPEND` — дозапись;
  - `os.O_CREAT` — создать файл, если он не существует и если не указан флаг `os.O_EXCL`;
  - `os.O_EXCL` — при использовании совместно с `os.O_CREAT` указывает, что создаваемый файл изначально не должен существовать, в противном случае будет сгенерировано исключение `FileExistsError`;
  - `os.O_TEMPORARY` — при использовании совместно с `os.O_CREAT` указывает создать временный файл, который будет автоматически удален сразу после закрытия;
  - `os.O_SHORT_LIVED` — то же самое, что `os.O_TEMPORARY`, но созданный файл по возможности будет храниться в оперативной памяти;
  - `os.O_TRUNC` — очистить содержимое открытого файла;
  - `os.O_BINARY` — файл будет открыт в двоичном подрежиме;
  - `os.O_TEXT` — файл будет открыт в текстовом подрежиме (в Windows файлы открываются в текстовом подрежиме по умолчанию).

  Параметр `mode` задает права доступа к созданному файлу.

  Несколько примеров:

  - открытие файла на запись и запись в него одной строки. Если файл не существует, он будет создан, если существует — очищен:

```
>>> import os                  # Подключаем модуль
>>> mode = os.O_WRONLY | os.O_CREAT | os.O_TRUNC
>>> f = os.open(r"c:\book\file.txt", mode)
>>> os.write(f, b"String1\n") # Записываем данные
    8
>>> os.close(f)               # Закрываем файл
```

- дозапись еще одной строки в конец файла:

```
>>> mode = os.O_WRONLY | os.O_CREAT | os.O_APPEND
>>> f = os.open(r"c:\book\file.txt", mode)
>>> os.write(f, b"String2\n")
    8
>>> os.close(f)
```

- чтение файла в текстовом подрежиме:

```
>>> f = os.open(r"c:\book\file.txt", os.O_RDONLY)
>>> os.read(f, 50)            # Читаем 50 байтов
    b'String1\nString2\n'
>>> os.close(f)
```

- чтение файла в двоичном подрежиме:

```
>>> f = os.open(r"c:\book\file.txt", os.O_RDONLY | os.O_BINARY)
>>> os.read(f, 50)            # Читаем 50 байтов
    b'String1\r\nString2\r\n'
>>> os.close(f)
```

♦ `read(<Дескриптор>, <Количество байтов>)` — читает из файла с заданным дескриптором указанное количество байтов. При достижении конца файла возвращается пустая строка. Пример:

```
>>> f = os.open(r"c:\book\file.txt", os.O_RDONLY)
>>> os.read(f, 5), os.read(f, 5), os.read(f, 5), os.read(f, 5)
    (b'Strin', b'g1\nS', b'tring', b'2\n')
>>> os.read(f, 5)             # Достигнут конец файла
    b''
>>> os.close(f)
```

♦ `write(<Дескриптор>, <Последовательность байтов>)` — записывает заданную последовательность байтов в файл с указанным дескриптором. Возвращает количество записанных байтов;

♦ `close(<Дескриптор>)` — закрывает файл с заданным дескриптором;

♦ `lseek(<Дескриптор>, <Смещение>, <Позиция>)` — устанавливает указатель файла с заданным дескриптором в позицию, имеющую заданное `<Смещение>` относительно параметра `<Позиция>`. Возвращает новую позицию указателя. В качестве параметра `<Позиция>` могут быть указаны следующие атрибуты или соответствующие им значения:

- `os.SEEK_SET` или 0 — начало файла;
- `os.SEEK_CUR` или 1 — текущая позиция указателя;
- `os.SEEK_END` или 2 — конец файла.

Примеры:

```
>>> f = os.open(r"c:\book\file.txt", os.O_RDONLY | os.O_BINARY)
>>> os.lseek(f, 0, os.SEEK_END) # Перемещение в конец файла
    18
>>> os.lseek(f, 0, os.SEEK_SET) # Перемещение в начало файла
    0
```

```
>>> os.lseek(f, 9, os.SEEK_CUR) # Относительно указателя
    9
>>> os.lseek(f, 0, os.SEEK_CUR) # Текущее положение указателя
    9
>>> os.close(f)
```

- `dup(<Дескриптор>)` — возвращает дубликат заданного файлового дескриптора;
- `fdopen(<Дескриптор>[, <Режим>[, <Размер буфера>]])` — возвращает файловый объект по указанному дескриптору. Параметры `<Режим>` и `<Размер буфера>` имеют тот же смысл, что и в функции `open()` (описана в *разд. 16.1*). Пример:

```
>>> fd = os.open(r"c:\book\file.txt", os.O_RDONLY)
>>> fd
    3
>>> f = os.fdopen(fd, "r")
>>> f.fileno() # Объект имеет тот же дескриптор
    3
>>> f.read()
    'String1\nString2\n'
>>> f.close()
```

16.4. Файлы в памяти

Python позволяет создавать файлы, хранящиеся в оперативной памяти. Такие файлы можно открывать в текстовом или двоичном подрежиме, записывать и читать их данные, перемещать их указатели.

Для создания файла в памяти и открытия его в текстовом подрежиме служит класс `StringIO` из модуля `io`. Формат конструктора класса:

```
StringIO([<Начальное значение>][, ][newline='\n'])
```

Если первый параметр не указан, то начальным значением будет пустая строка. Параметр `newline` задает символ, используемый для обозначения концов строк. После создания объекта указатель текущей позиции устанавливается на начало файла.

Объект текстового файла в памяти поддерживает следующие методы:

- `close()` — закрывает текущий файл. Проверить, открыт файл или закрыт, позволяет атрибут `closed`, который хранит `True`, если файл закрыт, и `False` — в противном случае;
- `getvalue()` — возвращает содержимое текущего файла в виде строки:

```
>>> import io                # Подключаем модуль
>>> f = io.StringIO("String1\n")
>>> f.getvalue()             # Получаем содержимое файла
    'String1\n'
>>> f.close()                # Закрываем файл
```

- `tell()` — возвращает текущую позицию указателя относительно начала текущего файла;
- `seek(<Смещение>[, <Позиция>])` — устанавливает указатель текущего файла в позицию, имеющую заданное `<Смещение>` относительно параметра `<Позиция>`. В качестве параметра `<Позиция>` могут быть указаны следующие значения:

- 0 — начало файла (значение по умолчанию);
- 1 — текущая позиция указателя;
- 2 — конец файла.

Примеры использования методов `seek()` и `tell()`:

```
>>> f = io.StringIO("String1\n")
>>> f.tell()        # Позиция указателя
    0
>>> f.seek(0, 2)    # Перемещаем указатель в конец файла
    8
>>> f.tell()        # Позиция указателя
    8
>>> f.seek(0)       # Перемещаем указатель в начало файла
    0
>>> f.tell()        # Позиция указателя
    0
>>> f.close()
```

- `write(<Строка>)` — записывает заданную строку в текущий файл:

```
>>> f = io.StringIO("String1\n")
>>> f.seek(0, 2)         # Перемещаем указатель в конец файла
    8
>>> f.write("String2\n") # Записываем строку в файл
    8
>>> f.getvalue()
    'String1\nString2\n'
>>> f.close()
```

- `writelines(<Последовательность>)` — записывает заданную последовательность строк в текущий файл:

```
>>> f = io.StringIO()
>>> f.writelines(["String1\n", "String2\n"])
>>> f.getvalue()
    'String1\nString2\n'
>>> f.close()
```

- `read([<Количество символов>])` — считывает и возвращает данные из текущего файла. За каждый вызов будет возвращаться указанное в параметре количество символов. Если параметр не задан, возвращается содержимое файла от текущей позиции указателя до конца файла. Когда достигается конец файла, метод возвращает пустую строку. Пример:

```
>>> f = io.StringIO("String1\nString2\n")
>>> f.read()
    'String1\nString2\n'
>>> f.seek(0) # Перемещаем указатель в начало «файла»
    0
>>> f.read(5), f.read(5), f.read(5), f.read(5), f.read(5)
    ('Strin', 'g1\nSt', 'ring2', '\n', '')
>>> f.close()
```

- `readline([<Количество символов>])` — считывает из текущего файла очередную строку и возвращает ее. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка — если она не завершается символом перевода, таковой добавлен не будет. При достижении конца файла возвращается пустая строка. Пример:

```
>>> f = io.StringIO("String1\nString2")
>>> f.readline(), f.readline(), f.readline()
    ('String1\n', 'String2', '')
>>> f.close()
```

  В параметре можно указать количество символов, которое требуется считать. В этом случае считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), конец файла или из файла не будет прочитано указанное количество символов. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3\n")
>>> f.readline(5), f.readline(5)
    ('Strin', 'g1\n')
>>> f.readline(100) # Возвращается одна строка, а не 100 символов
    'String2\n'
>>> f.close()
```

- `readlines([<Количество символов>])` — считывает из текущего файла все строки и сводит их в список, который и возвращает. Каждая строка в списке будет содержать символ перевода строки. Исключением станет последняя строка — если она не завершается символом перевода, таковой добавлен не будет. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.readlines()
   ['String1\n', 'String2\n', 'String3']
>>> f.close() # Закрываем «файл»
```

  В параметре можно указать количество символов, которое следует прочитать. В этом случае будет считано указанное количество символов плюс фрагмент до символа конца строки `\n`. Затем эта строка разбивается и добавляется построчно в список. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.readlines(14)
    ['String1\n', 'String2\n']
>>> f.seek(0) # Перемещаем указатель в начало «файла»
    0
>>> f.readlines(17)
    ['String1\n', 'String2\n', 'String3']
>>> f.close() # Закрываем «файл»
```

- `__next__()` — при каждом вызове считывает из текущего файла одну строку и возвращает ее. При достижении конца файла генерируется исключение `StopIteration`. Пример:

```
>>> f = io.StringIO("String1\nString2")
>>> f.__next__(), f.__next__()
    ('String1\n', 'String2')
>>> f.__next__()
    ... Фрагмент опущен ...
    StopIteration
>>> f.close()
```

Содержимое файла можно перебрать построчно в цикле перебора последовательности, поскольку этот цикл на каждой итерации автоматически вызывает метод `__next__()`:

```
>>> f = io.StringIO("String1\nString2")
>>> for line in f: print(line.rstrip())
...
String1
String2
>>> f.close()
```

- `flush()` — принудительно записывает данные из буфера в текущий файл;
- `truncate([<Количество символов>])` — обрезает или увеличивает (с заполнением свободного места нулями) текущий файл до указанного количества символов:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.truncate(15)      # Обрезаем «файл»
15
>>> f.getvalue()        # Получаем содержимое «файла»
'String1\nString2'
>>> f.close()
```

Если параметр не указан, то файл обрезается до текущей позиции указателя:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.seek(15)          # Перемещаем указатель
15
>>> f.truncate()        # Обрезаем файл до указателя
15
>>> f.getvalue()
'String1\nString2'
>>> f.close()
```

Описанные в *разд. 16.2* методы `writable()`, `readable()` и `seekable()`, вызванные у объекта класса `StringIO`, всегда возвращают `True`.

Для работы с файлами в памяти в двоичном подрежиме применяется класс `BytesIO` из модуля `io`, аналогичный классу `StringIO`. Формат конструктора класса `BytesIO`:

```
BytesIO([<Начальное значение>])
```

Класс `BytesIO` поддерживает те же методы, что и `StringIO`, но в качестве параметров они принимают и возвращают последовательности байтов. Рассмотрим основные операции на примерах:

```
>>> import io
>>> f = io.BytesIO(b"String1\n")
>>> f.seek(0, 2)          # Перемещаем указатель в конец файла
8
>>> f.write(b"String2\n") # Пишем в файл
8
>>> f.getvalue()          # Получаем содержимое файла
b'String1\nString2\n'
>>> f.seek(0)             # Перемещаем указатель в начало файла
0
```

```
>>> f.read()              # Считываем данные
    b'String1\nString2\n'
>>> f.close()             # Закрываем файл
```

Класс `BytesIO` также поддерживает метод `getbuffer()`, который возвращает ссылку на объект `memoryview`. С помощью этого объекта можно получать и изменять данные по индексу (только целые числа) или срезу (только последовательности байтов), преобразовывать данные в список целых чисел (с помощью метода `tolist()`) или в последовательность байтов (с помощью метода `tobytes()`). Примеры:

```
>>> f = io.BytesIO(b"Python")
>>> buf = f.getbuffer()
>>> buf[0]                # Получаем значение по индексу
    80
>>> buf[0] = 81           # Изменяем значение по индексу
>>> Результат извлечения среза также будет объектом memoryview
>>> buf[1:3]
    <memory at 0x000001C48349BAC0>
>>> bytes(buf[1:3])
    b'yt'
>>> buf[1:3] = b"us"      # Изменяем значение по срезу
>>> f.getvalue()
    b'Qushon'
>>> f.getvalue()          # Получаем содержимое
    b'Qushon'
>>> buf.tolist()          # Преобразуем в список
    [81, 117, 115, 104, 111, 110]
>>> buf.tobytes()         # Преобразуем в последовательность байтов
    b'Qushon'
```

16.5. Задание прав доступа к файлам и каталогам

В операционных системах семейства UNIX каждому объекту (файлу или каталогу) назначаются права доступа, предоставляемые той или иной разновидности пользователей: владельцу, группе, которой принадлежит владелец, и прочим. Могут быть назначены отдельные права на чтение, запись и выполнение.

Права доступа обозначаются буквами:

- `r` — файл можно читать, а содержимое каталога — просматривать;
- `w` — файл можно модифицировать, удалять и переименовывать, в каталоге разрешается создавать и удалять файлы, сам каталог можно переименовать или удалить;
- `x` — файл можно выполнять, в каталоге разрешается выполнять операции над файлами, в том числе производить в нем поиск файлов.

Права доступа к файлу определяются записью типа:

```
-rw-r--r--
```

Первый символ (`-`) означает, что это файл, и не задает никаких прав доступа. Далее три символа (`rw-`) задают права доступа для владельца: чтение и запись, символ (`-`) здесь означа-

ет, что права на выполнение нет. Следующие три символа задают права доступа для группы, в которую входит владелец, — здесь только чтение (`r--`). Последние три символа (`r--`) задают права для всех остальных пользователей — также только чтение.

Права доступа к каталогу определяются такой строкой:

```
drwxr-xr-x
```

Первая буква (`d`) означает, что это каталог. Владелец может выполнять в каталоге любые действия (`rwx`), а его группа и все остальные пользователи — только читать и выполнять поиск (`r-x`). Для того чтобы каталог можно было просматривать, должны быть установлены права на выполнение (`x`).

Права доступа могут обозначаться и числом (*маской прав доступа*). Число состоит из трех цифр: от 0 до 7. Первая цифра задает права для владельца, вторая — для его группы, а третья — для всех остальных пользователей. Например, права доступа `-rw-r--r--` соответствуют числу 644. Сопоставим числам, входящим в маску прав доступа, двоичную и буквенную записи (табл. 16.1).

Таблица 16.1. *Права доступа в разных записях*

Восьмеричная цифра	Двоичная запись	Буквенная запись	Восьмеричная цифра	Двоичная запись	Буквенная запись
0	000	`---`	4	100	`r--`
1	001	`--x`	5	101	`r-x`
2	010	`-w-`	6	110	`rw-`
3	011	`-wx`	7	111	`rwx`

Согласно этой таблице права доступа `rw-r--r--` можно записать так: 110 100 100. То есть, если право предоставлено, в соответствующей позиции стоит 1, а если нет — то 0. Именно поэтому число 110 100 100 в двоичной записи соответствует числу 644 в восьмеричной.

Для проверки прав доступа к файлу или каталогу с указанным путем предназначена функция `access(<Путь>, <Режим>)` из модуля `os`. Она возвращает `True`, если проверка прошла успешно, или `False` — в противном случае. В параметре `<Режим>` могут быть указаны следующие константы, определяющие тип проверки:

- `os.F_OK` — проверка наличия файла или каталога:
- `os.R_OK` — проверка на возможность чтения файла или каталога;
- `os.W_OK` — проверка на возможность записи в файл или каталог;
- `os.X_OK` — определение, является ли файл или каталог выполняемым.

Примеры:

```
>>> import os
>>> os.access(r"c:\book\file.txt", os.F_OK) # Файл существует
    True
>>> os.access(r"C:\book", os.F_OK)          # Каталог существует
    True
>>> os.access(r"C:\book2", os.F_OK)         # Каталог не существует
    False
```

Задать другие права доступа к файлу или каталогу с указанным путем можно вызовом функции chmod(<Путь>, <Права доступа>) из модуля os. Права доступа задаются в виде восьмеричного числа. Пример:

```
>>> os.chmod(r"c:\book\file.txt", 0o777) # Полный доступ к файлу
```

Вместо числа можно указать комбинацию констант из модуля stat. За дополнительной информацией обращайтесь к документации по модулю.

16.6. Работа с файлами

Для копирования и перемещения файлов предназначены следующие функции из модуля shutil:

- copyfile() — создает копию исходного файла. Метаданные (например, дата последнего доступа) и права доступа не копируются. Возвращает путь к созданной копии файла. Формат вызова:

  ```
  copyfile(<Путь к исходному файлу>, <Путь к копии>[, follow_symlinks=True])
  ```

  Если оба заданных пути совпадают, генерируется исключение SameFileError из модуля shtil. Если файл-копия существует, он будет перезаписан. Если файл не удалось скопировать, генерируется исключение OSError или одно из исключений, являющихся производным от этого класса. Примеры:

  ```
  >>> import shutil      # Подключаем модуль
  >>> shutil.copyfile(r"c:\book\file.txt", r"c:\book\file2.txt")
      'c:\\book\\file2.txt'
  >>> # Путь не существует:
  >>> shutil.copyfile(r"c:\book\file.txt", r"C:\book2\file2.txt")
      ... Фрагмент опущен ...
      FileNotFoundError: [Errno 2] No such file or directory:
      'C:\\book2\\file2.txt'
  ```

  Исключение FileNotFoundError является производным от OSError и генерируется, если указанный файл не найден (классы исключений, генерируемых при файловых операциях, рассмотрены в *разд. 16.11*).

  Если в первом параметре указать символическую ссылку на файл, будет создана копия файла. Однако если при этом дать параметру follow_symlinks значение False, будет создана копия символической ссылки;

- copy() — создает копию исходного файл. Права доступа копируются, метаданные — нет. Возвращает путь к созданной копии файла. Формат вызова:

  ```
  copy(<Путь к исходному файлу>, <Путь к копии>[, follow_symlinks=True])
  ```

  В качестве второго параметра также можно указать путь к каталогу, в котором следует создать копию исходного файла, — тогда копия получит то же имя, что и исходный файл. Если файл-копия существует, он будет перезаписан. Метод генерирует те же исключения, что и метод copyfile(). Назначение параметра follow_symlinks такое же, как и у метода copyfile(). Примеры:

  ```
  >>> shutil.copy(r"c:\book\file.txt", r"c:\book\file3.txt")
      'c:\\book\\file3.txt'
  ```

```
>>> shutil.copy(r"c:\book\file.txt", r"d:\work")
    'd:\\work\\file.txt'
```

- `copy2()` — аналогичен `copy()`, только копирует еще и метаданные:

```
>>> shutil.copy2(r"c:\book\file.txt", r"c:\book\file4.txt")
    'c:\\book\\file4.txt'
```

- `move()` — перемещает исходный файл по местоположению целевого файла. Возвращает путь к целевому файлу. Формат вызова:

```
move(<Путь к исходному файлу>, <Путь к целевому файлу>[,
                                copy_function=shutil.copy2])
```

Во втором параметре также можно указать путь к каталогу, в который следует переместить исходный файл, — тогда целевой файл получит то же имя, что и исходный. Если целевой файл существует, он будет перезаписан. Метод генерирует те же исключения, что и метод `copyfile()`. Пример:

```
>>> shutil.move(r"c:\book\file4.txt", r"d:\work")
    'd:\\work\\file4.txt'
```

Перемещение файла выполняется в два этапа: создание его копии по целевому пути и удаление исходного файла. Для создания копии файла на первом этапе применяется функция, указанная в параметре `copy_function`. По умолчанию применяется функция `copy2()`, однако можно указать другую (например, `copy()`, которая не копирует метаданные). Пример:

```
>>> shutil.move(r"c:\book\file3.txt", r"d:\work", copy_function=shutil.copy)
    'd:\\work\\file3.txt'
```

Для переименования и удаления файлов предназначены следующие функции из модуля `os`:

- `rename(<Путь к файлу>, <Путь с новым именем>)` — меняет имя файла с указанным путем на заданное новое (указывается вместе с путем к файлу). Если файл не удалось переименовать, генерируется исключение `OSError` или одно из исключений, производных от этого класса. Пример:

```
import os # Подключаем модуль
try:
    os.rename(r"d:\work\file3.txt", r"d:\work\file5.txt")
except OSError:
    print("Файл не удалось переименовать")
else:
    print("Файл успешно переименован")
```

- `remove(<Путь к файлу>)` и `unlink(<Путь к файлу>)` — удаляют файл с указанным путем. Если файл не удалось удалить, генерируется исключение `OSError` или одно из исключений, производных от этого класса. Примеры:

```
>>> os.remove(r"c:\book\file2.txt")
>>> os.unlink(r"d:\work\file5.txt")
```

Функции, позволяющие проверить наличие файла, получить размер файла и др., содержатся в модуле `os.path`:

- `exists(<Путь или дескриптор>)` — возвращает `True`, если файл с указанным путем существует, и `False` — в противном случае. В качестве параметра можно также указать цело-

численный дескриптор открытого файла, возвращенный функцией `open()` из того же модуля `os`. Примеры:

```
>>> import os.path
>>> os.path.exists(r"c:\book\file.txt")
    True
>>> os.path.exists(r"c:\book\file2.txt")
    False
```

- `getsize(<Путь к файлу>)` — возвращает размер файла с указанным путем в байтах. Если файл не существует, генерируется исключение `OSError`. Пример:

```
>>> os.path.getsize(r"c:\book\file.txt")  # Файл существует
    18
```

- `getatime(<Путь к файлу>)` — возвращает время последнего обращения к файлу с указанным путем в виде количества секунд, прошедших с начала эпохи. Если файл не существует, генерируется исключение `OSError`. Пример:

```
>>> import time    # Подключаем модуль time
>>> t = os.path.getatime(r"c:\book\file.txt")
>>> t
    1643369508.5388174
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
    '28.01.2022 14:31:48'
```

- `getctime(<Путь к файлу>)` — возвращает дату создания файла с указанным путем в виде количества секунд, прошедших с начала эпохи. Если файл не существует, генерируется исключение `OSError`. Пример:

```
>>> t = os.path.getctime(r"c:\book\file.txt")
>>> t
    1643210664.3589096
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
    '26.01.2022 18:24:24'
```

- `getmtime(<Путь к файлу>)` — возвращает время последнего изменения файла с указанным путем в виде количества секунд, прошедших с начала эпохи. Если файл не существует, генерируется исключение `OSError`. Пример:

```
>>> t = os.path.getmtime(r"c:\book\file.txt")
>>> t
    1643297017.4162638
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
    '27.01.2022 18:23:37'
```

Получить размер файла с указанным путем, временами создания и последних изменения и обращения к файлу, а также значения других метаданных позволяет функция `stat()` из модуля `os`. Формат вызова функции:

```
stat(<Путь к файлу>[, follow_symlinks=True])
```

Если первым параметром задана символическая ссылка на файл, будут выданы сведения о файле. Однако если при этом дать параметру `follow_symlinks` значение `False`, функция выдаст сведения о самой символической ссылке.

Функция возвращает объект `stat_result`, содержащий атрибуты: `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime` и `st_ctime`.

Пример использования функции `stat()`:

```
>>> import os, time
>>> s = os.stat(r"c:\book\file.txt")
>>> s
    os.stat_result(st_mode=33206, st_ino=37436171902544313, st_dev=2163356289,
                   st_nlink=1, st_uid=0, st_gid=0, st_size=18,
                   st_atime=1643369508, st_mtime=1643297017,
                   st_ctime=1643210664)
>>> s.st_size       # Размер файла
    18
>>> t = s.st_atime # Время последнего обращения к файлу
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
    '28.01.2022 14:31:48'
>>> t = s.st_ctime # Время создания файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
    '26.01.2022 18:24:24'
>>> t = s.st_mtime # Время последнего изменения файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
    '27.01.2022 18:23:37'
```

Изменить время последнего обращения и время изменения файла позволяет функция `utime()` из модуля `os`. Ее формат:

```
utime(<Путь к файлу или его дескриптор>[,
     (<Последний доступ>, <Изменение файла>)][, follow_symlinks=True])
```

Первым параметром также можно указать целочисленный дескриптор открытого файла, возвращенный функцией `open()` из модуля `os`. Вторым параметром указывается кортеж из новых значений в виде количества секунд, прошедших с начала эпохи. Если второй параметр не задан, то время обращения и изменения файла будет заменено на текущее. Если первым параметром задана символическая ссылка на файл, будут выданы сведения о файле. Однако если при этом дать параметру `follow_symlinks` значение `False`, функция выдаст время последнего обращения и изменения у самой символической ссылки. Если файл не существует, генерируется исключение `OSError`. Пример использования функции `utime()`:

```
>>> import os, time
>>> t = time.time() - 600
>>> os.utime(r"c:\book\file.txt", (t, t)) # Текущее время минус 600 с
>>> s = os.stat(r"c:\book\file.txt")
>>> s.st_atime, s.st_mtime
    (1643381757.3795638, 1643381757.3795638)
>>> os.utime(r"c:\book\file.txt")          # Текущее время
>>> s = os.stat(r"c:\book\file.txt")
>>> s.st_atime, s.st_mtime
    (1643382541.5189254, 1643382541.5189254)
```

16.7. Работа с путями

Выполнить различные действия с файловыми путями позволяют следующие функции из модуля os.path:

- abspath(<Относительный путь>) — преобразует заданный относительный путь в абсолютный, учитывая местоположение текущего рабочего каталога, и возвращает результат преобразования:

```
>>> import os, os.path
>>> os.chdir(r"c:\book")
>>> os.path.abspath(r"file.txt")
    'c:\\book\\file.txt'
>>> os.path.abspath(r"folderl/file.txt")
    'c:\\book\\folderl\\file.txt'
>>> os.path.abspath(r"../file.txt")
    'c:\\file.txt'
```

- isabs(<Путь>) — возвращает True, если заданный путь является абсолютным, и False — в противном случае:

```
>>> os.path.isabs(r"C:\book\file.txt")
    True
>>> os.path.isabs("file.txt")
    False
```

- basename(<Путь>) — возвращает последний сегмент указанного пути (имя файла или последнего каталога в пути):

```
>>> os.path.basename(r"C:\book\folderl\file.txt")
    'file.txt'
>>> os.path.basename(r"C:\book\folder")
    'folder'
>>> os.path.basename("C:\\book\\folder\\")
    ''
```

- dirname(<Путь>) — возвращает заданный путь без последнего сегмента:

```
>>> os.path.dirname(r"C:\book\folder\file.txt")
    'C:\\book\\folder'
>>> os.path.dirname(r"C:\book\folder")
    'C:\\book'
>>> os.path.dirname("C:\\book\\folder\\")
    'C:\\book\\folder'
```

- split(<Путь>) — возвращает кортеж из двух элементов: заданного пути без последнего сегмента и последнего сегмента этого пути:

```
>>> os.path.split(r"C:\book\folder\file.txt")
    ('C:\\book\\folder', 'file.txt')
>>> os.path.split(r"C:\book\folder")
    ('C:\\book', 'folder')
>>> os.path.split("C:\\book\\folder\\")
    ('C:\\book\\folder', '')
```

- `splitdrive(<Путь>)` — возвращает кортеж из двух элементов: обозначения диска, взятого из заданного пути, и остального пути без обозначения диска:

```
>>> os.path.splitdrive(r"C:\book\folder\file.txt")
    ('C:', '\\book\\folder\\file.txt')
```

- `splitext(<Путь>)` — возвращает кортеж из двух элементов: пути с именем файла, путь к которому был указан в параметре, без расширения, и расширения файла (фрагмента после последней точки):

```
>>> os.path.splitext(r"C:\book\folder\file.tar.gz")
    ('C:\\book\\folder\\file.tar', '.gz')
```

- `join(<Сегмент 1>[, . . ., <Сегмент N>])` — возвращает путь, составленный из указанных сегментов:

```
>>> os.path.join("C:\\", "book\\folder", "file.txt")
    'C:\\book\\folder\\file.txt'
>>> os.path.join(r"C:\\", "book/folder/", "file.txt")
    'C:\\\\book/folder/file.txt'
```

  В последнем примере в пути использованы разные слеши, что дало в результате некорректный путь. Чтобы этот путь сделать корректным, необходимо воспользоваться функцией `normpath(<Путь>)` из того же модуля `os.path`:

```
>>> p = os.path.join(r"C:\\", "book/folder/", "file.txt")
>>> os.path.normpath(p)
    'C:\\book\\folder\\file.txt'
```

- `commonpath(<Последовательность путей>)` — возвращает самый длинный общий фрагмент, расположенный в начале путей из заданной последовательности. Обрабатывает пути посегментно. Пример:

```
>>> os.path.commonpath((r"c:\book\folder1", r"c:\book\folder2",
...                                        r"c:\book\found"))
    'c:\\book'
```

- `commonprefix()` — аналогична `commonpath()`, только обрабатывает пути посимвольно (вследствие чего возвращенный путь может оказаться несуществующим):

```
>>> os.path.commonprefix((r"c:\book\folder1", r"c:\book\folder2",
...                                           r"c:\book\found"))
    'c:\\book\\fo'
```

16.8. Перенаправление ввода/вывода

Стандартные потоки ввода/вывода схожи с файловыми объектами. Они имеют целочисленные дескрипторы: стандартный поток ввода `stdin` — 0, поток вывода `stdout` — 1, поток вывода ошибок `stderr` — 2. Потоки `stdout` и `stderr` поддерживают метод `write()`, а поток `stdin` — метод `readline()`.

Если потоку `stdout` или `stderr` присвоить ссылку на файловый объект, то можно перенаправить стандартный вывод в соответствующий файл:

```
>>> import sys                          # Подключаем модуль sys
>>> tmp_out = sys.stdout                # Сохраняем ссылку на поток stdout
```

```
>>> f = open(r"c:\book\file.txt", "a")  # Открываем файл на дозапись
>>> sys.stdout = f                      # Перенаправляем вывод в файл
>>> print("Пишем строку в файл")
>>> sys.stdout = tmp_out                # Восстанавливаем стандартный вывод
>>> f.close()                           # Закрываем файл
>>> print("Пишем строку в стандартный вывод")
    Пишем строку в стандартный вывод
```

Функция `print()` напрямую поддерживает перенаправление вывода. Для этого используется параметр `file`, который по умолчанию ссылается на стандартный поток вывода. Например, записать строку в файл можно так:

```
>>> f = open(r"c:\book\file.txt", "a")
>>> print("Пишем строку в файл", file=f)
>>> f.close()
```

Параметр `flush` позволяет указать, когда следует выполнять непосредственное сохранение данных из промежуточного буфера в файл. Если его значение равно `False` (значение по умолчанию), сохранение будет выполнено лишь после закрытия файла или после вызова метода `flush()`. Чтобы указать интерпретатору Python выполнять сохранение после каждого вызова функции `print()`, следует присвоить этому параметру значение `True`, например:

```
>>> f = open(r"c:\book\file.txt", "a")
>>> print("Пишем строку в файл", file=f, flush=True)
>>> print("Пишем другую строку в файл", file=f, flush=True)
>>> f.close()
```

Стандартный ввод `stdin` также можно перенаправить. В этом случае функция `input()` будет читать одну строку из файла при каждом вызове. При достижении конца файла возникнет исключение `EOFError`. Для примера выведем содержимое файла с помощью перенаправления потока ввода (листинг 16.1).

Листинг 16.1. Перенаправление потока ввода

```
import sys
tmp_in = sys.stdin                   # Сохраняем ссылку на поток stdin
f = open(r"c:\book\file.txt", "r")   # Открываем файл на чтение
sys.stdin = f                        # Перенаправляем ввод
while True:
    try:
        line = input()               # Считываем строку из файла
        print(line)                  # Выводим строку
    except EOFError:                 # Если достигнут конец файла,
        break                        # выходим из цикла
sys.stdin = tmp_in                   # Восстанавливаем стандартный ввод
f.close()                            # Закрываем файл
input()
```

Метод `isatty()` объекта стандартного потока ввода возвращает `True`, если объект потока ссылается на консоль, и `False` — в противном случае:

```
>>> tmp_in = sys.stdin           # Сохраняем ссылку на поток stdin
>>> f = open(r"c:\book\file.txt", "r")
```

```
>>> sys.stdin = f                # Перенаправляем ввод
>>> sys.stdin.isatty()           # Не ссылаемся на терминал
    False
>>> sys.stdin = tmp_in           # Восстанавливаем стандартный ввод
>>> sys.stdin.isatty()           # Ссылаемся на терминал
    True
>>> f.close()                    # Закрываем файл
```

С помощью стандартного вывода stdout можно создать *индикатор процесса* непосредственно в консоли. Сделать это можно, используя символ перевода каретки \r, чтобы переместиться в начало строки и перезаписать ранее выведенную информацию. Рассмотрим вывод индикатора процесса на примере (листинг 16.2).

Листинг 16.2. Индикатор процесса в консоли

```
import sys, time
for i in range(5, 101, 5):
    sys.stdout.write("\r ... %s%%" % i) # Обновляем индикатор
    sys.stdout.flush()                  # Сбрасываем содержимое буфера
    time.sleep(1)                       # Засыпаем на 1 секунду
sys.stdout.write("\rПроцесс завершен\n")
input()
```

Сохраним код из листинга в файле и запустим его щелчком мыши. Индикатор процесса в консоли будет обновляться каждую секунду.

16.9. Сохранение объектов в файлах

Сохранить объект в файле и в дальнейшем восстановить этот объект из файла позволяют модули pickle и shelve.

Модуль pickle предоставляет следующие функции:

- dump(<Объект>, <Файл>[, <Протокол>][, fix_imports=True]) — производит сериализацию заданного объекта и записывает его в указанный файл. В параметре <Файл> указывается файловый объект, открытый на запись в двоичном подрежиме. Пример:

  ```
  >>> import pickle
  >>> f = open(r"c:\book\file.txt", "wb")
  >>> obj = ["Строка", (2, 3)]
  >>> pickle.dump(obj, f)
  >>> f.close()
  ```

  Формат, в котором представляется сериализованное значение, зависит от указанного протокола, который задается в виде числа от 0 до значения константы pickle.HIGHEST_PROTOCOL. Если протокол не указан, будет использован протокол 4 (переменная pickle.DEFAULT_PROTOCOL, до Python 3.8 этот параметр имел значение по умолчанию 3);

- load(<Файл>) — читает данные из указанного файла и преобразует их в исходный объект, который и возвращает. В параметре <Файл> указывается файловый объект, открытый на чтение в двоичном подрежиме. Пример:

```
>>> f = open(r"c:\book\file.txt", "rb")
>>> obj = pickle.load(f)
>>> obj
    ['Строка', (2, 3)]
>>> f.close()
```

В одном файле можно сохранить сразу несколько объектов, последовательно вызывая функцию `dump()`:

```
>>> obj1 = ["Строка", (2, 3)]
>>> obj2 = (1, 2)
>>> f = open(r"c:\book\file.txt", "wb")
>>> pickle.dump(obj1, f)          # Сохраняем первый объект
>>> pickle.dump(obj2, f)          # Сохраняем второй объект
>>> f.close()
```

Для восстановления объектов из файла необходимо несколько раз вызвать функцию `load()`:

```
>>> f = open(r"c:\book\file.txt", "rb")
>>> obj1 = pickle.load(f)        # Восстанавливаем первый объект
>>> obj2 = pickle.load(f)        # Восстанавливаем второй объект
>>> obj1, obj2
    (['Строка', (2, 3)], (1, 2))
>>> f.close()
```

Сохранить объект в файл можно также с помощью метода `dump(<Объект>)` класса `Pickler`. Конструктор класса имеет следующий формат:

```
Pickler(<Файл>[, <Протокол>])
```

Параметры, передаваемые конструктору, аналогичны таковым у функции `dump()`. Пример:

```
>>> f = open(r"c:\book\file.txt", "wb")
>>> obj = ["Строка", (2, 3)]
>>> pkl = pickle.Pickler(f)
>>> pkl.dump(obj)
>>> f.close()
```

Восстановить объект из файла позволяет метод `load()` класса `Unpickler`. Формат конструктора класса:

```
Unpickler(<Файл>)
```

Задаваемый параметр аналогичен таковому у функции `load()`. Пример:

```
>>> f = open(r"c:\book\file.txt", "rb")
>>> obj = pickle.Unpickler(f).load()
>>> obj
    ['Строка', (2, 3)]
>>> f.close()
```

Модуль `shelve` позволяет сохранять объекты под заданными строковыми ключами, формируя в файле нечто похожее на словарь. Для сериализации объекта используются возможности модуля `pickle`, а для его записи в файл — модуль `dbm`.

Открыть файл с указанным путем, хранящий набор объектов, позволяет функция `open()`. Функция имеет следующий формат:

```
open(<Путь к файлу>[, flag="c"][, protocol=None][, writeback=False])
```

В параметре `flag` можно указать режим открытия файла в виде строкового обозначения:

- `r` — только чтение;
- `w` — чтение и запись;
- `c` — чтение и запись (значение по умолчанию). Если файл не существует, он будет создан;
- `n` — чтение и запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан.

Параметр `protocol` задает протокол, применяемый при сериализации записываемых объектов, в виде числа от 0 до 4. По умолчанию используется значение переменной `pickle.DEFAULT_PROTOCOL` (4). Если параметру `writeback` дать значение `True`, считываемые из файла объекты будут кешироваться в памяти, что в ряде случаев может повысить производительность.

Функция `open()` возвращает объект, представляющий открытый файл. В этот объект можно добавлять новые элементы и считывать существующие, применяя тот же синтаксис, что используется для работы с отображениями (см. *главу 9*).

Объект-файл поддерживает следующие методы:

- `close()` — закрывает текущий файл. Для примера создадим файл и сохраним в нем список и кортеж:

```
>>> import shelve                              # Подключаем модуль
>>> db = shelve.open(r"c:\book\db1.dat")       # Открываем файл
>>> db["obj1"] = [1, 2, 3, 4, 5]              # Сохраняем список
>>> db["obj2"] = (6, 7, 8, 9, 10)             # Сохраняем кортеж
>>> db["obj1"], db["obj2"]                    # Выводим значения
    ([1, 2, 3, 4, 5], (6, 7, 8, 9, 10))
>>> db.close()                                # Закрываем файл
```

- `keys()` — возвращает объект с ключами, хранящимися в текущем файле;
- `values()` — возвращает объект со значениями, хранящимися в текущем файле;
- `items()` — возвращает итератор, который на каждой итерации выдает кортеж, содержащий очередные ключ и значение:

```
>>> db = shelve.open(r" db1.dat")
>>> db.keys(), db.values()
    (KeysView(<shelve.DbfilenameShelf object at 0x0000023FB67FA110>),
     ValuesView(<shelve.DbfilenameShelf object at 0x0000023FB67FA110>))
>>> list(db.keys()), list(db.values())
    (['obj1', 'obj2'], [[1, 2, 3, 4, 5], (6, 7, 8, 9, 10)])
>>> db.items()
    ItemsView(<shelve.DbfilenameShelf object at 0x0000023FB67F9FF0>)
>>> list(db.items())
    [('obj1', [1, 2, 3, 4, 5]), ('obj2', (6, 7, 8, 9, 10))]
>>> db.close()
```

- `get(<Ключ>[, <Значение по умолчанию>])` — возвращает значение, записанное в текущем файле под указанным ключом. Если ключ отсутствует, возвращается значение `None` или значение, указанное во втором параметре;
- `setdefault(<Ключ>[, <Значение по умолчанию>])` — возвращает значение, сохраненное в текущем файле под указанным ключом. Если ключ отсутствует, создается новый эле-

мент со значением, указанным во втором параметре, и в качестве результата возвращается это значение. Если второй параметр не указан, значением нового элемента будет None;

- pop(<Ключ>[, <Значение по умолчанию>]) — удаляет из текущего файла элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, возвращается значение из второго параметра. Если ключ отсутствует и второй параметр не указан, генерируется исключение KeyError;
- popitem() — удаляет из текущего файла последний элемент и возвращает кортеж из его ключа и значения (в версиях Python, предшествовавших 3.7, метод удалял произвольный элемент). Если файл пустой, генерируется исключение KeyError;
- clear() — удаляет из текущего файла все элементы. Ничего не возвращает;
- update() — добавляет элементы в текущий файл. Ничего не возвращает. Если элемент с указанным ключом уже присутствует, то его значение будет перезаписано. Форматы метода:

```
update(<Ключ 1>=<Значение 1>[, . . ., <Ключ N>=<Значение N>])
update(<Словарь>)
update(<Список кортежей с двумя элементами: ключом и значением>)
update(<Список списков с двумя элементами: ключом и значением >)
```

Помимо этих методов, можно пользоваться функцией len() для получения количества элементов, оператором del для удаления заданного элемента, операторами in и not in для проверки существования или несуществования ключа:

```
>>> db = shelve.open(r"c:\book\db1.dat")
>>> len(db)            # Количество элементов
    2
>>> "obj1" in db
    True
>>> del db["obj1"]     # Удаление элемента
>>> "obj1" in db
    False
>>> "obj1" not in db
    True
>>> db.close()
```

16.10. Работа с каталогами

Для работы с каталогами служат следующие функции из модуля os:

- mkdir(<Имя каталога>[, <Права доступа>]) — создает в текущем рабочем каталоге новый каталог с указанными именем и правами доступа. Права доступа задаются восьмеричным числом (по умолчанию: 0o777). Пример:

```
>>> os.mkdir("newfolder")
```

- makedirs(<Путь>[, mode=0o777][, exist_ok=False]) — создает каталог с указанным путем, также создавая все каталоги, записанные в заданном пути, если они не существуют:

```
>>> os.makedirs(r"c:\book\folder3\folder4\folder5")
```

Параметр mode задает права доступа у созданных каталогов. Если каталог с указанным путем существует и параметру exist_ok дано значение False, будет сгенерировано исключение FileExistsError, если же параметру дать значение True, ничего не произойдет;

- rmdir(<Имя каталога>) — удаляет каталог с заданным именем из текущего рабочего каталога. Можно удалить лишь пустой каталог. Если каталог не пуст или не существует, генерируется исключение, производное от класса OSError. Пример:

```
>>> os.rmdir("newfolder")
```

- listdir(<Путь>) — возвращает список имен файлов и каталогов, хранящихся в каталоге с указанным путем:

```
>>> os.listdir("C:\\book\\folder1\\")
    ['file1.txt', 'file2.txt', 'file3.txt', 'folder1', 'folder2']
```

- walk() — последовательно просматривает сначала начальный каталог, потом вложенные в него каталоги, потом каталоги следующего уровня вложенности и выдает хранящиеся в них файлы и каталоги. Формат функции:

```
walk(<Путь к начальному каталогу>[, topdown=True][, onerror=None][,
                                                   followlinks=False])
```

Функция walk() возвращает итератор, при каждом вызове выдающий кортеж из трех элементов: пути к очередному каталогу, списка путей к каталогам и списка путей к файлам, находящимся в этом каталоге.

Параметр onerror задает поведение функции при возникновении ошибки. Значение None (используется по умолчанию) предписывает ничего не делать при их возникновении. Также можно указать функцию, которая должна принимать в качестве параметра объект исключения. Эта функция может как просто сообщить об ошибке (тогда исполнение функции walk() продолжится), так и сгенерировать другое исключение (что вызовет прерывание работы функции walk()).

Параметр followlinks указывает функции, как поступить, если встретится символическая ссылка: проследовать по ней (значение True) или не следовать по ней (значение False, используемое по умолчанию).

Параметр topdown задает последовательность обхода каталогов. Если в качестве значения указано True (значение по умолчанию), последовательность обхода будет такой:

```
>>> for (p, d, f) in os.walk("C:\\book\\folder1\\"): print(p)
...
    C:\book\folder1\
    C:\book\folder1\folder1_1
    C:\book\folder1\folder1_1\folder1_1_1
    C:\book\folder1\folder1_1\folder1_1_2
    C:\book\folder1\folder1_2
```

Если в параметре topdown указано значение False, последовательность обхода будет другой:

```
>>> for (p, d, f) in os.walk("C:\\book\\folder1\\", False): print(p)
...
    C:\book\folder1\folder1_1\folder1_1_1
    C:\book\folder1\folder1_1\folder1_1_2
```

```
C:\book\folder1\folder1_1
C:\book\folder1\folder1_2
C:\book\folder1\
```

Благодаря такой последовательности обхода перед удалением заданного каталога можно удалить все находящиеся в нем файлы и каталоги (поскольку функция `rmdir()` позволяет удалять только пустые каталоги):

```
import os
for (p, d, f) in os.walk("C:\\book\\folder1\\", topdown=False):
    for file_name in f: # Удаляем все файлы
        os.remove(os.path.join(p, file_name))
    for dir_name in d: # Удаляем все каталоги
        os.rmdir(os.path.join(p, dir_name))
```

Внимание!

Очень осторожно используйте этот код. Если в качестве первого параметра в функции `walk()` указать корневой каталог диска, то все имеющиеся в нем файлы и каталоги будут удалены.

Удалить дерево каталогов позволяет также функция `rmtree()` из модуля `shutil`. Функция имеет следующий формат:

```
rmtree(<Путь>[, <Обработка ошибок>[, <Обработчик ошибок>]])
```

Если в параметре `<Обработка ошибок>` указано значение `True`, ошибки будут проигнорированы. Если указано значение `False` (используется по умолчанию), в третьем параметре можно задать ссылку на функцию, которая будет вызываться при возникновении исключения. Указываемая функция должна принимать три параметра: ссылку на функцию, выполнение которой вызвало возникновение ошибки, путь к файлу или каталогу, попытка удаления которого вызвала ошибку, и объект со сведениями о возникшей ошибке. Пример:

```
import shutil
shutil.rmtree("C:\\book\\folder1\\")
```

- `path.normcase(<Путь>)` — преобразует заданный путь к каталогу к виду, подходящему для использования в текущей операционной системе. В Windows преобразует все прямые слеши в обратные. Также во всех системах приводит все буквы пути к нижнему регистру. Пример:

```
>>> os.path.normcase(r"c:/BoOk/fIlE.TxT")
    'c:\\book\\file.txt'
```

Проверить, на что указывает заданный путь, можно с помощью следующих функций из модуля `os.path`:

- `isdir(<Путь>)` — возвращает `True`, если заданный путь указывает на каталог, и `False` — в противном случае:

```
>>> import os.path
>>> os.path.isdir(r"C:\book\file.txt")
    False
>>> os.path.isdir("C:\\book\\")
    True
```

- `isfile(<Путь>)` — возвращает `True`, если заданный путь указывает на файл, и `False` — в противном случае:

```
>>> os.path.isfile(r"C:\book\file.txt")
    True
>>> os.path.isfile("C:\\book\\")
    False
```

- `islink(<Путь>)` — возвращает `True`, если заданный путь указывает на символическую ссылку, и `False` — в противном случае. Если символические ссылки не поддерживаются, функция возвращает `False`.

Описанная ранее функция `listdir()` возвращает список всех файлов и каталогов, что хранятся в указанном каталоге. Если необходимо ограничить список определенными критериями, следует воспользоваться функцией `glob()` из модуля `glob`. Она возвращает список путей к файлам и каталогам, имена которых совпадают с заданным путем-шаблоном. Формат вызова этой функции:

```
glob(<Путь-шаблон>[, recursive=False][, root_dir=None])
```

Задаваемый путь-шаблон — это обычный путь, в котором можно использовать следующие специальные символы:

- `?` — любой одиночный символ;
- `*` — любое количество символов;
- `[<Символы>]` — символы, которые должны быть на этом месте в пути. Можно задать символы или определить их диапазон через дефис;
- `**` — любой файл или каталог.

Допускается указать как абсолютный, так и относительный путь-шаблон. Последний в этом случае будет отсчитываться от текущего рабочего каталога. Примеры:

```
>>> import glob
>>> glob.glob("C:\\book\\folder1\\*.txt")
    ['C:\\book\\folder1\\file.txt', 'C:\\book\\folder1\\file1.txt',
     'C:\\book\\folder1\\file2.txt']
>>> glob.glob("C:\\book\\folder1\\*.html") # Абсолютный путь-шаблон
    ['C:\\book\\folder1\\index.html']
>>> glob.glob("folder1/*.html")            # Относительный путь-шаблон
    ['folder1\\index.html']
>>> glob.glob("C:\\book\\folder1\\*[0-9].txt")
    ['C:\\book\\folder1\\file1.txt', 'C:\\book\\folder1\\file2.txt']
>>> glob.glob("C:\\book\\folder1\\*\\*.html")
    ['C:\\book\\folder1\\folder1_1\\index.html',
     'C:\\book\\folder1\\folder1_2\\test.html']
>>> glob.glob("C:\\book\\folder1\\**")
    ['C:\\book\\folder1\\file.txt', 'C:\\book\\folder1\\file1.txt',
     'C:\\book\\folder1\\file2.txt', 'C:\\book\\folder1\\folder1_1',
     'C:\\book\\folder1\\folder1_2']
```

Если параметр `recursive` имеет значение `True`, специальный символ `**`, будучи указанным в качестве сегмента пути-шаблона, обозначает любое количество каталогов, вложенных друг в друга:

```
>>> glob.glob("C:\\book\\**\\*.html", recursive=True)
    ['C:\\book\\folder1\\index.html',
     'C:\\book\\folder1\\folder1_1\\index.html',
     'C:\\book\\folder1\\folder1_2\\test.html']
```

В параметре `root_dir` (поддерживается, начиная с Python 3.10) можно указать путь к каталогу, от которого будет отсчитываться относительный путь-шаблон:

```
>>> glob.glob("folder1/*.html", root_dir="c:\\book")
    ['folder1\\index.html']
```

16.10.1. Функция *scandir()*

Функция `scandir([<Путь>])` из модуля `os` выполняет просмотр содержимого каталога с указанным путем. Если путь не задан, будет использоваться значение `"."` (путь к текущему рабочему каталогу).

Функция `scandir()` возвращает итератор, на каждом проходе выдающий очередной файл или каталог, который находится по указанному пути. Этот файл или каталог представляется объектом класса `DirEntry`, определенного в том же модуле `os`, который хранит всевозможные сведения о файле (каталоге).

Класс `DirEntry` поддерживает следующие атрибуты:

- `name` — имя файла (каталога);
- `path` — путь к файлу (каталогу), составленный из пути, что был указан в вызове функции `scandir()`, и имени файла (каталога), хранящегося в атрибуте `name`.

Выведем список всех файлов и каталогов, находящихся в текущем рабочем каталоге (при вводе команд в IDLE Shell текущим рабочим станет каталог, где установлен Python):

```
>>> import os
>>> for entry in os.scandir():
...     print(entry.path)
...
    ./DLLs
    ./Doc
    ./include
    ... Фрагмент пропущен ...
    ./Tools
    ./vcruntime140.dll
    ./vcruntime140_1.dll
```

Видно, что путь, хранимый атрибутом `path`, составляется из пути, заданного в вызове функции `scandir()` (в нашем случае это используемый по умолчанию путь `.`), и имени файла (каталога). Теперь попробуем указать путь явно:

```
>>> for entry in os.scandir("c:\python310"):
...     print(entry.path)
...
    c:\python310\DLLs
    c:\python310\Doc
    c:\python310\include
    ... Фрагмент пропущен ...
```

```
c:\python310\Tools
c:\python310\vcruntime140.dll
c:\python310\vcruntime140_1.dll
```

Еще класс `DirEntry` поддерживает следующие методы:

- `is_file([follow_symlinks=True])` — возвращает `True`, если текущий объект является файлом, и `False` — в противном случае. Если текущий объект представляет собой символическую ссылку, и параметру `follow_symlinks` дано значение `True`, проверяется элемент, на который указывает эта символическая ссылка. Если же параметр `follow_symlinks` имеет значение `False`, всегда возвращается `False`;
- `is_dir([follow_symlinks=True])` — возвращает `True`, если текущий объект является каталогом, и `False` — в противном случае. Если текущий объект представляет собой символическую ссылку и параметру `follow_symlinks` дано значение `True`, проверяется элемент, на который указывает эта символическая ссылка. Если же параметр `follow_symlinks` имеет значение `False`, всегда возвращается `False`;
- `is_symlink()` — возвращает `True`, если текущий объект является символической ссылкой, и `False` — в противном случае;
- `stat([follow_symlinks=True])` — возвращает объект `stat_result` (см. *разд. 16.6*), хранящий сведения о файле. Если текущий объект представляет собой символическую ссылку и параметр `follow_symlinks` имеет значение `True`, возвращаются сведения об элементе, на который указывает эта символическая ссылка. Если же параметр `follow_symlinks` имеет значение `False`, возвращаются сведения о самой символической ссылке. В Windows атрибуты `st_ino`, `st_dev` и `st_nlink` объекта `stat_result`, возвращенного методом `stat()`, всегда хранят 0, и для получения их значений следует воспользоваться функцией `stat()` из модуля `os`, описанной в *разд. 16.6*.

Пара примеров:

- вывод списка всех каталогов, что находятся в каталоге, где установлен Python:

```
>>> for entry in os.scandir("c:\python310"):
...     if entry.is_dir():
...         print(entry.name, end=", ")
...
    DLLs, Doc, include, Lib, libs, Scripts, tcl, Tools,
```

- вывод списка всех DLL-файлов, хранящихся в каталоге Windows, без обработки символических ссылок:

```
>>> for entry in os.scandir("c:\windows"):
...     if entry.is_file(follow_symlinks=False) and \
...        entry.name.endswith(".dll"):
...         print(entry.name, end=", ")
...
    pyshellext.amd64.dll, twain_32.dll, womtrust.dll, wontrust.dll,
```

Итератор, возвращаемый функцией `scandir()`, является менеджером контекста и может быть использован в обработчиках контекстов (см. *разд. 14.2*):

```
>>> with os.scandir() as it:
...     for entry in it:
...         print(entry.name)
```

Путь в вызове функции `scandir()` также можно указать в виде объекта `bytes`. В таком случае значения атрибутов `name` и `path` класса `DirEntry` тоже будут представлять собой объекты `bytes`. Пример:

```
>>> with os.scandir(b"c:\python310") as it:
...     for entry in it:
...         print(entry.name)
...
    b'DLLs'
    b'Doc'
    b'include'
    ... Фрагмент пропущен ...
```

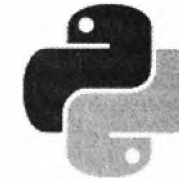
16.11. Исключения, генерируемые файловыми операциями

При возникновении нештатной ситуации во время работы с файлами и каталогами генерируется исключение класса `OSError` или одно из следующих исключений, являющихся его подклассами:

- `BlockingIOError` — не удалось заблокировать файл или поток ввода/вывода;
- `ConnectionError` — ошибка сетевого соединения. Может возникнуть при открытии файла по сети. Является базовым классом для ряда более специфических исключений, описанных в документации по Python;
- `FileExistsError` — файл или каталог с заданным путем уже существует;
- `FileNotFoundError` — файл или каталог с заданным путем не найден;
- `InterruptedError` — файловая операция прервана операционной системой;
- `IsADirectoryError` — вместо пути к файлу указан путь к каталогу;
- `NotADirectoryError` — вместо пути к каталогу указан путь к файлу;
- `PermissionError` — отсутствуют права на доступ к указанному файлу или каталогу;
- `TimeoutError` — истекло время, отведенное системой на выполнение операции.

Пример кода, обрабатывающего некоторые из указанных исключений:

```
try
    open("C:\temp\new\file.txt")
except FileNotFoundError:
    print("Файл отсутствует")
except IsADirectoryError:
    print("Это не файл, а каталог")
except PermissionError:
    print("Отсутствуют права на доступ к файлу")
except OSError:
    print("Неустановленная ошибка открытия файла")
```



ГЛАВА 17

Работа с механизмами Windows

Python предоставляет инструменты для работы с реестром Windows. А с помощью дополнительных библиотек можно получать пути к системным каталогам этой операционной системы и создавать ярлыки.

Следует помнить, что все эти инструменты работают лишь в Microsoft Windows. Выяснить, запущена ли программа в Windows, можно, проверив значение переменной `platform` из модуля `sys`, — оно должно быть равно `"win32"` (одинаково в 32- и 64-разрядных редакциях Windows). Пример:

```
import sys
if sys.platform == "win32":
    # Текущая платформа — Windows.
    # Можно использовать специфические механизмы этой системы.
else:
    # Какая-то другая система.
    # Специфические инструменты Windows использовать нельзя.
```

17.1. Работа с реестром

Реестр Windows организован в виде нескольких крупных разделов, называемых *кустами* (hive). Каждый куст имеет уникальное имя: `HKEY_CURRENT_USER` (хранит настройки текущего пользователя), `HKEY_LOCAL_MACHINE` (настройки уровня системы) и др.

В кусте может быть создано произвольное количество подразделов — *ветвей* (в англоязычной документации используется не очень удачный термин key — ключ). Каждая ветвь идентифицируется путем (так, ветвь, хранящая пользовательские настройки Windows, расположена по пути Software\Microsoft\Windows).

Любая ветвь может хранить произвольное количество именованных значений (*элементов*) и вложенных в нее ветвей, которые, в свою очередь, могут содержать произвольное количество элементов и вложенных в них ветвей. Кроме того, любая ветвь может хранить в себе одно произвольное значение. Значения, хранящиеся в элементах и ветвях, могут принадлежать к одному из ограниченного набора типов: строковому, целочисленному, списочному и пр.

Все программные инструменты Python, предназначенные для работы с реестром и описываемые далее, реализованы в модуле `winreg`. Этот модуль следует предварительно подключить с помощью инструкции:

```
import winreg
```

17.1.1. Открытие и закрытие ветвей реестра

Чтобы получить доступ к требуемой ветви реестра, хранящимся в ней элементам и ключам, предварительно следует открыть эту ветвь. Открытие ветви выполняют функции `OpenKey()` и `OpenKeyEx()`. Обе функции совершенно аналогичны и вызываются в формате:

```
OpenKey|OpenKeyEx(<Обозначение куста>, <Путь к открываемому ключу>[,
                  reserved=0][, access=winreg.KEY_READ])
```

Обозначение куста реестра, в котором хранится открываемая ветвь, указывается в виде значения одной из следующих переменных: `HKEY_CLASSES_ROOT`, `HKEY_CURRENT_USER`, `HKEY_LOCAL_MACHINE`, `HKEY_USERS`, `HKEY_PERFORMANCE_DATA`, `HKEY_CURRENT_CONFIG`, `HKEY_DYN_DATA`. Путь к открываемой ветви задается в виде строки. Параметр `reserved` зарезервирован для использования в будущих версиях Windows, в качестве значения ему всегда дается число 0.

Параметр `access` задает права доступа к открываемой ветви и ее содержимому в виде значения одной из следующих переменных или их комбинации через двоичный оператор `|`:

- `KEY_QUERY_VALUE` — чтение данных;
- `KEY_ENUMERATE_SUB_KEYS` — перебор вложенных ветвей;
- `KEY_NOTIFY` — получение оповещений об изменениях содержимого ветви;
- `KEY_SET_VALUE` — создание, запись, удаление элементов и удаление вложенных ветвей;
- `KEY_CREATE_SUB_KEY` — создание вложенных ветвей;
- `KEY_READ` и `KEY_EXECUTE` — полные права на чтение (комбинации `KEY_QUERY_VALUE`, `KEY_ENUMERATE_SUB_KEYS` и `KEY_NOTIFY`);
- `KEY_WRITE` — полные права на запись (комбинации `KEY_SET_VALUE` и `KEY_CREATE_SUB_KEY`);
- `KEY_ALL_ACCESS` — полные права на чтение и запись (комбинации `KEY_READ` и `KEY_WRITE`).

Следующие две переменные поддерживаются только в 64-разрядной редакции языка Python:

- `KEY_WOW64_64KEY` — работа с 64-разрядной редакцией реестра;
- `KEY_WOW64_32KEY` — работа с 32-разрядной редакцией реестра.

Функции `OpenKey()` и `OpenKeyEx()` возвращают объект класса `PyHKEY`, представляющий открытую ветвь реестра. Если ветвь не удалось открыть, генерируется исключение `OSError` или одно из исключений, производных от этого класса.

Закончив работу с открытой ветвью, ее следует закрыть, выполнив одно из следующих действий:

- вызвав функцию `CloseKey(<Закрываемая ветвь>)`;
- вызвав у объекта закрываемой ветви метод `Close()`.

Пример:

```
>>> import winreg                          # Подключаем модуль
>>> # Открываем ветвь Software куста HKEY_CURRENT_USER с правами на
>>> # запись
>>> k = winreg.OpenKey(winreg.HKEY_CURRENT_USER, "Software",
...                    access=winreg.KEY_WRITE)
>>> # Создаем в ветви Software вложенную ветвь Python
>>> sk = winreg.CreateKey(k, "Python")
```

```
>>> # Записываем в ветвь Python элемент Version со значением "3.10.1"
>>> winreg.SetValueEx(sk, "Version", 0, winreg.REG_SZ, "3.10.1")
>>> # Создаем в ветви Python вложенную ветвь 3.10.1 со значением
>>> # "Installed"
>>> winreg.SetValue(sk, "3.10.1", winreg.REG_SZ, "Installed")
>>> # Закрываем вложенную ветвь Python с помощью метода Close()
>>> sk.Close()
>>> # Закрываем ветвь Software с помощью функции CloseKey()
>>> winreg.CloseKey(k)
```

Класс `PyHKEY`, представляющий ветвь реестра, является менеджером контекста. Следовательно, для работы с открытыми ветвями реестра можно использовать обработчики контекста. Пример:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...                     r"Software\Python") as k:
...     print(winreg.QueryValueEx(k, "Version")[0])
...
    3.10.1
```

Чтобы открыть ветвь реестра удаленного компьютера, сначала следует подключиться к его реестру. Это выполняется вызовом функции `ConnectRegistry()`, которая вызывается в следующем формате:

```
ConnectRegistry(<Сетевое имя компьютера>, <Обозначение куста>)
```

Сетевое имя компьютера указывается в виде строки. Если задать значение `None`, будет выполнено подключение к реестру локального компьютера. Обозначение куста задается в виде значения одной из приведенных ранее переменных.

В качестве результата возвращается объект класса `PyHKEY`, представляющий указанный куст реестра заданного удаленного компьютера. Если подключиться к удаленному реестру не удалось, генерируется исключение `OSError` или одно из исключений, производных от этого класса.

Объект куста, полученный от функции `ConnectRegistry()`, следует передать первым параметром функции `OpenKey()` или `OpenKeyEx()`, чтобы открыть нужную ветвь удаленного реестра. Пример:

```
rh = winreg.ConnectRegistry(r"\\SomeComputer", winreg.HKEY_CURRENT_USER)
with winreg.OpenKey(rh, r"Software\Python") as k:
    . . .
```

17.1.2. Чтение и запись данных реестра

Функции, используемые для чтения данных из реестра, записи в него значений, создания и удаления элементов и ветвей, в качестве первого параметра принимают объект открытой ветви, относящийся к классу `PyHKEY` и возвращенный функцией `OpenKey()` или `OpenKeyEx()` (см. *разд. 17.1.1*). При возникновении ошибки они генерируют исключение `OSError` или одно из исключений, производных от этого класса. Вот эти функции:

- `QueryValueEx(<Ветвь>, <Имя элемента>)` — считывает из указанной ветви реестра значение элемента с заданным именем. Возвращает кортеж из двух элементов: собственно считанного значения и обозначения его типа, представленного целым числом. Пример:

```
>>> import winreg
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...                     r"Software\Python") as k:
...     print(winreg.QueryValueEx(k, "Version"))
...
    ('3.10.1', 1)
>>> # 1 — это обозначение строкового типа
```

- `QueryValue(<Ветвь>, <Имя вложенной ветви>)` — возвращает значение вложенной ветви с заданным именем, находящейся в указанной ветви. Возвращаемое значение всегда представляется в виде строки. Если ветвь не имеет значения, возвращается пустая строка. Если вторым параметром задано значение `None`, возвращает значение ветви, заданной первым параметром. Пример:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...                     r"Software\Python") as k:
...     print(winreg.QueryValue(k, "3.10.1"))
...     print(winreg.QueryValue(k, None))
...
    Installed

>>> # Ветвь Software\Python не имеет значения
```

- `CreateKey(<Ветвь>, <Имя вложенной ветви>)` — создает в указанной ветви вложенную ветвь с заданным именем. Возвращает объект класса `PyHKEY`, представляющий созданную вложенную ветвь. Если вложенная ветвь с указанным именем уже существует, просто открывает и возвращает ее;
- `CreateKeyEx()` — аналогична `CreateKey()`, только позволяет дополнительно указать права доступа к созданной вложенной ветви. Формат вызова:

```
CreateKeyEx(<Ветвь>, <Имя вложенной ветви>[, reserved=0][,
                                           access=KEY_WRITE])
```

Параметр `reserved` зарезервирован для использования в будущих версиях Windows, в качестве значения ему всегда дается число 0. Параметр `access` задает права доступа к создаваемой ветви в формате, описанном в *разд. 17.1.1*. Пример:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...                     r"Software\Python") as k:
...     winreg.CreateKeyEx(k, "3.10.0", access=winreg.KEY_READ)
...     winreg.CreateKeyEx(k, "3.9.0", access=winreg.KEY_READ)
```

- `SetValueEx()` — добавляет в указанную ветвь элемент с заданными именем, типом и значением. Если такой элемент уже существует, записывает в него новое значение. Формат функции:

```
SetValueEx(<Ветвь>, <Имя элемента>, <Зарезервирован>,
                    <Обозначение типа>, <Значение элемента>)
```

Имя элемента указывается в виде строки. Параметру `<Зарезервирован>` всегда следует давать в качестве значения число 0. Обозначение типа указывается в виде значения одной из следующих переменных:

- `REG_SZ` — строка;
- `REG_EXPAND_SZ` — строка со ссылками на переменные окружения;

- REG_MULTI_SZ — список строк;
- REG_DWORD или REG_DWORD_LITTLE_ENDIAN — 32-разрядное целое число (байты записываются в порядке «от младшего к старшему»);
- REG_DWORD_BIG_ENDIAN — 32-разрядное целое число (байты записываются в порядке «от старшего к младшему»);
- REG_QWORD или REG_QWORD_LITTLE_ENDIAN — 64-разрядное целое число (байты записываются в порядке «от младшего к старшему»);
- REG_BINARY — последовательность байтов типа bytes или bytearray;
- REG_LINK — символическая ссылка Unicode;
- REG_RESOURCE_LIST — список ресурсов драйвера устройства;
- REG_FULL_RESOURCE_DESCRIPTOR — настройки устройства;
- REG_RESOURCE_REQUIREMENTS_LIST — список ресурсов устройства;
- REG_NONE — отсутствие явно указанного типа.

Пример:

```
>>> versions = ["3.10.1", "3.10.0", "3.9.0"]
>>> bn = bytes("Python", "cp1251")
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...             r"Software\Python", access=winreg.KEY_SET_VALUE) as k:
...     winreg.SetValueEx(k, "Count", 0, winreg.REG_DWORD, 3)
...     winreg.SetValueEx(k, "Versions", 0, winreg.REG_MULTI_SZ,
...                                         versions)
...     winreg.SetValueEx(k, "BName", 0, winreg.REG_BINARY, bn)
```

- SetValue() — записывает во вложенную ветвь с заданным именем, находящуюся в указанной ветви, заданное значение с указанным типом. Если указанная вложенная ветвь не существует, создает ее. Формат функции:

```
SetValue(<Ветвь>, <Имя вложенной ветви>, <Обозначение типа>,
                                         <Значение вложенной ветви>)
```

Имя вложенной ветви указывается в виде строки, обозначение типа — в виде одной из переменных, описанных ранее. Пример:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...             r"Software\Python", access=winreg.KEY_SET_VALUE) as k:
...     winreg.SetValue(k, "3.10.0", winreg.REG_SZ, "Not installed")
```

- DeleteValue(<Ветвь>, <Имя элемента>) — удаляет из заданной ветви элемент с указанным именем:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...             r"Software\Python", access=winreg.KEY_SET_VALUE) as k:
...     winreg.DeleteValue(k, "Count")
```

- DeleteKey(<Ветвь>, <Имя вложенной ветви>) — удаляет из указанной ветви вложенную ветвь с заданным именем. Удаляемая ветвь не должна содержать вложенных ветвей — в противном случае при попытке ее удалить возникнет ошибка. Пример:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...             r"Software\Python", access=winreg.KEY_SET_VALUE) as k:
...     winreg.DeleteKey(k, "3.9.0")
```

- `DeleteKeyEx()` — то же самое, что и `DeleteKey()`, только дополнительно позволяет задать редакцию реестра (64- или 32-разрядную), из которой следует удалить вложенную ветвь. Формат функции:

```
DeleteKeyEx(<Ветвь>, <Имя вложенной ветви>[, access=KEY_WOW64_64KEY][,
                                            reserved=0])
```

Редакция реестра задается параметром `access`: значение переменной `KEY_WOW64_64KEY` указывает 64-разрядную редакцию реестра, а значение переменной `KEY_WOW64_32KEY` — 32-разрядную. Параметр `reserved` всегда должен иметь значение 0. Пример:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...              r"Software\Python", access=winreg.KEY_SET_VALUE) as k:
...     winreg.DeleteKeyEx(k, "3.9.0", access=winreg.KEY_WOW64_32KEY)
```

> ***Внимание!***
>
> Функция `DeleteKeyEx()` работает только в 64-разрядных редакциях Windows. При попытке вызвать ее в 32-разрядной редакции будет сгенерировано исключение `NotImplementedError`.

Все операции записи данных в какую-либо ветвь реестра, создания, удаления элементов и вложенных ветвей буферизуются в оперативной памяти для повышения производительности. Перенос измененной ветви реестра из памяти на диск выполняется только после закрытия этой ветви или вызова функции `FlushKey()`;

- `FlushKey(<Ветвь>)` — принудительно переносит хранящуюся в буфере исправленную копию заданной ветви на диск;
- `QueryInfoKey(<Ветвь>)` — возвращает сведения о заданной ветви в виде кортежа из трех элементов: количества вложенных ветвей, количества элементов и времени последнего изменения содержимого заданной ветви, которое представлено количеством сотен наносекунд, прошедших с 1 января 1601 года:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...                     r"Software\Python") as k:
...     print(winreg.QueryInfoKey(k))
(2, 3, 132882022669686509)
```

17.1.3. Перебор элементов и вложенных ветвей реестра

Функции, перебирающие элементы и вложенные ветви, в качестве первого параметра принимают объект открытой ветви, относящийся к классу `PyHKEY` и возвращенный функцией `OpenKey()` или `OpenKeyEx()` (см. *разд. 17.1.1*). При возникновении ошибки они генерируют исключение `OSError` или одно из производных исключений. Python предоставляет две функции такого рода:

- `EnumValue(<Ветвь>, <Индекс>)` — возвращает сведения об элементе указанной ветви с заданным индексом (нумерация элементов начинается с 0). Возвращаемые сведения представляют собой кортеж из трех элементов: имени элемента, его значения и целочисленного обозначения типа значения. При попытке получить элемент с несуществующим индексом генерируется исключение `OSError`. Пример:

```
>>> import winreg, itertools
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...                     r"Software\Python") as k:
```

```
...     for i in itertools.count():
...         try:
...             print(winreg.EnumValue(k, i))
...         except OSError:
...             break
...
('Version', '3.10.1', 1)
('Versions', ['3.10.1', '3.10.0', '3.9.0'], 7)
('BName', b'Python', 3)
```

- `EnumKey(<Ветвь>, <Индекс>)` — возвращает имя вложенной ветви с заданным индексом (нумерация элементов начинается с 0), находящейся в указанной ветви. При попытке получить вложенную ветвь с несуществующим индексом генерируется исключение `OSError`. Пример:

```
>>> with winreg.OpenKey(winreg.HKEY_CURRENT_USER,
...                     r"Software\Python") as k:
...     for i in itertools.count():
...         try:
...             print(winreg.EnumKey(k, i))
...         except OSError:
...             break
...
3.10.0
3.10.1
```

17.2. Получение путей к системным каталогам

Для получения путей к системным каталогам Windows (таким как Документы, Изображения, Рабочий стол, Program Files) удобно использовать дополнительную библиотеку winpath. Перед использованием ее следует установить с помощью утилиты pip (описана в *разд. 1.8*), набрав в консоли команду:

```
pip install winpath
```

> ***Примечание***
>
> Полное руководство по библиотеке winpath можно найти на странице: **https://pypi.org/project/winpath/**.

Чтобы получить пути к системным каталогам, следует использовать следующие функции, определенные в модуле `winpath`:

- `get_my_documents()` — каталог Документы текущего пользователя;
- `get_my_pictures()` — каталог Изображения текущего пользователя;
- `get_desktop()` — каталог Рабочий стол текущего пользователя;
- `get_recent()` — каталог недавних документов текущего пользователя;
- `get_programs()` — каталог с содержимым меню **Пуск | Программы** текущего пользователя;
- `get_startup()` — каталог меню **Пуск | Программы | Автозагрузка** текущего пользователя;

- `get_admin_tools()` — каталог меню **Пуск | Программы | Средства администрирования Windows** текущего пользователя;
- `get_local_appdata()` — каталог локальных настроек текущего пользователя (<каталог пользовательского профиля>\AppData\Local);
- `get_appdata()` — каталог переносимых настроек текущего пользователя (<каталог пользовательского профиля>\AppData\Roaming);
- `get_common_documents()` — каталог Документы, общий для всех пользователей;
- `get_common_admin_tools()` — каталог меню **Пуск | Программы | Средства администрирования Windows** всех пользователей;
- `get_common_appdata()` — каталог ProgramData;
- `get_program_files()` — каталог Program Files;
- `get_program_files_common()` — каталог Program Files\Common Files;
- `get_windows()` — каталог Windows;
- `get_system()` — каталог Windows\System32.

Примеры:

```
>>> import winpath
>>> winpath.get_my_documents()
    'D:\\Data\\Документы'
>>> winpath.get_recent()
    'C:\\Users\\vlad\\AppData\\Roaming\\Microsoft\\Windows\\Recent'
>>> winpath.get_programs()
    'C:\\Users\\vlad\\AppData\\Roaming\\Microsoft\\Windows\\Start Menu\\Programs'
>>> winpath.get_local_appdata()
    'C:\\Users\\vlad\\AppData\\Local'
>>> winpath.get_appdata()
    'C:\\Users\\vlad\\AppData\\Roaming'
>>> winpath.get_common_documents()
    'C:\\Users\\Public\\Documents'
>>> winpath.get_common_appdata()
    'C:\\ProgramData'
>>> winpath.get_program_files()
    'C:\\Program Files'
>>> winpath.get_program_files_common()
    'C:\\Program Files\\Common Files'
```

17.3. Создание ярлыков

Для создания ярлыков в меню **Пуск** и (или) на рабочем столе отлично подходит дополнительная библиотека pyshortcuts. Устанавливается она набором в консоли следующей команды:

```
pip install pyshortcuts
```

Для работы она требует наличия дополнительной библиотеки pywin32, которая будет установлена автоматически.

После установки библиотеки следует перезапустить интерпретатор, работающий в интерактивном режиме, если на момент установки он был запущен.

> ***Примечание***
>
> Полная документация по библиотеке pyshortcuts находится по интернет-адресу: **https://github.com/newville/pyshortcuts**.

Создание ярлыка, указывающего на заданный файл программы, выполняет функция `make_shortcut()` из модуля `pyshortcuts`. Формат вызова функции:

```
make_shortcuts(<Путь к файлу программы>[, name=None][,
               description=None][, icon=None][, folder=None][,
               terminal=True][, desktop=True][, startmenu=True])
```

Путь к файлу программы задается в виде строки. Остальные параметры указывают:

- `name` — имя ярлыка (если `None`, будет использовано имя файла программы из первого параметра);
- `description` — описание ярлыка (если `None`, будет использовано имя ярлыка);
- `icon` — путь к файлу со значком (если `None`, будет использован значок с логотипом Python, поставляемый в составе библиотеки);
- `folder` — относительный путь, указанный относительно каталога Рабочий стол, к каталогу, в котором будет создан ярлык (если `None`, ярлык будет создан непосредственно в каталоге Рабочий стол);
- `terminal` — если `True`, программа, на которую указывает ярлык, будет запущена в консоли, если `False` — нет. У программ с графическим интерфейсом этому параметру следует дать значение `False`;
- `desktop` — если `True`, ярлык будет создан на рабочем столе, если `False` — нет;
- `startmenu` — если `True`, ярлык будет создан в меню **Пуск**, если `False` — нет.

Функция возвращает объект класса `Shortcut`, определенного в модуле `pyshortcuts`, который представляет созданный ярлык. Обычно возвращаемый функцией результат игнорируют. Пример:

```
from pyshortcuts import make_shortcut
make_shortcut(r"c:\book\program.py", name="Программа",
              description="Небольшая учебная программа", startmenu=False)
```

ЧАСТЬ II

Библиотека PyQt 6

ГЛАВА 18

Введение в PyQt 6

PyQt 6 — это дополнительная библиотека, позволяющая разрабатывать на Python программы с графическим интерфейсом (*оконные программы*).

Для непосредственного вывода элементов графического интерфейса (окон, кнопок, полей ввода, списков и др.) эта библиотека использует популярный программный фреймворк Qt. Последний автоматически устанавливается одновременно с PyQt.

> ***Примечание***
>
> Описание библиотеки PyQt приведено по интернет-адресу: **https://www.riverbankcomputing.com/**. Полная документация по фреймворку Qt 6 находится на сайте: **https://doc.qt.io/qt-6/index.html**.

18.1. Установка PyQt 6

Для установки дополнительной библиотеки (и всех необходимых ей для работы библиотек) достаточно набрать в консоли команду:

```
pip install PyQt6
```

> ***Внимание!***
>
> Эта команда устанавливает библиотеку в базовой комплектации. Дополнительные компоненты (в частности, веб-браузер) и средства разработчика (такие как программа Qt Designer) следует установить отдельно.

Чтобы проверить правильность установки, выведем версии PyQt и Qt:

```
>>> from PyQt6 import QtCore
>>> QtCore.PYQT_VERSION_STR
    '6.2.3'
>>> QtCore.QT_VERSION_STR
    '6.2.3'
```

18.2. Первая оконная программа

Напишем программу, выводящую окно с надписью «Привет, мир!» и кнопкой закрытия окна (рис. 18.1).

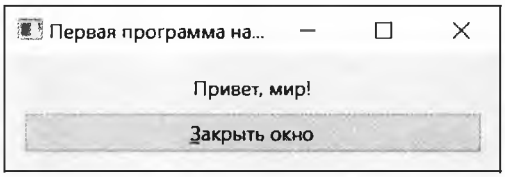


Рис. 18.1. Результат выполнения листинга 18.1

Листинг 18.1. Первая программа на PyQt

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Первая программа на PyQt")
window.resize(300, 70)
label = QtWidgets.QLabel("<center>Привет, мир!</center>")
btnQuit = QtWidgets.QPushButton("&Закрыть окно")
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
vbox.addWidget(btnQuit)
window.setLayout(vbox)
btnQuit.clicked.connect(app.quit)
window.show()
sys.exit(app.exec())
```

Файлу с оконной программой следует дать расширение pyw, чтобы при ее запуске не выводилось окно консоли.

18.3. Структура PyQt-программы

Рассмотрим код из листинга 18.1 построчно.

- В первой строке импортируется модуль `QtWidgets`, который содержит классы, реализующие компоненты графического интерфейса: окна, надписи, кнопки, текстовые поля и др.
- Во второй строке импортируется модуль `sys`, из которого нам потребуется список параметров, переданных в командной строке (`argv`), и функция `exit()`, завершающая выполнение программы.
- Инструкция:

  ```
  app = QtWidgets.QApplication(sys.argv)
  ```

  создает объект программы на основе класса `QApplication`. Конструктор этого класса принимает список параметров, переданных в командной строке.

  Получить объект программы в любом месте ее кода можно вызовом статического метода `instance()` класса `QApplication`. Например, вывести список параметров, переданных в командной строке, можно так:

  ```
  print(QtWidgets.QApplication.instance().argv())
  ```

- Следующая инструкция:

```
window = QtWidgets.QWidget()
```

создает объект окна в виде объекта класса QWidget. Этот класс наследуют практически все классы, реализующие компоненты графического интерфейса. Поэтому любой компонент, не имеющий родителя, выводится в своем собственном окне.

- Инструкция:

```
window.setWindowTitle("Первая программа на PyQt")
```

задает текст, который будет выводиться в заголовке окна, для чего используется метод `setWindowTitle()`.

- Очередная инструкция:

```
window.resize(300, 70)
```

задает минимальные размеры клиентской области окна (размеры заголовка и толщина границ окна не учитываются). В первом параметре метода `resize()` указывается ширина окна, а во втором параметре — его высота. Следует помнить, что эти размеры являются рекомендацией, — т. е. если компоненты не помещаются в окне, оно будет увеличено.

- Инструкция:

```
label = QtWidgets.QLabel("<center>Привет, мир!</center>")
```

создает объект надписи. Текст надписи задается в качестве параметра в конструкторе класса `QLabel`. С помощью HTML-тега `<center>` производится выравнивание текста по центру компонента. Возможность использования HTML-тегов и CSS-атрибутов является отличительной чертой библиотеки PyQt (например, внутри надписи можно вывести таблицу или графическое изображение).

- Следующая инструкция:

```
btnQuit = QtWidgets.QPushButton("&Закрыть окно")
```

создает объект кнопки. Текст, который будет отображен на кнопке, задается в качестве параметра в конструкторе класса `QPushButton`. Символ `&` перед буквой `З` задает клавишу быстрого доступа. Если нажать одновременно клавишу <Alt> и клавишу с буквой, перед которой в строке указан символ `&`, то кнопка сработает.

- Инструкция:

```
vbox = QtWidgets.QVBoxLayout()
```

создает вертикальный контейнер. Все компоненты, добавляемые в этот контейнер, будут располагаться по вертикали сверху вниз в порядке добавления, при этом размеры добавленных компонентов будут подогнаны под размеры контейнера. При изменении размеров контейнера будет произведено изменение размеров всех компонентов.

- В следующих двух инструкциях:

```
vbox.addWidget(label)
vbox.addWidget(btnQuit)
```

с помощью метода `addWidget()` контейнера производится добавление в него созданных ранее надписи и кнопки. Добавленные в контейнер компоненты станут его *потомками*, а сам контейнер — *родителем* добавленных в него компонентов.

- Инструкция:

```
window.setLayout(vbox)
```

помещает контейнер в окно с помощью метода `setLayout()`. В результате контейнер станет потомком окна (а окно — родителем контейнера).

- Инструкция:

```
btnQuit.clicked.connect(app.quit)
```

назначает обработчик сигнала `clicked()` кнопки, активизирующегося при возникновении события нажатия кнопки.

Событие — это уведомление о том, что в программе что-либо произошло: пользователь щелкнул мышью, нажал клавишу на клавиатуре, переместил окно и др.

Сигнал — это особый объект, соответствующий событию определенного типа и генерируемый каждый раз, когда возникает событие, относящееся к этому типу.

Обработчик сигнала — это метод (или функция), связанный с сигналом и вызываемый при его генерировании. Обработчик назначается сигналу вызовом у последнего метода `connect()`. Метод, используемый в качестве обработчика, называется *слотом*.

Сигнал `clicked()`, генерируемый при возникновении события нажатии кнопки, доступен через одноименный атрибут класса кнопки. Обработчик представляет собой метод `quit()` объекта программы, немедленно завершающий ее работу.

Отметим, что не все события связаны со слотами. Часть событий обрабатывается особыми специальными методами, называемых *обработчиками событий*. Подробнее об обработке событий и сигналов будет рассказано в *главе 20*.

- Очередная инструкция:

```
window.show()
```

выводит на экран окно и все компоненты, которые мы ранее в него добавили.

- И, наконец, последняя инструкция:

```
sys.exit(app.exec())
```

запускает цикл обработки сигналов в программе.

Код, расположенный после вызова метода `exec()`, будет выполнен только после завершения работы программы, — поскольку результат выполнения метода `exec()` мы передаем функции `exit()`, дальнейшее выполнение программы будет прекращено, а код возврата — передан операционной системе.

18.4. ООП-стиль создания окна

Библиотека PyQt содержит несколько сотен классов. Иерархия их наследования имеет слишком большой размер, и привести ее в книге возможности нет. Тем не менее, чтобы увидеть зависимости, при описании каждого класса мы будем приводить иерархию его наследования. В качестве примера выведем базовые классы класса `QWidget`:

```
>>> from PyQt6 import QtWidgets
>>> QtWidgets.QWidget.__bases__
    (<class 'PyQt6.QtCore.QObject'>, <class 'PyQt6.QtGui.QPaintDevice'>)
```

Как видно из этого примера, класс `QWidget` наследует два класса: `QObject` и `QPaintDevice`. Класс `QObject` является классом верхнего уровня, и его в PyQt наследует большинство классов. В свою очередь, класс `QWidget` является базовым для всех визуальных компонентов.

> ***Внимание!***
>
> В описании каждого класса PyQt будут приведены лишь атрибуты, методы, сигналы и слоты, определенные непосредственно в описываемом классе. Атрибуты, методы, сигналы и слоты базовых классов здесь не описываются — присутствуют лишь ссылки на соответствующие страницы документации.

Можно создавать новые классы компонентов на основе стандартных. В качестве примера переделаем код из листинга 18.1 и создадим окно в стиле объектно-ориентированного программирования (ООП), как показано в листинге 18.2.

Листинг 18.2. ООП-стиль создания окна

```
from PyQt6 import QtWidgets, QtCore

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Привет, мир!")
        self.label.setAlignment(QtCore.Qt.AlignmentFlag.AlignHCenter)
        self.btnQuit = QtWidgets.QPushButton("&Закрыть окно")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnQuit)
        self.setLayout(self.vbox)
        self.btnQuit.clicked.connect(QtWidgets.QApplication.instance().quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()    # Создаем объект окна
    window.setWindowTitle("ООП-стиль создания окна")
    window.resize(300, 70)
    window.show()          # Отображаем окно
    sys.exit(app.exec())   # Запускаем цикл обработки событий
```

Помимо уже знакомого модуля `QtWidgets` из пакета `PyQt6`, мы импортируем модуль `QtCore`, содержащий модуль `Qt`, в котором определено перечисление `AlignmentFlag`, задающее выравнивание текста.

Далее мы определяем класс `MyWindow`, который наследует класс `QWidget`:

```
class MyWindow(QtWidgets.QWidget):
```

Можно наследовать и другие классы, производные от `QWidget`, — например, `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При наследовании класса `QDialog` окно будет выравниваться по центру экрана (или по центру родительского окна) и иметь в заголовке только кнопки закрытия и вызова справки. Кроме того, можно наследовать класс

QMainWindow, который представляет главное окно программы с меню, панелями инструментов и строкой состояния. Наследование класса QMainWindow имеет свои отличия, которые мы рассмотрим в *главе 28*.

Конструктор класса MyWindow в качестве параметров принимает ссылки на текущий объект (self) и на родительский компонент (parent). Родительский компонент может отсутствовать, поэтому в определении конструктора параметру присваивается значение по умолчанию (None). Внутри конструктора вызывается конструктор базового класса, и ему передается ссылка на родительский компонент:

```
QtWidgets.QWidget.__init__(self, parent)
```

Далее в конструкторе создаются объекты надписи, кнопки и контейнера, компоненты добавляются в контейнер, а сам контейнер — в окно. Объекты надписи и кнопки сохраняются в атрибутах объекта, чтобы в дальнейшем этими объектами можно было управлять (например, изменять текст надписи) из методов класса. Если объекты не сохранить, то получить к ним доступ будет не так просто.

В предыдущем примере (см. листинг 18.1) мы выравнивали текст надписи с помощью HTML-тега. Здесь же использован метод setAlignment(), которому мы передали элемент AlignHCenter перечисления AlignmentFlag из модуля QtCore.Qt:

```
self.label.setAlignment(QtCore.Qt.AlignmentFlag.AlignHCenter)
```

В инструкции, назначающей обработчик сигнала clicked():

```
self.btnQuit.clicked.connect(QtWidgets.QApplication.instance().quit)
```

мы получаем доступ к объекту программы посредством рассмотренного в *разд. 18.3* статического метода instance() класса QApplication.

Создание объекта программы и объекта класса MyWindow производится внутри условия

```
if __name__ == "__main__":
```

Атрибут модуля __name__ будет содержать значение __main__ только в случае запуска модуля как главной программы. Если модуль импортировать, этот атрибут будет содержать другое значение. Поэтому весь последующий код создания объектов программы и окна выполняется только при запуске программы двойным щелчком на значке файла. Может возникнуть вопрос: зачем это нужно? Дело в том, что одним из преимуществ ООП-стиля программирования является повторное использование кода. Следовательно, можно импортировать модуль и использовать класс MyWindow в другой программе.

Рассмотрим эту возможность на примере, для чего сохраним код из листинга 18.2 в файле MyWindow.pyw, а затем создадим в той же папке еще один файл (например, с именем test.pyw) и вставим в него код из листинга 18.3.

Листинг 18.3. Повторное использование кода при ООП-стиле

```
from PyQt6 import QtWidgets
import MyWindow

class MyDialog(QtWidgets.QDialog):
    def __init__(self, parent=None):
        QtWidgets.QDialog.__init__(self, parent)
        self.myWidget = MyWindow.MyWindow()
```

```
        self.myWidget.vbox.setContentsMargins(0, 0, 0, 0)
        self.button = QtWidgets.QPushButton("&Изменить надпись")
        mainBox = QtWidgets.QVBoxLayout()
        mainBox.addWidget(self.myWidget)
        mainBox.addWidget(self.button)
        self.setLayout(mainBox)
        self.button.clicked.connect(self.on_clicked)
    def on_clicked(self):
        self.myWidget.label.setText("Новая надпись")
        self.button.setDisabled(True)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyDialog()
    window.setWindowTitle("Преимущество ООП-стиля")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec())
```

Запустим файл test.pyw. На экране появится окно с надписью и двумя кнопками (рис. 18.2). По нажатии на кнопку **Изменить надпись** производится изменение текста надписи, и кнопка делается неактивной. Нажатие кнопки **Закрыть окно** будет по-прежнему завершать работу программы.

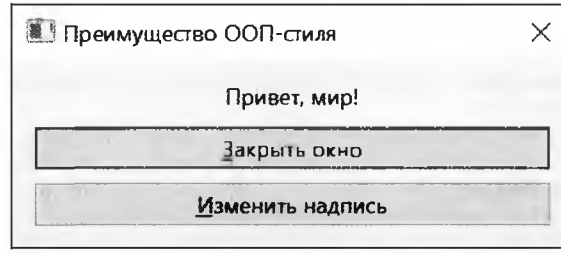


Рис. 18.2. Результат выполнения листинга 18.3

В этом примере мы создали класс `MyDialog`, производный от `QDialog`. Внутри конструктора мы создаем объект класса `MyWindow` и сохраняем его в атрибуте `myWidget`:

```
self.myWidget = MyWindow.MyWindow()
```

С его помощью позже мы получим доступ ко всем атрибутам класса `MyWindow`. Например, в следующей строке произведем изменение отступа между границами контейнера и границами соседних элементов:

```
self.myWidget.vbox.setContentsMargins(0, 0, 0, 0)
```

Далее в конструкторе создаются кнопки и контейнер, объект класса `MyWindow` и кнопка добавляются в контейнер, а сам контейнер помещается в окно.

Инструкция:

```
self.button.clicked.connect(self.on_clicked)
```

назначает обработчик нажатия кнопки. В качестве параметра указывается ссылка на метод `on_clicked()`, внутри которого производится изменение текста надписи (с помощью метода `setText()`), и кнопка делается неактивной (с помощью метода `setDisabled()`). Внутри метода `on_clicked()` доступна ссылка на текущий объект `self`, через который можно получить доступ к атрибутам окна.

Если модуль `MyWindow.pyw` запускается непосредственно как главная программа, выполнится код внутри инструкции ветвления (см. листинг 18.2):

```
if __name__ == "__main__":
```

и создаст объекты программы и окна. Если же этот модуль импортируется, объекты программы и окна не создаются.

В некоторых случаях использование ООП-стиля является обязательным. Например, чтобы обработать нажатия клавиш, необходимо наследовать какой-либо класс и переопределить в подклассе метод с предопределенным названием. Какие методы необходимо переопределять, мы рассмотрим при изучении обработки событий.

18.5. Создание окон с помощью программы Qt Designer

Программа Qt Designer позволяет создавать окна для PyQt-программ визуально, размещая компоненты на них мышью.

К сожалению, при попытке установить Qt Designer с помощью утилиты pip в Python 3.10 возникает ошибка. Однако дистрибутив этой программы в виде обычного EXE-файла доступен на странице **https://build-system.fman.io/qt-designer-download**, а процесс установки тривиален и не вызовет проблем.

Запустить установленную программу Qt Designer можно выбором в меню **Пуск** пункта **Qt Designer**.

18.5.1. Создание окон

После запуска Qt Designer откроется его главное окно, а поверх него — диалоговое окно **New Form** (Новая форма) (рис. 18.3). В списке **templates\forms** (шаблоны\формы) этого окна выберем пункт **Widget** (Обычное окно) и нажмем кнопку **Create** (Создать). В главном окне появится пустая заготовка для создания окна (в терминологии Qt Designer — *форма*), на которую с помощью мыши можно перетаскивать компоненты из панели **Widget Box** (рис. 18.4).

В качестве примера добавим на форму надпись и кнопку. Для этого на панели **Widget Box** в группе **Display Widgets** щелкнем левой кнопкой мыши на пункте **Label** и, не отпуская кнопку мыши, перетащим компонент на форму. Затем проделаем аналогичную операцию с компонентом **Push Button**, находящимся в группе **Buttons**, и разместим его ниже надписи. Теперь выделим одновременно надпись и кнопку, щелкнем правой кнопкой мыши на любом компоненте и в контекстном меню выберем пункт **Lay out | Lay Out Vertically**. Чтобы компоненты занимали всю область формы, щелкнем правой кнопкой мыши на свободном месте формы и в контекстном меню выберем пункт **Lay out | Lay Out Horizontally**.

Далее изменим некоторые свойства окна. Для этого в панели **Object Inspector** (рис. 18.5) выделим первый пункт (**Form**), перейдем в панель **Property Editor** (рис. 18.6), найдем свойство

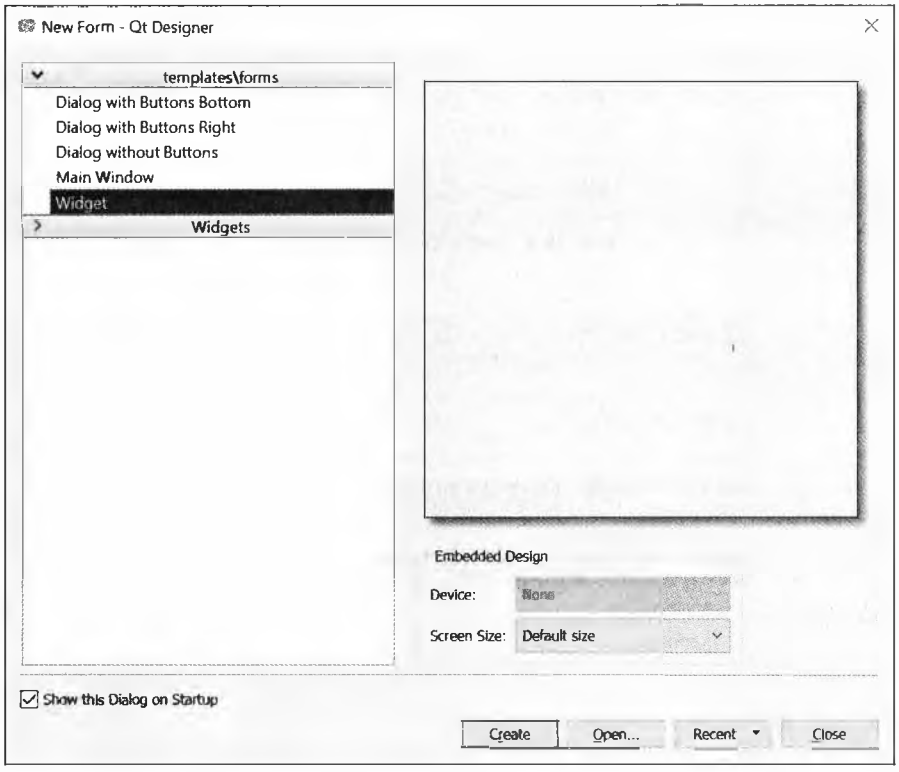


Рис. 18.3. Диалоговое окно **New Form**

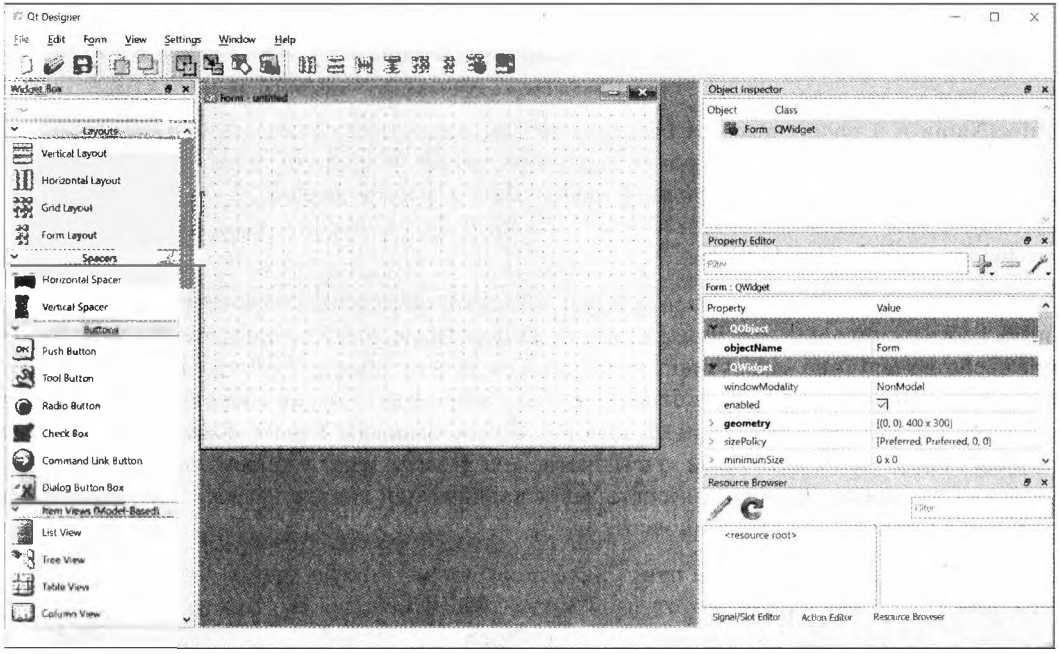


Рис. 18.4. Главное окно программы Qt Designer с пустой формой

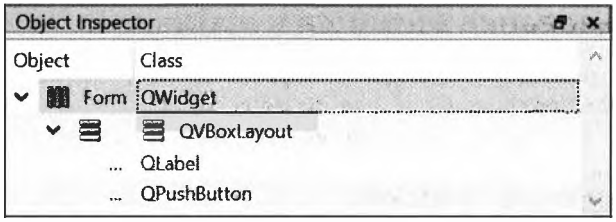


Рис. 18.5. Панель **Object Inspector**

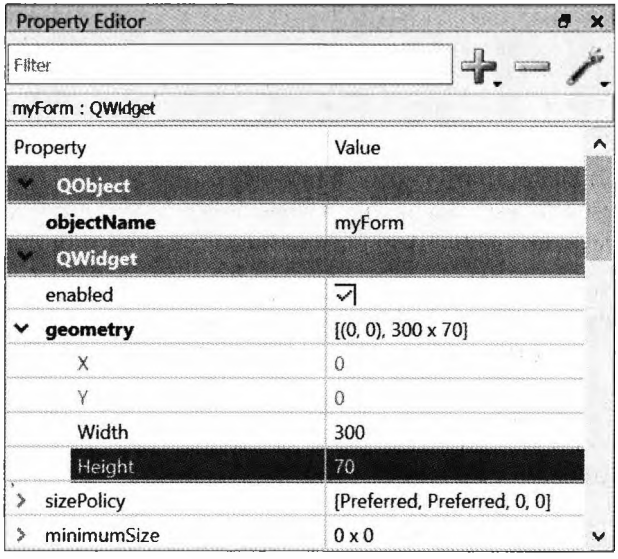


Рис. 18.6. Панель **Property Editor**

objectName и в поле справа от свойства введем новое имя класса формы: `MyForm`. Затем найдем свойство **geometry**, щелкнем мышью на значке ⮟ слева от него, чтобы отобразить скрытые свойства, и зададим ширину равной 300, а высоту равной 70, — размеры формы автоматически изменятся. Указать текст, который будет отображаться в заголовке окна, позволяет свойство **windowTitle**.

Чтобы изменить свойства надписи, следует выделить компонент с помощью мыши или выбрать соответствующий ему пункт в панели **Object Inspector**. Для примера изменим значение свойства **text** (оно задает текст надписи). После чего найдем свойство **alignment**, щелкнем мышью на значке ⮟ слева от него, чтобы отобразить скрытые свойства, и укажем для свойства **Horizontal** значение `AlignHCenter`. Теперь выделим кнопку и изменим значение свойства **objectName** на `btnQuit`, а в свойстве **text** укажем текст надписи, которая будет выводиться на кнопке. (Кстати, изменить текст надписи или кнопки также можно, выполнив двойной щелчок мышью на компоненте.)

Закончив, выберем в меню **File** пункт **Save** и сохраним готовую форму в файле MyForm.ui. При необходимости внести в этот файл какие-либо изменения его можно открыть в программе Qt Designer, выбрав в меню **File** пункт **Open**.

18.5.2. Использование UI-файла в программе

Описание формы, созданной в Qt Designer, сохраняется в формате XML в файле с расширением ui. Чтобы использовать такой файл в PyQt-программе, следует воспользоваться модулем `uic`, который входит в состав библиотеки PyQt. Предварительно следует подключить этот модуль с помощью инструкции:

```
from PyQt6 import uic
```

Для загрузки UI-файла и формирования на его основе окна предназначена функция `loadUi()`, определенная в указанном модуле. Формат функции:

```
loadUi(<Путь к UI-файлу>[, <Объект окна>])
```

Вторым параметром можно указать готовый объект окна, в котором будет сформирован интерфейс, описанный в UI-файле с указанным путем. Если второй параметр не указан, функция сама создаст новое окно и вернет ссылку на его объект в качестве результата.

В объекте окна будут созданы атрибуты, хранящие объекты компонентов, у которых в программе Qt Designer были указаны имена в свойстве **objectName**. Имена этих атрибутов будут совпадать с именами соответствующих компонентов.

Два варианта использования функции `loadUi()` для формирования окна на основе готового описания формы: без предварительного создания «пустого» объекта окна и с его предварительным созданием — показаны в листингах 18.4 и 18.5.

Листинг 18.4. Использование функции `loadUi()`, вариант 1

```
from PyQt6 import QtWidgets, uic
import sys
app = QtWidgets.QApplication(sys.argv)
window = uic.loadUi("MyForm.ui")
window.btnQuit.clicked.connect(app.quit)
window.show()
sys.exit(app.exec())
```

Листинг 18.5. Использование функции `loadUi()`, вариант 2

```
from PyQt6 import QtWidgets, uic

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        uic.loadUi("MyForm.ui", self)
        self.btnQuit.clicked.connect(QtWidgets.QApplication.instance().quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Загрузить UI-файл позволяет также функция `loadUiType(<Путь к UI-файлу>)` из того же модуля `uic`. Она возвращает кортеж из двух элементов: ссылки на класс формы и ссылки на базовый класс. Получив класс окна, мы можем создать на его основе множество объектов-окон. После создания объекта окна на основе полученного класса необходимо вызвать у этого объекта метод `setupUi(<Объект окна>)` и передать ему ссылку на этот же объект (листинг 18.6).

Листинг 18.6. Использование функции `loadUiType()`, вариант 1

```
from PyQt6 import QtWidgets, uic

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        Form, Base = uic.loadUiType("MyForm.ui")
        self.ui = Form()
        self.ui.setupUi(self)
        self.ui.btnQuit.clicked.connect(QtWidgets.QApplication.instance().quit)
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Загрузить UI-файл можно и вне класса, после чего при определении класса окна указать полученный от функции `loadUiType()` класс вторым в списке наследования, — чтобы определяемый класс унаследовал все его методы (листинг 18.7).

Листинг 18.7. Использование функции `loadUiType()`, вариант 2

```
from PyQt6 import QtWidgets, uic

Form, Base = uic.loadUiType("MyForm.ui")
class MyWindow(QtWidgets.QWidget, Form):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.btnQuit.clicked.connect(QtWidgets.QApplication.instance().quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

18.5.3. Преобразование UI-файла в модуль Python

На основе описания формы, хранящегося в UI-файле, можно сгенерировать программный код на языке Python, сохранив его в отдельном модуле. Для этого служит консольная утилита pyuic6, устанавливаемая в составе PyQt. Команда для ее запуска имеет такой формат:

```
pyuic6 <Путь к UI-файлу> --output|-o <Путь к генерируемому модулю> [--execute|x]
```

При указании командного ключа `--output` или `-x` в генерируемый модуль вставляется код, который создает и выводит окно, созданное на основе формы, в случае запуска этого модуля на выполнение.

Запустим консоль, перейдем в каталог, в котором находится UI-файл MyForm.ui, и отдадим команду:

```
pyuic6 MyForm.ui -o ui_MyForm.pyw
```

В результате будет создан модуль ui_MyForm.py, содержащий определение класса `Ui_MyForm` с методами `setupUi()` и `retranslateUi()`. При использовании процедурного стиля программирования следует создать объект «пустого» окна, объект класса формы и вызвать у последнего метод `setupUi()`, передав ему ссылку на созданный ранее объект окна (листинг 18.8).

Листинг 18.8. Использование сгенерированного класса формы, вариант 1

```
from PyQt6 import QtWidgets
import sys, ui_MyForm
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
ui = ui_MyForm.Ui_MyForm()
ui.setupUi(window)
ui.btnQuit.clicked.connect(QtWidgets.QApplication.instance().quit)
window.show()
sys.exit(app.exec())
```

При использовании ООП-стиля программирования следует создать объект класса формы и вызвать у него метод `setupUi()`, передав ему ссылку на этот же объект (листинг 18.9).

Листинг 18.9. Использование сгенерированного класса формы, вариант 2

```
from PyQt6 import QtWidgets
import ui_MyForm

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.ui = ui_MyForm.Ui_MyForm()
        self.ui.setupUi(self)
        self.ui.btnQuit.clicked.connect(QtWidgets.QApplication.instance().quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
```

```
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Класс формы можно указать во втором параметре в списке наследования — в этом случае он унаследует все методы класса формы (листинг 18.10).

Листинг 18.10. Использование сгенерированного класса формы, вариант 3

```
from PyQt6 import QtWidgets
import ui_MyForm

class MyWindow(QtWidgets.QWidget, ui_MyForm.Ui_MyForm):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.btnQuit.clicked.connect(QtWidgets.QApplication.instance().quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Создавать формы в программе Qt Designer очень удобно. Тем не менее, чтобы полностью овладеть программированием на PyQt, необходимо уметь писать код вручную. Поэтому в оставшейся части книги мы больше не станем задействовать программу Qt Designer.

18.6. Модули PyQt 6

В состав библиотеки PyQt 6 входит множество модулей, объединенных в пакет `PyQt6`. Упомянем самые важные из них:

- `QtCore` — содержит классы, не связанные с реализацией графического интерфейса. От этого модуля зависят все остальные модули;
- `QtGui` — содержит классы, реализующие низкоуровневую работу с оконными элементами, обработку сигналов, вывод двухмерной графики, текста и др.;
- `QtWidgets` — содержит классы, реализующие компоненты графического интерфейса: окна, надписи, кнопки, поля ввода, списки и др.;
- `QtWebEngineCore` — включает низкоуровневые классы для отображения веб-страниц;
- `QtWebEngineWidgets` — реализует высокоуровневые компоненты графического интерфейса, предназначенные для вывода веб-страниц и использующие модуль `QtWebEngineCore`;
- `QtMultimedia` — включает низкоуровневые классы для работы с мультимедиа;
- `QtMultimediaWidgets` — реализует высокоуровневые компоненты для работы с мультимедиа, использующие модуль `QtMultimedia`;
- `QtPrintSupport` — содержит классы, реализующие печать и предварительный просмотр документов;

- `QtSql` — включает средства для работы с базами данных, а также реализацию SQLite;
- `QtSvg` — служит для работы с векторной графикой (SVG);
- `QtNetwork` — содержит классы, предназначенные для работы с сетью;
- `QtXml` и `QtXmlPatterns` — предназначены для обработки XML;
- `QtHelp` — содержат инструменты для создания интерактивных справочных систем.

В этой книге не описываются все упомянутые модули — чтобы получить информацию по нерассматриваемым здесь модулям, обращайтесь к соответствующей документации.

18.7. Управление циклом обработки событий

После вывода на экран главного окна программы и выполнения всевозможных подготовительных действий программа ожидает возникновения первого события. В *разд. 18.3* говорилось, что события различных типов возникают при нажатии кнопки, занесении значения в поле ввода, щелчке мышью, нажатии клавиши, перемещении окна и др. Дождавшись события, программа вызывает назначенный ему обработчик (либо обработчик сигнала, соответствующего событию, либо обработчик события) и после его выполнения начинает ждать возникновения следующего события и т. д.

Можно сказать, что программа после запуска вводится в бесконечный цикл, в теле которого выполняется обработка возникающих событий. Он носит название *цикла обработки событий*. В том же *разд. 18.3* сообщалось, что этот цикл запускается вызовом метода `exec()` объекта программы.

Цикл обработки событий автоматически прерывается после закрытия последнего открытого окна программы. С помощью статического метода `setQuitOnLastWindowClosed()` класса `QApplication` это поведение можно изменить.

Чтобы завершить работу программы, необходимо вызвать слот `quit()` или — в теле обработчика какого-либо события — метод `exit([returnCode=0])` класса `QApplication`. В параметре `returnCode` метода `exit()` указывается код возврата.

Если внутри обработчика события выполняется длительная операция, программа перестает реагировать на события. В качестве примера изобразим длительный процесс с помощью функции `sleep()` из модуля `time` (листинг 18.11).

Листинг 18.11. Выполнение длительной операции

```
from PyQt6 import QtWidgets
import sys, time

def on_clicked():
    time.sleep(10) # "Засыпаем" на 10 секунд

app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Запустить процесс")
button.resize(200, 40)
button.clicked.connect(on_clicked)
button.show()
sys.exit(app.exec())
```

В этом примере по нажатии кнопки **Запустить процесс** вызывается функция `on_clicked()`, внутри которой выполнение программы, в том числе работа цикла обработки событий, приостанавливается на десять секунд. Попробуйте нажать кнопку, перекрыть окно другим окном, а затем заново его отобразить — вам не удастся это сделать, поскольку окно перестает реагировать на любые события, пока не истечет заданное время «засыпания». Программа просто зависнет.

Длительную операцию можно разбить на несколько этапов и по завершении каждого этапа принудительно запускать обработку накопившихся событий вызовом статического метода `processEvents()` класса `QApplication`. Формат вызова этого метода:

```
processEvents([<Типы обрабатываемых событий>[, <Время>]])
```

Первый параметр указывает типы событий, которые будут обработаны после вызова метода. Значение указывается в виде одного из следующих элементов перечисления `ProcessEventsFlag` из класса `QEventLoop`, определенного в модуле `QtCore`, или их комбинации через двоичный оператор `|`:

- `AllEvents` — все события;
- `ExcludeUserInputEvents` — все события, за исключением вызываемых действиями пользователя;
- `ExcludeSocketNotifiers` — все события, за исключением вызванных сетевой активностью;
- `WaitForMoreEvents` — ожидать возникновения событий, если накопившихся событий нет.

Второй параметр задает время, в течение которого будут обрабатываться накопившиеся события, в миллисекундах. Если заданное время еще не истекло, накопившихся событий нет и в первом параметре не указан элемент `WaitForMoreEvents`, метод завершит работу. Если второй параметр не указан, метод завершит работу сразу после обработки последнего из накопившихся событий.

Переделаем предыдущую программу, инсценировав с помощью цикла длительную операцию, которая выполняется 20 секунд (листинг 18.12).

Листинг 18.12. Использование метода `processEvents()`

```
from PyQt6 import QtWidgets, QtCore
import sys, time

def on_clicked():
    button.setDisabled(True)            # Делаем кнопку неактивной
    for i in range(1, 21):
        # Обрабатываем накопившиеся события всех типов в течение 1 с
        QtWidgets.QApplication.instance().processEvents(
                  QtCore.QEventLoop.ProcessEventsFlag.AllEvents, 1000)
        time.sleep(1)                   # "Засыпаем" на 1 секунду
        print("step -", i)
    button.setDisabled(False)           # Делаем кнопку активной

app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Запустить процесс")
```

```
button.resize(200, 40)
button.clicked.connect(on_clicked)
button.show()
sys.exit(app.exec())
```

В этом примере длительная операция разбита на одинаковые этапы, после выполнения каждого из которых выполняется принудительная обработка накопившихся событий. Теперь программа останется отзывчивой, хотя и с некоторой задержкой.

18.8. Многопоточные программы

Другой способ повысить отзывчивость программы — вынести выполнение длительной операции в другой поток. *Поток* можно рассматривать как отдельный процессор, сформированный программно операционной системой, выполняющий часть кода программы и не мешающий работе потока, в котором выполняется основной код программы (*основного потока*). Вся совокупность потоков, организуемая программой для работы, называется *процессом*, а программы, использующие более одного потока, — *многопоточными*.

В процессе можно запустить сразу произвольное количество независимых потоков. По возможности операционная система будет выполнять отдельные потоки на отдельных ядрах центрального процессора, что дополнительно повысит производительность. Завершение основного потока программы приведет к завершению работы всех потоков.

18.8.1. Потоки

Для создания потока в PyQt предназначен класс `QThread`, который определен в модуле `QtCore` и наследует класс `QObject`. Конструктор класса `QThread` имеет следующий формат:

```
QThread([parent=None])
```

Параметр `parent` задает родительский объект создаваемого потока. При уничтожении объекта-родителя поток также будет уничтожен.

Чтобы создать поток, следует определить класс, производный `QThread`, и вставить в него метод `run()`. Код, расположенный в методе `run()`, будет выполняться в отдельном потоке, а после завершения выполнения метода `run()` этот поток прекратит свое существование. Чтобы запустить поток, нужно создать объект потока и вызвать у него метод `start()` в следующем формате:

```
start([priority=Priority.InheritPriority])
```

Параметр `priority` задает приоритет выполнения потока по отношению к другим потокам в виде значения одного из следующих элементов перечисления `Priority` из класса `QThread`: `IdlePriority` (самый низкий приоритет), `LowestPriority`, `LowPriority`, `NormalPriority`, `HighPriority`, `HighestPriority`, `TimeCriticalPriority` (самый высокий приоритет) и `InheritPriority` (автоматический выбор приоритета — значение по умолчанию).

Следует учитывать, что при наличии потока с самым высоким приоритетом поток с самым низким приоритетом в некоторых операционных системах может быть проигнорирован.

Задать приоритет потока также позволяет метод `setPriority(<Приоритет Priority>)`. Узнать, какой приоритет использует запущенный поток, можно с помощью метода `priority()`.

После запуска потока генерируется сигнал `started()`, а после завершения — сигнал `finished()`. Назначив этим сигналам обработчики, можно контролировать состояние потока

из основного потока программы. Метод `isRunning()` потока возвращает значение `True`, если текущий поток выполняется, а метод `isFinished()` — значение `True`, если поток закончил выполнение.

Потоки выполняются внутри одного процесса и имеют доступ ко всем глобальным переменным. Однако из потока нельзя изменять что-либо в основном потоке программы (например, менять текст надписи). Для изменения данных в основном потоке нужно использовать сигналы. Внутри потока у нужного сигнала вызывается метод `emit()`, который, собственно, и генерирует его. В параметрах метода `emit()` можно указать данные, которые будут переданы обработчику сигнала. А внутри основного потока назначается обработчик этого сигнала, который и будет обновлять интерфейс программы.

Рассмотрим использование потоков на примере (листинг 18.13).

Листинг 18.13. Использование потоков

```
from PyQt6 import QtCore, QtWidgets

class MyThread(QtCore.QThread):
    mysignal = QtCore.pyqtSignal(str)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        for i in range(1, 21):
            self.sleep(3)              # "Засыпаем" на 3 секунды
            # Передача данных из потока через сигнал
            self.mysignal.emit("i = %s" % i)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignmentFlag.AlignHCenter)
        self.button = QtWidgets.QPushButton("Запустить процесс")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.mythread = MyThread()     # Создаем поток
        self.button.clicked.connect(self.on_clicked)
        self.mythread.started.connect(self.on_started)
        self.mythread.finished.connect(self.on_finished)
        self.mythread.mysignal.connect(self.on_change,
                            QtCore.Qt.ConnectionType.QueuedConnection)
    def on_clicked(self):
        self.button.setDisabled(True)  # Делаем кнопку неактивной
        self.mythread.start()          # Запускаем поток
    def on_started(self):              # Вызывается при запуске потока
        self.label.setText("Вызван метод on_started()")
    def on_finished(self):             # Вызывается при завершении потока
        self.label.setText("Вызван метод on_finished()")
        self.button.setDisabled(False) # Делаем кнопку активной
```

```
    def on_change(self, s):
        self.label.setText(s)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование потоков")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec())
```

Здесь мы создали класс `MyThread`, производный от `QThread`. В нем мы определили свой сигнал `mysignal`, для чего создали атрибут с таким же именем и занесли в него значение, возвращенное функцией `pyqtSignal()` из модуля `QtCore`. Функции `pyqtSignal()` мы передали в качестве параметра тип `str` (строка Python), тем самым указав PyQt, что вновь определенный сигнал будет принимать единственный параметр строкового типа:

```
mysignal = QtCore.pyqtSignal(str)
```

В том же классе мы определили обязательный для потоков метод `run()` — в нем производится имитация процесса с помощью цикла перебора последовательности и метода `sleep()`: каждые три секунды выполняется генерация сигнала `mysignal` и передача текущего значения переменной `i` в составе строки:

```
self.mysignal.emit("i = %s" % i)
```

Внутри конструктора класса `MyWindow` мы назначили обработчик этого сигнала с помощью выражения

```
self.mythread.mysignal.connect(self.on_change,
                        QtCore.Qt.ConnectionType.QueuedConnection)
```

Здесь все нам уже знакомо: у свойства `mysignal` потока, которое представляет одноименный сигнал, вызывается метод `connect()`, и ему первым параметром передается обработчик. Во втором параметре метода `connect()` с помощью элемента `QueuedConnection` перечисления `ConnectionType` указывается, что сигнал помещается в очередь обработки событий и обработчик должен выполняться в основном потоке программы. Из основного потока мы можем смело изменять свойства компонентов.

Теперь рассмотрим код метода класса `MyWindow`, который станет обработчиком сигнала `mysignal`:

```
def on_change(self, s):
    self.label.setText(s)
```

Второй параметр этого метода служит для приема параметра, переданного сигналу. Значение этого параметра будет выведено в надписи с помощью метода `setText()`.

Еще в конструкторе класса `MyWindow` производится создание надписи и кнопки, а затем их размещение внутри вертикального контейнера. Далее создается объект класса `MyThread` и сохраняется в атрибуте `mythread`, чтобы в дальнейшем потоком можно было управлять. Запуск потока производится с помощью метода `start()` внутри обработчика нажатия кнопки. Чтобы исключить повторный запуск потока, мы с помощью метода `setDisabled()` делаем

кнопку неактивной, а после окончания работы потока внутри обработчика сигнала `finished()` опять делаем кнопку активной.

Для имитации длительного процесса мы использовали статический метод `sleep()` из класса `QThread`. Вообще, приостановить выполнение потока позволяют следующие статические методы класса `QThread`:

- `sleep(<Продолжительность>)` — продолжительность задается в секундах:

```
QtCore.QThread.sleep(3)          # "Засыпаем" на 3 секунды
```

- `msleep(<Продолжительность>)` — продолжительность задается в миллисекундах:

```
QtCore.QThread.msleep(3000)      # "Засыпаем" на 3 секунды
```

- `usleep(<Продолжительность>)` — продолжительность задается в микросекундах:

```
QtCore.QThread.usleep(3000000) # "Засыпаем" на 3 секунды
```

Еще один полезный статичный метод класса `QThread` — `yieldCurrentThread()` — немедленно приостанавливает выполнение текущего потока и передает управление следующему ожидающему выполнения потоку, если таковой есть:

```
QtCore.QThread.yieldCurrentThread()
```

18.8.2. Управление потоками

Если поток предназначен для выполнения в течение неопределенно долгого времени (например, в нем планируется производить какие-либо сложные вычисления в цикле), следует предусмотреть средства для управления этим потоком (скажем, для досрочного прерывания вычислений). Сделать это можно, создав в классе потока атрибут, который будет содержать значение (обычно логическое), обозначающее текущее состояние потока: «остановлен» или «работает». На каждой итерации цикла вычислений выполняется обращение к этому атрибуту, и как только его значение сменится на «остановлен», цикл прерывается и поток завершает работу. Изменить значение указанного атрибута можно через объект потока, в обработчике какого-либо события. Пример применения такого подхода для запуска и остановки потока приведен в листинге 18.14.

Листинг 18.14. Запуск и остановка потока

```
from PyQt6 import QtCore, QtWidgets

class MyThread(QtCore.QThread):
    mysignal = QtCore.pyqtSignal(str)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.running = False    # Флаг выполнения
        self.count = 0
    def run(self):
        self.running = True
        while self.running:     # Проверяем значение флага
            self.count += 1
            self.mysignal.emit("count = %s" % self.count)
            self.sleep(1)       # Имитируем процесс
```

```
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignmentFlag.AlignHCenter)
        self.btnStart = QtWidgets.QPushButton("Запустить поток")
        self.btnStop = QtWidgets.QPushButton("Остановить поток")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnStart)
        self.vbox.addWidget(self.btnStop)
        self.setLayout(self.vbox)
        self.mythread = MyThread()
        self.btnStart.clicked.connect(self.on_start)
        self.btnStop.clicked.connect(self.on_stop)
        self.mythread.mysignal.connect(self.on_change,
                                       QtCore.Qt.ConnectionType.QueuedConnection)
    def on_start(self):
        if not self.mythread.isRunning():
            self.mythread.start()       # Запускаем поток
    def on_stop(self):
        self.mythread.running = False  # Изменяем флаг выполнения
    def on_change(self, s):
        self.label.setText(s)
    def closeEvent(self, event):        # Вызывается при закрытии окна
        self.hide()                     # Скрываем окно
        self.mythread.running = False  # Изменяем флаг выполнения
        self.mythread.wait(5000)        # Даем время, чтобы закончить
        event.accept()                  # Закрываем окно

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Запуск и остановка потока")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec())
```

В этом примере в конструкторе класса `MyThread` создается атрибут `running`, и ему присваивается значение `False`. При запуске потока внутри метода `run()` значение атрибута изменяется на `True`. Затем запускается цикл, в котором значение этого атрибута указывается в качестве условия. Как только значение атрибута станет равным значению `False`, цикл будет остановлен.

Внутри конструктора класса `MyWindow` производится создание надписи, двух кнопок и объекта класса `MyThread`. Далее назначаются обработчики сигналов. По нажатии кнопки **Запустить поток** выполнится метод `on_start()`, внутри которого с помощью метода `isRunning()` производится проверка текущего статуса потока. Если поток не запущен, он запускается

вызовом метода start(). По нажатии кнопки **Остановить поток** выполнится метод on_stop(), в котором атрибуту running присваивается значение False. Это значение является условием выхода из цикла внутри метода run().

Путем изменения значения атрибута можно прервать выполнение цикла только в том случае, если закончилось выполнение очередной итерации. Если поток длительное время ожидает какого-либо события (например, ответа сервера), можно так и не дождаться завершения потока. Чтобы принудительно прервать выполнение потока, следует воспользоваться методом terminate(). Однако к этому методу рекомендуется прибегать только в крайнем случае, поскольку прерывание производится в произвольном месте кода. При этом блокировки автоматически не снимаются и существует вероятность повредить обрабатываемые данные. После вызова метода terminate() следует вызвать метод wait().

При закрытии окна программа завершается, что также приводит к завершению всех потоков. Чтобы предотвратить повреждение данных, следует перехватить событие закрытия окна и дождаться окончания выполнения потоков. Чтобы перехватить событие, необходимо внутри класса создать метод с именем closeEvent(). Этот метод будет автоматически вызван при попытке закрыть окно. В качестве параметра метод должен принимать объект события event, через который можно получить дополнительную информацию о событии. Чтобы закрыть окно внутри метода closeEvent(), следует вызвать метод accept() объекта события. Если необходимо предотвратить закрытие окна, то надо вызвать метод ignore().

Внутри метода closeEvent() мы присваиваем атрибуту running значение False. Далее с помощью метода wait() даем возможность потоку нормально завершить работу. В качестве параметра метод wait() принимает количество миллисекунд, по истечении которых управление будет передано следующей инструкции. Необходимо учитывать, что это максимальное время: если поток закончит работу раньше, то и метод закончит выполнение раньше. Метод wait() возвращает значение True, если поток успешно завершил работу, и False — в противном случае. Ожидание завершения потока занимает некоторое время, в течение которого окно будет по-прежнему видимым. Чтобы не обескуражить пользователя, в самом начале метода closeEvent() мы скрываем окно вызовом метода hide().

Каждый поток может иметь собственный цикл обработки событий, запускаемый вызовом метода exec(). В этом случае потоки могут обмениваться сигналами между собой. Чтобы прервать цикл, следует вызвать слот quit() или метод exit([returnCode=0]). Рассмотрим обмен сигналами между потоками на примере (листинг 18.15).

Листинг 18.15. Обмен сигналами между потоками

```
from PyQt6 import QtCore, QtWidgets

class Thread1(QtCore.QThread):
    s1 = QtCore.pyqtSignal(int)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.count = 0
    def run(self):
        self.exec()          # Запускаем цикл обработки сигналов
    def on_start(self):
        self.count += 1
        self.s1.emit(self.count)
```

```
class Thread2(QtCore.QThread):
    s2 = QtCore.pyqtSignal(str)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        self.exec()          # Запускаем цикл обработки сигналов
    def on_change(self, i):
        i += 10
        self.s2.emit("%d" % i)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку")
        self.label.setAlignment(QtCore.Qt.AlignmentFlag.AlignHCenter)
        self.button = QtWidgets.QPushButton("Сгенерировать сигнал")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.thread1 = Thread1()
        self.thread2 = Thread2()
        self.thread1.start()
        self.thread2.start()
        self.button.clicked.connect(self.thread1.on_start)
        self.thread1.s1.connect(self.thread2.on_change)
        self.thread2.s2.connect(self.on_thread2_s2)
    def on_thread2_s2(self, s):
        self.label.setText(s)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Обмен сигналами между потоками")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec())
```

В этом примере мы создали классы `Thread1`, `Thread2` и `MyWindow`. Первые два класса представляют собой потоки. Внутри них в методе `run()` вызывается метод `exec()`, который запускает цикл обработки событий. В конструкторе класса окна `MyWindow` производится создание надписи, кнопки и объектов классов `Thread1` и `Thread2`. Далее выполняется запуск сразу двух потоков.

В следующей инструкции сигнал нажатия кнопки соединяется с методом `on_start()` первого потока. Внутри этого метода производится какая-либо операция (в нашем случае — увеличение значения атрибута `count`), а затем с помощью метода `emit()` генерируется сигнал `s1`, и в параметре передается результат выполнения метода. Сигнал `s1` соединен с методом

on_change() второго потока. Внутри этого метода также производится какая-либо операция, а затем генерируется сигнал s2, и передается результат выполнения метода. В свою очередь, сигнал s2 соединен со слотом on_thread2_s2 объекта окна, который выводит в надпись значение, переданное с этим сигналом. Таким образом, по нажатии кнопки **Сгенерировать сигнал** вначале будет вызван метод on_start() класса Thread1, затем — метод on_change() класса Thread2, а потом — метод on_thread2_s2() класса MyWindow, который выведет результат выполнения на экран.

18.8.3. Очереди

Очередь — это подобие списка с возможностью добавлять элементы лишь в его конец и извлекать элементы, в зависимости от типа очереди, либо из начала, либо из конца, при этом извлеченный элемент удаляется из очереди. Очереди часто применяются, если требуется распределять постоянно поступающие задачи между ограниченным набором работающих потоков.

Модуль queue из стандартной библиотеки Python содержит следующие классы потокобезопасных очередей:

- Queue — обычная очередь типа «первым пришел, первым вышел». Формат конструктора:

  ```
  Queue([maxsize=0])
  ```

  Параметр maxsize в этом и последующих случаях задает максимальный размер очереди (количество элементов, которое может содержать очередь). Если параметр равен нулю или отрицательному значению, то размер очереди не ограничен. Пример:

  ```
  >>> import queue
  >>> q = queue.Queue()
  >>> q.put_nowait("elem1")
  >>> q.put_nowait("elem2")
  >>> q.get_nowait()
      'elem1'
  >>> q.get_nowait()
      'elem2'
  ```

- LifoQueue — *стек* (очередь типа «первым пришел, последним вышел»). Формат конструктора:

  ```
  LifoQueue([maxsize=0])
  ```

  Пример:

  ```
  >>> q = queue.LifoQueue()
  >>> q.put_nowait("elem1")
  >>> q.put_nowait("elem2")
  >>> q.get_nowait()
      'elem2'
  >>> q.get_nowait()
      'elem1'
  ```

- PriorityQueue — обычная очередь с приоритетами. Элементы очереди должны быть кортежами, в которых первым элементом является число, означающее приоритет, а вторым — значение заносимого в очередь элемента. При получении значения возвращается

элемент с наивысшим приоритетом (наименьшим значением в первом параметре кортежа). Формат конструктора класса:

```
PriorityQueue([maxsize=0])
```

Пример:

```
>>> q = queue.PriorityQueue()
>>> q.put_nowait((10, "elem1"))
>>> q.put_nowait((3, "elem2"))
>>> q.put_nowait((12, "elem3"))
>>> q.get_nowait()
    (3, 'elem2')
>>> q.get_nowait()
    (10, 'elem1')
>>> q.get_nowait()
    (12, 'elem3')
```

- `SimpleQueue` (начиная с Python 3.7) — аналогичен `Queue`, но не поддерживает некоторые программные инструменты и отнимает меньше системных ресурсов. Формат конструктора:

  ```
  SimpleQueue()
  ```

  Пример:

  ```
  >>> q = queue.SimpleQueue()
  >>> q.put_nowait("elem1")
  >>> q.put_nowait("elem2")
  >>> q.get_nowait()
      'elem1'
  >>> q.get_nowait()
      'elem2'
  ```

Все классы очередей поддерживают следующие методы:

- `put(<Элемент>[, block=True][, timeout=None])` — добавляет заданный элемент в текущую очередь.

  Если очередь имеет ограниченный размер и полностью заполнена, а в параметре `block` указано значение `False`, будет сгенерировано исключение `Full` из модуля `queue`. Если же параметром `block` передано значение `True`, метод приостановит выполнение потока и будет ждать, пока в очереди не появится свободное место, в течение времени, заданного в параметре `timeout` в секундах. Если в очереди за это время не освободилось место для добавления элемента, также генерируется исключение `Full`.

  Поскольку очередь, реализуемая классом `SimpleQueue`, всегда имеет неограниченный размер, значения параметров `block` и `timeout` в ее случае игнорируются. Эти параметры предусмотрены лишь для совместимости;

- `put_nowait(<Элемент>)` — добавляет заданный элемент в текущую очередь без ожидания. Эквивалентен выражению

  ```
  put(<Элемент>, False)
  ```

- `get([block=True][, timeout=None])` — извлекает из текущей очереди очередной элемент и возвращает его (при этом удаляя из очереди).

Если очередь пуста и параметр `block` имеет значение `False`, генерируется исключение `Empty` из модуля `queue`. Если в параметре `block` указано значение `True`, метод приостановит выполнение потока и будет ожидать, когда в очереди появится хотя бы один элемент, в течение времени, заданного параметром `timeout` в секундах. Если в очереди за заданное время так и не появилось ни одного элемента, также генерируется исключение `Empty`;

- `get_nowait()` — извлекает элемент из текущей очереди без ожидания. Эквивалентен выражению

  ```
  get(False);
  ```

- `qsize()` — возвращает приблизительное количество элементов в текущей очереди. Если доступ к очереди имеют несколько потоков, доверять этому значению не следует — в любой момент времени размер очереди может измениться;
- `empty()` — возвращает `True`, если текущая очередь пуста, и `False` — в противном случае. Если доступ к очереди имеют несколько потоков, доверять этому значению также не следует — в любой момент времени размер очереди может измениться.

Следующие методы не поддерживаются классом `SimpleQueue`:

- `join()` — блокирует поток, пока из текущей очереди не будут выбраны все элементы. Другие потоки после выборки очередного элемента из очереди должны вызывать метод `task_done()`;
- `task_done()` — уведомляет текущую очередь о выборке очередного элемента;
- `full()` — возвращает `True`, если текущая очередь заполнена до предела, и `False` — в противном случае. Если доступ к очереди имеют несколько потоков, доверять этому значению также не следует — в любой момент времени размер очереди может измениться.

Рассмотрим использование очереди в многопоточной программе на примере (листинг 18.16).

Листинг 18.16. Использование очереди

```
from PyQt6 import QtCore, QtWidgets
import queue

class MyThread(QtCore.QThread):
    task_done = QtCore.pyqtSignal(int, int, name='taskDone')
    def __init__(self, id, queue, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
        self.queue = queue
    def run(self):
        while True:
            task = self.queue.get()             # Получаем задание
            self.sleep(5)                       # Имитируем обработку
            self.task_done.emit(task, self.id) # Передаем данные обратно
            self.queue.task_done()

class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
        QtWidgets.QPushButton.__init__(self)
```

```
        self.setText("Раздать задания")
        self.queue = queue.Queue()          # Создаем очередь
        self.threads = []
        for i in range(1, 3):               # Создаем потоки и запускаем
            thread = MyThread(i, self.queue)
            self.threads.append(thread)
            thread.task_done.connect(self.on_task_done,
                                     QtCore.Qt.ConnectionType.QueuedConnection)
            thread.start()
        self.clicked.connect(self.on_add_task)
    def on_add_task(self):
        for i in range(0, 11):
            self.queue.put(i)               # Добавляем элементы в очередь
    def on_task_done(self, data, id):
        print(data, "- id =", id)           # Выводим обработанные данные

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование очереди")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec())
```

В этом примере конструктор класса `MyThread` принимает уникальный идентификатор (`id`) и ссылку на очередь (`queue`), которые сохраняются в одноименных атрибутах класса. В методе `run()` внутри бесконечного цикла из очереди с помощью метода `get()` извлекается очередной элемент. Если очередь пуста, поток будет ожидать, пока не появится хотя бы один элемент. Далее производится обработка извлеченного элемента (в нашем случае — просто задержка), а затем результаты обработки передаются главному потоку через сигнал `taskDone`, принимающий два целочисленных параметра. После чего вызовом метода `task_done()` у очереди указывается, что извлеченный элемент успешно обработан.

Отметим, что в вызове функции `pyqtSignal()` присутствует именованный параметр `name`:

```
task_done = QtCore.pyqtSignal(int, int, name='taskDone')
```

Он задает имя сигнала и может быть полезен в том случае, если это имя отличается от имени атрибута класса, соответствующего сигналу (как в нашем случае, где имя сигнала `taskDone` отличается от имени атрибута `task_done`). После чего мы можем обращаться к сигналу как по имени соответствующего ему атрибута:

```
self.task_done.emit(task, self.id)
```

так и по имени, заданному в параметре `name` функции `pyqtSignal()`:

```
self.taskDone.emit(task, self.id)
```

Поскольку в окне программы будет присутствовать лишь кнопка, класс окна `MyWindow` сделан производным от класса кнопки `QPushButton`, а не от класса окна `QWidget`. Все визуальные компоненты (включая кнопку) являются наследниками класса `QWidget`, поэтому любой компонент, не имеющий родителя, будет выведен в своем собственном окне.

Внутри конструктора класса `MyWindow` с помощью метода `setText()` задается текст надписи на кнопке, затем создается объект класса `Queue` и сохраняется в атрибуте `queue`. В следующем выражении создается список, в котором будут храниться ссылки на объекты потоков. Сами объекты потоков (в нашем случае их два) создаются внутри цикла и добавляются в список. Внутри цикла также производится назначение обработчика сигнала `taskDone` и запуск потока с помощью метода `start()`. Далее назначается обработчик нажатия кнопки.

По нажатии кнопки **Раздать задания** вызывается метод `on_add_task()`, внутри которого обрабатываемые элементы добавляются в очередь. После этого потоки начинают работать, и каждый из них выбирает из очереди один элемент. Завершив обработку элемента, поток генерирует сигнал `taskDone` и вызывает метод `task_done()`, информирующий об окончании обработки задания. Главный поток получает сигнал и вызывает метод `on_task_done()`, внутри которого через параметры будут доступны результаты обработки. В нашем примере эти результаты просто выводятся в консоль (чтобы увидеть их, следует сохранить файл с расширением `py` или просто запустить программу из среды IDLE Shell). Обработав элемент и сигнализировав об этом, поток выбирает из очереди следующий элемент и т. д. Если очередь окажется пуста, потоки приостановят работу, пока в очереди не появятся элементы.

18.8.4. Блокировщики и автоблокировщики

Попытка одновременного доступа со стороны нескольких потоков к какому-либо ресурсу, не поддерживающему одновременный доступ (например, консоли), может привести к непредсказуемому поведению программы или даже ее аварийному завершению. В таких случаях в программе следует временно блокировать ресурс при использовании его одним из потоков, чтобы другие потоки не смогли получить к нему доступ.

Блокировщик (*мьютекс*) позволяет явно заблокировать ресурс на произвольное время. Впоследствии заблокированный ресурс должен быть явно разблокирован.

Блокировщик реализуется классом `QMutex` из модуля `QtCore`. Конструктор класса имеет следующий формат:

```
QMutex()
```

Класс `QMutex` поддерживает следующие методы:

- `lock()` — устанавливает блокировку. Если ресурс был заблокирован другим потоком, работа текущего потока приостанавливается до снятия блокировки;
- `tryLock([timeout=0])` — устанавливает блокировку. Если блокировка была успешно установлена, метод возвращает значение `True`, если ресурс заблокирован другим потоком — значение `False` без ожидания возможности установить блокировку. В параметре `timeout` можно указать максимальное время ожидания в миллисекундах. Если в параметре указано отрицательное значение, то метод `tryLock()` ведет себя аналогично методу `lock()`;
- `unlock()` — снимает блокировку.

Рассмотрим использование блокировщика на примере (листинг 18.17).

Листинг 18.17. Использование блокировщика

```
from PyQt6 import QtCore, QtWidgets

class MyThread(QtCore.QThread):
    x = 10                                  # Атрибут класса
```

```
    mutex = QtCore.QMutex()                 # Блокировщик
    def __init__(self, id, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
    def run(self):
        self.change_x()
    def change_x(self):
        MyThread.mutex.lock()               # Блокируем
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        MyThread.mutex.unlock()             # Снимаем блокировку

class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
        QtWidgets.QPushButton.__init__(self)
        self.setText("Запустить")
        self.thread1 = MyThread(1)
        self.thread2 = MyThread(2)
        self.clicked.connect(self.on_start)
    def on_start(self):
        if not self.thread1.isRunning(): self.thread1.start()
        if not self.thread2.isRunning(): self.thread2.start()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование блокировщика")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec())
```

В классе `MyThread` мы создали атрибут класса `x`, который и будем блокировать блокировщиком. Последний мы сохранили в другом атрибуте класса — `mutex`. Код метода `change_x()`, в котором изменяется значение атрибута класса `x`, поместили между вызовами методов `lock()` и `unlock()` блокировщика, — для гарантии, что к атрибуту класса в каждый конкретный момент времени имеет доступ лишь один поток.

Внутри конструктора класса `MyWindow` создаются два объекта класса `MyThread` и назначается обработчик нажатия кнопки. По нажатии кнопки **Запустить** будет вызван метод `on_start()`, внутри которого одновременно запускаются оба потока (конечно, если они не были запущены ранее). В результате мы получим в консоли следующий результат:

```
x = 10 id = 1
x = 15 id = 1
```

```
x = 49 id = 1
x = 49 id = 2
x = 54 id = 2
x = 88 id = 2
```

Как можно видеть, сначала значение атрибута класса изменил поток с идентификатором 1, а лишь затем — поток с идентификатором 2. Если блокировку не указать, то результат будет иным:

```
x = 10 id = 1
x = 15 id = 2
x = 20 id = 1
x = 54 id = 1
x = 54 id = 2
x = 88 id = 2
```

В этом случае поток с идентификатором 2 изменил значение атрибута класса `x` до окончания выполнения метода `change_x()` в потоке с идентификатором 1.

При возникновении исключения внутри метода `change_x()` атрибут класса останется заблокированным, т. к. вызов метода `unlock()` не будет выполнен. Кроме того, можно по случайности забыть вызвать метод `unlock()`, что также приведет к вечной блокировке.

Автоблокировщик автоматически блокирует указанный блокировщик при создании и так же автоматически разблокирует его при уничтожении. Применение автоблокировщика позволяет исключить описанную только что ситуацию.

Автоблокировщик реализуется классом `QMutexLocker` из модуля `QtCore`. Формат вызова конструктора:

```
QMutexLocker(<Блокировщик>)
```

Переделаем метод `change_x()` из класса `MyThread` (см. листинг 18.17) с использованием автоблокировщика (листинг 18.18).

Листинг 18.18. Использование автоблокировщика

```
    def change_x(self):
        # Создаем автоблокировщик и тем самым накладываем блокировку
        ml = QtCore.QMutexLocker(MyThread.mutex)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        # Здесь локальная переменная с автоблокировщиком будет уничтожена,
        # автоблокировщик перестанет существовать, и блокировка снимется
```

Класс `QMutexLocker` является менеджером контекста, что позволяет использовать автоблокировщики в обработчиках контекстов. Переделаем снова метод `change_x()` из класса `MyThread` (см. листинг 18.17), применив в этот раз инструкцию обработчика контекста (листинг 18.19).

Листинг 18.19. Использование обработчика контекста

```
def change_x(self):
    with QtCore.QMutexLocker(MyThread.mutex):
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        # Блокировка автоматически снимется
```

Осталось сделать несколько замечаний:

- установка и снятие блокировки занимают некоторое время, что снижает быстродействие программы. Поэтому встроенные типы данных не обеспечивают безопасную работу в многопоточных программах. Прежде чем использовать блокировки, подумайте — может быть, в вашей программе они и не нужны;
- ожидание снятия блокировки может заблокировать основной поток, и программа перестанет реагировать на события. Если такое происходит достаточно часто, следует использовать сигналы, а не прямой доступ;
- если первый поток, владея ресурсом `A`, захочет получить доступ к ресурсу `B`, а второй поток, владея ресурсом `B`, захочет получить доступ к ресурсу `A`, возникнет *взаимная блокировка*, снятия которой потоки будут ждать вечно. В этой ситуации следует временно разблокировать занятый ресурс в одном из потоков после превышения определенного времени.

Класс `QMutexLocker` также поддерживает методы `unlock()` и `relock()`. Первый метод выполняет разблокировку без уничтожения текущего автоблокировщика, а второй накладывает блокировку повторно.

> ***Примечание***
>
> Для синхронизации и координации потоков также можно использовать классы `QSemaphore` и `QWaitCondition`. За подробной информацией по этим классам обращайтесь к документации по PyQt. А в стандартную библиотеку языка Python входят модули `multiprocessing` и `threading`, которые позволяют работать с потоками в любой программе. Однако при использовании PyQt нужно отдать предпочтение классу `QThread`, поскольку он позволяет работать с сигналами.

18.9. Вывод заставки

При запуске программы начальная инициализация может отнять много времени. Поэтому сразу после запуска следует вывести *заставку* — окно, показывающее логотип программы и процесс запуска. По окончании запуска программы заставку нужно скрыть.

Для вывода заставки в PyQt предназначен класс `QSplashScreen` из модуля `QtWidgets`. Конструктор класса имеет следующие форматы:

```
QSplashScreen([<Изображение>][, ][flags=0])
QSplashScreen(<Родитель>[, <Изображение>][, flags=0])
```

Параметр <Родитель> позволяет указать ссылку на родительский компонент. В параметре <Изображение> указывается ссылка на изображение (объект класса QPixmap из модуля QtGui), которое будет отображаться на заставке. Конструктору класса QPixmap можно передать путь к файлу с изображением. Параметр flags предназначен для указания типа окна — например, чтобы заставка отображалась поверх всех остальных окон, следует передать элемент WindowStaysOnTopHint перечисления WindowType из модуля QtCore.Qt.

Класс QSplashScreen поддерживает следующие методы:

- show() — отображает текущую заставку;
- finish(<Окно>) — закрывает текущую заставку. В качестве параметра указывается ссылка на главное окно программы;
- showMessage(<Надпись>[, <Местоположение>[, <Цвет>]]) — выводит указанную надпись в текущей заставке. Во втором параметре задается местоположение надписи в окне в виде одного из следующих элементов перечисления AlignmentFlag из модуля QtCore.Qt или их комбинации через оператор |: AlignTop (по верху), AlignCenter (по центру вертикали и горизонтали), AlignBottom (по низу), AlignHCenter (по центру горизонтали), AlignVCenter (по центру вертикали), AlignLeft (по левой стороне), AlignRight (по правой стороне). В третьем параметре указывается цвет текста в виде элемента перечисления GlobalColor из модуля QtCore.Qt (скажем, black) объекта класса QColor (например, QColor("red"), QColor("#ff0000"), QColor(255, 0, 0) и др.);
- clearMessage() — удаляет ранее выведенную надпись;
- setPixmap(<Изображение>) — выводит указанное изображение в текущей заставке. В качестве параметра указывается объект класса QPixmap;
- pixmap() — возвращает изображение, выведенное в текущей заставке, в виде объекта класса QPixmap.

Пример кода, выводящего заставку, показан в листинге 18.20, а сама заставка — на рис. 18.7.

Листинг 18.20. Вывод заставки

```
from PyQt6 import QtGui, QtWidgets, QtCore
import time

class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
        QtWidgets.QPushButton.__init__(self)
        self.setText("Закрыть окно")
        self.clicked.connect(QtWidgets.QApplication.instance().quit)
    def load_data(self, sp):
        for i in range(1, 11):              # Имитируем процесс
            time.sleep(2)                   # Что-то загружаем
            sp.showMessage("Загрузка данных... {0}%".format(i * 10),
               QtCore.Qt.AlignmentFlag.AlignHCenter |
               QtCore.Qt.AlignmentFlag.AlignBottom,
               QtCore.Qt.GlobalColor.black)
            # Принудительно обрабатываем события
            QtWidgets.QApplication.instance().processEvents()
```

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    splash = QtWidgets.QSplashScreen(QtGui.QPixmap("splashscreen.svg"))
    splash.showMessage("Загрузка данных... 0%",
           QtCore.Qt.AlignmentFlag.AlignHCenter |
           QtCore.Qt.AlignmentFlag.AlignBottom,
           QtGui.QColor("black"))
    splash.show()                              # Отображаем заставку
    # Принудительно обрабатываем события
    QtWidgets.QApplication.instance().processEvents()
    window = MyWindow()
    window.setWindowTitle("Вывод заставки")
    window.resize(300, 30)
    window.load_data(splash)                   # Загружаем данные
    window.show()
    splash.finish(window)                      # Скрываем заставку
    sys.exit(app.exec())
```

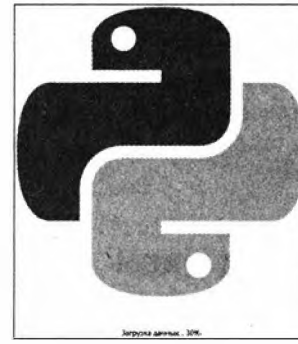


Рис. 18.7. Заставка

ГЛАВА 19

Окна

Обычное окно представляется классом `QWidget` из модуля `QtWidgets`.

Класс `QWidget` является базовым для всех классов, представляющих компоненты. Благодаря этому любой компонент (например, кнопка), у которого не был явно указан родитель, будет автоматически выведен в собственном окне (чем мы и пользовались в *главе 18*). Кроме того, классы компонентов поддерживают все методы, имеющиеся в классе `QWidget` (в том числе методы для вывода компонентов, их скрытия, изменения размеров, перемещения, получения состояния видимости, размеров, местоположения компонентов и др.).

Для создания окна также можно использовать более специализированные классы: `QFrame` (окно с рамкой), `QDialog` (диалоговое окно) или `QMainWindow` (главное окно программы с меню, панелями инструментов и строкой состояния). Последним двум классам будут посвящены *главы 27* и *28* соответственно.

19.1. Создание и вывод окон

Самый простой способ создать пустое окно показан в листинге 19.1.

Листинг 19.1. Создание и отображение окна

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()                # Создаем окно
window.setWindowTitle("Заголовок окна")     # Указываем заголовок
window.resize(300, 50)                      # Минимальные размеры
window.show()                               # Отображаем окно
sys.exit(app.exec())
```

Конструктор класса `QWidget` имеет следующий формат:

```
QWidget([parent=Null][, flags=0])
```

В параметре `parent` указывается ссылка на родительский компонент. Если этот параметр имеет значение `None` и у параметра `flags` указано значение 0, компонент не будет иметь родителя и получит свое собственное окно.

Параметр flags определяет тип создаваемого окна. Доступные для задания типы будут рассмотрены в *разд. 19.1.1*.

Указать ссылку на родительский компонент и, возможно, тип окна уже после его создания позволяет метод setParent() окна. Формат метода:

```
setParent(<Родитель>[, <Тип окна>])
```

Получить ссылку на родительский компонент можно с помощью метода parentWidget(). Если компонент не имеет родителя, возвращается значение None.

Для изменения текста в заголовке окна предназначен метод setWindowTitle(<Новый текст>). А метод windowTitle() возвращает текст, выведенный в заголовке окна.

После создания окна необходимо вызвать у него метод show(), чтобы вывести окно на экран. Для скрытия окна предназначен метод hide(). Для отображения и скрытия окна также можно пользоваться методом setVisible(<Состояние>). Если параметром этого метода передано значение True, окно будет отображено, а если значение False — скрыто. Пример отображения окна:

```
window.setVisible(True)
```

Проверить, видимо ли окно в настоящее время или нет, позволяет метод isVisible(), который возвращает True, если окно видимо, и False — в противном случае. Кроме того, можно воспользоваться методом isHidden() — он возвращает True, если окно скрыто, и False — в противном случае.

19.1.1. Типы окон

Тип окна можно задать как при его создании, во втором параметре конструктора класса QWidget, так и вызовом метода setWindowFlags(<Тип окна>) *перед выводом окна*.

В качестве типа окна можно указать один из следующих элементов перечисления WindowType из модуля QtCore.Qt:

- Widget — в зависимости от наличия родителя:
  - родитель указан — компонент выводится в составе родительского окна;
  - родитель не указан — компонент выводится в окне изменяемых размеров с рамкой, заголовком, системным меню, кнопками сворачивания, разворачивания и закрытия.

  Этот тип используется по умолчанию у класса QWidget;
- Window — то же самое, что и Widget, только компонент всегда выводится в собственном окне, независимо от наличия или отсутствия у него родителя;
- Dialog — диалоговое окно: изменяемых размеров, с рамкой, заголовком и кнопкой закрытия. Используется умолчанию у класса QDialog. Пример:

  ```
  window.setWindowFlags(QtCore.Qt.WindowType.Dialog)
  ```

- Sheet и Drawer — окна в стиле Apple Macintosh;
- Popup — всплывающее окно: неизменяемых размеров, без рамки, заголовка и кнопок, может отбрасывать тень;
- Tool — панель инструментов: изменяемых размеров, с рамкой, уменьшенным заголовком и кнопкой закрытия;
- ToolTip — всплывающая подсказка: неизменяемых размеров, без рамки, заголовка и кнопок;

- `SplashScreen` — заставка: неизменяемых размеров, без рамки, заголовка и кнопок. Используется по умолчанию у класса `QSplashScreen`;
- `SubWindow` — подчиненное окно: неизменяемых размеров, с рамкой, уменьшенным заголовком, без кнопок;
- `ForeignWindow` — окно, созданное другим процессом;
- `CoverWindow` — окно, представляющее свернутую программу на некоторых платформах.

Получить тип окна в программе позволяет метод `windowType()`.

У окон можно через оператор `|` дополнительно указать следующие элементы перечисления `WindowType` (упомянуты только наиболее часто используемые элементы, полный их список ищите в документации):

- `MSWindowsFixedSizeDialogHint` — запрещает изменение размеров окна. Кнопка разворачивания в заголовке окна станет неактивной;
- `FramelessWindowHint` — убирает рамку и заголовок окна. Изменять размеры окна и перемещать его будет нельзя;
- `NoDropShadowWindowHint` — убирает отбрасываемую окном тень;
- `CustomizeWindowHint` — убирает рамку и заголовок окна, но добавляет эффект объемности. Размеры окна можно изменять;
- `WindowTitleHint` — добавляет заголовок окна. Пример вывода окна фиксированного размера с заголовком, в котором находится только текст:

```
window.setWindowFlags(QtCore.Qt.WindowType.Window |
                      QtCore.Qt.WindowType.FramelessWindowHint |
                      QtCore.Qt.WindowType.WindowTitleHint)
```

- `WindowSystemMenuHint` — добавляет в заголовок системное меню и кнопку закрытия;
- `WindowMinimizeButtonHint` — добавляет в заголовок кнопку сворачивания;
- `WindowMaximizeButtonHint` — добавляет в заголовок кнопку разворачивания;
- `WindowMinMaxButtonsHint` — добавляет в заголовок кнопки сворачивания и разворачивания;
- `WindowCloseButtonHint` — добавляет в заголовок кнопку закрытия;
- `WindowContextHelpButtonHint` — добавляет в заголовок кнопку вывода справки;
- `WindowStaysOnTopHint` — сообщает системе, что окно всегда должно отображаться поверх всех других окон;
- `WindowStaysOnBottomHint` — сообщает системе, что окно всегда должно быть расположено позади всех других окон.

Получить все установленные элементы-типы окна из программы позволяет метод `windowFlags()`.

19.2. Размеры окон и управление ими

Для изменения размеров окна предназначены следующие методы объекта окна:

- `resize()` — изменяет размеры текущего окна. Форматы вызова:

```
resize(<Ширина>, <Высота>)
resize(<Объект размеров>)
```

В первом формате ширина и высота указываются в виде чисел, а во втором — в виде объекта класса `QSize` из модуля `QtCore`. Пример:

```
window.resize(100, 70)
window.resize(QtCore.QSize(100, 70))
```

Если содержимое окна не помещается в установленный размер, то размер будет выбран так, чтобы компоненты поместились без искажения при условии, что используются менеджеры геометрии. Если используется абсолютное позиционирование, компоненты могут оказаться наполовину или полностью за пределами видимой части окна:

- `setGeometry()` — изменяет одновременно местоположение и размеры текущего окна. Форматы вызова:

```
setGeometry(<X>, <Y>, <Ширина>, <Высота>)
setGeometry(<Объект местоположения и размеров>)
```

В первом формате первые два параметра задают координаты левого верхнего угла относительно родительского компонента, а третий и четвертый параметры — ширину и высоту. Во втором формате указывается объект класса `QRect` из модуля `QtCore`.

> ***Внимание!***
>
> Начало координат расположено в левом верхнем углу. Ось `X` направлена вправо, а ось `Y` — вниз.

Примеры:

```
window.setGeometry(100, 100, 100, 70)
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
```

- `setFixedSize()` — задает у текущего окна фиксированные ширину и высоту и делает его размеры неизменяемыми. Форматы вызова такие же, как и у метода `resize()`. Пример:

```
window.setFixedSize(100, 70)
window.setFixedSize(QtCore.QSize(100, 70))
```

- `setFixedWidth(<Ширина>)` — задает у текущего окна фиксированную ширину и делает ее неизменяемой;
- `setFixedHeight(<Высота>)` — задает у текущего окна фиксированную высоту и делает ее неизменяемой;
- `setMinimumSize()` — задает минимальные размеры окна. Форматы вызова такие же, как и у метода `resize()`. Пример:

```
window.setMinimumSize(100, 70)
window.setMinimumSize(QtCore.QSize(100, 70))
```

- `setMinimumWidth(<Ширина>)` — задает минимальную ширину;
- `setMinimumHeight(<Высота>)` — задает минимальную высоту;
- `setMaximumSize()` — задает максимальные размеры окна. Форматы вызова такие же, как и у метода `resize()`. Пример:

```
window.setMaximumSize(100, 70)
window.setMaximumSize(QtCore.QSize(100, 70))
```

- `setMaximumWidth(<Ширина>)` — задает максимальную ширину;

- `setMaximumHeight(<Высота>)` — задает максимальную высоту;
- `setBaseSize()` — задает базовые размеры. Форматы вызова такие же, как и у метода `resize()`. Пример:

```
window.setBaseSize(500, 500)
window.setBaseSize(QtCore.QSize(500, 500))
```

- `adjustSize()` — подгоняет размеры окна под содержимое, учитывая рекомендуемые размеры, которые выдаются методом `sizeHint()`.

Получить размеры позволяют следующие методы окна:

- `width()` — возвращает ширину;
- `height()` — возвращают высоту:

```
window.resize(50, 70)
print(window.width(), window.height())                # 50 70
```

- `size()` — возвращает объект класса `QSize`, содержащий текущие размеры:

```
window.resize(50, 70)
print(window.size().width(), window.size().height()) # 50 70
```

- `minimumSize()` — возвращает объект класса `QSize`, содержащий минимальные размеры;
- `minimumWidth()` — возвращает минимальную ширину;
- `minimumHeight()` — возвращает минимальную высоту;
- `maximumSize()` — возвращает объект класса `QSize`, содержащий максимальные размеры;
- `maximumWidth()` — возвращает максимальную ширину;
- `maximumHeight()` — возвращает максимальную высоту;
- `baseSize()` — возвращает объект класса `QSize`, содержащий базовые размеры;
- `sizeHint()` — возвращает объект класса `QSize`, содержащий рекомендуемые размеры компонента. Если таковые являются отрицательными, считается, что нет рекомендуемого размера;
- `minimumSizeHint()` — возвращает объект класса `QSize`, содержащий рекомендуемые минимальные размеры окна. Если таковые являются отрицательными, то считается, что нет рекомендуемого минимального размера;
- `rect()` — возвращает объект класса `QRect`, содержащий координаты и размеры прямоугольника, в который вписано окно:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.rect()
print(rect.left(), rect.top())      # 0 0
print(rect.width(), rect.height()) # 100 70
```

- `geometry()` — возвращает объект класса `QRect`, содержащий координаты относительно родительского компонента:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.geometry()
print(rect.left(), rect.top())      # 100 100
print(rect.width(), rect.height()) # 100 70
```

При изменении и получении размеров окна следует учитывать, что:

- размеры не включают высоту заголовка окна и толщину границ;
- размеры могут изменяться в зависимости от настроек стиля. Например, на разных компьютерах может быть задан шрифт разного наименования и кегля, поэтому от указания фиксированных размеров лучше отказаться;
- размер может изменяться в промежутке между получением значения и действиями, обрабатывающими эти значения. Например, сразу после получения размеров пользователь может изменить размеры окна с помощью мыши.

Чтобы получить размеры окна, включающие высоту заголовка и ширину границ, следует воспользоваться методом `frameSize()`, который возвращает объект класса `QSize`. Обратите внимание, что полные размеры окна доступны только после его отображения, — до этого момента они совпадают с размерами клиентской области окна, без учета высоты заголовка и толщины границ. Пример получения полных размеров окна:

```
window.resize(200, 70)                    # Задаем размеры
. . .
window.show()                             # Отображаем окно
print(window.width(), window.height())    # 200 70
print(window.frameSize().width(),
      window.frameSize().height())        # 208 104
```

Чтобы получить координаты окна с учетом высоты заголовка и толщины границ, следует воспользоваться методом `frameGeometry()`, который возвращает объект класса `QRect`. И в этом случае полные размеры окна доступны только после его отображения. Пример:

```
window.setGeometry(100, 100, 200, 70)
. . .
window.show()                             # Отображаем окно
rect = window.geometry()
print(rect.left(), rect.top())            # 100 100
print(rect.width(), rect.height())        # 200 70
rect = window.frameGeometry()
print(rect.left(), rect.top())            # 96 70
print(rect.width(), rect.height())        # 208 104
```

19.3. Местоположение окна и управление им

Задать местоположение окна на экране позволяют следующие методы объекта окна:

- `move()` — задает положение текущего окна относительно родителя с учетом высоты заголовка и толщины границ. Форматы вызова:

  ```
  move(<X>, <Y>)
  move(<Объект местоположения>)
  ```

  В первом формате координаты левого верхнего угла окна указываются в виде чисел, а во втором — в виде объекта класса `QPoint` из модуля `QtCore`. Пример вывода окна в левом верхнем углу экрана:

  ```
  window.move(0, 0)
  window.move(QtCore.QPoint(0, 0))
  ```

- `setGeometry(<X>, <Y>, <Ширина>, <Высота>)` — задает одновременно положение и размеры текущего окна. Подробно этот метод был описан в *разд. 19.2*.

Получить местоположение окна позволяют следующие методы:

- `x()` — возвращает горизонтальную координату левого верхнего угла окна относительно родителя с учетом толщины границ.
- `y()` — возвращает вертикальную координату левого верхнего угла окна относительно родителя с учетом высоты заголовка и толщины границ:

```
window.move(10, 10)
rint(window.x(), window.y())              # 10 10
```

- `pos()` — возвращает объект класса `QPoint`, содержащий координаты левого верхнего угла окна относительно родителя с учетом высоты заголовка и толщины границ:

```
window.move(10, 10)
print(window.pos().x(), window.pos().y()) # 10 10
```

- `geometry()` — возвращает объект класса `QRect`, содержащий координаты относительно родительского компонента, без учета высоты заголовка и толщины границ:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.geometry()
print(rect.left(), rect.top())            # 14 40
print(rect.width(), rect.height())        # 300 100
```

- `frameGeometry()` — возвращает объект класса `QRect`, содержащий координаты относительно родителя с учетом высоты заголовка и ширины границ. Полные размеры окна доступны только после отображения окна. Пример:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.frameGeometry()
print(rect.left(), rect.top())            # 10 10
print(rect.width(), rect.height())        # 308 134
```

Для отображения окна по центру экрана, у правой или нижней его границы необходимо знать размеры экрана. Для получения их следует вызвать метод `screen()` окна, возвращающий ссылку на экран, на котором выведено это окно и который представлен объектом класса `QScreen` из модуля `QtGui`. Получить размеры экрана позволяют следующие методы этого класса:

- `size()` — возвращает объект класса `QSize`, содержащий полные размеры экрана:

```
screen = window.screen()
siz = screen.size()
print(siz.width(), siz.height())          # 2560 1440
```

- `availableSize()` — возвращает объект класса `QSize`, содержащий размеры доступной части экрана (не занятой панелью задач и иными панелями):

```
siz = screen.availableSize()
print(siz.width(), siz.height())          # 2560 1400
```

- `geometry()` — возвращает объект класса `QRect`, содержащий координаты и полные размеры экрана:

```
screen = window.screen()
rect = screen.geometry()
print(rect.left(), rect.top())          # 0 0
print(rect.width(), rect.height())      # 2560 1440
```

- `availableGeometry()` — возвращает объект класса `QRect`, содержащий координаты и размеры доступной части экрана (не занятой панелью задач и иными панелями):

```
rect = screen.availableGeometry()
print(rect.left(), rect.top())          # 0 0
print(rect.width(), rect.height())      # 2560 100
```

Пример отображения окна приблизительно по центру экрана показан в листинге 19.2.

Листинг 19.2. Вывод окна приблизительно по центру экрана

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Вывод окна примерно по центру экрана")
window.resize(300, 100)
screen_size = window.screen().availableSize()
x = (screen_size.width() - window.width()) // 2
y = (screen_size.height() - window.height()) // 2
window.move(x, y)
window.show()
sys.exit(app.exec())
```

В этом примере использовались методы `width()` и `height()` окна, которые не учитывают высоту заголовка и толщину границ. Если требуется поместить окно точно в центре экрана, следует воспользоваться методом `frameSize()`. Однако этот метод возвращает корректные значения лишь после вывода окна. Если код выравнивания по центру расположить после вызова метода `show()`, окно вначале отобразится в одном месте экрана, а затем переместится в центр, что вызовет неприятное мелькание. Чтобы исключить такое мелькание, следует вначале вывести окно за пределами экрана, а затем переместить его в центр (листинг 19.3).

Листинг 19.3. Вывод окна точно по центру экрана

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Вывод окна точно по центру экрана")
window.resize(300, 100)
window.move(window.width() * -2, 0)
window.show()
screen_size = window.screen().availableSize()
x = (screen_size.width() - window.frameSize().width()) // 2
y = (screen_size.height() - window.frameSize().height()) // 2
window.move(x, y)
sys.exit(app.exec())
```

Если нужно расположить окно в правом верхнем углу экрана, надо заменить код из предыдущего примера, выравнивающий окно по центру, следующим кодом:

```
screen_size = window.screen().availableSize()
x = screen_size.width() - window.frameSize().width()
window.move(x, 0)
```

Следует заметить, что экран может быть разделен на несколько рабочих столов. Это необходимо учитывать при размещении окна (за более подробной информацией обращайтесь к документации по классу `QScreen`).

19.4. Классы, задающие координаты и размеры

В двух предыдущих разделах были упомянуты классы `QPoint`, `QSize` и `QRect`, определенные в модуле `QtCore`. Рассмотрим их.

> ***Примечание***
>
> Классы `QPoint`, `QSize` и `QRect` предназначены для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать классы `QPointF`, `QSizeF` и `QRectF` из модуля `QtCore`.

19.4.1. Класс *QPoint*: координаты точки

Класс `QPoint` описывает координаты точки. Его конструктор поддерживает следующие форматы вызова:

```
QPoint()
QPoint(<X>, <Y>)
QPoint(<Объект класса QPoint>)
```

Первый формат создает объект класса с нулевыми координатами:

```
>>> from PyQt6 import QtCore
>>> p = QtCore.QPoint()
>>> p.x(), p.y()
    (0, 0)
```

Второй формат позволяет явно указать координаты точки:

```
>>> p = QtCore.QPoint(10, 88)
>>> p.x(), p.y()
    (10, 88)
```

Третий формат создает копию заданного объекта:

```
>>> p = QtCore.QPoint(QtCore.QPoint(10, 88))
>>> p.x(), p.y()
    (10, 88)
```

Класс поддерживает следующие методы:

- `x()` — возвращает горизонтальную координату;
- `y()` — возвращает вертикальную координату;
- `setX(<X>)` — задает новую горизонтальную координату;

- `setY(<Y>)` — задает новую вертикальную координату;
- `isNull()` — возвращает `True`, если координаты равны нулю, и `False` — в противном случае:

```
>>> p = QtCore.QPoint()
>>> p.isNull()
    True
>>> p.setX(10); p.setY(88)
>>> p.x(), p.y()
    (10, 88)
```

- `manhattanLength()` — возвращает сумму абсолютных значений координат:

```
>>> QtCore.QPoint(10, 88).manhattanLength()
    98
```

Над двумя объектами класса `QPoint` можно выполнять операции `+`, `+=`, – (минус), `-=`, `==` и `!=`. Объект класса `QPoint` можно умножить или разделить на вещественное число (операторами `*`, `*=`, `/` и `/=`). Примеры:

```
>>> p1 = QtCore.QPoint(10, 20); p2 = QtCore.QPoint(5, 9)
>>> p1 + p2, p1 - p2
    (PyQt6.QtCore.QPoint(15, 29), PyQt6.QtCore.QPoint(5, 11))
>>> p1 * 2.5, p1 / 2.0
    (PyQt6.QtCore.QPoint(25, 50), PyQt6.QtCore.QPoint(5, 10))
>>> -p1, p1 == p2, p1 != p2
    (PyQt6.QtCore.QPoint(-10, -20), False, True)
```

19.4.2. Класс *QSize*: размеры прямоугольной области

Класс `QSize` описывает размеры прямоугольной области. Для создания объектов класса предназначены следующие форматы конструкторов:

```
QSize()
QSize(<Ширина>, <Высота>)
QSize(<Объект класса QSize>)
```

Первый формат создает невалидный объект с отрицательными шириной и высотой. Второй формат позволяет явно указать ширину и высоту. Третий формат создает копию указанного объекта. Примеры:

```
>>> from PyQt6 import QtCore
>>> s1 = QtCore.QSize(); s2 = QtCore.QSize(10, 55); s3 = QtCore.QSize(s2)
>>> s1
    PyQt6.QtCore.QSize(-1, -1)
>>> s2, s3
    (PyQt6.QtCore.QSize(10, 55), PyQt6.QtCore.QSize(10, 55))
```

Класс поддерживает следующие методы:

- `width()` — возвращает ширину;
- `height()` — возвращает высоту;
- `setWidth(<Ширина>)` — задает ширину;

- `setHeight(<Высота>)` — задает высоту.

  Примеры:

```
>>> s = QtCore.QSize()
>>> s.setWidth(10); s.setHeight(55)
>>> s.width(), s.height()
    (10, 55)
```

- `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;
- `isValid()` — возвращает `True`, если ширина и высота больше или равны нулю, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если ширина или высота меньше или равна нулю, и `False` — в противном случае;
- `scale()` — задает у текущей области новые размеры. Форматы метода:

```
scale(<Новые размеры QSize>, <Тип преобразования>)
scale(<Новая ширина>, <Новая высота>, <Тип преобразования>)
```

  В параметре `<Тип преобразования>` могут быть указаны следующие элементы перечисления `AspectRatioMode` из модуля `QtCore.Qt`:

  - `IgnoreAspectRatio` — не сохраняет пропорции сторон;
  - `KeepAspectRatio` — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
  - `KeepAspectRatioByExpanding` — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

  Если новая ширина или высота имеет значение 0, размеры изменяются без сохранения пропорций, вне зависимости от значения параметра `<Тип преобразования>`. Примеры:

```
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.AspectRatioMode.IgnoreAspectRatio); s
    PyQt6.QtCore.QSize(70, 60)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.AspectRatioMode.KeepAspectRatio); s
    PyQt6.QtCore.QSize(70, 28)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.AspectRatioMode.KeepAspectRatioByExpanding); s
    PyQt6.QtCore.QSize(150, 60)
```

- `scaled()` — то же самое, что `scale()`, только не изменяет текущий объект, а возвращает новый объект класса `QSize` с новыми размерами:

```
>>> s1 = QtCore.QSize(50, 20)
>>> s2 = s1.scaled(70, 60, QtCore.Qt.AspectRatioMode.IgnoreAspectRatio)
>>> s1, s2
    (PyQt6.QtCore.QSize(50, 20), PyQt6.QtCore.QSize(70, 60))
```

- `boundedTo(<Размеры QSize>)` — возвращает объект класса `QSize`, который содержит минимальную ширину и высоту из текущих размеров и размеров, указанных в параметре:

```
>>> s = QtCore.QSize(50, 20)
>>> s.boundedTo(QtCore.QSize(400, 5))
    PyQt6.QtCore.QSize(50, 5)
>>> s.boundedTo(QtCore.QSize(40, 50))
    PyQt6.QtCore.QSize(40, 20)
```

- `expandedTo(<Размеры QSize>)` — возвращает объект класса `QSize`, который содержит максимальную ширину и высоту из текущих размеров и размеров, указанных в параметре:

```
>>> s = QtCore.QSize(50, 20)
>>> s.expandedTo(QtCore.QSize(400, 5))
    PyQt6.QtCore.QSize(400, 20)
>>> s.expandedTo(QtCore.QSize(40, 50))
    PyQt6.QtCore.QSize(50, 50)
```

- `transpose()` — меняет ширину и высоту местами. Метод изменяет текущий объект и ничего не возвращает. Пример:

```
>>> s = QtCore.QSize(50, 20)
>>> s.transpose(); s
    PyQt6.QtCore.QSize(20, 50)
```

- `transposed()` — то же самое, что `transpose()`, только не изменяет текущий объект, а возвращает новый объект класса `QSize` с измененными размерами:

```
>>> s1 = QtCore.QSize(50, 20)
>>> s2 = s1.transposed()
>>> s1, s2
    (PyQt6.QtCore.QSize(50, 20), PyQt6.QtCore.QSize(20, 50))
```

Над двумя объектами класса `QSize` можно выполнять операции +, +=, – (минус), -=, == и !=. Кроме того, объект класса `QSize` можно умножить или разделить на вещественное число (операторами *, *=, / и /=). Примеры:

```
>>> s1 = QtCore.QSize(50, 20); s2 = QtCore.QSize(10, 5)
>>> s1 + s2, s1 - s2
    (PyQt6.QtCore.QSize(60, 25), PyQt6.QtCore.QSize(40, 15))
>>> s1 * 2.5, s1 / 2
    (PyQt6.QtCore.QSize(125, 50), PyQt6.QtCore.QSize(25, 10))
>>> s1 == s2, s1 != s2
    (False, True)
```

19.4.3. Класс *QRect*: координаты и размеры прямоугольной области

Класс `QRect` описывает координаты и размеры прямоугольной области. Для создания объектов класса предназначены следующие форматы конструктора:

```
QRect()
QRect(<X>, <Y>, <Ширина>, <Высота>)
QRect(<Координаты левого верхнего угла QPoint>, <Размеры QSize>)
QRect(<Координаты левого верхнего угла QPoint>,
      <Координаты правого нижнего угла QPoint>)
QRect(<Объект класса QRect>)
```

Первый формат создает объект с нулевыми координатами и размерами. Второй и третий форматы позволяют указать координаты левого верхнего угла и размеры области. Четвертый формат позволяет указать координаты левого верхнего угла и правого нижнего угла. Пятый конструктор создает копию заданного объекта.

Примеры:

```
>>> from PyQt6 import QtCore
>>> r = QtCore.QRect()
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
    (0, 0, -1, -1, 0, 0)
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
    (10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QSize(400, 300))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
    (10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QPoint(409, 314))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
    (10, 15, 409, 314, 400, 300)
>>> QtCore.QRect(r)
    PyQt6.QtCore.QRect(10, 15, 400, 300)
```

Изменить значения уже после создания объекта позволяют следующие методы:

- `setLeft(<X1>)` и `setX(<X1>)` — задает горизонтальную координату левого верхнего угла;
- `setTop(<Y1>)` и `setY(<Y1>)` — задает вертикальную координату левого верхнего угла:

```
>>> r = QtCore.QRect()
>>> r.setLeft(10); r.setTop(55); r
    PyQt6.QtCore.QRect(10, 55, -10, -55)
>>> r.setX(12); r.setY(81); r
    PyQt6.QtCore.QRect(12, 81, -12, -81)
```

- `setRight(<X2>)` — задает горизонтальную координату правого нижнего угла;
- `setBottom(<Y2>)` — задает вертикальную координату правого нижнего угла:

```
>>> r = QtCore.QRect()
>>> r.setRight(12); r.setBottom(81); r
    PyQt6.QtCore.QRect(0, 0, 13, 82)
```

- `setTopLeft(<Координаты QPoint>)` — задает координаты левого верхнего угла;
- `setTopRight(<Координаты QPoint>)` — задает координаты правого верхнего угла;
- `setBottomLeft(<Координаты QPoint>)` — задает координаты левого нижнего угла;
- `setBottomRight(<Координаты QPoint>)` — задает координаты правого нижнего угла.

Примеры:

```
>>> r = QtCore.QRect()
>>> r.setTopLeft(QtCore.QPoint(10, 5))
>>> r.setBottomRight(QtCore.QPoint(39, 19)); r
    PyQt6.QtCore.QRect(10, 5, 30, 15)
```

```
>>> r.setTopRight(QtCore.QPoint(39, 5))
>>> r.setBottomLeft(QtCore.QPoint(10, 19)); r
    PyQt6.QtCore.QRect(10, 5, 30, 15)
```

- `setWidth(<Ширина>)` — задает ширину;
- `setHeight(<Высота>)` — задает высоту;
- `setSize(<Размеры QSize>)` — задает ширину и высоту;
- `setRect(<X1>, <Y1>, <Ширина>, <Высота>)` — задает координаты левого верхнего угла и размеры области;
- `setCoords(<X1>, <Y1>, <X2>, <Y2>)` — задает координаты левого верхнего и правого нижнего углов.

  Примеры:

```
>>> r = QtCore.QRect()
>>> r.setRect(10, 10, 100, 500); r
    PyQt6.QtCore.QRect(10, 10, 100, 500)
>>> r.setCoords(10, 10, 109, 509); r
    PyQt6.QtCore.QRect(10, 10, 100, 500)
```

- `marginsAdded(<Границы QMargins>)` — возвращает новый объект класса `QRect`, который представляет текущую область, увеличенную на заданные величины границ. Границы указываются в виде объекта класса `QMargins` из модуля `QtCore`, конструктор которого имеет следующий формат:

```
QMargins(<Граница слева>, <Граница сверху>, <Граница справа>,
                                            <Граница снизу>)
```

  Текущий объект при этом не изменяется. Примеры:

```
>>> r1 = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(5, 2, 5, 2)
>>> r2 = r1.marginsAdded(m)
>>> r2
    PyQt6.QtCore.QRect(5, 13, 410, 304)
>>> r1
    PyQt6.QtCore.QRect(10, 15, 400, 300)
```

- `marginsRemoved()` — то же самое, что `marginsAdded()`, только уменьшает новую область на заданные величины границ:

```
>>> r1 = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(2, 10, 2, 10)
>>> r2 = r1.marginsRemoved(m)
>>> r2
    PyQt6.QtCore.QRect(12, 25, 396, 280)
>>> r1
    PyQt6.QtCore.QRect(10, 15, 400, 300)
```

- `moveTo()` — задает новые координаты левого верхнего угла. Форматы вызова:

```
moveTo(<X>, <Y>)
moveTo(<Координаты QPoint>)
```

- `moveLeft(<X>)` — задает новую горизонтальную координату левого верхнего угла;

- `moveTop(<Y>)` — задает новую вертикальную координату левого верхнего угла.

  Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTo(0, 0); r
    PyQt6.QtCore.QRect(0, 0, 400, 300)
>>> r.moveTo(QtCore.QPoint(10, 10)); r
    PyQt6.QtCore.QRect(10, 10, 400, 300)
>>> r.moveLeft(5); r.moveTop(0); r
    PyQt6.QtCore.QRect(5, 0, 400, 300)
```

- `moveRight(<X>)` — задает новую горизонтальную координату правого нижнего угла;
- `moveBottom(<Y>)` — задает новую вертикальную координату правого нижнего угла;
- `moveTopLeft(<Координаты QPoint>)` — задает новые координаты левого верхнего угла;
- `moveTopRight(<Координаты QPoint>)` — задает новые координаты правого верхнего угла;
- `moveBottomLeft(<Координаты QPoint>)` — задает новые координаты левого нижнего угла;
- `moveBottomRight(<Координаты QPoint>)` — задает новые координаты правого нижнего угла.

  Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTopLeft(QtCore.QPoint(0, 0)); r
    PyQt6.QtCore.QRect(0, 0, 400, 300)
>>> r.moveBottomRight(QtCore.QPoint(599, 499)); r
    PyQt6.QtCore.QRect(200, 200, 400, 300)
```

- `moveCenter(<Координаты QPoint>)` — задает новые координаты центра;
- `translate()` — смещает левый верхний угол на заданные значения. Форматы вызова:

```
translate(<Сдвиг по оси X>, <Сдвиг по оси Y>)
translate(<Величины сдвигов QPoint)
```

  Примеры:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.translate(20, 15); r
    PyQt6.QtCore.QRect(20, 15, 400, 300)
>>> r.translate(QtCore.QPoint(10, 5)); r
    PyQt6.QtCore.QRect(30, 20, 400, 300)
```

- `translated()` — аналогичен методу `translate()`, только возвращает новый объект класса `QRect` с новыми координатами, а не изменяет текущий;
- `adjust(<X1>, <Y1>, <X2>, <Y2>)` — смещает левый верхний и правый нижний углы на заданные значения:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.adjust(10, 5, 10, 5); r
    PyQt6.QtCore.QRect(10, 5, 400, 300)
```

- `adjusted()` — аналогичен методу `adjust()`, только возвращает новый объект класса `QRect`, а не изменяет текущий.

Для получения параметров области предназначены следующие методы:

- `left()` и `x()` — возвращают горизонтальную координату левого верхнего угла;
- `top()` и `y()` — возвращают вертикальную координату левого верхнего угла;
- `right()` — возвращает горизонтальную координату правого нижнего угла;
- `bottom()` — возвращает вертикальную координату правого нижнего угла;
- `width()` — возвращает ширину;
- `height()` — возвращает высоту;
- `size()` — возвращает размеры в виде объекта класса `QSize`.

  Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.x(), r.y(), r.right(), r.bottom()
    (10, 15, 10, 15, 409, 314)
>>> r.width(), r.height(), r.size()
    (400, 300, PyQt6.QtCore.QSize(400, 300))
```

- `topLeft()` — возвращает координаты левого верхнего угла в виде объекта класса `QPoint`;
- `topRight()` — возвращает координаты правого верхнего угла в виде объекта класса `QPoint`;
- `bottomLeft()` — возвращает координаты левого нижнего угла в виде объекта класса `QPoint`;
- `bottomRight()` — возвращает координаты правого нижнего угла в виде объекта класса `QPoint`.

  Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.topLeft(), r.topRight()
    (PyQt6.QtCore.QPoint(10, 15), PyQt6.QtCore.QPoint(409, 15))
>>> r.bottomLeft(), r.bottomRight()
    (PyQt6.QtCore.QPoint(10, 314), PyQt6.QtCore.QPoint(409, 314))
```

- `center()` — возвращает координаты центра области в виде объекта класса `QPoint`. Пример вывода окна в центре доступной области экрана:

```
screen = window.screen()
window.move(screen.availableGeometry().center() - window.rect().center())
```

- `getRect()` — возвращает кортеж с координатами левого верхнего угла и размерами области;
- `getCoords()` — возвращает кортеж с координатами левого верхнего и правого нижнего углов:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.getRect(), r.getCoords()
    ((10, 15, 400, 300), (10, 15, 409, 314))
```

Прочие методы:

- `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;

- `isValid()` — возвращает `True`, если `left() < right()` и `top() < bottom()`, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если `left() > right()` или `top() > bottom()`, и `False` — в противном случае;
- `normalized()` — исправляет ситуацию, при которой `left() > right()` или `top() > bottom()`, и возвращает новый объект класса `QRect` с исправленными координатами:

```
>>> r = QtCore.QRect(QtCore.QPoint(409, 314), QtCore.QPoint(10, 15))
>>> r
    PyQt6.QtCore.QRect(409, 314, -398, -298)
>>> r.normalized()
    PyQt6.QtCore.QRect(10, 15, 400, 300)
```

- `contains()` — возвращает `True`, если точка с указанными координатами расположена внутри текущей области или на ее границе, и `False` — в противном случае. Форматы вызова:

```
contains(<X>, <Y>[, <Флаг>])
contains(<Объект класса QPoint>[, <Флаг>])
```

Если во втором параметре указано значение `True`, то точка должна быть расположена строго внутри области, а не на ее границе (значение параметра по умолчанию — `False`). Пример:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(0, 10), r.contains(0, 10, True)
    (True, False)
```

- `contains(<Область QRect>[, <Флаг>])` — возвращает `True`, если указанная область расположена внутри текущей области или на ее краю, и `False` — в противном случае. Если во втором параметре указано значение `True`, то указанная область должна быть расположена строго внутри текущей области, а не на ее краю (значение параметра по умолчанию — `False`). Примеры:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(QtCore.QRect(0, 0, 20, 5))
    True
>>> r.contains(QtCore.QRect(0, 0, 20, 5), True)
    False
```

- `intersects(<Область QRect>)` — возвращает `True`, если указанная область пересекается с текущей, и `False` — в противном случае;
- `intersected(<Область QRect>)` — возвращает область, которая расположена на пересечении текущей и указанной областей, в виде объекта класса `QRect`:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.intersects(QtCore.QRect(10, 10, 20, 20))
    True
>>> r.intersected(QtCore.QRect(10, 10, 20, 20))
    PyQt6.QtCore.QRect(10, 10, 10, 10)
```

- `united(<Область QRect>)` — возвращает область, которая охватывает текущую и указанную области, в виде объекта класса `QRect`:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.united(QtCore.QRect(30, 30, 20, 20))
    PyQt6.QtCore.QRect(0, 0, 50, 50)
```

Над двумя объектами класса `QRect` можно выполнять операции `&` и `&=` (пересечение), `|` и `|=` (объединение), `in` (проверка на вхождение), `==` и `!=`:

```
>>> r1, r2 = QtCore.QRect(0, 0, 20, 20), QtCore.QRect(10, 10, 20, 20)
>>> r1 & r2, r1 | r2
    (PyQt6.QtCore.QRect(10, 10, 10, 10), PyQt6.QtCore.QRect(0, 0, 30, 30))
>>> r1 in r2, r1 in QtCore.QRect(0, 0, 30, 30)
    (False, True)
>>> r1 == r2, r1 != r2
    (False, True)
```

Также поддерживаются операторы `+` и `-`, выполняющие увеличение и уменьшение области на заданные величины границ, заданные в виде объекта класса `QMargins`:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(5, 15, 5, 15)
>>> r + m
    PyQt6.QtCore.QRect(5, 0, 410, 330)
>>> r - m
    PyQt6.QtCore.QRect(15, 30, 390, 270)
```

19.5. Разворачивание и сворачивание окон

Развернуть или свернуть окно позволяют следующие методы класса `QWidget`:

- `showMinimized()` — сворачивает окно;
- `showMaximized()` — разворачивает окно до максимального размера;
- `showFullScreen()` — включает полноэкранный режим отображения окна. Окно отображается без заголовка и границ;
- `showNormal()` — возвращает окно к изначальным размерам;
- `activateWindow()` — делает окно активным (т. е. имеющим фокус ввода). В Windows, если окно ранее было свернуто, оно не восстанавливается;
- `setWindowState(<Флаги>)` — изменяет состояние окна на основе указанных флагов. В их качестве указывается комбинация следующих элементов перечисления `WindowState` из модуля `QtCore.Qt` через оператор `|`:
  - `WindowNoState` — изначальное состояние окна;
  - `WindowMinimized` — окно свернуто;
  - `WindowMaximized` — окно максимально развернуто;
  - `WindowFullScreen` — полноэкранный режим;
  - `WindowActive` — окно активно.

  Например, включить полноэкранный режим можно так:

```
window.setWindowState((window.windowState() &
    ~(QtCore.Qt.WindowState.WindowMinimized |
```

```
        QtCore.Qt.WindowState.WindowMaximized))
    | QtCore.Qt.WindowState.WindowFullScreen)
```

Проверить текущее состояние окна позволяют следующие методы:

- `isMinimized()` — возвращает `True`, если окно свернуто, и `False` — в противном случае;
- `isMaximized()` — возвращает `True`, если окно развернуто, и `False` — в противном случае;
- `isFullScreen()` — возвращает `True`, если у окна включен полноэкранный режим, и `False` — в противном случае;
- `isActiveWindow()` — возвращает `True`, если окно активно, и `False` — в противном случае;
- `windowState()` — возвращает комбинацию флагов, обозначающих текущее состояние окна.

Пример проверки использования полноэкранного режима:

```
if window.windowState() & QtCore.Qt.WindowState.WindowFullScreen:
    print("Полноэкранный режим")
```

Пример разворачивания и сворачивания окна приведен в листинге 19.4.

Листинг 19.4. Разворачивание и сворачивание окна

```
from PyQt6 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.btnMin = QtWidgets.QPushButton("Свернуть")
        self.btnMax = QtWidgets.QPushButton("Развернуть")
        self.btnFull = QtWidgets.QPushButton("Полный экран")
        self.btnNormal = QtWidgets.QPushButton("Нормальный размер")
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.btnMin)
        vbox.addWidget(self.btnMax)
        vbox.addWidget(self.btnFull)
        vbox.addWidget(self.btnNormal)
        self.setLayout(vbox)
        self.btnMin.clicked.connect(self.on_min)
        self.btnMax.clicked.connect(self.on_max)
        self.btnFull.clicked.connect(self.on_full)
        self.btnNormal.clicked.connect(self.on_normal)
    def on_min(self):
        self.showMinimized()
    def on_max(self):
        self.showMaximized()
    def on_full(self):
        self.showFullScreen()
    def on_normal(self):
        self.showNormal()

if __name__ == "__main__":
    import sys
```

```
app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.setWindowTitle("Разворачивание и сворачивание окна")
window.resize(300, 100)
window.show()
sys.exit(app.exec())
```

19.6. Управление прозрачностью окна

Сделать окно полупрозрачным позволяет метод `setWindowOpacity()` окна. Формат метода:

```
setWindowOpacity(<Вещественное число от 0.0 до 1.0>)
```

Число `0.0` соответствует полностью прозрачному окну, а число `1.0` — отсутствию прозрачности. Для получения степени прозрачности окна предназначен метод `windowOpacity()`. Выведем окно со степенью прозрачности `0.5` (листинг 19.5).

Листинг 19.5. Полупрозрачное окно

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Полупрозрачное окно")
window.resize(300, 100)
window.setWindowOpacity(0.5)
window.show()
print(window.windowOpacity()) # Выведет: 0.4980392156862745
sys.exit(app.exec())
```

19.7. Модальные окна

Модальным называется окно, которое перекрывает другие окна программы и не дает пользователю взаимодействовать с ними. Модальными обычно делают всевозможные диалоговые окна.

Сделать окно модальным позволяет метод `setWindowModality(<Флаг>)` окна. В качестве параметра могут быть указаны следующие элементы перечисления `WindowModality` из модуля `QtCore.Qt`:

- `NonModal` — окно не является модальным (поведение по умолчанию);
- `WindowModal` — окно блокирует только родительские окна в пределах иерархии;
- `ApplicationModal` — окно блокирует все окна программы.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

Получить текущее значение модальности позволяет метод `windowModality()`. Проверить, является ли окно модальным, можно с помощью метода `isModal()` — он возвращает `True`, если окно модально, и `False` — в противном случае.

Создадим два независимых окна. В первом окне разместим кнопку, по нажатии которой откроется модальное окно — оно будет блокировать только первое окно, но не второе. Модальное окно выведем примерно по центру родительского окна (листинг 19.6).

Листинг 19.6. Модальные окна

```
from PyQt6 import QtCore, QtWidgets
import sys

def show_modal_window():
    global modalWindow
    modalWindow = QtWidgets.QWidget(window1, QtCore.Qt.WindowType.Window)
    modalWindow.setWindowTitle("Модальное окно")
    modalWindow.resize(200, 50)
    modalWindow.setWindowModality(QtCore.Qt.WindowModality.WindowModal)
    modalWindow.setAttribute(QtCore.Qt.WidgetAttribute.WA_DeleteOnClose, True)
    modalWindow.move(window1.geometry().center() -
                     modalWindow.rect().center() - QtCore.QPoint(4, 30))
    modalWindow.show()

app = QtWidgets.QApplication(sys.argv)
window1 = QtWidgets.QWidget()
window1.setWindowTitle("Обычное окно")
window1.resize(300, 100)
button = QtWidgets.QPushButton("Открыть модальное окно")
button.clicked.connect(show_modal_window)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(button)
window1.setLayout(vbox)
window1.show()

window2 = QtWidgets.QWidget()
window2.setWindowTitle("Это окно не будет блокировано")
window2.resize(500, 100)
window2.show()

sys.exit(app.exec())
```

Если запустить программу и нажать кнопку **Открыть модальное окно**, откроется окно, выровненное примерно по центру родительского окна. При этом получить доступ к родительскому окну можно только после закрытия модального окна. Второе окно блокировано не будет. Однако, если заменить элемент `WindowModal` перечисления `WindowModality` элементом `ApplicationModal`, оба окна будут блокированы.

Обратите внимание, что в конструктор модального окна мы передали ссылку на первое окно и элемент `Window` перечисления `WindowType`. Если не указать ссылку, то первое окно блокироваться не будет, а если не указать элемент `Window`, окно вообще не откроется. Кроме того, мы объявили переменную `modalWindow` глобальной, иначе при достижении конца функции переменная выйдет из области видимости и окно будет автоматически удалено. Чтобы объект окна автоматически удалялся при закрытии окна, атрибуту `WA_DeleteOnClose` в методе `setAttribute()` было присвоено значение `True`.

PyQt предоставляет класс QDialog, который предназначен для создания диалоговых окон, самостоятельно делает окно модальным и выводит его в центре экрана или родительского окна. Кроме того, этот класс предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус завершения и выполнить другие действия. Подробно класс QDialog мы рассмотрим в *главе 27*.

19.8. Смена значка окна

Сменить значок, выводящийся в заголовке окна, позволяет метод setWindowIcon() окна. В качестве параметра метод принимает объект класса QIcon из модуля QtGui (описан в *разд. 25.3.4*).

Конструктору класса QIcon передается путь к графическому файлу со значком. Получить список поддерживаемых графических форматов можно вызовом статического метода supportedImageFormats() класса QImageReader из модуля QtGui. Метод возвращает список с объектами класса QByteArray. Получим список поддерживаемых форматов:

```
>>> from PyQt6 import QtGui
>>> for i in QtGui.QImageReader.supportedImageFormats():
...     print(str(i, "ascii").upper(), end=" ")
    BMP CUR GIF ICNS ICO JPEG JPG PBM PGM PNG PPM SVG SVGZ TGA TIF TIFF WBMP
    WEBP XBM XPM
```

Если у окна не указать значок, будет использоваться значок уровня программы, установленный с помощью метода setWindowIcon(<Значок QIcon>) объекта программы. Получить текущий значок программы можно вызовом у того же объекта метода windowIcon(), который возвращает объект класса QIcon.

Вместо загрузки значка из файла можно воспользоваться одним из встроенных значков. Список всех встроенных значков приведен на странице **https://doc.qt.io/qt-6/qstyle.html #StandardPixmap-enum**. Пример:

```
ico = window.style().standardIcon(
                    QtWidgets.QStyle.StandardPixmap.SP_MessageBoxCritical)
window.setWindowIcon(ico)
```

Создадим значок размером 16 на 16 пикселов в формате PNG, сохраним его в одном каталоге с программой и установим у окна и всей программы (листинг 19.7).

Листинг 19.7. Смена значка окна

```
from PyQt6 import QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Смена значка в заголовке окна")
window.resize(300, 100)
ico = QtGui.QIcon("icon.png")
window.setWindowIcon(ico)              # Значок для окна
app.setWindowIcon(ico)                 # Значок программы
window.show()
sys.exit(app.exec())
```

19.9. Изменение цвета фона окна

Изменить цвет фона окна (или компонента) можно через его палитру. *Палитра* — это набор цветов для различных составных частей и состояний окна.

Получить текущую палитру окна позволяет его метод `palette()`. Палитра возвращается в виде объекта класса `QPalette` из модуля `QtGui`.

Чтобы изменить цвет для какой-либо роли и состояния, следует воспользоваться методом `setColor()` палитры. Формат метода:

```
setColor([<Состояние>, ]<Составная часть окна>, <Цвет>)
```

Состояние окна указывается в виде одного из следующих элементов перечисления `ColorGroup` из класса `QPalette`:

- `Active` и `Normal` — окно активно;
- `Disabled` — окно недоступно;
- `Inactive` — окно неактивно.

Составная часть окна задается в виде одного из элементов перечисления `ColorRole` из класса `QPalette`. Так, элемент `Window` (или `Background`) обозначает фон окна, а элемент `WindowText` (или `Foreground`) — текста. Полный список элементов перечисления `ColorRole` имеется по интернет-адресу **https://doc.qt.io/qt-6/qpalette.html#ColorRole-enum**.

Цвет указывается в виде элемента перечисления `GlobalColor` из модуля `QtCore.Qt` (скажем, `black`) или объекта класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.).

После настройки палитры необходимо вызвать метод `setPalette(<Палитра>)` окна и передать этому методу измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку, необходимо передать значение `True` методу `setAutoFillBackground(<Флаг>)` окна.

Изменить цвет фона также можно с помощью CSS-атрибута `background-color`. Для этого следует передать таблицу стилей в метод `setStyleSheet(<Таблица стилей>)` компонента. Таблицы стилей могут быть внешними (подключение через командную строку), установленными на уровне программы (с помощью метода `setStyleSheet()` класса `QApplication`) или установленными на уровне компонента (с помощью метода `setStyleSheet()` класса `QWidget`). Атрибуты, установленные последними, обычно перекрывают значения аналогичных атрибутов, указанных ранее.

Создадим окно с надписью. У активного окна установим зеленый цвет, а у неактивного — красный. Цвет фона надписи сделаем белым. Для изменения фона окна используем палитру, а для изменения цвета фона надписи — CSS-атрибут `background-color` (листинг 19.8).

Листинг 19.8. Изменение цвета фона окна

```
from PyQt6 import QtCore, QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Изменение цвета фона окна")
window.resize(300, 100)
```

```
pal = window.palette()
pal.setColor(QtGui.QPalette.ColorGroup.Normal, QtGui.QPalette.ColorRole.Window,
             QtGui.QColor("#008800"))
pal.setColor(QtGui.QPalette.ColorGroup.Inactive,
             QtGui.QPalette.ColorRole.Window, QtGui.QColor("#ff0000"))
window.setPalette(pal)
label = QtWidgets.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignmentFlag.AlignHCenter)
label.setStyleSheet("background-color: #ffffff;")
label.setAutoFillBackground(True)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec())
```

19.10. Вывод изображения в качестве фона

В качестве фона окна (или компонента) можно использовать изображение. Для этого необходимо получить текущую палитру компонента с помощью метода `palette()`, а затем вызвать метод `setBrush()` объекта палитры. Формат метода:

```
setBrush([<Состояние>, ]<Составная часть окна>, <Кисть QBrush>)
```

Первые два параметра аналогичны таковым у метода `setColor()` (см. *разд. 19.9*). В третьем параметре указывается кисть — объект класса `QBrush` из модуля `QtGui`. Форматы конструктора класса:

```
QBrush(<Стиль кисти>)
QBrush(<Цвет>[, <Стиль кисти>=BrushStyle.SolidPattern])
QBrush(<Цвет>, <Изображение QPixmap>)
QBrush(<Изображение QPixmap>)
QBrush(<Изображение QImage>)
QBrush(<Градиент QGradient>)
QBrush(<Объект класса QBrush>)
```

В качестве стиля кисти указывается один из элементов перечисления `BrushStyle` из модуля `QtCore.Qt`: `NoBrush`, `SolidPattern`, `Dense1Pattern`, `Dense2Pattern`, `Dense3Pattern`, `Dense4Pattern`, `Dense5Pattern`, `Dense6Pattern`, `Dense7Pattern`, `CrossPattern` и др. С помощью этого параметра можно сделать цвет сплошным (`SolidPattern`) или имеющим текстуру (например, элемент `CrossPattern` задает текстуру в виде сетки).

Цвет можно задать в виде элемента перечисления `GlobalColor` из модуля `QtCore.Qt` (например, `black`, `white` и т. д.) или объекта класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). При этом установка сплошного цвета фона окна может выглядеть так:

```
pal = window.palette()
pal.setBrush(QtGui.QPalette.ColorGroup.Normal, QtGui.QPalette.ColorRole.Window,
    QtGui.QBrush(QtGui.QColor("#008800"), QtCore.Qt.BrushStyle.SolidPattern))
window.setPalette(pal)
```

Третий, четвертый и пятый форматы позволяют указать изображение в виде объекта класса `QPixmap` или `QImage`. Конструкторы этих классов принимают путь к файлу с нужным изображением.

Шестой формат создает новую кисть на основе градиента, представленного объектом класса `QGradient` (см. *главу 25*), а седьмой — на основе указанной кисти.

После настройки палитры у окна также следует вызвать метод `setPalette()` (см. *разд. 19.9*), передав ему измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку, необходимо передать значение `True` в метод `setAutoFillBackground()`.

Указать фоновое изображение также можно с помощью CSS-атрибутов `background` и `background-image`. Воспользовавшись CSS-атрибутом `background-repeat`, можно задать режим повтора фонового рисунка: `repeat` (повтор по горизонтали и вертикали), `repeat-x` (только по горизонтали), `repeat-y` (только по вертикали) и `no-repeat` (не повторяется).

Создадим окно с надписью. Для активного окна установим одно изображение (с помощью изменения палитры), а для надписи — другое (с помощью CSS-атрибута `background-image`) (листинг 19.9).

Листинг 19.9. Использование изображения в качестве фона

```
from PyQt6 import QtCore, QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Изображение в качестве фона")
window.resize(300, 100)
pal = window.palette()
pal.setBrush(QtGui.QPalette.ColorGroup.Normal, QtGui.QPalette.ColorRole.Window,
             QtGui.QBrush(QtGui.QPixmap("backgroundl.jpg")))
window.setPalette(pal)
label = QtWidgets.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignmentFlag.AlignCenter)
label.setStyleSheet("background-image: url(background2.jpg);")
label.setAutoFillBackground(True)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec())
```

19.11. Окна произвольной формы

Чтобы создать окно произвольной формы, нужно выполнить следующие шаги:

1. Создать изображение нужной формы с прозрачным фоном и сохранить его, например, в формате PNG.
2. Создать объект класса `QPixmap`, передав конструктору класса путь к файлу с ранее созданным изображением.

3. Установить изображение в качестве фона окна с помощью палитры (см. *разд. 19.10*).

   Если для создания окна используется класс надписи `QLabel`, то установить изображение в качестве фона можно вызовом метода `setPixmap(<Изображение QPixmap>)` надписи.

4. Создать на основе изображения битовую маску с помощью метода `mask()` класса `QPixmap`.
5. Применить полученную маску к окну, вызвав у него метод `setMask(<Маска>)`.
6. Убрать рамку у окна, например, передав комбинацию следующих элементов перечисления `WindowType`:

```
QtCore.Qt.WindowType.Window | QtCore.Qt.WindowType.FramelessWindowHint
```

Для примера создадим круглое окно с кнопкой, закрывающей его. Для использования в качестве маски создадим PNG-изображение в виде белого круга с прозрачным фоном. Окно выведем без заголовка и границ (листинг 19.10).

Листинг 19.10. Создание окна произвольной формы

```
from PyQt6 import QtCore, QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowFlags(QtCore.Qt.WindowType.Window |
                      QtCore.Qt.WindowType.FramelessWindowHint)
window.setWindowTitle("Создание окна произвольной формы")
window.resize(300, 300)
pixmap = QtGui.QPixmap("mask.png")
pal = window.palette()
pal.setBrush(QtGui.QPalette.ColorGroup.Normal, QtGui.QPalette.ColorRole.Window,
             QtGui.QBrush(pixmap))
pal.setBrush(QtGui.QPalette.ColorGroup.Inactive,
             QtGui.QPalette.ColorRole.Window, QtGui.QBrush(pixmap))
window.setPalette(pal)
window.setMask(pixmap.mask())
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 135)
button.clicked.connect(QtWidgets.QApplication.instance().quit)
window.show()
sys.exit(app.exec())
```

Получившееся окно можно увидеть на рис. 19.1.

Рис. 19.1. Окно круглой формы

19.12. Всплывающие и расширенные подсказки

Всплывающая подсказка содержит справочную информацию о каком-либо компоненте и выводится на экран при наведении курсора мыши на этот компонент.

Расширенная подсказка по компоненту выводится после нажатия кнопки вывода справки в заголовке диалогового окна и щелчка мышью на этом компоненте, также можно сделать компонент активным нажатием комбинации клавиш <Shift>+<F1>.

Для создания всплывающих подсказок и управления ими служат следующие методы класса `QWidget`:

- `setToolTip(<Текст>)` — задает текст всплывающей подсказки. В качестве параметра можно указать простой текст или HTML-код. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;
- `toolTip()` — возвращает текст всплывающей подсказки;
- `setToolTipDuration(<Время>)` — задает время, в течение которого всплывающая подсказка будет присутствовать на экране, в миллисекундах. Если задать значение `-1`, PyQt будет сама вычислять необходимое время, основываясь на длине текста подсказки (поведение по умолчанию);
- `toolTipDuration()` — возвращает время, в течение которого всплывающая подсказка будет присутствовать на экране;
- `setWhatsThis(<Текст>)` — задает текст расширенной подсказки. В качестве параметра можно указать простой текст или HTML-код. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;
- `whatsThis()` — возвращает текст расширенной подсказки.

Создадим окно с кнопкой и зададим для них текст всплывающих и расширенных подсказок (листинг 19.11).

Листинг 19.11. Всплывающие и расширенные подсказки

```
from PyQt6 import QtCore, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget(flags=QtCore.Qt.WindowType.Dialog |
                           QtCore.Qt.WindowType.WindowContextHelpButtonHint)
window.setWindowTitle("Всплывающие и расширенные подсказки")
window.resize(300, 70)
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
button.setToolTip("Это всплывающая подсказка для кнопки")
button.setToolTipDuration(3000)
window.setToolTip("Это всплывающая подсказка для окна")
button.setToolTipDuration(5000)
button.setWhatsThis("Это расширенная подсказка для кнопки")
window.setWhatsThis("Это расширенная подсказка для окна")
button.clicked.connect(QtWidgets.QApplication.instance().quit)
window.show()
sys.exit(app.exec())
```

19.13. Программное закрытие окна

Чтобы закрыть окно, у него следует вызвать метод `close()`. Он возвращает значение `True`, если окно успешно закрыто, и `False` — в противном случае. Закрыть сразу все окна программы позволяет слот `closeAllWindows()` объекта программы.

Если у окна элемент `WA_DeleteOnClose` перечисления `WidgetAttribute` из модуля `QtCore.Qt` установлен в значение `True`, после закрытия окна его объект будет автоматически удален, в противном случае окно просто скроется. Установить или сбросить элемент можно с помощью метода `setAttribute()` окна, например:

```
window.setAttribute(QtCore.Qt.WidgetAttribute.WA_DeleteOnClose, True)
```

После вызова метода `close()` или нажатия кнопки закрытия в заголовке окна возникает событие `QEvent.Close`. Если внутри класса определить метод `closeEvent()`, это событие можно перехватить и обработать. В качестве параметра метод принимает объект класса `QCloseEvent`, который поддерживает методы `accept()` (позволяет закрыть окно) и `ignore()` (запрещает закрытие окна). Вызывая эти методы, можно контролировать процесс закрытия окна.

В качестве примера закроем окно по нажатии кнопки (листинг 19.12).

Листинг 19.12. Программное закрытие окна

```
from PyQt6 import QtCore, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget(flags=QtCore.Qt.WindowType.Dialog)
window.setWindowTitle("Программное закрытие окна ")
window.resize(300, 70)
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
button.clicked.connect(window.close)
window.show()
sys.exit(app.exec())
```

Внимание!

После закрытия последнего окна программы выполнение программы завершится.

19.14. Использование таблиц стилей CSS для оформления окон

Задавать оформление у компонентов можно также с помощью таблиц стилей CSS. Это позволяет сделать метод `setStyleSheet(<Таблица стилей>)`, упоминавшийся в *разд. 19.9*, который можно вызвать:

- у самой программы — тогда заданные в таблице стилей параметры оформления будут применены ко всем компонентам всех окон программы;
- у отдельного окна — тогда эти параметры будут действовать в пределах этого окна;
- у отдельного компонента — тогда они будут действовать только на этот компонент.

При указании таблицы стилей у программы и окна можно использовать привычный формат объявления CSS-стилей:

```
<Селектор> {<Определение стилей>}
```

<Селектор> записывается в следующем формате:

```
<Основной селектор>[<Дополнительный селектор>][<Псевдокласс>][<Псевдоселектор>]
```

Параметр <Основной селектор> указывает на класс компонента. Его можно задать в одном из следующих форматов:

- * (звездочка) — указывает на все компоненты. Например, так можно задать у всех компонентов зеленый цвет текста:

```
* {color: green;}
```

- <Класс> — указывает на компоненты, относящиеся к заданному <Классу> и его подклассам. Задание красного цвета текста у всех компонентов, относящихся к классу QAbstractButton и его подклассам (т. е. у командных кнопок, флажков и переключателей), осуществляется так:

```
QAbstractButton {color: red;}
```

- .<Класс> — указывает только на компоненты, относящиеся лишь к заданному <Классу>, но не к его подклассам. Указание полужирного шрифта у всех компонентов класса QPushButton (командных кнопок), но не у его подклассов, осуществляется так:

```
.QPushButton {font-weight: bold;}
```

Параметр <Дополнительный селектор> задает дополнительные параметры компонента. Его форматы:

- [<Свойство>="<Значение>"] — указанное <Свойство> компонента должно иметь заданное <Значение>. Полужирный шрифт у кнопки, чье свойство default имеет значение true (кнопки по умолчанию), задается так:

```
QPushButton[default="true"] {font-weight: bold;}
```

- #<Имя> — указывает на компонент с заданным <Именем>. <Имя> можно задать вызовом у компонента метода setObjectName(<Имя>), а получить — вызовом метода objectName(). Красный цвет текста у кнопки с именем btnRed задается так:

```
QPushButton#btnRed {color: red;}
```

Параметр <Псевдокласс> указывает на отдельную составную часть сложного компонента. Он записывается в формате ::<Обозначение составной части>. Вот пример указания графического изображения у кнопки разворачивания раскрывающегося списка (обозначение этой составной части — down-arrow):

```
QComboBox::down-arrow {image: url(arrow.png);}
```

Параметр <Псевдоселектор> указывает на состояние компонента (должна ли быть кнопка нажата, должен ли флажок быть установленным и т. п.). Он может быть записан в двух форматах:

- :<Обозначение состояния> — компонент должен находиться в указанном состоянии. Пример указания белого цвета фона у кнопки, когда она нажата (это состояние имеет обозначение pressed):

```
QPushButton:pressed {background-color: white;}
```

- `:!<Обозначение состояния>` — компонент должен находиться в любом состоянии, кроме указанного. Пример указания желтого цвета фона у кнопки, когда она не нажата:

  ```
  QPushButton:!pressed {background-color: yellow;}
  ```

  Можно указать сразу несколько псевдоселекторов, расположив их непосредственно друг за другом — тогда селектор будет указывать на компонент, находящийся одновременно во всех состояниях, которые обозначены этими селекторами. Вот пример указания черного цвета фона и белого цвета текста у кнопки, которая нажата и над которой находится курсор мыши (обозначение — `hover`):

  ```
  QPushButton:pressed:hover {color: white; background-color: black;}
  ```

Если нужно указать стиль компонента, вложенного в другой компонент, применяется следующий формат указания селектора:

```
<Селектор «внешнего» компонента><Разделитель><Селектор вложенного компонента>
```

Поддерживаются два варианта параметра `<Разделитель>`:

- пробел — `<Вложенный компонент>` не обязательно должен быть вложен непосредственно во `<<Внешний>>`. Так мы указываем зеленый цвет фона у всех надписей (`QLabel`), вложенных в группу (`QGroupBox`), и вложенные в нее компоненты:

  ```
  QGroupBox QLabel {background-color: green;}
  ```

- `>` — `<Вложенный компонент>` обязательно должен быть вложен непосредственно во `<<Внешний>>`. Так мы укажем синий цвет текста у всех надписей, непосредственно вложенных в группу:

  ```
  QGroupBox>QLabel {color: blue;}
  ```

В стиле можно указать сразу несколько селекторов, записав их через запятую, — тогда стиль будет применен к компонентам, на которые указывают эти селекторы. Вот пример задания зеленого цвета фона у кнопок и надписей:

```
QLabel, QPushButton {background-color: green;}
```

В PyQt компоненты *не* наследуют параметры оформления от их родителей. Скажем, если мы укажем у группы красный цвет текста:

```
app.setStyleSheet("QGroupBox {color: red;}")
```

вложенные в эту группу компоненты не унаследуют его и будут иметь цвет текста, заданный по умолчанию. Придется задать для них нужный цвет явно:

```
app.setStyleSheet("QGroupBox, QGroupBox * {color: red;}")
```

Есть возможность указать компонентам-потомкам наследовать параметры оформления у родителя. Для этого достаточно вызвать у объекта программы статический метод `setAttribute()`, передав ему в качестве первого параметра элемент `AA_UseStyleSheetPropagationInWidgetStyles` перечисления `ApplicationAttribute` из модуля `QtCore.Qt`, а в качестве второго параметра — значение `True`:

```
QtWidgets.QApplication.setAttribute(
      QtCore.Qt.ApplicationAttribute.AA_UseStyleSheetPropagationInWidgetStyles,
      True)
```

Чтобы отключить такую возможность, достаточно вызвать этот метод еще раз, указав в нем вторым параметром `False`.

При вызове метода `setStyleSheet()` у компонента в назначаемой ему таблице стилей не указываются ни селектор, ни фигурные скобки — они просто не нужны.

В случае PyQt, как и в CSS, также действуют правила каскадности. Так, таблица стилей, заданная у окна, имеет больший приоритет, нежели таковая, указанная у программы, а стиль, что был задан у компонента, имеет наивысший приоритет. Более специфические стили имеют больший приоритет, чем менее специфические, — так, стиль с селектором, в состав которого входит имя компонента, перекроет стиль с селектором любого другого типа.

За более подробным описанием поддерживаемых PyQt псевдоклассов, псевдоселекторов и особенностей указания стилей для отдельных классов компонентов обращайтесь по интернет-адресу **https://doc.qt.io/qt-6/stylesheet-reference.html**.

В листинге 19.13 показан пример задания таблиц стилей у компонентов разными способами. Результат выполнения приведенного кода можно увидеть на рис. 19.2.

Листинг 19.13. Использование таблиц стилей для указания оформления

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
# На уровне программы задаем синий цвет текста у надписей, вложенных в группы,
# и курсивное начертание текста кнопок
app.setStyleSheet(
            "QGroupBox QLabel {color: blue;} QPushButton {font-style: italic}")
window = QtWidgets.QWidget()
window.setWindowTitle("Таблицы стилей")
# На уровне окна задаем зеленый цвет текста у надписи с именем first и
# красный цвет текста у надписи, на которую наведен курсор мыши
window.setStyleSheet("QLabel#first {color: green;} QLabel:hover {color: red;}")
window.resize(200, 150)
# Создаем три надписи
lbl1 = QtWidgets.QLabel("Зеленый текст")
# Даем первой надписи имя first
lbl1.setObjectName("first")
lbl2 = QtWidgets.QLabel("Полужирный текст")
# У второй надписи указываем полужирный шрифт
lbl2.setStyleSheet("font-weight: bold")
lbl3 = QtWidgets.QLabel("Синий текст")
# Создаем кнопку
btn = QtWidgets.QPushButton("Курсивный текст")
# Создаем группу
box = QtWidgets.QGroupBox("Группа")
# Создаем контейнер, помещаем в него третью надпись и вставляем в группу
bbox = QtWidgets.QVBoxLayout()
bbox.addWidget(lbl3)
box.setLayout(bbox)
# Создаем еще один контейнер, помещаем в него две первые надписи, группу,
# кнопку и вставляем в окно
mainbox = QtWidgets.QVBoxLayout()
mainbox.addWidget(lbl1)
```

```
mainbox.addWidget(lbl2)
mainbox.addWidget(box)
mainbox.addWidget(btn)
window.setLayout(mainbox)
window.show()
sys.exit(app.exec())
```

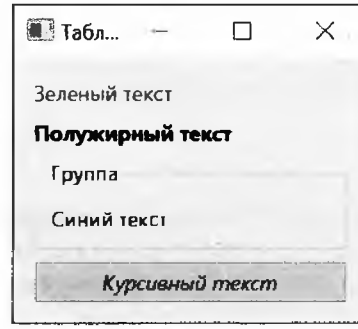


Рис. 19.2. Пример использования таблиц стилей

ГЛАВА 20

Обработка сигналов и событий

Событие — это уведомление о том, что в программе что-либо произошло: пользователь щелкнул мышью, нажал клавишу на клавиатуре, нажал кнопку и др. *Обработка события* — это реагирование на него заданным образом (например, запуск действия, закрепленного за нажатой кнопкой).

Обработка событий разных типов в PyQt выполняется двумя способами:

- низкоуровневые события (нажатие кнопки мыши, клавиши, перемещение мыши и т. п.) — посредством *обработчиков событий*, которые представляют собой особые специальные методы, определенные в классе, в котором могут возникать события (окне или каком-либо компоненте);
- высокоуровневые события (например, нажатие экранной кнопки) — приводят к генерированию в классе, в котором возникло событие, соответствующего *сигнала* (представления события внутри PyQt) и вызову назначенных ему обработчиков сигнала. *Обработчики сигналов* реализуются в виде *слотов* — особым образом помеченных методов класса, в которых возникают события, или функций.

Класс, в котором возникло событие, называется *источником события*, а класс, в котором был сгенерирован соответствующий событию сигнал, — *источником сигнала*. Класс, в котором выполняется обработка события или сигнала, носит название *приемника* соответственно *события* и *сигнала*.

20.1. Назначение обработчиков сигналов

Каждый сигнал, реализованный в классе, представляется в виде особого объекта и хранится в одноименном атрибуте этого класса (не обычном атрибуте!). Так, сигнал `clicked()`, генерируемый при щелчке мышью, соответствует атрибуту `clicked`.

Чтобы назначить сигналу обработчик, следует использовать метод `connect()` компонента. Форматы вызова этого метода таковы:

```
<Сигнал>.connect(<Обработчик>[,
    <Тип соединения>=ConnectionType.AutoConnection][, no_receiver_check=False])
<Сигнал>[<Тип>].connect(<Обработчик>[,
    <Тип соединения>=ConnectionType.AutoConnection][, no_receiver_check=False])
```

В качестве обработчика можно указать:

- ссылку на пользовательскую функцию;
- ссылку на метод класса;
- ссылку на объект класса, в котором определен метод `__call__()`;
- анонимную функцию;
- ссылку на слот класса.

Остальные два параметра будут описаны позже.

Вот пример назначения функции `on_clicked_button()` в качестве обработчика сигнала `clicked` кнопки `button`:

```
button.clicked.connect(on_clicked_button)
```

Сигналы могут принимать произвольное число параметров, каждый из которых может относиться к любому типу данных. При этом бывает и так, что в классе существуют два сигнала с одинаковыми наименованиями, но разными наборами параметров. Тогда следует дополнительно в квадратных скобках указать `<Тип>` данных, принимаемых сигналом, в виде либо ссылки на его класс, либо в виде строкового имени. Например, оба следующих выражения назначают обработчик сигнала, принимающего один параметр логического типа:

```
button.clicked[bool].connect(on_clicked_button)
button.clicked["bool"].connect(on_clicked_button)
```

Одному и тому же сигналу можно назначить произвольное количество обработчиков. Это иллюстрирует код из листинга 20.1, где сигналу `clicked` кнопки назначены сразу четыре обработчика.

Листинг 20.1. Назначение сигналу нескольких обработчиков

```
from PyQt6 import QtWidgets
import sys

def on_clicked():
    print("Кнопка нажата. Функция on_clicked()")
class MyClass():
    def __init__(self, x=0):
        self.x = x
    def __call__(self):
        print("Кнопка нажата. Метод MyClass.__call__()")
        print("x =", self.x)
    def on_clicked(self):
        print("Кнопка нажата. Метод MyClass.on_clicked()")

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Нажми меня")
# Назначаем обработчиком функцию
button.clicked.connect(on_clicked)
# Назначаем обработчиком метод класса
button.clicked.connect(obj.on_clicked)
```

```
# Назначаем обработчиком ссылку на объект класса
button.clicked.connect(MyClass(10))
# Назначаем обработчиком анонимную функцию
button.clicked.connect(lambda: MyClass(5)())
button.show()
sys.exit(app.exec())
```

В четвертом обработчике мы использовали анонимную функцию. В ней мы сначала создаем объект класса `MyClass`, передав ему в качестве параметра число 5, после чего сразу же вызываем его как функцию, указав после конструктора пустые скобки:

```
button.clicked.connect(lambda: MyClass(5)())
```

Результат выполнения в окне консоли при щелчке на кнопке:

```
Кнопка нажата. Функция on_clicked()
Кнопка нажата. Метод MyClass.on_clicked()
Кнопка нажата. Метод MyClass.__call__()
x = 10
Кнопка нажата. Метод MyClass.__call__()
x = 5
```

Тип соединения в вызове метода `connect()` следует указывать при использовании нескольких потоков. Можно указать один из следующих элементов перечисления `ConnectionType` из модуля `QtCore.Qt`:

- `AutoConnection` — если источник сигнала и обработчик находятся в одном потоке, то он эквивалентен элементу `DirectConnection`, а если в разных потоках, то — `QueuedConnection`;
- `DirectConnection` — обработчик вызывается сразу после генерирования сигнала и выполняется в потоке его источника;
- `QueuedConnection` — сигнал помещается в очередь обработки событий, а его обработчик выполняется в потоке приемника сигнала;
- `BlockingQueuedConnection` — аналогичен элементу `QueuedConnection` за тем исключением, что поток блокируется на время обработки сигнала. Обратите внимание, что источник и обработчик сигнала обязательно должны выполняться в разных потоках;
- `UniqueConnection` — обработчик можно назначить только один раз. Этот элемент с помощью оператора `|` может быть объединен с любым из представленных. Примеры:

```
# Эти два обработчика будут успешно назначены и выполнены
button.clicked.connect(on_clicked)
button.clicked.connect(on_clicked)
# А эти два обработчика назначены не будут
button.clicked.connect(on_clicked, QtCore.Qt.AutoConnection | \
                                   QtCore.Qt.UniqueConnection)
button.clicked.connect(on_clicked, QtCore.Qt.AutoConnection | \
                                   QtCore.Qt.UniqueConnection)
# Тем не менее эти два обработчика будут назначены, поскольку они разные
button.clicked.connect(on_clicked, QtCore.Qt.AutoConnection | \
                                   QtCore.Qt.UniqueConnection)
button.clicked.connect(obj.on_clicked, QtCore.Qt.AutoConnection | \
                                       QtCore.Qt.UniqueConnection)
```

- SingleShotConnection — обработчик выполнится всего один раз, после чего его назначение сигналу будет автоматически отменено. Этот элемент с помощью оператора | может быть объединен с любым из представленных. Пример:

```
# Назначенный обработчик будет выполнен лишь единожды
button.clicked.connect(on_once_clicked, QtCore.Qt.AutoConnection | \
                                        QtCore.Qt.SingleShotConnection)
```

Если параметру no_receiver_check дано значение False, то в случае, если приемник сигнала уже не существует, возникнет ошибка. Если же параметру дано значение True, то в таком случае ошибка не возникнет.

20.1.1. Слоты

Классы PyQt уже содержат ряд слотов, которые могут быть использованы в качестве обработчиков сигналов. Например, класс QApplication поддерживает слот quit(), завершающий программу. В листинге 19.2 показан пример его использования.

Листинг 20.2. Использование слота

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Завершить работу")
button.clicked.connect(app.quit)
button.show()
sys.exit(app.exec())
```

Любой пользовательский метод можно сделать слотом, для чего необходимо перед его определением вставить декоратор @pyqtSlot(). Формат декоратора:

```
@QtCore.pyqtSlot([<Тип 1>, <Тип 2>,. . ., <Тип N>][, name=None][, result=None])
```

Можно указать типы параметров, принимаемых слотом (bool, int и др.). При задании типа данных C++ его имя необходимо указать в виде строки. Если слот не принимает параметров, типы не указываются. В параметре name можно передать имя слота в виде строки. Если этот параметр не задан, в качестве имени слота будет использовано имя метода. Параметр result задает тип результата, возвращаемого слотом. Если слот не возвращает результата, этот параметр не задается.

Можно создать несколько одноименных слотов, принимающих разные параметры (*перегруженный слот*). Пример приведен в листинге 20.3.

Листинг 20.3. Создание слотов

```
from PyQt6 import QtCore, QtWidgets
import sys

class MyClass(QtCore.QObject):
    def __init__(self):
        QtCore.QObject.__init__(self)
```

```
    @QtCore.pyqtSlot()
    def on_clicked(self):
        print("Кнопка нажата. Слот on_clicked()")
    @QtCore.pyqtSlot(bool, name="myslot")
    def on_clicked2(self, status):
        print("Кнопка нажата. Слот myslot(bool)", status)

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Нажми меня")
button.clicked.connect(obj.on_clicked)
button.clicked.connect(obj.myslot)
button.show()
sys.exit(app.exec())
```

Превращать метод, используемый в качестве обработчика сигнала, в слот необязательно. Однако вызов слота выполняется быстрее вызова обычного метода.

20.1.2. Передача данных в обработчик сигнала

Передать какие-либо произвольные данные в обработчик можно следующими способами:

- создать анонимную функцию и внутри ее выполнить вызов обработчика с параметрами. Вот пример передачи обработчику числа 10:

  ```
  self.button1.clicked.connect(lambda : self.on_clicked_button1(10))
  ```

  Если передаваемое обработчику значение вычисляется в процессе выполнения кода, переменную, хранящую это значение, следует указывать в анонимной функции как значение по умолчанию, иначе функции будет передана ссылка на это значение, а не оно само:

  ```
  y = 10
  self.button1.clicked.connect(lambda x=y: self.on_clicked_button1(x))
  ```

- передать ссылку на объект класса, внутри которого определен метод `__call__()`. Передаваемое значение указывается в качестве параметра конструктора этого класса. Пример:

  ```
  class MyClass():
      def __init__(self, x=0):
          self.x = x
      def __call__(self):
          print("x =", self.x)
  . . .
  self.button1.clicked.connect(MyClass(10))
  ```

- передать ссылку на обработчик и данные в функцию `partial()` из модуля `functools`. Формат функции:

  ```
  partial(<Функция>[, *<Позиционные параметры>][, **<Именованные параметры>])
  ```

  Пример передачи параметра в обработчик:

  ```
  from functools import partial
  self.button1.clicked.connect(partial(self.on_clicked_button1, 10))
  ```

Значения, передаваемые в составе сигнала, будут переданы в обработчик после остальных параметров. Назначим обработчик сигнала clicked, принимающего логический параметр, и дополнительно передадим число:

```
self.button1.clicked.connect(partial(self.on_clicked_button1, 10))
```

Обработчик будет иметь следующий вид:

```
def on_clicked_button1(self, x, status):
    print("Нажата кнопка button1", x, status)
```

Результат выполнения:

```
Нажата кнопка button1 10 False
```

20.2. Блокировка и удаление обработчиков сигналов

Для блокировки и удаления обработчиков предназначены следующие методы компонентов:

- blockSignals(<Состояние>) — временно блокирует прием сигналов, если параметр имеет значение True, и снимает блокировку, если параметр имеет значение False. Метод возвращает предыдущее состояние;
- signalsBlocked() — возвращает значение True, если блокировка сигналов установлена, и False — в противном случае;
- disconnect() — удаляет заданный обработчик. Форматы метода:

```
<Сигнал>.disconnect([<Обработчик>])
<Сигнал>[<Тип>].disconnect([<Обработчик>])
```

Если обработчик не указан, удаляются все обработчики. Параметр <Тип> указывается лишь в том случае, если существуют сигналы с одинаковыми именами, но принимающие разные параметры. Примеры:

```
button.clicked.disconnect()
button.clicked[bool].disconnect(on_clicked_button)
button.clicked["bool"].disconnect(on_clicked_button)
```

Создадим окно с четырьмя кнопками (листинг 19.4). У кнопки **Нажми меня** назначим обработчик сигнала clicked. Чтобы информировать о нажатии кнопки, выведем сообщение в консоль. У кнопок **Блокировать**, **Разблокировать** и **Удалить обработчик** создадим обработчики, которые будут изменять состояние обработчика у кнопки **Нажми меня**.

Листинг 20.4. Блокировка и удаление обработчика сигнала

```
from PyQt6 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Блокировка и удаление обработчика")
        self.reize(300, 150)
        self.button1 = QtWidgets.QPushButton("Нажми меня")
```

```
        self.button2 = QtWidgets.QPushButton("Блокировать")
        self.button3 = QtWidgets.QPushButton("Разблокировать")
        self.button4 = QtWidgets.QPushButton("Удалить обработчик")
        self.button3.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        vbox.addWidget(self.button3)
        vbox.addWidget(self.button4)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
        self.button3.clicked.connect(self.on_clicked_button3)
        self.button4.clicked.connect(self.on_clicked_button4)
    @QtCore.pyqtSlot()
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
    @QtCore.pyqtSlot()
    def on_clicked_button2(self):
        self.button1.blockSignals(True)
        self.button2.setEnabled(False)
        self.button3.setEnabled(True)
    @QtCore.pyqtSlot()
    def on_clicked_button3(self):
        self.button1.blockSignals(False)
        self.button2.setEnabled(True)
        self.button3.setEnabled(False)
    @QtCore.pyqtSlot()
    def on_clicked_button4(self):
        self.button1.clicked.disconnect(self.on_clicked_button1)
        self.button2.setEnabled(False)
        self.button3.setEnabled(False)
        self.button4.setEnabled(False)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Если нажать кнопку **Нажми меня**, в консоли будет выведена строка `Нажата кнопка button1`. Нажатие кнопки **Блокировать** производит блокировку обработчика — теперь при нажатии кнопки **Нажми меня** никаких сообщений в консоли не выводится. Отменить блокировку можно с помощью кнопки **Разблокировать**. Нажатие кнопки **Удалить обработчик** производит полное удаление обработчика — в этом случае, чтобы обрабатывать нажатие кнопки **Нажми меня**, необходимо заново назначить обработчик.

Также можно временно сделать компонент недоступным, воспользовавшись следующими методами:

- `setEnabled(<Состояние>)` — если в параметре указано значение `False`, компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `True`;
- `setDisabled(<Состояние>)` — если в параметре указано значение `True`, компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `False`.

Проверить, доступен компонент или нет, позволяет метод `isEnabled()`. Он возвращает значение `True`, если компонент доступен, и `False` — в противном случае.

20.3. Генерирование сигналов

Сгенерировать сигнал программно позволяет метод `emit()`. Форматы этого метода:

```
<Сигнал>.emit([<Данные>])
<Сигнал>[<Тип>].emit([<Данные>])
```

Метод `emit()` всегда вызывается у объекта, которому посылается сигнал, например:

```
button.clicked.emit()
```

Сигналу и соответственно его обработчику можно передать данные, указав их в вызове метода `emit()`:

```
button.clicked[bool].emit(False)
button.clicked["bool"].emit(False)
```

Некоторые компоненты предоставляют методы, которые генерируют сигналы. Например, у кнопок существует метод `click()`. Используя его, инструкцию:

```
button.clicked.emit()
```

можно записать следующим образом:

```
button.click()
```

20.4. Пользовательские сигналы

Также можно создавать свои собственные, *пользовательские* сигналы. Для этого следует определить в классе какой-либо атрибут класса (не обычный!) и присвоить ему результат, возвращенный функцией `pyqtSignal()` из модуля `QtCore`. Формат функции:

```
pyqtSignal([<Тип 1>, <Тип 2>, . . ., <Тип N>][, name=None])
```

Можно указать типы значений, передаваемых сигналу (например: `bool` или `int`):

```
mysignal1 = QtCore.pyqtSignal(int)
mysignal2 = QtCore.pyqtSignal(int, str)
```

При использовании типа данных C++ его имя необходимо указать в виде строки:

```
mysignal3 = QtCore.pyqtSignal("QDate")
```

Если сигнал не принимает параметров, типы не указываются.

Параметр `name` задает имя создаваемого сигнала. Если он не указан, имя сигнала совпадет с именем атрибута класса, в котором он хранится. Пример:

```
mysignal4 = QtCore.pyqtSignal(int, name="mySignal")
```

Можно создать несколько одноименных сигналов, принимающих разные параметры (*перегруженный сигнал*). В этом случае типы передаваемых значений указываются внутри квадратных скобок. Вот пример сигнала, передающего данные типа `int` или `str`:

```
mysignal5 = QtCore.pyqtSignal([int], [str])
```

Создадим окно с двумя кнопками (листинг 19.5), которым назначим обработчики сигнала `clicked` (нажатие кнопки). Внутри обработчика щелчка на первой кнопке сгенерируем два сигнала: первый будет имитировать нажатие второй кнопки, а второй станет пользовательским, привязанным к окну. Внутри обработчиков выведем сообщения в консоли.

Листинг 20.5. Пользовательский сигнал

```
from PyQt6 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    mysignal = QtCore.pyqtSignal(int, int)
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Пользовательский сигнал")
        self.resize(300, 100)
        self.button1 = QtWidgets.QPushButton("Нажми меня")
        self.button2 = QtWidgets.QPushButton("Кнопка 2")
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
        self.mysignal.connect(self.on_mysignal)
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
        # Генерируем сигналы
        self.button2.clicked[bool].emit(False)
        self.mysignal.emit(10, 20)
    def on_clicked_button2(self):
        print("Нажата кнопка button2")
    def on_mysignal(self, x, y):
        print("Обработан пользовательский сигнал mysignal()")
        print("x =", x, "y =", y)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Результат выполнения после нажатия первой кнопки:

```
Нажата кнопка button1
Нажата кнопка button2
```

```
Обработан пользовательский сигнал mysignal()
x = 10 y = 20
```

Вместо конкретного типа принимаемого сигналом параметра можно указать тип `QVariant` из модуля `QtCore`. В этом случае сигналу допускается передавать значение любого типа. Пример:

```
mysignal = QtCore.pyqtSignal(QtCore.QVariant)
. . .
self.mysignal.emit(20)
self.mysignal.emit("Привет!")
self.mysignal.emit([1, "2"])
```

20.5. Использование таймеров

Таймеры позволяют через заданный интервал времени выполнять специальный метод `timerEvent()`. Для назначения таймера используется метод `startTimer()` компонента. Формат метода:

```
startTimer(<Интервал>[, timerType=TimerType.CoarseTimer])
```

Интервал времени задается в миллисекундах. Минимальное значение интервала зависит от операционной системы. Если указать значение 0, таймер будет срабатывать регулярно при отсутствии других необработанных событий.

Необязательный параметр `timerType` задает тип таймера в виде одного из элементов перечисления `TimerType` из модуля `QtCore.Qt`:

- `PreciseTimer` — точный таймер, обеспечивающий точность до миллисекунд;
- `CoarseTimer` — «приблизительный» таймер, обеспечивающий точность в пределах 5% от заданного интервала;
- `VeryCoarseTimer` — «приблизительный» таймер, обеспечивающий точность до секунд.

Метод `startTimer()` возвращает идентификатор таймера, с помощью которого впоследствии можно остановить таймер.

Формат специального метода `timerEvent()`:

```
timerEvent(self, <Объект класса QTimerEvent>)
```

Внутри этого метода можно получить идентификатор таймера, вызвав метод `timerId()` у объекта класса `QTimerEvent`, который передается методу `timerEvent()` со вторым параметром.

Чтобы остановить таймер с указанным идентификатором, необходимо воспользоваться методом `killTimer(<Идентификатор>)` компонента.

Создадим в окне часы, которые будут отображать текущее системное время с точностью до секунды, и добавим возможность запуска и остановки этих часов с помощью соответствующих кнопок (листинг 20.6).

Листинг 20.6. Часы, вариант 1

```
from PyQt6 import QtCore, QtWidgets
import time
```

```
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Часы ")
        self.resize(200, 100)
        self.timer_id = 0
        self.label = QtWidgets.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignmentFlag.AlignHCenter)
        self.button1 = QtWidgets.QPushButton("Запустить")
        self.button2 = QtWidgets.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
    def on_clicked_button1(self):
        # Задаем интервал в 1 секунду и «приблизительный» таймер
        self.timer_id = self.startTimer(1000,
                                        timerType=QtCore.Qt.TimerType.VeryCoarseTimer)
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        if self.timer_id:
            self.killTimer(self.timer_id)
            self.timer_id = 0
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
    def timerEvent(self, event):
        self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Вместо методов `startTimer()` и `killTimer()` класса `QObject` можно воспользоваться классом `QTimer` из модуля `QtCore`. Конструктор класса имеет следующий формат:

```
QTimer([parent=None])
```

Методы этого класса:

- `setInterval(<Интервал>)` — задает интервал времени в миллисекундах, по истечении которого генерируется сигнал `timeout`. Минимальное значение интервала зависит от операционной системы. Если указать значение 0, таймер будет срабатывать регулярно при отсутствии других необработанных сигналов;

- `start([<Интервал>])` — запускает таймер. В параметре можно указать интервал времени в миллисекундах. Если параметр не указан, используется значение, заданное в вызове метода `setInterval()`;
- `stop()` — останавливает таймер;
- `isActive()` — возвращает значение `True`, если таймер запущен, и `False` — в противном случае;
- `timerId()` — возвращает идентификатор таймера, если он запущен, и значение `-1` — в противном случае;
- `remainingTime()` — возвращает время, оставшееся до очередного срабатывания таймера, в миллисекундах;
- `interval()` — возвращает установленный интервал;
- `setSingleShot(<Состояние>)` — если в параметре указано значение `True`, таймер сработает только один раз, в противном случае — будет срабатывать многократно;
- `isSingleShot()` — возвращает значение `True`, если таймер сработает только один раз, и `False` — в противном случае;
- `setTimerType(<Тип таймера>)` — задает тип таймера, который указывается в том же виде, что и в вызове метода `startTimer()`;
- `timerType()` — возвращает тип таймера.

Переделаем предыдущий пример, использовав класс `QTimer` (листинг 20.7).

Листинг 20.7. Часы, вариант 2

```
from PyQt6 import QtCore, QtWidgets
import time

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Часы")
        self.resize(200, 100)
        self.label = QtWidgets.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignmentFlag.AlignHCenter)
        self.button1 = QtWidgets.QPushButton("Запустить")
        self.button2 = QtWidgets.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.on_timeout);
    def on_clicked_button1(self):
        self.timer.start(1000) # 1 секунда
```

```
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        self.timer.stop()
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
    def on_timeout(self):
        self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Статический метод `singleShot()` класса `QTimer` запускает таймер, настраивает его для однократного срабатывания и указывает функцию или метод, который будет вызван по истечении заданного интервала. Формат метода:

```
singleShot(<Интервал>[, <Тип таймера>], <Функция или метод>)
```

Примеры:

```
QtCore.QTimer.singleShot(1000, self.on_timeout)
QtCore.QTimer.singleShot(1000, QtWidgets.QApplication.instance().quit)
```

20.6. Обработка всех событий

Обработка событий отличается от обработки сигналов. Чтобы обработать событие, необходимо создать подкласс какого-либо компонента и переопределить в нем соответствующий специальный метод (например, чтобы обработать нажатие клавиши, следует переопределить метод `keyPressEvent()`). Специальные методы принимают объект, содержащий детальную информацию о событии (обозначение нажатой кнопки мыши, код нажатой клавиши и т. п.).

Классы событий являются производными от класса `QEvent` и наследуют от него следующие методы:

- `accept()` — устанавливает флаг, разрешающий дальнейшую обработку события. Скажем, если в методе `closeEvent()` у объекта события вызвать метод `accept()`, окно будет закрыто. Этот флаг обычно установлен по умолчанию;
- `ignore()` — сбрасывает флаг, разрешающий дальнейшую обработку события. Так, если в методе `closeEvent()` у объекта события вызвать метод `ignore()`, окно закрыто не будет;
- `setAccepted(<Флаг>)` — если в качестве параметра указано значение `True`, флаг, разрешающий дальнейшую обработку события, будет установлен (аналогично вызову метода `accept()`), а если `False` — сброшен (аналогично вызову метода `ignore()`);
- `isAccepted()` — возвращает текущее состояние флага, разрешающего дальнейшую обработку события;
- `spontaneous()` — возвращает `True`, если событие сгенерировано операционной системой, и `False` — если самой программой;

- `type()` — возвращает тип события в виде одного из следующих элементов перечисления `Type` из класса `QEvent` (приведены основные типы событий, полный их список содержится на странице **https://doc.qt.io/qt-6/qevent.html#Type-enum**):
  - `None_` — нет события;
  - `MouseButtonPress` — нажата кнопка мыши;
  - `MouseButtonRelease` — отпущена кнопка мыши;
  - `MouseButtonDblClick` — двойной щелчок мышью;
  - `MouseMove` — перемещение мыши;
  - `Wheel` — прокручено колесико мыши;
  - `Enter` — курсор мыши помещен на компонент;
  - `Leave` — курсор мыши уведен с компонента;
  - `DragEnter` — курсор мыши помещен на компонент при операции перетаскивания;
  - `DragMove` — производится операция перетаскивания;
  - `DragLeave` — курсор мыши уведен с компонента при операции перетаскивания;
  - `Drop` — операция перетаскивания завершена;
  - `KeyPress` — нажата клавиша;
  - `KeyRelease` — отпущена клавиша;
  - `Show` — компонент отображен;
  - `Hide` — компонент скрыт;
  - `FocusIn` — компонент получил фокус ввода;
  - `FocusOut` — компонент потерял фокус ввода;
  - `Paint` — компонент требует перерисовки;
  - `Move` — местоположение компонента изменилось;
  - `Resize` — размеры компонента изменились;
  - `PolishRequest` — компонент должен быть настроен;
  - `Polish` — компонент настроен;
  - `ShowToParent` — дочерний компонент отображен;
  - `HideToParent` — дочерний компонент скрыт;
  - `ChildAdded` — дочерний компонент добавлен;
  - `ChildPolished` — производится настройка дочернего компонента;
  - `ChildRemoved` — дочерний компонент удален;
  - `Close` — окно закрыто;
  - `WindowActivate` — окно стало активным;
  - `WindowDeactivate` — окно стало неактивным;
  - `ActivationChange` — изменилось состояние активности окна верхнего уровня;
  - `WindowBlocked` — окно блокировано модальным окном;
  - `WindowUnblocked` — окно разблокировано после закрытия модального окна;

- `WindowStateChange` — состояние окна изменилось;
- `ApplicationStateChange` — текущее состояние программы изменилось;
- `ContextMenu` — событие контекстного меню;
- `Clipboard` — содержимое буфера обмена изменилось;
- `Timer` — событие таймера;
- `1000` — `User` — пользовательское событие;
- `65535` — `MaxUser` — максимальный идентификатор пользовательского события.

Статический метод `registerEventType([<Идентификатор>])` класса `QEvent` позволяет зарегистрировать пользовательский тип события с указанным идентификатором. В качестве последнего можно указать значение в пределах от `QEvent.User` (`1000`) до `QEvent.MaxUser` (`65535`). Если заданный идентификатор занят, выбирается ближайший свободный. В качестве результата возвращается идентификатор зарегистрированного события.

Перехват всех событий осуществляется с помощью специального метода `event(self, <event>)`. Через параметр `<event>` доступен объект события. Этот объект различен у разных типов событий (например, у события `MouseButtonPress` он относится к классу `QMouseEvent`, а у события `KeyPress` — к классу `QKeyEvent`). Эти классы будут рассмотрены позже.

Из метода `event()` следует вернуть в качестве результата значение `True`, если событие было обработано, и `False` — в противном случае. Если возвращается значение `True`, то родительский компонент не получит событие. Чтобы продолжить распространение события на родителя, необходимо вызвать метод `event()` базового класса и передать ему текущий объект события. Обычно это делается так:

```
return QtWidgets.QWidget.event(self, e)
```

В этом случае пользовательский класс является наследником класса `QWidget` и переопределяет метод `event()`. Если вы наследуете другой класс, следует вызывать метод именно этого класса. Например, при наследовании класса `QLabel` инструкция будет выглядеть так:

```
return QtWidgets.QLabel.event(self, e)
```

Пример обработки нажатий клавиш, щелчков мышью и закрытия окна показан в листинге 20.8.

Листинг 20.8. Перехват всех событий

```
from PyQt6 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def event(self, e):
        if e.type() == QtCore.QEvent.Type.KeyPress:
            print("Нажата клавиша на клавиатуре")
            print("Код:", e.key(), ", текст:", e.text())
        elif e.type() == QtCore.QEvent.Type.Close:
            print("Окно закрыто")
```

```
        elif e.type() == QtCore.QEvent.Type.MouseButtonPress:
            p = e.pos()
            print("Щелчок мышью. Координаты:", p.x(), p.y())
        return QtWidgets.QWidget.event(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Перехватывать все события следует только в самом крайнем случае. В обычных ситуациях надо обрабатывать только нужные события.

20.7. События окна

20.7.1. Изменение состояния окна

Отследить изменение состояния окна (сворачивание, разворачивание, скрытие и отображение) позволяют следующие методы:

- `changeEvent(self, <event>)` — вызывается при изменении состояния окна, компонента или программы, заголовка окна, его палитры, состояния активности окна верхнего уровня, языка, локали и др. (полный список смотрите в документации). При обработке события типа `Type.WindowStateChange` через параметр `<event>` доступен объект класса `QWindowStateChangeEvent`. Этот класс поддерживает только метод `oldState()`, с помощью которого можно получить предыдущее состояние окна;
- `showEvent(self, <event>)` — вызывается при отображении компонента. Через параметр `<event>` доступен объект класса `QShowEvent`;
- `hideEvent(self, <event>)` — вызывается при скрытии компонента. Через параметр `<event>` доступен объект класса `QHideEvent`.

Для примера выведем в консоль текущее состояние окна при его сворачивании, разворачивании, скрытии и отображении (листинг 20.9).

Листинг 20.9. Отслеживание состояния окна

```
from PyQt6 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def changeEvent(self, e):
        if e.type() == QtCore.QEvent.Type.WindowStateChange:
            if self.isMinimized():
                print("Окно свернуто")
            elif self.isMaximized():
                print("Окно раскрыто до максимальных размеров")
```

```
            elif self.isFullScreen():
                print("Полноэкранный режим")
            elif self.isActiveWindow():
                print("Окно находится в фокусе ввода")
        QtWidgets.QWidget.changeEvent(self, e) # Отправляем дальше
    def showEvent(self, e):
        print("Окно отображено")
        QtWidgets.QWidget.showEvent(self, e)   # Отправляем дальше
    def hideEvent(self, e):
        print("Окно скрыто")
        QtWidgets.QWidget.hideEvent(self, e)   # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

20.7.2. Изменение местоположения и размеров окна

При перемещении и изменении размеров окна вызываются следующие специальные методы:

- `moveEvent(self, <event>)` — непрерывно вызывается при перемещении окна. Через параметр `<event>` доступен объект класса `QMoveEvent`. Получить координаты левого верхнего угла окна позволяют следующие методы этого класса:
  - `pos()` — возвращает объект класса `QPoint` с текущими координатами;
  - `oldPos()` — возвращает объект класса `QPoint` с предыдущими координатами;
- `resizeEvent(self, <event>)` — непрерывно вызывается при изменении размеров окна. Через параметр `<event>` доступен объект класса `QResizeEvent`. Получить размеры окна позволяют следующие методы этого класса:
  - `size()` — возвращает объект класса `QSize` с текущими размерами;
  - `oldSize()` — возвращает объект класса `QSize` с предыдущими размерами.

Пример обработки изменения местоположения и размеров окна показан в листинге 20.10.

Листинг 20.10. Отслеживание смены местоположения и размеров окна

```
from PyQt6 import QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def moveEvent(self, e):
        print("x = {0}; y = {1}".format(e.pos().x(), e.pos().y()))
        QtWidgets.QWidget.moveEvent(self, e)   # Отправляем дальше
    def resizeEvent(self, e):
        print("w = {0}; h = {1}".format(e.size().width(), e.size().height()))
        QtWidgets.QWidget.resizeEvent(self, e) # Отправляем дальше
```

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

20.7.3. Перерисовка окна или его части

Когда компонент (или часть компонента) становится видимым, требуется выполнить его перерисовку. В этом случае вызывается специальный метод `paintEvent(self, <event>)`. Через параметр `<event>` доступен объект класса `QPaintEvent`, который поддерживает следующие методы:

- `rect()` — возвращает объект класса `QRect` с координатами и размерами прямоугольной области, которую требуется перерисовать;
- `region()` — возвращает объект класса `QRegion` с регионом, требующим перерисовки.

С помощью этих методов можно получить координаты области, которая, например, была ранее перекрыта другим окном и теперь вновь оказалась в зоне видимости. Перерисовывая только область, а не весь компонент, можно заметно повысить быстродействие программы. Следует также заметить, что в целях эффективности несколько событий перерисовки могут быть объединены в одно событие с общей областью перерисовки.

В некоторых случаях перерисовку окна необходимо выполнить вне зависимости от внешних действий системы или пользователя (например, при изменении каких-либо значений требуется обновить нарисованный на их основе график). Вызвать событие перерисовки компонента позволяют следующие методы класса `QWidget`:

- `repaint()` — немедленно вызывает метод `paintEvent()` для перерисовки компонента при условии, что таковой не скрыт и обновление не было запрещено вызовом метода `setUpdatesEnabled()`. Форматы метода:

  ```
  repaint()
  repaint(<X>, <Y>, <Ширина>, <Высота>)
  repaint(<Прямоугольная область QRect>)
  repaint(<Область QRegion>)
  ```

  Первый формат вызова выполняет перерисовку всего компонента, а остальные — только области с указанными координатами;

- `update()` — генерирует событие перерисовки при условии, что компонент не скрыт и обновление не запрещено. Событие будет обработано на следующей итерации цикла обработки событий. Если генерируется сразу несколько событий, они объединяются в одно, чтобы избежать неприятного мерцания. Рекомендуется использовать этот метод вместо метода `repaint()`. Форматы вызова такие же, как и у метода `repaint()`.

20.7.4. Предотвращение закрытия окна

При закрытии окна нажатием кнопки закрытия в его заголовке или вызовом метода `close()` вызывается специальный метод `closeEvent(self, <event>)`. Через параметр `<event>` доступен объект класса `QCloseEvent`. Чтобы предотвратить закрытие окна, у объекта события следует вызвать метод `ignore()`, в противном случае — метод `accept()`.

В качестве примера по нажатии кнопки закрытия выведем стандартное диалоговое окно с запросом подтверждения закрытия окна (листинг 20.11). Если пользователь нажмет кнопку **Да**, закроем окно, а если щелкнет кнопку **Нет** или просто закроет диалоговое окно, не будем его закрывать.

Листинг 20.11. Обработка закрытия окна

```
from PyQt6 import QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def closeEvent(self, e):
        result = QtWidgets.QMessageBox.question(self,
                    "Подтверждение закрытия окна",
                    "Вы действительно хотите закрыть окно?")
        if result == QtWidgets.QMessageBox.StandardButton.Yes:
            e.accept()
            QtWidgets.QWidget.closeEvent(self, e)
        else:
            e.ignore()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

20.8. События клавиатуры

20.8.1. Управление фокусом ввода

Для управления фокусом ввода предназначены следующие методы класса `QWidget`:

- `setFocus([<Причина>])` — устанавливает фокус ввода, если компонент находится в активном окне. В качестве причины изменения фокуса ввода можно указать один из следующих элементов перечисления `FocusReason` из модуля `QtCore.Qt`:
  - `MouseFocusReason` — фокус изменен с помощью мыши;
  - `TabFocusReason` — нажата клавиша <Tab>;
  - `BacktabFocusReason` — нажата комбинация клавиш <Shift>+<Tab>;
  - `ActiveWindowFocusReason` — окно стало активным или неактивным;
  - `PopupFocusReason` — открыто или закрыто всплывающее окно;
  - `ShortcutFocusReason` — нажата комбинация клавиш быстрого доступа;
  - `MenuBarFocusReason` — фокус ввода переместился на меню;
  - `OtherFocusReason` — другая причина;

- `clearFocus()` — убирает фокус ввода с текущего компонента;
- `hasFocus()` — возвращает значение `True`, если компонент имеет фокус ввода, и `False` — в противном случае;
- `focusWidget()` — возвращает ссылку на последний дочерний компонент, у которого вызывался метод `setFocus()`. Для компонентов верхнего уровня возвращается ссылка на компонент, который получит фокус после того, как окно станет активным;
- `setFocusProxy(<Компонент>)` — указывает компонент, который будет получать фокус ввода вместо текущего компонента;
- `focusProxy()` — возвращает ссылку на компонент, который получает фокус ввода вместо текущего компонента. Если такого компонента нет, возвращает значение `None`;
- `focusNextChild()` — передает фокус ввода следующему компоненту (аналогично нажатию клавиши <Tab>). Возвращает значение `True`, если фокус ввода удалось передать, и `False` — в противном случае;
- `focusPreviousChild()` — передает фокус ввода предыдущему компоненту (аналогично нажатию комбинации клавиш <Shift>+<Tab>). Возвращает значение `True`, если фокус ввода удалось передать, и `False` — в противном случае;
- `focusNextPrevChild(<Направление>)` — если в параметре указано значение `True`, работает аналогично методу `focusNextChild()`, если указано `False` — аналогично методу `focusPreviousChild()`;
- `setTabOrder(<Компонент 1>, <Компонент 2>)` — статический метод, задает последовательность смены фокуса при нажатии клавиши <Tab>. В параметре `<Компонент 2>` указывается ссылка на компонент, на который переместится фокус с компонента `<Компонент 1>`. Вот пример указания цепочки перехода `widget1 -> widget2 -> widget3 -> widget4`:

```
QtWidgets.QWidget.setTabOrder(widget1, widget2)
QtWidgets.QWidget.setTabOrder(widget2, widget3)
QtWidgets.QWidget.setTabOrder(widget3, widget4)
```

- `setFocusPolicy(<Способ>)` — задает способ получения фокуса текущим компонентом в виде одного из следующих элементов перечисления `FocusPolicy` из модуля `QtCore.Qt`:
  - `NoFocus` — компонент не может получать фокус;
  - `TabFocus` — с помощью клавиши <Tab>;
  - `ClickFocus` — посредством щелчка мышью;
  - `StrongFocus` — с помощью клавиши <Tab> и щелчка мышью;
  - `WheelFocus` — с помощью клавиши <Tab>, щелчка мышью и колесика мыши;
- `focusPolicy()` — возвращает текущий способ получения фокуса;
- `grabKeyboard()` — захватывает ввод с клавиатуры. Другие компоненты не будут получать события клавиатуры, пока не будет вызван метод `releaseKeyboard()`;
- `releaseKeyboard()` — освобождает захваченный ранее ввод с клавиатуры.

Получить ссылку на компонент, находящийся в фокусе ввода, позволяет статический метод `focusWidget()` класса `QApplication`. Если ни один компонент не имеет фокуса ввода, метод возвращает значение `None`. Не путайте этот метод с одноименным методом из класса `QWidget`.

Обработать получение и потерю фокуса ввода позволяют следующие специальные методы класса `QWidget`:

- `focusInEvent(self, <event>)` — вызывается при получении фокуса ввода;
- `focusOutEvent(self, <event>)` — вызывается при потере фокуса ввода.

Через параметр `<event>` доступен объект класса `QFocusEvent`, который поддерживает следующие методы:

- `gotFocus()` — возвращает значение `True`, если фокус ввода был получен, и `False` — если был потерян;
- `lostFocus()` — возвращает значение `True`, если фокус ввода был потерян, и `False` — если был получен;
- `reason()` — возвращает причину установки фокуса. Значение аналогично значению параметра `<Причина>` в методе `setFocus()`.

Создадим окно с кнопкой и двумя полями ввода (листинг 20.12). У полей ввода обработаем получение и потерю фокуса ввода, а по нажатии кнопки установим фокус ввода на второе поле. Кроме того, зададим последовательность перехода при нажатии клавиши <Tab>.

Листинг 20.12. Установка фокуса ввода

```
from PyQt6 import QtWidgets

class MyLineEdit(QtWidgets.QLineEdit):
    def __init__(self, id, parent=None):
        QtWidgets.QLineEdit.__init__(self, parent)
        self.id = id
    def focusInEvent(self, e):
        print("Получен фокус полем", self.id)
        QtWidgets.QLineEdit.focusInEvent(self, e)
    def focusOutEvent(self, e):
        print("Потерян фокус полем", self.id)
        QtWidgets.QLineEdit.focusOutEvent(self, e)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.button = QtWidgets.QPushButton("Установить фокус на поле 2")
        self.line1 = MyLineEdit(1)
        self.line2 = MyLineEdit(2)
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.button)
        self.vbox.addWidget(self.line1)
        self.vbox.addWidget(self.line2)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
        # Задаем порядок обхода с помощью клавиши <Tab>
        QtWidgets.QWidget.setTabOrder(self.line1, self.line2)
        QtWidgets.QWidget.setTabOrder(self.line2, self.button)
```

```
    def on_clicked(self):
        self.line2.setFocus()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

20.8.2. Назначение клавиш быстрого доступа

Клавиша быстрого доступа (или *горячая клавиша*) — это какая-либо из алфавитно-цифровых клавиш, нажатие которой совместно с клавишей-модификатором (обычно <Alt>) устанавливает фокус ввода на связанный с ней компонент. Если связанным компонентом является кнопка (или пункт меню), она будет нажата.

У компонента, имеющего текстовую надпись (например, кнопки), указать клавишу быстрого доступа можно, поставив в тексте надписи перед буквой, соответствующей указываемой клавише, символ `&`. Чтобы вставить в надпись сам символ `&`, его следует продублировать.

Если компонент не имеет собственной текстовой надписи, придется создать надпись к нему и связать ее с компонентом с помощью метода `setBuddy(<Компонент>)` объекта надписи.

Если же создать отдельную надпись у компонента не представляется возможным, можно воспользоваться следующими методами класса `QWidget`:

- `grabShortcut(<Клавиша>[, <Контекст>])` — регистрирует заданную клавишу быстрого доступа и возвращает идентификатор, с помощью которого можно управлять ею в дальнейшем. Клавиша указывается в виде объекта класса `QKeySequence` из модуля `QtGui`. Конструктор этого класса поддерживает следующие форматы вызова:

  ```
  QKeySequence(<Строка с обозначениями клавиш>)
  QKeySequence(<Числовое обозначение клавиши>)
  ```

  В первом формате в передаваемой строке обозначения клавиш указываются последовательно и отделяются друг от друга символами +. Во втором формате обозначение клавиши указывается в виде значения одного из элементов перечисления `Key` из модуля `QtCore.Qt` или комбинации значений через оператор +.

  Также можно воспользоваться статическим методом `mnemonic(<Обозначение клавиши>)` класса `QKeySequence`, где обозначение клавиши указывается в виде строки и предваряется символом `&`.

  Примеры создания объекта класса `QKeySequence` для комбинации клавиш <Alt>+<E>:

  ```
  QtGui.QKeySequence.mnemonic("&e")
  QtGui.QKeySequence("Alt+e")
  QtGui.QKeySequence(QtCore.Qt.Key.Key_F5)
  ```

  В качестве контекста можно указать следующие элементы перечисления `ShortcutContext` из модуля `QtCore.Qt`:

  - `WidgetShortcut` — клавиша быстрого доступа работает, когда родитель компонента имеет фокус ввода;
  - `WidgetWithChildrenShortcut` — клавиша быстрого доступа работает, когда его родитель компонента или любой из его потомков имеет фокус ввода;

- `WindowShortcut` — клавиша быстрого доступа работает, когда окно, содержащее компонент, активно (значение по умолчанию);
- `ApplicationShortcut` — клавиша быстрого доступа работает, когда программа активна;

- `releaseShortcut(<Идентификатор>)` — удаляет комбинацию с указанным идентификатором;
- `setShortcutEnabled(<Идентификатор>[, <Состояние>])` — если в качестве состояния указано `True` (значение по умолчанию), клавиша быстрого доступа с заданным идентификатором разрешена. Состояние `False` запрещает использование клавиши быстрого доступа.

При нажатии клавиши быстрого доступа генерируется событие типа `Type.Shortcut`, которое можно обработать в методе `event(self, <event>)`. Через параметр `<event>` доступен объект класса `QShortcutEvent`, поддерживающий следующие методы:

- `shortcutId()` — возвращает идентификатор комбинации клавиш;
- `isAmbiguous()` — возвращает значение `True`, если нажатая клавиша быстрого доступа была указана сразу у нескольких компонентов, и `False` — в противном случае;
- `key()` — возвращает объект класса `QKeySequence`, представляющий нажатую клавишу быстрого доступа.

Создадим окно с надписью, двумя полями ввода и кнопкой (листинг 20.13). У первого поля ввода назначим клавишу быстрого доступа <Alt>+<B> через надпись, а у второго поля — клавишу <Alt>+<E> с помощью метода `grabShortcut()`. У кнопки назначим клавишу быстрого доступа <Alt>+<У> через надпись на кнопке.

Листинг 20.13. Назначение клавиш быстрого доступа разными способами

```
from PyQt6 import QtCore, QtGui, QtWidgets

class MyLineEdit(QtWidgets.QLineEdit):
    def __init__(self, parent=None):
        QtWidgets.QLineEdit.__init__(self, parent)
        self.id = None
    def event(self, e):
        if e.type() == QtCore.QEvent.Type.Shortcut:
            if self.id == e.shortcutId():
                self.setFocus(QtCore.Qt.FocusReason.ShortcutFocusReason)
                return True
        return QtWidgets.QLineEdit.event(self, e)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.label = QtWidgets.QLabel("Устано&вить фокус на поле 1")
        self.lineEdit1 = QtWidgets.QLineEdit()
        self.label.setBuddy(self.lineEdit1)
        self.lineEdit2 = MyLineEdit()
        self.lineEdit2.id = self.lineEdit2.grabShortcut(
                                        QtGui.QKeySequence.mnemonic("&e"))
```

```
        self.button = QtWidgets.QPushButton("&Убрать фокус с поля 1")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.lineEditl)
        self.vbox.addWidget(self.lineEdit2)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
    def on_clicked(self):
        self.lineEditl.clearFocus()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
```

Для назначения клавиш быстрого доступа также можно воспользоваться классом `QShortcut` из модуля `QtGui`. В этом случае назначение клавиши у второго текстового поля будет выглядеть так:

```
self.lineEdit2 = QtWidgets.QLineEdit()
self.shc = QtGui.QShortcut(QtGui.QKeySequence.mnemonic("&e"), self)
self.shc.setContext(QtCore.Qt.ShortcutContext.WindowShortcut)
self.shc.activated.connect(self.lineEdit2.setFocus)
```

Еще можно использовать класс `QAction` из модуля `QtGui`. Назначение клавиши у второго текстового поля выглядит следующим образом:

```
self.lineEdit2 = QtWidgets.QLineEdit()
self.act = QtGui.QAction(self)
self.act.setShortcut(QtGui.QKeySequence.mnemonic("&e"))
self.act.triggered.connect(self.lineEdit2.setFocus)
self.addAction(self.act)
```

20.8.3. Нажатие и отпускание клавиш

При нажатии и отпускании клавиши вызываются следующие специальные методы:

- `keyPressEvent(self, <event>)` — вызывается при нажатии клавиши. Если клавишу удерживать нажатой, метод будет вызываться многократно, пока клавишу не отпустят;
- `keyReleaseEvent(self, <event>)` — вызывается при отпускании нажатой ранее клавиши.

Через параметр `<event>` доступен объект класса `QKeyEvent`, хранящий дополнительную информацию о событии. Он поддерживает следующие полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-6/qkeyevent.html**):

- `key()` — возвращает код нажатой клавиши:

```
if e.key() == QtCore.Qt.Key.Key_B:
    print("Нажата клавиша <B>")
```

- `text()` — возвращает текстовое представление введенного символа в кодировке Unicode или пустую строку, если была нажата специальная клавиша;
- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты. Возвращает значение одного из следующих элементов перечисления `KeyboardModifier` из модуля `QtCore.Qt` или их комбинацию:
  - `NoModifier` — модификаторы не были нажаты;
  - `ShiftModifier` — была нажата клавиша <Shift>;
  - `ControlModifier` — была нажата клавиша <Ctrl>;
  - `AltModifier` — была нажата клавиша <Alt>;
  - `MetaModifier` — была нажата клавиша <Meta>;
  - `KeypadModifier` — была нажата любая клавиша на дополнительной клавиатуре;
  - `GroupSwitchModifier` — была нажата клавиша <Mode_switch> (только в X11).

  Пример определения, была ли нажата клавиша-модификатор <Shift>:

```
if e.modifiers() & QtCore.Qt.KeyboardModifier.ShiftModifier:
    print("Нажата клавиша-модификатор <Shift>")
```

- `isAutoRepeat()` — возвращает `True`, если событие было вызвано удержанием клавиши нажатой, и `False` — в противном случае;
- `matches(<Обозначение>)` — возвращает значение `True`, если была нажата специальная комбинация клавиш, соответствующая указанному обозначению, и `False` — в противном случае. В качестве параметра указывается один из элементов перечисления `StandardKey` из класса `QKeySequence` — например, `Copy` соответствует комбинации клавиш <Ctrl>+<C> (полный список элементов перечисления приведен на странице **https://doc.qt.io/qt-6/qkeysequence.html#StandardKey-enum**). Пример:

```
if e.matches(QtGui.QKeySequence.StandardKey.Copy):
    print("Нажата комбинация клавиш <Ctrl>+<C>")
```

При обработке нажатия клавиш следует учитывать, что:

- компонент должен иметь возможность принимать фокус ввода. Некоторые компоненты (в частности, надписи) по умолчанию не могут принимать фокус ввода. Чтобы изменить способ получения фокуса, следует воспользоваться методом `setFocusPolicy(<Способ>)` (рассмотрен в *разд. 20.8.1*);
- чтобы захватить эксклюзивный ввод с клавиатуры, следует воспользоваться методом `grabKeyboard()`, а чтобы освободить ввод — методом `releaseKeyboard()`;
- можно перехватить нажатие любых клавиш, кроме клавиши <Tab> и комбинации <Shift>+<Tab>. Эти клавиши используются для передачи фокуса следующему и предыдущему компоненту соответственно. Перехватить нажатие этих клавиш можно только в методе `event(self, <event>)`;
- если событие обработано, следует вызвать метод `accept()` объекта события. Чтобы родительский компонент смог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

20.9. События мыши

20.9.1. Нажатие и отпускание кнопок мыши

При нажатии и отпускании кнопок мыши вызываются следующие специальные методы:

- `mousePressEvent(self, <event>)` — вызывается при нажатии кнопки мыши в области компонента;
- `mouseReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой кнопки мыши;
- `mouseDoubleClickEvent(self, <event>)` — вызывается при двойном щелчке мышью. Следует учитывать, что двойному щелчку предшествуют другие события. Последовательность событий при двойном щелчке выглядит так:

```
MouseButtonPress
MouseButtonRelease
MouseButtonDblClick
MouseButtonPress
MouseButtonRelease
```

  Задать интервал двойного щелчка в миллисекундах позволяет статический метод `setDoubleClickInterval(<Интервал>)` класса `QApplication`, а получить его текущее значение — статический метод `doubleClickInterval()` того же класса.

Через параметр `<event>` доступен объект класса `QMouseEvent`, хранящий дополнительную информацию о событии. Он поддерживает такие методы:

- `pos()` — возвращает объект класса `QPoint` с целочисленными координатами в пределах компонента;
- `position()` — возвращает объект класса `QPointF` с вещественными координатами в пределах компонента;
- `scenePosition()` — возвращает объект класса `QPointF` с вещественными координатами в пределах окна или графической сцены (рассмотрена в *главе 26*);
- `globalPosition()` — возвращает объект класса `QPointF` с вещественными координатами в пределах экрана;
- `button()` — возвращает обозначение нажатой кнопки мыши в виде одного из следующих элементов перечисления `MouseButton` из модуля `QtCore.Qt` (описаны не все элементы, полный их список приведен на странице **https://doc.qt.io/qt-6/qt.html#MouseButton-enum**):
  - `NoButton` — ни одна кнопка не нажата. Это значение возвращается методом `button()` при перемещении курсора мыши;
  - `LeftButton` — левая кнопка;
  - `RightButton` — правая кнопка;
  - `MiddleButton` — средняя кнопка или колесико;
  - `XButton1`, `ExtraButton1` и `BackButton` — первая из дополнительных кнопок;
  - `XButton2`, `ExtraButton2` и `ForwardButton` — вторая из дополнительных кнопок;
- `buttons()` — возвращает обозначение всех нажатых кнопок мыши в виде комбинации элементов перечисления `MouseButton`:

```
if e.buttons() & QtCore.Qt.MouseButton.LeftButton:
    print("Нажата левая кнопка мыши")
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Описан в *разд. 20.8.3*;
- `flags()` — возвращает дополнительные сведения о событии в виде значения одного из элементов перечисления `MouseEventFlag` из модуля `QtCore` или их комбинацию. По состоянию на сегодня перечисление содержит единственный элемент `MouseEventCreatedDoubleClick`, который устанавливается при возникновении события `MouseButtonPress`, если оно было вызвано двойным щелчком;
- `timestamp()` — возвращает в виде числа отметку системного времени, в которое возникло событие.

Если событие было успешно обработано, следует вызвать метод `accept()` объекта события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` нужно вызвать метод `ignore()`.

Если у компонента элемент `WA_NoMousePropagation` перечисления `WidgetAttribute` из модуля `QtCore.Qt` установлен в `True`, событие мыши не будет передаваться родительскому компоненту. Значение атрибута можно изменить с помощью метода `setAttribute()`, вызванного у этого компонента:

```
button.setAttribute(QtCore.Qt.WidgetAttribute.WA_NoMousePropagation, True)
```

По умолчанию событие мыши перехватывает компонент, над которым был произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне компонента, следует захватить мышь вызовом метода `grabMouse()`. Освободить захваченную ранее мышь позволяет метод `releaseMouse()`.

20.9.2. Перемещение курсора мыши

Чтобы обработать перемещение курсора мыши над компонентом, необходимо определить специальный метод `mouseMoveEvent(self, <event>)`. Через параметр `<event>` доступен объект класса `QMouseEvent`, содержащий дополнительную информацию о событии (описан в *разд. 20.9.1*). Следует учитывать, что метод `button()` при перемещении мыши возвращает элемент `NoButton` перечисления `QtCore.Qt.MouseButton`.

По умолчанию метод `mouseMoveEvent()` вызывается только в том случае, если при перемещении удерживается нажатой какая-либо кнопка мыши. Это сделано специально, чтобы не создавать лишних событий при обычном перемещении мыши. Если необходимо обрабатывать любые перемещения мыши в пределах компонента, следует вызвать у этого компонента метод `setMouseTracking(<Состояние>)`, которому передать значение `True`. Чтобы обработать все перемещения внутри окна, нужно дополнительно захватить мышь вызовом метода `grabMouse()`.

Метод `pos()` объекта события возвращает позицию точки в системе координат текущего компонента. Чтобы преобразовать координаты точки в систему координат родительского компонента или в глобальную систему координат, надо воспользоваться следующими методами класса `QWidget`:

- `mapToGlobal(<Координаты QPoint>)` — преобразует заданные координаты точки из системы координат компонента в глобальную систему координат. Возвращает объект класса `QPoint`;

- `mapFromGlobal(<Координаты QPoint>)` — преобразует заданные координаты точки из глобальной в систему координат компонента. Возвращает объект класса `QPoint`;
- `mapToParent(<Координаты QPoint>)` — преобразует заданные координаты точки из системы координат компонента в систему координат его родителя. Если компонент не имеет родителя, действует как метод `mapToGlobal()`. Возвращает объект класса `QPoint`;
- `mapFromParent(<Координаты QPoint>)` — преобразует заданные координаты точки из системы координат родительского компонента в систему координат текущего компонента. Если компонент не имеет родителя, работает подобно методу `mapFromGlobal()`. Возвращает объект класса `QPoint`;
- `mapTo(<Компонент>, <Координаты QPoint>)` — преобразует заданные координаты точки из системы координат текущего компонента в систему координат родителя указанного компонента. Возвращает объект класса `QPoint`;
- `mapFrom(<Компонент>, <Координаты QPoint>)` — преобразует заданные координаты точки из системы координат родителя указанного компонента в систему координат текущего компонента. Возвращает объект класса `QPoint`.

20.9.3. Наведение и увод курсора мыши

Обработать наведение курсора мыши на компонент и увод его с компонента позволяют следующие специальные методы:

- `enterEvent(self, <event>)` — вызывается при наведении курсора мыши на компонент. Через параметр `<event>` доступен объект класса `QEnterEvent`, не несущий никакой дополнительной информации;
- `leaveEvent(self, <event>)` — вызывается при уводе курсора мыши с компонента. Через параметр `<event>` доступен объект класса `QLeaveEvent`, не несущий никакой дополнительной информации.

20.9.4. Прокрутка колесика мыши

Обработать прокрутку колесика мыши позволяет специальный метод `wheelEvent(self, <event>)`. Через параметр `<event>` доступен объект класса `QWheelEvent` с дополнительной информацией о событии.

Класс `QWheelEvent` поддерживает методы:

- `angleDelta()` — возвращает углы поворота колесика по горизонтали и вертикали в градусах, умноженные на 8, в виде объекта класса `QPoint`. Значения углов могут быть положительными или отрицательными — в зависимости от направления поворота. Пример:

```
angle = e.angleDelta() / 8
angleX = angle.x()
angleY = angle.y()
```

- `pixelDelta()` — возвращает величины поворота колесика по горизонтали и вертикали в пикселах в виде объекта класса `QPoint`. Величины поворота могут быть положительными или отрицательными — в зависимости от направления поворота.

Также поддерживаются методы `position()`, `scenePosition()`, `globalPosition()`, `button()`, `buttons()`, `modifiers()` и `timestamp()`, описанные в *разд. 20.9.1*.

Если событие было успешно обработано, необходимо вызвать метод accept() объекта события. Чтобы родительский компонент мог получить событие, вместо метода accept() необходимо вызвать метод ignore().

20.9.5. Изменение курсора мыши

Для изменения внешнего вида курсора мыши при вхождении его в область компонента предназначены следующие методы класса QWidget:

- setCursor(<Курсор>) — задает внешний вид курсора мыши для компонента. В качестве параметра указывается объект класса QCursor или следующие элементы перечисления CursorShape из модуля QtCore.Qt: ArrowCursor (стандартная стрелка), UpArrowCursor (стрелка, направленная вверх), CrossCursor (крестообразный курсор), WaitCursor (песочные часы), IBeamCursor (I-образный курсор), SizeVerCursor (стрелки, направленные вверх и вниз), SizeHorCursor (стрелки, направленные влево и вправо), SizeBDiagCursor (стрелки, направленные в правый верхний угол и левый нижний угол), SizeFDiagCursor (стрелки, направленные в левый верхний угол и правый нижний угол), SizeAllCursor (стрелки, направленные вверх, вниз, влево и вправо), BlankCursor (невидимый курсор), SplitVCursor (указатель изменения высоты), SplitHCursor (указатель изменения ширины), PointingHandCursor (курсор в виде руки), ForbiddenCursor (перечеркнутый круг), OpenHandCursor (разжатая рука), ClosedHandCursor (сжатая рука), WhatsThisCursor (стрелка с вопросительным знаком), BusyCursor (стрелка с песочными часами), DragMoveCursor (обозначение перемещения путем перетаскивания), DragCopyCursor (обозначение копирования путем перетаскивания) и DragLinkCursor (обозначение создания ссылки путем перетаскивания). Пример:

```
self.setCursor(QtCore.Qt.CursorShape.WaitCursor)
```

- unsetCursor() — отменяет изменение курсора мыши для компонента. В результате внешний вид курсора будет наследоваться от родителя;
- cursor() — возвращает объект класса QCursor, представляющий текущий курсор.

Управлять видом курсора сразу на уровне программы можно с помощью следующих статических методов класса QApplication:

- setOverrideCursor(<Курсор>) — задает курсор мыши. В качестве параметра указывается объект класса QCursor или один из ранее описанных элементов перечисления CursorShape из модуля QtCore.Qt;
- restoreOverrideCursor() — отменяет изменение курсора мыши:

```
QtWidgets.QApplication.setOverrideCursor(QtCore.Qt.CursorShape.WaitCursor)
# Выполняем длительную операцию
QtWidgets.QApplication.restoreOverrideCursor()
```

- changeOverrideCursor(<Курсор>) — изменяет курсор мыши. Если до вызова этого метода *не* вызывался метод setOverrideCursor(), ничего не произойдет. В качестве параметра указывается объект класса QCursor или один из ранее описанных элементов перечисления CursorShape из модуля QtCore.Qt;
- overrideCursor() — возвращает объект класса QCursor, представляющий текущий курсор, или значение None, если таковой не был изменен.

Изменять курсор мыши на уровне программы следует на небольшой промежуток времени — обычно при выполнения какой-либо операции, в процессе которой программа не мо-

жет нормально реагировать на действия пользователя. Обычно курсор изменяют на песочные часы (элемент `WaitCursor`) или стрелку с песочными часами (элемент `BusyCursor`).

Метод `setOverrideCursor()` может быть вызван несколько раз. В этом случае курсоры помещаются в стек. Каждый вызов метода `restoreOverrideCursor()` удаляет последний курсор, добавленный в стек. Для нормальной работы программы необходимо вызывать методы `setOverrideCursor()` и `restoreOverrideCursor()` одинаковое количество раз.

Класс `QCursor` позволяет использовать в качестве курсора изображение любой формы. Чтобы загрузить изображение, следует передать конструктору класса `QPixmap` путь к файлу изображения. Для создания объекта курсора необходимо передать конструктору класса `QCursor` в первом параметре объект класса `QPixmap`, а во втором и третьем параметрах — координаты «горячей» точки будущего курсора. Вот пример создания и установки пользовательского курсора:

```
self.setCursor(QtGui.QCursor(QtGui.QPixmap("cursor.png"), 0, 0))
```

Класс `QCursor` поддерживает два статических метода:

- `pos()` — возвращает объект класса `QPoint` с координатами курсора мыши относительно экрана:

```
p = QtGui.QCursor.pos()
print(p.x(), p.y())
```

- `setPos()` — задает координаты курсора мыши. Форматы вызова:

```
setPos(<X>, <Y>)
setPos(<Координаты QPoint>)
```

20.10. Операция перетаскивания (drag & drop)

Операция перетаскивания состоит из двух частей: запуск процесса и обработка перетаскивания и сброса данных. Обе части могут обрабатываться как одной, так и разными программами.

20.10.1. Запуск перетаскивания

Запуск перетаскивания осуществляется следующим образом:

1. Внутри специального метода `mousePressEvent()` запоминаются координаты курсора мыши в момент нажатия ее левой кнопки.
2. Внутри специального метода `mouseMoveEvent()` вычисляется пройденное расстояние или измеряется продолжительность операции. Это необходимо для того, чтобы предотвратить случайное перетаскивание. Управлять задержкой позволяют следующие статические методы класса `QApplication`:
   - `setStartDragDistance(<Расстояние>)` — задает минимальное расстояние, после прохождения которого будет запущена операция перетаскивания, в пикселах (по умолчанию 10);
   - `startDragDistance()` — возвращает это расстояние;
   - `setStartDragTime(<Время>)` — задает время задержки в миллисекундах перед запуском операции перетаскивания;
   - `startDragTime()` — возвращает это время.

3. Если пройдено минимальное расстояние или истек минимальный промежуток времени, создается объект класса `QDrag` из модуля `QtGui` и у него вызывается метод `exec()`, который после завершения операции возвращает действие, выполненное с данными (например, их копирование или перемещение).

Конструктор класса `QDrag` вызывается так:

```
QDrag(<Компонент-источник, из которого перетаскиваются данные>)
```

Класс `QDrag` поддерживает следующие методы:

- `exec()` — запускает процесс перетаскивания и возвращает действие, которое было выполнено по завершении операции. Метод имеет два формата:

  ```
  exec([<Допустимые действия>=DropAction.MoveAction])
  exec(<Допустимые действия>, <Действие по умолчанию>)
  ```

  Допустимые действия над перетаскиваемыми данными задаются следующими элементами перечисления `DropAction` из модуля `QtCore.Qt` или их комбинацией через оператор `|`: `CopyAction` (копирование перетаскиваемых данных), `MoveAction` (перемещение), `LinkAction` (создание ссылки), `IgnoreAction` (действие игнорировано), `TargetMoveAction` (сохранить перетаскиваемые данные в компоненте-источнике). Действие по умолчанию, выполняемое, если ни одна из клавиш-модификаторов не была нажата, задается в виде одного из приведенных ранее элементов перечисления `DropAction`. Пример:

  ```
  drag = QtGui.QDrag(self)
  act = drag.exec(QtCore.Qt.DropAction.MoveAction |
                  QtCore.Qt.DropAction.CopyAction,
                  QtCore.Qt.DropAction.MoveAction)
  ```

- `setMimeData(<Данные>)` — задает сами перемещаемые данные в виде объекта класса `QMimeData` (описан в *разд. 20.10.2*). Пример задания текста:

  ```
  data = QtCore.QMimeData()
  data.setText("Перетаскиваемый текст")
  drag = QtGui.QDrag(self)
  drag.setMimeData(data)
  ```

- `mimeData()` — возвращает объект класса `QMimeData` с перемещаемыми данными;

- `setPixmap(<Изображение QPixmap>)` — задает изображение, которое будет перемещаться вместе с курсором мыши:

  ```
  drag.setPixmap(QtGui.QPixmap("dd_representer.png"))
  ```

- `pixmap()` — возвращает объект класса `QPixmap` с изображением, которое перемещается вместе с курсором;

- `setHotSpot(<Координаты QPoint>)` — задает координаты «горячей» точки на перемещаемом изображении:

  ```
  drag.setHotSpot(QtCore.QPoint(20, 20))
  ```

- `hotSpot()` — возвращает объект класса `QPoint` с координатами «горячей» точки на перемещаемом изображении;

- `setDragCursor(<Изображение QPixmap>, <Действие>)` — задает курсор мыши для указанного действия. Первым параметром передается изображение, которое, собственно, станет курсором мыши. Если в первом параметре указан пустой объект класса `QPixmap`,

ранее установленный для действия курсор будет отменен. Действие задается в виде одного из приведенных ранее элементов перечисления DropAction. Пример изменения курсора мыши для перемещения:

```
drag.setDragCursor(QtGui.QPixmap("move_cursor.png"),
                   QtCore.Qt.DropAction.MoveAction)
```

- dragCursor(<Действие>) — возвращает объект класса QPixmap, представляющий курсор мыши для заданного действия;
- source() — возвращает ссылку на компонент-источник;
- target() — возвращает ссылку на компонент-приемник или значение None, если компонент находится в другой программе;
- supportedActions() — возвращает значение, представляющее набор допустимых в текущей операции действий, в виде комбинации перечисленных ранее элементов перечисления DropAction;
- defaultAction() — возвращает действие по умолчанию в виде одного из приведенных ранее элементов перечисления DropAction.

Класс QDrag поддерживает два сигнала:

- actionChanged(<Действие>) — генерируется при изменении действия. Новое действие представляется одним из приведенных ранее элементов перечисления DropAction;
- targetChanged(<Компонент-приемник>) — генерируется при изменении компонента-приемника.

Примеры назначения обработчиков этих сигналов:

```
drag.actionChanged.connect(self.on_action_changed)
drag.targetChanged.connect(self.on_target_changed)
```

20.10.1.1. Задание перетаскиваемых данных

Перетаскиваемые данные, передаваемые методу setMimeData() класса QDrag, задаются объектом класса QMimeData из модуля QtCore. Его конструктор вызывается так:

```
QMimeData()
```

Класс QMimeData поддерживает следующие полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-6/qmimedata.html**):

- setText(<Текст>) — заносит текстовые данные (MIME-тип text/plain):

  ```
  data.setText("Перетаскиваемый текст")
  ```

- text() — возвращает текстовые данные;
- hasText() — возвращает значение True, если объект содержит текстовые данные, и False — в противном случае;
- setHtml(<HTML-код>) — заносит данные в формате HTML (MIME-тип text/html):

  ```
  data.setHtml("<b>Перетаскиваемый HTML-текст</b>")
  ```

- html() — возвращает данные в формате HTML;
- hasHtml() — возвращает значение True, если объект содержит данные в формате HTML, и False — в противном случае;

- `setUrls(<Список интернет-адресов>)` — заносит список интернет-адресов (MIME-тип `text/uri-list`), представленных объектами класса `QUrl` из модуля `QtCore`. С помощью этого MIME-типа можно обработать перетаскивание файлов. Пример:

```
data.setUrls([QtCore.QUrl("https://www.google.ru/")])
```

- `urls()` — возвращает список интернет-адресов:

```
uri = e.mimeData().urls()[0].toString()
```

- `hasUrls()` — возвращает значение `True`, если объект содержит список интернет-адресов, и `False` — в противном случае;
- `setImageData(<Изображение>)` — заносит изображение (MIME-тип `application/x-qt-image`). В параметре можно указать, например, объект класса `QImage` или `QPixmap`. Пример:

```
data.setImageData(QtGui.QImage("pixmap.png"))
data.setImageData(QtGui.QPixmap("pixmap.png"))
```

- `imageData()` — возвращает объект изображения;
- `hasImage()` — возвращает значение `True`, если объект содержит изображение, и `False` — в противном случае;
- `setColorData(<Цвет>)` — заносит цвет (MIME-тип `application/x-color`). Цвет может быть представлен значением произвольного типа. Пример:

```
data.setColorData("red")
```

- `colorData()` — возвращает цвет;
- `hasColor()` — возвращает значение `True`, если объект содержит цвет, и `False` — в противном случае;
- `setData(<MIME-тип>, <Данные>)` — заносит данные указанного MIME-типа. MIME-тип указывается в виде строки, данные — в виде объекта класса `QByteArray` из модуля `QtCore`. Метод можно вызвать несколько раз с различными MIME-типами. Пример занесения текстовых данных:

```
data.setData("text/plain", QtCore.QByteArray(bytes("Данные", "utf-8")))
```

- `data(<MIME-тип>)` — возвращает объект класса `QByteArray` с данными, соответствующими указанному MIME-типу;
- `hasFormat(<MIME-тип>)` — возвращает значение `True`, если объект содержит данные указанного MIME-типа, и `False` — в противном случае;
- `formats()` — возвращает список с MIME-типами данных, содержащихся в объекте;
- `removeFormat(<MIME-тип>)` — удаляет данные, соответствующие указанному MIME-типу;
- `clear()` — удаляет все данные.

Если необходимо перетаскивать данные какого-либо специфического типа, нужно наследовать класс `QMimeData` и переопределить в нем методы `retrieveData()` и `formats()`. За подробной информацией по этому вопросу обращайтесь к документации.

20.10.2. Обработка перетаскивания и сброса

Прежде чем обрабатывать перетаскивание и сбрасывание данных в какой-либо компонент, необходимо сообщить системе, что этот компонент может принимать перетаскивае-

мые данные. Для этого внутри конструктора компонента следует вызвать метод `setAcceptDrops(<Состояние>)`, передав ему `True`, например:

```
self.setAcceptDrops(True)
```

Обработка перетаскивания и сброса объекта выполняется следующим образом:

1. Внутри специального метода `dragEnterEvent()` компонента проверяется MIME-тип перетаскиваемых данных и предлагаемое действие. Если компонент способен обработать сброс этих данных и соглашается с предложенным действием, необходимо вызвать метод `acceptProposedAction()` объекта события. Если нужно изменить действие, методу `setDropAction()` объекта события передается новое действие, а затем у того же объекта вызывается метод `accept()` вместо `acceptProposedAction()`.
2. Если необходимо ограничить область сброса некоторым участком компонента, следует дополнительно определить в нем метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри компонента. При достижении курсором мыши нужного участка компонента следует вызвать метод `accept()` объекта события, передав ему объект класса `QRect` с координатами и размерами этого участка. В этом случае при перетаскивании внутри участка метод `dragMoveEvent()` повторно вызываться не будет.
3. Внутри метода `dropEvent()` компонента производится обработка сброса.

Обработать события, возникающие в процессе перетаскивания и сбрасывания, позволяют следующие специальные методы класса `QWidget`:

- `dragEnterEvent(self, <event>)` — вызывается, когда перетаскиваемые данные входят в область компонента. Через параметр `<event>` доступен объект класса `QDragEnterEvent`;
- `dragLeaveEvent(self, <event>)` — вызывается, когда перетаскиваемые данные покидают область компонента. Через параметр `<event>` доступен объект класса `QDragLeaveEvent`;
- `dragMoveEvent(self, <event>)` — вызывается при перетаскивании данных внутри компонента. Через параметр `<event>` доступен объект класса `QDragMoveEvent`;
- `dropEvent(self, <event>)` — вызывается при сбрасывании данных в компонент. Через параметр `<event>` доступен объект класса `QDropEvent`.

Класс `QDragLeaveEvent` наследует класс `QEvent` и не несет никакой дополнительной информации.

Цепочка наследования остальных классов выглядит так:

```
QEvent — QDropEvent — QDragMoveEvent — QDragEnterEvent
```

Класс `QDragEnterEvent` не содержит собственных методов.

Класс `QDropEvent` поддерживает следующие методы:

- `mimeData()` — возвращает объект класса `QMimeData` с перемещаемыми данными;
- `possibleActions()` — возвращает комбинацию допустимых действий при сбрасывании:

```
if e.possibleActions() & QtCore.Qt.DropAction.MoveAction:
    print("MoveAction")
if e.possibleActions() & QtCore.Qt.DropAction.CopyAction:
    print("CopyAction")
```

- `proposedAction()` — возвращает предлагаемое действие по умолчанию;
- `acceptProposedAction()` — сообщает о готовности принять переносимые данные и согласии с предлагаемым действием по умолчанию. Этот метод (или метод `accept()`, под-

держиваемый классом QDragMoveEvent) необходимо вызвать внутри метода dragEnterEvent(), иначе метод dropEvent() вызван не будет;

- setDropAction(<Действие>) — задает другое действие при сбрасывании. После изменения действия следует вызвать метод accept() объекта события, а не acceptProposedAction();
- dropAction() — возвращает действие, которое должно быть выполнено при сбрасывании. Оно может не совпадать со значением, возвращаемым методом proposedAction(), если действие было изменено с помощью метода setDropAction();
- source() — возвращает ссылку на компонент-источник, или None, если данные переносятся из другой программы.

Теперь рассмотрим методы класса QDragMoveEvent:

- accept([<Область QRect>]) — сообщает о согласии с дальнейшей обработкой события. В качестве параметра можно указать объект класса QRect с координатами и размерами прямоугольной области, в которой будет доступно сбрасывание;
- ignore([<Область QRect>]) — отменяет операцию переноса данных. В качестве параметра можно указать объект класса QRect с координатами и размерами прямоугольной области, в которой сбрасывание запрещено;
- answerRect() — возвращает объект класса QRect с координатами и размерами прямоугольной области, в которой произойдет сбрасывание, если событие будет принято.

Некоторые компоненты в PyQt уже поддерживают технологию drag & drop (так, в поле ввода можно перетащить текст из другой программы).

20.11. Работа с буфером обмена

Получить ссылку на глобальный объект буфера обмена позволяет статический метод clipboard() класса QApplication:

```
clipboard = QtWidgets.QApplication.clipboard()
```

Класс QClipboard, на основе которого создан этот объект, поддерживает следующие методы:

- setText(<Текст>) — заносит текст в буфер обмена:

```
clipboard.setText("Текст")
```

- text() — возвращает из буфера обмена текст или пустую строку;
- text(<Тип>) — возвращает кортеж из двух строк: первая хранит текст из буфера обмена, вторая — название типа. В параметре <Тип> могут быть указаны значения "plain" (простой текст), "html" (HTML-код) или пустая строка (любой тип);
- setImage(<Изображение QImage>) — заносит в буфер обмена изображение:

```
clipboard.setImage(QtGui.QImage("image.jpg"))
```

- image() — возвращает из буфера обмена изображение, представленное объектом класса QImage, или пустой объект этого класса;
- setPixmap(<Изображение QPixmap>) — заносит в буфер обмена изображение:

```
clipboard.setPixmap(QtGui.QPixmap("image.jpg"))
```

- `pixmap()` — возвращает из буфера обмена изображение, представленное объектом класса `QPixmap`, или пустой объект этого класса;
- `setMimeData(<Данные QMimeData>)` — заносит в буфер обмена заданные данные;
- `mimeData()` — возвращает данные, представленные объектом класса `QMimeData`;
- `clear()` — очищает буфер обмена.

Отследить изменение состояния буфера обмена позволяет сигнал `dataChanged()` объекта буфера обмена. Назначить обработчик этого сигнала можно так:

```
QtWidgets.QApplication.instance().clipboard().dataChanged.connect(
                                                    on_change_clipboard)
```

20.12. Фильтрация событий

События можно перехватывать еще до того, как они будут переданы компоненту. Для этого необходимо создать класс, который является наследником класса `QObject`, и определить в нем специальный метод `eventFilter(self, <Компонент>, <event>)`. Через второй параметр доступна ссылка на компонент, а через параметр `<event>` — на объект с дополнительной информацией о событии. Этот объект различен для разных типов событий — так, у события `MouseButtonPress` он будет создан на основе класса `QMouseEvent`, а у события `KeyPress` — на основе класса `QKeyEvent`. Из метода `eventFilter()` следует вернуть значение `True`, если событие не должно быть передано дальше, и `False` — в противном случае. Пример такого класса-фильтра, перехватывающего нажатие клавиши <B>:

```
class MyFilter(QtCore.QObject):
    def __init__(self, parent=None):
        QtCore.QObject.__init__(self, parent)
    def eventFilter(self, obj, e):
        if e.type() == QtCore.QEvent.Type.KeyPress:
            if e.key() == QtCore.Qt.Key.Key_B:
                print("Событие от клавиши <B> не дойдет до компонента")
                return True
        return QtCore.QObject.eventFilter(self, obj, e)
```

Далее следует создать объект этого класса, передав в конструктор ссылку на компонент, а затем вызвать у того же компонента метод `installEventFilter()`, передав в качестве единственного параметра ссылку на объект фильтра. Пример установки фильтра для надписи:

```
self.label.installEventFilter(MyFilter(self.label))
```

Метод `installEventFilter()` можно вызвать несколько раз, передавая ссылку на разные объекты фильтров. В этом случае первым будет вызван фильтр, который был добавлен последним. Кроме того, один фильтр можно установить сразу в нескольких компонентах. Ссылка на компонент, который является источником события, доступна через второй параметр метода `eventFilter()`.

Удалить заданный фильтр позволяет метод `removeEventFilter(<Фильтр>)`, вызываемый у компонента, у которого был назначен этот фильтр. Если таковой не был установлен, при вызове метода ничего не произойдет.

20.13. Генерирование событий

Для генерирования событий применяются следующие статические методы класса QApplication:

- sendEvent(<Компонент>, <Событие>) — немедленно посылает заданное событие указанному компоненту и возвращает результат выполнения обработчика;
- postEvent() — добавляет заданное событие в очередь. Формат вызова:

  ```
  postEvent(<Компонент>, <Событие>[,
                          priority=EventPriority.NormalEventPriority])
  ```

  Параметром priority можно передать приоритет события, использовав один из следующих элементов перечисления EventPriority из модуля QtCore.Qt: HighEventPriority (высокий приоритет), NormalEventPriority (обычный приоритет) и LowEventPriority (низкий приоритет). Этот метод является потокобезопасным, следовательно, его можно использовать в многопоточных программах для обмена событиями между потоками.

Пример отправки события QEvent.MouseButtonPress компоненту label:

```
e = QtGui.QMouseEvent(QtCore.QEvent.Type.MouseButtonPress,
                      QtCore.QPointF(5, 5), QtCore.Qt.MouseButton.LeftButton,
                      QtCore.Qt.MouseButton.LeftButton,
                      QtCore.Qt.KeyboardModifier.NoModifier)
QtWidgets.QApplication.sendEvent(self.label, e)
```

20.14. Пользовательские события

Пользовательским называется событие, созданное самим программистом. Класс пользовательского события должен быть производным от класса QEvent. В нем следует зарегистрировать пользовательское событие с помощью статического метода registerEventType() класса QEvent и сохранить идентификатор зарегистрированного события, возвращенный этим методом, в каком-либо атрибуте класса. Пример:

```
class MyEvent(QtCore.QEvent):
    idType = QtCore.QEvent.registerEventType()
    def __init__(self, data):
        QtCore.QEvent.__init__(self, MyEvent.idType)
        self.data = data
    def get_data(self):
        return self.data
```

Пример отправки события класса MyEvent компоненту label:

```
QtWidgets.QApplication.sendEvent(self.label, MyEvent("512"))
```

Обработать пользовательское событие можно с помощью специальных методов event(self, <event>) или customEvent(self, <event>). С параметром <event> в эти методы передается объект события. Пример:

```
def customEvent(self, e):
    if e.type() == MyEvent.idType:
        self.setText("Получены данные: {0}".format(e.get_data()))
```

ГЛАВА 21

Размещение компонентов в окнах. Контейнеры

Размещать компоненты в окнах можно двумя способами:

- посредством *абсолютного позиционирования* — явно указывая местоположение и размеры каждого компонента;
- используя *контейнеры* — особые компоненты, предназначенные для размещения других компонентов в определенном порядке (например, в стопку или в ячейках воображаемой сетки).

Все описанные далее классы определены в модуле `QtWidgets`.

21.1. Абсолютное позиционирование

При создании компонента с указанием ссылки на родителя созданный компонент будет выведен в позицию с координатами `(0, 0)` — в левый верхний угол родителя. Если создать таким образом несколько компонентов, то все они отобразятся в одной и той же позиции, наложившись друг на друга. Размеры добавляемых компонентов будут соответствовать их содержимому.

Для перемещения компонента можно воспользоваться методом `move()`, а для изменения размеров — методом `resize()`. Выполнить одновременное изменение позиции и размеров позволяет метод `setGeometry()`.

Для примера выведем внутри окна надпись и кнопку, применив абсолютное позиционирование (листинг 21.1).

Листинг 21.1. Абсолютное позиционирование

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Абсолютное позиционирование")
window.resize(300, 120)
label = QtWidgets.QLabel("Текст надписи", window)
button = QtWidgets.QPushButton("Текст на кнопке", window)
label.setGeometry(10, 10, 280, 60)
button.resize(280, 30)
```

```
button.move(10, 80)
window.show()
sys.exit(app.exec())
```

Абсолютное позиционирование имеет следующие недостатки:

- при изменении размеров окна координаты и размеры располагающихся в нем компонентов придется изменять самостоятельно;
- при запуске программы на другом компьютере, имеющем другие настройки экрана, системных шрифтов или другую операционную систему, надписи на компонентах могут выходить за их пределы.

  Аналогичная проблема возникнет, если программа поддерживает несколько языков интерфейса (поскольку длина слов в разных языках различается).

21.2. Контейнеры-стопки

Контейнер-*стопка* выстраивает находящиеся в нем компоненты в линию. Существуют два следующих класса, представляющих подобного рода контейнеры:

- `QHBoxLayout` — горизонтальная стопка;
- `QVBoxLayout` — вертикальная стопка.

Иерархия наследования для этих классов выглядит так:

```
(QObject, QLayoutItem) — QLayout — QBoxLayout — QHBoxLayout
(QObject, QLayoutItem) — QLayout — QBoxLayout — QVBoxLayout
```

Контейнеры-стопки не являются наследниками класса `QWidget`, вследствие чего не обладают собственным окном и не могут использоваться отдельно. Поэтому они обязательно должны быть привязаны к родительскому компоненту (обычно окну или другому контейнеру). При привязке контейнера к родителю контейнер автоматически выводится на экран.

Формат вызова конструктора обоих классов:

```
QHBoxLayout|QVBoxLayout([<Родитель>])
```

Если родитель не был указан при создании объекта контейнера, привязать контейнер к родителю можно вызовом у последнего метода `setLayout(<Привязываемый контейнер>)`.

Пример использования класса `QHBoxLayout` показан в листинге 21.2, а увидеть результат выполнения этого кода можно на рис. 21.1.

Листинг 21.2. Использование контейнера `QHBoxLayout`

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()              # Родительский компонент — окно
window.setWindowTitle("QHBoxLayout")
window.resize(300, 60)
button1 = QtWidgets.QPushButton("1")
button2 = QtWidgets.QPushButton("2")
button3 = QtWidgets.QPushButton("3")
```

```
button4 = QtWidgets.QPushButton("4")
hbox = QtWidgets.QHBoxLayout()              # Создаем контейнер
hbox.addWidget(button1)                     # Добавляем компоненты
hbox.addWidget(button2)
hbox.addWidget(button3)
hbox.addWidget(button4)
window.setLayout(hbox)                      # Привязываем контейнер к родителю
window.show()
sys.exit(app.exec())
```

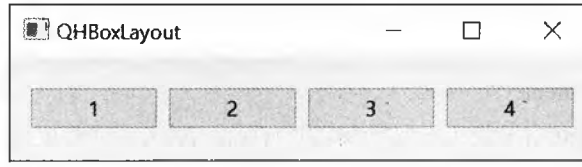


Рис. 21.1. Контейнер `QHBoxLayout` с четырьмя кнопками

Добавить компоненты в контейнер, удалить их и заменить другими позволяют следующие методы:

- `addWidget()` — добавляет компонент в конец контейнера. Формат метода:

  ```
  addWidget(<Компонент>[, stretch=0][, alignment=0])
  ```

  В первом параметре указывается ссылка на компонент. Необязательный параметр `stretch` задает фактор растяжения для ячейки в виде целого числа, а параметр `alignment` — выравнивание компонента внутри ячейки. Два последних параметра можно задавать как позиционные или именованные, например:

  ```
  hbox.addWidget(button1, 10, QtCore.Qt.AlignmentFlag.AlignRight)
  hbox.addWidget(button2, stretch=10)
  hbox.addWidget(button3, alignment=QtCore.Qt.AlignmentFlag.AlignRight)
  ```

- `insertWidget()` — добавляет компонент в позицию контейнера, обозначенную указанным индексом. Формат метода:

  ```
  insertWidget(<Индекс>, <Компонент>[, stretch=0][, alignment=0])
  ```

  Если в первом параметре указано значение 0, компонент будет добавлен в начало контейнера, а если отрицательное значение — в конец контейнера. Иное значение указывает определенную позицию. Остальные параметры аналогичны параметрам метода `addWidget()`. Пример:

  ```
  hbox.addWidget(button1)
  hbox.insertWidget(-1, button2) # Добавление в конец
  hbox.insertWidget(0, button3)  # Добавление в начало
  ```

- `removeWidget(<Компонент>)` — удаляет указанный компонент из контейнера;

- `replaceWidget()` — заменяет присутствующий в контейнере компонент другим. Формат метода:

  ```
  replaceWidget(<Заменяемый компонент>, <Заменяющий компонент>[,
                options=FindChildOption.FindChildrenRecursively])
  ```

В необязательном параметре `options` можно задать режим поиска заменяемого компонента в виде одного из элементов перечисления `FindChildOptions` из модуля `QtCore.Qt`: `FindDirectChildrenOnly` (искать только среди содержимого текущего контейнера) и `FindChildrenRecursively` (искать среди содержимого текущего и всех вложенных в него контейнеров);

- `addLayout(<Контейнер>[, stretch=0])` — добавляет другой контейнер в конец текущего контейнера. С помощью этого метода можно вкладывать один контейнер в другой, создавая таким образом структуру любой сложности;
- `insertLayout(Позиция>, <Контейнер>[, stretch=0])` — добавляет другой контейнер в позицию текущего контейнера, обозначенную указанным индексом. Если в первом параметре задано отрицательное значение, контейнер добавляется в конец;
- `addSpacing(<Размер>)` — добавляет пустое пространство указанного размера (в пикселах) в конец контейнера:

```
hbox.addSpacing(100)
```

- `insertSpacing(<Индекс>, <Размер>)` — добавляет пустое пространство указанного размера (в пикселах) в позицию с заданным индексом. Если первым параметром передается отрицательное значение, то пространство добавляется в конец;
- `addStretch([stretch=0])` — добавляет пустое растягиваемое пространство с нулевым минимальным размером и фактором растяжения `stretch` в конец контейнера. Это пространство можно сравнить с пружиной, вставленной между компонентами, а параметр `stretch` — с жесткостью пружины;
- `insertStretch(<Индекс>[, stretch=0])` — добавляет растягиваемое пространство в позицию с указанным индексом. Если в первом параметре задано отрицательное значение, пространство добавляется в конец контейнера.

Параметр `alignment` в методах `addWidget()` и `insertWidget()` задает выравнивание компонента внутри ячейки в виде одного из следующих элементов перечисления `AlignmentFlag` из модуля `QtCore.Qt`:

- `AlignLeft` или `AlignLeading` — горизонтальное выравнивание по левому краю;
- `AlignRight` или `AlignTrailing` — горизонтальное выравнивание по правому краю;
- `AlignHCenter` — горизонтальное выравнивание по центру;
- `AlignJustify` — заполнение всей ячейки;
- `AlignTop` — вертикальное выравнивание по верхнему краю;
- `AlignBottom` — вертикальное выравнивание по нижнему краю;
- `AlignVCenter` — вертикальное выравнивание по центру;
- `AlignBaseline` — вертикальное выравнивание по базовой линии;
- `AlignCenter` — горизонтальное и вертикальное выравнивание по центру;
- `AlignAbsolute` — если в методе `setLayoutDirection(<Выравнивание>)` класса `QWidget` указан элемент `RightToLeft` перечисления `LayoutDirection` из модуля `QtCore.Qt`, атрибут `AlignLeft` задает выравнивание по правому краю, а атрибут `AlignRight` — по левому краю. Чтобы атрибут `AlignLeft` всегда соответствовал именно левому краю, необходимо указать комбинацию `AlignAbsolute | AlignLeft`. Аналогично следует поступить с атрибутом `AlignRight`.

Можно задавать комбинации элементов, один из которых укажет выравнивание по горизонтали, другой — по вертикали. Например, комбинация `AlignLeft | AlignTop` задает выравнивание по левому и верхнему краям. Противоречивые значения приводят к непредсказуемым результатам.

Контейнеры поддерживают также следующие методы (здесь приведены только основные — полный их список ищите в документации):

- `setDirection(<Направление>)` — задает направление вывода компонентов. В параметре можно указать следующие элементы перечисления `Direction` из класса `QHBoxLayout` или `QVBoxLayout`:
  - `LeftToRight` — слева направо (значение по умолчанию у горизонтального контейнера);
  - `RightToLeft` — справа налево;
  - `TopToBottom` — сверху вниз (значение по умолчанию у вертикального контейнера);
  - `BottomToTop` — снизу вверх;
- `setContentsMargins()` — задает величины отступов от границ контейнера до его содержимого. Форматы метода:

```
setContentsMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)
setContentsMargins(<Отступы QMargins>)
```

  Примеры:

```
hbox.setContentsMargins(2, 4, 2, 4)
m = QtCore.QMargins(4, 2, 4, 2)
hbox.setContentsMargins(m)
```

- `setSpacing(<Расстояние>)` — задает расстояние между компонентами.

21.3. Контейнер-сетка

Контейнер-*сетка* располагает добавленные в него компоненты в ячейках воображаемой сетки. Он реализуется классом `QGridLayout`. Иерархия его наследования:

```
(QObject, QLayoutItem) — QLayout — QGridLayout
```

Формат конструктора класса `QGridLayout`:

```
QGridLayout([<Родитель>])
```

Если родитель не был указан при создании объекта контейнера, привязать контейнер к родителю можно вызовом у последнего метода `setLayout()`.

Пример использования класса `QGridLayout` представлен в листинге 21.3, а результат его выполнения — на рис. 21.2.

Листинг 21.3. Использование контейнера `QGridLayout`

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()                    # Родительский компонент — окно
window.setWindowTitle("QGridLayout")
```

```
window.resize(250, 100)
button1 = QtWidgets.QPushButton("1")
button2 = QtWidgets.QPushButton("2")
button3 = QtWidgets.QPushButton("3")
button4 = QtWidgets.QPushButton("4")
grid = QtWidgets.QGridLayout()              # Создаем сетку
grid.addWidget(button1, 0, 0)               # Добавляем компоненты
grid.addWidget(button2, 0, 1)
grid.addWidget(button3, 1, 0)
grid.addWidget(button4, 1, 1)
window.setLayout(grid)                      # Привязываем контейнер к родителю
window.show()
sys.exit(app.exec())
```

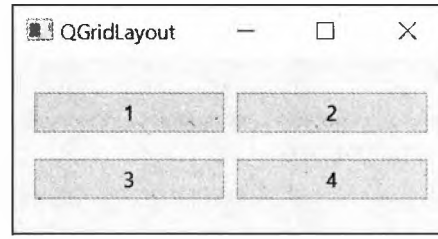


Рис. 21.2. Контейнер `QGridLayout` с четырьмя кнопками

Добавить компоненты в сетку позволяют следующие методы:

- `addWidget()` — добавляет компонент в указанную ячейку сетки. Метод имеет следующие форматы:

```
addWidget(<Компонент>, <Строка>, <Столбец>[, alignment=0])
addWidget(<Компонент>, <Строка>, <Столбец>, <Количество строк>,
          <Количество столбцов>[, alignment=0])
```

  В первом параметре указывается ссылка на компонент, во втором параметре — индекс строки, а в третьем — индекс столбца. Нумерация строк и столбцов начинается с нуля. Параметр `<Количество строк>` задает количество занимаемых компонентом ячеек по вертикали, а параметр `<Количество столбцов>` — по горизонтали. Параметр `alignment` задает выравнивание компонента внутри ячейки в виде одного из элементов перечисления `AlignmentFlag` из модуля `QtCore.Qt` (см. *разд. 21.2*). Пример:

```
grid = QtGui.QGridLayout()
grid.addWidget(button1, 0, 0, alignment=QtCore.Qt.AlignmentFlag.AlignLeft)
grid.addWidget(button2, 0, 1, QtCore.Qt.AlignmentFlag.AlignRight)
grid.addWidget(button3, 1, 0, 1, 2)
```

- `addLayout()` — добавляет заданный контейнер в указанную ячейку сетки. Метод имеет следующие форматы:

```
addLayout(<Контейнер>, <Строка>, <Столбец>[, alignment=0])
addLayout(<Контейнер>, <Строка>, <Столбец>, <Количество строк>,
          <Количество столбцов>[, alignment=0])
```

В первом параметре указывается ссылка на контейнер. Остальные параметры аналогичны параметрам метода `addWidget()`.

Для удаления и замены компонентов следует пользоваться методами `removeWidget()` и `replaceWidget()`, описанными в *разд. 21.2*.

Класс `QGridLayout` поддерживает следующие методы (здесь приведены только основные методы — полный их список смотрите на странице **https://doc.qt.io/qt-6/qgridlayout.html**):

- `setRowMinimumHeight(<Индекс>, <Высота>)` — задает минимальную высоту строки с индексом `<Индекс>`;
- `setColumnMinimumWidth(<Индекс>, <Ширина>)` — задает минимальную ширину столбца с индексом `<Индекс>`;
- `setRowStretch(<Индекс>, <Фактор растяжения>)` — задает фактор растяжения по вертикали для строки с индексом `<Индекс>`;
- `setColumnStretch(<Индекс>, <Фактор растяжения>)` — задает фактор растяжения по горизонтали для столбца с индексом `<Индекс>`;
- `setContentsMargins()` — задает величины отступов от границ сетки до ее содержимого. Форматы метода:

  ```
  setContentsMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)
  setContentsMargins(<Отступы QMargins>)
  ```

- `setSpacing(<Расстояние>)` — задает расстояние между компонентами по горизонтали и вертикали;
- `setHorizontalSpacing(<Расстояние>)` — задает расстояние между компонентами по горизонтали;
- `setVerticalSpacing(<Расстояние>)` — задает расстояние между компонентами по вертикали;
- `rowCount()` — возвращает количество строк сетки;
- `columnCount()` — возвращает количество столбцов сетки;
- `cellRect(<Индекс строки>, <Индекс колонки>)` — возвращает объект класса `QRect`, который хранит координаты и размеры ячейки, расположенной на пересечении строки и колонки с указанными индексами.

21.4. Контейнер-форма

Контейнер-*форма* выстраивает компоненты в виде таблицы из двух столбцов: в левом столбце располагаются надписи для компонентов, в правом — сами компоненты. Он реализуется классом `QFormLayout`. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QFormLayout
```

Формат конструктора класса `QFormLayout`:

```
QFormLayout([<Родитель>])
```

Если родитель не был указан при создании объекта контейнера, привязать контейнер к родителю можно вызовом у последнего метода `setLayout()`.

В листинге 21.4 показан код, создающий форму с применением контейнера `QFormLayout`. Результат выполнения этого кода можно увидеть на рис. 21.3.

Листинг 21.4. Использование контейнера QFormLayout

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QFormLayout")
window.resize(300, 150)
lineEdit = QtWidgets.QLineEdit()
textEdit = QtWidgets.QTextEdit()
button1 = QtWidgets.QPushButton("О&тправить")
button2 = QtWidgets.QPushButton("О&чистить")
hbox = QtWidgets.QHBoxLayout()
hbox.addWidget(button1)
hbox.addWidget(button2)
form = QtWidgets.QFormLayout()
form.addRow("&Название:", lineEdit)
form.addRow("&Описание:", textEdit)
form.addRow(hbox)
window.setLayout(form)
window.show()
sys.exit(app.exec())
```

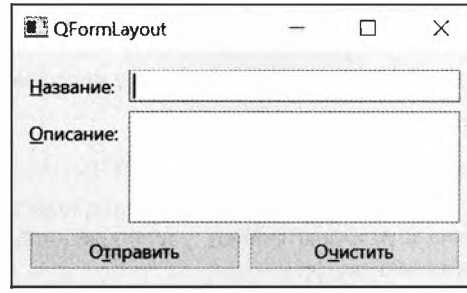


Рис. 21.3. Контейнер `QFormLayout`

Класс `QFormLayout` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qformlayout.html**):

- `addRow()` — добавляет в конец контейнера строку с указанными надписью и компонентом или другим контейнером. Форматы метода:

  ```
  addRow(<Текст надписи>, <Компонент> | <Контейнер>)
  addRow(<Надпись QLabel>, <Компонент> | <Контейнер>)
  addRow(<Компонент> | <Контейнер>)
  ```

  В первом формате в тексте надписи можно указать символ `&`, который пометит клавишу быстрого доступа для компонента (контейнера). Второй формат принимает в качестве надписи компонент класса `QLabel` — в этом случае связь с компонентом (контейнером) необходимо устанавливать вручную, передав ссылку на него в метод `setBuddy()`. Третий формат помещает компонент (контейнер) сразу в оба столбца формы;

- `insertRow()` — добавляет строку в позицию контейнера, обозначенную указанным индексом. Если в первом параметре задано отрицательное значение, компонент добавляется в конец контейнера. Форматы метода:

```
insertRow(<Индекс>, <Текст надписи>, <Компонент> | <Контейнер>)
insertRow(<Индекс>, <Надпись QLabel>, <Компонент> | <Контейнер>)
insertRow(<Индекс>, <Компонент> | <Контейнер>)
```

- `setFormAlignment(<Режим>)` — задает режим выравнивания формы (допустимые значения рассмотрены в *разд. 21.2*):

```
form.setFormAlignment(QtCore.Qt.AlignmentFlag.AlignRight |
                      QtCore.Qt.AlignmentFlag.AlignBottom)
```

- `setLabelAlignment(<Режим>)` — задает режим выравнивания надписей (допустимые значения рассмотрены в *разд. 21.2*):

```
form.setLabelAlignment(QtCore.Qt.AlignmentFlag.AlignRight)
```

- `setRowWrapPolicy(<Режим>)` — задает местоположение надписей относительно связанных с ними компонентов. В параметре указываются следующие элементы перечисления `RowWrapPolicy` из класса `QFormLayout`:
  - `DontWrapRows` — надписи расположены слева от компонентов;
  - `WrapLongRows` — длинные надписи могут находиться выше компонентов, а короткие надписи — слева от компонентов;
  - `WrapAllRows` — надписи всегда расположены выше компонентов;
- `setFieldGrowthPolicy(<Режим>)` — задает режим управления размерами компонентов. В параметре указываются следующие элементы перечисления `FieldGrowthPolicy` из класса `QFormLayout`:
  - `FieldsStayAtSizeHint` — компоненты всегда будут принимать рекомендуемые (возвращаемые методом `sizeHint()`) размеры;
  - `ExpandingFieldsGrow` — компоненты, у которых установлена политика изменения размеров `Expanding` или `MinimumExpanding` (см. *разд. 21.6*), займут всю доступную ширину. Размеры остальных компонентов всегда будут соответствовать рекомендуемым;
  - `AllNonFixedFieldsGrow` — все компоненты по возможности займут всю доступную ширину;
- `setContentsMargins()` — задает величины отступов от границ формы до ее содержимого. Форматы метода:

```
setContentsMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)
setContentsMargins(<Отступы QMargins>)
```

- `setSpacing(<Расстояние>)` — задает расстояние между компонентами по горизонтали и вертикали;
- `setHorizontalSpacing(<Расстояние>)` — задает расстояние между компонентами по горизонтали;
- `setVerticalSpacing(<Расстояние>)` — задает расстояние между компонентами по вертикали.

Для удаления и замены компонентов следует пользоваться методами `removeWidget()` и `replaceWidget()`, описанными в *разд. 21.2*.

21.5. Стеки

Контейнер-*стек* в каждый момент времени выводит лишь один компонент-потомок. Он реализуется классом `QStackedLayout`. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QStackedLayout
```

Формат конструктора класса `QStackedLayout`:

```
QStackedLayout([<Родитель>])
```

Если родитель не был указан при создании объекта контейнера, привязать контейнер к родителю можно вызовом у последнего метода `setLayout()`.

Класс `QStackedLayout` поддерживает следующие методы:

- `addWidget(<Компонент>)` — добавляет компонент в конец контейнера. Метод возвращает индекс добавленного компонента;
- `insertWidget(<Индекс>, <Компонент>)` — добавляет компонент в позицию контейнера, обозначенную указанным индексом. Метод возвращает индекс добавленного компонента;
- `setCurrentIndex(<Индекс>)` — делает видимым компонент с указанным индексом. Метод является слотом;
- `currentIndex()` — возвращает индекс видимого компонента;
- `setCurrentWidget(<Компонент>)` — делает видимым указанный компонент. Метод является слотом;
- `currentWidget()` — возвращает ссылку на видимый компонент;
- `setStackingMode(<Режим>)` — задает режим отображения компонентов. В параметре могут быть указаны следующие элементы перечисления `StackingMode` из класса `QStackedLayout`:
  - `StackOne` — видим только один компонент (значение по умолчанию);
  - `StackAll` — видны все компоненты;
- `stackingMode()` — возвращает режим отображения компонентов;
- `count()` — возвращает количество компонентов внутри контейнера;
- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None` при обращении по несуществующему индексу.

Для удаления и замены компонентов следует пользоваться методами `removeWidget()` и `replaceWidget()`, описанными в *разд. 21.2*.

Класс `QStackedLayout` поддерживает следующие сигналы:

- `currentChanged(<Индекс>)` — генерируется при выводе другого компонента. Через параметр внутри обработчика доступен целочисленный индекс выведенного компонента;
- `widgetRemoved(<Индекс>)` — генерируется при удалении компонента из контейнера. Через параметр внутри обработчика доступен целочисленный индекс удаленного компонента.

Класс `QStackedWidget` также реализует стек компонентов, но представляет собой полноценный компонент. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QStackedWidget
```

Формат конструктора этого класса:

```
QStackedWidget([<Родитель>])
```

Класс QStackedWidget поддерживает описанные ранее методы addWidget(), insertWidget(), removeWidget(), replaceWidget(), count(), currentIndex(), currentWidget(), widget(), setCurrentIndex() и setCurrentWidget(). Кроме того, класс QStackedWidget наследует все методы из базовых классов и определяет два дополнительных:

- indexOf(<Компонент>) — возвращает индекс компонента, ссылка на который указана в параметре;
- __len__() — возвращает количество компонентов. Метод вызывается при использовании функции len(), а также для проверки объекта на логическое значение.

Поддерживаются и описанные ранее сигналы currentChanged() и widgetRemoved().

21.6. Управление размерами компонентов

Если в вертикальный контейнер большой высоты добавить надпись и кнопку, то кнопка займет пространство, совпадающее с рекомендуемыми размерами (которые возвращает метод sizeHint()), а под надпись будет выделено все остальное место. Управление размерами компонентов внутри контейнера определяется правилами, установленными с помощью объектов класса QSizePolicy. Установить эти правила для компонента можно с помощью метода setSizePolicy(<Правило QSizePolicy>) класса QWidget, а получить их — вызовом метода sizePolicy().

Формат конструктора класса QSizePolicy:

```
QSizePolicy([<Правило для горизонтали>, <Правило для вертикали>[,
                                        <Тип компонента>]])
```

Если параметры не заданы, размер компонента будет точно соответствовать размерам, возвращаемым методом sizeHint(). В первом и втором параметрах указывается один из следующих элементов перечисления Policy из класса QSizePolicy:

- Fixed — размеры компонента будут точно соответствовать размерам, возвращаемым методом sizeHint();
- Minimum — размеры, возвращаемые методом sizeHint(), станут минимальными для компонента и могут быть увеличены при необходимости;
- Maximum — размеры, возвращаемые методом sizeHint(), станут максимальными для компонента и могут быть уменьшены при необходимости;
- Preferred — размеры, возвращаемые методом sizeHint(), являются предпочтительными и могут быть как увеличены, так и уменьшены;
- Expanding — размеры, возвращаемые методом sizeHint(), могут быть как увеличены, так и уменьшены. Компонент займет все свободное пространство в контейнере;
- MinimumExpanding — размеры, возвращаемые методом sizeHint(), являются минимальным для компонента. Компонент займет все свободное пространство в контейнере;
- Ignored — размеры, возвращаемые методом sizeHint(), игнорируются. Компонент займет все свободное пространство в контейнере.

В качестве типа компонента указывается один из следующих элементов перечисления ControlType из класса QSizePolicy: DefaultType (нет какого-либо специфического типа —

значение по умолчанию), ButtonBox (набор кнопок диалогового окна, представленный объектом класса QDialogButtonBox), CheckBox (флажок), ComboBox (раскрывающийся список), Frame (панель с рамкой), GroupBox (группа), Label (надпись), Line (панель с рамкой, выводящаяся в виде горизонтальной или вертикальной линии), LineEdit (поле ввода), PushButton (кнопка), RadioButton (переключатель), Slider (шкала с ползунком), SpinBox (поле для ввода целых чисел), TabWidget (панель с вкладками) или ToolButton (кнопка панели инструментов).

Пример:

```
sp = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Policy.Minimum,
                           QtWidgets.QSizePolicy.Policy.Fixed,
                           QtWidgets.QSizePolicy.ControlType.Frame)
```

Изменить правила управления размерами уже после создания объекта класса QSizePolicy позволяют методы setHorizontalPolicy(<Правило для горизонтали>) и setVerticalPolicy(<Правило для вертикали>).

Изменить тип компонента после создания объекта класса QSizePolicy позволяет метод setControlType(<Тип компонента>).

С помощью методов setHorizontalStretch(<Фактор для горизонтали>) и setVerticalStretch(<Фактор для вертикали>) можно указать фактор растяжения. Чем больше указанное значение относительно значения, заданного в других компонентах, тем больше места будет выделяться под текущий компонент.

Можно указать, что предпочтительная высота компонента зависит от его ширины. Для этого необходимо передать значение True в метод setHeightForWidth(<Флаг>). Кроме того, следует в классе компонента переопределить метод heightForWidth(<Ширина>) — переопределенный метод должен возвращать высоту компонента для указанной в параметре ширины.

Метод setRetainSizeWhenHidden(<Флаг>) позволяет указать поведение контейнера при скрытии компонента. Значение False параметра предписывает контейнеру полностью убирать как сам скрытый компонент, так и пространство, занятое скрытым компонентом, а значение True — убирать лишь компонент, оставляя занятое им пространство свободным.

21.7. Группа

Группа — это область, содержащая какой-либо набор компонентов и окруженная рамкой, на верхней границе которой выводится текст заголовка. Она реализуется классом QGroupBox. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QGroupBox
```

Формат конструктора класса QGroupBox:

```
QGroupBox([<Родитель>])
QGroupBox(<Текст заголовка>[, <Родитель>])
```

В тексте заголовка символ &, указанный перед буквой, задает комбинацию клавиш быстрого доступа.

После создания объекта класса QGroupBox следует добавить компоненты в какой-либо контейнер, а затем передать ссылку на этот контейнер в метод setLayout() группы.

Пример использования класса QGroupBox представлен кодом из листинга 21.5. Созданная им группа показана на рис. 21.4.

Листинг 21.5. Использование компонента QGroupBox

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QGroupBox")
window.resize(250, 80)
mainbox = QtWidgets.QVBoxLayout()  # Создаем главный контейнер
radio1 = QtWidgets.QRadioButton("&Да")
radio2 = QtWidgets.QRadioButton("&Нет")
box = QtWidgets.QGroupBox("&Вы знаете язык Python?") # Объект группы
hbox = QtWidgets.QHBoxLayout()      # Создаем второй контейнер — для группы
hbox.addWidget(radio1)              # Добавляем компоненты во второй контейнер
hbox.addWidget(radio2)
box.setLayout(hbox)                 # Добавляем второй контейнер в группу
mainbox.addWidget(box)              # Добавляем группу в главный контейнер
window.setLayout(mainbox)           # Помещаем главный контейнер в окно
radio1.setChecked(True)             # Выбираем первый переключатель
window.show()
sys.exit(app.exec())
```

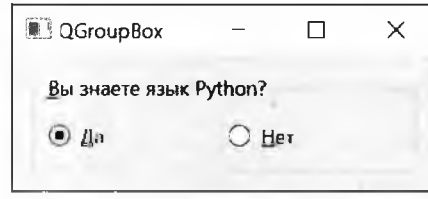


Рис. 20.4. Компонент QGroupBox

Класс QGroupBox поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qgroupbox.html**):

- setTitle(<Текст>) — задает текст заголовка;
- title() — возвращает текст заголовка;
- setAlignment(<Выравнивание>) — задает горизонтальное выравнивание текста заголовка. В параметре указывается один из следующих элементов перечисления AlignmentFlag из модуля QtCore.Qt: AlignLeft, AlignHCenter или AlignRight:

  ```
  box.setAlignment(QtCore.Qt.AlignmentFlag.AlignRight)
  ```

- alignment() — возвращает горизонтальное выравнивание текста заголовка;
- setCheckable(<Флаг>) — если в параметре указать значение True, то перед заголовком группы будет отображен флажок. Если флажок установлен, группа станет доступной для ввода, а если флажок снят — недоступной. По умолчанию флажок не отображается;

- `isCheckable()` — возвращает значение `True`, если перед заголовком группы выводится флажок, и `False` — в противном случае;
- `setChecked(<Флаг>)` — если в параметре указать значение `True`, флажок, отображаемый перед заголовком группы, будет установлен. Значение `False` сбрасывает флажок. Метод является слотом;
- `isChecked()` — возвращает значение `True`, если флажок, отображаемый перед заголовком группы, установлен, и `False` — в противном случае;
- `setFlat(<Флаг>)` — если в параметре указано значение `True`, будет отображаться только верхняя граница рамки, а если `False` — то все границы рамки;
- `isFlat()` — возвращает значение `True`, если отображается только верхняя граница рамки, и `False` — если все границы рамки.

Класс `QGroupBox` поддерживает сигналы:

- `clicked(<Состояние флажка>)` — генерируется при щелчке мышью на флажке, выводимом перед заголовком группы. Если состояние флажка изменяется с помощью метода `setChecked()`, сигнал не генерируется. Через параметр внутри обработчика доступно значение `True`, если флажок установлен, и `False` — если сброшен;
- `toggled(<Состояние флажка>)` — генерируется при изменении состояния флажка, выводимого перед заголовком группы. Через параметр внутри обработчика доступно значение `True`, если флажок установлен, и `False` — если сброшен.

21.8. Панель с рамкой

Класс `QFrame` расширяет возможности класса `QWidget`, добавляя рамку вокруг компонента. Этот класс, в свою очередь, наследуют некоторые компоненты — например, надписи, многострочные текстовые поля и др. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame
```

Конструктор класса `QFrame` имеет следующий формат:

```
QFrame([parent=None][, flags=0])
```

В параметре `parent` указывается ссылка на родительский компонент. Если этот параметр имеет значение `None` и у параметра `flags` указано значение 0, компонент не будет иметь родителя и получит свое собственное окно. Параметр `flags` указывает тип создаваемого окна. Доступные для задания типы рассмотрены в *разд. 19.1.1*.

Класс `QFrame` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qframe.html**):

- `setFrameShape(<Форма>)` — задает форму рамки. Могут быть указаны следующие элементы перечисления `Shape` из класса `QFrame`:
  - `NoFrame` — нет рамки;
  - `Box` — прямоугольная рамка;
  - `Panel` — панель, которая может быть выпуклой или вогнутой;
  - `WinPanel` — панель в стиле Windows 2000, которая может быть выпуклой или вогнутой. Толщина границы — 2 пиксела. Этот элемент присутствует для совместимости со старыми версиями Qt;

- `HLine` — горизонтальная линия. Используется как разделитель;
- `VLine` — вертикальная линия;
- `StyledPanel` — панель, внешний вид которой зависит от текущего стиля. Она может быть выпуклой или вогнутой. Это рекомендуемая форма рамки для панелей;

♦ `setFrameShadow(<Тень>)` — задает стиль тени. Могут быть указаны следующие элементы перечисления `Shadow` из класса `QFrame`:

- `Plain` — нет тени;
- `Raised` — панель отображается выпуклой;
- `Sunken` — панель отображается вогнутой;

♦ `setFrameStyle(<Стиль>)` — задает форму рамки и стиль тени одновременно. В качестве значения через оператор `|` указывается комбинация приведенных ранее элементов перечислений `Shape` и `Shadow` класса `QFrame`:

```
frame.setFrameStyle(QtWidgets.QFrame.Shape.Panel |
                    QtWidgets.QFrame.Shadow.Raised)
```

♦ `setLineWidth(<Ширина>)` — задает толщину линий у рамки;

♦ `setMidLineWidth(<Ширина>)` — задает толщину средней линии у рамки. Средняя линия используется для создания эффекта выпуклости и вогнутости и доступна только для форм рамки `Box`, `HLine` и `VLine`

♦ `setFrameRect(<Область QRect>)` — задает местоположение и размеры области внутри панели, вокруг которой будет проведена рамка, в виде объекта класса `QRect`. По умолчанию рамка рисуется вокруг всей панели.

21.9. Панель с вкладками

Панели с вкладками (*блокнот*) реализуется классом `QTabWidget`. Иерархия наследования у него выглядит так:

```
(QObject, QPaintDevice) — QWidget — QTabWidget
```

Конструктор класса `QTabWidget` имеет следующий формат:

```
QTabWidget([<Родитель>])
```

В параметре `<Родитель>` указывается ссылка на родительский компонент. Если он не указан, компонент будет обладать своим собственным окном.

Пример кода, создающего компонент `QTabWidget`, приведен в листинге 21.6. Сама панель с вкладками показана на рис. 21.5.

Листинг 21.6. Использование компонента `QTabWidget`

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QTabWidget")
window.resize(300, 100)
```

```
tab = QtWidgets.QTabWidget()
tab.addTab(QtWidgets.QLabel("Содержимое вкладки 1"), "Вкладка &1")
tab.addTab(QtWidgets.QLabel("Содержимое вкладки 2"), "Вкладка &2")
tab.addTab(QtWidgets.QLabel("Содержимое вкладки 3"), "Вкладка &3")
tab.setCurrentIndex(0)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(tab)
window.setLayout(vbox)
window.show()
window.show()
sys.exit(app.exec())
```

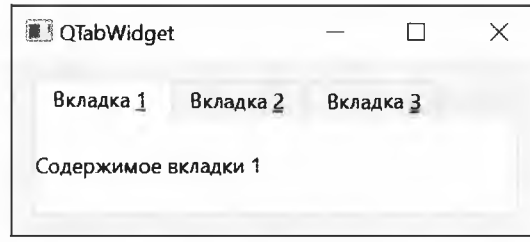


Рис. 21.5. Компонент `QTabWidget`

Класс `QTabWidget` поддерживает следующие методы (здесь приведены только основные — полное описание класса содержится на странице **https://doc.qt.io/qt-6/qtabwidget.html**):

- `addTab()` — добавляет вкладку в конец контейнера и возвращает ее индекс. Форматы метода:

  ```
  addTab(<Компонент>, <Текст ярлыка>)
  addTab(<Компонент>, <Значок QIcon>, <Текст ярлыка>)
  ```

  В параметре `<Компонент>` указывается ссылка на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр `<Текст ярлыка>` задает текст, который будет отображаться на ярлыке вкладки. Внутри текста ярлыка символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. Параметр `<Значок>` указывает значок, который отобразится перед текстом на ярлыке вкладки.

  Пример указания стандартного значка:

  ```
  style = window.style()
  icon = style.standardIcon(QtWidgets.QStyle.StandardPixmap.SP_DriveNetIcon)
  tab.addTab(QtWidgets.QLabel("Содержимое вкладки 1"), icon, "Вкладка &1")
  ```

  Пример загрузки значка из файла:

  ```
  icon = QtGui.QIcon("icon.png")
  tab.addTab(QtWidgets.QLabel("Содержимое вкладки 1"), icon, "Вкладка &1")
  ```

- `insertTab()` — добавляет вкладку по указанному индексу и возвращает индекс добавленной вкладки. Форматы метода:

  ```
  insertTab(<Индекс>, <Компонент>, <Текст ярлыка>)
  insertTab(<Индекс>, <Компонент>, <Значок QIcon>, <Текст ярлыка>)
  ```

- `removeTab(<Индекс>)` — удаляет вкладку с указанным индексом, при этом компонент, который отображался на вкладке, не удаляется;
- `clear()` — удаляет все вкладки, при этом компоненты, которые отображались на вкладках, не удаляются;
- `setTabText(<Индекс>, <Текст заголовка>)` — задает текст на ярлыке вкладки с указанным индексом;
- `setElideMode(<Режим>)` — задает режим обрезки текста на ярлыке вкладки, если он не помещается в отведенную область (в месте обрезки выводится троеточие). Могут быть указаны следующие элементы перечисления `TextElideMode` из модуля `QtCore.Qt`:
  - `ElideLeft` — текст обрезается слева;
  - `ElideRight` — текст обрезается справа;
  - `ElideMiddle` — текст вырезается посередине;
  - `ElideNone` — текст не обрезается;
- `tabText(<Индекс>)` — возвращает текст на ярлыке вкладки с указанным индексом;
- `setTabIcon(<Индекс>, <Значок QIcon>)` — устанавливает значок перед текстом на ярлыке вкладки с указанным индексом;
- `setIconSize(<Размеры QSize>)` — задает размеры значков, выводящихся на ярлыках вкладок;
- `setTabPosition(<Позиция>)` — задает местоположение заголовка панели, на котором выводятся ярлыки вкладок. Могут быть указаны следующие элементы перечисления `TabPosition` из класса `QTabWidget`: `North` (сверху), `South` (снизу), `West` (слева) или `East` (справа). Пример:

```
tab.setTabPosition(QtWidgets.QTabWidget.TabPosition.South)
```

- `setTabShape(<Форма>)` — задает форму углов у ярлыков вкладок. Могут быть указаны следующие элементы перечисления `TabShape` из класса `QTabWidget`: `Rounded` (скругленная форма — значение по умолчанию) или `Triangular` (треугольная форма);
- `setTabsClosable(<Флаг>)` — если в качестве параметра указано значение `True`, то после текста на ярлыке вкладки будет отображена кнопка ее закрытия. При нажатии этой кнопки генерируется сигнал `tabCloseRequested`;
- `setMovable(<Флаг>)` — если в качестве параметра указано значение `True`, вкладки можно перетаскивать мышью за их ярлыки;
- `setDocumentMode(<Флаг>)` — если в качестве параметра указано значение `True`, панель не будет окружена рамкой;
- `setUsesScrollButtons(<Флаг>)` — если в качестве параметра указано значение `True`, то, если все ярлыки вкладок не помещаются в заголовке панели, появятся две кнопки, с помощью которых можно прокручивать содержимое заголовка. Значение `False` предписывает панели выводить все ярлыки в несколько строк;
- `setTabBarAutoHide(<Флаг>)` — если в качестве параметра указано значение `True`, и в панели присутствует всего одна вкладка (или ни одной), заголовок панели будет скрыт. Значение `False` указывает панели всегда отображать заголовок;
- `setTabToolTip(<Индекс>, <Текст>)` — задает текст всплывающей подсказки у ярлыка вкладки с указанным индексом;

- `setTabWhatsThis(<Индекс>, <Текст>)` — задает текст расширенной подсказки у ярлыка вкладки с указанным индексом;
- `setCurrentIndex(<Индекс>)` — делает видимой вкладку с указанным индексом. Метод является слотом;
- `currentIndex()` — возвращает индекс видимой вкладки;
- `setCurrentWidget(<Компонент>)` — делает видимой вкладку с указанным компонентом. Метод является слотом;
- `setTabEnabled(<Индекс>, <Флаг>)` — если вторым параметром передано значение `False`, вкладка с указанным индексом станет недоступной. Значение `True` делает вкладку доступной;
- `isTabEnabled(<Индекс>)` — возвращает значение `True`, если вкладка с указанным индексом доступна, и `False` — в противном случае;
- `setTabVisible(<Индекс>, <Флаг>)` — если вторым параметром передано значение `False`, вкладка с указанным индексом станет невидимой. Значение `True` делает вкладку видимой;
- `isTabVisible(<Индекс>)` — возвращает значение `True`, если вкладка с указанным индексом видима, и `False` — в противном случае;
- `currentWidget()` — возвращает ссылку на компонент, расположенный на видимой вкладке;
- `widget(<Индекс>)` — возвращает ссылку на компонент, расположенный на вкладке с указанным индексом, или значение `None` в случае обращения по несуществующему индексу;
- `indexOf(<Компонент>)` — возвращает индекс вкладки, на которой расположен заданный компонент. Если компонент не найден, возвращается значение `-1`;
- `count()` — возвращает количество вкладок. Получить количество вкладок можно также с помощью функции `len()`:

```
print(tab.count(), len(tab))
```

Класс `QTabWidget` поддерживает такие сигналы:

- `currentChanged(<Индекс>)` — генерируется при переключении на другую вкладку. Через параметр внутри обработчика доступен индекс выведенной вкладки;
- `tabBarClicked(<Индекс>)` — генерируется при щелчке на ярлыке вкладки. Через параметр внутри обработчика доступен индекс вкладки, на которой был выполнен щелчок, или значение `-1`, если щелчок был выполнен на области заголовка панели, не занятой ярлыками;
- `tabBarDoubleClicked(<Индекс>)` — генерируется при двойном щелчке на ярлыке вкладки. Через параметр внутри обработчика доступен индекс вкладки, на которой был выполнен двойной щелчок, или значение `-1`, если щелчок был выполнен на области заголовка панели, не занятой ярлыками;
- `tabCloseRequested(<Индекс>)` — генерируется при нажатии кнопки закрытия вкладки. Через параметр внутри обработчика доступен индекс закрываемой вкладки.

21.10. Аккордеон

Аккордеон похож на панель с вкладками, только отдельные панели в нем расположены по вертикали: при переключении на какую-либо панель она разворачивается, а развернутая ранее — сворачивается. Аккордеон представляется классом `QToolBox`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QToolBox
```

Конструктор класса `QToolBox` имеет следующий формат:

```
QToolBox([parent=None][, flags=0])
```

В параметре `parent` указывается ссылка на родительский компонент. Если этот параметр имеет значение `None`, и у параметра `flags` указано значение 0, компонент не будет иметь родителя и получит свое собственное окно. Параметр `flags` указывает тип создаваемого окна. Доступные для задания типы рассмотрены в *разд. 19.1.1*.

Пример кода, создающего аккордеон, представлен в листинге 21.7, а созданный им аккордеон — на рис. 21.6.

Листинг 21.7. Использование класса QToolBox

```
from PyQt6 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QToolBox")
window.resize(250, 100)
toolBox = QtWidgets.QToolBox()
toolBox.addItem(QtWidgets.QLabel("Содержимое вкладки 1"), "Вкладка &1")
toolBox.addItem(QtWidgets.QLabel("Содержимое вкладки 2"), "Вкладка &2")
toolBox.addItem(QtWidgets.QLabel("Содержимое вкладки 3"), "Вкладка &3")
toolBox.setCurrentIndex(0)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(toolBox)
window.setLayout(vbox)
window.show()
sys.exit(app.exec())
```

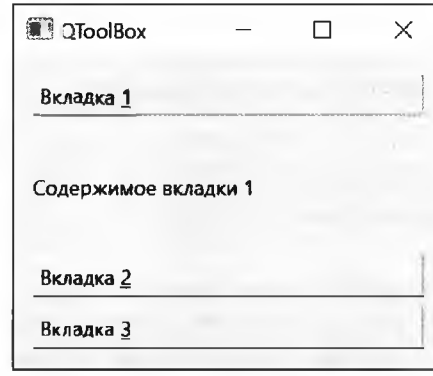


Рис. 21.6. Компонент `QToolBox`

Класс `QToolBox` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qtoolbox.html**):

- `addItem()` — добавляет вкладку в конец контейнера. Метод возвращает индекс добавленной вкладки. Форматы метода:

  ```
  addItem(<Компонент>, <Текст ярлыка>)
  addItem(<Компонент>, <Значок QIcon>, <Текст ярлыка>)
  ```

  В параметре `<Компонент>` указывается ссылка на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр `<Текст ярлыка>` задает текст, который будет отображаться на ярлыке вкладки. Внутри текста вкладки символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. Параметр `<Значок>` указывает значок, который отобразится перед текстом на ярлыке вкладки;

- `insertItem()` — добавляет вкладку по указанному индексу и возвращает индекс добавленной вкладки. Форматы метода:

  ```
  insertItem(<Индекс>, <Компонент>, <Текст ярлыка>)
  insertItem(<Индекс>, <Компонент>, <Значок QIcon>, <Текст заголовка>)
  ```

- `removeItem(<Индекс>)` — удаляет вкладку с указанным индексом, при этом компонент, который отображался на вкладке, не удаляется;

- `setItemText(<Индекс>, <Текст заголовка>)` — задает текст ярлыка у вкладки с указанным индексом;

- `itemText(<Индекс>)` — возвращает текст ярлыка у вкладки с указанным индексом;

- `setItemIcon(<Индекс>, <Значок QIcon>)` — устанавливает значок на ярлыке вкладки с указанным индексом;

- `setItemToolTip(<Индекс>, <Текст>)` — задает текст всплывающей подсказки у ярлыка вкладки с указанным индексом;

- `setItemEnabled(<Индекс>, <Флаг>)` — если вторым параметром передается значение `False`, вкладка с указанным в первом параметре индексом станет недоступной. Значение `True` делает вкладку доступной;

- `isItemEnabled(<Индекс>)` — возвращает значение `True`, если вкладка с указанным индексом доступна, и `False` — в противном случае;

- `setCurrentIndex(<Индекс>)` — делает видимой вкладку с указанным индексом. Метод является слотом;

- `currentIndex()` — возвращает индекс видимой вкладки;

- `setCurrentWidget(<Компонент>)` — разворачивает вкладку с указанным компонентом. Метод является слотом;

- `currentWidget()` — возвращает ссылку на компонент, расположенный на развернутой вкладке;

- `widget(<Индекс>)` — возвращает ссылку на компонент, расположенный на вкладке с указанным индексом, или значение `None` в случае обращения по несуществующему индексу;

- `indexOf(<Компонент>)` — возвращает индекс вкладки, на которой расположен заданный компонент. Если компонент не найден, возвращается значение `-1`;

- `count()` — возвращает количество вкладок. Получить количество вкладок можно также с помощью функции `len()`:

```
print(toolBox.count(), len(toolBox))
```

При переключении на другую вкладку генерируется сигнал `currentChanged(<Индекс>)`. Через параметр внутри обработчика доступен индекс вкладки, на которую было выполнено переключение.

21.11. Панель с изменяемыми областями

Панель с изменяемыми областями включает ряд областей, каждая из которых содержит один компонент. На границах между областями располагаются захваты, перемещая которые с помощью мыши можно менять относительные размеры этих областей. Такая панель реализуется классом `QSplitter`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QSplitter
```

Конструктор класса `QSplitter` имеет два формата:

```
QSplitter([parent=None])
QSplitter(<Ориентация>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент. Если таковой не указан или имеет значение `None`, компонент будет обладать своим собственным окном. Параметр `<Ориентация>` задает ориентацию расположения областей в виде одного из следующих элементов перечисления `Orientation` из модуля `QtCore.Qt`: `Horizontal` (горизонтальная — значение по умолчанию) или `Vertical` (вертикальная).

Пример использования класса `QSplitter` показан в листинге 21.8, а результат можно увидеть на рис. 21.7.

Листинг 21.8. Использование класса `QSplitter`

```
from PyQt6 import QtCore, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QSplitter")
window.resize(250, 200)
splitter = QtWidgets.QSplitter(QtCore.Qt.Orientation.Vertical)
label1 = QtWidgets.QLabel("Содержимое компонента 1")
label2 = QtWidgets.QLabel("Содержимое компонента 2")
label1.setFrameStyle(QtWidgets.QFrame.Shape.Box |
                     QtWidgets.QFrame.Shadow.Plain)
label2.setFrameStyle(QtWidgets.QFrame.Shape.Box |
                     QtWidgets.QFrame.Shadow.Plain)
splitter.addWidget(label1)
splitter.addWidget(label2)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(splitter)
window.setLayout(vbox)
window.show()
sys.exit(app.exec())
```

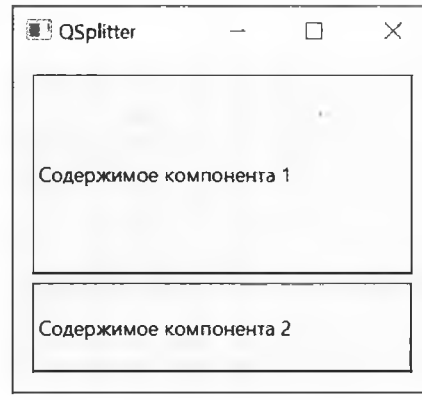


Рис. 21.7. Компонент `QSplitter`

Класс `QSplitter` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qsplitter.html**):

- `addWidget(<Компонент>)` — добавляет компонент в конец панели, создавая в ней новую область;
- `insertWidget(<Индекс>, <Компонент>)` — добавляет компонент по указанному индексу. Если компонент был добавлен ранее, он будет перемещен по указанному индексу;
- `setOrientation(<Ориентация>)` — задает ориентацию расположения областей. Могут быть заданы элементы `Horizontal` (горизонтальная) или `Vertical` (вертикальная) перечисления `Orientation` из модуля `QtCore.Qt`;
- `setHandleWidth(<Ширина>)` — задает толщину захвата;
- `saveState()` — возвращает объект класса `QByteArray` с размерами всех областей. Эти данные можно сохранить (например, в файл), а затем восстановить с помощью метода `restoreState()`;
- `restoreState(<Объект QByteArray>)` — восстанавливает размеры областей из объекта класса `QByteArray`, возвращенного методом `saveState()`;
- `setChildrenCollapsible(<Флаг>)` — если в параметре указано значение `False`, пользователь не сможет уменьшить размеры всех областей до нуля. По умолчанию размер может быть нулевым, даже если для какого-либо компонента установлены минимальные размеры;
- `setCollapsible(<Индекс>, <Флаг>)` — значение `False` в параметре `<Флаг>` запрещает уменьшение размеров до нуля у компонента с указанным индексом;
- `setOpaqueResize(<Флаг>)` — если в качестве параметра указано значение `False`, размеры областей изменятся только после окончания перемещения захвата и отпускания кнопки мыши. В процессе перемещения мыши вместе с ней будет перемещаться специальный компонент в виде линии;
- `setStretchFactor(<Индекс>, <Фактор>)` — задает целочисленный фактор растяжения для компонента с указанным индексом;
- `setSizes(<Список>)` — задает размеры всех областей. У панели с горизонтальной ориентацией указывается список со значениями ширины каждой области, у панели с вертикальной ориентацией — список со значениями высоты каждой области;

- `sizes()` — возвращает список с размерами (шириной или высотой):

  ```
  print(splitter.sizes()) # Результат: [308, 15]
  ```

- `count()` — возвращает количество областей. Получить количество областей можно также с помощью функции `len()`:

  ```
  print(splitter.count(), len(splitter))
  ```

- `widget(<Индекс>)` — возвращает ссылку на компонент, расположенный по указанному индексу, или значение `None` в случае обращения по несуществующему индексу;
- `indexOf(<Компонент>)` — возвращает индекс области, в которой расположен заданный компонент. Если таковой не найден, возвращается значение `-1`.

При перемещении какого-либо из захватов генерируется сигнал `splitterMoved(<Позиция>, <Индекс>)`. Через первый параметр внутри обработчика доступна новая позиция перемещенного захвата, а через второй параметр — индекс этого захвата.

21.12. Прокручиваемая панель

Панель с полосами прокрутки (которые выводятся автоматически, если содержимое панели не помещается в ней) представляется классом `QScrollArea`. Иерархия наследования выглядит так:

```
QObject, QPaintDevice) — QWidget — QFrame —
                         QAbstractScrollArea — QScrollArea
```

Конструктор класса `QScrollArea` имеет следующий формат:

```
QScrollArea([<Родитель>])
```

Класс `QScrollArea` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qscrollarea.html**):

- `setWidget(<Компонент>)` — помещает заданный компонент в панель;
- `setWidgetResizable(<Флаг>)` — если в качестве параметра указано значение `True`, при изменении размеров панели будут изменяться и размеры компонента. Значение `False` запрещает изменение размеров компонента;
- `setAlignment(<Выравнивание>)` — задает местоположение компонента внутри панели, когда размеры панели больше размеров компонента, в виде одного из элементов перечисления `AlignmentFlag` из модуля `QtCore.Qt` (см. *разд. 21.2*):

  ```
  scrollArea.setAlignment(QtCore.Qt.AlignmentFlag.AlignCenter)
  ```

- `ensureVisible(<X>, <Y>[, xMargin=50][, yMargin=50])` — прокручивает панель таким образом, чтобы точка с координатами `(<X>, <Y>)` оказалась в видимой области. Параметры `xMargin` и `yMargin` задают отступы соответственно по горизонтали и вертикали между границами панели и видимой области;
- `ensureWidgetVisible(<Компонент>[, xMargin=50][, yMargin=50])` — прокручивает панель таким образом, чтобы `<Компонент>` оказался в видимой области. Параметры `xMargin` и `yMargin` задают отступы соответственно по горизонтали и вертикали между границами панели и видимой области;
- `widget()` — возвращает ссылку на компонент, который расположен внутри панели, или значение `None`, если панель пуста;

- `takeWidget()` — удаляет компонент из панели и возвращает ссылку на него. Сам компонент не удаляется.

Класс `QScrollArea` наследует следующие методы из класса `QAbstractScrollArea` (здесь перечислены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qabstractscrollarea.html**):

- `horizontalScrollBar()` — возвращает ссылку на горизонтальную полосу прокрутки (объект класса `QScrollBar`);
- `verticalScrollBar()` — возвращает ссылку на вертикальную полосу прокрутки (объект класса `QScrollBar`);
- `setHorizontalScrollBarPolicy(<Режим>)` — устанавливает режим отображения горизонтальной полосы прокрутки;
- `setVerticalScrollBarPolicy(<Режим>)` — устанавливает режим отображения вертикальной полосы прокрутки.

  В параметре `<Режим>` могут быть указаны следующие элементы перечисления `ScrollBarPolicy` из модуля `QtCore.Qt`:

  - `ScrollBarAsNeeded` — полоса прокрутки отображается только в том случае, если размеры компонента больше размеров панели;
  - `ScrollBarAlwaysOff` — полоса прокрутки никогда не отображается;
  - `ScrollBarAlwaysOn` — полоса прокрутки отображается всегда.

ГЛАВА 22

Основные компоненты

Практически все основные компоненты (за исключением `QWebEngineView`) определены в модуле `QtWidgets`, наследуют класс `QWidget` и, соответственно, поддерживают все методы этого класса (были описаны в *главах 19* и *20*). Если компонент не имеет родителя, он будет выводиться в собственном окне.

22.1. Надпись

Надпись применяется для вывода подсказки пользователю, информирования пользователя о ходе выполнения операции, назначении клавиш быстрого доступа и т. п. Надпись может выводить обычный текст, текст, отформатированный HTML-тегами или тегами Markdown.

Надпись реализуется классом `QLabel`. Иерархия наследования выглядит так:

```
QObject, QPaintDevice) — QWidget — QFrame — QLabel
```

Конструктор класса `QLabel` имеет два формата:

```
QLabel([parent=None][, flags=0])
QLabel(<Текст>[, parent=None][, flags=0])
```

В параметре `parent` указывается ссылка на родительский компонент. Если он не указан или имеет значение `None`, компонент будет обладать своим собственным окном, тип которого можно задать с помощью параметра `flags`. Параметр `<Текст>` позволяет задать текст, который будет отображен на надписи. Пример:

```
label = QtWidgets.QLabel("Текст надписи", flags=QtCore.Qt.WindowType.Window)
label.resize(300, 50)
label.show()
```

Класс `QLabel` поддерживает следующие основные методы (полный их список смотрите на странице **https://doc.qt.io/qt-6/qlabel.html**):

- `setText(<Текст>)` — задает текст, который будет отображен на надписи. Можно указать как обычный текст, так и содержащий CSS-форматирование текст в формате HTML. Пример:

  ```
  label.setText("Текст <b>полужирный</b>")
  ```

  Перевод строки в простом тексте осуществляется с помощью символа `\n`, а в тексте в формате HTML — с помощью тега `<br>`:

  ```
  label.setText("Текст\nна двух строках")
  ```

Внутри текста символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ &, будет — в качестве подсказки пользователю — подчеркнута. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент, ссылка на который передана в метод `setBuddy()`, окажется в фокусе ввода. Чтобы вывести сам символ &, необходимо его удвоить. Если надпись не связана с другим компонентом, символ & выводится в составе текста. Пример:

```
label = QtWidgets.QLabel("&Пароль")
lineEdit = QtWidgets.QLineEdit()
label.setBuddy(lineEdit)
```

Метод является слотом;

- `setNum(<Число>)` — выводит в надписи заданное целое или вещественное число. Метод является слотом;
- `setWordWrap(<Флаг>)` — если в параметре указано значение `True`, текст может переноситься на другую строку. По умолчанию перенос строк не осуществляется;
- `text()` — возвращает текст надписи;
- `setTextFormat(<Режим>)` — задает режим отображения текста. Могут быть указаны следующие элементы перечисления `TextFormat` из модуля `QtCore.Qt`:
  - `PlainText` — простой текст;
  - `RichText` — текст, отформатированный тегами HTML;
  - `AutoText` — автоматическое определение (режим по умолчанию). Если текст содержит HTML-теги, то используется режим `RichText`, в противном случае — режим `PlainText`;
  - `MarkdownText` — текст, отформатированный тегами Markdown;
- `setAlignment(<Режим>)` — задает режим выравнивания текста внутри надписи (допустимые значения рассмотрены в *разд. 21.2*):

```
label.setAlignment(QtCore.Qt.AlignmentFlag.AlignRight |
                   QtCore.Qt.AlignmentFlag.AlignBottom)
```

- `setOpenExternalLinks(<Флаг>)` — если в качестве параметра указано значение `True`, теги <a>, присутствующие в тексте, будут преобразованы в гиперссылки:

```
label.setText('<a href="https://www.google.ru/">Это гиперссылка</a>')
label.setOpenExternalLinks(True)
```

- `setBuddy(<Компонент>)` — связывает надпись с другим компонентом. В этом случае в тексте надписи можно задавать клавиши быстрого доступа, указав символ & перед буквой или цифрой;
- `setPixmap(<Изображение QPixmap>)` — позволяет вывести изображение на надпись:

```
label.setPixmap(QtGui.QPixmap("picture.jpg"))
```

Метод является слотом;

- `setPicture(<Рисунок QPicture>)` — выводит заданный рисунок. Метод является слотом;
- `setScaledContents(<Флаг>)` — если в параметре указано значение `True`, то при изменении размеров надписи размер содержимого также будет изменяться. По умолчанию изменение размеров содержимого не осуществляется;

- `setMargin(<Отступ>)` — задает отступы от границ компонента до его содержимого;
- `setIndent(<Отступ>)` — задает отступ от рамки до текста надписи в зависимости от значения выравнивания. Если выравнивание производится по левой стороне, то задает отступ слева, если по правой стороне, то справа;
- `clear()` — удаляет содержимое надписи. Метод является слотом;
- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом надписи. Можно указать следующие элементы (или их комбинацию через оператор |) перечисления `TextInteractionFlag` из модуля `QtCore.Qt`:
  - `NoTextInteraction` — пользователь не может взаимодействовать с текстом надписи;
  - `TextSelectableByMouse` — текст можно выделить мышью;
  - `TextSelectableByKeyboard` — текст можно выделить с помощью клавиш на клавиатуре. Внутри надписи будет отображен текстовый курсор;
  - `LinksAccessibleByMouse` — на гиперссылках, присутствующих в тексте надписи, можно щелкать мышью;
  - `LinksAccessibleByKeyboard` — с гиперссылками, присутствующими в тексте надписи, допускается взаимодействовать с помощью клавиатуры: перемещаться между гиперссылками — нажатиями клавиши <Tab>, а переходить по гиперссылке — по нажатии клавиши <Enter>;
  - `TextEditable` — текст надписи можно редактировать;
  - `TextEditorInteraction` — комбинация `TextSelectableByMouse | TextSelectableByKeyboard | TextEditable`;
  - `TextBrowserInteraction` — комбинация `TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard`;
- `setSelection(<Индекс>, <Длина>)` — выделяет фрагмент длиной `<Длина>`, начиная с позиции `<Индекс>`;
- `selectionStart()` — возвращает начальный индекс выделенного фрагмента или значение `-1`, если ничего не выделено;
- `selectedText()` — возвращает выделенный текст или пустую строку, если ничего не выделено;
- `hasSelectedText()` — возвращает значение `True`, если фрагмент текста надписи выделен, и `False` — в противном случае.

Класс `QLabel` поддерживает следующие сигналы:

- `linkActivated(<Интернет-адрес>)` — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен целевой интернет-адрес, заданный в виде строки;
- `linkHovered(<Интернет-адрес>)` — генерируется при наведении курсора мыши на гиперссылку. Через параметр внутри обработчика доступен интернет-адрес гиперссылки в виде строки или пустая строка.

22.2. Кнопка

Кнопка реализуется классом `QPushButton`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QPushButton
```

Конструктор класса `QPushButton` имеет три формата:

```
QPushButton([parent=None])
QPushButton(<Текст>[, parent=None])
QPushButton(<Значок QIcon>, <Текст>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент. Если таковой не задан или имеет значение `None`, компонент будет обладать своим собственным окном. Параметр `<Текст>` задает текст, который отобразится на кнопке, а параметр `<Значок>` — значок, выводимый перед текстом.

Класс `QPushButton` наследует следующие методы из класса `QAbstractButton` (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qabstractbutton.html**):

- `setText(<Текст>)` — задает текст, который будет отображен на кнопке. Внутри текста символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет — в качестве подсказки пользователю — подчеркнута. Одновременное нажатие клавиши <Alt> и подчеркнутой буквы приведет к нажатию этой кнопки. Чтобы вывести сам символ `&`, необходимо его удвоить;
- `text()` — возвращает текст, отображаемый на кнопке;
- `setShortcut(<Клавиша QKeySequence>)` — задает клавишу быстрого доступа:

```
button.setShortcut("Alt+B")
button.setShortcut(QtGui.QKeySequence.mnemonic("&B"))
button.setShortcut(QtGui.QKeySequence("Alt+B"))
button.setShortcut(QtGui.QKeySequence(QtCore.Qt.Key.Key_F5))
```

- `setIcon(<Значок QIcon>)` — вставляет значок перед текстом кнопки;
- `setIconSize(<Размеры QSize>)` — задает размеры значка. Метод является слотом;
- `setAutoRepeat(<Флаг>)` — если в качестве параметра указано значение `True`, сигнал `clicked` будет периодически генерироваться, пока кнопка находится в нажатом состоянии;
- `animateClick()` — имитирует нажатие пользователем кнопки с анимацией. После нажатия кнопка находится в этом состоянии в течении 100 миллисекунд. Метод является слотом;
- `click()` — имитирует нажатие кнопки без анимации. Метод является слотом;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка является переключателем, который может находиться в двух состояниях: установленном и неустановленном;
- `setChecked(<Флаг>)` — если в качестве параметра указано значение `True`, кнопка-переключатель будет находиться в установленном состоянии. Метод является слотом;
- `isChecked()` — возвращает значение `True`, если кнопка-переключатель находится в установленном состоянии, и `False` — в противном случае;
- `toggle()` — переключает кнопку-переключатель. Метод является слотом;
- `setAutoExclusive(<Флаг>)` — если в качестве параметра указано значение `True`, внутри контейнера может быть установлена только одна кнопка-переключатель;

- `setDown(<Флаг>)` — если в качестве параметра указано значение `True`, кнопка будет находиться в нажатом состоянии;
- `isDown()` — возвращает значение `True`, если кнопка находится в нажатом состоянии, и `False` — в противном случае.

Класс `QAbstractButton` поддерживает следующие сигналы:

- `pressed()` — генерируется при нажатии кнопки;
- `released()` — генерируется при отпускании ранее нажатой кнопки;
- `clicked(<Состояние>)` — генерируется при нажатии и отпускании кнопки. Передаваемый обработчику параметр имеет значение `True`, если кнопка-переключатель установлена, и `False`, если она сброшена или это обычная кнопка, а не переключатель;
- `toggled(<Состояние>)` — генерируется при изменении состояния кнопки-переключателя. Через параметр доступно новое состояние кнопки.

Класс `QPushButton` определяет свои собственные методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qpushbutton.html**):

- `setFlat(<Флаг>)` — если в качестве параметра указано значение `True`, кнопка будет отображаться без рамки;
- `setAutoDefault(<Флаг>)` — если в качестве параметра указано значение `True`, кнопка может быть нажата с помощью клавиши <Enter> при условии, что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>. В диалоговых окнах у всех кнопок по умолчанию указано значение `True`, а в остальных окнах — значение `False`;
- `setDefault(<Флаг>)` — задает кнопку по умолчанию. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент, — например, на текстовое поле. Метод работает только в диалоговых окнах;
- `setMenu(<Меню QMenu>)` — устанавливает всплывающее меню, которое будет отображаться при нажатии кнопки;
- `menu()` — возвращает ссылку на всплывающее меню или значение `None`;
- `showMenu()` — отображает всплывающее меню. Метод является слотом.

22.3. Переключатель

Переключатели всегда компонуются группами. Для объединения переключателей в группу можно воспользоваться компонентом `QGroupBox` (см *разд. 21.7)* или классом `QButtonGroup`.

Переключатель реализуется классом `QRadioButton`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QRadioButton
```

Конструктор класса `QRadioButton` имеет два формата:

```
QRadioButton([parent=<None>])
QRadioButton(<Текст>[, parent=None])
```

Класс `QRadioButton` наследует все методы класса `QAbstractButton` (см. *разд. 22.2*). Установить или сбросить переключатель позволяет метод `setChecked(<Флаг>)`, а проверить его текущее состояние можно с помощью метода `isChecked()`. Отследить изменение состояния

можно в обработчике сигнала `toggled(<Состояние>)`, в параметре которого передается логическая величина, указывающая новое состояние переключателя.

22.4. Флажок

Флажок может находиться в нескольких состояниях: установленном, сброшенном и промежуточном (неопределенном) — последнее состояние может быть запрещено программно. Флажок реализуется с помощью класса `QCheckBox`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QCheckBox
```

Конструктор класса `QCheckBox` имеет два формата:

```
QCheckBox([parent=None])
QCheckBox(<Текст>[, parent=None])
```

Класс `QCheckBox` наследует все методы класса `QAbstractButton` (см. *разд. 22.2*), а также добавляет несколько новых:

- `setCheckState(<Состояние>)` — задает состояние флажка. Могут быть указаны следующие элементы перечисления `CheckState` из модуля `QtCore.Qt`:
  - `Unchecked` — флажок сброшен;
  - `PartiallyChecked` — флажок находится в промежуточном состоянии;
  - `Checked` — флажок установлен;
- `checkState()` — возвращает текущее состояние флажка;
- `setTristate([<Флаг>=True])` — если в качестве параметра указано значение `True` (значение по умолчанию), флажок может находиться во всех трех состояниях. По умолчанию поддерживаются только установленное и сброшенное состояния;
- `isTristate()` — возвращает значение `True`, если флажок поддерживает три состояния, и `False` — если только два.

Чтобы перехватить изменение состояния флажка, следует назначить обработчик сигнала `stateChanged(<Состояние>)`. Через параметр внутри обработчика доступно новое состояние флажка, заданное в виде целого числа: 0 (сброшенное), 1 (промежуточное) или 2 (установленное).

Если используется флажок, поддерживающий только два состояния, установить или сбросить его позволяет метод `setChecked()`, а проверить текущее состояние — метод `isChecked`. Обработать изменение состояния можно в обработчике сигнала `toggled(<Состояние>)`, параметр которого имеет логический тип.

22.5. Поле ввода

Поле ввода самостоятельно поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Оно реализуется классом `QLineEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QLineEdit
```

Конструктор класса `QLineEdit` имеет два формата:

```
QLineEdit([parent=None])
QLineEdit(<Текст>[, parent=None])
```

В параметре parent указывается ссылка на родительский компонент. Если родитель не указан или имеет значение None, компонент будет обладать своим собственным окном. Параметр <Текст> позволяет задать текст, который будет отображен в поле ввода.

22.5.1. Основные методы и сигналы

Класс QLineEdit поддерживает следующие методы (полный их список смотрите на странице **https://doc.qt.io/qt-6/qlineedit.html**):

- setText(<Текст>) — помещает указанный текст в поле ввода. Метод является слотом;
- insert(<Текст>) — вставляет текст в текущую позицию текстового курсора. Если в поле ввода был выделен фрагмент, он будет удален;
- text() — возвращает текст, содержащийся в поле ввода;
- displayText() — возвращает текст, который видит пользователь. Результат зависит от режима отображения, заданного с помощью метода setEchoMode() (например, в режиме Password строка будет состоять из звездочек);
- clear() — удаляет весь текст из поля. Метод является слотом;
- backspace() — удаляет выделенный фрагмент. Если выделенного фрагмента нет, удаляет символ, стоящий слева от текстового курсора;
- del() — удаляет выделенный фрагмент. Если выделенного фрагмента нет, удаляет символ, стоящий справа от текстового курсора;
- setSelection(<Индекс>, <Длина>) — выделяет фрагмент длиной <Длина>, начиная с позиции <Индекс>. Во втором параметре можно указать отрицательное значение;
- selectedText() — возвращает выделенный фрагмент или пустую строку, если ничего не выделено;
- selectAll() — выделяет весь текст в поле. Метод является слотом;
- selectionStart() — возвращает начальный индекс выделенного фрагмента или значение -1, если ничего не выделено;
- selectionEnd() — возвращает индекс символа, следующего сразу за последним символом выделенного фрагмента, или значение -1, если ничего не выделено;
- hasSelectedText() — возвращает значение True, если поле ввода содержит выделенный фрагмент, и False — в противном случае;
- deselect() — снимает выделение;
- isModified() — возвращает True, если текст в поле был изменен пользователем, и False — в противном случае. Отметьте, что вызов метода setText() помечает поле как неизмененное;
- setModified(<Флаг>) — если передано значение True, поле ввода помечается как измененное, если False — как неизмененное;
- setEchoMode(<Режим>) — задает режим отображения текста. Могут быть указаны следующие элементы перечисления EchoMode из класса QLineEdit:
  - Normal — показывать вводимые символы;
  - NoEcho — не показывать вводимые символы;

- `Password` — вместо символов выводить звездочки (*);
- `PasswordEchoOnEdit` — показывать символы при вводе, а после потери фокуса вместо них отображать звездочки;

- `setCompleter(<Список>)` — задает список вариантов значений для автозавершения. В качестве параметра указывается объект класса `QCompleter`. Пример:

```
lineEdit = QtWidgets.QLineEdit()
arr = ["кадр", "каменный", "камень", "камера"]
completer = QtWidgets.QCompleter(arr, parent=window)
lineEdit.setCompleter(completer)
```

- `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, поле будет доступно только для чтения;
- `isReadOnly()` — возвращает значение `True`, если поле доступно только для чтения, и `False` — в противном случае;
- `setAlignment(<Выравнивание AlignmentFlag>)` — задает выравнивание текста внутри поля;
- `setMaxLength(<Количество>)` — задает максимальную длину вводимого значения в символах;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, поле будет отображаться без рамки;
- `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, режим перетаскивания текста из текстового поля с помощью мыши будет включен. По умолчанию поле ввода только принимает перетаскиваемый текст;
- `setPlaceholderText(<Текст>)` — задает текст подсказки, который будет выводиться непосредственно в поле ввода, когда оно не содержит значения и не имеет фокуса ввода;
- `setTextMargins()` — задает величины отступов от границ компонента до находящегося в нем текста. Форматы метода:

```
setTextMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)
setTextMargins(<Отступы QMargins>)
```

- `setCursorPosition(<Индекс>)` — задает положение текстового курсора;
- `cursorPosition()` — возвращает текущее положение текстового курсора;
- `cursorForward(<Флаг>[, steps=1])` — перемещает текстовый курсор вперед на указанное во втором параметре количество символов. Если в первом параметре указано значение `True`, выполняется выделение фрагмента;
- `cursorBackward(<Флаг>[, steps=1])` — перемещает текстовый курсор назад на указанное во втором параметре количество символов. Если в первом параметре указано значение `True`, выполняется выделение фрагмента;
- `cursorWordForward(<Флаг>)` — перемещает текстовый курсор вперед на одно слово. Если в параметре указано значение `True`, выполняется выделение фрагмента;
- `cursorWordBackward(<Флаг>)` — перемещает текстовый курсор назад на одно слово. Если в параметре указано значение `True`, выполняется выделение фрагмента;
- `home(<Флаг>)` — перемещает текстовый курсор в начало поля. Если в параметре указано значение `True`, выполняется выделение фрагмента;

- `end(<Флаг>)` — перемещает текстовый курсор в конец поля. Если в параметре указано значение `True`, выполняется выделение фрагмента;
- `cut()` — вырезает выделенный текст в буфер обмена при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом;
- `copy()` — копирует выделенный текст в буфер обмена при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом;
- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора при условии, что поле доступно для редактирования. Метод является слотом;
- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `isUndoAvailable()` — возвращает значение `True`, если можно отменить последнюю операцию ввода, и `False` — в противном случае;
- `isRedoAvailable()` — возвращает значение `True`, если можно повторить последнюю отмененную операцию ввода, и `False` — в противном случае;
- `createStandardContextMenu()` — создает стандартное контекстное меню, которое отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню, следует создать класс, производный от `QLineEdit`, и переопределить в нем метод `contextMenuEvent(self, <event>)`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню;
- `setClearButtonEnabled(<Флаг>)` — если передано `True`, в правой части непустого поля будет выводиться кнопка, нажатием которой можно очистить это поле, если `False`, кнопка очистки выводиться не будет.

Класс `QLineEdit` поддерживает следующие сигналы:

- `cursorPositionChanged(<Старая позиция>, <Новая позиция>)` — генерируется при перемещении текстового курсора. Внутри обработчика через первый параметр доступна старая позиция курсора, а через второй параметр — новая позиция. Оба параметра являются целочисленными;
- `editingFinished()` — генерируется при нажатии клавиши <Enter> или потере полем фокуса ввода;
- `inputRejected()` — генерируется, если операция ввода, выполненная пользователем, отвергнута полем ввода. Это может случиться, например, при попытке ввести значение с длиной большей, чем задана вызовом метода `setMaxLength()`;
- `returnPressed()` — генерируется при нажатии клавиши <Enter>;
- `selectionChanged()` — генерируется при изменении выделения;
- `textChanged(<Новый текст>)` — генерируется при изменении текста внутри поля пользователем или программно. Внутри обработчика через параметр доступен новый текст в виде строки;
- `textEdited(<Новый текст>)` — генерируется при изменении текста внутри поля пользователем. При задании текста вызовом метода `setText()` не генерируется. Внутри обработчика через параметр доступен новый текст в виде строки.

22.5.2. Ввод данных по маске

С помощью метода `setInputMask(<Маска>)` можно указать маску, которой должно соответствовать заносимое пользователем значение. Маска указывается в виде строки следующего формата:

```
"<Последовательность символов>[;<Символ-заполнитель>]"
```

В первом параметре указывается комбинация из следующих специальных символов:

- `9` — обязательна цифра от `0` до `9`;
- `0` — разрешена, но не обязательна цифра от `0` до `9`;
- `D` — обязательна цифра от `1` до `9`;
- `d` — разрешена, но не обязательна цифра от `1` до `9`;
- `B` — обязательна цифра `0` или `1`;
- `b` — разрешена, но не обязательна цифра `0` или `1`;
- `H` — обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `h` — разрешен, но не обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `#` — разрешена, но не обязательна цифра, знак «плюс» или «минус»;
- `A` — обязательна буква в любом регистре;
- `a` — разрешена, но не обязательна буква;
- `N` — обязательна буква в любом регистре или цифра от `0` до `9`;
- `n` — разрешена, но не обязательна буква или цифра от `0` до `9`;
- `X` — обязателен любой непробельный символ;
- `x` — разрешен, но не обязателен любой непробельный символ;
- `>` — все последующие буквы переводятся в верхний регистр;
- `<` — все последующие буквы переводятся в нижний регистр;
- `!` — отключает изменение регистра;
- `\` — используется для отмены действия перечисленных ранее спецсимволов.

В необязательном параметре `<Символ-заполнитель>` можно указать символ, который будет отображаться в поле, обозначая место ввода. Если параметр не указан, заполнителем будет служить пробел.

Все остальные символы трактуются как есть.

Примеры:

```
lineEdit.setInputMask("Дата: 99.B9.9999;_")  # Дата: __.__.____
lineEdit.setInputMask("Дата: 99.B9.9999;#")  # Дата: ##.##.####
lineEdit.setInputMask("Дата: 99.B9.9999 г.") # Дата:    .    г.
```

Проверить соответствие введенных данных маске позволяет метод `hasAcceptableInput()`. Если данные соответствуют маске, метод возвращает значение `True`, а в противном случае — `False`.

22.5.3. Контроль ввода с помощью валидаторов

Валидатор — механизм, проверяющий соответствие введенного в поле значения заданным правилам.

Валидатор у поля ввода задается методом `setValidator(<Валидатор>)`. В качестве параметра указывается объект класса, производного от класса `QValidator` из модуля `QtGui`. Существуют следующие стандартные классы валидаторов:

- `QIntValidator` — допускает ввод только целых чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

```
QIntValidator([parent=None])
QIntValidator(<Минимальное значение>, <Максимальное значение>[,
                                      parent=None])
```

  Пример ограничения ввода диапазоном целых чисел от 0 до 100:

```
lineEdit.setValidator(QtGui.QIntValidator(0, 100, parent=window))
```

- `QDoubleValidator` — допускает ввод только вещественных чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

```
QDoubleValidator([parent=None])
QDoubleValidator(<Минимальное значение>, <Максимальное значение>,
                 <Количество цифр после точки>[, parent=None])
```

  Пример ограничения ввода диапазоном вещественных чисел от `0.0` до `100.0` и двумя цифрами после десятичной точки:

```
lineEdit.setValidator(QtGui.QDoubleValidator(0.0, 100.0, 2, parent=window))
```

  Чтобы позволить вводить числа в экспоненциальной форме, необходимо передать в метод `setNotation(<Тип записи>)` элемент `ScientificNotation` перечисления `Notation` из класса `QDoubleValidator`. Если передать методу элемент `StandardNotation` того же перечисления, будет разрешено вводить числа только в десятичной форме. Пример:

```
validator = QtGui.QDoubleValidator(0.0, 100.0, 2, parent=window)
validator.setNotation(QtGui.QDoubleValidator.Notation.StandardNotation)
lineEdit.setValidator(validator)
```

- `QRegularExpressionValidator` — позволяет проверить данные на соответствие заданному регулярному выражению. Форматы конструктора:

```
QRegularExpressionValidator([parent=None])
QRegularExpressionValidator(<Регулярное выражение>[, parent=None])
```

  Регулярное выражение указывается в виде объекта класса `QRegularExpression` из модуля `QtCore`. Пример ввода только цифр от `0` до `9`:

```
validator = QtGui.QRegularExpressionValidator(
                  QtCore.QRegularExpression("[0-9]+"), parent=window)
lineEdit.setValidator(validator)
```

  Обратите внимание, что здесь производится проверка полного соответствия шаблону, поэтому символы `^` и `$` явным образом указывать не нужно.

Проверить корректность введенных данных позволяет метод `hasAcceptableInput()`. Если данные корректны, метод возвращает значение `True`, а в противном случае — `False`.

22.6. Область редактирования

Область редактирования предназначена для ввода и редактирования обычного текста, текста в формате HTML или Markdown. Она изначально поддерживает технологию drag & drop, стандартные комбинации клавиш, работу с буфером обмена и многое другое.

Область редактирования реализуется с помощью класса `QTextEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QTextEdit
```

Конструктор класса `QTextEdit` имеет два формата вызова:

```
QTextEdit([parent=None])
QTextEdit(<Текст>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, компонент будет обладать своим собственным окном. Параметр `<Текст>` позволяет задать текст в формате HTML, который будет отображен в области редактирования.

> ***Примечание***
>
> Если поддержка HTML не нужна, то следует воспользоваться классом `QPlainTextEdit`, который оптимизирован для работы с простым текстом большого объема.

22.6.1. Основные методы и сигналы

Класс `QTextEdit` поддерживает следующие основные методы (полный их список смотрите на странице **https://doc.qt.io/qt-6/qtextedit.html**):

- `setText(<Текст>)` — помещает указанный текст в область редактирования. Текст может быть простым или в формате HTML. Метод является слотом;
- `setPlainText(<Текст>)` — помещает в область редактирования простой текст. Метод является слотом;
- `setHtml(<Текст>)` — помещает в область редактирования текст в формате HTML. Метод является слотом;
- `setMarkdown(<Текст>)` — помещает в область редактирования текст в формате Markdown. Метод является слотом;
- `insertPlainText(<Текст>)` — вставляет простой текст в текущую позицию текстового курсора. Если в области редактирования был выделен фрагмент, он будет удален. Метод является слотом;
- `insertHtml(<Текст>)` — вставляет текст в формате HTML в текущую позицию текстового курсора. Если в области был выделен фрагмент, он будет удален. Метод является слотом;
- `append(<Текст>)` — добавляет новый абзац с указанным текстом в формате HTML в конец области редактирования. Метод является слотом;
- `setDocumentTitle(<Текст>)` — задает текст заголовка документа (в теге `<title>`);
- `documentTitle()` — возвращает текст заголовка (из тега `<title>`);

- `toPlainText()` — возвращает содержимое области редактирования в виде простого текста;
- `toHtml()` — возвращает текст в формате HTML;
- `toMarkdown()` — возвращает текст в формате Markdown;
- `clear()` — удаляет весь текст из области. Метод является слотом;
- `selectAll()` — выделяет весь текст в области. Метод является слотом;
- `zoomIn([range=1])` — увеличивает размер шрифта на заданное в параметре `range` количество пунктов. Метод является слотом;
- `zoomOut([range=1])` — уменьшает размер шрифта на заданное в параметре `range` количество пунктов. Метод является слотом;
- `cut()` — вырезает выделенный текст в буфер обмена при условии, что есть выделенный фрагмент. Метод является слотом;
- `copy()` — копирует выделенный текст в буфер обмена при условии, что есть выделенный фрагмент. Метод является слотом;
- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора при условии, что область доступна для редактирования. Метод является слотом;
- `canPaste()` — возвращает `True`, если из буфера обмена можно вставить текст, и `False` — в противном случае;
- `setAcceptRichText(<Флаг>)` — если в качестве параметра указано значение `True`, в область редактирования можно будет ввести, вставить из буфера обмена или перетащить текст в формате HTML. Значение `False` дает возможность заносить в область лишь обычный текст;
- `acceptRichText()` — возвращает значение `True`, если в область можно занести текст в формате HTML, и `False` — если доступно занесение лишь обычного текста;
- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `setUndoRedoEnabled(<Флаг>)` — если в качестве значения указано значение `True`, операции отмены и повтора действий будут разрешены, а если `False` — то будут запрещены;
- `isUndoRedoEnabled()` — возвращает значение `True`, если операции отмены и повтора действий разрешены, и `False` — если запрещены;
- `createStandardContextMenu([<Координаты QPoint>])` — создает стандартное контекстное меню, которое отображается при щелчке правой кнопкой мыши в области редактирования. В параметре можно указать координаты точки внутри содержимого области редактирования. Чтобы изменить стандартное меню, следует создать класс, производный от `QTextEdit`, и переопределить в нем метод `contextMenuEvent(self, <event>)`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню;
- `ensureCursorVisible()` — прокручивает область таким образом, чтобы текстовый курсор оказался в зоне видимости;
- `find()` — производит поиск фрагмента (по умолчанию в прямом направлении без учета регистра символов) в области редактирования. Если фрагмент найден, то он выделяется

и метод возвращает значение True, в противном случае — значение False. Форматы метода:

```
find(<Искомый текст> | <Регулярное выражение> [, <Режим>])
```

Искомый текст можно указать либо строкой, либо регулярным выражением, представленным объектом класса QRegularExpression из модуля QtCore. В необязательном параметре <Режим> можно указать комбинацию (через оператор |) следующих элементов перечисления FindFlag класса QTextDocument из модуля QtGui:

- FindBackward — поиск в обратном направлении;
- FindCaseSensitively — поиск с учетом регистра символов;
- FindWholeWords — поиск целых слов, а не фрагментов;

♦ print(<Принтер>) — отправляет содержимое текстового поля на заданный принтер. В качестве параметра указывается объект одного из классов, производных от QPagedPaintDevice (модуль QtGui): QPrinter (модуль QtPrintSupport) или QPdfWriter (модуль QtGui). Пример вывода документа в PDF-файл:

```
pdf = QtGui.QPdfWriter("document.pdf")
textEdit.print(pdf)
```

Класс QTextEdit поддерживает следующие сигналы:

♦ copyAvailable(<Флаг>) — генерируется при выделении текста или, наоборот, снятии выделения. Значение параметра True указывает, что фрагмент выделен и его можно скопировать, значение False — обратное;

♦ currentCharFormatChanged(<Формат>) — генерируется при изменении формата текста. Внутри обработчика через параметр доступен новый формат в виде объекта класса QTextCharFormat;

♦ cursorPositionChanged() — генерируется при изменении положения текстового курсора;

♦ redoAvailable(<Флаг>) — генерируется при изменении возможности повторить отмененную операцию ввода. Значение параметра True обозначает возможность повтора отмененной операции, а значение False — невозможность сделать это;

♦ selectionChanged() — генерируется при изменении выделения текста;

♦ textChanged() — генерируется при изменении текста в области редактирования;

♦ undoAvailable(<Флаг>) — генерируется при изменении возможности отменить операцию ввода. Значение параметра True указывает, что операция ввода может быть отменена, значение False говорит об обратном.

22.6.2. Задание параметров области редактирования

Задать другие параметры области редактирования можно вызовами следующих методов класса QTextEdit (полный их список смотрите на странице **https://doc.qt.io/qt-6/qtextedit.html**):

♦ setTextInteractionFlags(<Режим>) — задает режим взаимодействия пользователя с текстом. Можно указать следующие элементы (или их комбинацию через оператор |) перечисления TextInteractionFlag из модуля QtCore.Qt:

- NoTextInteraction — пользователь не может взаимодействовать с текстом;
- TextSelectableByMouse — текст можно выделить мышью;

- TextSelectableByKeyboard — текст можно выделить с помощью клавиатуры. Внутри поля будет отображен текстовый курсор;
- LinksAccessibleByMouse — на гиперссылках, присутствующих в тексте, можно щелкать мышью;
- LinksAccessibleByKeyboard — с гиперссылками, присутствующими в тексте, можно взаимодействовать с клавиатуры: перемещаться между гиперссылками — с помощью клавиши <Tab>, а переходить по гиперссылке — нажав клавишу <Enter>;
- TextEditable — текст можно редактировать;
- TextEditorInteraction — комбинация TextSelectableByMouse | TextSelectableByKeyboard | TextEditable;
- TextBrowserInteraction — комбинация TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard;

♦ setReadOnly(<Флаг>) — если в качестве параметра указано значение True, область редактирования будет доступна только для чтения;

♦ isReadOnly() — возвращает значение True, если область доступна только для чтения, и False — в противном случае;

♦ setLineWrapMode(<Режим>) — задает режим переноса строк. В качестве значения могут быть указаны следующие элементы перечисления LineWrapMode из класса QTextEdit:

- NoWrap — перенос строк не производится;
- WidgetWidth — перенос строк при достижении ими правого края области редактирования;
- FixedPixelWidth — перенос строк при достижении ими фиксированной ширины в пикселах, которую можно задать с помощью метода setLineWrapColumnOrWidth();
- FixedColumnWidth — перенос строк при достижении ими фиксированной ширины в символах, которую можно задать с помощью метода setLineWrapColumnOrWidth();

♦ setLineWrapColumnOrWidth(<Ширина>) — задает фиксированную ширину строк, при достижении которой будет выполняться перенос, в пикселах или символах (в зависимости от заданного режима переноса);

♦ setWordWrapMode(<Режим>) — задает режим переноса по словам. В качестве значения могут быть указаны следующие элементы перечисления WrapMode класса QTextOption из модуля QtGui:

- NoWrap или ManualWrap — перенос по словам не производится;
- WordWrap — перенос строк только по словам;
- WrapAnywhere — перенос строки может быть внутри слова;
- WrapAtWordBoundaryOrAnywhere — перенос по возможности по словам, но может быть выполнен и внутри слова;

♦ setOverwriteMode(<Флаг>) — если в качестве параметра указано значение True, вводимый текст будет замещать ранее введенный. Значение False отключает замещение;

♦ overwriteMode() — возвращает значение True, если вводимый текст замещает ранее введенный, и False — в противном случае;

♦ setAutoFormatting(<Режим>) — задает режим автоматического форматирования. В качестве значения могут быть указаны следующие элементы перечисления AutoFormattingFlag из класса QTextEdit:

- AutoNone — автоматическое форматирование не используется;
- AutoBulletList — автоматическое создание маркированного списка при вводе пользователем в начале строки символа *;
- AutoAll — включить все режимы. По состоянию на сегодня эквивалентно режиму AutoBulletList;

- setCursorWidth(<Толщина>) — задает толщину текстового курсора;
- setTabChangesFocus(<Флаг>) — если параметром передать значение False, то с помощью нажатия клавиши <Tab> можно вставить в область редактирования символ табуляции. Если указано значение True, клавиша <Tab> используется для передачи фокуса следующему компоненту;
- setTabStopDistance(<Ширина>) — задает ширину табуляции в пикселах;
- tabStopDistance() — возвращает ширину табуляции в пикселах.

22.6.3. Указание параметров текста и фона

Для изменения параметров текста и фона предназначены следующие методы класса QTextEdit (полный их список смотрите на странице **https://doc.qt.io/qt-6/qtextedit.html**):

- setCurrentFont(<Шрифт>) — задает текущий шрифт. Метод является слотом. В качестве параметра указывается объект класса QFont из модуля QtGui. Конструктор этого класса имеет следующий формат:

  ```
  <Шрифт> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1][, italic=False])
  ```

  В первом параметре задается название шрифта в виде строки. Необязательный параметр pointSize устанавливает размер шрифта. В параметре weight можно указать степень жирности шрифта в виде числа от 0 до 99 или следующего элемента перечисления Weight из класса QFont: Thin, ExtraLight, Light, Normal, Medium, DemiBold, Bold, ExtraBold или Black. Если в параметре italic указано значение True, шрифт будет курсивным;
- currentFont() — возвращает объект класса QFont с текущими характеристиками шрифта;
- setFontFamily(<Название шрифта>) — задает название текущего шрифта. Метод является слотом;
- fontFamily() — возвращает название текущего шрифта;
- setFontPointSize(<Размер>) — задает размер текущего шрифта в пунктах. Метод является слотом;
- fontPointSize() — возвращает размер текущего шрифта;
- setFontWeight(<Жирность>) — задает жирность текущего шрифта в виде числа от 0 до 99 или одного из элементов перечисления Weight из класса QFont. Метод является слотом;
- fontWeight() — возвращает жирность текущего шрифта;
- setFontItalic(<Флаг>) — если в качестве параметра указано значение True, шрифт будет курсивным. Метод является слотом;
- fontItalic() — возвращает True, если шрифт курсивный, и False — в противном случае;
- setFontUnderline(<Флаг>) — если в качестве параметра указано значение True, текст будет подчеркнутым. Метод является слотом;

- `fontUnderline()` — возвращает `True`, если текст подчеркнутый, и `False` — в противном случае;
- `setTextColor(<Цвет>)` — задает цвет текущего текста. В качестве значения можно указать один из элементов перечисления `GlobalColor` из модуля `QtCore.Qt` (например, `black`, `white` и т. д.) или объект класса `QColor` из модуля `QtGui` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). Метод является слотом;
- `textColor()` — возвращает объект класса `QColor` с цветом текущего текста;
- `setTextBackgroundColor(<Цвет>)` — задает цвет фона. В качестве значения можно указать один из элементов перечисления `GlobalColor` из модуля `QtCore.Qt` (например, `black`, `white` и т. д.) или объект класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). Метод является слотом;
- `textBackgroundColor()` — возвращает объект класса `QColor` с цветом фона;
- `setAlignment(<Выравнивание>)` — задает горизонтальное выравнивание текста внутри абзаца (допустимые значения рассмотрены в *разд. 21.2*). Метод является слотом;
- `alignment()` — возвращает значение выравнивания текста внутри абзаца.

Задать формат символов также можно с помощью класса `QTextCharFormat`, который определен в модуле `QtGui` и поддерживает дополнительные настройки. После создания объекта класса его следует передать в метод `setCurrentCharFormat(<Формат>)`. Получить объект класса с текущими настройками позволяет метод `currentCharFormat()`. За подробной информацией по классу `QTextCharFormat` обращайтесь к странице **https://doc.qt.io/qt-6/qtextcharformat.html**.

22.6.4. Класс *QTextDocument*

Класс `QTextDocument` из модуля `QtGui` представляет документ, который отображается в области редактирования. Получить ссылку на текущий документ позволяет метод `document()` класса `QTextEdit`. Занести в область редактирования новый документ можно с помощью метода `setDocument(<Документ>)`. Иерархия наследования:

```
QObject — QTextDocument
```

Конструктор класса `QTextDocument` имеет два формата:

```
QTextDocument([parent=None])
QTextDocument(<Текст>[, parent=None])
```

В параметре `parent` указывается ссылка на родитель. Параметр `<Текст>` задает простой текст (не в HTML-формате), который будет отображен в области редактирования.

Класс `QTextDocument` поддерживает следующий набор методов (полный их список смотрите на странице **https://doc.qt.io/qt-6/qtextdocument.html**):

- `setPlainText(<Текст>)` — помещает в документ простой текст;
- `setHtml(<Текст>)` — помещает в документ текст в формате HTML;
- `setMarkdown(<Текст>)` — помещает в документ текст в формате Markdown;
- `toPlainText()` — возвращает простой текст, содержащийся в документе;
- `toHtml()` — возвращает текст в формате HTML;
- `toMarkdown()` — возвращает текст в формате Markdown;

- `clear()` — удаляет весь текст из документа;
- `isEmpty()` — возвращает значение `True`, если документ пустой, и `False` — в противном случае;
- `setModified(<Флаг>)` — если передано значение `True`, документ помечается как измененный, если `False` — как неизмененный. Метод является слотом;
- `isModified()` — возвращает значение `True`, если документ был изменен, и `False` — в противном случае;
- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `isUndoAvailable()` — возвращает значение `True`, если можно отменить последнюю операцию ввода, и `False` — в противном случае;
- `isRedoAvailable()` — возвращает значение `True`, если можно повторить последнюю отмененную операцию ввода, и `False` — в противном случае;
- `setUndoRedoEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то операции отмены и повтора действий разрешены, а если `False` — то запрещены;
- `isUndoRedoEnabled()` — возвращает значение `True`, если операции отмены и повтора действий разрешены, и `False` — если запрещены;
- `availableUndoSteps()` — возвращает количество возможных операций отмены;
- `availableRedoSteps()` — возвращает количество возможных повторов отмененных операций;
- `clearUndoRedoStacks([stacks=Stacks.UndoAndRedoStacks])` — очищает список возможных отмен и/или повторов. В качестве параметра можно указать следующие элементы перечисления `Stacks` из класса `QTextDocument`:
  - `UndoStack` — только список возможных отмен;
  - `RedoStack` — только список возможных повторов;
  - `UndoAndRedoStacks` — оба списка;
- `print(<Принтер>)` — отправляет содержимое документа на указанный принтер. В качестве параметра указывается объект одного из классов, производных от `QPagedPaintDevice` (модуль `QtGui`): `QPrinter` (модуль `QtPrintSupport`) или `QPdfWriter` (модуль `QtGui`);
- `find()` — производит поиск фрагмента в документе. Метод возвращает объект класса `QTextCursor` из модуля `QtGui`. Если фрагмент не найден, то возвращенный объект будет нулевым. Проверить успешность операции можно с помощью метода `isNull()` класса `QTextCursor`. Форматы метода:

```
find(<Текст> | <Регулярное выражение> [, position=0][, options=0])
find(<Текст> | <Регулярное выражение>, <Курсор>[, options=0])
```

Параметр `<Текст>` задает искомый фрагмент, а параметр `<Регулярное выражение>` — регулярное выражение в виде объекта класса `QRegularExpression` из модуля `QtCore`. По умолчанию обычный поиск производится без учета регистра символов в прямом направлении, начиная с позиции `position` или от текстового курсора, указанного в параметре `<Курсор>` в виде объекта класса `QTextCursor`. Поиск по регулярному выражению по

умолчанию производится с учетом регистра символов. Чтобы поиск производился без учета регистра, необходимо передать элемент `CaseInsensitiveOption` перечисления `PatternOption` из класса `QRegularExpression` в метод `setPatternOptions()` регулярного выражения. В необязательном параметре `options` можно указать комбинацию (через оператор `|`) следующих элементов перечисления `FindFlag` из класса `QTextDocument`:

- `FindBackward` — поиск в обратном направлении;
- `FindCaseSensitively` — поиск с учетом регистра символов. При использовании регулярного выражения значение игнорируется;
- `FindWholeWords` — поиск целых слов, а не фрагментов;

- `setDefaultFont(<Шрифт>)` — задает шрифт по умолчанию для документа. В качестве параметра указывается объект класса `QFont` из модуля `QtGui` (описан в разд. *22.6.3*);
- `setDefaultStyleSheet(<Таблица стилей>)` — устанавливает для документа таблицу стилей CSS по умолчанию. Таблица стилей задается в виде строки;
- `setDocumentMargin(<Отступ>)` — задает отступ от краев поля до текста;
- `documentMargin()` — возвращает величину отступа от краев поля до текста;
- `setMaximumBlockCount(<Количество>)` — задает максимальное количество текстовых блоков в документе. Если количество блоков становится больше указанного значения, первые блоки будут удалены;
- `maximumBlockCount()` — возвращает максимальное количество текстовых блоков;
- `characterCount()` — возвращает количество символов в документе;
- `lineCount()` — возвращает количество строк в документе;
- `blockCount()` — возвращает количество текстовых блоков в документе;
- `firstBlock()` — возвращает объект класса `QTextBlock`, объявленного в модуле `QtGui`, который содержит первый текстовый блок документа;
- `lastBlock()` — возвращает объект класса `QTextBlock`, который содержит последний текстовый блок документа;
- `findBlock(<Индекс символа>)` — возвращает объект класса `QTextBlock`, который содержит текстовый блок документа, включающий символ с указанным индексом;
- `findBlockByLineNumber(<Индекс абзаца>)` — возвращает объект класса `QTextBlock`, который содержит текстовый блок документа, включающий абзац с указанным индексом;
- `findBlockByNumber(<Индекс блока>)` — возвращает объект класса `QTextBlock`, который содержит текстовый блок документа с указанным индексом.

Класс `QTextDocument` поддерживает сигналы:

- `blockCountChanged(<Новое количество блоков>)` — генерируется при изменении количества текстовых блоков. Внутри обработчика через параметр доступно новое количество текстовых блоков, заданное целым числом;
- `contentsChange(<Позиция курсора>, <Количество удаленных символов>, <Количество добавленных символов>)` — генерируется при изменении текста. Все три параметра целочисленные;
- `contentsChanged()` — генерируется при любом изменении документа;
- `cursorPositionChanged(<Курсор QTextCursor>)` — генерируется при изменении позиции текстового курсора из-за операции редактирования. При простом перемещении тексто-

вого курсора сигнал не генерируется. Внутри обработчика через параметр доступен объект обновленного курсора;

- `modificationChanged(<Флаг>)` — генерируется при изменении состояния документа: из неизмененного в измененное или наоборот. Значение параметра `True` обозначает, что документ помечен как измененный, значение `False` — что он теперь неизмененный;
- `redoAvailable(<Флаг>)` — генерируется при изменении возможности повторить отмененную операцию ввода. Значение параметра `True` обозначает наличие возможности повторить отмененную операцию ввода, а `False` — отсутствие такой возможности;
- `undoAvailable(<Флаг>)` — генерируется при изменении возможности отменить операцию ввода. Значение параметра `True` обозначает наличие возможности отменить операцию ввода, а `False` — отсутствие такой возможности;
- `undoCommandAdded()` — генерируется при добавлении операции ввода в список возможных отмен.

22.6.5. Класс *QTextCursor*

Класс `QTextCursor` из модуля `QtGui` предоставляет текстовый курсор — инструмент для работы с документом, представленным объектом класса `QTextDocument`. Конструктор класса `QTextCursor` поддерживает следующие форматы:

```
QTextCursor()
QTextCursor(<Объект класса QTextDocument>)
QTextCursor(<Объект класса QTextFrame>)
QTextCursor(<Объект класса QTextBlock>)
QTextCursor(<Объект класса QTextCursor>)
```

Создать текстовый курсор, установить его в документе и управлять им позволяют следующие методы класса `QTextEdit`:

- `textCursor()` — возвращает видимый в текущий момент текстовый курсор в виде объекта класса `QTextCursor`;
- `setTextCursor(<Курсор QTextCursor>)` — устанавливает новый текстовый курсор;
- `cursorForPosition(<Позиция>)` — возвращает текстовый курсор, который соответствует позиции, указанной в качестве параметра. Позиция задается в виде объекта класса `QPoint` в координатах области редактирования;
- `moveCursor(<Позиция>[, mode=MoveMode.MoveAnchor])` — перемещает текстовый курсор внутри документа. В первом параметре можно указать следующие элементы перечисления `MoveOperation` из класса `QTextCursor`:
  - `NoMove` — не перемещать курсор;
  - `Start` — в начало документа;
  - `Up` — на одну строку вверх;
  - `StartOfLine` — в начало текущей строки;
  - `StartOfBlock` — в начало текущего текстового блока;
  - `StartOfWord` — в начало текущего слова;
  - `PreviousBlock` — в начало предыдущего текстового блока;
  - `PreviousCharacter` — на предыдущий символ;

- PreviousWord — в начало предыдущего слова;
- Left — на один символ влево;
- WordLeft — на одно слово влево;
- End — в конец документа;
- Down — на одну строку вниз;
- EndOfLine — в конец текущей строки;
- EndOfWord — в конец текущего слова;
- EndOfBlock — в конец текущего текстового блока;
- NextBlock — в начало следующего текстового блока;
- NextCharacter — на следующий символ;
- NextWord — в начало следующего слова;
- Right — сдвинуть на один символ вправо;
- WordRight — в начало следующего слова.

Помимо указанных, в перечислении существуют также элементы NextCell, PreviousCell, NextRow и PreviousRow, позволяющие перемещать текстовый курсор внутри таблицы.

В параметре mode можно указать следующие элементы перечисления MoveMode из класса QTextCursor:

- MoveAnchor — если существует выделенный фрагмент, то выделение будет снято и текстовый курсор переместится в новое место (значение по умолчанию);
- KeepAnchor — фрагмент текста от старой позиции курсора до новой будет выделен.

Класс QTextCursor поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qtextcursor.html**):

- isNull() — возвращает значение True, если объект курсора является нулевым (создан вызовом конструктора без параметра), и False — в противном случае;
- setPosition(<Позиция>[, mode=MoveMode.MoveAnchor]) — перемещает текстовый курсор внутри документа. В первом параметре указывается позиция внутри документа. Необязательный параметр mode аналогичен одноименному параметру метода moveCursor() класса QTextEdit;
- movePosition(<Позиция>[, mode=MoveMode.MoveAnchor][, n=1]) — перемещает текстовый курсор внутри документа. Параметры <Позиция> и mode аналогичны одноименным параметрам метода moveCursor() класса QTextEdit. Необязательный параметр n позволяет указать количество перемещений — например, переместить курсор на 10 символов вперед можно так:

```
cur = textEdit.textCursor()
cur.movePosition(QtGui.QTextCursor.MoveOperation.NextCharacter,
                 mode=QtGui.QTextCursor.MoveMode.MoveAnchor, n=10)
textEdit.setTextCursor(cur)
```

Метод возвращает значение True, если операция успешно выполнена указанное количество раз. Если было выполнено меньшее количество перемещений (например, из-за достижения конца документа), метод возвращает значение False;

- `position()` — возвращает позицию текстового курсора внутри документа;
- `positionInBlock()` — возвращает позицию текстового курсора внутри блока;
- `block()` — возвращает объект класса `QTextBlock`, который описывает текстовый блок, содержащий курсор;
- `blockNumber()` — возвращает номер текстового блока, содержащего курсор;
- `atStart()` — возвращает значение `True`, если текстовый курсор находится в начале документа, и `False` — в противном случае;
- `atEnd()` — возвращает значение `True`, если текстовый курсор находится в конце документа, и `False` — в противном случае;
- `atBlockStart()` — возвращает значение `True`, если текстовый курсор находится в начале блока, и `False` — в противном случае;
- `atBlockEnd()` — возвращает значение `True`, если текстовый курсор находится в конце блока, и `False` — в противном случае;
- `select(<Режим>)` — выделяет фрагмент в документе в соответствии с указанным режимом. В качестве параметра можно указать следующие элементы перечисления `SelectionType` из класса `QTextCursor`:
  - `WordUnderCursor` — выделяет слово, в котором расположен курсор;
  - `LineUnderCursor` — выделяет строку, в которой расположен курсор;
  - `BlockUnderCursor` — выделяет текстовый блок, в котором находится курсор;
  - `Document` — выделяет весь документ;
- `hasSelection()` — возвращает значение `True`, если существует выделенный фрагмент, и `False` — в противном случае;
- `hasComplexSelection()` — возвращает значение `True`, если выделенный фрагмент содержит сложное форматирование, а не просто текст, и `False` — в противном случае;
- `clearSelection()` — снимает выделение;
- `selectionStart()` — возвращает начальную позицию выделенного фрагмента;
- `selectionEnd()` — возвращает конечную позицию выделенного фрагмента;
- `selectedText()` — возвращает текст выделенного фрагмента;

> ***Внимание!***
>
> Если выделенный фрагмент занимает несколько строк, то вместо символа перевода строки вставляется символ с кодом `\u2029`. Попытка вывести этот символ в консоли приведет к исключению, поэтому следует произвести замену символа с помощью метода `replace()`:
>
> ```
> print(cur.selectedText().replace("\u2029", "\n"))
> ```

- `selection()` — возвращает объект класса `QTextDocumentFragment`, который описывает выделенный фрагмент. Получить текст позволяют методы `toPlainText()` (возвращает простой текст) и `toHtml()` (возвращает текст в формате HTML) этого класса;
- `removeSelectedText()` — удаляет выделенный фрагмент;
- `deleteChar()` — если нет выделенного фрагмента, удаляет символ справа от курсора, в противном случае удаляет выделенный фрагмент;

- `deletePreviousChar()` — если нет выделенного фрагмента, удаляет символ слева от курсора, в противном случае удаляет выделенный фрагмент;
- `beginEditBlock()` — задает начало блока операций ввода. Операции, входящие в такой блок, могут быть отменены или повторены как единое целое с помощью методов `undo()` и `redo()`;
- `endEditBlock()` — задает конец блока операций ввода;
- `joinPreviousEditBlock()` — делает последующие операции ввода частью предыдущего блока операций;
- `setKeepPositionOnInsert(<Флаг>)` — если в качестве параметра указано значение `True`, то после операции вставки курсор сохранит свою предыдущую позицию. По умолчанию позиция курсора при вставке изменяется;
- `insertText(<Текст>[, <Формат>])` — вставляет простой текст с форматом, указанным в виде объекта класса `QTextFormat`;
- `insertHtml(<Текст>)` — вставляет текст в формате HTML.

С помощью методов `insertBlock()`, `insertFragment()`, `insertFrame()`, `insertImage()`, `insertList()` и `insertTable()` можно вставить различные элементы: изображения, списки и др. Изменить формат выделенного фрагмента позволяют методы `mergeBlockCharFormat()`, `mergeBlockFormat()` и `mergeCharFormat()`. За подробной информацией по этим методам обращайтесь к странице документации **https://doc.qt.io/qt-6/qtextcursor.html**.

22.7. Текстовый браузер

Класс `QTextBrowser` расширяет возможности класса `QTextEdit` и реализует текстовый браузер с возможностью перехода по гиперссылкам. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QTextEdit — QTextBrowser
```

Формат конструктора класса `QTextBrowser`:

```
QTextBrowser([parent=None])
```

Класс `QTextBrowser` поддерживает следующие основные методы (полный их список смотрите на странице **https://doc.qt.io/qt-6/qtextbrowser.html**):

- `setSource()` — загружает ресурс. Формат метода:

  ```
  setSource(<Интернет-адрес>[, type=ResourceType.UnknownResource])
  ```

  Интернет-адрес указывается в виде объекта класса `QUrl` из класса `QtCore`. Параметр `type` задает тип открываемого ресурса в виде одного из следующих элементов перечисления `ResourceType` из класса `QTextDocument`:

  - `UnknownResource` — неизвестный ресурс или ресурс вообще не загружен;
  - `HtmlResource` — HTML-документ;
  - `ImageResource` — графическое изображение;
  - `StyleSheetResource` — таблица стилей CSS;
  - `MarkdownResource` — документ в формате Markdown.

Пример:

```
# Загружаем и выводим содержимое текстового файла
url = QtCore.QUrl("text.txt")
browser.setSource(url)
```

Метод является слотом;

- `source()` — возвращает объект класса `QUrl` с адресом текущего ресурса;
- `reload()` — перезагружает текущий ресурс. Метод является слотом;
- `home()` — загружает первый ресурс из списка истории. Метод является слотом;
- `backward()` — загружает предыдущий ресурс из списка истории. Метод является слотом;
- `forward()` — загружает следующий ресурс из списка истории. Метод является слотом;
- `backwardHistoryCount()` — возвращает количество предыдущих ресурсов из списка истории;
- `forwardHistoryCount()` — возвращает количество следующих ресурсов из списка истории;
- `isBackwardAvailable()` — возвращает значение `True`, если существует предыдущий ресурс в списке истории, и `False` — в противном случае;
- `isForwardAvailable()` — возвращает значение `True`, если существует следующий ресурс в списке истории, и `False` — в противном случае;
- `clearHistory()` — очищает список истории;
- `historyTitle(<Количество позиций>)` — если в качестве параметра указано отрицательное число, возвращает заголовок предыдущего ресурса, отстоящего от текущего на заданное число позиций, если 0 — заголовок текущего ресурса, а если положительное число — заголовок следующего ресурса, также отстоящего от текущего на заданное число позиций;
- `historyUrl(<Количество позиций>)` — то же самое, что `historyTitle()`, но возвращает адрес ресурса в виде объекта класса `QUrl`;
- `setOpenLinks(<Флаг>)` — если в качестве параметра указано значение `True`, то переход по гиперссылкам будет разрешен (поведение по умолчанию). Значение `False` запрещает переход;
- `setOpenExternalLinks(<Флаг>)` — если в качестве параметра указано значение `True`, то переход по гиперссылкам, ведущим на внешние ресурсы, будет разрешен (при этом сигнал `anchorClicked()` не генерируется). Значение `False` запрещает переход (поведение по умолчанию).

Класс `QTextBrowser` поддерживает сигналы:

- `anchorClicked(<Интернет-адрес>)` — генерируется при переходе по гиперссылке. Внутри обработчика через параметр доступен интернет-адрес гиперссылки в виде объекта класса `QUrl`;
- `backwardAvailable(<Признак>)` — генерируется при изменении списка предыдущих ресурсов. Внутри обработчика через параметр доступно значение `True`, если в списке истории имеются предыдущие ресурсы, и `False` — в противном случае;
- `forwardAvailable(<Признак>)` — генерируется при изменении списка следующих ресурсов. В обработчике через параметр доступно значение `True`, если в списке истории имеются следующие ресурсы, и `False` — в противном случае;

- `highlighted(<Интернет-адрес>)` — генерируется при наведении указателя мыши на гиперссылку. Внутри обработчика через параметр доступен интернет-адрес гиперссылки в виде объекта класса `QUrl` или пустой объект;
- `historyChanged()` — генерируется при изменении списка истории;
- `sourceChanged(<Интернет-адрес>)` — генерируется при переходе на новый ресурс. Внутри обработчика через параметр доступен интернет-адрес загруженного ресурса в виде объекта класса `QUrl`.

22.8. Поля для ввода целых и вещественных чисел

Для ввода целых чисел предназначен класс `QSpinBox`, для ввода вещественных чисел — класс `QDoubleSpinBox`. Эти поля могут содержать две кнопки, которые позволяют щелчками мыши увеличивать и уменьшать значение внутри поля. Пример такого поля ввода можно увидеть на рис. 22.1. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QSpinBox
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDoubleSpinBox
```

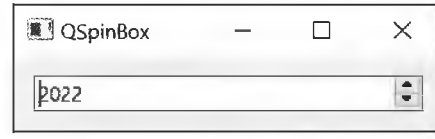


Рис. 22.1. Компонент `QSpinBox`

Форматы конструкторов классов `QSpinBox` и `QDoubleSpinBox`:

```
QSpinBox([parent=None])
QDoubleSpinBox([parent=None])
```

Классы `QSpinBox` и `QDoubleSpinBox` наследуют следующие методы из класса `QAbstractSpinBox` (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qabstractspinbox.html**):

- `setButtonSymbols(<Режим>)` — задает режим отображения кнопок, предназначенных для изменения значения поля. Можно указать следующие элементы перечисления `ButtonSymbols` из класса `QAbstractSpinBox`:
  - `UpDownArrows` — отображаются кнопки со стрелками;
  - `PlusMinus` — отображаются кнопки с символами + и -. Обратите внимание, что при использовании некоторых стилей это значение может быть проигнорировано;
  - `NoButtons` — кнопки не отображаются;
- `setCorrectionMode(<Режим>)` — задает режим преобразования введенных вручную недопустимых значений в допустимые в виде одного из следующих элементов перечисления `CorrectionMode` из класса `QAbstractSpinBox`:
  - `CorrectToPreviousValue` — преобразовывать в предыдущее допустимое значение;
  - `CorrectToNearestValue` — преобразовывать в ближайшее допустимое значение;

- `setAccelerated(<Флаг>)` — если в качестве параметра указано значение `True`, то при удержании какой-либо из кнопок нажатой скорость смены значений в поле увеличится;
- `setAlignment(<Режим>)` — задает режим выравнивания значения внутри поля (допустимые значения рассмотрены в *разд. 21.2*);
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `True`, то значение внутри поля будет при нажатии кнопок изменяться по кругу: максимальное значение сменится минимальным и наоборот;
- `setSpecialValueText(<Строка>)` — задает строку, которая будет отображаться внутри поля вместо минимального значения;
- `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, поле будет доступно только для чтения;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, поле будет отображаться без рамки;
- `stepDown()` — уменьшает значение на одно приращение. Метод является слотом;
- `stepUp()` — увеличивает значение на одно приращение. Метод является слотом;
- `stepBy(<Количество>)` — увеличивает (при положительном значении) или уменьшает (при отрицательном значении) значение поля на указанное количество приращений;
- `text()` — возвращает текст, содержащийся внутри поля;
- `clear()` — очищает поле. Метод является слотом;
- `selectAll()` — выделяет все содержимое поля. Метод является слотом.

Класс `QAbstractSpinBox` поддерживает сигнал `editingFinished()`, который генерируется при потере полем фокуса ввода или при нажатии клавиши <Enter>.

Классы `QSpinBox` и `QDoubleSpinBox` поддерживают следующие методы (здесь приведены только основные — полные их списки доступны на страницах **https://doc.qt.io/qt-6/qspinbox.html** и **https://doc.qt.io/qt-6/qdoublespinbox.html** соответственно):

- `setValue(<Число>)` — задает значение поля. Метод является слотом, принимающим, в зависимости от компонента, целое или вещественное значение;
- `value()` — возвращает целое или вещественное число, содержащееся в поле;
- `cleanText()` — возвращает целое или вещественное число в виде строки, без дополнительного текста, заданного методами `setPrefix()` и `setSuffix()`, начальных и конечных пробелов;
- `setRange(<Минимум>, <Максимум>)` — задает минимальное и максимальное допустимые значения;
- `setMinimum(<Минимум>)` — задает минимальное значение;
- `setMaximum(<Максимум>)` — задает максимальное значение;
- `setPrefix(<Текст>)` — задает текст, который будет отображаться внутри поля перед значением;
- `setSuffix(<Текст>)` — задает текст, который будет отображаться внутри поля после значения;
- `setSingleStep(<Число>)` — задает число, которое будет прибавляться или вычитаться из текущего значения поля на каждом шаге.

Класс QDoubleSpinBox также поддерживает метод setDecimals(<Количество>), который задает количество цифр после десятичной точки.

Классы QSpinBox и QDoubleSpinBox поддерживают сигналы valueChanged(<Целое число>) (в классе QSpinBox), valueChanged(<Вещественное число>) (в классе QDoubleSpinBox) и textChanged(<Строка>), которые генерируются при изменении значения внутри поля. Внутри обработчика через параметр доступно новое значение в виде числа или строки.

22.9. Поля для ввода даты и времени

Для ввода даты и времени предназначены классы QDateTimeEdit (ввод временной отметки, т. е. значения даты и времени), QDateEdit (ввод даты) и QTimeEdit (ввод времени). Поля могут содержать кнопки, которые позволяют щелчками мыши увеличивать и уменьшать значение внутри поля. Пример такого поля показан на рис. 22.2.

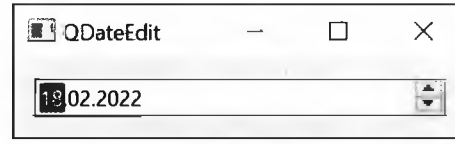


Рис. 22.2. Компонент QDateEdit с кнопками-стрелками

Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit — QDateEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit — QTimeEdit
```

Форматы конструкторов классов:

```
QDateTimeEdit([parent=None])
QDateTimeEdit(<Временная отметка>[, parent=None])
QDateTimeEdit(<Дата>[, parent=None])
QDateTimeEdit(<Время>[, parent=None])
QDateEdit([parent=None])
QDateEdit(<Дата>[, parent=None])
QTimeEdit([parent=None])
QTimeEdit(<Время>[, parent=None])
```

В параметре <Временная отметка> можно указать объект класса QDateTime или datetime (из Python). Преобразовать объект класса QDateTime в объект класса datetime позволяет метод toPyDateTime() класса QDateTime:

```
>>> from PyQt6 import QtCore
>>> d = QtCore.QDateTime(2022, 2, 18, 14, 23)
>>> d
    PyQt6.QtCore.QDateTime(2022, 2, 18, 14, 23)
>>> d.toPyDateTime()
    datetime.datetime(2022, 2, 18, 14, 23)
```

В качестве параметра <Дата> можно указать объект класса QDate или date (из Python). Преобразовать объект класса QDate в объект класса date позволяет метод toPyDate() класса QDate.

В параметре <Время> можно указать объект класса QTime или time (из Python). Преобразовать объект класса QTime в объект класса time позволяет метод toPyTime() класса QTime.

Классы QDateTime, QDate и QTime определены в модуле QtCore.

Класс QDateTimeEdit наследует все методы из класса QAbstractSpinBox (см. *разд. 22.8*) и дополнительно реализует следующие методы (здесь приведены только самые полезные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qdatetimeedit.html**):

- setDateTime(<Временна́я отметка QDateTime или datetime>) — устанавливает временну́ю отметку. Метод является слотом;
- setDate(<Дата QDate или date>) — устанавливает дату. Метод является слотом;
- setTime(<Время QTime или time>) — устанавливает время. Метод является слотом;
- dateTime() — возвращает объект класса QDateTime с временно́й отметкой;
- date() — возвращает объект класса QDate с датой;
- time() — возвращает объект класса QTime со временем;
- setDateTimeRange(<Минимум>, <Максимум>) — задает минимальное и максимальное допустимые значения для временно́й отметки;
- setMinimumDateTime(<Минимум>) — задает минимальное допустимое значение временно́й отметки;
- setMaximumDateTime(<Максимум>) — задает максимальное допустимое значение временно́й отметки.

  В параметрах трех последних методов указываются объекты класса QDateTime или datetime;
- setDateRange(<Минимум>, <Максимум>) — задают минимальное и максимальное допустимые значения для даты;
- setMinimumDate(<Минимум>) — задает минимальное допустимое значение даты;
- setMaximumDate(<Максимум>) — задает максимальное допустимое значение даты.

  В параметрах трех последних методов указываются объекты класса QDate или date;
- setTimeRange(<Минимум>, <Максимум>) — задает минимальное и максимальное допустимые значения для времени;
- setMinimumTime(<Минимум>) — задает минимальное допустимое значение времени;
- setMaximumTime(<Максимум>) — задает максимальное допустимое значение времени.

  В параметрах трех последних методов указываются объекты класса QTime или time;
- setDisplayFormat(<Формат>) — задает формат отображения даты и времени. В качестве параметра указывается строка, содержащая специальные символы. Пример задания строки формата:

  ```
  dateTimeEdit.setDisplayFormat("dd.MM.yyyy HH:mm:ss")
  ```
- setTimeSpec(<Зона>) — задает временну́ю зону. В качестве параметра можно указать следующие элементы перечисления TimeSpec из модуля QtCore.Qt: LocalTime (местное время), UTC (всемирное координированное время) или OffsetFromUTC (смещение относительно всемирного координированного времени, исчисляемое в секундах);

- `setCalendarPopup(<Флаг>)` — если в качестве параметра указано значение `True`, то дату можно будет выбрать с помощью календаря, который появится на экране при щелчке на кнопке с направленной вниз стрелкой, выведенной вместо стандартных кнопок-стрелок (рис. 22.3);

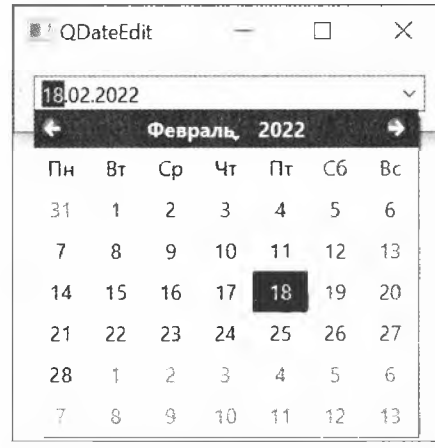


Рис. 22.3. Компонент `QDateEdit` с открытым календарем

- `setSelectedSection(<Секция>)` — выделяет указанную секцию. В качестве параметра можно задать один из следующих элементов перечисления `Section` из класса `QDateTimeEdit`:
  - `NoSection` — ни одна секция не будет выделена;
  - `DaySection` — будет выделена секция числа;
  - `MonthSection` — секция месяца;
  - `YearSection` — секция года;
  - `HourSection` — секция часов;
  - `MinuteSection` — секция минут;
  - `SecondSection` — секция секунд;
  - `MSecSection` — секция миллисекунд;
  - `AmPmSection` — секция времени суток (AM или PM);
- `setCurrentSection(<Секция Section>)` — делает указанную секцию текущей;
- `setCurrentSectionIndex(<Индекс>)` — делает секцию с указанным индексом текущей;
- `currentSection()` — возвращает тип текущей секции в виде одного из элементов перечисления `Section` из класса `QDateTimeEdit`;
- `currentSectionIndex()` — возвращает индекс текущей секции;
- `sectionCount()` — возвращает количество секций внутри поля;
- `sectionAt(<Индекс>)` — возвращает обозначение типа секции по указанному индексу в виде одного из элементов перечисления `Section` из класса `QDateTimeEdit`;
- `sectionText(<Секция Section>)` — возвращает текст указанной секции.

При изменении значений даты или времени генерируются сигналы `timeChanged(<Время>)`, `dateChanged(<Дата>)` и `dateTimeChanged(<Временная отметка>)`. Внутри обработчиков через параметр доступно новое значение в виде объекта класса `QTime`, `QDate` или `QDateTime` соответственно.

Классы `QDateEdit` и `QTimeEdit` отличаются от класса `QDateTimeEdit` только форматом отображаемых данных. Эти классы наследуют методы базовых классов и не добавляют никаких своих методов.

22.10. Календарь

Класс `QCalendarWidget` реализует календарь с возможностью выбора даты и перемещения по месяцам с помощью мыши и клавиатуры (рис. 22.4). Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QCalendarWidget
```

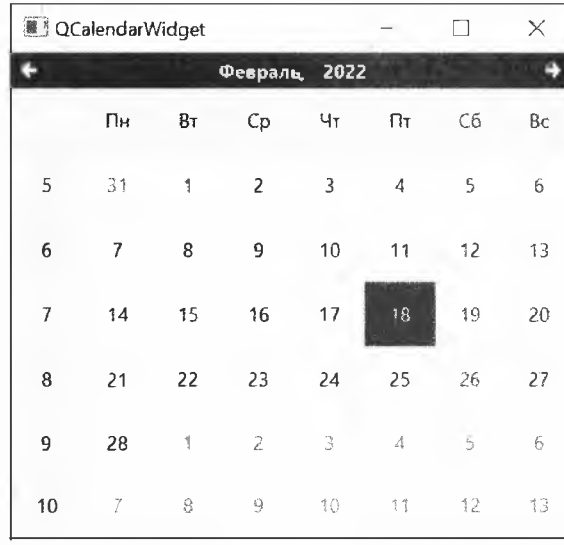


Рис. 22.4. Компонент `QCalendarWidget`

Формат конструктора класса `QCalendarWidget`:

```
QCalendarWidget([parent=None])
```

Класс `QCalendarWidget` поддерживает следующие методы (здесь представлены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qcalendarwidget.html**):

- `setSelectedDate(<Дата QDate или date>)` — устанавливает дату. Метод является слотом;
- `selectedDate()` — возвращает объект класса `QDate` с выбранной датой;
- `setDateRange(<Минимум>, <Максимум>)` — задает минимальное и максимальное допустимые значения для даты. Метод является слотом;
- `setMinimumDate(<Минимум>)` — задает минимальное допустимое значение даты;
- `setMaximumDate(<Максимум>)` — задает максимальное допустимое значение даты.

  В параметрах трех последних методов указывается объект класса `QDate` или `date`;

- `setCurrentPage(<Год>, <Месяц>)` — делает текущей страницу календаря с указанными годом и месяцем, которые задаются целыми числами. Выбранная дата при этом не изменяется. Метод является слотом;
- `monthShown()` — возвращает месяц (число от 1 до 12), отображаемый на текущей странице;
- `yearShown()` — возвращает год, отображаемый на текущей странице;
- `showSelectedDate()` — отображает страницу с выбранной датой. Выбранная дата при этом не изменяется. Метод является слотом;
- `showToday()` — отображает страницу с сегодняшней датой. Выбранная дата при этом не изменяется. Метод является слотом;
- `showPreviousMonth()` — отображает страницу с предыдущим месяцем. Выбранная дата при этом не изменяется. Метод является слотом;
- `showNextMonth()` — отображает страницу со следующим месяцем. Выбранная дата при этом не изменяется. Метод является слотом;
- `showPreviousYear()` — отображает страницу с текущим месяцем в предыдущем году. Выбранная дата не изменяется. Метод является слотом;
- `showNextYear()` — отображает страницу с текущим месяцем в следующем году. Выбранная дата при этом не изменяется. Метод является слотом;
- `setFirstDayOfWeek(<День>)` — задает первый день недели. По умолчанию используется воскресенье. Чтобы первым днем недели сделать понедельник, следует в качестве параметра указать элемент `Monday` перечисления `DayOfWeek` из класса `QtCore.Qt`;
- `setNavigationBarVisible(<Флаг>)` — если в качестве параметра указано значение `False`, то панель навигации выводиться не будет. Метод является слотом;
- `setHorizontalHeaderFormat(<Формат>)` — задает формат горизонтального заголовка. В качестве параметра можно указать следующие элементы перечисления `HorizontalHeaderFormat` класса `QCalendarWidget`:
  - `NoHorizontalHeader` — заголовок не отображается;
  - `SingleLetterDayNames` — отображается только первая буква из названия дня недели;
  - `ShortDayNames` — отображается сокращенное название дня недели;
  - `LongDayNames` — отображается полное название дня недели;
- `setVerticalHeaderFormat(<Формат>)` — задает формат вертикального заголовка. В качестве параметра можно указать следующие элементы перечисления `VerticalHeaderFormat` из класса `QCalendarWidget`:
  - `NoVerticalHeader` — заголовок не отображается;
  - `ISOWeekNumbers` — отображается номер недели в году;
- `setGridVisible(<Флаг>)` — если в качестве параметра указано значение `True`, линии сетки будут отображены. Метод является слотом;
- `setSelectionMode(<Режим>)` — задает режим выделения даты. В качестве параметра можно указать следующие элементы перечисления `SelectionMode` из класса `QCalendarWidget`:
  - `NoSelection` — дата не может быть выбрана пользователем;
  - `SingleSelection` — может быть выбрана одна дата;

- `setHeaderTextFormat(<Формат>)` — задает формат ячеек заголовка. В параметре указывается объект класса `QTextCharFormat` из модуля `QtGui`;
- `setWeekdayTextFormat(<День недели>, <Формат>)` — определяет формат ячеек для указанного дня недели. В первом параметре задаются следующие элементы перечисления `DayOfWeek` из модуля `QCore.Qt`: `Monday` (понедельник), `Tuesday` (вторник), `Wednesday` (среда), `Thursday` (четверг), `Friday` (пятница), `Saturday` (суббота) или `Sunday` (воскресенье), а во втором параметре — объект класса `QTextCharFormat`;
- `setDateTextFormat(<Дата QDate или date>, <Формат QTextCharFormat>)` — задает формат ячейки с указанной датой.

Класс `QCalendarWidget` поддерживает такие сигналы:

- `activated(<Дата QDate>)` — генерируется при двойном щелчке мышью или нажатии клавиши <Enter>. Внутри обработчика через параметр доступна текущая дата;
- `clicked(<Дата QDate>)` — генерируется при щелчке мышью на какой-либо дате. Внутри обработчика через параметр доступна выбранная дата;
- `currentPageChanged(<Год>, <Месяц>)` — генерируется при изменении страницы. Внутри обработчика через первый параметр доступен год, а через второй — месяц. Обе величины задаются целыми числами;
- `selectionChanged()` — генерируется при изменении выбранной даты пользователем или из программного кода.

22.11. Семисегментный индикатор

Класс `QLCDNumber` реализует семисегментный индикатор, аналогичный используемым в электронных часах и калькуляторах (рис. 22.5). Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QLCDNumber
```

Рис. 22.5. Компонент `QLCDNumber`

Форматы конструктора класса `QLCDNumber`:

```
QLCDNumber([parent=None])
QLCDNumber(<Количество цифр>[, parent=None])
```

Если количество отображаемых цифр не указано, используется значение 5.

Класс `QLCDNumber` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qlcdnumber.html**):

- `display(<Значение>)` — задает новое выводимое значение: целое число, вещественное число или строку:

  ```
  lcd.display(1054871)
  ```

  Метод является слотом;

- `checkOverflow(<Число>)` — возвращает значение `True`, если целое или вещественное число, указанное в параметре, не может быть отображено индикатором. В противном случае возвращает значение `False`;
- `intValue()` — возвращает выводимое значение в виде целого числа;
- `value()` — возвращает выводимое значение в виде вещественного числа;
- `setSegmentStyle(<Стиль>)` — задает стиль индикатора. В качестве параметра можно указать следующие элементы перечисления `SegmentStyle` из класса `QLCDNumber`: `Outline` (контурные сегменты), `Filled` (выпуклые сплошные сегменты) или `Flat` (плоские сплошные сегменты);
- `setMode(<Режим>)` — задает режим отображения чисел. В качестве параметра можно указать следующие элементы перечисления `Mode` из класса `QLCDNumber`: `Hex` (шестнадцатеричный режим), `Dec` (десятичный), `Oct` (восьмеричный) или `Bin` (двоичный).

  Вместо метода `setMode()` удобнее воспользоваться методами-слотами `setHexMode()`, `setDecMode()`, `setOctMode()` и `setBinMode()`;
- `setSmallDecimalPoint(<Флаг>)` — если в качестве параметра указано значение `True`, десятичная точка будет отображаться как отдельный элемент (при этом значение выводится более компактно — без пробелов до и после точки), а если значение `False` — то десятичная точка будет занимать позицию цифры (поведение по умолчанию). Метод является слотом;
- `setDigitCount(<Число>)` — задает количество отображаемых цифр. Если в методе `setSmallDecimalPoint()` указано значение `False`, десятичная точка считается отдельной цифрой.

Класс `QLCDNumber` поддерживает сигнал `overflow()`, генерируемый при попытке задать значение, которое не может быть отображено индикатором.

22.12. Индикатор процесса

Класс `QProgressBar` реализует индикатор процесса. Иерархия наследования выглядит так:

```
QObject, QPaintDevice) — QWidget — QProgressBar
```

Формат конструктора класса `QProgressBar`:

```
QProgressBar([parent=None])
```

Класс `QProgressBar` поддерживает следующий набор методов, которые могут быть нам полезны (полный их список смотрите на странице **https://doc.qt.io/qt-6/qprogressbar.html**):

- `setValue(<Значение>)` — задает новое целочисленное значение процесса. Метод является слотом;
- `value()` — возвращает текущее значение процесса в виде числа;
- `text()` — возвращает текст, отображаемый на индикаторе или рядом с ним;
- `setRange(<Минимум>, <Максимум>)` — задает минимальное и максимальное значения;
- `setMinimum(<Минимум>)` — задает минимальное значение;
- `setMaximum(<Максимум>)` — задает максимальное значение.

  Параметры всех трех последних методов задаются в виде целых чисел. Если и минимальное, и максимальное значения равны нулю, то внутри индикатора будут постоянно

по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Все три метода являются слотами;

- `reset()` — сбрасывает значение индикатора. Метод является слотом;
- `setOrientation(<Ориентация>)` — задает ориентацию индикатора. В качестве значения указываются элементы `Horizontal` (горизонтальная) или `Vertical` (вертикальная) перечисления `Orientation` из модуля `QtCore.Qt`. Метод является слотом;
- `setTextVisible(<Флаг>)` — если в качестве параметра указано значение `False`, текст с текущим значением индикатора отображаться не будет;
- `setTextDirection(<Направление>)` — задает направление вывода текста при вертикальной ориентации индикатора. В качестве значения указываются следующие элементы перечисления `Direction` из класса `QProgressBar`:
  - `TopToBottom` — текст поворачивается на 90° по часовой стрелке;
  - `BottomToTop` — текст поворачивается на 90° против часовой стрелки.

  При использовании стилей `"windows"`, `"windowsxp"` и `"macintosh"` при вертикальной ориентации текст вообще не отображается.
- `setInvertedAppearance(<Флаг>)` — если в качестве параметра указано значение `True`, направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево — при горизонтальной ориентации);
- `setFormat(<Формат>)` — задает формат вывода текстового представления значения. Параметром передается строка формата, в которой могут использоваться следующие специальные символы: `%v` — само текущее значение, `%m` — заданный методами `setMaximun()` или `setRange()` максимум, `%p` — текущее значение в процентах:

  ```
  lcd.setFormat('Выполнено %v шагов из %m')
  ```

При изменении значения процесса генерируется сигнал `valueChanged(<Новое значение>)`. Внутри обработчика через параметр доступно новое значение, заданное целым числом.

22.13. Шкала с ползунком

Класс `QSlider` реализует шкалу с ползунком. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QSlider
```

Форматы конструктора класса `QSlider`:

```
QSlider([parent=None])
QSlider(<Ориентация>[, parent=None])
```

В качестве значения параметра `<Ориентация>` указываются элементы `Horizontal` (горизонтальная) или `Vertical` (вертикальная — значение по умолчанию) перечисления `Orientation` из модуля `QtCore.Qt`.

Класс `QSlider` наследует следующие методы из класса `QAbstractSlider` (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-2/qabstractslider.html**):

- `setValue(<Значение>)` — задает новое целочисленное значение для шкалы. Метод является слотом;
- `value()` — возвращает текущее значение в виде числа;

- `setSliderPosition(<Положение>)` — задает текущее положение ползунка;
- `sliderPosition()` — возвращает текущее положение ползунка в виде числа. Если отслеживание перемещения ползунка включено (принято по умолчанию), то возвращаемое значение будет совпадать со значением, возвращаемым методом `value()`. Если отслеживание выключено, то при перемещении метод `sliderPosition()` вернет текущее положение, а метод `value()` — положение, которое имел ползунок до перемещения;
- `setRange(<Минимум>, <Максимум>)` — задает минимальное и максимальное значения, представленные целыми числами. Метод является слотом;
- `setMinimum(<Минимум>)` — задает минимальное значение в виде целого числа;
- `setMaximum(<Максимум>)` — задает максимальное значение в виде целого числа;
- `setOrientation(<Ориентация>)` — задает ориентацию шкалы. В качестве значения указываются элементы `Horizontal` (горизонтальная) или `Vertical` (вертикальная) перечисления `Orientation` из модуля `QtCore.Qt`. Метод является слотом;
- `setSingleStep(<Значение>)` — задает значение, на которое сдвинется ползунок при нажатии клавиш со стрелками;
- `setPageStep(<Значение>)` — задает значение, на которое сдвинется ползунок при нажатии клавиш <Page Up> и <Page Down>, повороте колесика мыши или щелчке мышью на шкале;
- `setInvertedAppearance(<Флаг>)` — если в качестве параметра указано значение `True`, направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево — при горизонтальной ориентации);
- `setInvertedControls(<Флаг>)` — если в качестве параметра указано значение `False`, то при изменении направления увеличения значения будет изменено и направление перемещения ползунка при нажатии клавиш <Page Up> и <Page Down>, повороте колесика мыши и нажатии клавиш со стрелками вверх и вниз;
- `setTracking(<Флаг>)` — если в качестве параметра указано значение `True`, отслеживание перемещения ползунка будет включено (принято по умолчанию). При этом сигнал `valueChanged()` при перемещении ползунка станет генерироваться постоянно. Если в качестве параметра указано значение `False`, то сигнал `valueChanged()` будет сгенерирован только при отпускании ползунка;
- `hasTracking()` — возвращает значение `True`, если отслеживание перемещения ползунка включено, и `False` — в противном случае.

Класс `QAbstractSlider` поддерживает сигналы:

- `actionTriggered(<Действие>)` — генерируется, когда происходит взаимодействие с ползунком (например, при нажатии клавиши <PgUp>). Внутри обработчика через параметр доступно произведенное действие, которое описывается целым числом. Также можно использовать следующие элементы перечисления `SliderAction` из класса `QAbstractSlider`:
  - `SliderNoAction` (`0`) — нет взаимодействия с ползунком;
  - `SliderSingleStepAdd` (`1`) — увеличение значения путем щелчка на клавише-стрелке (какой именно, зависит от указанного направления увеличения значения);
  - `SliderSingleStepSub` (`2`) — уменьшение значения путем щелчка на клавише-стрелке;

- `SliderPageStepAdd` (3) — увеличение значения путем щелчка на клавише <PgUp> или <PgDn> (какой именно, зависит от указанного направления увеличения значения);
- `SliderPageStepSub` (4) — уменьшение значения путем щелчка на клавише <PgUp> или <PgDn>;
- `SliderToMinimum` (5) — сдвиг ползунка в начало шкалы нажатием клавиши <Home>;
- `SliderToMaximum` (6) — сдвиг ползунка в конец шкалы нажатием клавиши <End>;
- `SliderMove` (7) — перемещение ползунка мышью;

- `rangeChanged(<Минимум>, <Максимум>)` — генерируется при изменении диапазона значений. Внутри обработчика через параметры доступны новые минимальное и максимальное значения, заданные целыми числами;
- `sliderPressed()` — генерируется при нажатии ползунка;
- `sliderMoved(<Положение>)` — генерируется постоянно при перемещении ползунка. Внутри обработчика через параметр доступно новое положение ползунка, выраженное целым числом;
- `sliderReleased()` — генерируется при отпускании ранее нажатого ползунка;
- `valueChanged(<Значение>)` — генерируется при изменении значения. Внутри обработчика через параметр доступно новое значение в виде целого числа.

Класс `QSlider` дополнительно определяет следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-6/qslider.html**):

- `setTickPosition(<Позиция>)` — задает позицию рисок. В качестве параметра указываются следующие элементы перечисления `TickPosition` из класса `QSlider`:
  - `NoTicks` — без рисок;
  - `TicksBothSides` — риски по обе стороны;
  - `TicksAbove` — риски выводятся сверху;
  - `TicksBelow` — риски выводятся снизу;
  - `TicksLeft` — риски выводятся слева;
  - `TicksRight` — риски выводятся справа;
- `setTickInterval(<Расстояние>)` — задает расстояние между отдельными рисками.

22.14. Круговая шкала с ползунком

Класс `QDial` реализует круглую шкалу с ползунком, который можно перемещать по кругу с помощью мыши или клавиатуры. Компонент, показанный на рис. 22.6, напоминает регулятор, используемый в различных устройствах для изменения каких-либо настроек. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QDial
```

Формат конструктора класса `QDial`:

```
QDial([parent=None])
```

Класс `QDial` наследует все методы и сигналы класса `QAbstractSlider` (см. *разд. 22.13*) и определяет несколько дополнительных методов (здесь приведена только часть методов — полный их список смотрите на странице **https://doc.qt.io/qt-6/qdial.html**):

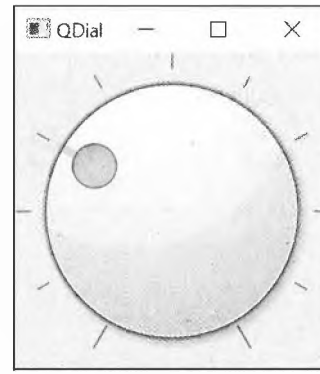


Рис. 22.6. Компонент `QDial`

- `setNotchesVisible(<Флаг>)` — если в качестве параметра указано значение `True`, будут отображены риски. По умолчанию риски не выводятся. Метод является слотом;
- `setNotchTarget(<Значение>)` — задает рекомендуемое расстояние между рисками в пикселах. В качестве параметра указывается вещественное число;
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `True`, то начало шкалы будет совпадать с ее концом. По умолчанию между началом шкалы и концом расположено пустое пространство. Метод является слотом.

22.15. Полоса прокрутки

Класс `QScrollBar` реализует горизонтальную или вертикальную полосу прокрутки. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QScrollBar
```

Форматы конструктора класса `QScrollBar`:

```
QScrollBar([parent=None])
QScrollBar(<Ориентация>[, parent=None])
```

В качестве значения параметра `<Ориентация>` указываются элементы `Horizontal` (горизонтальная) или `Vertical` (вертикальная — значение по умолчанию) перечисления `Orientation` из модуля `QtCore.Qt`.

Класс `QScrollBar` наследует все методы и сигналы класса `QAbstractSlider` (см. *разд. 22.13*) и не определяет дополнительных методов.

22.16. Веб-браузер

Чтобы использовать компонент полнофункционального веб-браузера, потребуется установить дополнительную библиотеку PyQt6-WebEngine, для чего следует в консоли подать следующую команду:

```
pip install PyQt6-WebEngine
```

Веб-браузер реализуется классом `QWebEngineView` из модуля `QtWebEngineWidgets` (рис. 22.7). Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QWebEngineView
```

Рис. 22.7. Компонент `QWebEngineView`

Формат конструктора класса `QWebEngineView`:

```
QWebEngineView([parent=None])
```

Класс `QWebEngineView` поддерживает следующие полезные для нас методы (полный их список смотрите на странице **https://doc.qt.io/qt-6/qwebengineview.html**):

- `load(<Интернет-адрес>)` и `setUrl(<Интернет-адрес>)` — загружают и выводят страницу с указанным в параметре адресом, который задается в виде объекта класса `QUrl` из модуля `QtCore`:

  ```
  web.load(QtCore.QUrl('https://www.google.ru/'))
  ```

- `url()` — возвращает адрес текущей страницы в виде объекта класса `QUrl`;
- `title()` — возвращает заголовок (содержимое тега `<title>`) текущей страницы;
- `setHtml(<HTML-код>[, baseUrl=QUrl()])` — задает HTML-код страницы, которая будет отображена в компоненте.

  Необязательный параметр `baseUrl` указывает базовый адрес, относительно которого будут отсчитываться относительные адреса в гиперссылках, ссылках на файлы изображений, таблицы стилей, файлы сценариев и пр. Если этот параметр не указан, в качестве значения по умолчанию используется «пустой» объект класса `QUrl`, и относительные адреса будут отсчитываться от каталога, где находится сам файл страницы. Пример:

  ```
  wv.setHtml("<h1>Заголовок</h1>")
  # Файл page2.html будет загружен с сайта http://www.somesite.ru/
  webview.setHtml("<a href='page2.html'>Вторая страница</a>",
                  QtCore.QUrl('http://www.somesite.ru/'))
  ```

- `selectedText()` — возвращает выделенный текст или пустую строку, если ничего не было выделено;
- `hasSelection()` — возвращает `True`, если какой-либо фрагмент страницы был выделен, и `False` — в противном случае;
- `setZoomFactor(<Множитель>)` — задает масштаб самой страницы. Значение 1 указывает, что страница будет выведена в оригинальном масштабе, значение меньше единицы — в уменьшенном, значение больше единицы — увеличенном масштабе;
- `zoomFactor()` — возвращает масштаб страницы;
- `back()` — загружает предыдущий ресурс из списка истории. Метод является слотом;
- `forward()` — загружает следующий ресурс из списка истории. Метод является слотом;
- `reload()` — перезагружает страницу. Метод является слотом;
- `stop()` — останавливает загрузку страницы. Метод является слотом;
- `icon()` — возвращает в виде объекта класса `QIcon` значок, заданный для страницы;
- `findText(<Искомый текст>[, options={}][, resultCallback=0])` — ищет на странице заданный фрагмент текста. Все найденные фрагменты будут выделены желтым фоном непосредственно в компоненте.

  Необязательный параметр `option` задает дополнительные параметры в виде одного из следующих элементов перечисления `FindFlag` класса `QWebEnginePage` из модуля `QtWebEngineCore` (или их комбинации через оператор `|`):

  - `FindBackward` — выполнять поиск в обратном, а не в прямом направлении;
  - `FindCaseSensitively` — поиск с учетом регистра символов (по умолчанию выполняется поиск без учета регистра).

  В необязательном параметре `resultCallback` можно указать функцию, которая будет вызвана по окончании поиска. Эта функция должна принимать в качестве единственного параметра объект класса `QWebEngineFindTextResult` из модуля `QtWebEngineCore`. Этот класс поддерживает следующие методы:

  - `numberOfMatches()` — возвращает количество совпавших фрагментов;
  - `activeMatch()` — индекс выделенного в текущий момент совпавшего фрагмента.

  Пример поиска с учетом регистра и выделением всех найденных совпадений:

```
web.findText('Python',
    options=QtWebEngineCore.QWebEnginePage.FindFlag.FindBackward |
            QtWebEngineCore.QWebEnginePage.FindFlag.FindCaseSensitively)
```

- `triggerPageAction(<Действие>[, checked=False])` — выполняет над страницей указанное действие. В качестве действия задается один из элементов перечисления `WebAction` класса `QWebEnginePage` из модуля `QtWebEngineCore` — этих атрибутов очень много, и все они приведены на странице **https://doc.qt.io/qt-6/qwebenginepage.html#WebAction-enum**. Необязательный параметр `checked` имеет смысл указывать лишь для действий, принимающих логический флаг:

```
# Выделение всей страницы
web.triggerPageAction(QtWebEngineCore.QWebEnginePage.WebAction.SelectAll)
# Копирование выделенного фрагмента страницы
web.triggerPageAction(QtWebEngineCore.QWebEnginePage.WebAction.Copy)
```

```
# Перезагрузка страницы, минуя кеш
web.triggerPageAction(
        QtWebEngineCore.QWebEnginePage.WebAction.ReloadAndBypassCache)
```

- `page()` — возвращает объект класса `QWebEnginePage` из модуля `QtWebEngineCore`, представляющий открытую веб-страницу (описан далее);
- `print(<Принтер>)` — печатает содержимое открытой страницы на заданном принтере (указывается в виде объекта класса `QPrinter`);
- `printToPdf()` — преобразует страницу в формат PDF. Форматы метода:

```
printToPdf(<Путь к файлу>,
        pageLayout=QPageLayout(QPageSize(QPageSize.PageSizeId.A4),
                               QPageLayout.Orientation.Portrait,
                               QMarginsF()),
        ranges={})
printToPdf(<Функция>,
        pageLayout=QPageLayout(QPageSize(QPageSize.PageSizeId.A4),
                               QPageLayout.Orientation.Portrait,
                               QMarginsF()),
        ranges={})
```

Первый формат сразу сохраняет преобразованную страницу в файле, чей путь указан первым параметром. Второй формат после преобразования вызывает указанную в первом параметре функцию, передавая ей в качестве единственного параметра преобразованную страницу в виде объекта класса `QByteArray`.

Параметр `pageLayout` задает настройки страницы в виде объекта класса `QPageLayout` (описан в *главе 30*). Если параметр не указан, будет выполнена печать на бумаге типоразмера A4, в портретной ориентации, без отступов.

Параметр `ranges` задает номера страниц, которые попадут в формируемый PDF-документ, в виде множества. Если он не указан, в документ попадут все страницы.

Класс `QWebEngineView` поддерживает следующий набор полезных сигналов (полный их список смотрите на странице **https://doc.qt.io/qt-6/qwebengineview.html**):

- `iconChanged(<Значок QIcon>)` — генерируется после загрузки или изменения значка, заданного для страницы. В параметре, передаваемом обработчику, доступен полученный значок;
- `loadFinished(<Флаг>)` — генерируется по окончании загрузки страницы. Значение `True` параметра указывает, что загрузка выполнена без проблем, `False` — что при загрузке произошли ошибки;
- `loadProgress(<Процент выполнения>)` — периодически генерируется в процессе загрузки страницы. В качестве параметра передается целое число от 0 до 100, показывающее процент загрузки;
- `loadStarted()` — генерируется после начала загрузки страницы;
- `selectionChanged()` — генерируется при выделении нового фрагмента содержимого страницы;
- `titleChanged(<Текст>)` — генерируется при изменении текста заголовка страницы (содержимого тега `<title>`). В параметре, передаваемом обработчику, доступен этот текст в виде строки;

- `urlChanged(<Интернет-адрес QUrl>)` — генерируется при изменении интернет-адреса текущей страницы, что может быть вызвано, например, загрузкой новой страницы. Параметр — новый интернет-адрес.

Класс `QWebEnginePage`, представляющий открытую в веб-браузере страницу, поддерживает следующие полезные методы (полный их список смотрите на странице **https://doc.qt.io/qt-6/qwebenginepage.html**):

- `save(<Путь к файлу>, format=SaveFormatPage.MimeHtmlSaveFormat)` — сохраняет страницу в файле, чей путь указан в первом параметре. Параметр `format` задает формат файла в виде одного из следующих элементов перечисления `SavePageFormat` класса `QWebEngineDownloadRequest` из модуля `QtWebEngineCore`:
  - `SingleHtmlSaveFormat` — обычный HTML-файл. Связанные со страницей файлы (изображения, аудио- и видеоролики, таблицы стилей и пр.) не сохраняются;
  - `CompleteHtmlSaveFormat` — то же самое, только связанные файлы будут сохранены в каталоге, находящемся там же, где и файл со страницей, и имеющем то же имя;
  - `MimeHtmlSaveFormat` — страница и все связанные файлы сохраняются в одном файле;
- `contentsSize()` — возвращает объект класса `QSizeF`, хранящий размеры содержимого страницы;
- `scrollPosition()` — возвращает объект класса `QPointF`, хранящий позицию прокрутки содержимого страницы;
- `setAudioMuted(<Флаг>)` — если с параметром передать значение `True`, все звуки, воспроизводящиеся на странице, будут приглушены. Чтобы снова сделать их слышимыми, нужно передать значение `False`;
- `isAudioMuted()` — возвращает `True`, если все звуки, воспроизводящиеся на странице, приглушены, и `False` — в противном случае;
- `setBackgroundColor(<Цвет QColor>)` — задает для страницы фоновый цвет;
- `backgroundColor()` — возвращает фоновый цвет страницы в виде объекта класса `QColor`.

ГЛАВА 23

Списки и таблицы

Классы, реализующие всевозможные списки (раскрывающийся, обычный и иерархический) и таблицу, определены в модуле `QtWidgets`.

23.1. Раскрывающийся список

Класс `QComboBox` реализует раскрывающийся список с возможностью выбора единственного пункта. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QComboBox
```

Формат конструктора класса `QComboBox`:

```
QComboBox([parent=None])
```

23.1.1. Добавление, изменение и удаление элементов

Для добавления, изменения, удаления и получения значений элементов предназначены следующие методы класса `QComboBox`:

- `addItem()` — добавляет один элемент в конец списка. Форматы метода:

  ```
  addItem(<Текст элемента>[, <Данные>])
  addItem(<Значок QIcon>, <Текст элемента>[, <Данные>])
  ```

  В параметре `<Значок>` задается значок, который будет отображен перед текстом. Необязательный параметр `<Данные>` указывает произвольные данные, сохраняемые в элементе (например, индекс в таблице базы данных);

- `addItems(<Список строк>)` — добавляет несколько элементов в конец списка;

- `insertItem()` — вставляет один элемент в указанную позицию списка. Форматы метода:

  ```
  insertItem(<Индекс позиции>, <Текст элемента>[, <Данные>])
  insertItem(<Индекс позиции>, <Значок QIcon>, <Текст элемента>[, <Данные>])
  ```

- `insertItems(<Индекс позиции>, <Список строк>)` — вставляет несколько элементов в указанную позицию списка;

- `insertSeparator(<Индекс позиции>)` — вставляет разделительную линию в указанную позицию;

- `setItemText(<Индекс>, <Строка>)` — изменяет текст элемента с указанным индексом;

- `setItemIcon(<Индекс>, <Значок QIcon>)` — изменяет значок элемента с указанным индексом;
- `setItemData(<Индекс>, <Данные>[, role=ItemDataRole.UserRole])` — изменяет данные у элемента с указанным индексом. Необязательный параметр `role` указывает роль, для которой задаются данные. Например, если указать элемент `ToolTipRole` перечисления `ItemDataRole` из модуля `QtCore.Qt`, данные зададут текст всплывающей подсказки, которая будет отображена при наведении курсора мыши на элемент. По умолчанию изменяются пользовательские данные;
- `removeItem(<Индекс>)` — удаляет элемент с указанным индексом;
- `setCurrentIndex(<Индекс>)` — делает элемент с указанным индексом текущим. Метод является слотом;
- `currentIndex()` — возвращает индекс текущего элемента;
- `setCurrentText(<Текст>)` — делает элемент с указанным текстом текущим. Метод является слотом;
- `currentText()` — возвращает текст текущего элемента;
- `currentData([role=ItemDataRole.UserRole])` — возвращает данные текущего элемента, относящиеся к заданной роли;
- `itemText(<Индекс>)` — возвращает текст элемента с указанным индексом;
- `itemData(<Индекс>[, role=ItemDataRole.UserRole])` — возвращает данные, сохраненные в роли `role` элемента с индексом `<Индекс>`;
- `count()` — возвращает общее количество элементов списка. Получить количество элементов можно также с помощью функции `len()`;
- `clear()` — удаляет все элементы списка. Метод является слотом.

23.1.2. Изменение параметров списка

Управлять параметрами раскрывающегося списка позволяют следующие методы класса `QComboBox`:

- `setEditable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь сможет вводить текст в раскрывающийся список и, возможно, добавлять таким образом в него новые элементы;
- `setInsertPolicy(<Режим>)` — задает режим добавления в список элементов, введенных пользователем. В качестве параметра указываются следующие элементы перечисления `InsertPolicy` из класса `QComboBox`:
  - `NoInsert` — элемент не будет добавлен;
  - `InsertAtTop` — элемент вставляется в начало списка;
  - `InsertAtCurrent` — будет изменен текст текущего элемента;
  - `InsertAtBottom` — элемент добавляется в конец списка;
  - `InsertAfterCurrent` — элемент вставляется после текущего элемента;
  - `InsertBeforeCurrent` — элемент вставляется перед текущим элементом;
  - `InsertAlphabetically` — при вставке учитывается алфавитный порядок следования элементов;

- `setEditText(<Текст>)` — вставляет текст в поле редактирования. Метод является слотом;
- `clearEditText()` — удаляет текст из поля редактирования. Метод является слотом;
- `setCompleter(<Список QCompleter>)` — задает список вариантов значений для автозавершения;
- `setValidator(<Валидатор>)` — устанавливает валидатор в виде объекта класса, производного от `QValidator` (см. *разд. 22.5.3*);
- `setDuplicatesEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может добавить элемент с повторяющимся текстом. По умолчанию повторы запрещены;
- `setMaxCount(<Количество>)` — задает максимальное количество элементов в списке. Если до вызова метода количество элементов превышало это количество, лишние элементы будут удалены;
- `setMaxVisibleItems(<Количество>)` — задает максимальное количество видимых элементов в раскрывающемся списке;
- `setMinimumContentsLength(<Количество>)` — задает минимальное количество символов, которое должно помещаться в раскрывающемся списке;
- `setSizeAdjustPolicy(<Режим>)` — задает режим установки ширины списка при изменении содержимого. В качестве параметра указываются следующие элементы перечисления `SizeAdjustPolicy` из класса `QComboBox`:
  - `AdjustToContents` — ширина списка подстраивается под ширину текущего содержимого;
  - `AdjustToContentsOnFirstShow` — ширина списка подстраивается под ширину содержимого, имевшегося в списке при первом его отображении;
  - `AdjustToMinimumContentsLengthWithIcon` — используется значение минимальной ширины, которое установлено с помощью метода `setMinimumContentsLength()`, плюс ширина значка;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, список будет отображаться без рамки;
- `setIconSize(<Размеры QSize>)` — задает максимальные размеры значков;
- `showPopup()` — разворачивает список;
- `hidePopup()` — сворачивает список.

23.1.3. Поиск элементов

Произвести поиск элемента в списке позволяют методы `findText()` (по тексту элемента) и `findData()` (по данным с указанной ролью). Методы возвращают индекс найденного элемента или значение `-1`, если таковой не был найден. Форматы методов:

```
findText(<Текст>[,  flags=MatchFlag.MatchExactly |
                        MatchFlag.MatchCaseSensitive])
findData(<Данные>[, role=ItemDataRole.UserRole][,
                 flags=MatchFlag.MatchExactly | MatchFlag.MatchCaseSensitive])
```

Параметр `flags` задает режим поиска. В качестве значения через оператор `|` можно указать комбинацию следующих элементов перечисления `MatchFlag` из модуля `QtCore.Qt`:

- MatchExactly — полное совпадение с текстом элемента;
- MatchContains — совпадение с любой частью текста элемента;
- MatchStartsWith — совпадение с началом;
- MatchEndsWith — совпадение с концом;
- MatchRegularExpression — поиск с помощью регулярного выражения;
- MatchWildcard — используются подстановочные знаки;
- MatchFixedString — поиск полного совпадения внутри строки, выполняемый по умолчанию без учета регистра символов;
- MatchCaseSensitive — поиск с учетом регистра символов;
- MatchWrap — если просмотрены все элементы и подходящий элемент не найден, поиск начнется с начала списка;
- MatchRecursive — просмотр всей иерархии.

23.1.4. Сигналы

Класс QComboBox поддерживает следующие сигналы:

- activated(<Индекс>) — генерируется при выборе пользователем пункта в списке (даже если индекс не изменился). Внутри обработчика доступен целочисленный индекс элемента;
- currentIndexChanged(<Индекс>) — генерируется при изменении текущего индекса. Внутри обработчика доступен целочисленный индекс элемента (или значение -1, если ни один элемент не выбран);
- currentTextChanged(<Текст>) — то же самое, что и currectIndexChanged(), только в обработчик передается текст элемента (или пустая строка, если ни один элемент не выбран);
- editTextChanged(<Текст>) — генерируется при изменении текста в поле. Внутри обработчика через параметр доступен новый текст;
- highlighted(<Индекс>) — генерируется при наведении курсора мыши на пункт в списке. Внутри обработчика доступен целочисленный индекс элемента;
- textActivated(<Индекс>) — то же самое, что и activated(), только в обработчик передается текст элемента;
- textHighlighted(<Текст>) — то же самое, что и highlighted(), только в обработчик передается текст элемента.

23.2. Список для выбора шрифта

Класс QFontComboBox реализует раскрывающийся список с названиями шрифтов. Шрифт можно выбрать из списка или ввести его название в поле — при этом станут отображаться названия, начинающиеся с введенных букв. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QComboBox — QFontComboBox
```

Формат конструктора класса QFontComboBox:

```
QFontComboBox([parent=None])
```

Класс QFontComboBox наследует все методы и сигналы класса QComboBox (см. *разд. 23.1*) и определяет несколько дополнительных методов:

- setCurrentFont(<Шрифт QFont>) — делает текущим элемент, соответствующий указанному шрифту:

  ```
  comboBox.setCurrentFont(QtGui.QFont("Verdana"))
  ```

  Метод является слотом;

- currentFont() — возвращает объект класса QFont с выбранным шрифтом. Вот пример вывода названия шрифта:

  ```
  print(comboBox.currentFont().family())
  ```

- setFontFilters(<Фильтр>) — выводит в списке только шрифты типов, соответствующих указанному фильтру. В качестве параметра указывается комбинация следующих элементов перечисления FontFilter из класса QFontComboBox:
  - AllFonts — все типы шрифтов;
  - ScalableFonts — масштабируемые шрифты;
  - NonScalableFonts — немасштабируемые шрифты;
  - MonospacedFonts — моноширинные шрифты;
  - ProportionalFonts — пропорциональные шрифты.

Класс QFontComboBox поддерживает сигнал currentFontChanged(<Шрифт QFont>), который генерируется при изменении выбранного шрифта. Внутри обработчика доступен выбранный шрифт.

23.3. Роли элементов

Каждый элемент списка хранит набор величин, каждая из которых относится к определенной роли: текст элемента, шрифт и цвет, которыми отображается элемент, текст всплывающей подсказки и многое другое. Приведем роли элементов (элементы перечисления ItemDataRole из модуля QtCore.Qt):

- DisplayRole — отображаемые данные (обычно текст);
- DecorationRole — изображение (обычно значок);
- EditRole — данные в виде, удобном для редактирования;
- ToolTipRole — текст всплывающей подсказки;
- StatusTipRole — текст для строки состояния;
- WhatsThisRole — текст для расширенной подсказки;
- FontRole — шрифт элемента (объект класса QFont);
- TextAlignmentRole — выравнивание текста внутри элемента;
- BackgroundRole — фон элемента (объект класса QBrush);
- ForegroundRole — цвет текста (объект класса QBrush);
- CheckStateRole — статус флажка. Могут быть указаны следующие элементы перечисления CheckState из модуля QtCore.Qt:

- `Unchecked` — флажок сброшен;
- `PartiallyChecked` — флажок частично установлен;
- `Checked` — флажок установлен;

- `AccessibleTextRole` — текст, выводящийся специализированными устройствами вывода (например, системами чтения с экрана);
- `AccessibleDescriptionRole` — описание элемента, выводящееся специализированными устройствами вывода (например, системами чтения с экрана);
- `SizeHintRole` — рекомендуемый размер элемента (объект класса `QSize`);
- `UserRole` (32) — любые пользовательские данные (например, индекс элемента в базе данных). Можно сохранить несколько данных, указав их в роли с индексом более 32, например:

```
comboBox.setItemData(0, 50, role=QtCore.Qt.ItemDataRole.UserRole)
comboBox.setItemData(0, "Другие данные",
                     role=QtCore.Qt.ItemDataRole.UserRole + 1)
```

23.4. Модели

Для отображения данных в виде списков и таблиц применяется концепция «модель-представление», позволяющая отделить данные от их внешнего вида и избежать дублирования данных. В основе концепции лежат следующие составляющие:

- *модель* — является «оберткой» над данными. Позволяет считывать, добавлять, изменять, удалять данные и управлять ими;
- *представление* — отображает данные модели на экране. Сразу несколько представлений могут выводить одну и ту же модель;
- *модель выделения* — управляет выделением. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом;
- *промежуточная модель* — является прослойкой между базовой моделью и представлением. Позволяет производить сортировку и фильтрацию данных без изменения порядка следования элементов в базовой модели;
- *делегат* — представляет компонент для редактирования данных. Существуют стандартные классы делегатов, кроме того, разработчик может создать свои классы.

23.4.1. Доступ к данным внутри модели

Доступ к данным внутри модели реализуется с помощью класса `QModelIndex` из модуля `QtCore`.

Чаще всего объект класса `QModelIndex` создается с помощью метода `index()` какого-либо класса модели или метода `currentIndex()`, унаследованного моделями из класса `QAbstractItemView`. Такой объект указывает на конкретные данные.

Если запрошенных данных в модели нет, возвращается пустой, невалидный объект класса `QModelIndex`. Его также можно создать обычным вызовом конструктора:

```
QModelIndex()
```

Класс QModelIndex поддерживает следующие методы:

- isValid() — возвращает значение True, если объект является валидным, и False — в противном случае;
- data([role=ItemDataRole.DisplayRole]) — возвращает данные, относящиеся к указанной в параметре role роли (по умолчанию — выводимый на экран текст элемента списка);
- flags() — возвращает свойства элемента в виде комбинации следующих элементов перечисления ItemFlag из модуля QtCore.Qt:
  - NoItemFlags — элемент не имеет свойств;
  - ItemIsSelectable — элемент можно выделить;
  - ItemIsEditable — элемент можно редактировать;
  - ItemIsDragEnabled — элемент можно перетаскивать;
  - ItemIsDropEnabled — в элемент можно сбрасывать перетаскиваемые данные;
  - ItemIsUserCheckable — элемент может быть установлен и сброшен;
  - ItemIsEnabled — пользователь может взаимодействовать с элементом;
  - ItemIsAutoTristate — состояние элемента зависит от состояния вложенных в него элементов (например, родительский элемент устанавливается, если все вложенные в него элементы установлены, и частично устанавливается, если установлена лишь часть вложенных элементов);
  - ItemNeverHasChildren — элемент не может иметь вложенные элементы;
  - ItemIsUserTristate — элемент может находиться в трех состояниях (сброшенном, частично установленном и установленном);
- row() — возвращает индекс строки;
- column() — возвращает индекс столбца;
- parent() — возвращает индекс элемента (объект класса QModelIndex), расположенного на один уровень выше по иерархии. Если такого элемента нет, возвращается невалидный объект класса QModelIndex;
- sibling(<Строка>, <Столбец>) — возвращает индекс элемента (объект класса QModelIndex), расположенного на том же уровне вложенности на указанных строке и столбце. Если такого элемента нет, возвращается невалидный объект класса QModelIndex;
- siblingAtRow(<Строка>) — возвращает индекс элемента (объект класса QModelIndex), расположенного на том же уровне вложенности на указанной строке. Если такого элемента нет, возвращается невалидный объект класса QModelIndex;
- siblingAtColumn(<Столбец>) — возвращает индекс элемента (объект класса QModelIndex), расположенного на том же уровне вложенности на указанном столбце. Если такого элемента нет, возвращается невалидный объект класса QModelIndex;
- model() — возвращает ссылку на модель.

Также класс QModelIndex поддерживает операторы сравнения ==, < и !=.

Надо учитывать, что модель может измениться — тогда объект класса QModelIndex будет ссылаться на уже несуществующий элемент. Если необходимо сохранить ссылку на элемент, следует воспользоваться классом QPersistentModelIndex, который содержит те же самые методы, но обеспечивает валидность ссылки.

23.4.2. Класс *QStringListModel*

Класс `QStringListModel` из модуля `QtCore` реализует одномерную модель, содержащую список строк. Ее содержимое можно отобразить с помощью классов `QListView`, `QComboBox` и др., передав в метод `setModel()` представления. Иерархия наследования:

```
QObject — QAbstractItemModel — QAbstractListModel — QStringListModel
```

Форматы конструктора класса `QStringListModel`:

```
QStringListModel([parent=None])
QStringListModel(<Список строк>[, parent=None])
```

Пример:

```
lst = ['Perl', 'PHP', 'Python', 'Ruby']
slm = QtCore.QStringListModel(lst, parent=window)
cbo = QtWidgets.QComboBox()
cbo.setModel(slm)
```

Класс `QStringListModel` наследует метод `index()` из класса `QAbstractListModel`, который возвращает индекс (объект класса `QModelIndex`) элемента модели. Формат метода:

```
index(<Строка>[, column=0][, parent=QModelIndex()])
```

Первый параметр задает номер строки в модели, в которой хранится нужный элемент. Необязательный параметр `column` указывает номер столбца модели — для класса `QStringListModel`, позволяющего хранить простые списки строк, его следует задать равным 0. Необязательный параметр `parent` позволяет задать элемент верхнего уровня для искомого элемента — если таковой не задан, будет выполнен поиск элемента на самом верхнем уровне иерархии.

Класс `QStringListModel` поддерживает также следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qstringlistmodel.html**):

- `setStringList(<Список строк>)` — устанавливает список строк в качестве содержимого модели;
- `stringList()` — возвращает список строк, хранящихся в модели;
- `insertRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — вставляет указанное количество пустых элементов в позицию, заданную первым параметром, остальные элементы сдвигаются в конец списка. Необязательный параметр `parent` позволяет указать элемент верхнего уровня, в который будут вложены добавляемые элементы, — если таковой не задан, элементы будут добавлены на самый верхний уровень иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество элементов, начиная с позиции, заданной первым параметром. Необязательный параметр `parent` позволяет указать элемент верхнего уровня, в который вложены удаляемые элементы, — если таковой не задан, элементы будут удалены из самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `setData()` — задает значение для роли `role` элемента, на который указывает индекс, заданный первым параметром. Формат метода:

  ```
  setData(<Индекс QModelIndex>, <Значение>[, role=ItemDataRole.EditRole])
  ```

Метод возвращает значение True, если операция выполнена успешно. Пример:

```
lst = QtWidgets.QComboBox()
slm = QtCore.QStringListModel(parent=window)
slm.insertRows(0, 4)
slm.setData(slm.index(0), 'Perl')
slm.setData(slm.index(1), 'PHP')
slm.setData(slm.index(2), 'Python')
slm.setData(slm.index(3), 'Ruby')
lst.setModel(slm)
```

- data(<Индекс QModelIndex>[, role=ItemDataRole.DisplayRole]) — возвращает данные, хранимые в роли, указанной в параметре role, элемента, на который ссылается индекс из первого параметра;
- flags() — возвращает свойства элемента в виде комбинации элементов перечисления ItemFlag из модуля QtCore.Qt (см. *разд. 23.4.1*);
- rowCount([parent=QModelIndex()]) — возвращает количество элементов в модели. Необязательный параметр parent указывает элемент верхнего уровня, при этом будет возвращено количество вложенных в него элементов. Если параметр не задан, возвращается количество элементов верхнего уровня иерархии;
- sort(<Индекс столбца>[, order=SortOrder.AscendingOrder]) — производит сортировку. Если во втором параметре указан элемент AscendingOrder перечисления SordOrder из класса QtCore.Qt, сортировка производится в прямом порядке, а если элемент DescendingOrder того же перечисления — в обратном.

22.4.3. Класс *QStandardItemModel*

Класс QStandardItemModel из модуля QtGui реализует двумерную (таблица) и иерархическую модели. Каждый элемент такой модели представлен классом QStandardItem из того же модуля. Вывести на экран ее содержимое можно с помощью классов QTableView, QTreeView и др., передав модель в метод setModel() представления. Иерархия наследования:

```
QObject — QAbstractItemModel — QStandardItemModel
```

Форматы конструктора класса QStandardItemModel:

```
QStandardItemModel([parent=None])
QStandardItemModel(<Количество строк>, <Количество столбцов>[, parent=None])
```

Пример создания и вывода на экран таблицы из трех столбцов: значка, названия языка программирования и адреса веб-сайта приведен в листинге 23.1.

Листинг 23.1. Использование класса QStandardItemModel

```
from PyQt6 import QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QStandardItemModel")
tv = QtWidgets.QTableView(parent=window)
sti = QtGui.QStandardItemModel(parent=window)
```

```
lst1 = ['Perl', 'PHP', 'Python', 'Ruby']
lst2 = ['http://www.perl.org/', 'http://php.net/', 'https://www.python.org/',
        'https://www.ruby-lang.org/']
for row in range(0, 4):
    if row == 2:
        iconfile = 'python.png'
    else:
        iconfile = 'icon.png'
    item1 = QtGui.QStandardItem(QtGui.QIcon(iconfile), '')
    item2 = QtGui.QStandardItem(lst1[row])
    item3 = QtGui.QStandardItem(lst2[row])
    sti.appendRow([item1, item2, item3])
sti.setHorizontalHeaderLabels(['Значок', 'Название', 'Сайт'])
tv.setModel(sti)
tv.setColumnWidth(0, 50)
tv.setColumnWidth(2, 180)
tv.resize(350, 150)
window.show()
sys.exit(app.exec())
```

Класс `QStandardItemModel` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qstandarditemmodel.html**):

- `setRowCount(<Количество строк>)` — задает количество строк;
- `setColumnCount(<Количество столбцов>)` — задает количество столбцов;
- `rowCount([parent=QModelIndex()])` — возвращает количество строк. Необязательный параметр `parent` указывает элемент верхнего уровня, при этом будет возвращено количество вложенных в этот элемент строк, — если параметр не задан, возвращается количество строк верхнего уровня иерархии;
- `columnCount([parent=QModelIndex()])` — возвращает количество столбцов. Необязательный параметр `parent` в этом случае не используется;
- `setItem(<Строка>, <Столбец>, <Элемент QStandardItem>)` — устанавливает элемент в ячейку, располагающуюся на пересечении указанных строки и столбца;
- `appendRow(<Список элементов QStandardItem>)` — добавляет одну строку в конец модели. В качестве параметра указывается список отдельных столбцов добавляемой строки;
- `appendRow(<Элемент QStandardItem>)` — добавляет строку из одного столбца в конец модели;
- `appendColumn(<Список элементов QStandardItem>)` — добавляет один столбец в конец модели. В качестве параметра указывается список отдельных строк добавляемого столбца;
- `insertRow(<Индекс строки>, <Список элементов QStandardItem>)` — добавляет одну строку в указанную позицию модели. В качестве параметра `<Список>` указывается список отдельных столбцов добавляемой строки;
- `insertRow(<Индекс строки>, <Элемент QStandardItem>)` — добавляет одну строку из одного столбца в указанную позицию модели;
- `insertRow(<Индекс>[, parent=QModelIndex()])` — добавляет одну пустую строку в указанную позицию модели. Необязательный параметр `parent` указывает элемент верхнего

уровня, в который будет вложена добавляемая строка, — если параметр не задан, строка добавляется на самый верхний уровень иерархии. Метод возвращает значение `True`, если операция успешно выполнена;

- `insertRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — добавляет несколько пустых строк в указанную позицию модели. Необязательный параметр `parent` указывает элемент верхнего уровня, в который будут вложены добавляемые строки, — если параметр не задан, строки добавляются на самый верхний уровень иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertColumn(<Индекс столбца>, <Список элементов QStandardItem>)` — добавляет один столбец в указанную позицию модели. В качестве параметра `<Список>` указывается список отдельных строк добавляемого столбца;
- `insertColumn(<Индекс>[, parent=QModelIndex()])` — добавляет один пустой столбец в указанную позицию. Необязательный параметр `parent` указывает элемент верхнего уровня — владелец элементов, в который будет добавлен столбец. Если этот параметр не задан, столбец добавляется в элементы самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция выполнена успешно;
- `insertColumns(<Индекс>, <Количество>[, parent=QModelIndex()])` — добавляет несколько пустых столбцов в указанную позицию. Необязательный параметр `parent` указывает элемент верхнего уровня — владелец элементов, в который будут добавлены столбцы. Если этот параметр не задан, столбцы добавляются в элементы самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество строк, начиная со строки, имеющей заданный индекс. Необязательный параметр `parent` указывает элемент верхнего уровня — владелец удаляемых строк. Если этот параметр не задан, будут удалены строки из самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeColumns(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество столбцов, начиная со столбца, имеющего заданный индекс. Необязательный параметр `parent` указывает элемент верхнего уровня — владелец элементов, из которых будут удалены столбцы. Если этот параметр не задан, удаляются столбцы из элементов самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `takeItem(<Строка>[, <Столбец>=0])` — удаляет указанный элемент из модели и возвращает его в виде объекта класса `QStandardItem`;
- `takeRow(<Индекс>)` — удаляет строку с указанным индексом и возвращает ее в виде списка объектов класса `QStandardItem`;
- `takeColumn(<Индекс>)` — удаляет столбец с указанным индексом и возвращает его в виде списка объектов класса `QStandardItem`;
- `clear()` — удаляет все элементы из модели;
- `item(<Строка>[, <Столбец>=0])` — возвращает ссылку на элемент (объект класса `QStandardItem`), расположенный в указанной ячейке;
- `invisibleRootItem()` — возвращает ссылку на невидимый корневой элемент модели в виде объекта класса `QStandardItem`;

- `itemFromIndex(<Индекс QModelIndex>)` — возвращает ссылку на элемент (объект класса `QStandardItem`), на который ссылается заданный индекс;
- `index(<Строка>, <Столбец>[, parent=QModelIndex()])` — возвращает индекс элемента (объект класса `QModelIndex`), расположенного в указанной ячейке. Необязательный параметр `parent` задает элемент верхнего уровня для искомого элемента. Если таковой не задан, будет выполнен поиск элемента на самом верхнем уровне иерархии;
- `indexFromItem(<Элемент QStandardItem>)` — возвращает индекс указанного элемента в виде объекта класса `QModelIndex`;
- `setData()` — задает значение для роли `role` элемента, на который указывает заданный индекс. Метод возвращает значение `True`, если операция успешно выполнена. Формат метода:

  ```
  setData(<Индекс QModelIndex>, <Значение>[, role=ItemDataRole.EditRole])
  ```

- `data(<Индекс QModelIndex>[, role=ItemDataRole.DisplayRole])` — возвращает данные, относящиеся к указанной роли элемента, на который ссылается заданный индекс;
- `setHorizontalHeaderLabels(<Список строк>)` — задает заголовки столбцов. В качестве параметра указывается список строк;
- `setVerticalHeaderLabels(<Список строк>)` — задает заголовки строк. В качестве параметра указывается список строк;
- `setHorizontalHeaderItem(<Индекс>, <Заголовок QStandardItem>)` — задает заголовок столбца с указанным индексом;
- `setVerticalHeaderItem(<Индекс>, <Заголовок QStandardItem>)` — задает заголовок строки с указанным индексом;
- `horizontalHeaderItem(<Индекс>)` — возвращает заголовок (объект класса `QStandardItem`) столбца с указанным индексом;
- `verticalHeaderItem(<Индекс>)` — возвращает заголовок (объект класса `QStandardItem`) строки с указанным индексом;
- `setHeaderData()` — задает новое значение для указанной роли заголовка. Формат метода:

  ```
  setHeaderData(<Индекс>, <Ориентация>, <Значение>[, role=ItemDataRole.EditRole])
  ```

  В первом параметре указывается индекс строки или столбца, а во втором — ориентация в виде элемента `Horizontal` (горизонтальная) или `Vertical` (вертикальная) перечисления `Orientation` из модуля `QtCore.Qt`. Метод возвращает значение `True`, если операция успешно выполнена;
- `headerData(<Индекс>, <Ориентация>[, role=ItemDataRole.DisplayRole])` — возвращает значение, соответствующее указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором — ориентация;
- `findItems(<Текст>[, flags=MatchFlag.MatchExactly][, column=0])` — производит поиск элемента внутри модели в указанном в параметре `column` столбце по заданному тексту. Допустимые значения параметра `flags` мы рассматривали в *разд. 23.1.3*. В качестве значения метод возвращает список объектов класса `QStandardItem` или пустой список;
- `sort(<Индекс столбца>[, order=SortOrder.AscendingOrder])` — производит сортировку. Если во втором параметре указан элемент `AscendingOrder` перечисления `SortOrder` из модуля `QtCore.Qt`, сортировка производится в прямом порядке, а если элемент `DescendingOrder` того же перечисления — в обратном;

- `setSortRole(<Роль>)` — задает роль (см. *разд. 23.3*), по которой производится сортировка;
- `parent(<Индекс QModelIndex>)` — возвращает индекс (объект класса `QModelIndex`) родительского элемента. В качестве параметра указывается индекс элемента-потомка;
- `hasChildren([parent=QModelIndex()])` — возвращает `True`, если заданный элемент имеет хотя бы одного потомка, и `False` — в противном случае.

При изменении значения элемента генерируется сигнал `itemChanged(<Элемент QStandardItem>)`. Внутри обработчика через параметр доступна ссылка на изменившийся элемент.

23.4.4. Класс *QStandardItem*

Каждый элемент модели `QStandardItemModel` представлен классом `QStandardItem` из модуля `QtGui`. Этот класс не только описывает элемент, но и позволяет создавать вложенные структуры, в которых любой элемент может иметь произвольное количество вложенных в него дочерних элементов или элементов-потомков (что пригодится при выводе иерархического списка). Форматы конструктора класса:

```
QStandardItem()
QStandardItem(<Текст>)
QStandardItem(<Значок QIcon>, <Текст>)
QStandardItem(<Количество строк>[, <Количество столбцов>=1])
```

Последний формат задает количество дочерних элементов и столбцов в них.

Наиболее часто используемые методы класса `QStandardItem` приведены далее (полный их список можно найти на странице **https://doc.qt.io/qt-6/qstandarditem.html**):

- `setRowCount(<Количество строк>)` — задает количество дочерних строк;
- `setColumnCount(<Количество столбцов>)` — задает количество столбцов в дочерних строках;
- `rowCount()` — возвращает количество дочерних строк;
- `columnCount()` — возвращает количество столбцов в дочерних строках;
- `row()` — возвращает индекс строки в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя (находится на самом верхнем уровне иерархии);
- `column()` — возвращает индекс столбца в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя;
- `setChild(<Строка>, <Столбец>, <Элемент QStandardItem>)` — устанавливает заданный третьим параметром элемент в указанную ячейку дочерней таблицы текущего элемента.

Пример создания иерархии и вывода ее на экран с применением иерархического списка показан в листинге 23.2.

Листинг 23.2. Вывод иерархического списка

```
from PyQt6 import QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QStandardItemModel")
```

```
tv = QtWidgets.QTreeView(parent=window)
sti = QtGui.QStandardItemModel(parent=window)
rootitem1 = QtGui.QStandardItem('QAbstractItemView')
rootitem2 = QtGui.QStandardItem('Базовый класс')
item1 = QtGui.QStandardItem('QListView')
item2 = QtGui.QStandardItem('Список')
rootitem1.appendRow([item1, item2])
item1 = QtGui.QStandardItem('QTableView')
item2 = QtGui.QStandardItem('Таблица')
rootitem1.appendRow([item1, item2])
item1 = QtGui.QStandardItem('QTreeView')
item2 = QtGui.QStandardItem('Иерархический список')
rootitem1.appendRow([item1, item2])
sti.appendRow([rootitem1, rootitem2])
sti.setHorizontalHeaderLabels(['Класс', 'Описание'])
tv.setModel(sti)
tv.setColumnWidth(0, 170)
tv.resize(400, 100)
window.show()
sys.exit(app.exec())
```

- `appendRow(<Список элементов QStandardItem>)` — добавляет одну строку в конец дочерней таблицы текущего элемента. В качестве параметра указывается список отдельных столбцов;
- `appendRow(<Элемент QStandardItem>)` — добавляет заданный элемент в конец дочерней таблицы текущего элемента, формируя строку с одним столбцом;
- `appendRows(<Список элементов QStandardItem>)` — добавляет несколько строк, содержащих по одному столбцу, в конец дочерней таблицы текущего элемента. В качестве параметра указывается список добавляемых строк;
- `appendColumn(<Список элементов QStandardItem>)` — добавляет один столбец в конец дочерней таблицы текущего элемента. В качестве параметра указывается список отдельных строки;
- `insertRow(<Индекс строки>, <Список элементов QStandardItem>)` — вставляет одну строку в указанную позицию дочерней таблицы у текущего элемента. В качестве параметра `<Список>` указывается список объектов отдельных столбцов;
- `insertRow(<Индекс строки>, <Элемент QStandardItem>)` — вставляет заданный элемент в указанную позицию дочерней таблицы у текущего элемента, формируя строку с одним столбцом;
- `insertRows(<Индекс строки>, <Список элементов QStandardItem>)` — вставляет несколько строк, содержащих по одному столбцу, в указанную позицию дочерней таблицы у текущего элемента. В качестве параметра `<Список>` указывается список отдельных строк;
- `insertRows(<Индекс строки>, <Количество>)` — вставляет заданное количество пустых строк в указанную позицию дочерней таблицы для текущего элемента;
- `insertColumn(<Индекс столбца>, <Список элементов QStandardItem>)` — вставляет один столбец в указанную позицию дочерней таблицы у текущего элемента. В качестве параметра `<Список>` указывается список отдельных строк;

- `insertColumns(<Индекс>, <Количество>)` — вставляет заданное количество пустых столбцов в указанную позицию дочерней таблицы у текущего элемента;
- `removeRow(<Индекс>)` — удаляет строку с указанным индексом;
- `removeRows(<Индекс>, <Количество>)` — удаляет указанное количество строк, начиная со строки, имеющей заданный индекс;
- `removeColumn(<Индекс>)` — удаляет столбец с указанным индексом;
- `removeColumns(<Индекс>, <Количество>)` — удаляет указанное количество столбцов, начиная со столбца, имеющего заданный индекс;
- `takeChild(<Строка>[, <Столбец>=0])` — удаляет указанный дочерний элемент и возвращает его в виде объекта класса `QStandardItem`;
- `takeRow(<Индекс>)` — удаляет указанную строку из дочерней таблицы и возвращает ее в виде списка объектов класса `QStandardItem`;
- `takeColumn(<Индекс>)` — удаляет указанный столбец из дочерней таблицы и возвращает его в виде списка объектов класса `QStandardItem`;
- `parent()` — возвращает ссылку на родительский элемент (объект класса `QStandardItem`) или значение `None`, если текущий элемент не имеет родителя;
- `child(<Строка>[, <Столбец>=0])` — возвращает ссылку на дочерний элемент (объект класса `QStandardItem`) или значение `None`, если такового нет;
- `hasChildren()` — возвращает значение `True`, если существует хотя бы один дочерний элемент, и `False` — в противном случае;
- `setData(<Значение>[, role=ItemDataRole.UserRole+1])` — устанавливает значение для указанной роли;
- `data([<Роль>=ItemDataRole.UserRole+1])` — возвращает значение, которое соответствует указанной роли;
- `setText(<Текст>)` — задает текст элемента;
- `text()` — возвращает текст элемента;
- `setTextAlignment(<Выравнивание>)` — задает выравнивание текста внутри элемента в виде элемента перечисления `AlignmentFlag` из модуля `QtCore.Qt`;
- `setIcon(<Значок QIcon>)` — задает значок, который будет отображен перед текстом;
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `setWhatsThis(<Текст>)` — задает текст расширенной подсказки;
- `setFont(<Шрифт QFont>)` — задает шрифт элемента;
- `setBackground(<Цвет QBrush>)` — задает цвет фона;
- `setForeground(<Цвет QBrush>)` — задает цвет текста;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, после текста элемента будет выведен флажок, который можно устанавливать и сбрасывать;
- `isCheckable()` — возвращает значение `True`, если после текста элемента выводится флажок, и `False` — в противном случае;
- `setCheckState(<Статус>)` — задает состояние флажка. Могут быть указаны следующие элементы перечисления `CheckState` из модуля `QtCore.Qt`: `Unchecked` (флажок сброшен), `PartiallyChecked` (находится в неопределенном состоянии) и `Checked` (установлен);

- `checkState()` — возвращает текущее состояние флажка в виде элемента перечисления `CheckState` из модуля `QtCore.Qt`;
- `setUserTristate(<Флаг>)` — если в качестве параметра указано значение `True`, флажок может иметь три состояния: установленное, сброшенное и неопределенное (промежуточное);
- `isUserTristate()` — возвращает значение `True`, если флажок может иметь три состояния, и `False` — в противном случае;
- `setFlags(<Флаги>)` — задает свойства элемента (см. *разд. 23.4.1*);
- `flags()` — возвращает значение установленных свойств элемента;
- `setSelectable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может выделить элемент;
- `setEditable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может редактировать текст элемента;
- `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, перетаскивание элемента разрешено;
- `setDropEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, сброс перетаскиваемых данных в элемент разрешен;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может взаимодействовать с элементом. Значение `False` делает элемент недоступным;
- `clone()` — возвращает копию элемента в виде объекта класса `QStandardItem`;
- `index()` — возвращает индекс элемента (объект класса `QModelIndex`);
- `model()` — возвращает ссылку на модель (объект класса `QStandardItemModel`);
- `sortChildren(<Индекс столбца>[, order=SortOrder.AscendingOrder])` — производит сортировку дочерней таблицы. Если во втором параметре указан элемент `AscendingOrder` перечисления `SordOrder` из модуля `QtCore.Qt`, сортировка производится в прямом порядке, а если элемент `DescendingOrder` того же перечисления — в обратном.

23.5. Представления

Для отображения элементов модели предназначены следующие классы представлений:

- `QListView` — простой список с возможностью выбора как одного, так и нескольких пунктов. Пункты списка, помимо текстовой надписи, могут содержать значки;
- `QTableView` — таблица;
- `QTreeView` — иерархический список.

Также можно воспользоваться классами `QComboBox` (раскрывающийся список — см. *разд. 23.1*), `QListWidget` (простой список), `QTableWidget` (таблица) и `QTreeWidget` (иерархический список). Последние три класса нарушают концепцию «модель-представление», хотя и отчасти базируются на ней. За подробной информацией по этим классам обращайтесь к документации.

23.5.1. Класс *QAbstractItemView*

Абстрактный класс `QAbstractItemView` является базовым для всех рассмотренных ранее представлений. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                                             QAbstractItemView
```

Класс `QAbstractItemView` поддерживает следующий набор полезных для нас методов (полный их список можно найти на странице **https://doc.qt.io/qt-6/qabstractitemview.html**):

- `setModel(<Модель>)` — задает для представления модель. В качестве параметра передается объект одного из классов, производных от `QAbstractItemModel`;
- `model()` — возвращает заданную для представления модель;
- `selectedIndexes()` — возвращает выделенные элементы, представленные списком объектов класса `QModelIndex`;
- `setCurrentIndex(<Индекс QModelIndex>)` — делает элемент с указанным индексом текущим. Метод является слотом;
- `currentIndex()` — возвращает индекс (объект класса `QModelIndex`) текущего элемента;
- `setRootIndex(<Индекс QModelIndex>)` — делает элемент с заданным индексом корневым. Метод является слотом;
- `rootIndex()` — возвращает индекс (объект класса `QModelIndex`) корневого элемента;
- `setAlternatingRowColors(<Флаг>)` — если в качестве параметра указано значение `True`, то четные и нечетные строки будут иметь разный цвет фона;
- `setIndexWidget(<Индекс QModelIndex>, <Компонент>)` — устанавливает компонент в позицию, указанную индексом, и делает его потомком представления. Если в той позиции уже находится какой-либо компонент, он удаляется;
- `indexWidget(<Индекс QModelIndex>)` — возвращает ссылку на компонент, который был ранее установлен в позицию, указанную индексом;
- `setSelectionModel(<Модель выделения>)` — устанавливает модель выделения (объект класса `QItemSelectionModel`);
- `selectionModel()` — возвращает модель выделения (объект класса `QItemSelectionModel`);
- `setSelectionMode(<Режим>)` — задает режим выделения элементов. В качестве параметра указываются следующие элементы перечисления `SelectionMode` из класса `QAbstractItemView`:
  - `NoSelection` — элементы не могут быть выделены;
  - `SingleSelection` — можно выделить только один элемент;
  - `MultiSelection` — можно выделить несколько элементов. Повторный щелчок на элементе снимает выделение;
  - `ExtendedSelection` — можно выделить несколько элементов, щелкая на них мышью и удерживая при этом нажатой клавишу <Ctrl>. Можно также нажать на элементе левую кнопку мыши и перемещать мышь, не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;
  - `ContiguousSelection` — можно выделить несколько элементов, нажав на элементе левую кнопку мыши и перемещая мышь, не отпуская кнопку. Если удерживать нажа-

той клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;

- `setSelectionBehavior(<Режим>)` — задает режим представления выделенных элементов. В качестве параметра указываются следующие элементы перечисления `SelectionBehavior` из класса `QAbstractItemView`:
  - `SelectItems` — выделяется отдельный элемент;
  - `SelectRows` — выделяется строка целиком;
  - `SelectColumns` — выделяется столбец целиком;
- `selectAll()` — выделяет все элементы. Метод является слотом;
- `clearSelection()` — снимает выделение. Метод является слотом;
- `edit(<Индекс QModelIndex>)` — переключает элемент с заданным индексом в режим редактирования, не делая его выделенным. Метод является слотом;
- `setEditTriggers(<Действие>)` — задает действие, при котором элемент переключается в режим редактирования. В качестве параметра указывается комбинация следующих элементов перечисления `EditTrigger` из класса `QAbstractItemView`:
  - `NoEditTriggers` — элемент не будет поддерживать редактирование;
  - `CurrentChanged` — при выделении элемента;
  - `DoubleClicked` — при двойном щелчке мышью;
  - `SelectedClicked` — при щелчке мышью на уже выделенном элементе;
  - `EditKeyPressed` — при нажатии клавиши <F2>;
  - `AnyKeyPressed` — при нажатии любой символьной клавиши;
  - `AllEditTriggers` — при любом упомянутом ранее действии;
- `setIconSize(<Размеры QSize>)` — задает размеры значков;
- `setTextElideMode(<Режим>)` — задает режим обрезки текста, если он не помещается в отведенную область (в месте пропуска выводится троеточие). Могут быть указаны следующие элементы перечисления `TextElideMode` из модуля `QtCore.Qt`:
  - `ElideLeft` — текст обрезается слева;
  - `ElideRight` — текст обрезается справа;
  - `ElideMiddle` — текст вырезается посередине;
  - `ElideNone` — текст не обрезается;
- `setTabKeyNavigation(<Флаг>)` — если в качестве параметра указано значение `True`, между элементами можно перемещаться с помощью клавиш <Tab> и <Shift>+<Tab>;
- `scrollTo(<Индекс QModelIndex>[, hint=ScrollHint.EnsureVisible])` — прокручивает представление таким образом, чтобы элемент, на который ссылается заданный индекс, был видим. В параметре `hint` указываются следующие элементы перечисления `ScrollHint` из класса `QAbstractItemView`:
  - `EnsureVisible` — элемент должен находиться в области видимости;
  - `PositionAtTop` — элемент должен находиться в верхней части;
  - `PositionAtBottom` — элемент должен находиться в нижней части;
  - `PositionAtCenter` — элемент должен находиться в центре;

- `scrollToTop()` — прокручивает представление в самое начало. Метод является слотом;
- `scrollToBottom()` — прокручивает представление в самый конец. Метод является слотом;
- `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, перетаскивание элементов разрешено;
- `setDragDropMode(<Режим>)` — задает режим работы drag & drop. В качестве параметра указываются следующие элементы перечисления `DragDropMode` из класса `QAbstractItemView`:
  - `NoDragDrop` — drag & drop не поддерживается;
  - `DragOnly` — поддерживается только перетаскивание;
  - `DropOnly` — поддерживается только сбрасывание;
  - `DragDrop` — поддерживается перетаскивание и сбрасывание;
  - `InternalMove` — допускается лишь перетаскивание внутри компонента;
- `setDrapDropOverwriteMode(<Флаг>)` — если в качестве параметра задано значение `True`, сбрасываемые данные заменят имевшиеся в элементе ранее, если `False` — будут вставлены в позицию сброса (поведение по умолчанию);
- `setDropIndicatorShown(<Флаг>)` — если в качестве параметра указано значение `True`, позиция возможного сброса элемента будет подсвечена;
- `setAutoScroll(<Флаг>)` — если в качестве параметра указано значение `True`, при перетаскивании пункта будет производиться автоматическая прокрутка;
- `setAutoScrollMargin(<Отступ>)` — задает расстояние от края области, при достижении которого будет производиться автоматическая прокрутка области;
- `update(<Индекс QModelIndex>)` — обновляет элемент с заданным индексом. Метод является слотом.

Класс `QAbstractItemView` поддерживает следующие сигналы:

- `activated(<Индекс QModelIndex>)` — генерируется при активизации элемента путем одинарного, двойного щелчка мышью или нажатия клавиши <Enter>. В обработчике через параметр доступен индекс активного элемента;
- `clicked(<Индекс QModelIndex>)` — генерируется при щелчке мышью над элементом. Параметр хранит индекс элемента;
- `doubleClicked(<Индекс QModelIndex>)` — генерируется при двойном щелчке мышью над элементом. Параметр хранит индекс элемента;
- `entered(<Индекс QModelIndex>)` — генерируется при вхождении курсора мыши в область элемента. Чтобы сигнал сработал, необходимо включить обработку перемещения курсора вызовом метода `setMouseTracking()`, унаследованного от класса `QWidget`. Внутри обработчика через параметр доступен индекс элемента;
- `pressed(<Индекс QModelIndex>)` — генерируется при нажатии кнопки мыши над элементом. Внутри обработчика через параметр доступен индекс элемента;
- `viewportEntered()` — генерируется при вхождении курсора мыши в область компонента. Чтобы сигнал сработал, необходимо включить обработку перемещения курсора с помощью метода `setMouseTracking()`, унаследованного от класса `QWidget`.

23.5.2. Простой список

Класс `QListView` реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Кроме текста, в любом пункте такого списка может присутствовать значок (рис. 23.1). Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QListView
```

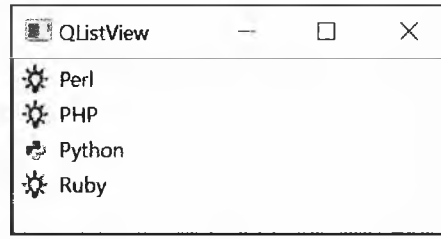


Рис. 23.1. Компонент `QListView`

Формат конструктора класса `QListView`:

```
QListView([parent=None])
```

Типичный пример использования списка приведен в листинге 23.3.

Листинг 23.3. Простой список `QListView`

```
from PyQt6 import QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QListView")
lv = QtWidgets.QListView(parent=window)
sti = QtGui.QStandardItemModel(parent=window)
lst = ['Perl', 'PHP', 'Python', 'Ruby']
for row in range(0, 4):
    if row == 2:
        iconfile = 'python.png'
    else:
        iconfile = 'icon.png'
    item = QtGui.QStandardItem(QtGui.QIcon(iconfile), lst[row])
    sti.appendRow(item)
lv.setModel(sti)
lv.resize(250, 100)
window.show()
sys.exit(app.exec())
```

Класс `QListView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 23.5.1*), включая методы `setModel()`, `model()` и `selectedIndexes()`. Помимо этого, он дополнительно определяет следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qlistview.html**):

- `setModelColumn(<Индекс>)` — задает индекс отображаемого столбца в табличной модели (по умолчанию отображается первый столбец с индексом 0);
- `setViewMode(<Режим>)` — задает режим отображения элементов. В качестве параметра указываются следующие элементы перечисления `ViewMode` из класса `QListView`:
  - `ListMode` — элементы выстраиваются по вертикали, а значки имеют маленькие размеры;
  - `IconMode` — элементы выстраиваются по горизонтали, а значки имеют большие размеры. Элементы можно свободно перемещать мышью;
- `setMovement(<Режим>)` — задает режим перемещения элементов. В качестве параметра указываются следующие элементы перечисления `Movement` из класса `QListView`:
  - `Static` — пользователь не может перемещать элементы;
  - `Free` — свободное перемещение;
  - `Snap` — перемещаемые элементы автоматически выравниваются по сетке, размеры которой задаются методом `setGridSize()`;
- `setGridSize(<Размеры QSize>)` — задает размеры отдельной ячейки сетки, по которой выравниваются перемещаемые элементы;
- `setResizeMode(<Режим>)` — задает режим расположения элементов при изменении размера списка. В качестве параметра указываются следующие элементы перечисления `ResizeMode` из класса `QListView`:
  - `Fixed` — элементы остаются в том же положении;
  - `Adjust` — положение элементов изменяется при изменении размеров;
- `setFlow(<Режим>)` — задает порядок вывода элементов. В качестве параметра указываются следующие элементы перечисления `Flow` из класса `QListView`: `LeftToRight` (слева направо) и `TopToBottom` (сверху вниз);
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `False`, перенос элементов на новую строку (если они не помещаются в ширину области) запрещен;
- `setWordWrap(<Флаг>)` — если в качестве параметра указано значение `True`, текст элементов при необходимости будет переноситься по строкам;
- `setLayoutMode(<Режим>)` — задает режим размещения элементов. В качестве параметра указываются следующие элементы перечисления `LayoutMode` из класса `QListView`:
  - `SinglePass` — элементы размещаются все сразу. Если список слишком большой, то окно останется заблокированным, пока все элементы не будут отображены;
  - `Batched` — элементы размещаются блоками. Размер такого блока задается методом `setBatchSize()`;
- `setBatchSize(<Количество>)` — задает количество элементов в отдельном блоке, если в качестве режима размещения элементов указан `Batched`;
- `setUniformItemSizes(<Флаг>)` — если в качестве параметра указано значение `True`, все элементы будут иметь одинаковый размер (по умолчанию они имеют разные размеры, зависящие от содержимого);
- `setSpacing(<Отступ>)` — задает отступ вокруг элемента;
- `setSelectionRectVisible(<Флаг>)` — если в качестве параметра указано значение `True`, будет отображаться вспомогательная рамка, показывающая область выделения. Метод доступен только при использовании режима множественного выделения;

- `setRowHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, строка с индексом, указанным в первом параметре, будет скрыта. Значение `False` отображает строку;
- `isRowHidden(<Индекс>)` — возвращает значение `True`, если строка с указанным индексом скрыта, и `False` — в противном случае.

Класс `QListView` поддерживает сигнал `indexesMoved(<Список индексов QModelIndex>)`, генерируемый при перемещении элементов. Внутри обработчика через параметр доступен список индексов перемещаемых элементов.

23.5.3. Таблица

Класс `QTableView` реализует таблицу (рис. 23.2). Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QTableView
```

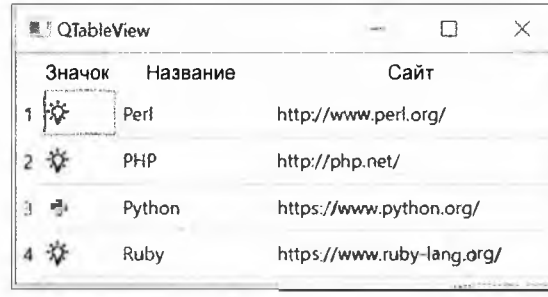


Рис. 23.2. Компонент `QTableView`

Формат конструктора класса `QTableView`:

```
QTableView([parent=None])
```

Класс `QTableView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 23.5.1*) и дополнительно поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qtableview.html**):

- `selectRow(<Индекс>)` — выделяет строку с указанным индексом. Метод является слотом;
- `selectColumn(<Индекс>)` — выделяет столбец с указанным индексом. Метод является слотом;
- `horizontalHeader()` — возвращает ссылку на горизонтальный заголовок, представленный объектом класса `QHeaderView`;
- `verticalHeader()` — возвращает ссылку на вертикальный заголовок, представленный объектом класса `QHeaderView`. Например, вывести таблицу без заголовков можно следующим образом:

```
view.horizontalHeader().hide()
view.verticalHeader().hide()
```

- `setRowHeight(<Индекс>, <Высота>)` — задает высоту строки с указанным в первом параметре индексом;
- `rowHeight(<Индекс>)` — возвращает высоту строки с указанным индексом;

- `setColumnWidth(<Индекс>, <Ширина>)` — задает ширину столбца с указанным в первом параметре индексом;
- `columnWidth(<Индекс>)` — возвращает ширину столбца с указанным индексом;
- `resizeRowToContents(<Индекс строки>)` — изменяет размер указанной строки таким образом, чтобы в нее поместилось все содержимое. Метод является слотом;
- `resizeRowsToContents()` — изменяет размер всех строк таким образом, чтобы в них поместилось все содержимое. Метод является слотом;
- `resizeColumnToContents(<Индекс столбца>)` — изменяет размер указанного столбца таким образом, чтобы в него поместилось все содержимое. Метод является слотом;
- `resizeColumnsToContents()` — изменяет размер всех столбцов таким образом, чтобы в них поместилось содержимое. Метод является слотом;
- `setSpan()` — растягивает элемент с указанными в первых двух параметрах индексами на заданное количество строк и столбцов, производя как бы объединение ячеек таблицы. Формат метода:

```
setSpan(<Индекс строки>, <Индекс столбца>, <Количество строк>,
                                           <Количество столбцов>)
```

- `rowSpan(<Индекс строки>, <Индекс столбца>)` — возвращает количество ячеек в строке, которое занимает элемент с указанными индексами;
- `columnSpan(<Индекс строки>, <Индекс столбца>)` — возвращает количество ячеек в столбце, которое занимает элемент с указанными индексами;
- `clearSpans()` — отменяет все объединения ячеек;
- `setRowHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то строка с индексом, указанным в первом параметре, будет скрыта. Значение `False` отображает строку;
- `hideRow(<Индекс>)` — скрывает строку с указанным индексом. Метод является слотом;
- `showRow(<Индекс>)` — отображает строку с указанным индексом. Метод является слотом;
- `isRowHidden(<Индекс>)` — возвращает значение `True`, если строка с указанным индексом скрыта, и `False` — в противном случае;
- `setColumnHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то столбец с индексом, указанным в первом параметре, будет скрыт. Значение `False` отображает столбец;
- `hideColumn(<Индекс>)` — скрывает столбец с указанным индексом. Метод является слотом;
- `showColumn(<Индекс>)` — отображает столбец с указанным индексом. Метод является слотом;
- `isColumnHidden(<Индекс>)` — возвращает значение `True`, если столбец с указанным индексом скрыт, и `False` — в противном случае;
- `isIndexHidden(<Индекс QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом скрыт, и `False` — в противном случае;
- `setGridStyle(<Стиль>)` — задает стиль линий сетки. В качестве параметра указываются следующие элементы перечисления `PenStyle` из модуля `QtCore.Qt`:

- NoPen — линии не выводятся;
- SolidLine — сплошная линия;
- DashLine — штриховая линия;
- DotLine — точечная линия;
- DashDotLine — штрих и точка, штрих и точка и т. д.;
- DashDotDotLine — штрих и две точки, штрих и две точки и т. д.;

- setShowGrid(<Флаг>) — если в качестве параметра указано значение True, то сетка будет отображена, а если False — то скрыта. Метод является слотом;
- setSortingEnabled(<Флаг>) — если в качестве параметра указано значение True, столбцы можно сортировать, щелкая мышью на их заголовках. При этом в заголовке показывается текущее направление сортировки;
- setCornerButtonEnabled(<Флаг>) — если в качестве параметра указано значение True, с помощью кнопки в левом верхнем углу заголовка можно выделить всю таблицу. Значение False отключает кнопку;
- setWordWrap(<Флаг>) — если в качестве параметра указано значение True, текст элементов при необходимости будет переноситься по строкам;
- sortByColumn(<Индекс столбца>, <Направление>) — производит сортировку. Если во втором параметре указан элемент AscendingOrder перечисления SortOrder из модуля QtCore.Qt, сортировка производится в прямом порядке, а если элемент DescendingOrder того же перечисления — в обратном. Метод является слотом.

23.5.4. Иерархический список

Класс QTreeView реализует иерархический список (рис. 23.3). Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QTreeView
```

Формат конструктора класса QTreeView:

```
QTreeView([parent=None])
```

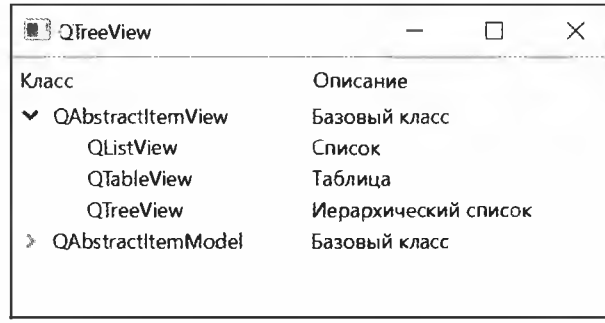


Рис. 23.3. Компонент QTreeView

Класс QTreeView наследует все методы и сигналы класса QAbstractItemView (см. *разд. 22.5.1*) и дополнительно поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qtreeview.html**):

- `header()` — возвращает ссылку на горизонтальный заголовок (объект класса `QHeaderView`);
- `setColumnWidth(<Индекс>, <Ширина>)` — задает ширину столбца с указанным в первом параметре индексом;
- `columnWidth(<Индекс>)` — возвращает ширину столбца;
- `rowHeight(<Индекс QModelIndex>)` — возвращает высоту строки, в которой находится элемент с указанным индексом;
- `resizeColumnToContents(<Индекс столбца>)` — изменяет ширину указанного столбца таким образом, чтобы в нем поместилось все содержимое. Метод является слотом;
- `setUniformRowHeights(<Флаг>)` — если в качестве параметра указано значение `True`, все элементы будут иметь одинаковую высоту;
- `setHeaderHidden(<Флаг>)` — если в качестве параметра указано значение `True`, заголовок будет скрыт. Значение `False` отображает заголовок;
- `isHeaderHidden()` — возвращает значение `True`, если заголовок скрыт, и `False` — в противном случае;
- `setColumnHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то столбец с индексом, указанным в первом параметре, будет скрыт. Значение `False` отображает столбец;
- `hideColumn(<Индекс>)` — скрывает столбец с указанным индексом. Метод является слотом;
- `showColumn(<Индекс>)` — отображает столбец с указанным индексом. Метод является слотом;
- `isColumnHidden(<Индекс>)` — возвращает значение `True`, если столбец с указанным индексом скрыт, и `False` — в противном случае;
- `setRowHidden(<Индекс>, <Индекс родителя QModelIndex>, <Флаг>)` — если в третьем параметре указано значение `True`, то строка с указанными индексом и индексом родителя будет скрыта. Значение `False` отображает строку;
- `isRowHidden(<Индекс>, <Индекс родителя QModelIndex>)` — возвращает значение `True`, если строка с указанными индексом и индексом родителя скрыта, и `False` — в противном случае;
- `isIndexHidden(<Индекс QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом скрыт, и `False` — в противном случае;
- `setExpanded(<Индекс QModelIndex>, <Флаг>)` — если во втором параметре указано значение `True`, то элементы, которые являются дочерними для элемента с указанным в первом параметре индексом, будут отображены, а если `False` — то скрыты;
- `expand(<Индекс QModelIndex>)` — отображает элементы, которые являются дочерними для элемента с указанным индексом. Метод является слотом;
- `expandToDepth(<Уровень>)` — отображает все дочерние элементы до указанного уровня. Метод является слотом;
- `expandAll()` — отображает все дочерние элементы. Метод является слотом;
- `collapse(<Индекс QModelIndex>)` — скрывает элементы, которые являются дочерними для элемента с указанным индексом. Метод является слотом;
- `collapseAll()` — скрывает все дочерние элементы. Метод является слотом;

- isExpanded(<Индекс QModelIndex>) — возвращает значение True, если элементы, которые являются дочерними для элемента с указанным индексом, отображены, и False — в противном случае;
- setItemsExpandable(<Флаг>) — если в качестве параметра указано значение False, пользователь не сможет отображать или скрывать дочерние элементы;
- setAnimated(<Флаг>) — если в качестве параметра указано значение True, отображение и сокрытие дочерних элементов будет производиться с анимацией;
- setIndentation(<Отступ>) — задает отступ для дочерних элементов;
- setRootIsDecorated(<Флаг>) — если в качестве параметра указано значение False, у элементов верхнего уровня не будут показываться элементы управления, с помощью которых производится отображение и сокрытие дочерних элементов;
- setFirstColumnSpanned() — если третьим параметром передано значение True, содержимое первого столбца строки с указанными индексом и индексом родителя, займет всю ширину списка. Формат метода:

  setirstColumnSpanned(<Индекс строки>, <Индекс родителя QModelIndex>, <Флаг>)

- setExpandsOnDoubleClick(<Флаг>) — если передать в параметре значение False, сворачивание и разворачивание пунктов списка будут выполняться по двойному щелчку мыши;
- setSortingEnabled(<Флаг>) — если в качестве параметра указано значение True, столбцы можно сортировать щелчками мыши на их заголовках. При этом в заголовке показывается текущее направление сортировки;
- setWordWrap(<Флаг>) — если в качестве параметра указано значение True, текст элементов при необходимости будет переноситься по строкам;
- sortByColumn(<Индекс столбца>, <Направление>) — производит сортировку. Если во втором параметре указан элемент AscendingOrder перечисления SortOrder из модуля QtCore.Qt, сортировка производится в прямом порядке, а если элемент DescendingOrder того же перечисления — в обратном. Метод является слотом.

Класс QTreeView поддерживает сигналы:

- expanded(<Индекс QModelIndex>) — генерируется при отображении дочерних элементов. Внутри обработчика через параметр доступен индекс элемента-родителя;
- collapsed(<Индекс QModelIndex>) — генерируется при сокрытии дочерних элементов. Внутри обработчика через параметр доступен индекс элемента-родителя.

23.5.5. Управление заголовками строк и столбцов

Класс QHeaderView представляет заголовки строк и столбцов в компонентах QTableView и QTreeView. Получить ссылки на заголовки в классе QTableView позволяют методы horizontalHeader() и verticalHeader(), а для установки заголовков предназначены методы setHorizontalHeader() и setVerticalHeader(). Получить ссылку на заголовок в классе QTreeView позволяет метод header(), а для установки заголовка предназначен метод setHeader(). Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                                    QAbstractItemView — QHeaderView
```

Формат конструктора класса QHeaderView:

```
QHeaderView(<Ориентация>[, parent=None])
```

В качестве ориентации указываются элементы `Horizontal` (горизонтальная) или `Vertical` (вертикальная) перечисления `Orientation` из модуля `QtCore.Qt`.

Класс `QHeaderView` наследует все методы и сигналы класса `QAbstractItemView` (см. *разд. 23.5.1*) и дополнительно определяет следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qheaderview.html**):

- `count()` — возвращает количество секций в заголовке. Получить количество секций можно также с помощью функции `len()`;
- `setDefaultSectionSize(<Размер>)` — задает размер секций по умолчанию;
- `defaultSectionSize()` — возвращает размер секций по умолчанию;
- `setMinimumSectionSize(<Размер>)` — задает минимальный размер секций;
- `minimumSectionSize()` — возвращает минимальный размер секций;
- `setMaximumSectionSize(<Размер>)` — задает максимальный размер секций;
- `maximumSectionSize()` — возвращает максимальный размер секций;
- `resizeSection(<Индекс>, <Размер>)` — задает новый размер у секции с указанным индексом;
- `sectionSize(<Индекс>)` — возвращает размер секции с указанным индексом;
- `setSectionResizeMode(<Режим>)` — задает режим изменения размеров у всех секций. В качестве параметра могут быть указаны следующие элементы перечисления `ResizeMode` из класса `QHeaderView`:
  - `Interactive` — размер может быть изменен пользователем или программно;
  - `Stretch` — секции равномерно распределяют свободное пространство между собой. Размер не может быть изменен ни пользователем, ни программно;
  - `Fixed` — размер может быть изменен только программно;
  - `ResizeToContents` — размер определяется автоматически по содержимому секции. Размер не может быть изменен ни пользователем, ни программно;
- `setSectionResizeMode(<Индекс>, <Режим>)` — задает режим изменения размеров у секции с указанным индексом;
- `setStretchLastSection(<Флаг>)` — если в качестве параметра указано значение `True`, последняя секция будет занимать все оставшееся свободное пространство;
- `setCascadingSectionResizes(<Флаг>)` — если в качестве параметра указано значение `True`, изменение размеров одной секции может привести к изменению размеров других секций;
- `setSectionHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, секция с индексом, указанным в первом параметре, будет скрыта. Значение `False` отображает секцию;
- `hideSection(<Индекс>)` — скрывает секцию с указанным индексом;
- `showSection(<Индекс>)` — отображает секцию с указанным индексом;
- `isSectionHidden(<Индекс>)` — возвращает значение `True`, если секция с указанным индексом скрыта, и `False` — в противном случае;
- `sectionsHidden()` — возвращает значение `True`, если существует хотя бы одна скрытая секция, и `False` — в противном случае;

- `hiddenSectionCount()` — возвращает количество скрытых секций;
- `setDefaultAlignment(<Выравнивание>)` — задает выравнивание текста внутри заголовков в виде элемента перечисления `AlignmentFlag` из модуля `QtCore.Qt` (см. *разд. 21.2*);
- `setHighlightSections(<Флаг>)` — если в качестве параметра указано значение `True`, то текст заголовка текущей секции будет выделен;
- `setSectionsClickable(<Флаг>)` — если в качестве параметра указано значение `True`, заголовок будет реагировать на щелчок мышью, при этом выделяя все элементы секции;
- `setSectionsMovable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может перемещать секции с помощью мыши;
- `sectionsMovable()` — возвращает значение `True`, если пользователь может перемещать секции с помощью мыши, и `False` — в противном случае;
- `moveSection(<Откуда>, <Куда>)` — позволяет переместить секцию. В параметрах указываются визуальные индексы;
- `swapSections(<Секция 1>, <Секция 2>)` — меняет две секции местами. В параметрах указываются визуальные индексы;
- `visualIndex(<Логический индекс>)` — преобразует логический индекс (первоначальный порядок следования) в визуальный (отображаемый в настоящее время порядок следования). Если преобразование прошло неудачно, возвращается значение `-1`;
- `logicalIndex(<Визуальный индекс>)` — преобразует визуальный индекс (отображаемый в настоящее время порядок следования) в логический (первоначальный порядок следования). Если преобразование прошло неудачно, возвращается значение `-1`;
- `saveState()` — возвращает объект класса `QByteArray` с текущими размерами и положением секций;
- `restoreState(<Объект QByteArray>)` — восстанавливает размеры и положение секций на основе заданного объекта класса `QByteArray`, возвращаемого методом `saveState()`.

Класс `QHeaderView` поддерживает следующие сигналы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qheaderview.html**):

- `sectionPressed(<Логический индекс>)` — генерируется при нажатии левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен целочисленный логический индекс секции;
- `sectionClicked(<Логический индекс>)` — генерируется при нажатии и отпускании левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен целочисленный логический индекс секции;
- `sectionDoubleClicked(<Логический индекс>)` — генерируется при двойном щелчке мышью на заголовке секции. Внутри обработчика через параметр доступен целочисленный логический индекс секции;
- `sectionMoved(<Логический индекс>, <Старый визуальный индекс>, <Новый визуальный индекс>)` — генерируется при изменении положения секции. Все параметры целочисленные;
- `sectionResized(<Логический индекс>, <Старый размер>, <Новый размер>)` — генерируется непрерывно при изменении размера секции. Все параметры целочисленные.

23.6. Управление выделением элементов

Класс `QItemSelectionModel`, определенный в модуле `QtCore`, реализует модель, позволяющую централизованно управлять выделением сразу в нескольких представлениях. Установить модель выделения позволяет метод `setSelectionModel()` класса `QAbstractItemView`, а получить ссылку на модель можно с помощью метода `selectionModel()`. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом представлении. Иерархия наследования выглядит так:

```
QObject — QItemSelectionModel
```

Форматы конструктора класса `QItemSelectionModel`:

```
QItemSelectionModel([<Модель>])
QItemSelectionModel(<Модель>, <Родитель>)
```

Класс `QItemSelectionModel` поддерживает следующие полезные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qitemselectionmodel.html**):

- `hasSelection()` — возвращает значение `True`, если существует выделенный элемент, и `False` — в противном случае;
- `isSelected(<Индекс QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом выделен, и `False` — в противном случае;
- `isRowSelected(<Индекс>[, <Индекс родителя QModelIndex>])` — возвращает значение `True`, если строка с указанными индексом и индексом родителя выделена, и `False` — в противном случае. Если индекс родителя не указан, проверяется строка верхнего уровня иерархии;
- `isColumnSelected(<Индекс>[, <Индекс родителя QModelIndex>])` — возвращает значение `True`, если столбец с указанными индексом и индексом родителя выделен, и `False` — в противном случае. Если индекс родителя не указан, проверяется столбец верхнего уровня иерархии;
- `rowIntersectsSelection(<Индекс>[, <Индекс родителя QModelIndex>])` — возвращает значение `True`, если строка с указанными индексом и индексом родителя содержит выделенный элемент, и `False` — в противном случае. Если индекс родителя не указан, проверяется строка верхнего уровня иерархии;
- `columnIntersectsSelection(<Индекс>[, <Индекс родителя QModelIndex>])` — возвращает значение `True`, если столбец с указанными индексом и индексом родителя содержит выделенный элемент, и `False` — в противном случае. Если индекс родителя не указан, проверяется столбец верхнего уровня иерархии;
- `selectedIndexes()` — возвращает список индексов (объектов класса `QModelIndex`) выделенных элементов или пустой список, если выделенных элементов нет;
- `selectedRows([<Индекс столбца>=0])` — возвращает список индексов (объектов класса `QModelIndex`) выделенных элементов из указанного столбца. Элемент попадет в список только в том случае, если строка выделена полностью;
- `selectedColumns([<Индекс строки>=0])` — возвращает список индексов (объектов класса `QModelIndex`) выделенных элементов из указанной строки. Элемент попадет в список только в том случае, если столбец выделен полностью;

- `selection()` — возвращает ссылку на объект класса `QItemSelection`;
- `select(<Индекс QModelIndex>, <Режим>)` — изменяет выделение элемента с указанным индексом. Во втором параметре указываются следующие элементы (или их комбинация через оператор `|`) перечисления `SelectionFlag` из класса `QItemSelectionModel`:
  - `NoUpdate` — без изменений;
  - `Clear` — снимает выделение всех элементов;
  - `Select` — выделяет элемент;
  - `Deselect` — снимает выделение с элемента;
  - `Toggle` — выделяет элемент, если он не выделен, или снимает выделение, если элемент был выделен;
  - `Current` — обновляет выделение текущего элемента;
  - `Rows` — индекс будет расширен так, чтобы охватить всю строку;
  - `Columns` — индекс будет расширен так, чтобы охватить весь столбец;
  - `SelectCurrent` — комбинация `Select | Current`;
  - `ToggleCurrent` — комбинация `Toggle | Current`;
  - `ClearAndSelect` — комбинация `Clear | Select`.

  Метод является слотом;
- `select(<Выделение QItemSelection>, <Режим>)` — задает новое выделение элементов. Метод является слотом;
- `setCurrentIndex(<Индекс QModelIndex>, <Режим>)` — делает элемент с заданным индексом текущим и изменяет режим выделения. Метод является слотом;
- `currentIndex()` — возвращает индекс (объект класса `QModelIndex`) текущего элемента;
- `clearSelection()` — снимает все выделения. Метод является слотом.

Класс `QItemSelectionModel` поддерживает следующие сигналы:

- `currentChanged(<Предыдущий индекс>, <Новый индекс>)` — генерируется при выделении другого элемента. Внутри обработчика через первый параметр доступен индекс ранее выделенного элемента, а через второй — индекс вновь выделенного элемента (объекты класса `QModelIndex`);
- `currentRowChanged(<Предыдущий индекс>, <Новый индекс>)` — генерируется при выделении элемента из другой строки. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй — индекс нового элемента (объекты класса `QModelIndex`);
- `currentColumnChanged(<Предыдущий индекс>, <Новый индекс>)` — генерируется при выделении элемента из другого столбца. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй — индекс нового элемента (объекты класса `QModelIndex`);
- `selectionChanged(<Предыдущее выделение>, <Новое выделение>)` — генерируется при изменении выделения. Внутри обработчика через первый параметр доступно предыдущее выделение, а через второй — новое выделение (объекты класса `QItemSelection`).

22.7. Промежуточные модели

Одну модель можно установить в нескольких представлениях. При этом изменение порядка следования элементов в одном представлении повлечет за собой изменение порядка следования элементов в другом. Чтобы предотвратить изменение порядка следования элементов в базовой модели, следует создать промежуточную модель с помощью класса `QSortFilterProxyModel` из модуля `QtCore` и установить ее в представлении. Иерархия наследования класса `QSortFilterProxyModel` выглядит так:

```
QObject — QAbstractItemModel — QAbstractProxyModel — QSortFilterProxyModel
```

Формат конструктора класса `QSortFilterProxyModel`:

```
QSortFilterProxyModel([parent=None])
```

Класс `QSortFilterProxyModel` наследует следующие методы из класса `QAbstractProxyModel` (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qabstractproxymodel.html**):

- `setSourceModel(<Модель>)` — устанавливает базовую модель;
- `sourceModel()` — возвращает ссылку на базовую модель.

Класс `QSortFilterProxyModel` поддерживает основные методы обычных моделей и дополнительно определяет следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qsortfilterproxymodel.html**):

- `sort(<Индекс столбца>[, order=SortOrder.AscendingOrder])` — производит сортировку. Если во втором параметре указан элемент `AscendingOrder` перечисления `SortOrder` из модуля `QtCore.Qt`, сортировка производится в прямом порядке, а если элемент `DescendingOrder` того же перечисления — в обратном. Если в параметре `<Индекс столбца>` указать значение `-1`, будет использован порядок следования элементов из базовой модели.

  > ***Примечание***
  >
  > Чтобы включить сортировку столбцов пользователем, следует передать значение `True` в метод `setSortingEnabled()` объекта представления.

- `setSortRole(<Роль>)` — задает роль (см. *разд. 23.3*), по которой производится сортировка. По умолчанию сортировка производится по роли `ItemDataRole.DisplayRole`;
- `setSortCaseSensitivity(<Режим>)` — если в качестве параметра указать элемент `CaseInsensitive` перечисления `CaseSensitivity` из модуля `QtCore.Qt`, при сортировке регистр символов учитываться не будет, а если элемент `CaseSensitive` того же перечисления — то будет;
- `setSortLocaleAware(<Флаг>)` — если в качестве параметра указать значение `True`, при сортировке будут учитываться настройки локали;
- `setFilterFixedString(<Фрагмент>)` — выбор из модели элементов, которые содержат заданный фрагмент. Если указать пустую строку, в результат попадут все строки из базовой модели. Метод является слотом;
- `setFilterRegularExpression()` — выбор из модели элементов, соответствующих указанному регулярному выражению. Если указать пустую строку, в результат попадут все строки из базовой модели. Форматы метода:

```
setFilterRegularExpression(<Регулярное выражение QRegularExpression>)
setFilterRegularExpression(<Строка с регулярным выражением>)
```

Второй формат метода является слотом;

- `setFilterWildcard(<Шаблон>)` — выбор из модели элементов, соответствующих указанной строке, которая может содержать подстановочные знаки:
  - `?` — один любой символ;
  - `*` — нуль или более любых символов;
  - `[...]` — диапазон значений.

  Остальные символы трактуются как есть. Если в качестве параметра указать пустую строку, в результат попадут все элементы из базовой модели. Метод является слотом;
- `setFilterKeyColumn(<Индекс>)` — задает индекс столбца, по которому будет производиться фильтрация. Если в качестве параметра указать значение `-1`, будут просматриваться элементы во всех столбцах. По умолчанию фильтрация производится по первому столбцу;
- `setFilterRole(<Роль>)` — задает роль (см. *разд. 23.3*), по которой производится фильтрация. По умолчанию фильтрация производится по роли `ItemDataRole.DisplayRole`;
- `setFilterCaseSensitivity(<Режим>)` — если в качестве параметра указать элемент `CaseInsensitive` перечисления `CaseSensitivity` из модуля `QtCore.Qt`, при фильтрации регистр символов учитываться не будет, а если элемент `CaseSensitive` того же перечисления — то будет;
- `setRecursiveFilteringEnabled(<Флаг>)` — если в качестве параметра указано `True`, также будет выполняться фильтрация вложенных элементов, и у всех вложенных элементов, соответствующих фильтру, будут видны родители. Если `False`, дочерние элементы фильтроваться не будут (поведение по умолчанию);
- `setAutoAcceptChildRows(<Флаг>)` — если в качестве параметра указано `True`, у родительских элементов будут показываться все дочерние элементы, даже не удовлетворяющие заданному фильтру. Если `False`, будут показываться лишь дочерние элементы, удовлетворяющие фильтру (поведение по умолчанию);
- `setDynamicSortFilter(<Флаг>)` — если в качестве параметра указано значение `False`, при изменении базовой модели не будет производиться повторная сортировка или фильтрация.

23.8. Использование делегатов

Все три представления, рассмотренные в *разд. 23.5*, дают возможность редактирования текста их элементов. Например, в таблице (класс `QTableView`) мы можем дважды щелкнуть мышью на любом элементе, после чего в нем появится поле ввода. Введем в это поле новый текст и нажмем клавишу <Enter> для подтверждения ввода или <Esc> — для отмены.

За редактирование данных в представлении отвечает особый класс, называемый *делегатом*. Он создает компонент, в котором будет выполняться редактирование значения (*редактор*), задает его параметры, заносит в него само редактируемое значение, а по окончании редактирования переносит его назад, в модель.

По умолчанию в качестве делегата используется класс `QItemDelegate` из модуля `QtWidgets`. А в качестве компонента-редактора применяется однострочное поле ввода (класс `QLineEdit`, рассмотренный в *главе 22*).

Если мы хотим использовать для редактирования значения в каком-либо столбце или строке другой редактор (например, многострочное поле ввода, поле ввода даты или целого числа), мы создадим другой делегат и назначим его представлению. Класс, представляющий делегат, должен быть унаследован от класса `QStyledItemDelegate`.

Иерархия наследования классов `QItemDelegate` и `QStyledItemDelegate`:

```
QObject - QAbstractItemDelegate - QItemDelegate
QObject - QAbstractItemDelegate - QStyledItemDelegate
```

В новом классе-делегате следует переопределить следующие методы:

- `createEditor()` — создает компонент, который будет использоваться для редактирования данных, и задает его параметры. Формат метода:

  ```
  createEditor(self, <Родитель>, <Настройки>, <Индекс>)
  ```

  Вторым параметром передается ссылка на компонент-представление, который станет родителем создаваемого редактора (список, таблица или иерархический список). Третьим параметром передается объект класса `QStyleOptionViewItem`, хранящий дополнительные настройки делегата, четвертым — индекс текущего элемента модели в виде объекта класса `QModelIndex`.

  Метод `createEditor()` должен создать компонент-редактор, задать у него в качестве родителя компонент-представление (он передается вторым параметром) и вернуть созданный компонент в качестве результата.

  Чтобы отказаться от использования собственного делегата и указать представлению использовать делегат по умолчанию, в методе `createEditor()` следует вернуть значение `None`;

- `setEditorData()` — заносит в компонент-редактор, созданный в методе `createEditor()`, данные из текущего элемента модели, тем самым подготавливая редактор для редактирования этих данных. Формат метода:

  ```
  setEditorData(self, <Редактор>, <Индекс>)
  ```

  Вторым параметром передается компонент-редактор, а третьим — индекс текущего элемента модели в виде объекта класса `QModelIndex`;

- `updateEditorGeometry()` — задает размеры редактора соответственно размерам области, отведенной под него в компоненте-представлении. Формат:

  ```
  updateEditorGeometry(self, <Редактор>, <Настройки>, <Индекс>)
  ```

  Вторым параметром передается ссылка на компонент-редактор, третьим — ссылка на объект класса `QStyleOptionViewItem`, хранящий настройки делегата, четвертым — индекс текущего элемента модели, представленный объектом класса `QModelIndex`.

  Размеры отведенной под редактор области можно получить из атрибута `rect` объекта класса `QStyleOptionViewItem`, переданного третьим параметром (полное описание класса `QStyleOptionViewItem` приведено на странице **https://doc.qt.io/qt-6/qstyleoptionviewitem.html**, а описание класса `QStyleOption`, от которого он порожден, — на странице **https://doc.qt.io/qt-6/qstyleoption.html**);

- `setModelData()` — по окончании редактирования переносит значение из редактора в текущий элемент модели. Формат:

  ```
  setModelData(self, <Редактор>, <Модель>, <Индекс>)
  ```

  Вторым параметром передается ссылка на компонент-редактор, третьим — ссылка на модель, четвертым — индекс текущего элемента модели в виде объекта класса `QModelIndex`.

Полное описание базового класса-делегата `QAbstractItemDelegate` можно найти на странице **https://doc.qt.io/qt-6/qabstractitemdelegate.html**, класса `QitemDelegate` — на странице **https://doc.qt.io/qt-6/qitemdelegate.html**, а класса `QStyledItemDelegate` — на странице **https://doc.qt.io/qt-6/qstyleditemdelegate.html**.

Для назначения делегатов представлению следует применять следующие методы, унаследованные от класса `QAbstractItemView`:

- `setItemDelegate(<Делегат>)` — назначает заданный делегат для всего представления;
- `setItemDelegateForColumn(<Индекс столбца>, <Делегат>)` — назначает заданный делегат для столбца представления с указанным индексом;
- `setItemDelegateForRow(<Индекс строки>, <Делегат>)` — назначает заданный делегат для строки представления с указанным индексом.

  Если в какой-либо ячейке представления действуют одновременно два делегата, заданные для столбца и для строки, будет использоваться делегат, заданный для строки.

В качестве примера рассмотрим небольшую складскую программу (листинг 23.4), позволяющую править количество каких-либо имеющихся на складе позиций с применением поля для ввода целочисленных значений (класс `QSpinBox`).

Листинг 23.4. Использование делегата

```
from PyQt6 import QtCore, QtWidgets, QtGui
import sys

# Создаем класс делегата
class SpinBoxDelegate(QtWidgets.QStyledItemDelegate):
    def createEditor(self, parent, options, index):
        # Создаем компонент-редактор, используемый для правки значений
        # количества позиций
        editor = QtWidgets.QSpinBox(parent)
        editor.setFrame(False)
        editor.setMinimum(0)
        editor.setSingleStep(1)
        return editor
    def setEditorData(self, editor, index):
        # Заносим в компонент-редактор значение количества
        value = int(index.model().data(index, QtCore.Qt.ItemDataRole.EditRole))
        editor.setValue(value)
    def updateEditorGeometry(self, editor, options, index):
        # Указываем размеры компонента-редактора
        editor.setGeometry(options.rect)
```

```
    def setModelData(self, editor, model, index):
        # Заносим исправленное значение количества в модель
        value = str(editor.value())
        model.setData(index, value, QtCore.Qt.ItemDataRole.EditRole);

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QTableView()
window.setWindowTitle("Использование делегата")
sti = QtGui.QStandardItemModel(parent=window)
lst1 = ['Флеш-диск', 'Бумага для принтера', 'Картридж для принтера']
lst2 = ["10", "3", "8"]
for row in range(0, 3):
    item1 = QtGui.QStandardItem(lst1[row])
    item2 = QtGui.QStandardItem(lst2[row])
    sti.appendRow([item1, item2])
sti.setHorizontalHeaderLabels(['Товар', 'Кол-во'])
window.setModel(sti)
# Назначаем делегат второму столбцу таблицы
window.setItemDelegateForColumn(1, SpinBoxDelegate())
window.setColumnWidth(0, 150)
window.resize(350, 150)
window.show()
sys.exit(app.exec())
```

Результат выполнения кода из листинга 23.4 показан на рис. 23.4.

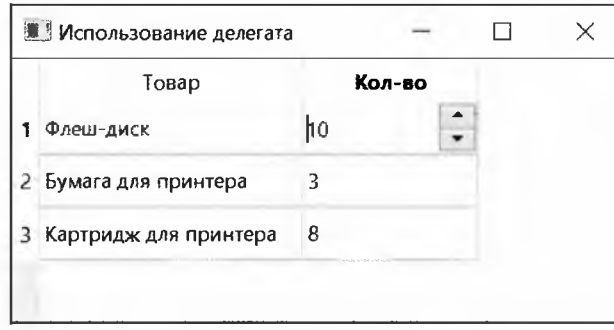
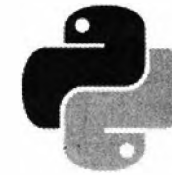


Рис. 23.4. Использование делегата



ГЛАВА 24

Работа с базами данных

PyQt включает в свой состав средства для работы с базами данных формата SQLite, MySQL, MariaDB, Oracle, PostgreSQL, IBM DB2 и др., не требующие установки никаких дополнительных Python-библиотек. Кроме того, поддерживается работа с любыми базами данных посредством платформы ODBC.

> ***Внимание!***
>
> Для успешного доступа к базам данных всех форматов, кроме SQLite и ODBC, требуется установить соответствующие клиенты.

Все классы, обеспечивающие работу с базами данных и рассмотренные в этой главе, определены в модуле `QtSql`.

24.1. Соединение с базой данных

За соединение с базой данных и обработку транзакций отвечает класс `QSqlDatabase`.

Чтобы установить соединение с базой, следует вызвать статический метод `addDatabase()` этого класса. Формат вызова:

```
addDatabase(<Формат базы данных>[, connectionName=""])
```

Формат базы данных указывается в виде одного из следующих строковых обозначений: `"QSQLITE"` (SQLite) `"QMYSQL"` или `"MARIADB"` (MySQL или MariaDB), `"QOCI"` (Oracle), `"QPSQL"` (PostgreSQL), `"QDB2"` (DB2) или `"QODBC"` (ODBC).

Вторым параметром можно задать имя соединения, что может оказаться полезным, если программа одновременно работает с несколькими базами. Если имя соединения не указано, устанавливаемое соединение будет помечено как используемое по умолчанию.

Метод `addDatabase()` возвращает объект класса `QSqlDatabase`, представляющий «пустое» соединение с базой данных. В этот объект следует занести параметры базы, с которой требуется соединиться, с помощью следующих методов класса `QSqlDatabase`:

- `setHostName(<Хост>)` — задает хост, на котором расположена база данных. Используется только для серверных баз данных наподобие MySQL;
- `setPort(<Номер порта>)` — задает номер TCP-порта, через который будет выполнено подключение к хосту. Используется только для серверных баз данных и лишь в том случае, если для соединения с базой задействуется порт, отличный от порта по умолчанию;

- `setDatabaseName(<Имя или путь к базе данных>)` — задает имя базы данных (для серверных баз), путь к ней (для локальных баз данных, таких как SQLite) или полный набор параметров подключения (если используется ODBC);
- `setUserName(<Имя>)` — задает имя пользователя для подключения к базе. Используется только для серверных баз данных;
- `setPassword(<Пароль>)` — задает пароль для подключения к базе. Используется только для серверных баз данных;
- `setConnectOptions(<Параметры>)` — задает набор дополнительных параметров для подключения к базе в виде строки. Набор поддерживаемых дополнительных параметров различен в зависимости от выбранного формата и приведен в документации по классу `QSqlDatabase`.

Для работы с базой предназначены следующие методы класса `QSqlDatabase`:

- `open()` — открывает базу данных. Возвращает `True`, если база была успешно открыта, и `False` — в противном случае;

> ***Внимание!***
>
> Перед созданием соединения с базой данных обязательно следует создать объект программы (объект класса `QApplication`). Если этого не сделать, PyQt не сможет загрузить драйвер указанного формата баз данных, и соединение не будет создано.
>
> Открываемая база данных уже должна существовать на локальном диске или сервере. Единственное исключение — база формата SQLite, которая в случае ее отсутствия будет создана автоматически.

- `open(<Имя>, <Пароль>)` — открывает базу данных под указанными именем пользователя и паролем. Возвращает `True`, если база была успешно открыта, и `False` — в противном случае;
- `isOpen()` — возвращает `True`, если база данных в настоящее время открыта, и `False` — в противном случае;
- `isOpenError()` — возвращает `True`, если при попытке открытия базы данных возникли ошибки, и `False` — в противном случае;
- `transaction()` — запускает транзакцию, если формат базы поддерживает таковые. Если же формат базы не поддерживает транзакции, то не делает ничего. Возвращает `True`, если транзакция была успешно запущена, и `False` — в противном случае;
- `commit()` — подтверждает транзакцию, если формат базы поддерживает таковые. Если же формат базы не поддерживает транзакции, то не делает ничего. Возвращает `True`, если транзакция была успешно подтверждена, и `False` — в противном случае;
- `rollback()` — отклоняет транзакцию, если формат базы поддерживает таковые. Если же формат базы не поддерживает транзакции, то не делает ничего. Возвращает `True`, если транзакция была успешно отклонена, и `False` — в противном случае;
- `lastError()` — возвращает сведения о последней возникшей при работе с базой ошибке в виде объекта класса `QSqlError`;
- `connectionName()` — возвращает строку с именем соединения с базой или пустую строку для соединения по умолчанию;
- `tables([type=TableType.Tables])` — возвращает список таблиц, хранящихся в базе. В параметре `type` можно указать тип таблиц в виде одного из элементов перечисления `TableType` из класса `QSql` или их комбинации через оператор `|`:

- Tables — обычные таблицы;
- SystemTables — служебные таблицы;
- Views — представления;
- AllTables — все упомянутое ранее;

- record(<Имя таблицы>) — возвращает сведения о структуре таблицы с переданным именем, представленные объектом класса QSqlRecord, или пустой объект этого класса, если таблицы с таким именем нет;
- primaryIndex(<Имя таблицы>) — возвращает сведения о ключевом индексе таблицы с переданным именем, представленные объектом класса QSqlIndex, или пустой объект этого класса, если таблицы с таким именем нет;
- close() — закрывает базу данных;
- isValid() — возвращает True, если текущий объект корректен, и False — в противном случае.

Также могут пригодиться следующие статические методы класса QSqlDatabase:

- contains([connectionName=""]) — возвращает True, если имеется соединение с базой данных с именем, указанным в параметре connectionName, и False — в противном случае;
- connectionNames() — возвращает список имен всех созданных соединений с базами данных. Соединение по умолчанию обозначается пустой строкой;
- database([connectionName=""][,][open=True]) — возвращает сведения о соединении с базой данных, имя которого указано в параметре connectionName, в виде объекта класса QSqlDatabase. Если в параметре open указано значение True, база данных будет открыта. Если такового соединения нет, возвращается некорректно сформированный объект класса QSqlDatabase;
- cloneDatabase(<Соединение QSqlDatabase>, <Имя соединения>) — создает копию указанного в первом параметре соединения с базой и дает ему имя, заданное во втором параметре. Возвращаемый результат — объект класса QSqlDatabase, представляющий созданную копию соединения;
- removeDatabase(<Имя соединения>) — удаляет соединение с указанным именем. Соединение по умолчанию обозначается пустой строкой;
- isDriverAvailable(<Формат>) — возвращает True, если указанное в виде строки обозначение формата баз данных поддерживается PyQt, и False — в противном случае:

```
>>> from PyQt6 import QtSql
>>> QtSql.QSqlDatabase.isDriverAvailable("QSQLITE")
    True
>>> QtSql.QSqlDatabase.isDriverAvailable("QMSSQL")
    False
```

- drivers() — возвращает список обозначений всех форматов баз данных, с которыми в текущий момент может работать PyQt. Список может быть неполным, если на компьютере не установлены клиенты каких-либо из баз данных, поддерживаемых библиотекой. Пример:

```
>>> QtSql.QSqlDatabase.drivers()
    ['QSQLITE', 'QODBC', 'QPSQL']
```

В листинге 24.1 показан код, выполняющий соединение с базами данных различных форматов и их открытие.

Листинг 24.1. Соединение с базами данных различных форматов

```
from PyQt6 import QtWidgets, QtSql
import sys
# Создаем объект программы, иначе поддержка баз данных не будет работать
app = QtWidgets.QApplication(sys.argv)

# Открываем базу данных SQLite, находящуюся в том же каталоге, что и файл
# с этой программой
con1 = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con1.setDatabaseName('data.sqlite')
con1.open()
con1.close()

# Открываем базу данных MySQL
con2 = QtSql.QSqlDatabase.addDatabase('QMYSQL')
con2.setHostName("somehost");
con2.setDatabaseName("somedb");
con2.setUserName("someuser");
con2.setPassword("password");
con2.open();
con2.close()

# Открываем базу данных Microsoft Access через ODBC
con3 = QtSql.QSqlDatabase.addDatabase("QODBC");
con3.setDatabaseName("DRIVER={Microsoft Access Driver (*.mdb)};" +
                     "FIL={MS Access};DBQ=c:/book/data.mdb");
con3.open()
con3.close()
```

Полное описание класса `QSqlDatabase` приведено на странице **https://doc.qt.io/qt-6/qsqldatabase.html**.

24.2. Получение сведений о структуре таблиц

PyQt позволяет получить сведения о структуре таблиц, хранящихся в базе: списки полей таблицы, параметры отдельного поля, индекса и ошибки, возникшей при работе с базой.

24.2.1. Получение сведений о таблицах

Сведения о структуре таблицы можно получить вызовом метода `record()` класса `QSqlDatabase` (см. *разд. 24.1*). Эти сведения представляются объектом класса `QSqlRecord`.

Для получения сведений о полях таблицы используются следующие методы этого класса:

- `count()` — возвращает количество полей в таблице;
- `fieldName(<Индекс поля>)` — возвращает имя поля, имеющее заданный индекс, или пустую строку, если индекс некорректен;

- `field(<Индекс поля>)` — возвращает сведения о поле (объект класса `QSqlField`), чей индекс задан в качестве параметра;
- `field(<Имя поля>)` — возвращает сведения о поле (объект класса `QSqlField`), чье имя задано в качестве параметра;
- `indexOf(<Имя поля>)` — возвращает индекс поля с указанным именем или `-1`, если такого поля нет. При поиске поля не учитывается регистр символов;
- `contains(<Имя поля>)` — возвращает `True`, если поле с указанным именем существует, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если в таблице нет полей, и `False` — в противном случае.

Полное описание класса `QSqlRecord` приведено на странице **https://doc.qt.io/qt-6/qsqlrecord.html**.

24.2.2. Получение сведений о полях таблиц

Сведения об отдельном поле таблицы возвращаются методами `field()` класса `QSqlRecord` (см. *разд. 24.2.1*). Они представляются объектом класса `QSqlField`, поддерживающим следующие методы:

- `name()` — возвращает имя поля;
- `typeID()` — возвращает объект класса `QMetaType` (модуль `QtCore`), описывающий тип поля.

  Класс `QMetaType` поддерживает метод `id()`, возвращающий целочисленный идентификатор типа поля. Идентификаторы наиболее часто применяемых типов применительно к Python приведены далее (полный их список можно найти по адресу **https://doc.qt.io/qt-6/qmetatype.html#Type-enum**):
  - 0 — тип определить не удалось;
  - 1 — логический;
  - 2, 4 или 32 — целочисленный;
  - 6 или 38 — вещественный;
  - 10 — строковый;
  - 14 — дата;
  - 15 — время;
  - 16 — временная отметка;
  - 3, 5 или 35 — положительное целое число;
  - 7 — строка из одного символа;
  - 12 — двоичные данные `QByteArray`;
- `length()` — возвращает максимальную длину поля в символах;
- `precision()` — возвращает количество знаков после запятой у полей, хранящих вещественные числа;
- `defaultValue()` — возвращает значение поля по умолчанию;
- `requiredStatus()` — возвращает признак, является ли поле обязательным к заполнению, в виде одного из элементов перечисления `RequiredStatus` из класса `QSqlField`:

- `Required` — поле является обязательным к заполнению;
- `Optional` — поле не является обязательным к заполнению;
- `Unknown` — определить признак обязательности заполнения поля не представляется возможным;

- `isAutoValue()` — возвращает `True`, если значение в поле заносится автоматически (что может быть, например, у поля автоинкремента), и `False` — в противном случае;
- `isReadOnly()` — возвращает `True`, если поле доступно только для чтения, и `False` — в противном случае.

Полное описание класса `QSqlField` приведено на странице **https://doc.qt.io/qt-6/qsqlfield.html**.

24.2.3. Получение сведений о ключевом индексе

Сведения о ключевом индексе, возвращаемые методом `primaryIndex()` класса `QSqlDatabase` (см. *разд. 24.2.1*), представляются объектом класса `QSqlIndex`. Он наследует все методы класса `QSqlRecord`, тем самым позволяя узнать, в частности, список полей, на основе которых создан индекс.

Также он определяет следующие методы:

- `name()` — возвращает имя индекса или пустую строку для ключевого индекса;
- `isDescending(<Номер поля>)` — возвращает `True`, если поле с указанным номером в индексе отсортировано по убыванию, и `False` — в противном случае.

Полное описание класса `QSqlIndex` приведено на странице **https://doc.qt.io/qt-6/qsqlindex.html**.

24.2.4. Получение сведений об ошибке

Сведения об ошибке, возникшей при работе с базой данных, представляются объектом класса `QSqlError`. Выяснить, что за ошибка произошла и каковы ее причины, позволят следующие методы этого класса:

- `type()` — возвращает код ошибки в виде одного из следующих элементов перечисления `ErrorType` из класса `QSqlError`:
  - `NoError` — никакой ошибки не возникло;
  - `ConnectionError` — ошибка соединения с базой данных;
  - `StatementError` — ошибка в коде SQL-запроса;
  - `TransactionError` — ошибка в обработке транзакции;
  - `UnknownError` — ошибка неустановленной природы.

  Если код ошибки не удается определить, возвращается `-1`;
- `text()` — возвращает полное текстовое описание ошибки (фактически — значения, возвращаемые методами `databaseText()` и `driverText()`, объединенные в одну строку);
- `databaseText()` — возвращает текстовое описание ошибки, сгенерированное базой данных;
- `driverText()` — возвращает текстовое описание ошибки, сгенерированное драйвером базы данных, который входит в состав PyQt;

- `nativeErrorCode()` — возвращает строковый код ошибки, специфический для выбранного формата баз данных;
- `isValid()` — возвращает `True`, если текущий объект корректен (описывает реально возникшую ошибку), и `False` — в противном случае.

Пример:

```
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
if con.open():
    # Работаем с базой данных
else:
    # Выводим текст описания ошибки
    print(con.lastError().text())
```

Полное описание класса `QSqlError` можно найти на странице **https://doc.qt.io/qt-6/qsqlerror.html**.

24.3. Выполнение SQL-запросов и получение их результатов

Класс `QSqlQuery` позволяет выполнять SQL-запросы любого назначения: создания таблиц, выборки, добавления, изменения, удаления и др.

> ***Примечание***
>
> Далее будут рассмотрены лишь наиболее часто используемые возможности класса `QSqlQuery`. Полное его описание приведено на странице **https://doc.qt.io/qt-6/qsqlquery.html**.

24.3.1. Выполнение запросов

Чтобы выполнить запрос к базе, сначала следует создать объект класса `QSqlQuery`. Для этого используется один из следующих форматов вызова его конструктора:

```
QSqlQuery([<SQL-код>][, db=QSqlDatabase()])
QSqlQuery(<Соединение с базой данных QSqlDatabase>)
QSqlQuery(<Объект класса QSqlQuery>)
```

Первый формат позволяет сразу задать SQL-код, который следует выполнить, и немедленно запустить его на исполнение. Необязательный параметр `db` задает соединение с базой данных, запрос к которой следует выполнить, — если он не указан, будет использоваться соединение по умолчанию.

Второй формат создает пустой запрос, не содержащий ни SQL-кода, ни каких-либо прочих параметров, и сразу задает соединение с указанной базой данных. Третий запрос создает копию запроса, переданного в параметре.

Для выполнения запросов используются следующие методы класса `QSqlQuery`:

- `exec(<SQL-код>)` — немедленно выполняет переданный в параметре SQL-код. Если последний был успешно выполнен, возвращает `True` и переводит запрос в активное состояние, в противном случае возвращает `False`. Пример использования этого метода показан в листинге 24.2.

Листинг 24.2. Использование метода exec()

```
from PyQt6 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
# Проверяем, есть ли в базе данных таблица good, и, если таковой нет,
# создаем ее SQL-командой CREATE TABLE
if 'good' not in con.tables():
    query = QtSql.QSqlQuery()
    query.exec("create table good(id integer primary key autoincrement, " +
               "goodname text, goodcount integer) ")
con.close()
```

Совет

Метод `exec()` следует использовать в тех случаях, если SQL-запрос не принимает параметров. В противном случае рекомендуется применять методы, рассмотренные далее.

- `prepare(<SQL-код>)` — подготавливает запрос с заданным SQL-кодом к выполнению. Применяется, если SQL-запрос содержит параметры. Параметры в коде запроса могут быть либо позиционными (в стиле ODBC — заданные вопросительными знаками), либо именованными (в стиле Oracle — задаются произвольными именами, предваренными знаком двоеточия). Метод возвращает `True`, если SQL-запрос был успешно подготовлен, и `False` — в противном случае;
- `exec()` — выполняет подготовленный ранее запрос. Возвращает `True`, если запрос был успешно выполнен, и `False` — в противном случае;
- `addBindValue(<Значение параметра>[, paramType=ParamType.In])` — задает значение очередного по счету позиционного параметра: первый вызов этого метода задает значение первого параметра, второй вызов — второго и т. д. Необязательный параметр `paramType` указывает тип параметра — здесь практически всегда используется элемент `In` перечисления `ParamType` из класса `QSql`, означающий, что этот параметр служит для занесения значения в запрос.

В листинге 24.3 приведен пример использования методов `prepare()`, `addBindValue()` и `exec()`.

Листинг 24.3. Использование методов prepare(), addBindValue() и exec()

```
from PyQt6 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
# Добавляем в только что созданную таблицу good запись,
# используя SQL-команду INSERT
```

```
query.prepare("insert into good values(null, ?, ?)")
query.addBindValue('Дискета')
query.addBindValue(10)
query.exec()
con.close()
```

- `bindValue()` — задает значение позиционного параметра с указанным порядковым номером (нумерация параметров начинается с нуля). Формат метода:

```
bindValue(<Номер параметра>, <Значение параметра>[, paramType=ParamType.In])
```

Пример применения метода показан в листинге 23.4.

Листинг 24.4. Использование метода `bindValue()` для задания значений позиционных параметров

```
from PyQt6 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
query.prepare("insert into good values(null, ?, ?)")
query.bindValue(0, 'Компакт-диск')
query.bindValue(1, 5)
query.exec()
con.close()
```

- `bindValue()` — задает значение именованного параметра с указанным именем. Формат метода:

```
bindValue(<Имя параметра>, <Значение параметра>[, paramType=ParamType.In])
```

Пример применения метода показан в листинге 24.5.

Листинг 24.5. Использование метода `bindValue()` для задания значений именованных параметров

```
from PyQt6 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
query.prepare("insert into good values(null, :name, :count)")
query.bindValue(':name', 'Флеш-накопитель')
query.bindValue(':count', 20)
query.exec()
con.close()
```

- `execBatch([mode=BatchExecutionMode.ValuesAsRows])` — если в вызове метода `addBindValue()` или `bindValue()` вторым параметром был указан список, выполнит подготовленный запрос.

  Необязательный параметр `mode` указывает, как будут интерпретироваться отдельные элементы заданного списка. В настоящее время в качестве его значения для всех форматов баз данных поддерживается лишь элемент `ValuesAsRows` перечисления `BatchExecutionMode` из класса `QSqlQuery`, указывающий, что подготовленный запрос должен быть выполнен столько раз, сколько элементов присутствует в списке, при этом на каждом выполнении запроса в его код подставляется очередной элемент списка.

  Метод возвращает `True`, если запрос был успешно выполнен, и `False` — в противном случае.

  Листинг 24.6 представляет пример добавления в таблицу сразу нескольких записей с применением метода `execBatch()`.

Листинг 24.6. Использование метода execBatch()

```
from PyQt6 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
query.prepare("insert into good values(null, :name, :count)")
lst1 = ['Бумага офисная', 'Фотобумага', 'Картридж']
lst2 = [15, 8, 3]
query.bindValue(':name', lst1)
query.bindValue(':count', lst2)
query.execBatch()
con.close()
```

- `setForwardOnly(<Флаг>)` — если передано значение `True`, по результату запроса можно будет перемещаться только «вперед», т. е. от начала к концу. Такой режим выполнения запроса существенно сокращает потребление системных ресурсов. Этот метод должен быть вызван перед выполнением запроса, который возвращает результат, например:

```
query.prepare("select * from good order by goodname")
query.setForwardOnly(True)
query.exec()
```

24.3.2. Обработка результатов выполнения запросов

Если был выполнен запрос на выборку данных (SQL-команда `SELECT`), следует получить результат его выполнения. Для этого мы используем методы класса `QSqlQuery`, описанные в этом разделе.

Запрос на выборку данных поддерживает особый внутренний указатель, указывающий на запись результата, содержимое которой в настоящее время доступно для получения. Однако сразу после выполнения запроса этот указатель хранит неопределенное значение, не иден-

тифицирующее никакую реальную запись. Поэтому перед собственно выборкой данных необходимо позиционировать этот указатель на нужную запись:

- `first()` — позиционирует указатель запроса на первую запись результата. Возвращает `True`, если позиционирование прошло успешно, и `False` — если результат пуст (не содержит записей);
- `next()` — позиционирует указатель запроса на следующую запись результата или на первую запись, если этот метод был вызван сразу после выполнения запроса. Возвращает `True`, если позиционирование прошло успешно, и `False` — если записей в результате больше нет, результат пуст или указатель ранее не был установлен ни на какую запись;
- `previous()` — позиционирует указатель запроса на предыдущую запись результата или на последнюю запись, если указатель в текущий момент находится за последней записью. Возвращает `True`, если позиционирование прошло успешно, и `False` — если записей в результате больше нет, результат пуст или указатель ранее не был установлен ни на какую запись;
- `last()` — позиционирует указатель запроса на последнюю запись результата. Возвращает `True`, если позиционирование прошло успешно, и `False` — если результат пуст;
- `seek(<Номер записи>[, relative=False])` — позиционирует указатель на запись с указанным номером (нумерация записей начинается с нуля). Если необязательным параметром `relative` передано значение `True`, то позиционирование выполняется относительно текущей записи: положительные значения вызывают смещение указателя «вперед» (к концу), а отрицательные — «назад» (к началу). Возвращает `True`, если позиционирование прошло успешно, и `False` — в противном случае;
- `isValid()` — возвращает `True`, если указатель указывает на какую-либо запись, и `False`, если он имеет неопределенное значение;
- `at()` — возвращает номер записи, на которую указывает указатель результата, или отрицательное число, если указатель имеет неопределенное значение;
- `size()` — возвращает количество записей, возвращенных в результате выполнения запроса, или `-1`, если этот запрос не выполнял выборку данных.

Для собственно выборки данных следует применять описанные далее методы:

- `value(<Индекс поля>)` — возвращает значение поля текущей записи с заданным индексом. Поля нумеруются в том порядке, в котором они присутствуют в таблице базы или в SQL-коде запроса;
- `value(<Имя поля>)` — возвращает значение поля текущей записи с заданным именем;
- `isNull(<Индекс поля>)` — возвращает `True`, если в поле с указанным индексом нет значения, и `False` — в противном случае;
- `isNull(<Имя поля>)` — возвращает `True`, если в поле с указанным именем нет значения, и `False` — в противном случае;
- `record()` — если внутренний указатель установлен на какую-либо запись, возвращает сведения об этой записи, в противном случае возвращаются сведения о самой таблице. Возвращаемым результатом является объект класса `QSqlRecord`;
- `recort(<Индекс записи>)` — возвращает сведения о записи с указанным индексом. Возвращаемым результатом является объект класса `QSqlRecord`;
- `isSelect()` — возвращает `True`, если был выполнен запрос на выборку данных, и `False`, если исполнялся запрос иного рода.

В листинге 24.7 приведен код, извлекающий данные из таблицы `good` созданной ранее базы данных и выводящий их на экран.

Листинг 24.7. Выборка данных из базы

```
from PyQt6 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
query.exec("select * from good order by goodname")
lst = []
if query.isActive():
    query.first()
    while query.isValid():
        lst.append(query.value('goodname') + ': ' +
                   str(query.value('goodcount')) + ' шт.')
        query.next()
    for p in lst: print(p)
con.close()
```

Результат выполнения этого кода:

```
Бумага офисная: 15 шт.
Дискета: 10 шт.
Картридж: 3 шт.
Компакт-диск: 5 шт.
Флеш-накопитель: 20 шт.
Фотобумага: 8 шт.
```

24.3.3. Очистка запроса

После первого выполнения метода `exec()` или `execBatch()` запрос переходит в активное состояние, и выполнить любую другую SQL-команду с его помощью станет невозможно.

Метод `isActive()` класса `QSqlQuery` возвращает `True`, если запрос находится в активном состоянии. Если же запрос неактивен, метод возвращает `False`.

Если один и тот же объект класса `QSqlQuery` планируется использовать для выполнения нескольких SQL-команд, перед выполнением новой команды следует сбросить его, переведя тем самым в неактивное состояние и освободив занимаемые им системные ресурсы. Это выполняется вызовом метода `clear()`:

```
query = QtSql.QSqlQuery()
query.exec("select * from good order by goodname")
# Обрабатываем результат запроса
query.clear()
query.exec("select count(*) as cnt from good")
# Работаем с новым запросом
```

24.3.4. Получение служебных сведений о запросе

Класс `QSqlQuery` также позволяет получить всевозможные служебные сведения о запросе. Для этого применяются следующие методы:

- `numRowsAffected()` — возвращает количество записей, обработанных в процессе выполнения запроса, или `-1`, если это количество не удается определить. Для запросов выборки данных возвращает `None` — в этом случае следует вызывать метод `size()`;
- `lastInsertId()` — возвращает идентификатор последней добавленной записи. Если запрос не добавлял записи или если формат базы данных не позволяет определить идентификатор последней добавленной записи, возвращает `None`;
- `lastError()` — возвращает объект класса `QSqlError`, описывающий последнюю возникшую в базе данных ошибку;
- `executedQuery()` — возвращает SQL-код последнего выполненного запроса или пустую строку, если никакой запрос еще не был выполнен. В возвращенном коде запроса вместо параметров будут подставлены указанные у них значения;
- `lastQuery()` — возвращает код последнего выполненного запроса или пустую строку, если никакой запрос еще не был выполнен. В возвращенном коде запроса вместо параметров *не* будут подставлены указанные у них значения;
- `boundValue(<Номер параметра>)` — возвращает значение параметра запроса с указанным номером;
- `boundValue(<Имя параметра>)` — возвращает значение параметра запроса с указанным именем;
- `boundValues()` — возвращает словарь, ключами элементов которого служат имена параметров, а значениями элементов — значения этих параметров. В случае позиционных параметров в качестве ключей будут использованы произвольные строки вида `:a` для первого параметра, `:bb` для второго и т. д.

24.4. Модели, связанные с данными

PyQt предоставляет два класса-модели, извлекающие данные из базы и позволяющие вывести их в списках или таблицах.

24.4.1. Модель, связанная с SQL-запросом

Если данные, извлеченные в результате выполнения SQL-запроса, *не* требуется редактировать, имеет смысл использовать класс `QSqlQueryModel`. Он представляет модель, связанную с SQL-запросом. Иерархия наследования этого класса:

```
QObject - QAbstractItemModel - QAbstractTableModel - QSqlQueryModel
```

Формат конструктора класса:

```
QSqlQueryModel([parent=None])
```

Класс `QSqlQueryModel` поддерживает следующие методы (здесь приведен их сокращенный список, а полный список методов этого класса доступен на страницах **https://doc.qt.io/qt-6/qsqlquerymodel.html** и **https://doc.qt.io/qt-6/qabstracttablemodel.html**):

- `setQuery(<Код запроса>[, db=QSqlDatabase()])` — задает SQL-код запроса у модели. Необязательный параметр `db` задает соединение с базой данных, запрос к которой следует выполнить, — если он не указан, будет использоваться соединение по умолчанию;
- `setQuery(<Запрос QSqlQuery>)` — задает запрос у модели;
- `query()` — возвращает запрос (объект класса `QSqlQuery`), заданный у модели;
- `record()` — возвращает объект класса `QSqlRecord`, хранящий сведения о структуре результата запроса;
- `record(<Индекс записи>)` — возвращает объект класса `QSqlRecord`, хранящий сведения о записи с указанным индексом;
- `lastError()` — возвращает объект объекта `QSqlError`, описывающий последнюю возникшую в базе данных ошибку;
- `index(<Строка>, <Столбец>[, parent=QModelIndex()])` — возвращает индекс (объект класса `QModelIndex`) элемента модели, находящегося на пересечении строки и столбца с указанными индексами. Необязательный параметр `parent` задает элемент верхнего уровня (объект класса `QModelIndex`) для искомого элемента — если таковой не задан, будет выполнен поиск элемента на самом верхнем уровне иерархии;
- `data(<Индекс QModelIndex>[, role=ItemDataRole.DisplayRole])` — возвращает данные, хранимые в указанной в параметре `role` роли элемента, на который ссылается заданный индекс;
- `rowCount([parent=QModelIndex()])` — возвращает количество элементов в модели. Необязательный параметр `parent` указывает элемент верхнего уровня (объект класса `QModelIndex`), при этом будет возвращено количество вложенных в него элементов. Если параметр не задан, возвращается количество элементов верхнего уровня иерархии;
- `setHeaderData()` — задает значение для указанной роли заголовка. Формат метода:

  ```
  SetHeaderData(<Индекс>, <Ориентация>, <Значение>[, role=ItemDataRole.EditRole])
  ```

  В первом параметре указывается индекс строки или столбца, а во втором — ориентация (элемент `Horizontal` или `Vertical` перечисления `Orientation` из модуля `QtCore.Qt`). Метод возвращает значение `True`, если операция успешно выполнена;
- `headerData(<Индекс>, <Ориентация>[, role=ItemDataRole.DisplayRole])` — возвращает значение, соответствующее указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором — ориентация.

Рассмотрим пример, выводящий данные из созданной нами ранее базы с помощью компонента таблицы (листинг 24.8).

Листинг 24.8. Использование модели, привязанной к SQL-запросу

```
from PyQt6 import QtCore, QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QTableView()
window.setWindowTitle("QSqlQueryModel")
# Устанавливаем соединение с базой данных
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
```

```
# Создаем модель
sqm = QtSql.QSqlQueryModel(parent=window)
sqm.setQuery('select * from good order by goodname')
# Задаем заголовки для столбцов модели
sqm.setHeaderData(1, QtCore.Qt.Orientation.Horizontal, 'Название')
sqm.setHeaderData(2, QtCore.Qt.Orientation.Horizontal, 'Кол-во')
# Задаем для таблицы только что созданную модель
window.setModel(sqm)
# Скрываем первый столбец, в котором выводится идентификатор
window.hideColumn(0)
window.setColumnWidth(1, 150)
window.setColumnWidth(2, 60)
window.resize(260, 160)
window.show()
sys.exit(app.exec())
```

24.4.2. Модель, связанная с таблицей

Если необходимо дать пользователю возможность редактировать данные, хранящиеся в базе, следует использовать класс `QSqlTableModel`. Он представляет модель, связанную непосредственно с указанной таблицей базы данных. Иерархия наследования:

```
QObject - QAbstractItemModel - QAbstractTableModel - QSqlQueryModel - QSqlTableModel
```

Формат конструктора класса:

```
QSqlTableModel([parent=None][, db=QSqlDatabase()])
```

Необязательный параметр `db` задает соединение с базой данных, запрос к которой следует выполнить, — если он не указан, будет использоваться соединение по умолчанию.

Класс `QSqlTableModel` наследует все методы из класса `QSqlQueryModel` (см. *разд. 24.4.1*) и в дополнение к ним определяет следующие наиболее полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-6/qsqltablemodel.html**):

- `setTable(<Имя таблицы>)` — задает таблицу, данные из которой будут представлены в модели. Отметим, что этот метод лишь выполняет получение из базы данных структуры указанной таблицы, но не загружает сами эти данные;
- `tableName()` — возвращает имя таблицы, заданной для модели;
- `setSort(<Индекс столбца>, <Порядок сортировки>)` — задает сортировку данных по столбцу с указанным индексом. Если вторым параметром передан элемент `AscendingOrder` перечисления `SortOrder` из модуля `QtCore.Qt`, сортировка будет производиться в прямом порядке, а если элемент `DescendingOrder` того же перечисления — в обратном;
- `setFilter(<Условие фильтрации>)` — задает условие для фильтрации данных в виде строки в формате, который применяется в SQL-команде `WHERE`;
- `filter()` — возвращает строку с фильтром, заданным для модели;
- `select()` — считывает в модель данные из заданной ранее таблицы с учетом указанных параметров сортировки и фильтрации. Возвращает `True`, если считывание данных прошло успешно, и `False` — в противном случае:

```
stm = QtSql.QSqlTableModel(parent=window)
stm.setTable('good')
```

```
stm.setSort(1, QtCore.Qt.SortOrder.DescendingOrder)
stm.setFilter('goodcount > 2')
stm.select()
```

Метод является слотом;

- `setEditStrategy(<Режим>)` — указывает режим редактирования данных в модели. В качестве параметра задается один из элементов перечисления `EditStrategy` из класса `QSqlTableModel`:
  - `OnFieldChange` — все изменения переносятся в базу данных немедленно;
  - `OnRowChange` — изменения переносятся в базу лишь после того, как пользователь перейдет на другую строку;
  - `OnManualSubmit` — изменения переносятся в базу только после вызова метода `submit()` или `submitAll()`;
- `insertRow(<Индекс>)` — добавляет пустую запись по индексу, заданному первым параметром. Возвращает значение `True`, если запись была успешно добавлена, и `False` — в противном случае;
- `insertRows(<Индекс>, <Количество>)` — добавляет указанное количество пустых записей по индексу, заданному первым параметром. Если указан индекс `-1`, записи добавляются в конец модели. Возвращает значение `True`, если записи были успешно добавлены, и `False` — в противном случае;
- `removeRow(<Индекс>)` — удаляет запись с указанным индексом. Возвращает значение `True`, если запись была успешно удалена, и `False` — в противном случае;
- `removeRows(<Индекс>, <Количество>)` — удаляет указанное количество записей, начиная с записи с указанным индексом. Возвращает значение `True`, если записи были успешно удалены, и `False` — в противном случае;

> ***Примечание***
>
> Нужно отметить, что после удаления записи вызовом метода `removeRow()` или `removeRows()` в модели останется пустая запись, реально не представляющая никакой записи из таблицы. Чтобы убрать ее, достаточно выполнить повторное считывание данных в модель вызовом метода `select()`.

- `insertRecord(<Индекс>, <Запись>)` — добавляет в модель новую запись по индексу, указанному первым параметром. Если задан отрицательный индекс, запись добавляется в конец модели. Добавляемая запись представляется объектом класса `QSqlRecord`, уже заполненным необходимыми данными. Возвращает `True`, если запись была успешно добавлена, и `False` — в противном случае;
- `setRecord(<Индекс>, <Запись>)` — заменяет запись по индексу, указанному первым параметром, другой записью, которая передается вторым параметром в виде объекта класса `QSqlRecord`, уже заполненного необходимыми данными. Возвращает `True`, если запись была успешно изменена, и `False` — в противном случае;
- `submit()` — переносит в базу данных изменения, сделанные в текущей записи, если был задан режим редактирования `OnManualSubmit`. Возвращает `True`, если изменения были успешно перенесены, и `False` — в противном случае. Метод является слотом;
- `submitAll()` — переносит в базу данных изменения, сделанные во всех записях, если был задан режим редактирования `OnManualSubmit`. Возвращает `True`, если изменения были успешно перенесены, и `False` — в противном случае. Метод является слотом;

- `revert()` — отменяет изменения, сделанные в текущей записи, если был задан режим редактирования `OnManualSubmit`. Возвращает `True`, если изменения были успешно отменены, и `False` — в противном случае. Метод является слотом;
- `revertRow(<Индекс записи>)` — отменяет изменения, сделанные в записи с заданным индексом, если был задан режим редактирования `OnManualSubmit`;
- `revertAll()` — отменяет изменения, сделанные во всех записях, если был задан режим редактирования `OnManualSubmit`. Возвращает `True`, если изменения были успешно отменены, и `False` — в противном случае. Метод является слотом;
- `selectRow(<Индекс записи>)` — обновляет содержимое записи с указанным индексом. Возвращает `True`, если запись была успешно обновлена, и `False` — в противном случае. Метод является слотом;
- `isDirty(<Индекс QModelIndex>)` — возвращает `True`, если запись с указанным индексом была изменена, но эти изменения еще не были перенесены в базу данных, и `False` — в противном случае;
- `isDirty()` — возвращает `True`, если хотя бы одна запись в модели была изменена, но эти изменения еще не были перенесены в базу данных, и `False` — в противном случае;
- `fieldIndex(<Имя поля>)` — возвращает индекс поля с указанным именем или `-1`, если такого поля нет;
- `primaryKey()` — возвращает сведения о ключевом индексе таблицы, представленные объектом класса `QSqlIndex`, или пустой объект этого класса, если таблица не содержит ключевого индекса.

Методы `insertRecord()` и `setRecord()`, предназначенные соответственно для добавления и изменения записи, принимают в качестве второго параметра объект класса `QSqlRecord`. Формат вызова конструктора этого класса:

```
QSqlRecord([<Объект класса QSqlRecord>])
```

Если в параметре указать объект класса `QSqlRecord`, будет создана его копия. Обычно при создании новой записи здесь указывают значение, возвращенное методом `record()` класса `QSqlDatabase` (оно хранит сведения о структуре таблицы и, следовательно, представляет пустую запись), а при правке существующей записи — значение, возвращенное методом `record()`, который унаследован классом `QSqlTableModel` от класса `QSqlQueryModel` (оно представляет запись, которую нужно отредактировать).

Класс `QSqlRecord`, в дополнение к методам, рассмотренным нами в *разд. 24.2.1*, поддерживает следующие методы:

- `value(<Индекс поля>)` — возвращает значение поля текущей записи с заданным индексом;
- `value(<Имя поля>)` — возвращает значение поля текущей записи с заданным именем;
- `setValue(<Индекс поля>, <Значение>)` — заносит в поле с указанным индексом новое значение;
- `setValue(<Имя поля>, <Значение>)` — заносит в поле с указанным именем новое значение;
- `isNull(<Индекс поля>)` — возвращает `True`, если в поле с указанным индексом нет значения, и `False` — в противном случае;

- `isNull(<Имя поля>)` — возвращает `True`, если в поле с указанным именем нет значения, и `False` — в противном случае;
- `setNull(<Индекс поля>)` — удаляет значение из поля с указанным индексом;
- `setNull(<Имя поля>)` — удаляет значение из поля с указанным именем;
- `clearValues()` — удаляет значения из всех полей записи;
- `setGenerated(<Индекс поля>, <Флаг>)` — если вторым параметром передано `False`, поле с указанным индексом помечается как неактуальное, и хранящееся в нем значение не будет перенесено в таблицу;
- `setGenerated(<Имя поля>, <Флаг>)` — если вторым параметром передано `False`, поле с указанным именем помечается как неактуальное, и хранящееся в нем значение не будет перенесено в таблицу;
- `isGenerated(<Индекс поля>)` — возвращает `False`, если поле с указанным индексом помечено как неактуальное, и `True` — в противном случае;
- `isGenerated(<Имя поля>)` — возвращает `False`, если поле с указанным именем помечено как неактуальное, и `True` — в противном случае.

Пример кода, добавляющего новую запись в модель:

```
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
stm = QtSql.QSqlTableModel()
stm.setTable('good')
stm.select()
rec = con.record('good')
rec.setValue('goodname', 'Коврик для мыши')
rec.setValue('goodcount', 2)
stm.insertRecord(-1, rec)
```

Пример кода, редактирующего существующую запись с индексом 3:

```
rec = stm.record(3)
rec.setValue('goodcount', 5)
stm.setRecord(3, rec)
```

Класс `QSqlTableModel` поддерживает такие сигналы:

- `primeInsert(<Индекс>, <Запись QSqlRecord>)` — генерируется при вызове метода `insertRows()` перед добавлением каждой из записей в модель. В первом параметре доступен целочисленный индекс добавляемой записи, а во втором — сама добавляемая запись, пустая, в которую можно занести какие-либо значения по умолчанию;
- `beforeInsert(<Запись QSqlRecord>)` — генерируется перед добавлением новой записи в таблицу. В параметре доступна добавляемая запись;
- `beforeUpdate(<Индекс>, <Запись QSqlRecord>)` — генерируется перед изменением записи в таблице. В параметрах доступны целочисленный индекс изменяемой записи и сама изменяемая запись;
- `beforeDelete(<Индекс>)` — генерируется перед удалением записи из таблицы. В параметре доступен целочисленный индекс удаляемой записи;

- `dataChanged(<Индекс QSqlRecord 1>, <Индекс QSqlRecord 2>, roles=[])` — генерируется при изменении данных в модели пользователем. Первым параметром передается индекс верхней левой из набора измененных записей, вторым — индекс правой нижней. Необязательный параметр `roles` хранит список ролей, данные которых изменились. Если указан пустой список, значит, изменились данные во всех ролях.

  Сигнал `dataChanged` — идеальное место для вызова методов `submit()` или `submitAll()` в случае, если у модели был задан режим редактирования `OnManualSubmit`. Как мы знаем, эти методы выполняют сохранение отредактированных данных в базе.

В листинге 24.9 представлен код складской программы, позволяющей править, добавлять и удалять записи нажатием соответствующих кнопок. А на рис. 24.1 можно увидеть саму программу в работе.

Листинг 24.9. Использование модели, привязанной к таблице

```
from PyQt6 import QtCore, QtWidgets, QtSql
import sys

def addRecord():
    # Вставляем пустую запись, в которую пользователь сможет
    # ввести нужные данные
    stm.insertRow(stm.rowCount())

def delRecord():
    # Удаляем запись из модели
    stm.removeRow(tv.currentIndex().row())
    # Выполняем повторное считывание данных в модель,
    # чтобы убрать пустую "мусорную" запись
    stm.select()

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QSqlTableModel")
# Устанавливаем соединение с базой данных
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
# Создаем модель
stm = QtSql.QSqlTableModel(parent=window)
stm.setTable('good')
stm.setSort(1, QtCore.Qt.SortOrder.AscendingOrder)
stm.select()
# Задаем заголовки для столбцов модели
stm.setHeaderData(1, QtCore.Qt.Orientation.Horizontal, 'Название')
stm.setHeaderData(2, QtCore.Qt.Orientation.Horizontal, 'Кол-во')
# Задаем для таблицы только что созданную модель
vbox = QtWidgets.QVBoxLayout()
tv = QtWidgets.QTableView()
tv.setModel(stm)
```

```
# Скрываем первый столбец, в котором выводится идентификатор
tv.hideColumn(0)
tv.setColumnWidth(1, 150)
tv.setColumnWidth(2, 60)
vbox.addWidget(tv)
btnAdd = QtWidgets.QPushButton("&Добавить запись")
btnAdd.clicked.connect(addRecord)
vbox.addWidget(btnAdd)
btnDel = QtWidgets.QPushButton("&Удалить запись")
btnDel.clicked.connect(delRecord)
vbox.addWidget(btnDel)
window.setLayout(vbox)
window.resize(370, 250)
window.show()
sys.exit(app.exec())
```

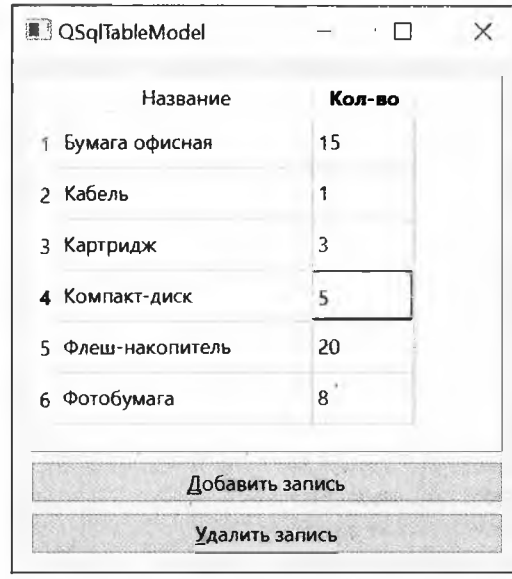


Рис. 24.1. Пример складской программы, использующей модель `QSqlTableModel`

24.4.3. Модель, поддерживающая межтабличные связи

Предположим, что мы решили расширить нашу простенькую складскую программу, введя разбиение товаров на категории. В базе данных `data.sqlite` мы создали таблицу `category` с полями `id` и `catname`, а в таблицу `good` добавили поле `category`, где будут храниться идентификаторы категорий.

Теперь попытаемся вывести содержимое таблицы `good` на экран с помощью модели `QSqlTableModel` и таблицы `QTableView`. И сразу увидим, что в колонке, где показывается содержимое поля `category`, выводятся числовые идентификаторы категорий (рис. 24.2). А нам хотелось бы видеть там наименования категорий вместо непонятной цифири.

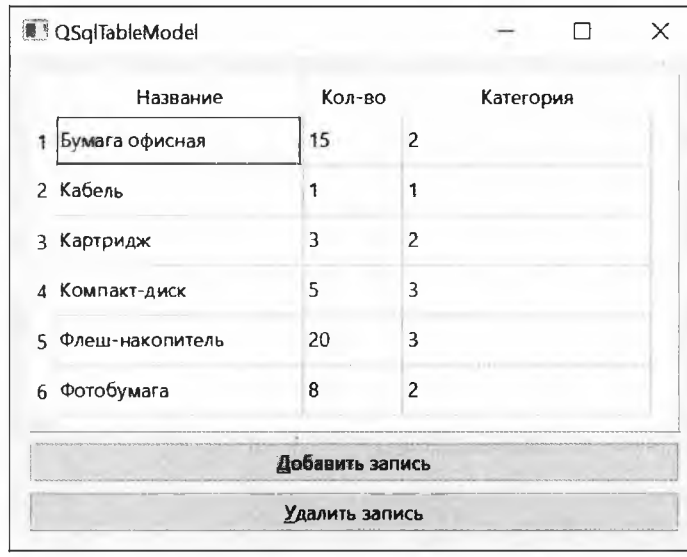


Рис. 24.2. Пример складской программы после доработки: вместо названий категорий выводятся их числовые идентификаторы

Сделать это поможет класс `QSqlRelationalTableModel`, реализующий модель вторичной таблицы, которая связывается с первичной таблицей. Иерархия наследования этого класса:

```
QObject - QAbstractItemModel - QAbstractTableModel - QSqlQueryModel -
                                QSqlTableModel - QSqlRelationalTableModel
```

Формат конструктора класса:

```
QSqlRelationalQueryModel([parent=None][, db=QSqlDatabase()])
```

Необязательный параметр `db` задает соединение с базой данных, запрос к которой следует выполнить, — если он не указан, будет использоваться соединение по умолчанию.

Класс `QSqlRelationalTableModel` наследует все методы класса `QSqlTableModel` (см. *разд. 24.4.2*) и в дополнение к ним определяет следующие полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-6/qsqlrelationaltablemodel.html**):

- `setRelation(<Индекс поля>, <Связь>)` — задает связь для поля внешнего ключа с указанным индексом. Сама связь представляется объектом класса `QSqlRelation`, рассматриваемым чуть позже;
- `setJoinMode(<Режим связывания>)` — задает режим связывания для всей модели. В качестве параметра указывается один из элементов перечисления `JoinMode` из класса `QSqlRelationalTableModel`:
  - `InnerJoin` — каждой записи вторичной таблицы должна соответствовать связанная с ней запись первичной таблицы. Используется по умолчанию;
  - `LeftJoin` — записи вторичной таблицы не обязательно должна соответствовать связанная запись первичной таблицы.

Класс `QSqlRelation` представляет саму межтабличную связь. Его конструктор имеет такой формат:

```
QSqlRelation(<Имя первичной таблицы>, <Имя поля первичного ключа>,
                                      <Имя поля, выводящегося на экран>)
```

Поля, чьи имена указываются во втором и третьем параметрах, относятся к первичной таблице.

В листинге 24.10 приведен код исправленной складской программы, а на рис. 24.3 показан ее интерфейс.

Листинг 24.10. Использование модели QSqlRelationalTableModel

```
from PyQt6 import QtCore, QtWidgets, QtSql
import sys

def addRecord():
    stm.insertRow(stm.rowCount())

def delRecord():
    stm.removeRow(tv.currentIndex().row())
    stm.select()

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QRelationalSqlTableModel")
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
stm = QtSql.QSqlRelationalTableModel(parent=window)
stm.setTable('good')
stm.setSort(1, QtCore.Qt.SortOrder.AscendingOrder)
# Задаем для поля категории связь с таблицей списка категорий
stm.setRelation(3, QtSql.QSqlRelation('category', 'id', 'catname'))
stm.select()
stm.setHeaderData(1, QtCore.Qt.Orientation.Horizontal, 'Название')
stm.setHeaderData(2, QtCore.Qt.Orientation.Horizontal, 'Кол-во')
stm.setHeaderData(3, QtCore.Qt.Orientation.Horizontal, 'Категория')
vbox = QtWidgets.QVBoxLayout()
tv = QtWidgets.QTableView()
tv.setModel(stm)
tv.hideColumn(0)
tv.setColumnWidth(1, 150)
tv.setColumnWidth(2, 60)
tv.setColumnWidth(3, 150)
vbox.addWidget(tv)
btnAdd = QtWidgets.QPushButton("&Добавить запись")
btnAdd.clicked.connect(addRecord)
vbox.addWidget(btnAdd)
btnDel = QtWidgets.QPushButton("&Удалить запись")
btnDel.clicked.connect(delRecord)
vbox.addWidget(btnDel)
window.setLayout(vbox)
window.resize(420, 250)
window.show()
sys.exit(app.exec())
```

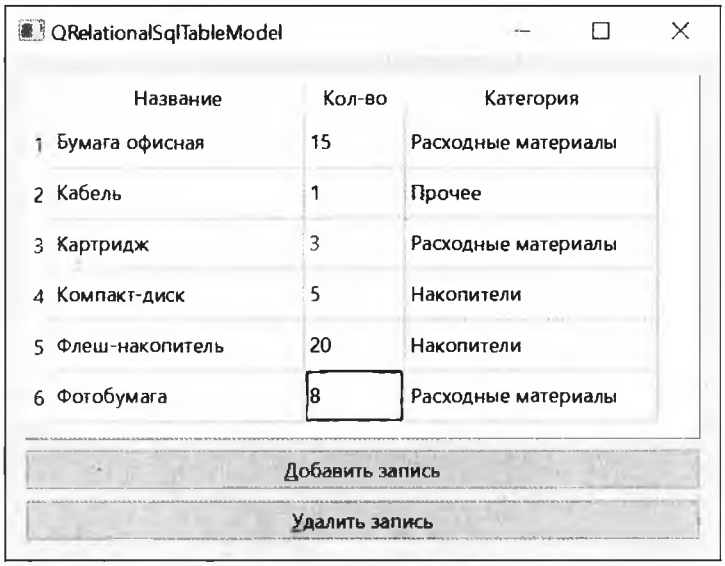


Рис. 24.3. Пример складской программы, использующей модель `QSqlRelationalTableModel`: на экран выводятся названия категорий

24.4.4. Использование связанных делегатов

Исправленная складская программа имеет существеннейший недостаток — как только мы решим добавить новую запись или исправить уже существующую, то столкнемся с тем, что все сделанные нами изменения не сохраняются. Почему?

Дело в том, что модель `QSqlRelationalTableModel` «не знает», как перевести введенное название категории в ее идентификатор, который и хранится в поле `category` таблицы `good`. Она лишь выполняет попытку занести строковое название категории в поле целочисленного типа, что вполне ожидаемо вызывает ошибку, и запись в таблице не сохраняется.

Исправить такое положение дел нам позволит особый делегат, называемый *связанным* (о делегатах рассказывалось в *разд. 23.8*). Он способен выполнить поиск в первичной таблице нужной записи, извлечь ее идентификатор и сохранить его в поле вторичной таблицы. А, кроме того, он представляет все доступные для занесения в поле значения, взятые из первичной таблицы, в виде раскрывающегося списка — очень удобно!

Функциональность связанного делегата реализует класс `QSqlRelationalDelegate`. Иерархия наследования:

```
QObject - QAbstractItemDelegate - QStyledItemDelegate - QSqlRelationalDelegate
```

Использовать связанный делегат очень просто — нужно лишь создать его объект, передав конструктору класса ссылку на компонент-представление (в нашем случае — таблицу), и вызвать у представления метод `setItemDelegate()`, `setItemDelegateForColumn()` или `setItemDelegateForRow()`, указав в нем только что созданный делегат.

Доработаем складскую программу, дав пользователю возможность выбирать категории товаров из списка. Для чего вставим в код листинга 24.10 одно новое выражение (в приведенном далее листинге 24.11 оно выделено полужирным шрифтом):

Листинг 24.11. Использование связанного делегата (фрагмент исправленного кода из листинга 24.10)

```
. . .
tv = QtWidgets.QTableView()
tv.setModel(stm)
tv.setItemDelegateForColumn(3, QtSql.QSqlRelationalDelegate(tv))
tv.hideColumn(0)
. . .
```

Интерфейс законченной программы показан на рис. 24.4.

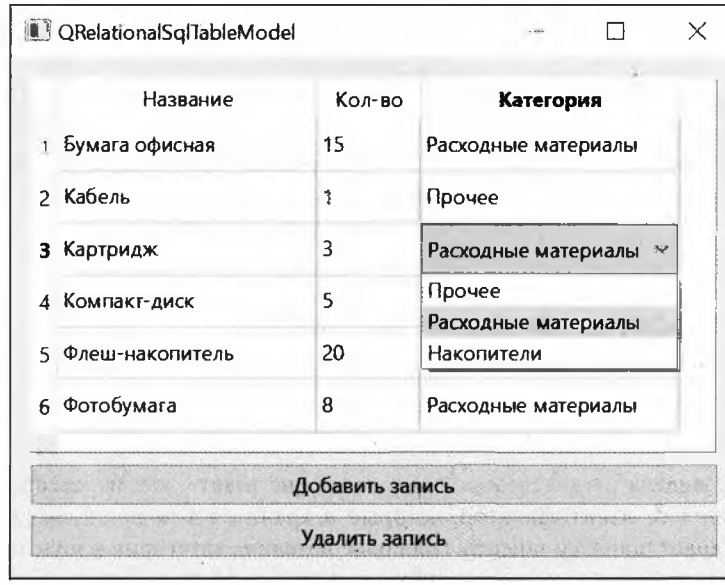


Рис. 24.4. Окончательный вариант складской программы, использующей связанный делегат

ГЛАВА 25

Работа с графикой

PyQt предоставляет класс `QPainter`, содержащий инструменты для рисования произвольной графики. Рисование выполняется на поверхности, реализуемой классом `QPaintDevice` или одним из его подклассов, в число которых входит и `QWidget` — базовый класс всех компонентов. Это дает возможность выводить произвольную графику на произвольном компоненте (включая обычное окно).

Когда у компонента возникает необходимость выполнить перерисовку (что может произойти, например, при первом выводе содержащего его окна, активизации этого окна, его разворачивании и т. п.), в его объекте вызывается специальный метод `paintEvent()`. Внутри этого метода и выполняется рисование на компоненте необходимой графики. Инициировать принудительную перерисовку компонента можно вызовом метода `repaint()` или `update()` (подробности — в *разд. 20.7.3*).

Все описанные в этой главе классы определены в модуле `QtGui`, если не указано обратное.

PyQt также позволяет работать с SVG-графикой и включает в свой состав поддержку технологии OpenGL, предназначенной для вывода двумерной и трехмерной графики. Рассмотрение этих инструментов выходит за рамки нашей книги, поэтому за подробной информацией о них обращайтесь к соответствующей документации.

25.1. Вспомогательные классы

25.1.1. Класс *QColor*: цвет

Класс `QColor` описывает цвет в цветовых моделях RGB, CMYK, HSV или HSL. Форматы конструктора класса `QColor`:

```
QColor()
QColor(<Красный>, <Зеленый>, <Синий>[, alpha=255])
QColor(<Строка>)
QColor(<Элемент перечисления GlobalColor>)
QColor(<Число>)
QColor(<Исходный цвет QColor>)
```

Первый формат создает невалидный объект. Проверить объект на валидность можно с помощью метода `isValid()`. Метод возвращает значение `True`, если объект является валидным, и `False` — в противном случае.

Второй формат принимает целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются числа от 0 до 255. Необязательный параметр alpha задает степень прозрачности цвета: значение 0 соответствует прозрачному цвету, а значение 255 — полностью непрозрачному.

Вот пример указания красного цвета:

```
red = QtGui.QColor(255, 0, 0)
```

В третьем формате цвет указывается в виде строки в форматах "#RGB", "#RRGGBB", "#AARRGGBB" (здесь AA обозначает степень прозрачности цвета), "#RRRGGGBBB", "#RRRRGGGGBBBB", "<Название цвета>" или "transparent" (прозрачный цвет):

```
red = QtGui.QColor("#f00")
darkBlue = QtGui.QColor("#000080")
semiTransparentDarkBlue = QtGui.QColor("#7F000080")
white = QtGui.QColor("white")
```

Получить список всех поддерживаемых названий цветов позволяет статический метод colorNames(). Проверить правильность строки с названием цвета можно с помощью статического метода isValidColor(<Строка>), который возвращает значение True, если строка является правильным наименованием цвета, и False — в противном случае:

```
print(QtGui.QColor.colorNames()) # ['aliceblue', 'antiquewhite', ...]
print(QtGui.QColor.isValidColor("lightcyan")) # True
```

В четвертом формате указываются следующие элементы перечисления GlobalColor из модуля QtCore.Qt: white, black, red, darkRed, green, darkGreen, blue, darkBlue, cyan, darkCyan, magenta, darkMagenta, yellow, darkYellow, gray, darkGray, lightGray, color0, color1 или transparent (прозрачный цвет). Элементы color0 (прозрачный цвет) и color1 (непрозрачный цвет) используются в двухцветных изображениях:

```
black = QtCore.Qt.GlobalColor.black
```

В пятом формате указывается целочисленное значение цвета:

```
darkBlue = QtGui.QColor(0x000080)
```

Шестой формат создает новый объект на основе указанного в параметре.

Задать или получить значения в цветовой модели RGB (Red, Green, Blue — красный, зеленый, синий) позволяют следующие методы:

- setNamedColor(<Строка>) — задает название цвета в виде строки в форматах "#RGB", "#RRGGBB", "#AARRGGBB", "#RRRGGGBBB", "#RRRRGGGGBBBB", "<Название цвета>" или "transparent" (прозрачный цвет);
- name([format=NameFormat.HexRgb]) — возвращает строковое представление цвета в виде строки в формате, заданном параметром format. Формат указывается в виде элементов перечисления NameFormat из класса QColor:
  - HexRgb — в формате "#RRGGBB";
  - HexArgb — в формате "#AARRGGBB";
- setRgb(<Красный>, <Зеленый>, <Синий>[, alpha=255]) — задает целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются числа от 0 до 255. Необязательный параметр alpha задает степень прозрачности цвета: значение 0 соответствует прозрачному цвету, а значение 255 — полностью непрозрачному;

- `setRgb(<Число>)` — задает целочисленное значение цвета;
- `setRgba(<Число>)` — задает целочисленное значение цвета со степенью прозрачности;
- `setRed(<Красный>)`, `setGreen(<Зеленый>)`, `setBlue(<Синий>)` и `setAlpha(<Прозрачность>)` — задают значения отдельных составляющих цвета. В качестве параметров указываются числа от 0 до 255;
- `fromRgb(<Красный>, <Зеленый>, <Синий>[, alpha=255])` — статический, возвращает объект класса `QColor` с указанными значениями. В качестве параметров указываются числа от `0` до `255`:

```
white = QtGui.QColor.fromRgb(255, 255, 255, 255)
```

- `fromRgb(<Число>)` и `fromRgba(<Число>)` — статические, возвращают объект класса `QColor` со значениями, соответствующими целым числам, которые указаны в параметрах:

```
white = QtGui.QColor.fromRgba(4294967295)
```

- `getRgb()` — возвращает кортеж из четырех целочисленных значений (`<Красный>`, `<Зеленый>`, `<Синий>`, `<Прозрачность>`);
- `red()`, `green()`, `blue()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета;
- `rgb()` и `rgba()` — возвращают целочисленное значение цвета;
- `setRgbF(<Красный>, <Зеленый>, <Синий>[, alpha=1.0])` — задает значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Необязательный параметр `alpha` задает степень прозрачности цвета: значение `0.0` соответствует прозрачному цвету, а значение `1.0` — полностью непрозрачному;
- `setRedF(<Красный>)`, `setGreenF(<Зеленый>)`, `setBlueF(<Синий>)` и `setAlphaF(<Прозрачность>)` — задают значения отдельных составляющих цвета. В качестве параметров указываются вещественные числа от `0.0` до `1.0`;
- `fromRgbF(<Красный>, <Зеленый>, <Синий>[, alpha=1.0])` — статический, возвращает объект класса `QColor` с указанными значениями. В качестве параметров указываются вещественные числа от `0.0` до `1.0`:

```
white = QtGui.QColor.fromRgbF(1.0, 1.0, 1.0, 1.0)
```

- `getRgbF()` — возвращает кортеж из четырех вещественных значений (`<Красный>`, `<Зеленый>`, `<Синий>`, `<Прозрачность>`);
- `redF()`, `greenF()`, `blueF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета;
- `lighter([factor=150])` — если параметр имеет значение больше 100, то возвращает новый объект с более светлым цветом, а если меньше 100 — то с более темным;
- `darker([factor=200])` — если параметр имеет значение больше 100, то возвращает новый объект с более темным цветом, а если меньше 100 — то с более светлым.

Задать или получить значения в цветовой модели CMYK (Cyan, Magenta, Yellow, Key — голубой, пурпурный, желтый, «ключевой» — он же черный) позволяют следующие методы:

- `setCmyk(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=255])` — задает целочисленные значения составляющих цвета модели CMYK. В качестве параметров указываются числа от 0 до 255. Необязательный параметр `alpha` задает степень прозрачности

цвета: значение 0 соответствует прозрачному цвету, а значение 255 — полностью непрозрачному;

- `fromCmyk(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=255])` — статический, возвращает объект класса `QColor` с указанными значениями. В качестве параметров указываются числа от 0 до 255:

```
white = QtGui.QColor.fromCmyk(0, 0, 0, 0, 255)
```

- `getCmyk()` — возвращает кортеж из пяти целочисленных значений (`<Голубой>`, `<Пурпурный>`, `<Желтый>`, `<Черный>`, `<Прозрачность>`);
- `cyan()`, `magenta()`, `yellow()`, `black()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета;
- `setCmykF(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=1.0])` — задает значения составляющих цвета модели CMYK. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Необязательный параметр `alpha` задает степень прозрачности цвета: значение `0.0` соответствует прозрачному цвету, а значение `1.0` — полностью непрозрачному;
- `fromCmykF(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=1.0])` — статический, возвращает объект класса `QColor` с указанными значениями. В качестве параметров указываются вещественные числа от `0.0` до `1.0`:

```
white = QtGui.QColor.fromCmykF(0.0, 0.0, 0.0, 0.0, 1.0)
```

- `getCmykF()` — возвращает кортеж из пяти вещественных значений (`<Голубой>`, `<Пурпурный>`, `<Желтый>`, `<Черный>`, `<Прозрачность>`);
- `cyanF()`, `magentaF()`, `yellowF()`, `blackF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета.

Задать или получить значения в цветовой модели HSV (Hue, Saturation, Value — оттенок, насыщенность, значение — оно же яркость) позволяют следующие методы:

- `setHsv(<Оттенок>, <Насыщенность>, <Значение>[, alpha=255])` — задает целочисленные значения составляющих цвета модели HSV. В первом параметре указывается число от 0 до 359, а в остальных параметрах — числа от 0 до 255;
- `fromHsv(<Оттенок>, <Насыщенность>, <Значение>[, alpha=255])` — статический, возвращает объект класса `QColor` с указанными значениями:

```
white = QtGui.QColor.fromHsv(0, 0, 255, 255)
```

- `getHsv()` — возвращает кортеж из четырех целочисленных значений (`<Оттенок>`, `<Насыщенность>`, `<Значение>`, `<Прозрачность>`);
- `hsvHue()`, `hsvSaturation()`, `value()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета;
- `setHsvF(<Оттенок>, <Насыщенность>, <Значение>[, alpha=1.0])` — задает значения составляющих цвета модели HSV. В качестве параметров указываются вещественные числа от `0.0` до `1.0`;
- `fromHsvF(<Оттенок>, <Насыщенность>, <Значение>[, alpha=1.0])` — статический, возвращает объект класса `QColor` с указанными значениями. В качестве параметров указываются вещественные числа от `0.0` до `1.0`:

```
white = QtGui.QColor.fromHsvF(0.0, 0.0, 1.0, 1.0)
```

- `getHsvF()` — возвращает кортеж из четырех вещественных значений (`<Оттенок>`, `<Насыщенность>`, `<Значение>`, `<Прозрачность>`);
- `hsvHueF()`, `hsvSaturationF()`, `valueF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета.

Цветовая модель HSL (Hue, Saturation, Lightness — оттенок, насыщенность, яркость) отличается от модели HSV только последней составляющей. Описание этой модели и полный перечень методов для установки и получения значений приведены в документации.

Для получения типа используемой модели и преобразования между моделями предназначены следующие методы:

- `spec()` — возвращает тип используемой модели в виде одного из следующих элементов перечисления `Spec` из класса `QColor`: `Invalid` (объект не валиден), `Rgb`, `Hsv`, `Cmyk` или `Hsl`;
- `convertTo(<Целевой тип модели>)` — преобразует тип модели. В качестве параметра указываются элементы перечисления `Spec` из класса `QColor` (было описано ранее). Метод возвращает новый объект цвета. Пример:

```
whiteHSV = QtGui.QColor.fromHsv(0, 0, 255)
whiteRGB = whiteHSV.convertTo(QtGui.QColor.Spec.Rgb)
```

  Также можно пользоваться методами `toRgb()`, `toCmyk()`, `toHsv()` и `toHsl()`, которые возвращают новые объекты цвета:

```
whiteHSV = QtGui.QColor.fromHsv(0, 0, 255)
whiteRGB = whiteHSV.toRgb()
```

25.1.2. Класс *QPen*: перо

Класс `QPen` описывает виртуальное перо, с помощью которого производится рисование точек, линий и контуров фигур. Форматы конструктора класса:

```
QPen()
QPen(<Цвет QColor>)
QPen(<Стиль линий>)
QPen(<Цвет QColor или кисть QBrush>, <Ширина>[, style=PenStyle.SolidLine][,
    cap=PenCapStyle.SquareCap][, join=PenJoinStyle.BevelJoin])
QPen(<Исходное перо QPen>)
```

Первый формат создает перо черного цвета с настройками по умолчанию. Второй формат задает только цвет пера. Третий формат позволяет указать стиль линий — в виде элемента перечисления `PenStyle` из модуля `QtCore.Qt`:

- `NoPen` — линия не выводится;
- `SolidLine` — сплошная линия;
- `DashLine` — штриховая линия;
- `DotLine` — точечная линия;
- `DashDotLine` — штрих и точка, штрих и точка и т. д.;
- `DashDotDotLine` — штрих и две точки, штрих и две точки и т. д.;
- `CustomDashLine` — пользовательский стиль.

Четвертый формат позволяет задать все характеристики пера за один раз. В параметре `style` указывается стиль линий. Необязательный параметр `cap` задает стиль концов линий в виде элемента перечисления `PenCapStyle` из модуля `QtCore.Qt`:

- `FlatCap` — квадратные концы линии. Длина линии не превышает указанных граничных точек;
- `SquareCap` — квадратные концы. Длина линии увеличивается с обоих концов на половину ширины линии;
- `RoundCap` — скругленные концы. Длина линии увеличивается с обоих концов на половину ширины линии.

Необязательный параметр `join` задает стиль соединения линий — в качестве значения указываются следующие элементы перечисления `PenJoinStyle` из модуля `QtCore.Qt`:

- `BevelJoin` — линии соединяются под острым углом;
- `MiterJoin` — острые углы срезаются на определенную величину;
- `RoundJoin` — скругленные углы;
- `SvgMiterJoin` — линии соединяются под острым углом, как определено в спецификации SVG 1.2 Tiny.

Последний формат создает новый объект на основе указанного в параметре.

Класс `QPen` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qpen.html**):

- `setColor(<Цвет QColor>)` — задает цвет линий;
- `setBrush(<Кисть QBrush>)` — задает кисть;
- `setWidth(<Ширина типа int>)` и `setWidthF(<Ширина типа float>)` — задают ширину линий целым числом или числом с плавающей точкой соответственно;
- `setStyle(<Стиль PenStyle>)` — задает стиль линий;
- `setCapStyle(<Стиль PenCapStyle>)` — задает стиль концов линий;
- `setJoinStyle(<Стиль PenJoinStyle>)` — задает стиль соединения линий;
- `setMiterLimit(<Величина срезания>)` — задает величину срезания острых углов, если указан режим соединения `MiterJoin`.

25.1.3. Класс *QBrush*: кисть

Класс `QBrush` описывает виртуальную кисть, с помощью которой производится заливка фигур. Форматы конструктора класса:

```
QBrush()
QBrush(<Цвет>[, style=BrushStyle.SolidPattern])
QBrush(<Стиль кисти>)
QBrush(<Градиент>)
QBrush(<Цвет>, <Изображение QPixmap>)
QBrush(<Изображение QPIxmap или QImage>)
QBrush(<Объект класса QBrush>)
```

Первый формат создает кисть с черной сплошной заливкой и стилем `NoBrush` (т. е. заливка фактически не рисуется).

Во втором и пятом форматах цвет может быть задан в виде объекта класса `QColor` или элемента перечисления `GlobalColor` из модуля `QtCore.Qt` (например: `black`).

Во втором и третьем форматах стиль кисти в параметрах `<Стиль кисти>` и `style` указывается в виде элемента перечисления `BrushStyle` из модуля `QtCore.Qt`: `NoBrush`, `SolidPattern`, `Dense1Pattern`, `Dense2Pattern`, `Dense3Pattern`, `Dense4Pattern`, `Dense5Pattern`, `Dense6Pattern`, `Dense7Pattern`, `CrossPattern` и др. Так, можно сделать цвет сплошным (`SolidPattern`) или имеющим текстуру (например, элемент `CrossPattern` задает текстуру в виде сетки).

Четвертый формат создает кисть с градиентной заливкой. В параметре `<Градиент>` указывается объект одного из классов, производных от `QGradient`: `QLinearGradient` (линейный градиент), `QConicalGradient` (конический градиент) или `QRadialGradient` (радиальный градиент). За подробной информацией по этим классам обращайтесь к документации.

Пятый формат создает кисть с заданным сплошным цветом и текстурой, создаваемой указанным изображением. Шестой формат создает кисть с черным цветом и текстурой, создаваемой указанным изображением.

Последний формат создает копию указанной кисти.

Класс `QBrush` поддерживает следующие полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-6/qbrush.html**):

- `setColor(<Цвет>)` — задает цвет кисти (объект класса `QColor` или элемент перечисления `GlobalColor` из модуля `QtCore.Qt`);
- `setStyle(<Стиль BrushStyle>)` — задает стиль кисти;
- `setTexture(<Изображение QPixmap>)` — устанавливает растровое изображение в качестве текстуры;
- `setTextureImage(<Изображение QImage>)` — устанавливает изображение в качестве текстуры.

25.1.4. Класс *QLine*: линия

Класс `QLine` из модуля `QtCore` описывает координаты линии. Форматы конструктора класса:

```
QLine()
QLine(<Начальная точка QPoint>, <Конечная точка QPoint>)
QLine(<X1>, <Y1>, <X2>, <Y2>)
```

Первый формат создает линию, имеющую неустановленные местоположение и размеры. В третьем формате координаты начальной и конечной точек указываются в виде целочисленных значений.

Класс `QLine` поддерживает следующие основные методы (полный их список приведен на странице **https://doc.qt.io/qt-6/qline.html**):

- `isNull()` — возвращает значение `True`, если начальная или конечная точка не установлены, и `False` — в противном случае;
- `setPoints(<Начальная точка QPoint>, <Конечная точка QPoint>)` — задает координаты начальной и конечной точек;
- `setLine(<X1>, <Y1>, <X2>, <Y2>)` — задает координаты начальной и конечной точек в виде целочисленных значений;
- `setP1(<Точка QPoint>)` — задает координаты начальной точки;

- `setP2(<Точка QPoint>)` — задает координаты конечной точки;
- `p1()` — возвращает координаты (объект класса `QPoint`) начальной точки;
- `p2()` — возвращает координаты (объект класса `QPoint`) конечной точки;
- `center()` — возвращает координаты (объект класса `QPoint`) центральной точки;
- `x1()`, `y1()`, `x2()` и `y2()` — возвращают значения отдельных составляющих координат начальной и конечной точек в виде целых чисел;
- `dx()` — возвращает горизонтальную составляющую вектора линии;
- `dy()` — возвращает вертикальную составляющую вектора линии.

> ***Примечание***
>
> Класс `QLine` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QLineF`.

24.1.5. Класс *QPolygon*: многоугольник

Класс `QPolygon` описывает координаты вершин многоугольника. Форматы конструктора класса:

```
QPolygon()
QPolygon(<Список с координатами вершин QPoint>)
QPolygon(<Прямоугольник QRect>[, closed=False])
QPolygon(<Количество вершин>)
QPolygon(<Исходный многоугольник QPolygon>)
```

Первый формат создает многоугольник, не имеющий вершин. Заполнить его координатами вершин можно с помощью оператора <<. Пример:

```
polygon = QtGui.QPolygon()
polygon << QtCore.QPoint(20, 50) << QtCore.QPoint(280, 50)
polygon << QtCore.QPoint(150, 280)
```

Во втором формате указывается список с координатами отдельных вершин:

```
polygon = QtGui.QPolygon([QtCore.QPoint(20, 50), QtCore.QPoint(280, 50),
                          QtCore.QPoint(150, 280)])
```

Третий формат создает многоугольник, имеющий вид заданного прямоугольника. Если параметр `closed` имеет значение `False`, то созданный многоугольник не будет замкнут, а если значение `True` — то будет замкнут.

В четвертом формате можно указать количество вершин, а затем задать координаты путем присваивания значения по индексу:

```
polygon = QtGui.QPolygon(3)
polygon[0] = QtCore.QPoint(20, 50)
polygon[1] = QtCore.QPoint(280, 50)
polygon[2] = QtCore.QPoint(150, 280)
```

Пятый конструктор создает копию заданного исходного объекта.

Класс `QPolygon` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qpolygon.html**):

- `setPoints()` — устанавливает координаты вершин. Ранее установленные значения удаляются. Форматы метода:

```
setPoints(<Список с координатами вершин QPoint>)
setPoints(<X1>, <Y1>[, . . ., <Xn>, <Yn>])
```

  Пример указания значений:

```
polygon = QtGui.QPolygon()
polygon.setPoints([20,50, 280,50, 150,280])
```

- `prepend(<Вершина QPoint>)` — добавляет новую вершину в начало объекта;
- `append(<Вершина QPoint>)` — добавляет новую вершину в конец объекта. Добавить вершину можно также с помощью операторов << и +=;
- `insert(<Индекс>, <Вершина QPoint>)` — добавляет новую вершину в позицию с указанным индексом;
- `setPoint()` — задает координаты вершины с указанным индексом. Форматы метода:

```
setPoint(<Индекс>, <Вершина QPoint>)
setPoint(<Индекс>, <X>, <Y>)
```

  Также можно задать координаты путем присваивания значения по индексу:

```
polygon = QtGui.QPolygon(3)
polygon.setPoint(0, QtCore.QPoint(20, 50))
polygon.setPoint(1, 280, 50)
polygon[2] = QtCore.QPoint(150, 280)
```

- `point(<Индекс>)` — возвращает объект класса `QPoint` с координатами вершины, индекс которой указан в параметре. Получить значение можно также с помощью операции доступа по индексу, например:

```
polygon = QtGui.QPolygon([20,50, 280,50, 150,280])
print(polygon.point(0)) # PyQt6.QtCore.QPoint(20, 50)
print(polygon[1])       # PyQt6.QtCore.QPoint(280, 50)
```

- `remove(<Индекс>[, <Количество>])` — удаляет указанное количество вершин, начиная с индекса `<Индекс>`. Если второй параметр не указан, удаляется одна вершина. Удалить вершину можно также с помощью оператора `del` по индексу или срезу;
- `clear()` — удаляет все вершины;
- `size()` — возвращает количество вершин;
- `count([<Координаты QPoint>])` — возвращает количество вершин. Если указан параметр `<Координаты>`, возвращается только количество вершин с этими координатами.

  Получить количество вершин можно также с помощью функции `len()`;

- `isEmpty()` — возвращает значение `True`, если многоугольник не содержит ни одной вершины, и `False` — в противном случае;
- `boundingRect()` — возвращает объект класса `QRect` с координатами и размерами прямоугольной области, в которую вписан многоугольник.

> ***Примечание***
>
> Класс `QPolygon` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QPolygonF`.

25.1.6. Класс *QFont*: шрифт

Класс `QFont` описывает характеристики шрифта. Форматы конструктора класса:

```
QFont()
QFont(<Название шрифта>[, pointSize=-1][, weight=-1][, italic=False])
QFont(<Исходный объект QFont>)
```

Первый формат создает объект шрифта с настройками, используемыми приложением по умолчанию. Установить шрифт приложения по умолчанию позволяет статический метод `setFont()` класса `QApplication`.

Второй формат позволяет указать основные характеристики шрифта. В первом параметре указывается название шрифта или семейства в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта: число от 0 до 99 или значение элемента `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `DemiBold`, `Bold`, `ExtraBold` или `Black` перечисления `Weight` из класса `QFont`. Если в параметре `italic` указано значение `True`, шрифт будет курсивным.

Третий формат создает копию заданного исходного объекта.

Класс `QFont` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти по адресу **https://doc.qt.io/qt-6/qfont.html**):

- `setFamily(<Название шрифта>)` — задает название шрифта или семейства шрифтов;
- `family()` — возвращает название шрифта;
- `setPointSize(<Размер типа int>)` и `setPointSizeF(<Размер типа float>)` — задают размер шрифта в пунктах;
- `setPixelSize(<Размер>)` — задает размер шрифта в пикселах;
- `pointSize()` — возвращает размер шрифта в пунктах в виде целого числа или значение `-1`, если размер шрифта был установлен в пикселах;
- `pointSizeF()` — возвращает размер шрифта в пунктах в виде вещественного числа или значение `-1`, если размер шрифта был установлен в пикселах;
- `pixelSize()` — возвращает размер шрифта в пикселах или `-1`, если размер шрифта был установлен в пунктах;
- `setWeight(<Жирность Weight>)` — задает степень жирности шрифта;
- `weight()` — возвращает степень жирности шрифта;
- `setBold(<Флаг>)` — если в качестве параметра указано значение `True`, то жирность шрифта устанавливается равной значению элемента `Bold`, а если `False` — то равной значению элемента `Normal` перечисления `Weight` из класса `QFont`;
- `bold()` — возвращает значение `True`, если степень жирности шрифта больше значения элемента `Normal` перечисления `Weight` из класса `QFont`, и `False` — в противном случае;
- `setItalic(<Флаг>)` — если в качестве параметра указано значение `True`, шрифт будет курсивным, а если `False` — обычного начертания;
- `italic()` — возвращает значение `True`, если шрифт курсивный, и `False` — в противном случае;
- `setUnderline(<Флаг>)` — если в качестве параметра указано значение `True`, текст будет подчеркнутым, а если `False` — неподчеркнутым;

- underline() — возвращает значение True, если текст подчеркнут, и False — в противном случае;
- setOverline(<Флаг>) — если в качестве параметра указано значение True, то текст будет надчеркнутым;
- overline() — возвращает значение True, если текст надчеркнут, и False — в противном случае;
- setStrikeOut(<Флаг>) — если в качестве параметра указано значение True, текст будет зачеркнутым;
- strikeOut() — возвращает значение True, если текст зачеркнут, и False — в противном случае.

Получить список всех доступных шрифтов позволяет метод families() класса QFontDatabase. Метод возвращает список строк. Отметим, что перед его вызовом следует создать объект класса QApplication, в противном случае мы получим ошибку исполнения:

```
from PyQt6 import QtGui, QtWidgets
app = QtWidgets.QApplication(list())
print(QtGui.QFontDatabase.families())
```

Чтобы получить список доступных стилей для указанного шрифта, следует воспользоваться статическим методом styles(<Название шрифта>) класса QFontDatabase:

```
print(QtGui.QFontDatabase.styles("Arial"))
# ['Обычный', 'Полужирный', 'Полужирный Курсив', 'Курсив']
```

Получить допустимые размеры для указанного стиля можно с помощью статического метода smoothSizes(<Название шрифта>, <Стиль>) класса QFontDatabase:

```
print(QtGui.QFontDatabase.smoothSizes("Arial", "Обычный"))
# [6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72]
```

Очень часто необходимо произвести выравнивание выводимого текста внутри некоторой области. Чтобы это сделать, нужно знать размеры области, в которую вписан текст. Получить эти значения можно с помощью класса QFontMetrics, конструктор которого имеет следующий формат вызова:

```
QFontMetrics(<Шрифт QFont>)
```

Класс QFontMetrics поддерживает следующие полезные методы:

- horiontalAdvance(<Текст>[, length=-1]) — возвращает расстояние от начала текста <Текст> до позиции, в которой должен начаться другой текст. Параметр length позволяет ограничить количество символов;
- height() — возвращает высоту шрифта;
- boundingRect(<Текст>) — возвращает объект класса QRect с координатами и размерами прямоугольной области, в которую вписан текст.

Вот пример получения размеров области:

```
font = QtGui.QFont("Tahoma", 16)
fm = QtGui.QFontMetrics(font)
print(fm.horizontalAdvance("Строка"))     # 67
print(fm.height())                        # 25
print(fm.boundingRect("Строка"))          # PyQt6.QtCore.QRect(0, -21, 66, 25)
```

Обратите внимание, что значения, возвращаемые методами `horizontalAdvance()` и `QRect.width()`, различаются.

> ***Примечание***
>
> Класс `QFontMetrics` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QFontMetricsF`.

25.2. Класс *QPainter*

Класс `QPainter` содержит все необходимые средства, позволяющие выполнять рисование геометрических фигур и вывод текста на поверхности, реализуемой классом `QPaintDevice`. Класс `QPaintDevice` наследуют классы `QWidget`, `QPicture`, `QPixmap`, `QImage`, `QPagedPaintDevice` и некоторые другие. Это дает возможность рисовать на поверхности любого компонента, на изображении или на печатаемой странице. Форматы конструктора класса:

```
QPainter()
QPainter(<Поверхность рисования>)
```

Первый формат создает объект, который не подключен ни к одной поверхности рисования. Чтобы подключиться к поверхности и захватить контекст рисования, необходимо вызвать метод `begin(<Поверхность>)` и передать ему ссылку на объект класса, производного от `QPaintDevice`. Метод возвращает значение `True`, если контекст успешно захвачен, и `False` — в противном случае. В один момент времени только один объект может рисовать на поверхности, поэтому после окончания рисования необходимо освободить контекст рисования с помощью метода `end()`. С учетом сказанного код, позволяющий рисовать на компоненте, будет выглядеть так:

```
def paintEvent(self, e):
    # Компонент, на котором выполняется рисование, передается в параметре self
    painter = QtGui.QPainter()
    painter.begin(self)
    # Здесь производим рисование на компоненте
    painter.end()
```

Второй формат принимает поверхность рисования, которая задана в виде объекта класса, производного от `QPaintDevice`, подключается к этой поверхности и сразу захватывает контекст рисования. Контекст рисования автоматически освобождается внутри деструктора класса `QPainter` при уничтожении объекта. Так как объект автоматически уничтожается при выходе из метода `paintEvent()`, то метод `end()` можно и не вызывать.

Вот пример рисования на компоненте:

```
def paintEvent(self, e):
    painter = QtGui.QPainter(self)
    # Здесь производим рисование на компоненте
```

Проверить успешность захвата контекста рисования можно с помощью метода `isActive()`: он возвращает значение `True`, если контекст захвачен, и `False` — в противном случае.

25.2.1. Рисование линий и фигур

После захвата контекста рисования следует установить перо и кисть. С помощью пера производится рисование точек, линий и контуров фигур, а с помощью кисти — заполнение фона фигур. Установить перо позволяет метод `setPen()` класса `QPainter`. Форматы метода:

```
setPen(<Перо QPen>)
setPen(<Цвет QColor>)
setPen(<Стиль пера PenStyle>)
```

Для установки кисти предназначен метод `setBrush()`. Форматы метода:

```
setBrush(<Кисть QBrush>)
setBrush(<Стиль кисти BrushStyle>)
```

Устанавливать перо или кисть необходимо перед каждой операцией рисования, требующей изменения цвета или стиля. Если перо или кисть не установлены, будут использоваться объекты с настройками по умолчанию. После установки пера и кисти можно приступать к рисованию точек, линий, фигур, текста и др.

Для рисования точек, линий и фигур класс `QPainter` предоставляет следующие наиболее часто употребляемые методы (полный их список приведен на странице **https://doc.qt.io/qt-6/qpainter.html**):

- `drawPoint()` — рисует точку. Форматы метода:

```
drawPoint(<X>, <Y>)
drawPoint(<Координаты QPoint или QPointF>)
```

- `drawPoints()` — рисует несколько точек. Форматы метода:

```
drawPoints(<Координаты QPoint 1>[, . . ., <Координаты QPoint N>])
drawPoints(<Координаты QPointF 1>[, . . ., <Координаты QPointF N>])
drawPoints(<Многоугольник QPolygon или QPolygonF>)
```

- `drawLine()` — рисует линию. Форматы метода:

```
drawLine(<Линия QLine или QLineF>)
drawLine(<Начальная точка QPoint>, <Конечная точка QPoint>)
drawLine(<Начальная точка QPointF>, <Конечная точка QPointF>)
drawLine(<X1>, <Y1>, <X2>, <Y2>)
```

- `drawLines()` — рисует несколько отдельных линий. Форматы метода:

```
drawLines(<Линия QLine 1>[, . . ., <Линия QLine N>])
drawLines(<Линия QLineF 1>[, . . ., <Линия QLineF N>])
drawLines(<Список с линиями QLineF>)
drawLines(<Точка QPoint 1>[, . . ., <Точка QPoint N>])
drawLines(<Точка QPointF 1>[, . . ., <Точка QPointF N>])
```

- `drawPolyline()` — рисует несколько линий, которые соединяют указанные точки. Первая и последняя точки не соединяются. Форматы метода:

```
drawPolyline(<Точка QPoint 1>[, . . ., <Точка QPoint N>])
drawPolyline(<Точка QPointF 1>[, . . ., <Точка QPointF N>])
drawPolyline(<Многоугольник QPolygon или QPolygonF>)
```

- `drawRect()` — рисует прямоугольник с границей и заливкой. Чтобы убрать границу, следует использовать перо со стилем `NoPen`, а чтобы убрать заливку — кисть со стилем `NoBrush`. Форматы метода:

```
drawRect(<X>, <Y>, <Ширина>, <Высота>)
drawRect(<Прямоугольник QRect или QRectF>)
```

- `fillRect()` — рисует прямоугольник с заливкой без границы. Форматы метода:

```
fillRect(<X>, <Y>, <Ширина>, <Высота>, <Заливка>)
fillRect(<Прямоугольник QRect или QRectF>, <Заливка>)
```

`<Заливка>` может быть задана объектами классов `<QColor>`, `<QBrush>`, в виде стиля кисти или элемента перечисления `GlobalColor`;

- `drawRoundedRect()` — рисует прямоугольник с границей, заливкой и скругленными краями. Форматы метода:

```
drawRoundedRect(<X>, <Y>, <Ширина>, <Высота>,
                <Скругление по горизонтали>, <Скругление по вертикали>[,
                mode=SizeMode.AbsoluteSize])
drawRoundedRect(<Прямоугольник QRect или QRectF>,
                <Скругление по горизонтали>, <Скругление по вертикали>[,
                mode=SizeMode.AbsoluteSize])
```

Параметры `<Скругление по горизонтали>` и `<Скругление по вертикали>` задают радиусы скругления углов по горизонтали и вертикали. Необязательный параметр `mode` указывает, в каких единицах измеряются радиусы скругления углов, и задается одним из следующих элементов перечисления `SizeMode` из модуля `QtCore.Qt`:

  - `AbsoluteSize` — радиусы указываются в пикселах;
  - `RelativeSize` — радиусы указываются в процентах от соответствующего размера рисуемого прямоугольника;

- `drawPolygon()` — рисует многоугольник с границей и заливкой. Форматы метода:

```
drawPolygon(<Вершина QPoint 1>[, . . ., <Вершина QPoint N>])
drawPolygon(<Вершина QPointF 1>[, . . ., <Вершина QPointF N>])
drawPolygon(<Многоугольник QPolygon или QPolygonF>[,
            fillRule=FillRule.OddEvenFill])
```

Необязательный параметр `fillRule` задает алгоритм определения, находится ли какая-либо точка внутри нарисованного многоугольника или вне его. В качестве его значения указывается атрибут `OddEvenFill` или `WindingFill` перечисления `FillRule` из модуля `QtCore.Qt`;

- `drawEllipse()` — рисует эллипс с границей и заливкой. Форматы метода:

```
drawEllipse(<X>, <Y>, <Ширина>, <Высота>)
drawEllipse(<Прямоугольник QRect или QRectF>)
drawEllipse(<Точка QPoint или QPointF>, <int rX>, <int rY>)
```

В первых двух форматах указываются координаты и размеры прямоугольника, в который необходимо вписать эллипс. В последнем формате первый параметр задает координаты центра, параметр `rX` — радиус по оси `X`, а параметр `rY` — радиус по оси `Y`;

- `drawArc()` — рисует дугу. Форматы метода:

```
drawArc(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>, <Угол>)
drawArc(<Прямоугольник QRect или QRectF>, <Начальный угол>, <Угол>)
```

  Значения углов задаются в значениях $^1/_{16}°$. Полный круг эквивалентен значению 5760 = 16 × 360. Нулевой угол находится в позиции «трех часов». Положительные значения углов отсчитываются против часовой стрелки, а отрицательные — по часовой стрелке;

- `drawChord()` — рисует замкнутую дугу. Аналогичен методу `drawArc()`, но соединяет крайние точки дуги прямой линией. Форматы метода:

```
drawChord(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>, <Угол>)
drawChord(<Прямоугольник QRect или QRectF>, <Начальный угол>, <Угол>)
```

- `drawPie()` — рисует замкнутый сектор. Аналогичен методу `drawArc()`, но соединяет крайние точки дуги с центром окружности. Форматы метода:

```
drawPie(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>,  <Угол>)
drawPie(<Прямоугольник QRect или QRectF>, <Начальный угол>, <Угол>)
```

При выводе некоторых фигур (например, эллипса) контур может отображаться в виде «лесенки». Чтобы сгладить контуры фигур, следует вызвать метод `setRenderHint(<Режим сглаживания>)` и передать ему в качестве единственного параметра элемент `Antialiasing` перечисления `RenderHint` из класса `QPainter`:

```
painter.setRenderHint(QtGui.QPainter.RenderHint.Antialiasing)
```

Если требуется отключить сглаживание, следует вызвать тот же метод, но передать ему вторым параметром значение `False`:

```
painter.setRenderHint(QtGui.QPainter.RenderHint.Antialiasing, False)
```

25.2.2. Вывод текста

Вывести текст позволяет метод `drawText()` класса `QPainter`. Форматы метода:

```
drawText(<X>, <Y>, <Текст>)
drawText(<Координаты QPoint или QPointF>, <Текст>)
drawText(<X>, <Y>, <Ширина>, <Высота>, <Флаги>, <Текст>)
drawText(<Прямоугольник QRect или QRectF>, <Флаги>, <Текст>)
drawText(<Прямоугольник QRectF>, <Текст>[, option=QTextOption()])
```

Первые два формата метода выводят текст, начиная с указанных координат.

Следующие два формата выводят текст в указанную прямоугольную область. При этом текст, который не помещается в эту область, будет обрезан, если не указан флаг `TextDontClip`. Методы возвращают объект класса `QRect` (`QRectF` для четвертого формата) с координатами и размерами прямоугольника, в который вписан текст. В параметре `<Флаги>` через оператор `|` указываются элементы `AlignLeft`, `AlignRight`, `AlignHCenter`, `AlignTop`, `AlignBottom`, `AlignVCenter` или `AlignCenter` перечисления `AlignmentFlag` из модуля `QtCore.Qt`, задающие выравнивание текста внутри прямоугольной области, а также следующие элементы перечисления `TextFlag` из того же модуля:

- `TextSingleLine` — все пробельные символы (табуляция, возвраты каретки и переводы строки) трактуются как пробелы, и текст выводится в одну строку;

- TextDontClip — часть текста, вышедшая за пределы указанной прямоугольной области, не будет обрезаться;
- TextExpandTabs — символы табуляции будут обрабатываться;
- TextShowMnemonic — символ, перед которым указан знак &, будет подчеркнут. Чтобы вывести символ &, его необходимо удвоить;
- TextWordWrap — если текст не помещается на одной строке, будет произведен перенос слова без его разрыва;
- TextWrapAnywhere — перенос строки может быть выполнен внутри слова;
- TextHideMnemonic — то же самое, что и TextShowMnemonic, но символ не подчеркивается;
- TextDontPrint — текст не будет напечатан;
- TextIncludeTrailingSpaces — размеры текста будут возвращаться с учетом начальных и конечных пробелов, если таковые есть в тексте;
- TextJustificationForced — задает выравнивание по ширине у последней строки текста.

Пятый формат метода drawText() также выводит текст в указанную прямоугольную область, но выравнивание текста и другие опции задаются с помощью объекта класса QTextOption. Например, с помощью этого класса можно отобразить непечатаемые символы (символ пробела, табуляцию и др.).

Получить координаты и размеры прямоугольника, в который вписывается текст, позволяет метод boundingRect() класса QPainter. Форматы метода:

```
boundingRect(<X>, <Y>, <Ширина>, <Высота>, <Флаги>, <Текст>)
boundingRect(<Прямоугольник QRect>, <Флаги>, <Текст>)
boundingRect(<Прямоугольник QRectF>, <Флаги>, <Текст>)
boundingRect(<Прямоугольник QRectF>, <Текст>[, option=QTextOption()])
```

Первые два формата возвращают объект класса QRect, а последние два — объект класса QRectF.

При выводе текста линии букв могут отображаться в виде «лесенки». Чтобы сгладить контуры, следует вызвать метод setRenderHint(<Режим сглаживания>) и передать ему элемент TextAntialiasing перечисления RenderHint из класса QPainter:

```
painter.setRenderHint(QtGui.QPainter.RenderHint.TextAntialiasing)
```

Если требуется отключить сглаживание, следует вызвать тот же метод, но передать ему вторым параметром значение False:

```
painter.setRenderHint(QtGui.QPainter.RenderHint.TextAntialiasing, False)
```

25.2.3. Вывод изображений

Для вывода растровых изображений предназначены методы drawPixmap() и drawImage() класса QPainter. Метод drawPixmap() обеспечивает вывод изображений, хранимых в объекте класса QPixmap. Форматы метода:

```
drawPixmap(<X>, <Y>, <Изображение QPixmap>)
drawPixmap(<Координаты QPoint или QPointF>, <Изображение QPixmap>)
drawPixmap(<X>, <Y>, <Ширина>, <Высота>, <Изображение QPixmap>)
drawPixmap(<Прямоугольник QRect>, <Изображение QPixmap>)
```

```
drawPixmap(<X1>, <Y1>, <Изображение QPixmap>, <X2>, <Y2>,
                                              <Ширина 2>, <Высота 2>)
drawPixmap(<Координаты QPoint>, <Изображение QPixmap>,
                                <Прямоугольник QRect>)
drawPixmap(<Координаты QPointF>, <Изображение QPixmap>,
                                 <Прямоугольник QRectF>)
drawPixmap(<X1>, <Y1>, <Ширина 1>, <Высота 1>, <Изображение QPixmap>,
           <X2>, <Y2>, <Ширина 2>, <Высота 2>)
drawPixmap(<Прямоугольник QRect>, <Изображение QPixmap>,
                                  <Прямоугольник QRect>)
drawPixmap(<Прямоугольник QRectF>, <Изображение QPixmap>,
                                   <Прямоугольник QRectF>)
```

Первые два формата задают координаты, в которые будет установлен левый верхний угол выводимого изображения:

```
pixmap = QtGui.QPixmap("foto.jpg")
painter.drawPixmap(3, 3, pixmap)
```

Третий и четвертый форматы позволяют ограничить вывод изображения указанной прямоугольной областью. Если размеры области не совпадают с размерами изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Пятый, шестой и седьмой форматы задают координаты, в которые будет установлен левый верхний угол фрагмента изображения. Координаты и размеры вставляемого фрагмента изображения указываются после объекта класса `QPixmap` в виде отдельных составляющих или объектов классов `QRect` или `QRectF`.

Последние три формата ограничивают вывод фрагмента изображения указанной прямоугольной областью. Координаты и размеры вставляемого фрагмента изображения указываются после изображения в виде отдельных составляющих или объектов классов `QRect` или `QRectF`. Если размеры области не совпадают с размерами фрагмента изображения, производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Метод `drawImage()` предназначен для вывода изображений, хранимых в объектах класса `QImage`. Форматы метода:

```
drawImage(<Координаты QPoint или QPointF>, <Изображение QImage>)
drawImage(<Прямоугольник QRect или QRectF>, <Изображение QImage>)
drawImage(<X1>, <Y1>, <Изображение QImage>[, sx=0][, sy=0][,
                      sw=-1][, sh=-1][, flags=ImageConversionFlag.AutoColor])
drawImage(<Координаты QPoint>, <Изображение QImage>,
          <Прямоугольник QRect>[, flags=ImageConversionFlag.AutoColor])
drawImage(<Координаты QPointF>, <Изображение QImage>,
          <Прямоугольник QRectF>[, flags=ImageConversionFlag.AutoColor])
drawImage(<Прямоугольник QRect>, <Изображение QImage>,
          <Прямоугольник QRect>[, flags=ImageConversionFlag.AutoColor])
drawImage(<Прямоугольник QRectF>, <Изображение QImage>,
          <Прямоугольник QRectF>[, flags=ImageConversionFlag.AutoColor])
```

Первый, а также третий формат со значениями по умолчанию задают координаты, по которым будет находиться левый верхний угол выводимого изображения:

```
img = QtGui.QImage("foto.jpg")
painter.drawImage(3, 3, img)
```

Второй формат ограничивает вывод изображения указанной прямоугольной областью. Если размеры области не совпадают с размерами изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Третий, четвертый и пятый форматы задают координаты, в которые будет установлен левый верхний угол фрагмента изображения. Координаты и размеры вставляемого фрагмента изображения указываются после изображения в виде отдельных составляющих или объектов классов `QRect` или `QRectF`.

Последние два формата ограничивают вывод фрагмента изображения указанной прямоугольной областью. Координаты и размеры вставляемого фрагмента изображения указываются после изображения в виде объектов классов `QRect` или `QRectF`. Если размеры области не совпадают с размерами фрагмента изображения, производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Необязательный параметр `flags` задает цветовые преобразования, которые будут выполнены при выводе изображения (фактически — при неявном преобразовании объекта класса `QImage` в объект класса `QPixmap`, которое обязательно выполняется перед выводом). Они указываются в виде элементов перечисления `ImageConversionFlag` из модуля `QtCore.Qt`, приведенных на странице **https://doc.qt.io/qt-6/qt.html#ImageConversionFlag-enum** . В большинстве случаев имеет смысл использовать заданный по умолчанию элемент `AutoColor` этого перечисления.

25.2.4. Преобразование систем координат

Существуют две системы координат: физическая (viewport, система координат устройства) и логическая (window). При рисовании координаты из логической системы координат преобразуются в систему координат устройства. По умолчанию эти две системы координат совпадают.

В некоторых случаях возникает необходимость изменить координаты. Выполнить изменение физической системы координат позволяет метод `setViewport(<X>, <Y>, <Ширина>, <Высота>)` или `setViewport(<Объект класса QRect>)`, а получить текущие значения — метод `viewport()`. Выполнить изменение логической системы координат позволяет метод `setWindow(<X>, <Y>, <Ширина>, <Высота>)` или `setWindow(<Объект класса QRect>)`, а получить текущие значения — метод `window()` класса `QPainter`.

Произвести дополнительную трансформацию системы координат позволяют следующие методы того же класса `QPainter`:

- `translate()` — перемещает начало координат в указанную точку. По умолчанию начало координат находится в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз. Форматы метода:

  ```
  translate(<X>, <Y>)
  translate(<Точка QPoint или QPointF>)
  ```

- `rotate(<Угол>)` — поворачивает систему координат на заданный угол (указывается в виде вещественного числа в градусах). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки;

- scale(<По оси X>, <По оси Y>) — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы — то увеличение;
- shear(<По горизонтали>, <По вертикали>) — сдвигает систему координат. В качестве значений указываются вещественные числа.

Все указанные трансформации влияют на последующие операции рисования и не изменяют ранее нарисованные фигуры. Чтобы после трансформации восстановить систему координат, следует предварительно сохранить состояние в стеке с помощью метода save(), а после окончания рисования вызвать метод restore():

```
painter.save()      # Сохраняем состояние
# Трансформируем и рисуем
painter.restore()   # Восстанавливаем состояние
```

Несколько трансформаций можно произвести последовательно друг за другом. При этом надо учитывать, что порядок следования трансформаций имеет значение.

Если одна и та же последовательность трансформаций выполняется несколько раз, то ее можно сохранить в объекте класса QTransform, а затем установить с помощью метода setTransform(<Трансформация>):

```
transform = QtGui.QTransform()
transform.translate(105, 105)
transform.rotate(45.0)
painter.setTransform(transform)
painter.fillRect(-25, -25, 50, 50, QtCore.Qt.green)
```

25.2.5. Сохранение команд рисования в файл

Класс QPicture выполняет роль устройства для рисования с возможностью сохранения команд рисования в файле специального формата для последующего вывода его на экран. Иерархия наследования:

```
QPaintDevice — QPicture
```

Форматы конструктора класса:

```
<Объект> = QPicture([formatVersion=-1])
<Объект> = QPicture(<Исходный объект QPicture>)
```

Первый конструктор создает пустой рисунок. Необязательный параметр formatVersion задает версию формата. Если параметр не указан, то используется формат, принятый в текущей версии PyQt. Второй конструктор создает копию заданного объекта.

Для сохранения и загрузки рисунка предназначены следующие методы:

- save(<Путь к файлу>) — сохраняет рисунок в файле с указанным путем. Возвращает значение True, если рисунок успешно сохранен, и False — в противном случае;
- load(<Путь к файлу>) — загружает рисунок из файла с указанным путем. Возвращает значение True, если рисунок успешно загружен, и False — в противном случае.

Для вывода загруженного рисунка на устройство рисования по указанным координатам предназначен метод drawPicture() класса QPainter. Форматы метода:

```
drawPicture(<X>, <Y>, <Изображение QPicture>)
drawPicture(<Координаты QPoint или QPointF>, <Изображение QPicture>)
```

Пример сохранения рисунка:

```
painter = QtGui.QPainter()
pic = QtGui.QPicture()
painter.begin(pic)
# Здесь что-то рисуем
painter.end()
pic.save(r"c:\book\pic.dat")
```

Пример вывода загруженного рисунка на поверхность компонента:

```
def paintEvent(self, e):
    painter = QtGui.QPainter(self)
    pic = QtGui.QPicture()
    pic.load(r"c:\book\pic.dat")
    painter.drawPicture(0, 0, pic)
```

25.3. Работа с растровыми изображениями

Библиотека PyQt включает несколько классов, позволяющих работать с растровыми изображениями в контекстно-зависимом (классы `QPixmap` и `QBitmap`) и контекстно-независимом (класс `QImage`) представлениях.

Получить список графических форматов, доступных для загрузки, позволяет статический метод `supportedImageFormats()` класса `QImageReader`, возвращающий список с объектами класса `QByteArray`. Получим список поддерживаемых форматов для чтения:

```
>>> for i in QtGui.QImageReader.supportedImageFormats():
...     print(str(i, "ascii").upper(), end=" ")
...
BMP CUR GIF ICNS ICO JPEG JPG PBM PGM PNG PPM SVG SVGZ TGA TIF TIFF WBMP
WEBP XBM XPM
```

Получить список графических форматов, доступных для сохранения, позволяет статический метод `supportedImageFormats()` класса `QImageWriter`, возвращающий список с объектами класса `QByteArray`. Получим список поддерживаемых форматов для записи:

```
>>> for i in QtGui.QImageWriter.supportedImageFormats():
...     print(str(i, "ascii").upper(), end=" ")
...
BMP CUR ICNS ICO JPEG JPG PBM PGM PNG PPM TIF TIFF WBMP WEBP XBM XPM
```

Обратите внимание, что мы можем загрузить изображение в формате GIF, но не имеем возможности сохранить изображение в этом формате, поскольку алгоритм сжатия, используемый в нем, защищен патентом.

25.3.1. Класс *QPixmap*

Класс `QPixmap` предназначен для работы с изображениями в контекстно-зависимом представлении. Данные хранятся в виде, позволяющем отображать изображение на экране наиболее эффективным способом, поэтому класс `QPixmap` часто используется для предварительного рисования графики перед выводом ее на экран. Иерархия наследования:

```
QPaintDevice — QPixmap
```

Поскольку класс QPixmap наследует класс QPaintDevice, мы можем использовать его как поверхность для рисования. Вывести изображение позволяет метод drawPixmap() класса QPainter (см. *разд. 25.2.3*).

Форматы конструктора класса:

```
QPixmap()
QPixmap(<Ширина>, <Высота>)
QPixmap(<Размеры QSize>)
QPixmap(<Путь к файлу>[, format=None][, flags=Image.ConversionFlag.AutoColor])
QPixmap(<Исходный объект QPixmap>)
```

Первый формат создает пустой объект изображения. Второй и третий форматы позволяют указать размеры изображения — если размеры равны нулю, то будет создан пустой объект. Четвертый формат предназначен для загрузки изображения из файла. Во втором параметре указывается тип изображения в виде строки (например, "PNG") — если он не указан, то формат будет определен по расширению загружаемого файла. Пятый конструктор создает копию указанного изображения.

Класс QPixmap поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qpixmap.html**):

- isNull() — возвращает значение True, если объект является пустым, и False — в противном случае;
- load() — загружает изображение из файла с указанным путем. Формат метода:

  ```
  load(<Путь к файлу>[, format=None][, flags=ImageConversionFlag.AutoColor])
  ```

  Во втором параметре можно задать формат файла в виде строки — если он не указан, формат определяется по расширению файла. Необязательный параметр flags задает тип преобразования цветов. Метод возвращает значение True, если изображение успешно загружено, и False — в противном случае;
- loadFromData() — загружает изображение из заданного объекта класса QByteArray. Формат метода:

  ```
  loadFromData(<Объект QByteArray или bytes>[, format=None][,
               flags=ImageConversionFlag.AutoColor])
  ```

  Метод возвращает значение True, если изображение успешно загружено, и False — в противном случае;
- save(<Путь к файлу>[, format=None][, quality=-1]) — сохраняет изображение в файл с указанным путем. Во втором параметре можно задать формат файла в виде строки — если он не указан, формат будет определен по расширению файла. Необязательный параметр quality позволяет задать качество изображения. Можно передать значение в диапазоне от 0 до 100, значение -1 указывает качество по умолчанию. Метод возвращает значение True, если изображение успешно сохранено, и False — в противном случае;
- convertFromImage() — преобразует заданный объект класса QImage в объект класса QPixmap. Формат метода:

  ```
  convertFromImage(<Изображение QImage>[, flags=ImageConversionFlag.AutoColor])
  ```

  Метод возвращает значение True, если изображение успешно преобразовано, и False — в противном случае;

- `fromImage()` — статический, преобразует заданный объект класса `QImage` в объект класса `QPixmap`, который и возвращает. Формат метода:

```
fromImage(<Изображение QImage>[, flags=ImageConversionFlag.AutoColor])
```

- `toImage()` — преобразует текущее изображение в объект класса `QImage` и возвращает его;
- `fill([color=GlobalColor.white])` — производит заливку изображения указанным цветом;
- `width()` — возвращает ширину изображения;
- `height()` — возвращает высоту изображения;
- `size()` — возвращает объект класса `QSize` с размерами изображения;
- `rect()` — возвращает объект класса `QRect` с координатами и размерами прямоугольной области, ограничивающей изображение;
- `depth()` — возвращает глубину цвета;
- `isQBitmap()` — возвращает значение `True`, если глубина цвета равна одному биту (т. е. это монохромное изображение), и `False` — в противном случае;
- `createMaskFromColor(<Цвет QColor>[, mode=MaskMode.MaskInColor])` — создает на основе текущего изображения маску в виде объекта класса `QBitmap` и возвращает ее. Первый параметр задает цвет — области, закрашенные этим цветом, будут на маске либо прозрачными, либо непрозрачными. Необязательный параметр `mode` задает режим создания маски в виде следующих элементов перечисления `MaskMode` из модуля `QtCore.Qt`:
  - `MaskInColor` — области, закрашенные указанным цветом, будут прозрачными;
  - `MaskOutColor` — области, закрашенные указанным цветом, будут непрозрачными;
- `setMask(<Маска QBitmap>)` — устанавливает маску;
- `mask()` — возвращает объект класса `QBitmap` с маской изображения;
- `copy()` — возвращает объект класса `QPixmap` с фрагментом изображения. Если параметр `rect` не указан, изображение копируется полностью. Форматы метода:

```
copy([rect=QRect()])
copy(<X>, <Y>, <Ширина>, <Высота>)
```

- `scaled()` — изменяет размер изображения и возвращает результат в виде объекта класса `QPixmap`. Исходное изображение не изменяется. Форматы метода:

```
scaled(<Ширина>, <Высота>[,
       aspectRatioMode=AspectRatioMode.IgnoreAspectRatio][,
       transformMode=TransformationMode.FastTransformation])
scaled(<Размеры QSize>[,
       aspectRatioMode=AspectRatioMode.IgnoreAspectRatio][,
       transformMode=TransformationMode.FastTransformation])
```

В необязательном параметре `aspectRatioMode` могут быть указаны следующие элементы перечисления `AspectRatioMode` из модуля `QtCore.Qt`:

- `IgnoreAspectRatio` — изменяет размеры без сохранения пропорций сторон;
- `KeepAspectRatio` — изменяет размеры с сохранением пропорций сторон. При этом часть области нового изображения может оказаться незаполненной;

- KeepAspectRatioByExpanding — изменяет размеры с сохранением пропорций сторон. При этом часть нового изображения может выйти за пределы его области.

В необязательном параметре transformMode могут быть указаны следующие элементы перечисления TransformationMode из модуля QtCore.Qt:

- FastTransformation — сглаживание выключено;
- SmoothTransformation — сглаживание включено;

♦ scaledToWidth() — изменяет ширину изображения и возвращает результат в виде объекта класса QPixmap. Формат метода:

```
scaledToWidth(<Ширина>[, mode=TransformationMode.FastTransformation])
```

Высота изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр mode аналогичен параметру transformMode в методе scaled();

♦ scaledToHeight() — изменяет высоту изображения и возвращает результат в виде объекта класса QPixmap. Формат метода:

```
scaledToHeight(<Высота>[, mode=TransformationMode.FastTransformation])
```

Ширина изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр mode аналогичен параметру transformMode в методе scaled();

♦ transformed() — производит трансформацию изображения (например, поворот) и возвращает результат в виде объекта класса QPixmap. Формат метода:

```
transformed(<Трансформация QTransform>[,
            mode=TransformationMode.FastTransformation])
```

Исходное изображение не изменяется. Параметр mode аналогичен параметру transformMode в методе scaled();

♦ swap(<Изображение QPixmap>) — заменяет текущее изображение указанным в параметре;

♦ hasAlpha() — возвращает True, если изображение имеет прозрачные области, и False — в противном случае;

♦ hasAlphaChannel() — возвращает True, если формат изображения поддерживает прозрачность, и False — в противном случае.

25.3.2. Класс *QBitmap*

Класс QBitmap предназначен для работы в контекстно-зависимом представлении с монохромными изображениями. Наиболее часто класс QBitmap используется для создания масок изображений. Иерархия наследования:

```
QPaintDevice — QPixmap — QBitmap
```

Поскольку класс QBitmap наследует класс QPaintDevice, мы можем использовать его как поверхность для рисования. Цвет пера и кисти задается элементами color0 (прозрачный цвет) и color1 (непрозрачный цвет) перечисления GlobalColor из модуля QtCore.Qt. Вывести изображение позволяет метод drawPixmap() класса QPainter (см. *разд. 25.2.3*).

Форматы конструктора класса:

```
QBitmap()
QBitmap(<Ширина>, <Высота>)
QBitmap(<Размеры QSize>)
```

```
QBitmap(<Путь к файлу>[, format=None])
QBitmap(<Изображение QPixmap>)
QBitmap(<Исходный объект QBitmap>)
```

Класс `QBitmap` наследует все методы класса `QPixmap` и определяет следующие дополнительные методы (здесь приведены только нас интересующие — полный их список можно найти на странице **https://doc.qt.io/qt-6/qbitmap.html**):

- `fromImage()` — статический, преобразует заданный объект класса `QImage` в объект класса `QBitmap`, который и возвращает. Формат метода:

  ```
  fromImage(<Изображение QImage>[, flags=ImageConversionFlag.AutoColor])
  ```

- `transformed(<Трансформация QTransform>)` — производит трансформацию изображения (например, поворот) и возвращает объект класса `QBitmap`. Исходное изображение не изменяется;
- `clear()` — очищает изображение, устанавливая у всех точек изображения цвет `color0`.

25.3.3. Класс *QImage*

Класс `QImage` предназначен для работы с изображениями в контекстно-независимом представлении. Иерархия наследования:

```
QPaintDevice — QImage
```

Поскольку класс `QImage` наследует класс `QPaintDevice`, мы можем использовать его как поверхность для рисования. Однако следует учитывать, что не на всех форматах изображения можно рисовать, — для рисования лучше использовать изображение формата `Format_ARGB32_Premultiplied`. Вывести изображение позволяет метод `drawImage()` класса `QPainter` (см. *разд. 25.2.3*).

Форматы конструктора класса:

```
QImage()
QImage(<Ширина>, <Высота>, <Формат>)
QImage(<Размеры QSize>, <Формат>)
QImage(<Путь к файлу>[, <Тип изображения>])
QImage(<Исходный объект QImage>)
```

Первый конструктор создает пустой объект изображения. Второй и третий конструкторы позволяют указать размеры изображения — если таковые равны нулю, будет создан пустой объект. Четвертый конструктор предназначен для загрузки изображения из файла. Во втором параметре указывается тип изображения в виде строки — если он не указан, формат будет определен по расширению загружаемого файла. Пятый конструктор создает копию заданного исходного объекта.

В параметре `<Формат>` можно указать следующие элементы перечисления `Format` из класса `QImage` (приведены только наиболее часто используемые — полный их список можно найти на странице **https://doc.qt.io/qt-6/qimage.html#Format-enum**):

- `Format_Invalid` — неверный формат;
- `Format_Mono` — глубина цвета 1 бит;
- `Format_MonoLSB` — глубина цвета 1 бит;
- `Format_Indexed8` — глубина цвета 8 битов;
- `Format_RGB32` — RGB без альфа-канала, глубина цвета 32 бита;

- `Format_ARGB32` — RGB с альфа-каналом, глубина цвета 32 бита;
- `Format_ARGB32_Premultiplied` — RGB с альфа-каналом, глубина цвета 32 бита. Для рисования лучше использовать этот формат.

Класс `QImage` поддерживает большое количество методов, из которых мы рассмотрим лишь основные (полный их список приведен на странице **https://doc.qt.io/qt-6/qimage.html**):

- `isNull()` — возвращает значение `True`, если текущий объект является пустым, и `False` — в противном случае;
- `load(<Путь к файлу>[, format=None])` — загружает изображение из файла с указанным путем. Во втором параметре задается формат файла в виде строки — если он не указан, формат определяется по расширению файла. Метод возвращает значение `True`, если изображение успешно загружено, и `False` — в противном случае;
- `loadFromData(<Объект QByteArray или bytes>[, format=None])` — загружает изображение из заданного объекта. Во втором параметре указывается тип изображения в виде строки (например: `"PNG"`). Метод возвращает значение `True`, если изображение успешно загружено, и `False` — в противном случае;
- `fromData(<Объект QByteArray или bytes>[, format=None])` — статический, загружает изображение из заданного объекта. Во втором параметре указывается тип изображения в виде строки (например: `"PNG"`). Метод возвращает объект класса `QImage`;
- `save(<Путь к файлу>[, format=None][, quality=-1])` — сохраняет изображение в файле с указанным путем. Во втором параметре можно задать формат файла в виде строки — если он не указан, формат определяется по расширению файла. Необязательный параметр `quality` позволяет задать качество изображения. Можно передать значение в диапазоне от 0 до 100; значение `-1` указывает качество по умолчанию. Метод возвращает значение `True`, если изображение успешно сохранено, и `False` — в противном случае;
- `fill(<Цвет>)` — производит заливку изображения заданным цветом. В качестве параметра указывается элемент перечисления `GlobalColor`, объект класса `QColor` или целочисленное значение цвета, возвращаемое методами `rgb()` и `rgba()` класса `QColor`. Примеры:

```
img.fill(QtCore.Qt.GlobalColor.red)
img.fill(QtGui.QColor("#ff0000"))
img.fill(QtGui.QColor("#ff0000").rgb())
```

- `width()` — возвращает ширину изображения;
- `height()` — возвращает высоту изображения;
- `size()` — возвращает объект класса `QSize` с размерами изображения;
- `rect()` — возвращает объект класса `QRect` с координатами и размерами прямоугольной области, ограничивающей изображение;
- `depth()` — возвращает глубину цвета;
- `format()` — возвращает формат изображения в виде элемента перечисления `Format` из класса `QImage`;
- `setPixel()` — задает цвет пиксела с указанными координатами. Форматы метода:

```
setPixel(<X>, <Y>, <Индекс или цвет>)
setPixel(<Координаты QPoint>, <Индекс или цвет>)
```

В параметре <Индекс или цвет> у 8-битных изображений задается индекс цвета в палитре, а у 32-битных — целочисленное значение цвета, получить которое позволяют методы rgb() и rgba() класса QColor;

- pixel() — возвращает целочисленное значение цвета пиксела с указанными координатами. Это значение можно передать конструктору класса QColor, а затем получить значения различных составляющих цвета. Форматы метода:

```
pixel(<X>, <Y>)
pixel(<Координаты QPoint>)
```

- convertToFormat() — преобразует текущее изображение в указанный формат (задается в виде элемента перечисления Format из класса QImage) и возвращает новый объект класса QImage. Исходное изображение не изменяется. Форматы метода:

```
convertToFormat(<Формат>[, flags=ImageConversionFlag.AutoColor])
convertToFormat(<Формат>, <Таблица цветов>[,
                          flags=ImageConversionFlag.AutoColor])
```

- copy() — возвращает объект класса QImage с прямоугольным фрагментом изображения с заданными параметрами. Форматы метода:

```
copy([<Параметры фрагмента QRect>])
copy(<X>, <Y>, <Ширина>, <Высота>)
```

Если в первом формате параметры фрагмента не указаны, изображение копируется целиком;

- scaled() — изменяет размер изображения и возвращает результат в виде объекта класса QImage. Исходное изображение не изменяется. Форматы метода:

```
scaled(<Ширина>, <Высота>[,
       aspectRatioMode=AspectRatioMode.IgnoreAspectRatio][,
       transformMode=TransformationMode.FastTransformation])
scaled(<Размеры QSize>[,
       aspectRatioMode=AspectRatioMode.IgnoreAspectRatio][,
       transformMode=TransformationMode.FastTransformation])
```

В необязательном параметре aspectRatioMode могут быть указаны следующие элементы перечисления AspectRatioMode из модуля QtCore.Qt:

- IgnoreAspectRatio — изменяет размеры без сохранения пропорций сторон;
- KeepAspectRatio — изменяет размеры с сохранением пропорций сторон. При этом часть области нового изображения может оказаться незаполненной;
- KeepAspectRatioByExpanding — изменяет размеры с сохранением пропорций сторон. При этом часть нового изображения может выйти за пределы его области.

В необязательном параметре transformMode могут быть указаны следующие элементы перечисления TransformationMode из модуля QtCore.Qt:

- FastTransformation — сглаживание выключено;
- SmoothTransformation — сглаживание включено;

- scaledToWidth() — изменяет ширину изображения и возвращает результат в виде объекта класса QImage. Формат метода:

```
scaledToWidth(<Ширина>[, mode=TransformationMode.FastTransformation])
```

Высота изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр mode аналогичен параметру transformMode в методе scaled();

- scaledToHeight() — изменяет высоту изображения и возвращает результат в виде объекта класса QImage. Формат метода:

```
scaledToHeight(<Высота>[, mode=TransformationMode.FastTransformation])
```

Ширина изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр mode аналогичен параметру transformMode в методе scaled();

- transformed() — производит трансформацию изображения (например, поворот) и возвращает результат в виде объекта класса QImage. Формат метода:

```
transformed(<Трансформация QTransform>[,
            mode=TransformationMode.FastTransformation])
```

Исходное изображение не изменяется. Параметр mode аналогичен параметру transformMode в методе scaled();

- invertPixels([mode=InvertMode.InvertRgb]) — инвертирует значения всех пикселов в изображении. В необязательном параметре mode может быть указан элемент InvertRgb или InvertRgba перечисления InvertMode из класса QImage;
- mirror([horizontal=False][, vertical=True]) — зеркально отображает текущее изображение. Если значение True дано параметру horizontal, выполняется отображение по горизонтали, а если параметру vertical — по вертикали;
- mirrored() — аналогичен mirror(), только создает новый объект, хранящий зеркальную копию текущего изображения, и возвращает его. Текущее изображение не изменяется.

25.3.4. Класс *QIcon*

Класс QIcon представляет значки в различных размерах, режимах и состояниях. Он не наследует класс QPaintDevice, — следовательно, мы не можем использовать его как поверхность для рисования. Форматы конструктора:

```
QIcon()
QIcon(<Путь к файлу>)
QIcon(<Изображение QPixmap>)
QIcon(<Исходный объект QIcon>)
```

Первый конструктор создает пустой объект значка. Второй конструктор выполняет загрузку значка из файла, причем файл загружается при первой попытке использования, а не сразу. Третий конструктор создает значок на основе заданного изображения, а четвертый — создает копию исходного объекта.

Класс QIcon поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qicon.html**):

- isNull() — возвращает значение True, если объект является пустым, и False — в противном случае;
- addFile() — добавляет значок для указанных размера, режима и состояния. Формат метода:

```
addFile(<Путь к файлу>[, size=QSize()][, mode=Mode.Normal][, state=State.Off])
```

Можно добавить несколько значков, вызывая метод с разными значениями параметров. Параметр `size` задает размер значка в виде объекта класса `QSize` (поскольку загрузка значка производится при первой попытке использования, заранее размер значка неизвестен, и нам придется задать его явно). В параметре `mode` можно указать следующие элементы перечисления `Mode` из класса `QIcon`: `Normal` (обычное состояние компонента), `Disabled` (компонент недоступен), `Active` (компонент активен) или `Selected` (компонент выделен — обычно используется для элементов представлений). В параметре `state` указываются атрибуты `Off` (отключенное состояние) или `On` (включенное состояние) перечисления `State` из класса `QIcon`;

- `addPixmap()` — добавляет заданное изображение в качестве значка для указанного режима и состояния. Формат метода:

```
addPixmap(<Изображение QPixmap>[, mode=Mode.Normal][, state=State.Off])
```

- `availableSizes([mode=Mode.Normal][, state=State.Off])` — возвращает доступные размеры (список с объектами класса `QSize`) значков для указанного режима и состояния;
- `actualSize(<Размер QSize>[, mode=Mode.Normal][, state=State.Off])` — возвращает фактический размер (объект класса `QSize`) для указанного размера, режима и состояния. Фактический размер может быть меньше размера, указанного в первом параметре, но не больше;
- `pixmap()` — возвращает значок (объект класса `QPixmap`), который примерно соответствует указанному размеру, режиму и состоянию. Форматы метода:

```
pixmap(<Ширина>, <Высота>[, mode=Mode.Normal][, state=State.Off])
pixmap(<Ширина и высота>[, mode=Mode.Normal][, state=State.Off])
pixmap(<Размер QSize>[, mode=Mode.Normal][, state=State.Off])
```

Во втором формате первый параметр задает и ширину, и высоту значка, имеющего квадратную форму.

Вместо загрузки значка из файла можно воспользоваться одним из встроенных в PyQt стандартных значков. Загрузить стандартный значок позволяет следующий код:

```
ico = window.style().standardIcon(
                    QtWidgets.QStyle.StandardPixmap.SP_MessageBoxCritical)
```

Найти список всех встроенных значков можно в документации к классу `QStyle` на странице **https://doc.qt.io/qt-6/qstyle.html#StandardPixmap-enum**.

ГЛАВА 26

Графическая сцена

Графическая сцена — это поверхность, позволяющая размещать всевозможные графические объекты (линии, прямоугольники, многоугольники и т. п.) и производить над ними различные манипуляции (перемещать с помощью мыши, преобразовывать систему координат и др.). Для вывода графической сцены применяется особый компонент, называемый *графическим представлением*. Одна и та же графическая сцена может быть выведена в нескольких разных графических представлениях.

Все описанные в этой главе классы объявлены в модуле `QtWidgets`, если не указано иное.

26.1. Графическая сцена

Класс `QGraphicsScene` исполняет роль графический сцены. Иерархия наследования выглядит так:

```
QObject — QGraphicsScene
```

Форматы конструктора:

```
QGraphicsScene([parent=None])
QGraphicsScene(<X>, <Y>, <Ширина>, <Высота>[, parent=None])
QGraphicsScene(<Прямоугольник QRectF>[, parent=None])
```

Первый конструктор создает сцену, не имеющую определенного размера. Второй и третий конструкторы позволяют указать координаты и размеры сцены. В качестве параметра `parent` можно указать ссылку на родительский компонент.

26.1.1. Настройка графической сцены

Для настройки различных параметров графической сцены предназначены следующие методы класса `QGraphicsScene`:

- `setSceneRect()` — задает координаты и размеры сцены. Форматы метода:

  ```
  setSceneRect(<X>, <Y>, <Ширина>, <Высота>)
  setSceneRect(<Прямоугольник QRectF>)
  ```

- `sceneRect()` — возвращает объект класса `QRectF` с координатами и размерами сцены;
- `width()` и `height()` — возвращают ширину и высоту сцены соответственно в виде вещественных чисел;

- `itemsBoundingRect()` — возвращает объект класса `QRectF` с координатами и размерами прямоугольника, в который вписываются все объекты, расположенные на сцене;
- `setBackgroundBrush(<Кисть QBrush>)` — задает кисть для заднего плана (фона), расположенного под графическими объектами. Изменить задний план также можно, переопределив у сцены метод `drawBackground()` и выполнив перерисовку заднего плана внутри него;
- `setForegroundBrush(<Кисть QBrush>)` — задает кисть для переднего плана, расположенного над графическими объектами. Изменить передний план также можно, переопределив у сцены метод `drawForeground()` и выполнив перерисовку переднего плана внутри него;
- `setFont(<Шрифт QFont>)` — задает шрифт сцены по умолчанию;
- `setItemIndexMethod(<Режим>)` — задает режим индексации объектов сцены для ускорения их поиска. В качестве параметра указываются следующие элементы перечисления `ItemIndexMethod` из класса `QGraphicsScene`:
  - `BspTreeIndex` — для поиска объектов используется индекс в виде бинарного дерева. Этот режим следует применять для сцен, большинство объектов которых являются статическими;
  - `NoIndex` — индекс не используется. Этот режим следует применять для сцен, содержимое которых часто меняется;
- `setBspTreeDepth(<Глубина>)` — задает целочисленную глубину дерева при использовании режима `BspTreeIndex`. По умолчанию установлено значение 0, которое говорит, что глубина выбирается автоматически;
- `bspTreeDepth()` — возвращает текущее значение глубины дерева при использовании режима `BspTreeIndex`.

26.1.2. Добавление и удаление графических объектов

Для добавления графических объектов на сцену и удаления их оттуда предназначены следующие методы класса `QGraphicsScene`:

- `addItem(<Графический объект>)` — добавляет графический объект на сцену;
- `addLine()` — создает линию, добавляет ее на сцену и возвращает ссылку на представляющий ее объект класса `QGraphicsLineItem`. Форматы метода:

```
addLine(<X1>, <Y1>, <X2>, <Y2>[, pen=QPen()])
addLine(<Линия QLineF>[, pen=QPen()])
```

- `addRect()` — создает прямоугольник, добавляет его на сцену и возвращает ссылку на представляющий его объект класса `QGraphicsRectItem`. Форматы метода:

```
addRect(<X>, <Y>, <Ширина>, <Высота>[, pen=QPen()][, brush=QBrush()])
addRect(<Прямоугольник QRectF>[, pen=QPen()][, brush=QBrush()])
```

- `addPolygon()` — создает многоугольник, добавляет его на сцену и возвращает ссылку на представляющий его объект класса `QGraphicsPolygonItem`. Формат метода:

```
addPolygon(<Многоугольник QPolygonF>[, pen=QPen()][, brush=QBrush()])
```

- `addEllipse()` — создает эллипс, добавляет его на сцену и возвращает ссылку на представляющий его объект класса `QGraphicsEllipseItem`. Форматы метода:

```
addEllipse(<X>, <Y>, <Ширина>, <Высота>[, pen=QPen()][, brush=QBrush()])
addEllipse(<Прямоугольник QRectF>[, pen=QPen()][, brush=QBrush()])
```

- `addPixmap(<Изображение QPixmap>)` — создает изображение, добавляет его на сцену и возвращает ссылку на представляющий его объект класса `QGraphicsPixmapItem`;
- `addSimpleText(<Текст>[, font=QFont()])` — создает фрагмент простого текста, добавляет его на сцену в позицию с координатами `(0, 0)` и возвращает ссылку на представляющий его объект класса `QGraphicsSimpleTextItem`;
- `addText(<Текст>[, font=QFont()])` — создает фрагмент форматированного текста, добавляет его на сцену в позицию с координатами `(0, 0)` и возвращает ссылку на представляющий его объект класса `QGraphicsTextItem`;
- `addPath(<Путь QPainterPath>[, pen=QPen()][, brush=QBrush()])` — создает сложную фигуру («путь»), добавляет ее на сцену и возвращает ссылку на представляющий ее объект класса `QGraphicsPathItem`;
- `removeItem(<Графический объект>)` — убирает заданный графический объект и всех его потомков со сцены. Сам объект при этом не удаляется и, например, может быть добавлен на другую сцену;
- `clear()` — удаляет все элементы со сцены. Метод является слотом;
- `createItemGroup(<Список с графическими объектами>)` — объединяет объекты из заданного списка в группу, добавляет ее на сцену и возвращает представляющий созданную группу объект класса `QGraphicsItemGroup`;
- `destroyItemGroup(<Группа QGraphicsItemGroup>)` — удаляет заданную группу со сцены, при этом сохраняя все содержащиеся в группе элементы.

26.1.3. Добавление компонентов на сцену

Помимо графических объектов, на сцену можно добавить компоненты, которые будут функционировать как обычно. Добавить компонент на сцену позволяет метод `addWidget(<Компонент>[, flags=0])` класса `QGraphicsScene`. В параметре `flags` задается тип окна (см. *разд. 19.1.1*). Метод возвращает ссылку на объект класса `QGraphicsProxyWidget`, представляющий добавленный компонент.

26.1.4. Поиск графических объектов

Для поиска графических объектов, находящихся на сцене, предназначены следующие методы класса `QGraphicsScene`:

- `itemAt()` — возвращает ссылку на верхний видимый объект, который расположен по указанным координатам, или значение `None`, если никакого объекта там нет. Форматы метода:

```
itemAt(<X>, <Y>, <Преобразования QTransform>)
itemAt(<Координаты QPointF>, <Преобразования QTransform>)
```

Третий параметр задает примененные к сцене преобразования системы координат (см. *разд. 25.2.4*). Его необходимо указывать, если на сцене присутствуют объекты, игнорирующие преобразования. В противном случае следует указать пустой объект класса `QTransform`;

- `collidingItems()` — возвращает список со ссылками на объекты, которые находятся в указанном в первом параметре объекте или пересекаются с ним. Если таких объектов нет, метод возвращает пустой список. Формат метода:

```
collidingItems(<Графический объект>[,
               mode=ItemSelectionMode.IntersectsItemShape])
```

Необязательный параметр `mode` указывает режим поиска графических объектов и должен задаваться следующими элементами перечисления `ItemSelectionMode` из модуля `QtCore.Qt`:

  - `ContainsItemShape` — ищутся объекты, чьи границы полностью находятся внутри заданного объекта;
  - `IntersectsItemShape` — ищутся объекты, чьи границы полностью находятся внутри заданного объекта или пересекаются с его границей;
  - `ContainsItemBoundingRect` — ищутся объекты, чьи охватывающие прямоугольники (т. е. прямоугольники минимальных размеров, в которые искомые объекты помещаются полностью) полностью находятся внутри охватывающего прямоугольника заданного объекта;
  - `IntersectsItemBoundingRect` — ищутся объекты, чьи охватывающие прямоугольники (т. е. прямоугольники минимальных размеров, в которые искомые объекты помещаются полностью) полностью находятся внутри охватывающего прямоугольника заданного объекта или пересекаются с его границами;

- `items()` — возвращает список со ссылками на все объекты, или на объекты, расположенные по указанным координатам, или на объекты, попадающие в указанную область. Если объектов нет, то возвращается пустой список. Форматы метода:

```
items([order=SortOrder.DescendingOrder])
items(<Координаты QPointF>[, mode=ItemSelectionMode.IntersectsItemShape][,
      order=SortOrder.DescendingOrder][, deviceTransform=QTransform()])
items(<X>, <Y>, <Ширина>, <Высота>[,
      mode=ItemSelectionMode.IntersectsItemShape][,
      order=SortOrder.DescendingOrder][, deviceTransform=QTransform()])
items(<Координаты QRectF>[, mode=ItemSelectionMode.IntersectsItemShape][,
      order=SortOrder.DescendingOrder][, deviceTransform=QTransform()])
items(<Многоугольник QPolygonF>[,
      mode=ItemSelectionMode.IntersectsItemShape][,
      order=SortOrder.DescendingOrder][, deviceTransform=QTransform()])
items(<Путь QPainterPath>[, mode=ItemSelectionMode.IntersectsItemShape][,
      order=SortOrder.DescendingOrder][, deviceTransform=QTransform()])
```

Параметр `mode` аналогичен таковому у метода `collidingItems()`. В необязательном параметре `order`, задающем порядок сортировки объектов, указываются атрибуты `AscendingOrder` (в алфавитном порядке) или `DescendingOrder` (в обратном порядке) перечисления `SordOrder` из модуля `QtCore.Qt`.

Необязательный параметр `deviceTransform` задает примененные к сцене преобразования системы координат (см. *разд. 25.2.4*). Его необходимо указывать, если на сцене присутствуют объекты, игнорирующие преобразования.

26.1.5. Управление фокусом ввода

Получить фокус ввода с клавиатуры может как сцена в целом, так и отдельный объект на ней. Если фокус установлен на отдельный объект, все события клавиатуры перенаправляются этому объекту. Чтобы объект мог принимать фокус ввода, необходимо установить флаг `ItemIsFocusable` — например, с помощью метода `setFlag()` класса `QGraphicsItem`. Для управления фокусом ввода предназначены следующие методы класса `QGraphicScene`:

- `setFocus([focusReason=FocusReason.OtherFocusReason])` — устанавливает фокус ввода на сцену. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 20.8.1*);
- `setFocusItem()` — устанавливает фокус ввода на указанный графический объект на сцене. Формат метода:

  ```
  setFocusItem(<Графический объект>[, focusReason=FocusReason.OtherFocusReason])
  ```

  Если сцена была вне фокуса ввода, она автоматически получит фокус. Если в первом параметре указано значение `None`, или объект не может принимать фокус, метод просто убирает фокус с объекта, обладающего им в текущий момент. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 20.8.1*);
- `clearFocus()` — убирает фокус ввода со сцены. Объект, который обладает фокусом ввода в текущий момент, потеряет его, но получит его снова, когда фокус будет опять установлен на сцену;
- `hasFocus()` — возвращает значение `True`, если сцена имеет фокус ввода, и `False` — в противном случае;
- `focusItem()` — возвращает ссылку на объект, который обладает фокусом ввода, или значение `None`;
- `setStickyFocus(<Флаг>)` — если в качестве параметра указано значение `True`, то при щелчке мышью на фоне сцены или на объекте, который не может принимать фокус, объект, владеющий фокусом, не потеряет его. По умолчанию фокус убирается;
- `stickyFocus()` — возвращает значение `True`, если фокус ввода не будет убран с объекта при щелчке мышью на фоне или на объекте, который не может принимать фокус.

26.1.6. Управление выделением объектов

Чтобы объект можно было выделить (с помощью мыши или программно), необходимо установить флаг `ItemIsSelectable` — например, с помощью метода `setFlag()` класса `QGraphicsItem`. Для управления выделением объектов предназначены следующие методы класса `QGraphicScene`:

- `setSelectionArea()` — выделяет объекты внутри указанной области. Чтобы выделить только один объект, следует воспользоваться методом `setSelected()` класса `QGraphicsItem`. Форматы метода `setSelectionArea()`:

  ```
  setSelectionArea(<Путь QPainterPath>, <Преобразования QTransform>)
  setSelectionArea(<Путь QPainterPath>[,
                   mode=ItemSelectionMode.IntersectsItemShape][,
                   deviceTransform=QTransform()])
  setSelectionArea(<Путь QPainterPath>, <Операция>[,
                   mode=ItemSelectionMode.IntersectsItemShape][,
                   deviceTransform=QTransform()])
  ```

Параметр `mode` аналогичен таковому у метода `collidingItems()` (см. *разд. 26.1.4*). Второй параметр первого формата и параметр `deviceTransform` второго формата задают примененные к сцене преобразования системы координат (см. *разд. 25.2.4*).

Третий формат позволяет дополнительно указать, какую операцию следует выполнить с ранее выделенными объектами сцены, в виде одного из следующих элементов перечисления `ItemSelectionOperation` из модуля `QtCore.Qt`:

- `ReplaceSelection` — ранее выделенные объекты перестанут быть выделенными;
- `AddToSelection` — ранее выделенные объекты останутся выделенными;

- `selectionArea()` — возвращает область выделения в виде объекта класса `QPainterPath`;
- `selectedItems()` — возвращает список со ссылками на выделенные объекты или пустой список, если выделенных объектов нет;
- `clearSelection()` — снимает выделение. Метод является слотом.

26.1.7. Прочие методы и сигналы

Помимо рассмотренных ранее, класс `QGraphicsScene` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsscene.html**):

- `isActive()` — возвращает значение `True`, если сцена отображается на экране с помощью какого-либо графического представления, и `False` — в противном случае;
- `views()` — возвращает список с графическими представлениями (объектами класса `QGraphicsView`), в которых выводится сцена. Если сцена вообще не выводится на экран, возвращается пустой список;
- `mouseGrabberItem()` — возвращает ссылку на объект, который владеет мышью, или `None`, если такого объекта нет;
- `render()` — выводит сцену на устройство рисования или принтер. Формат метода:

```
render(<Устройство QPainter>[, target=QRectF()][, source=QRectF()][,
                             mode=AspectRatioMode.KeepAspectRatio])
```

Параметр `target` задает координаты и размеры устройства рисования, а параметр `source` — координаты и размеры прямоугольной области на сцене. Если параметры не указаны, используются размеры устройства рисования и сцены. В параметре `mode`, задающем режим изменения размеров графики при выводе, могут быть указаны следующие элементы перечисления `AspectRatioMode` из модуля `QtCore.Qt`:

- `IgnoreAspectRatio` — изменяет размеры без сохранения пропорций сторон;
- `KeepAspectRatio` — изменяет размеры с сохранением пропорций сторон. При этом часть области на устройстве рисования может оказаться незаполненной;
- `KeepAspectRatioByExpanding` — изменяет размеры с сохранением пропорций сторон. При этом часть выводимой графики может выйти за пределы области на устройстве рисования;

- `invalidate()` — вызывает перерисовку заданных слоев внутри прямоугольной области с указанными параметрами. Форматы метода:

```
invalidate(<X>, <Y>, <Ширина>, <Высота>[, layers=SceneLayers.AllLayers])
invalidate([rect=QRectF()][, layers=SceneLayers.AllLayers])
```

В параметре `layers`, задающем слои, которые требуется перерисовать, могут быть указаны следующие элементы перечисления `SceneLayers` из класса `QGraphicsScene`:

- `ItemLayer` — слой объекта;
- `BackgroundLayer` — слой заднего плана;
- `ForegroundLayer` — слой переднего плана;
- `AllLayers` — все слои. Сначала отрисовывается слой заднего плана, затем — слой объекта и в конце — слой переднего плана.

Второй формат метода является слотом;

- `update()` — вызывает перерисовку прямоугольной области с указанными параметрами. Форматы метода:

```
update(<X>, <Y>, <Ширина>, <Высота>)
update([rect=QRectF()])
```

Второй формат метода является слотом.

Класс `QGraphicsScene` поддерживает следующие сигналы:

- `changed(<Список областей>)` — генерируется при изменении сцены. Внутри обработчика через параметр доступен список с объектами класса `QRectF`, представляющими изменившиеся области сцены;
- `sceneRectChanged(<Размеры QRectF>)` — генерируется при изменении размеров сцены. Внутри обработчика через параметр доступен объект класса `QRectF` с новыми координатами и размерами сцены;
- `selectionChanged()` — генерируется при изменении выделения объектов;
- `focusItemChanged(<Объект 1>, <Объект 2>, <Причина изменения фокуса>)` — генерируется при изменении фокуса клавиатурного ввода. Через параметры доступны объект, получивший фокус ввода, объект, потерявший фокус ввода, и причина изменения фокуса (см. *разд. 20.8.1*).

26.2. Графическое представление

Класс `QGraphicsView` реализует графическое представление, выводящее графическую сцену. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea — QGraphicsView
```

Форматы конструктора класса:

```
QGraphicsView([parent=None])
QGraphicsView(<Графическая сцена>[, parent=None])
```

Второй формат сразу же позволяет установить в представлении выводимую сцену.

26.2.1. Настройка графического представления

Для настройки различных параметров представления применяются следующие методы класса `QGraphicsView` (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsview.html**):

- `setScene(<Графическая сцена>)` — устанавливает выводимую сцену;
- `scene()` — возвращает ссылку на установленную сцену в виде объекта класса `QGraphicsScene`;
- `setSceneRect()` — задает координаты и размеры сцены. Форматы метода:

```
setSceneRect(<X>, <Y>, <Ширина>, <Высота>)
setSceneRect(<Прямоугольник QRectF>)
```

- `sceneRect()` — возвращает объект класса `QRectF` с координатами и размерами сцены;
- `setBackgroundBrush(<Кисть QBrush>)` — задает кисть для заднего плана (фона) сцены (расположен под графическими объектами);
- `setForegroundBrush(<Кисть QBrush>)` — задает кисть для переднего плана сцены (расположен над графическими объектами);
- `setCacheMode(<Режим>)` — задает режим кеширования выводимых объектов. В качестве параметра могут быть указаны следующие элементы перечисления `CacheModeFlag` из класса `QGraphicsView`:
  - `CacheNone` — без кеширования;
  - `CacheBackground` — кешируется только задний план;
- `resetCachedContent()` — сбрасывает кеш;
- `setAlignment(<Выравнивание AlignmentFlag>)` — задает выравнивание сцены в случае, если содержимое сцены полностью помещается в представлении. По умолчанию сцена центрируется по горизонтали и вертикали. Пример установки сцены в левом верхнем углу представления:

```
view.setAlignment(QtCore.Qt.AlignmentFlag.AlignLeft |
                  QtCore.Qt.AlignmentFlag.AlignTop)
```

- `setInteractive(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может взаимодействовать с объектами на сцене (интерактивный режим, используется по умолчанию). Значение `False` разрешает только просмотр сцены;
- `isInteractive()` — возвращает значение `True`, если задан интерактивный режим, и `False` — в противном случае;
- `setDragMode(<Режим>)` — задает действие, производимое при щелчке левой кнопкой мыши на фоне и перемещении мыши. В качестве параметра могут быть указаны следующие элементы перечисления `DragMode` из класса `QGraphicsView`:
  - `NoDrag` — никакого действия;
  - `ScrollHandDrag` — прокрутка сцены. При этом указатель мыши примет вид сжатой или разжатой руки;
  - `RubberBandDrag` — создание области выделения. Объекты, частично или полностью (задается с помощью метода `setRubberBandSelectionMode()`) попавшие в эту область, будут выделены (при условии, что у объектов установлен флаг `ItemIsSelectable`). Действие выполняется только в интерактивном режиме;
- `dragMode()` — возвращает обозначение действия, производимого при щелчке левой кнопкой мыши на фоне и перемещении мыши, в виде элемента перечисления `DragMode` из класса `QGraphicsView`;

- setRubberBandSelectionMode(<Режим>) — задает режим выделения объектов при установленном флаге RubberBandDrag. В параметре могут быть указаны элементы перечисления ItemSelectionMode из модуля QtCore.Qt, описанные в *разд. 26.1.4*;
- setRenderHint(<Опция вывода>[, <Флаг>]) — управляет опциями, влияющими на качество отображения сцены. Если вторым параметром передано значение True или если второй параметр вообще не указан, <Опция вывода> активизируется, если же передано False, опция деактивируется. Сама <Опция вывода> указывается в виде одного из следующих элементов перечисления RenderHint из класса QPainter:
  - Antialiasing — выполнять сглаживание краев у графических объектов;
  - TextAntialiasing — выполнять сглаживание текста. Активизирован по умолчанию;
  - SmoothPixmapTransform — использовать более качественное сглаживание при трансформациях растровых изображений;
  - VerticalSubpixelPositioning — выполнять субпиксельное позиционирование текста не только по горизонтали, но и по вертикали. Обычно применяется при выравнивании текста относительно других графических объектов;
  - LosslessImageRendering — выполнять вывод графики без потерь. Учитывается только при сохранении изображения в формате PDF.

  Пример активизирования для представления всех доступных опций вывода:

```
view.setRenderHint(QtGui.QPainter.RenderHint.Antialiasing)
view.setRenderHint(QtGui.QPainter.RenderHint.TestAntialiasing)
view.setRenderHint(QtGui.QPainter.RenderHint.SmoothPixmapTransform)
view.setRenderHint(QtGui.QPainter.RenderHint.VerticalSubpixelPositioning)
view.setRenderHint(QtGui.QPainter.RenderHint.LosslessImageRendering)
```

- setRenderHints(<Опции вывода>) — активизирует сразу все указанные <Опции вывода> (те, что не были указаны, деактивируются). <Опции вывода> задаются в виде комбинации описанных ранее элементов перечисления RenderHint, указанных через оператор |.

  Пример активизирования для представления всех доступных опций:

```
view.setRenderHints(QtGui.QPainter.RenderHint.Antialiasing |
                    QtGui.QPainter.RenderHint.TestAntialiasing |
                    QtGui.QPainter.RenderHint.SmoothPixmapTransform |
                    QtGui.QPainter.RenderHint.VerticalSubpixelPositioning |
                    QtGui.QPainter.RenderHint.LosslessImageRendering)
```

26.2.2. Преобразования между координатами представления и сцены

Для преобразования между координатами представления и сцены предназначены следующие методы класса QGraphicsView:

- mapFromScene() — преобразует заданные координаты точки из системы координат сцены в систему координат представления. Форматы метода (справа указан тип возвращаемого значения):

```
mapFromScene(<X>, <Y>)                          -> QPoint
mapFromScene(<QPointF>)                         -> QPoint
mapFromScene(<X>, <Y>, <Ширина>, <Высота>) -> QPolygon
```

```
mapFromScene(<QRectF>)                    -> QPolygon
mapFromScene(<QPolygonF>)                 -> QPolygon
mapFromScene(<QPainterPath>)              -> QPainterPath
```

- `mapToScene()` — преобразует заданные координаты точки из системы координат представления в систему координат сцены. Форматы метода (справа указан тип возвращаемого значения):

```
mapToScene(<X>, <Y>)                      -> QPointF
mapToScene(<QPoint>)                      -> QPointF
mapToScene(<X>, <Y>, <Ширина>, <Высота>)  -> QPolygonF
mapToScene(<QRect>)                       -> QPolygonF
mapToScene(<QPolygon>)                    -> QPolygonF
mapToScene(<QPainterPath>)                -> QPainterPath
```

26.2.3. Поиск объектов

Для поиска объектов на сцене предназначены следующие методы класса `QGraphicsView`:

- `itemAt()` — возвращает ссылку на верхний видимый объект, который расположен по указанным координатам, или значение `None`, если никакого объекта там нет. В качестве значений указываются координаты в системе координат представления, а не сцены. Форматы метода:

```
itemAt(<X>, <Y>)
itemAt(<Координаты QPoint>)
```

- `items()` — возвращает список со ссылками на все объекты, или на объекты, расположенные по указанным координатам, или на объекты, попадающие в указанную область. Если объектов нет, возвращается пустой список. В качестве значений указываются координаты в системе координат представления, а не сцены. Форматы метода:

```
items()
items(<X>, <Y>)
items(<Координаты QPoint>)
items(<X>, <Y>, <Ширина>, <Высота>[,
                          mode=ItemSelectionMode.IntersectsItemShape])
items(<Прямоугольник QRect>[, mode=ItemSelectionMode.IntersectsItemShape])
items(<Многоугольник QPolygon>[, mode=ItemSelectionMode.IntersectsItemShape])
items(<Путь QPainterPath>[, mode=ItemSelectionMode.IntersectsItemShape])
```

Допустимые значения параметра `mode` были рассмотрены в *разд. 26.1.4*.

26.2.4. Преобразование системы координат

Выполнить преобразование системы координат позволяют следующие методы класса `QGraphicsView`:

- `translate(<X>, <Y>)` — перемещает начало координат в точку с указанными координатами. По умолчанию начало координат находится в левом верхнем углу, ось `X` направлена вправо, а ось `Y` — вниз;

- `rotate(<Угол>)` — поворачивает систему координат на заданный угол (указывается в виде вещественного числа в градусах). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки;

- `scale(<По оси X>, <По оси Y>)` — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы — то увеличение;
- `shear(<По горизонтали>, <По вертикали>)` — сдвигает систему координат. В качестве значений указываются вещественные числа;
- `resetTransform()` — отменяет все преобразования.

Несколько преобразований можно произвести последовательно друг за другом. При этом надо учитывать, что порядок следования преобразований имеет значение.

Если одна и та же последовательность преобразований выполняется несколько раз, то ее можно сохранить в объекте класса `QTransform`, а затем установить с помощью метода `setTransform()`. Получить ссылку на установленное преобразование позволяет метод `transform()`.

26.2.5. Прочие методы

Помимо рассмотренных ранее, класс `QGraphicsView` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsview.html**):

- `centerOn()` — прокручивает представление таким образом, чтобы указанная точка или объект находились в центре видимой области. Форматы метода:

```
centerOn(<X>, <Y>)
centerOn(<Координаты QPointF>)
centerOn(<Графический объект>)
```

- `ensureVisible()` — прокручивает представление таким образом, чтобы указанный прямоугольник или объект находились в видимой области. Форматы метода:

```
ensureVisible(<X>, <Y>, <Ширина>, <Высота>[, xMargin=50][, yMargin=50])
ensureVisible(<Прямоугольник QRectF>[, xMargin=50][, yMargin=50])
ensureVisible(<Графический объект>[, xMargin=50][, yMargin=50])
```

  Необязательные параметры `xMargin` и `yMargin` задают минимальные значения просветов между границами графического объекта и краями представления по горизонтали и вертикали соответственно;

- `fitInView()` — прокручивает и масштабирует представление таким образом, чтобы указанный прямоугольник или объект занимали всю видимую область. Форматы метода:

```
fitInView(<X>, <Y>, <Ширина>, <Высота>[, mode=AspectRatioMode.IgnoreAspectRatio])
fitInView(<Прямоугольник QRectF>[, mode=AspectRatioMode.IgnoreAspectRatio])
fitInView(<Графический объект>[, mode=AspectRatioMode.IgnoreAspectRatio])
```

  Необязательный параметр `mode` задает режим изменения размеров. Его значения рассматривались в *разд. 26.1.7* (см. описание метода `render()` класса `QGraphicsScene`);

- `render()` — выводит содержимое представления на устройство рисования или принтер. Формат метода:

```
render(<Устройство рисования>[, target=QRectF()][, source=QRectF()][,
                                mode=AspectRatioMode.KeepAspectRatio])
```

Назначение параметров этого метода сходно с таковым у метода `render()` класса `QGraphicsScene` (см. *разд. 26.1.7*);

- `invalidateScene([rect=QRectF()][, layers=SceneLayers.AllLayers])` — вызывает перерисовку указанных слоев внутри заданной прямоугольной области на сцене. Назначение его параметров сходно с таковым у одноименного метода класса `QGraphicsScene` (см. *разд. 26.1.7*). Метод является слотом;
- `updateSceneRect(<Прямоугольник QRectF>)` — вызывает перерисовку указанной прямоугольной области сцены. Метод является слотом;
- `updateScene(<Список с прямоугольниками QRectF>)` — вызывает перерисовку указанных прямоугольных областей. Метод является слотом.

26.3. Графические объекты

26.3.1. Класс *QGraphicsItem*: базовый класс для графических объектов

Абстрактный класс `QGraphicsItem` является базовым классом для графических объектов. Формат конструктора класса:

```
QGraphicsItem([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский объект (объект класса, производного от `QGraphicsItem`).

Поскольку класс `QGraphicsItem` является абстрактным, создать его объект нельзя. Чтобы создать новый графический объект, следует наследовать этот класс и переопределить как минимум методы `boundingRect()` и `paint()`. Метод `boundingRect()` должен возвращать объект класса `QRectF` с координатами и размерами прямоугольной области, ограничивающей объект. Внутри метода `paint()` необходимо выполнить рисование объекта. Формат метода `paint()`:

```
paint(self, <Устройство рисования QPainter>,
            <Параметры перерисовки QStyleOptionGraphicsItem>, widget=None)
```

Для обработки столкновений следует также переопределить метод `shape()`. Метод должен возвращать объект класса `QPainterPath`.

26.3.1.1. Настройка графического объекта

Для настройки различных параметров объекта предназначены следующие методы класса `QGraphicsItem` (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsitem.html**):

- `setPos()` — задает позицию объекта относительно родителя или сцены (при отсутствии родителя). Форматы метода:

  ```
  setPos(<X>, <Y>)
  setPos(<Координаты QPointF>)
  ```

- `pos()` — возвращает объект класса `QPointF` с текущими координатами относительно родителя или сцены (при отсутствии родителя);

- `scenePos()` — возвращает объект класса `QPointF` с текущими координатами относительно сцены;
- `sceneBoundingRect()` — возвращает объект класса `QRectF`, который содержит координаты (относительно сцены) и размеры прямоугольника, ограничивающего объект;
- `setX(<X>)` и `setY(<Y>)` — задают позицию объекта по отдельным осям;
- `x()` и `y()` — возвращают позицию объекта по отдельным осям;
- `setZValue(<Z>)` — задает позицию объекта по оси `Z`. Объект с бóльшим значением этого параметра рисуется выше объекта с меньшим значением. По умолчанию для всех объектов значение позиции по оси `Z` равно `0.0`;
- `zValue()` — возвращает позицию объекта по оси `Z`;
- `moveBy(<По оси X>, <По оси Y>)` — сдвигает объект на указанное смещение относительно текущей позиции;
- `prepareGeometryChange()` — этот метод следует вызвать перед изменением размеров объекта, чтобы поддержать индекс сцены в актуальном состоянии;
- `scene()` — возвращает ссылку на сцену (объект класса `QGraphicsScene`) или значение `None`, если объект не помещен на сцену;
- `setFlag(<Флаг>[, enabled=True])` — устанавливает (если второй параметр имеет значение `True`) или сбрасывает (если второй параметр имеет значение `False`) заданный флаг. В первом параметре могут быть указаны следующие элементы перечисления `GraphicItemFlag` из класса `QGraphicsItem` (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsitem.html #GraphicsItemFlag-enum**):
  - `ItemIsMovable` — объект можно перемещать с помощью мыши;
  - `ItemIsSelectable` — объект можно выделять;
  - `ItemIsFocusable` — объект может получать фокус ввода;
  - `ItemIgnoresTransformations` — объект игнорирует наследуемые преобразования;
  - `ItemIgnoresParentOpacity` — объект игнорирует прозрачность родителя;
  - `ItemDoesntPropagateOpacityToChildren` — прозрачность объекта не распространяется на потомков;
  - `ItemStacksBehindParent` — объект располагается ниже родителя;
  - `ItemIsPanel` — объект является панелью;
- `setFlags(<Флаги>)` — устанавливает сразу несколько флагов. Элементы перечисления `GraphicItemFlag` (см. описание метода `setFlag()`) указываются через оператор `|`;
- `flags()` — возвращает комбинацию установленных флагов (см. описание метода `setFlag()`);
- `setOpacity(<Число>)` — задает степень прозрачности объекта. В качестве значения указывается вещественное число от `0.0` (полностью прозрачный) до `1.0` (полностью непрозрачный);
- `opacity()` — возвращает степень прозрачности объекта;
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;

- `setCursor(<Курсор>)` — задает внешний вид курсора мыши при наведении его на объект (см. *разд. 20.9.5*);
- `unsetCursor()` — отменяет изменение курсора мыши;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `True`, то объект будет видим. Значение `False` скрывает объект;
- `show()` — делает объект видимым;
- `hide()` — скрывает объект;
- `isVisible()` — возвращает значение `True`, если объект видим, и `False` — если скрыт;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, объект будет доступен. Значение `False` делает объект недоступным. Недоступный объект не получает никаких событий, и его нельзя выделить;
- `isEnabled()` — возвращает значение `True`, если объект доступен, и `False` — если недоступен;
- `setSelected(<Флаг>)` — если в качестве параметра указано значение `True`, объект будет выделен. Значение `False` снимает выделение. Чтобы объект можно было выделить, необходимо установить флаг `ItemIsSelectable` — например, с помощью метода `setFlag()`;
- `isSelected()` — возвращает значение `True`, если объект выделен, и `False` — в противном случае;
- `setFocus([focusReason=FocusReason.OtherFocusReason])` — устанавливает фокус ввода на объект. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 20.8.1*). Чтобы объект мог принимать фокус ввода, необходимо установить флаг `ItemIsFocusable` — например, с помощью метода `setFlag()`;
- `clearFocus()` — убирает фокус ввода с объекта;
- `hasFocus()` — возвращает значение `True`, если объект находится в фокусе ввода, и `False` — в противном случае;
- `grabKeyboard()` — захватывает ввод с клавиатуры;
- `ungrabKeyboard()` — освобождает ввод с клавиатуры;
- `grabMouse()` — захватывает мышь;
- `ungrabMouse()` — освобождает мышь.

26.3.1.2. Выполнение преобразований

Задать преобразования для графического объекта можно, воспользовавшись классом `QTransform`. Для работы с преобразованиями, заданными объектами этого класса, класс `QGraphicsItem` поддерживает следующие методы:

- `setTransform(<Преобразование QTransform>[, combine=False])` — устанавливает преобразования, заданные в первом параметре. Если вторым параметром передать значение `True`, новые преобразования будут объединены с уже установленными, — в противном случае они заменят установленные ранее преобразования;
- `transform()` — возвращает объект класса `QTransform`, представляющий заданные для объекта преобразования;
- `sceneTransform()` — возвращает объект класса `QTransform`, который представляет преобразования, заданные у самой графической сцены.

Есть и более простой способ задания преобразований — использование следующих методов класса QGraphicsItem:

- setTransformOriginPoint() — перемещает начало координат в указанную точку. Форматы метода:

```
setTransformOriginPoint(<X>, <Y>)
setTransformOriginPoint(<Координаты QPointF>)
```

- setRotation(<Угол>) — поворачивает систему координат на указанное количество градусов (указывается вещественное число): положительное значение вызывает поворот по часовой стрелке, а отрицательное — против часовой стрелки;
- rotation() — возвращает текущий угол поворота;
- setScale(<Значение>) — масштабирует систему координат. В качестве значений указываются вещественные числа: если значение меньше единицы, выполняется уменьшение, а если больше — увеличение;
- scale() — возвращает текущий масштаб;
- resetTransform() — отменяет все преобразования.

26.3.1.3. Прочие методы

Помимо рассмотренных ранее, класс QGraphicsItem поддерживает следующие полезные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsitem.html**):

- setParentItem(<Графический объект>) — назначает заданный графический объект родителем для текущего объекта. Местоположение дочернего объекта задается в координатах родительского объекта;
- parentItem() — возвращает ссылку на родительский объект;
- topLevelItem() — возвращает ссылку на родительский объект верхнего уровня;
- childItems() — возвращает список с дочерними объектами;
- contains(<Координаты QPointF>) — возвращает True, если точка с указанными координатами входит в состав текущего графического объекта, и False — в противном случае;
- collidingItems([mode=ItemSelectionMode.IntersectsItemShape]) — возвращает список со ссылками на объекты, которые находятся в текущем объекте или пересекаются с ним. Если таких объектов нет, возвращается пустой список. Возможные значения параметра mode описаны в *разд. 26.1.4*;
- collidesWithItem() — возвращает значение True, если текущий объект находится в указанном объекте или пересекается с ним. Формат метода:

```
collidesWithItem(<Графический объект>[,
                 mode=ItemSelectionMode.IntersectsItemShape])
```

  Возможные значения параметра mode см. в *разд. 26.1.4*;

- collidesWithPath() — возвращает значение True, если текущий объект находится внутри указанного пути или пересекается с ним. Формат метода:

```
collidesWithPath(<Путь QGraphicsPath>[,
                 mode=ItemSelectionMode.IntersectsItemShape])
```

  Возможные значения параметра mode см. в *разд. 26.1.4*;

- `ensureVisible()` — прокручивает представление таким образом, чтобы указанный прямоугольник находился в видимой области. Форматы метода:

```
ensureVisible(<X>, <Y>, <Ширина>, <Высота>[, xMargin=50][, yMargin=50])
ensureVisible([rect=QRectF()][, xMargin=50][, yMargin=50])
```

  Необязательные параметры `xMargin` и `yMargin` задают минимальные значения просветов между границами графического объекта и краями представления по горизонтали и вертикали соответственно;

- `update()` — вызывает перерисовку прямоугольной области с указанными параметрами. Форматы метода:

```
update(<X>, <Y>, <Ширина>, <Высота>)
update([rect=QRectF()])
```

26.3.2. Готовые графические объекты

Вместо создания собственного объекта путем наследования класса `QgraphicsItem` можно воспользоваться стандартными классами, описываемыми далее.

26.3.2.1. Линия

Класс `QGraphicsLineItem` описывает линию. Иерархия наследования:

```
QGraphicsItem — QGraphicsLineItem
```

Форматы конструктора класса:

```
QGraphicsLineItem([parent=None])
QGraphicsLineItem(<X1>, <Y1>, <X2>, <Y2>[, parent=None])
QGraphicsLineItem(<Линия QLineF>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsLineItem` наследует все методы класса `QGraphicsItem` и поддерживает следующие методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicslineitem.html**):

- `setLine()` — задает параметры линии. Форматы метода:

```
setLine(<X1>, <Y1>, <X2>, <Y2>)
setLine(<Линия QLineF>)
```

- `line()` — возвращает параметры линии в виде объекта класса `QLineF`;
- `setPen(<Перо QPen>)` — устанавливает перо.

26.3.2.2. Класс *QAbstractGraphicsShapeItem*

Класс `QAbstractGraphicsShapeItem` является базовым классом всех графических фигур, более сложных, чем линия. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem
```

Класс `QAbstractGraphicsShapeItem` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qabstractgraphicsshapeitem.html**):

- `setPen(<Перо QPen>)` — устанавливает перо;
- `setBrush(<Кисть QBrush>)` — устанавливает кисть.

26.3.2.3. Прямоугольник

Класс `QGraphicsRectItem` описывает прямоугольник. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsRectItem
```

Форматы конструктора класса:

```
QGraphicsRectItem([parent=None])
QGraphicsRectItem(<X>, <Y>, <Ширина>, <Высота>[, parent=None])
QGraphicsRectItem(<Прямоугольник QRectF>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsRectItem` наследует все методы базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsrectitem.html**):

- `setRect()` — задает параметры прямоугольника. Форматы метода:

  ```
  setRect(<X>, <Y>, <Ширина>, <Высота>)
  setRect(<Прямоугольник QRectF>)
  ```

- `rect()` — возвращает параметры прямоугольника в виде объекта класса `QRectF`.

26.3.2.4. Многоугольник

Класс `QGraphicsPolygonItem` описывает многоугольник. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsPolygonItem
```

Форматы конструктора класса:

```
QGraphicsPolygonItem([parent=None])
QGraphicsPolygonItem(<Многоугольник QPolygonF>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsPolygonItem` наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicspolygonitem.html**):

- `setPolygon(<Многоугольник QPolygonF>)` — задает параметры многоугольника;
- `polygon()` — возвращает параметры многоугольника в виде объекта класса `QPolygonF`;
- `setFillRule(<Алгоритм>)` — задает алгоритм определения, находится ли какая-либо точка внутри нарисованного прямоугольника или вне его. В качестве значения параметра указывается элемент `OddEvenFill` или `WindingFill` перечисления `FillRule` из модуля `QtCore.Qt`.

26.3.2.5. Эллипс

Класс `QGraphicsEllipseItem` описывает эллипс. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsEllipseItem
```

Форматы конструктора класса:

```
QGraphicsEllipseItem([parent=None])
QGraphicsEllipseItem(<X>, <Y>, <Ширина>, <Высота>[, parent=None])
QGraphicsEllipseItem(<Прямоугольник QRectF>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsEllipseItem` наследует все методы базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsellipseitem.html**):

- `setRect()` — задает параметры прямоугольника, в который необходимо вписать эллипс. Форматы метода:

  ```
  setRect(<X>, <Y>, <Ширина>, <Высота>)
  setRect(<Прямоугольник QRectF>)
  ```

- `rect()` — возвращает параметры прямоугольника, в который вписан эллипс, в виде объекта класса `QRectF`;
- `setStartAngle(<Угол>)` и `setSpanAngle(<Угол>)` — задают начальный и конечный углы сектора соответственно. Углы задаются в значениях $^1/_{16}°$. Полный круг эквивалентен значению 16 × 360 = 5760. Нулевой угол находится в позиции «трех часов». Положительные значения углов отсчитываются против часовой стрелки, а отрицательные — по часовой стрелке;
- `startAngle()` и `spanAngle()` — возвращают значения начального и конечного углов сектора соответственно.

26.3.2.6. Изображение

Класс `QGraphicsPixmapItem` описывает изображение. Иерархия наследования:

```
QGraphicsItem — QGraphicsPixmapItem
```

Форматы конструктора класса:

```
QGraphicsPixmapItem([parent=None])
QGraphicsPixmapItem(<Изображение QPixmap>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsPixmapItem` наследует все методы из класса `QGraphicsItem` и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicspixmapitem.html**):

- `setPixmap(<Изображение QPixmap>)` — задает изображение;
- `pixmap()` — возвращает представляющий изображение объект класса `QPixmap`;
- `setOffset()` — задает местоположение изображения в объекте. Форматы метода:

  ```
  setOffset(<X>, <Y>)
  setOffset(<Координаты QPointF>)
  ```

- `offset()` — возвращает местоположение изображения в виде объекта класса `QPointF`;
- `setShapeMode(<Режим>)` — задает режим определения формы изображения. В качестве параметра могут быть указаны следующие элементы перечисления `ShapeMode` из класса `QGraphicsPixmapItem`:

- MaskShape — используется результат выполнения метода mask() класса QPixmap (значение по умолчанию);
- BoundingRectShape — форма определяется по контуру изображения;
- HeuristicMaskShape — используется результат выполнения метода createHeuristicMask() класса QPixmap;

♦ setTransformationMode(<Режим>) — задает режим сглаживания. В качестве параметра могут быть указаны следующие элементы перечисления TransformationMode из класса QtCore.Qt:

- FastTransformation — сглаживание выключено (по умолчанию);
- SmoothTransformation — сглаживание включено.

26.3.2.7. Простой текст

Класс QGraphicsSimpleTextItem описывает простой текст. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsSimpleTextItem
```

Форматы конструктора класса:

```
QGraphicsSimpleTextItem([parent=None])
QGraphicsSimpleTextItem(<Текст>[, parent=None])
```

В параметре parent можно указать ссылку на родительский объект.

Класс QGraphicsSimpleTextItem наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicssimpletextitem.html**):

♦ setText(<Текст>) — задает текст;

♦ text() — возвращает текст;

♦ setFont(<Шрифт QFont>) — задает шрифт;

♦ font() — возвращает описывающий заданный шрифт объект класса QFont.

26.3.2.8. Форматированный текст

Класс QGraphicsTextItem описывает форматированный текст. Иерархия наследования:

```
(QObject, QGraphicsItem) — QGraphicsObject — QGraphicsTextItem
```

Форматы конструктора класса:

```
QGraphicsTextItem([parent=None])
QGraphicsTextItem(<Текст>[, parent=None])
```

В параметре parent можно указать ссылку на родительский объект.

Класс QGraphicsTextItem наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicstextitem.html**):

♦ setPlainText(<Простой текст>) — задает простой текст;

♦ toPlainText() — возвращает простой текст;

♦ setHtml(<HTML-код>) — задает форматированный текст в виде HTML-кода;

♦ toHtml() — возвращает форматированный текст в виде HTML-кода;

- `setFont(<Шрифт QFont>)` — устанавливает шрифт;
- `font()` — возвращает описывающий установленный шрифт объект класса `QFont`;
- `setDefaultTextColor(<Цвет QColor>)` — задает цвет текста по умолчанию;
- `defaultTextColor()` — возвращает цвет текста по умолчанию в виде объекта класса `QColor`;
- `setTextWidth(<Ширина>)` — задает предпочитаемую ширину строки. Если текст не помещается в установленную ширину, он будет перенесен на новую строку;
- `textWidth()` — возвращает предпочитаемую ширину текста;
- `setDocument(<Документ>)` — задает документ в виде объекта класса `QTextDocument` (см. *разд. 22.6.4*);
- `document()` — возвращает ссылку на документ (объект класса `QTextDocument`; см. *разд. 22.6.4*);
- `setTextCursor(<Курсор>)` — устанавливает курсор в виде объекта класса `QTextCursor` (см. *разд. 22.6.5*);
- `textCursor()` — возвращает курсор (объект класса `QtextCursor` — см. *разд. 22.6.5*);
- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом. По умолчанию используется режим `NoTextInteraction`, при котором пользователь не может взаимодействовать с текстом. Допустимые режимы приведены в *разд. 22.1* (см. описание метода `setTextInteractionFlags()`);
- `setTabChangesFocus(<Флаг>)` — если в качестве параметра указано значение `False`, то нажатием клавиши <Tab> можно будет вставить в текст символ табуляции. Если указано значение `True`, клавиша <Tab> станет использоваться для передачи фокуса;
- `setOpenExternalLinks(<Флаг>)` — если в качестве параметра указано значение `True`, щелчок на гиперссылке приведет к открытию браузера, используемого в системе по умолчанию, и загрузке страницы с указанным в гиперссылке интернет-адресом (URL). Метод работает только при использовании режима `TextBrowserInteraction`.

Класс `QGraphicsTextItem` поддерживает следующие сигналы:

- `linkActivated(<Интернет-адрес>)` — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен интернет-адрес гиперссылки в виде строки;
- `linkHovered(<Интернет-адрес>)` — генерируется при наведении указателя мыши на гиперссылку. Через параметр внутри обработчика доступен интернет-адрес гиперссылки в виде строки.

26.4. Группировка объектов

Объединить несколько объектов в группу позволяет класс `QGraphicsItemGroup`. Над сгруппированными объектами можно выполнять различные преобразования (например, перемещать или поворачивать их одновременно). Иерархия наследования для класса `QGraphicsItemGroup` выглядит так:

```
QGraphicsItem — QGraphicsItemGroup
```

Формат конструктора класса:

```
QGraphicsItemGroup([parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс QGraphicsItemGroup наследует все методы класса QGraphicsItem и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsitemgroup.html**):

- addToGroup(<Объект>) — добавляет указанный графический объект в текущую группу;
- removeFromGroup(<Объект>) — удаляет указанный графический объект из текущей группы.

Создать группу и добавить ее на сцену можно и с помощью метода createItemGroup(<Список с графическими объектами>) класса QGraphicsScene. Метод возвращает ссылку на группу (объект класса QGraphicsItemGroup). Удалить группу со сцены позволяет метод destroyItemGroup(<Группа>) класса QGraphicsScene.

Добавить объект в группу позволяет также метод setGroup(<Группа>) класса QGraphicsItem. Получить ссылку на группу, в которой находится объект, можно вызовом метода group() класса QGraphicsItem. Если объект не находится ни в какой группе, метод возвращает None.

26.5. Эффекты

К графическим объектам допускается применить различные эффекты: изменение прозрачности, цвета, отображение тени, размытие и пр. Наследуя класс QGraphicsEffect и переопределяя в нем метод draw(), можно создать свой эффект.

Для установки эффекта и получения ссылки на него предназначены следующие методы класса QGraphicsItem:

- setGraphicsEffect(<Эффект>) — задает эффект в виде объекта класса, производного от QGraphicsEffect;
- graphicsEffect() — возвращает ссылку на эффект (объект класса, производного от QGraphicsEffect) или значение None, если эффект не был установлен.

26.5.1. Класс *QGraphicsEffect*

Класс QGraphicsEffect является базовым классом для всех эффектов. Иерархия наследования выглядит так:

```
QObject — QGraphicsEffect
```

Формат конструктора класса:

```
QGraphicsEffect([parent=None])
```

Класс QGraphicsEffect поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicseffect.html**):

- draw(self, <Устройство рисования>) — собственно рисует эффект. Этот абстрактный метод должен быть переопределен в производных классах;
- setEnabled(<Флаг>) — если в качестве параметра указано значение False, эффект отключается. Значение True разрешает использование эффекта. Метод является слотом;
- isEnabled() — возвращает значение True, если эффект включен, и False — в противном случае;
- update() — вызывает перерисовку эффекта. Метод является слотом.

Класс QGraphicsEffect поддерживает сигнал enabledChanged(<Флаг>), который генерируется при изменении статуса эффекта. Внутри обработчика через параметр доступно значение True, если эффект включен, и False — в противном случае.

26.5.2. Тень

Класс QGraphicsDropShadowEffect реализует вывод тени у объекта. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsDropShadowEffect
```

Формат конструктора класса:

```
QGraphicsDropShadowEffect([parent=None])
```

Класс QGraphicsDropShadowEffect наследует все методы базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsdropshadoweffect.html**):

- setColor(<Цвет QColor>) — задает цвет тени. По умолчанию используется полупрозрачный темно-серый цвет (QColor(63, 63, 63, 180)). Метод является слотом;
- color() — возвращает цвет тени (объект класса QColor);
- setBlurRadius(<Радиус>) — задает радиус размытия тени в виде вещественного числа. По умолчанию используется значение 1. Метод является слотом;
- blurRadius() — возвращает радиус размытия тени;
- setOffset() — задает смещение тени. По умолчанию тень смещена на 8 пикселов вниз и вправо. Форматы метода:

  ```
  setOffset(<По оси X>, <По оси Y>)
  setOffset(<Смещение по обеим осям>)
  setOffset(<Смещение QPointF>)
  ```

  Второй конструктор задает смещение сразу по обеим осям координат. В первом и втором конструкторах параметры задаются вещественными числами. Метод является слотом;
- offset() — возвращает смещение тени в виде объекта класса QPointF;
- setXOffset(<Смещение>) — задает смещение по оси X в виде вещественного числа. Метод является слотом;
- xOffset() — возвращает смещение по оси X;
- setYOffset(<Смещение>) — задает смещение по оси Y в виде вещественного числа. Метод является слотом;
- yOffset() — возвращает смещение по оси Y.

Класс QGraphicsDropShadowEffect поддерживает сигналы:

- colorChanged(<Цвет QColor>) — генерируется при изменении цвета тени. Внутри обработчика через параметр доступен новый цвет;
- blurRadiusChanged(<Радиус размытия>) — генерируется при изменении радиуса размытия. Внутри обработчика через параметр доступно новое значение в виде вещественного числа;
- offsetChanged(<Смещение QPointF>) — генерируется при изменении смещения. Внутри обработчика через параметр доступно новое смещение.

26.5.3. Размытие

Класс `QGraphicsBlurEffect` реализует эффект размытия. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsBlurEffect
```

Формат конструктора класса:

```
QGraphicsBlurEffect([parent=None])
```

Класс `QGraphicsBlurEffect` наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsblureffect.html**):

- `setBlurRadius(<Радиус>)` — задает радиус размытия в виде вещественного числа. По умолчанию используется значение 5. Метод является слотом;
- `blurRadius()` — возвращает радиус размытия.

Класс `QGraphicsBlurEffect` поддерживает сигнал `blurRadiusChanged(<Радиус>)`, который генерируется при изменении радиуса размытия. Внутри обработчика через параметр доступно новое значение радиуса в виде вещественного числа.

26.5.4. Изменение цвета

Класс `QGraphicsColorizeEffect` реализует эффект изменения цвета. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsColorizeEffect
```

Формат конструктора класса:

```
QGraphicsColorizeEffect([parent=None])
```

Класс `QGraphicsColorizeEffect` наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicscolorizeeffect.html**):

- `setColor(<Цвет QColor>)` — задает цвет. По умолчанию используется светло-синий цвет (`QColor(0, 0, 192)`). Метод является слотом;
- `color()` — возвращает текущий цвет в виде объекта класса `QColor`;
- `setStrength(<Значение>)` — задает интенсивность цвета. В качестве значения указывается вещественное число от `0.0` до `1.0` (значение по умолчанию). Метод является слотом;
- `strength()` — возвращает интенсивность цвета.

Класс `QGraphicsColorizeEffect` поддерживает сигналы:

- `colorChanged(<Цвет QColor>)` — генерируется при изменении цвета. Внутри обработчика через параметр доступен новый цвет;
- `strengthChanged(<Интенсивность>)` — генерируется при изменении интенсивности цвета. Внутри обработчика через параметр доступно новое значение в виде вещественного числа.

26.5.5. Изменение прозрачности

Класс `QGraphicsOpacityEffect` реализует эффект прозрачности. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsOpacityEffect
```

Формат конструктора класса:

```
QGraphicsOpacityEffect([parent=None])
```

Класс `QGraphicsOpacityEffect` наследует все методы базовых классов и поддерживает следующие методы (здесь приведены только самые полезные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsopacityeffect.html**):

- `setOpacity(<Значение>)` — задает степень прозрачности. В качестве значения указывается вещественное число от `0.0` до `1.0`. По умолчанию используется значение `0.7`. Метод является слотом;
- `opacity()` — возвращает степень прозрачности;
- `setOpacityMask(<Кисть QBrush>)` — задает маску прозрачности. Метод является слотом;
- `opacityMask()` — возвращает маску прозрачности в виде объекта класса `QBrush`.

Класс `QGraphicsOpacityEffect` поддерживает следующие сигналы:

- `opacityChanged(<Прозрачность>)` — генерируется при изменении степени прозрачности. Внутри обработчика через параметр доступно новое значение в виде вещественного числа;
- `opacityMaskChanged(<Кисть QBrush>)` — генерируется при изменении маски прозрачности. Через параметр доступна новая маска.

26.6. Обработка событий

Все происходящие в графических объектах события изначально возникают в графическом представлении. Компонент-представление преобразует их и передает объекту графической сцены, который, в свою очередь, перенаправляет их объекту, способному обработать возникшее событие (так, щелчок мыши передается объекту, который расположен по координатам щелчка).

Обработка событий в представлении ничем не отличается от обычной обработки событий, рассмотренной в *главе 20*, за исключением некоторых особенностей, рассматриваемых в этом разделе.

26.6.1. События клавиатуры

При обработке событий клавиатуры следует учитывать, что:

- графический объект должен иметь возможность принимать фокус ввода. Для этого необходимо установить флаг `ItemIsFocusable` — например, с помощью метода `setFlag()` класса `QGraphicsItem`;
- объект должен быть в фокусе ввода. Методы, позволяющие управлять фокусом ввода, мы рассматривали в *разд. 26.1.5* и *26.3.1.1*;
- чтобы захватить эксклюзивный ввод с клавиатуры, следует вызвать у графического объекта метод `grabKeyboard()`, а чтобы освободить ввод — метод `ungrabKeyboard()`;
- можно перехватить нажатие любых клавиш, кроме <Tab> и <Shift>+<Tab>, используемых для передачи фокуса;
- если событие обработано, нужно вызвать метод `accept()` у объекта события, в противном случае — метод `ignore()`.

Для обработки событий клавиатуры следует наследовать класс, реализующий графический объект, и переопределить в нем методы:

- `focusInEvent(self, <event>)` — вызывается при получении фокуса ввода. Через параметр `<event>` доступен объект класса `QFocusEvent` (см. *разд. 20.8.1*);
- `focusOutEvent(self, <event>)` — вызывается при потере фокуса ввода. Через параметр `<event>` доступен объект класса `QFocusEvent` (см. *разд. 20.8.1*);
- `keyPressEvent(self, <event>)` — вызывается при нажатии клавиши. Если клавишу удерживать нажатой, метод будет вызываться постоянно, пока ее не отпустят. Через параметр `<event>` доступен объект класса `QKeyEvent` (см. *разд. 20.8.3*);
- `keyReleaseEvent(self, <event>)` — вызывается при отпускании нажатой ранее клавиши. Через параметр `<event>` доступен объект класса `QKeyEvent` (см. *разд. 20.8.3*).

С помощью метода `setFocusProxy(<Графический объект>)` класса `QGraphicsItem` можно указать графический объект, который будет обрабатывать события клавиатуры вместо текущего объекта. Получить ссылку на назначенный ранее объект-обработчик событий клавиатуры позволяет метод `focusProxy()`.

26.6.2. События мыши

Для обработки нажатия кнопок мыши и перемещения мыши следует наследовать класс, реализующий графический объект, и переопределить в нем такие методы:

- `mousePressEvent(self, <event>)` — вызывается при нажатии кнопки мыши над объектом. Через параметр `<event>` доступен объект класса `QGraphicsSceneMouseEvent`. Если событие принято, необходимо вызвать метод `accept()` объекта события, в противном случае — метод `ignore()`. Если был вызван метод `ignore()`, методы `mouseReleaseEvent()` и `mouseMoveEvent()` вызваны не будут.

  С помощью метода `setAcceptedMouseButtons(<Кнопки>)` класса `QGraphicsItem` можно указать кнопки, события от которых объект будет принимать. По умолчанию объект принимает события от всех кнопок мыши. В качестве параметра указывается элемент перечисления `MouseButton` из модуля `QtCore.Qt` или их комбинация через оператор `|`. Если указать элемент `NoButton`, объект вообще не станет принимать события от кнопок мыши;
- `mouseReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой кнопки мыши. Через параметр `<event>` доступен объект класса `QGraphicsSceneMouseEvent`;
- `mouseDoubleClickEvent(self, <event>)` — вызывается при двойном щелчке мышью в области объекта. Через параметр `<event>` доступен объект класса `QgraphicsSceneMouseEvent`;
- `mouseMoveEvent(self, <event>)` — вызывается при перемещении мыши. Через параметр `<event>` доступен объект класса `QGraphicsSceneMouseEvent`.

Класс `QGraphicsSceneMouseEvent` наследует все методы классов `QGraphicsSceneEvent` и `QEvent` и добавляет поддержку следующих методов:

- `pos()` — возвращает объект класса `QPointF` с координатами курсора мыши в пределах области объекта;
- `scenePos()` — возвращает объект класса `QPointF` с координатами курсора мыши в пределах сцены;

- `screenPos()` — возвращает объект класса `QPoint` с координатами курсора мыши в пределах экрана;
- `lastPos()` — возвращает объект класса `QPointF` с координатами последней запомненной представлением позиции мыши в пределах области объекта;
- `lastScenePos()` — возвращает объект класса `QPointF` с координатами последней запомненной представлением позиции мыши в пределах сцены;
- `lastScreenPos()` — возвращает объект класса `QPoint` с координатами последней запомненной представлением позиции мыши в пределах экрана;
- `buttonDownPos(<Кнопка>)` — возвращает объект класса `QPointF` с координатами щелчка указанной кнопки мыши в пределах области объекта;
- `buttonDownScenePos(<Кнопка>)` — возвращает объект класса `QPointF` с координатами щелчка указанной кнопки мыши в пределах сцены;
- `buttonDownScreenPos(<Кнопка>)` — возвращает объект класса `QPoint` с координатами щелчка указанной кнопки мыши в пределах экрана;
- `button()` — возвращает обозначение кнопки мыши, нажатие которой вызвало событие;
- `buttons()` — возвращает комбинацию обозначений всех кнопок мыши, одновременное нажатие которых вызвало событие.

  Обозначения кнопок представляются элементами перечисления `MouseButton` из модуля `QtCore.Qt` или их комбинациями через оператор `|`;
- `modifiers()` — возвращает набор обозначений всех клавиш-модификаторов (<Shift>, <Ctrl>, <Alt> и др.), что были нажаты вместе с кнопкой мыши, в виде комбинации элементов перечисления `KeyboardModifier` из модуля `QtCore.Qt` или их комбинации через оператор `|`.

По умолчанию событие мыши перехватывает объект, на котором был произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне объекта, следует захватить мышь с помощью метода `grabMouse()` класса `QGraphicsItem`. Освободить захваченную ранее мышь позволяет метод `ungrabMouse()`. Получить ссылку на объект, захвативший мышь, можно с помощью метода `mouseGrabberItem()` класса `QGraphicsScene`.

Для обработки прочих событий мыши нужно наследовать класс, реализующий графический объект, и переопределить следующие методы:

- `hoverEnterEvent(self, <event>)` — вызывается при наведении курсора мыши на область объекта. Через параметр `<event>` доступен объект класса `QGraphicsSceneHoverEvent`;
- `hoverLeaveEvent(self, <event>)` — вызывается, когда курсор мыши покидает область объекта. Через параметр `<event>` доступен объект класса `QGraphicsSceneHoverEvent`;
- `hoverMoveEvent(self, <event>)` — вызывается при перемещении курсора мыши внутри области объекта. Через параметр `<event>` доступен объект класса `QGraphicsSceneHoverEvent`;
- `wheelEvent(self, <event>)` — вызывается при повороте колесика мыши при нахождении курсора мыши над объектом. Чтобы обрабатывать событие, в любом случае следует захватить мышь. Через параметр `<event>` доступен объект класса `QGraphicsSceneWheelEvent`.

Следует учитывать, что методы `hoverEnterEvent()`, `hoverLeaveEvent()` и `hoverMoveEvent()` будут вызваны только в том случае, если обработка этих событий разрешена. Чтобы раз-

решить обработку событий перемещения мыши, следует вызвать метод `setAcceptHoverEvents(<Флаг>)` класса `QGraphicsItem` и передать ему значение `True`. Значение `False` запрещает обработку событий перемещения указателя. Получить текущее состояние позволяет метод `acceptHoverEvents()`.

Класс `QGraphicsSceneHoverEvent` наследует все методы классов `QGraphicsSceneEvent` и `QEvent` и добавляет поддержку своих методов:

- `pos()` — возвращает объект класса `QPointF` с координатами курсора мыши в пределах области объекта;
- `scenePos()` — возвращает объект класса `QPointF` с координатами курсора мыши в пределах сцены;
- `screenPos()` — возвращает объект класса `QPoint` с координатами курсора мыши в пределах экрана;
- `lastPos()` — возвращает объект класса `QPointF` с координатами последней запомненной представлением позиции мыши в пределах области объекта;
- `lastScenePos()` — возвращает объект класса `QPointF` с координатами последней запомненной представлением позиции мыши в пределах сцены;
- `lastScreenPos()` — возвращает объект класса `QPoint` с координатами последней запомненной представлением позиции мыши в пределах экрана;
- `modifiers()` — возвращает комбинацию обозначений всех клавиш-модификаторов (<Shift>, <Ctrl>, <Alt> и др.), что были нажаты одновременно с перемещением мыши, в виде комбинации элементов перечисления `KeyboardModifiers` из модуля `QtCore.Qt` или их комбинации через оператор `|`.

Класс `QGraphicsSceneWheelEvent` наследует все методы классов `QGraphicsSceneEvent` и `QEvent` и добавляет поддержку следующих методов:

- `delta()` — возвращает угол поворота колесика, измеряемого в $^1/_8°$. Положительное значение означает, что колесико поворачивалось в направлении от пользователя, отрицательное — к пользователю;
- `orientation()` — возвращает направление вращения колесика в виде значения одного из следующих элементом перечисления `Orientation` из модуля `QtCore.Qt`: `Horizontal` (по горизонтали) или `Vertical` (по вертикали);
- `pos()` — возвращает объект класса `QPointF` с координатами курсора мыши в пределах области объекта;
- `scenePos()` — возвращает объект класса `QPointF` с координатами курсора мыши в пределах сцены;
- `screenPos()` — возвращает объект класса `QPoint` с координатами курсора мыши в пределах экрана;
- `buttons()` — возвращает комбинацию обозначений всех кнопок мыши, нажатых одновременно с вращением колесика;
- `modifiers()` — возвращает комбинацию обозначений всех клавиш-модификаторов (<Shift>, <Ctrl>, <Alt> и др.), что были нажаты одновременно с вращением колесика.

26.6.3. Обработка перетаскивания и сброса

Прежде чем обрабатывать перетаскивание и сброс, необходимо сообщить системе, что графический объект может их обработать. Для этого следует вызвать метод `setAcceptDrops(<Флаг>)` класса `QGraphicsItem` и передать ему значение `True`.

Обработка перетаскивания и сброса в графическом объекте выполняется следующим образом:

- внутри метода `dragEnterEvent()` проверяется MIME-тип перетаскиваемых данных и действие. Если графический объект способен обработать сброс этих данных и соглашается с предложенным действием, необходимо вызвать метод `acceptProposedAction()` объекта события. Если нужно изменить действие, методу `setDropAction()` объекта события передается новое действие, а затем у того же объекта вызывается метод `accept()` вместо метода `acceptProposedAction()`;
- если необходимо ограничить область сброса некоторым участком графического объекта, можно дополнительно определить в нем метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри области графического объекта. При согласии со сбрасыванием следует вызвать у объекта события метод `accept()`;
- внутри метода `dropEvent()` производится обработка сброса.

Обработать события, возникающие при перетаскивании и сбрасывании объектов, позволяют следующие методы:

- `dragEnterEvent(self, <event>)` — вызывается, когда перетаскиваемый объект входит в область графического объекта. Через параметр `<event>` доступен объект класса `QGraphicsSceneDragDropEvent`;
- `dragLeaveEvent(self, <event>)` — вызывается, когда перетаскиваемый объект покидает область графического объекта. Через параметр `<event>` доступен объект класса `QGraphicsSceneDragDropEvent`;
- `dragMoveEvent(self, <event>)` — вызывается при перетаскивании объекта внутри области графического объекта. Через параметр `<event>` доступен объект класса `QGraphicsSceneDragDropEvent`;
- `dropEvent(self, <event>)` — вызывается при сбрасывании объекта в области графического объекта. Через параметр `<event>` доступен объект класса `QGraphicsSceneDragDropEvent`.

Класс `QGraphicsSceneDragDropEvent` наследует все методы классов `QGraphicsSceneEvent` и `QEvent` и добавляет поддержку методов:

- `mimeData()` — возвращает объект класса `QMimeData` с перемещаемыми данными и информацией о MIME-типе;
- `pos()` — возвращает объект класса `QPointF` с координатами курсора мыши в пределах области объекта;
- `scenePos()` — возвращает объект класса `QPointF` с координатами курсора мыши в пределах сцены;
- `screenPos()` — возвращает объект класса `QPoint` с координатами курсора мыши в пределах экрана;
- `possibleActions()` — возвращает набор возможных действий при сбрасывании в виде комбинации элементов перечисления `DropAction` из модуля `QtCore.Qt`;

- proposedAction() — возвращает действие по умолчанию при сбрасывании в виде элемента перечисления DropAction из модуля QtCore.Qt;
- acceptProposedAction() — подтверждает готовность принять перемещаемые данные и согласие с действием по умолчанию, возвращаемым методом proposedAction();
- setDropAction(<Действие>) — указывает другое действие при сбрасывании. После изменения действия следует вызвать метод accept(), а не acceptProposedAction();
- dropAction() — возвращает действие, которое должно быть выполнено при сбрасывании;
- buttons() — возвращает комбинацию обозначений всех кнопок мыши, нажатых в процессе перетаскивания;
- modifiers() — возвращает комбинацию обозначений всех клавиш-модификаторов (<Shift>, <Ctrl>, <Alt> и др.), что были нажаты в процессе перетаскивания;
- source() — возвращает ссылку на источник события или значение None.

26.6.4. Фильтрация событий

События можно перехватывать еще до того, как они будут переданы специализированному методу. Для этого в классе графического объекта необходимо переопределить метод sceneEvent(self, <event>). Через параметр <event> здесь будет доступен объект с дополнительной информацией о событии. Тип этого объекта различен для разных типов событий. Внутри метода следует вернуть значение True, если событие обработано, и False — в противном случае. Если вернуть значение True, специализированный метод (например, mousePressEvent()) вызываться не будет.

Чтобы произвести фильтрацию событий какого-либо объекта, в классе графического объекта необходимо переопределить метод sceneEventFilter(self, <Объект>, <event>). Через параметр <Объект> здесь будет доступна ссылка на графический объект, в котором возникло событие, а через параметр <event> — объект с информацией о самом событии. Тип этого объекта различен для разных типов событий. Внутри метода следует вернуть значение True, если событие обработано, и False — в противном случае. Если вернуть значение True, объект, в котором возникло событие, не получит его.

Указать, события какого объекта фильтруются, позволяют следующие методы класса QGraphicsItem:

- installSceneEventFilter(<Графический объект>) — задает объект, который будет производить фильтрацию событий текущего объекта;
- removeSceneEventFilter(<Графический объект>) — удаляет объект-фильтр событий;
- setFiltersChildEvents(<Флаг>) — если в качестве параметра указано значение True, текущий объект будет производить фильтрацию событий всех своих дочерних объектов.

26.6.5. Обработка изменения состояния объекта

Чтобы обработать изменение состояния объекта, следует переопределить метод itemChange(self, <Состояние>, <Значение>) в классе графического объекта. Метод должен возвращать новое значение. Через параметр <Состояние> доступно состояние, которое было изменено, в виде значения одного из следующих элементов перечисления GraphicItemChange из класса QGraphicsItem (здесь приведены только основные — полный

их список можно найти на странице **https://doc.qt.io/qt-6/qgraphicsitem.html#GraphicsItemChange-enum**):

- `ItemEnabledChange` — изменилось состояние доступности;
- `ItemEnabledHasChanged` — изменилось состояние доступности. Возвращаемое значение игнорируется;
- `ItemPositionChange` — изменилось местоположение объекта. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`;
- `ItemPositionHasChanged` — изменилось местоположение объекта. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`. Возвращаемое значение игнорируется;
- `ItemScenePositionHasChanged` — изменилось местоположение объекта на сцене с учетом преобразований, примененных к самому объекту и его родителям. Метод будет вызван, только если установлен флаг `ItemSendsScenePositionChanges`. Возвращаемое значение игнорируется;
- `ItemTransformChange` — изменилась матрица преобразований. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`;
- `ItemTransformHasChanged` — изменилась матрица преобразований. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`. Возвращаемое значение игнорируется;
- `ItemSelectedChange` — изменилось выделение объекта;
- `ItemSelectedHasChanged` — изменилось выделение объекта. Возвращаемое значение игнорируется;
- `ItemVisibleChange` — изменилось состояние видимости объекта;
- `ItemVisibleHasChanged` — изменилось состояние видимости объекта. Возвращаемое значение игнорируется;
- `ItemCursorChange` — изменился курсор;
- `ItemCursorHasChanged` — изменился курсор. Возвращаемое значение игнорируется;
- `ItemToolTipChange` — изменилась всплывающая подсказка;
- `ItemToolTipHasChanged` — изменилась всплывающая подсказка. Возвращаемое значение игнорируется;
- `ItemFlagsChange` — изменились флаги;
- `ItemFlagsHaveChanged` — изменились флаги. Возвращаемое значение игнорируется;
- `ItemZValueChange` — изменилось положение по оси `Z`;
- `ItemZValueHasChanged` — изменилось положение по оси `Z`. Возвращаемое значение игнорируется;
- `ItemOpacityChange` — изменилась прозрачность объекта;
- `ItemOpacityHasChanged` — изменилась прозрачность объекта. Возвращаемое значение игнорируется;
- `ItemParentChange` — изменился родитель объекта;
- `ItemParentHasChanged` — изменился родитель объекта. Возвращаемое значение игнорируется;

- `ItemChildAddedChange` — в объект был добавлен новый потомок;
- `ItemChildRemovedChange` — из объекта был удален потомок;
- `ItemSceneChange` — объект был перемещен на другую сцену;
- `ItemSceneHasChanged` — объект был перемещен на другую сцену. Возвращаемое значение игнорируется.

Внимание!

Вызов некоторых методов из метода `itemChange()` может привести к рекурсии. За подробной информацией обращайтесь к документации по классу `QGraphicsItem`.

Примечание

PyQt также поддерживает помещение на сцену видеозаписей в качестве отдельных графических объектов и создание анимации. За подробным описанием обращайтесь к документации по этой библиотеке.

ГЛАВА 27

Диалоговые окна

PyQt позволяет выводить как произвольные диалоговые окна, так и стандартные: открытия файла, каталога, для ввода данных, для вывода сообщений и др.

Все рассмотренные в этой главе классы определены в модуле `QtWidgets`, если не указано иное.

27.1. Пользовательские диалоговые окна

Класс `QDialog` реализует диалоговое окно. По умолчанию окно выводится с рамкой и заголовком, в котором расположены кнопки вывода справки и закрытия. Размеры окна можно изменять. Иерархия наследования класса `QDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog
```

Конструктор класса `QDialog` имеет следующий формат:

```
QDialog([parent=<Родитель>][, flags=0])
```

В параметре `parent` указывается ссылка на родительское окно. Если родитель не указан или имеет значение `None`, диалоговое окно будет центрироваться относительно экрана, если указана — относительно родительского окна (это также позволяет создать модальное диалоговое окно, которое будет блокировать только окно родителя, а не все окна программы). Значения, указываемые в параметре `flags`, были рассмотрены в *разд. 19.1.1*. Тип окна по умолчанию — `Dialog`.

Класс `QDialog` наследует все методы базовых классов и дополнительно реализует следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qdialog.html**):

- `exec()` — отображает модальное диалоговое окно, дожидается закрытия окна и возвращает код возврата в виде значения следующих элементов перечисления `DialogCode` из класса `QDialog`:
  - `Accepted` — нажата кнопка **OK**;
  - `Rejected` — нажата кнопка **Cancel**, кнопка закрытия в заголовке окна или клавиша <Esc>.

  Метод является слотом.

Пример отображения диалогового окна и обработки статуса внутри обработчика нажатия кнопки (класс MyDialog является наследником класса QDialog, а window — ссылка на главное окно):

```
def on_clicked():
    dialog = MyDialog(window)
    result = dialog.exec()
    if result == QtWidgets.QDialog.DialogCode.Accepted:
        print("Нажата кнопка OK")
        # Здесь получаем данные из диалогового окна
    else:
        print("Нажата кнопка Cancel")
```

- accept() — закрывает модальное диалоговое окно и устанавливает код возврата равным значению элемента Accepted перечисления DialogCode. Метод является слотом. Обычно его соединяют с сигналом нажатия кнопки **OK**:

```
self.btnOK.clicked.connect(self.accept)
```

- reject() — закрывает модальное диалоговое окно и устанавливает код возврата равным значению элемента Rejected перечисления DialogCode. Метод является слотом. Обычно его соединяют с сигналом нажатия кнопки **Cancel**:

```
self.btnCancel.clicked.connect(self.reject)
```

- done(<Код возврата>) — закрывает модальное диалоговое окно и устанавливает код возврата равным значению параметра. Метод является слотом;
- setResult(<Код возврата>) — устанавливает код возврата;
- result() — возвращает код возврата;
- setSizeGripEnabled(<Флаг>) — если в качестве параметра указано значение True, то в правом нижнем углу диалогового окна будет отображен значок изменения размера, а если False — то скрыт (значение по умолчанию);
- isSizeGripEnabled() — возвращает значение True, если в правом нижнем углу диалогового окна отображается значок изменения размера, и False — в противном случае;
- setVisible(<Флаг>) — если в параметре указано значение True, диалоговое окно будет отображено, а если значение False — то скрыто;
- show() — отображает диалоговое окно;
- hide() — закрывает диалоговое окно;
- open() — отображает диалоговое окно в модальном режиме, но не дожидается его закрытия. Блокируется только родительское окно, а не все окна приложения. Метод является слотом;
- setModal(<Флаг>) — если в качестве параметра указано значение True, окно будет модальным, а если False — обычным. Обратите внимание, что окно, открываемое с помощью метода exec(), всегда будет модальным — независимо от значения, заданного вызовом метода setModal(). Чтобы диалоговое окно было не модальным, нужно отображать его с помощью метода show() или setVisible(). После вызова этих методов следует вызвать методы raise_() (чтобы поместить окно поверх всех окон) и activateWindow() (чтобы сделать окно *активным* — т. е. имеющим фокус ввода).

Указать, что окно является модальным, также позволяет метод `setWindowModality(<Флаг>)` класса `QWidget`. В качестве параметра могут быть указаны следующие элементы перечисления `WindowModality` из модуля `QtCore.Qt`:

- `NonModal` — окно не является модальным;
- `WindowModal` — окно блокирует только родительские окна в пределах иерархии;
- `ApplicationModal` — окно блокирует все окна в программе.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

Получить текущее значение модальности окна позволяет метод `windowModality()` класса `QWidget`. Проверить, является ли окно модальным, можно с помощью метода `isModal()` того же класса, который возвращает `True`, если окно является модальным, и `False` — в противном случае.

Класс `QDialog` поддерживает следующие сигналы:

- `accepted()` — генерируется при установке флага `Accepted` (нажата кнопка **OK**). Не генерируется при закрытии окна с помощью метода `hide()` или `setVisible()`;
- `rejected()` — генерируется при установке флага `Rejected` (нажата кнопка **Cancel**, кнопка закрытия в заголовке окна или клавиша <Esc>). Не генерируется при закрытии окна с помощью метода `hide()` или `setVisible()`;
- `finished(<Код завершения>)` — генерируется при установке любого кода завершения в результате действий пользователя или программно — вызовом методов `accept()`, `reject()` или `done()`. Внутри обработчика через параметр доступен код завершения. Сигнал не генерируется при закрытии окна с помощью метода `hide()` или `setVisible()`.

Для всех кнопок, добавляемых в диалоговое окно, автоматически вызывается метод `setAutoDefault()` со значением `True` в качестве параметра. В этом случае кнопка может быть нажата с помощью клавиши <Enter> при условии, что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>.

С помощью метода `setDefault()` можно указать кнопку по умолчанию. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент — например, на текстовое поле.

27.2. Класс *QDialogButtonBox*

Класс `QDialogButtonBox` представляет контейнер, в который можно добавить различные кнопки: как стандартные, так и пользовательские. Внешний вид контейнера и расположение кнопок в нем зависят от используемой операционной системы. Иерархия наследования для класса `QDialogButtonBox`:

```
(QObject, QPaintDevice) — QWidget — QDialogButtonBox
```

Форматы конструктора класса `QDialogButtonBox`:

```
QDialogButtonBox([parent=None])
QDialogButtonBox(<Ориентация>[, parent=None])
QDialogButtonBox(<Стандартные кнопки>[, parent=None])
QDialogButtonBox(<Стандартные кнопки>, <Ориентация>[, parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент. Параметр `<Ориентация>` задает порядок расположения кнопок внутри контейнера. В качестве значения указываются элементы `Horizontal` (по горизонтали — значение по умолчанию) или `Vertical` (по вертикали) перечисления `Orientation` из модуля `QtCore.Qt`. В параметре `<Стандартные кнопки>` указываются следующие элементы перечисления `StandardButton` (или их комбинация через оператор `|`) класса `QDialogButtonBox`:

- `NoButton` — кнопки не установлены;
- `Ok` — кнопка **OK** с ролью `AcceptRole`;
- `Cancel` — кнопка **Cancel** с ролью `RejectRole`;
- `Yes` — кнопка **Yes** с ролью `YesRole`;
- `YesToAll` — кнопка **Yes to All** с ролью `YesRole`;
- `No` — кнопка **No** с ролью `NoRole`;
- `NoToAll` — кнопка **No to All** с ролью `NoRole`;
- `Open` — кнопка **Open** с ролью `AcceptRole`;
- `Close` — кнопка **Close** с ролью `RejectRole`;
- `Save` — кнопка **Save** с ролью `AcceptRole`;
- `SaveAll` — кнопка **Save All** с ролью `AcceptRole`;
- `Discard` — кнопка **Discard** или **Don't Save** (надпись на кнопке зависит от операционной системы) с ролью `DestructiveRole`;
- `Apply` — кнопка **Apply** с ролью `ApplyRole`;
- `Reset` — кнопка **Reset** с ролью `ResetRole`;
- `RestoreDefaults` — кнопка **Restore Defaults** с ролью `ResetRole`;
- `Help` — кнопка **Help** с ролью `HelpRole`;
- `Abort` — кнопка **Abort** с ролью `RejectRole`;
- `Retry` — кнопка **Retry** с ролью `AcceptRole`;
- `Ignore` — кнопка **Ignore** с ролью `AcceptRole`.

Класс `QDialogButtonBox` наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только интересующие нас — полный их список можно найти на странице **https://doc.qt.io/qt-6/qdialogbuttonbox.html**):

- `setOrientation(<Ориентация>)` — задает порядок расположения кнопок внутри контейнера. В качестве значения указываются элементы `Horizontal` (по горизонтали) или `Vertical` (по вертикали) перечисления `Orientation` из модуля `QtCore.Qt`;
- `setStandardButtons(<Стандартные кнопки>)` — добавляет в контейнер стандартные кнопки:

```
self.box.setStandardButtons(QtWidgets.QDialogButtonBox.StandardButton.Ok |
                            QtWidgets.QDialogButtonBox.StandardButton.Cancel)
```

- `addButton()` — добавляет кнопку в контейнер. Форматы метода:

```
addButton(<Стандартная кнопка StandardButton>)
addButton(<Текст>, <Роль>)
addButton(<Кнопка QAbstractButton>, <Роль>)
```

Первый формат добавляет стандартную кнопку или кнопки (в виде одного из рассмотренных ранее элементов перечисления `StandardButton` или их комбинации через оператор `|`). Второй формат принимает в качестве первого параметра надпись для добавляемой кнопки, а в качестве второго — ее роль. Третий формат принимает добавляемую кнопку в виде объекта одного из подклассов класса `QAbstractButton` — как правило, класса `QPushButton`, представляющего обычную кнопку.

В качестве роли указывается один из следующих элементов перечисления `ButtonRole` из класса `QDialogButtonBox`:

- `InvalidRole` — ошибочная роль;
- `AcceptRole` — нажатие кнопки устанавливает код возврата равным значению элемента `Accepted`;
- `RejectRole` — нажатие кнопки устанавливает код возврата равным значению элемента `Rejected`;
- `DestructiveRole` — кнопка для отказа от изменений и закрытия диалогового окна;
- `ActionRole` — нажатие кнопки приводит к выполнению операции, которая не связана с закрытием окна;
- `HelpRole` — кнопка для отображения справки;
- `YesRole` — кнопка **Yes**;
- `NoRole` — кнопка **No**;
- `ResetRole` — кнопка для установки значений по умолчанию;
- `ApplyRole` — кнопка для принятия изменений.

Если роль недействительна, кнопка добавлена не будет.

Первый и второй форматы возвращают ссылку на сгенерированную и добавленную в контейнер кнопку, третий не возвращает ничего:

```
self.btnYes = QtWidget.QPushButton("&Да")
self.box.addButton(self.btnYes,
                   QtWidget.QDialogButtonBox.ButtonRole.AcceptRole)
self.btnNo = self.box.addButton(QtWidget.QDialogButtonBox.StandardButton.No)
self.btnCancel = self.box.addButton("&Cancel",
                     QtWidget.QDialogButtonBox.StandardButton.RejectRole)
```

- `button(<Стандартная кнопка StandardButton>)` — возвращает ссылку на стандартную кнопку, соответствующую указанному обозначению, или `None`, если такой стандартной кнопки в контейнере нет;
- `standardButton(<Кнопка QAbstractButton>)` — возвращает обозначение стандартной кнопки, переданной в качестве параметра, или значение элемента `NoButton`, если такой кнопки в контейнере нет;
- `standardButtons()` — возвращает комбинацию обозначений всех стандартных кнопок, добавленных в контейнер;
- `buttonRole(<Кнопка QAbstractButton>)` — возвращает роль указанной в параметре кнопки. Если такая кнопка отсутствует, возвращается значение элемента `InvalidRole`;
- `buttons()` — возвращает список со ссылками на все кнопки, которые были добавлены в контейнер;

- `removeButton(<Кнопка QAbstractButton>)` — удаляет заданную кнопку из контейнера, при этом не удаляя объект кнопки;
- `clear()` — очищает контейнер и удаляет все кнопки;
- `setCenterButtons(<Флаг>)` — если в качестве параметра указано значение `True`, кнопки будут выравниваться по центру контейнера.

Класс `QDialogButtonBox` поддерживает следующие сигналы:

- `accepted()` — генерируется при нажатии кнопки с ролью `AcceptRole` или `YesRole`. Этот сигнал можно соединить со слотом `accept()` объекта диалогового окна:

```
self.box.accepted.connect(self.accept)
```

- `rejected()` — генерируется при нажатии кнопки с ролью `RejectRole` или `NoRole`. Этот сигнал можно соединить со слотом `reject()` объекта диалогового окна:

```
self.box.rejected.connect(self.reject)
```

- `helpRequested()` — генерируется при нажатии кнопки с ролью `HelpRole`;
- `clicked(<Кнопка QAbstractButton>)` — генерируется при нажатии любой кнопки внутри контейнера. Внутри обработчика через параметр доступна ссылка на объект кнопки.

27.3. Класс *QMessageBox*

Класс `QMessageBox` реализует стандартные окна-предупреждения для вывода сообщений. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QDialog — QMessageBox
```

Форматы конструктора класса `QMessageBox`:

```
QMessageBox([parent=None])
QMessageBox(<Значок>, <Текст заголовка>, <Текст сообщения>[,
            buttons=StandardButton.NoButton][, parent=None][,
            flags=WindowType.Dialog | WindowType.MSWindowsFixedSizeDialogHint])
```

Если в параметре `parent` указана ссылка на родительское окно, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `flags` задает тип окна (см. *разд. 19.1.1*). В параметре `<Значок>` могут быть указаны следующие элементы перечисления `Icon` из класса `QMessageBox`:

- `NoIcon` — нет значка;
- `Question` — значок со знаком вопроса;
- `Information` — значок информационного сообщения;
- `Warning` — значок предупреждающего сообщения;
- `Critical` — значок критического сообщения.

В параметре `buttons` указываются следующие элементы (или их комбинация через оператор `|`) перечисления `StandardButton` из класса `QMessageBox`:

- `NoButton` — кнопки не установлены;
- `Ok` — кнопка **OK** с ролью `AcceptRole`;

- `Cancel` — кнопка **Cancel** с ролью `RejectRole`;
- `Yes` — кнопка **Yes** с ролью `YesRole`;
- `YesToAll` — кнопка **Yes to All** с ролью `YesRole`;
- `No` — кнопка **No** с ролью `NoRole`;
- `NoToAll` — кнопка **No to All** с ролью `NoRole`;
- `Open` — кнопка **Open** с ролью `AcceptRole`;
- `Close` — кнопка **Close** с ролью `RejectRole`;
- `Save` — кнопка **Save** с ролью `AcceptRole`;
- `SaveAll` — кнопка **Save All** с ролью `AcceptRole`;
- `Discard` — кнопка **Discard** или **Don't Save** (надпись на кнопке зависит от операционной системы) с ролью `DestructiveRole`;
- `Apply` — кнопка **Apply** с ролью `ApplyRole`;
- `Reset` — кнопка **Reset** с ролью `ResetRole`;
- `RestoreDefaults` — кнопка **Restore Defaults** с ролью `ResetRole`;
- `Help` — кнопка **Help** с ролью `HelpRole`;
- `Abort` — кнопка **Abort** с ролью `RejectRole`;
- `Retry` — кнопка **Retry** с ролью `AcceptRole`;
- `Ignore` — кнопка **Ignore** с ролью `AcceptRole`.

После создания объекта класса следует вызвать метод `exec()`, чтобы вывести окно на экран. Метод возвращает числовое обозначение нажатой кнопки:

```
dialog = QtWidgets.QMessageBox(QtWidgets.QMessageBox.Icon.Critical,
                               "Текст заголовка", "Текст сообщения",
                               buttons=QtWidgets.QMessageBox.StandardButton.Ok |
                               QtWidgets.QMessageBox.StandardButton.Cancel,
                               parent=window)
result = dialog.exec()
```

27.3.1. Основные методы и сигналы

Класс `QMessageBox` наследует все методы базовых классов и поддерживает следующие дополнительные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qmessagebox.html**):

- `setIcon(<Значок Icon>)` — устанавливает стандартный значок;
- `setIconPixmap(<Значок QPixmap>)` — устанавливает пользовательский значок;
- `setWindowTitle(<Текст заголовка>)` — задает текст заголовка окна;
- `setText(<Текст сообщения>)` — задает текст сообщения. Можно указать обычный текст, текст в формате Markdown или HTML-код. Перенос строки в обычной строке осуществляется с помощью символа `\n`, а в строке в формате HTML — с помощью тега `<br>`;
- `setInformativeText(<Текст>)` — задает текст дополнения, которое выводится под сообщением. Можно указать обычный текст, текст в формате Markdown или HTML-код;

- `setDetailedText(<Текст>)` — задает текст подробного сообщения. Если он задан, в окно будет добавлена кнопка **Show Details**, с помощью которой можно отобразить скрытую панель с подробным сообщением;
- `setTextFormat(<Режим>)` — задает режим отображения текста сообщения. Могут быть указаны следующие элементы перечисления `TextFormat` из модуля `QtCore.Qt`:
  - `PlainText` — простой текст;
  - `RichText` — HTML-код;
  - `AutoText` — автоматическое определение (режим по умолчанию). Если текст содержит HTML-теги, используется режим `RichText`, в противном случае — режим `PlainText`;
  - `MarkdownText` — текст в формате Markdown;
- `setStandardButtons(<Стандартные кнопки>)` — добавляет стандартные кнопки:

```
dialog.setStandardButtons(QtWidgets.QMessageBox.StandardButton.Ok |
                          QtWidgets.QMessageBox.StandardButton.Cancel)
```

- `addButton()` — добавляет кнопку в окно. Форматы метода:

```
addButton(<Стандартная кнопка StandardButton>)
addButton(<Текст>, <Роль>)
addButton(<Кнопка QAbstractButton>, <Роль>)
```

Первый формат добавляет стандартную кнопку или кнопки (в виде одного из рассмотренных ранее элементов перечисления `StandardButton` или их комбинации через оператор `|`). Второй формат принимает в качестве первого параметра надпись для добавляемой кнопки, а в качестве второго — ее роль. Третий формат принимает добавляемую кнопку в виде объекта одного из подклассов класса `QAbstractButton` — как правило, класса `QPushButton`, представляющего обычную кнопку.

В качестве роли указывается один из следующих элементов перечисления `ButtonRole` из класса `QMessageBox`:

- `InvalidRole` — ошибочная роль;
- `AcceptRole` — нажатие кнопки устанавливает код возврата равным значению элемента `Accepted`;
- `RejectRole` — нажатие кнопки устанавливает код возврата равным значению элемента `Rejected`;
- `DestructiveRole` — кнопка для отказа от изменений и закрытия диалогового окна;
- `ActionRole` — нажатие кнопки приводит к выполнению операции, которая не связана с закрытием окна;
- `HelpRole` — кнопка для отображения справки;
- `YesRole` — кнопка **Yes**;
- `NoRole` — кнопка **No**;
- `ResetRole` — кнопка для установки значений по умолчанию;
- `ApplyRole` — кнопка для принятия изменений.

Если роль недействительна, кнопка добавлена не будет.

Первый и второй форматы возвращают ссылку на сгенерированную и добавленную в окно кнопку, третий не возвращает ничего:

```
btnYes = QtWidgets.QPushButton("&Да")
dialog.addButton(btnYes, QtWidgets.QMessageBox.ButtonRole.AcceptRole)
btnNo = dialog.addButton("&Нет",
                         QtWidgets.QMessageBox.ButtonRole.RejectRole)
btnCancel = dialog.addButton(QtWidgets.QMessageBox.StandardButton.Cancel)
```

- `setDefaultButton()` — задает кнопку по умолчанию, которая сработает при нажатии клавиши <Enter>. Форматы метода:

```
setDefaultButton(<Стандартная кнопка StandardButton>)
setDefaultButton(<Кнопка QPushButton>)
```

- `setEscapeButton()` — задает кнопку, которая сработает при нажатии клавиши <Esc>. Форматы метода:

```
setEscapeButton(<Стандартная кнопка StandardButton>)
setEscapeButton(<Кнопка QAbstractButton>)
```

- `setCheckBox(<Флажок QCheckBox>)` — задает флажок, который будет выводиться в окне. Чтобы убрать заданный ранее флажок, нужно передать значение `None`;
- `checkBox()` — возвращает заданный в методе `setCheckBox()` флажок или `None`, если такового нет;
- `clickedButton()` — возвращает ссылку на кнопку, которая была нажата, или значение `None`;
- `button(<Стандартная кнопка StandardButton>)` — возвращает ссылку на стандартную кнопку, соответствующую указанному обозначению, или `None`, если такой стандартной кнопки в контейнере нет;
- `standardButton(<Кнопка QAbstractButton>)` — возвращает обозначение стандартной кнопки, переданной в качестве параметра, или значение элемента `NoButton`, если такой кнопки в контейнере нет;
- `standardButtons()` — возвращает комбинацию обозначений всех стандартных кнопок, добавленных в окно;
- `buttonRole(<Кнопка QAbstractButton>)` — возвращает роль указанной в параметре кнопки. Если такая кнопка отсутствует, метод возвращает значение элемента `InvalidRole`;
- `buttons()` — возвращает список со ссылками на все кнопки, которые были добавлены в окно;
- `removeButton(<Кнопка QAbstractButton>)` — удаляет заданную кнопку из окна, при этом не удаляя объект кнопки.

Класс `QMessageBox` поддерживает сигнал `buttonClicked(<Кнопка QAbstractButton>)`, генерируемый при нажатии кнопки в окне. Внутри обработчика через параметр доступна ссылка на нажатую кнопку.

27.3.2. Окно информационного сообщения

Помимо рассмотренных ранее, класс `QMessageBox` предлагает несколько статических методов, выводящих типовые окна-предупреждения.

Для вывода информационного сообщения предназначен статический метод `information()`. Формат метода:

```
information(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
        buttons=StandardButton.Ok][, defaultButton=StandardButton.NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать стандартные кнопки (элементы перечисления, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию выводится кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `information()` возвращает числовое обозначение нажатой кнопки:

```
QtWidgets.QMessageBox.information(window, "Текст заголовка",
                    "Текст сообщения",
                    buttons=QtWidgets.QMessageBox.StandardButton.Close,
                    defaultButton=QtWidgets.QMessageBox.StandardButton.Close)
```

Результат выполнения этого кода показан на рис. 27.1.

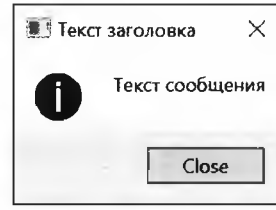


Рис. 27.1. Информационное окно-сообщение

27.3.3. Окно подтверждения

Для вывода окна с запросом подтверждения каких-либо действий предназначен статический метод `question()`. Формат метода:

```
question(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
        buttons=StandardButton.Yes | StandardButton.No][,
        defaultButton=StandardButton.NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать выводимые стандартные кнопки (элементы перечисления, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображаются кнопки **Yes и No**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `question()` возвращает числовое обозначение нажатой кнопки:

```
result = QtWidgets.QMessageBox.question(window, "Текст заголовка",
                    "Вы действительно хотите выполнить действие?",
                    buttons=QtWidgets.QMessageBox.StandardButton.Yes |
                            QtWidgets.QMessageBox.StandardButton.No |
                    QtWidgets.QMessageBox.StandardButton.Cancel,
                    defaultButton=QtWidgets.QMessageBox.StandardButton.Cancel)
```

Результат выполнения этого кода показан на рис. 27.2.

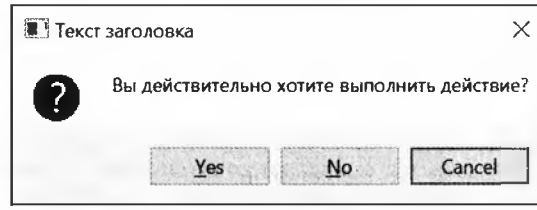


Рис. 27.2. Окно запроса подтверждения

27.3.4. Окно предупреждающего сообщения

Для вывода окна с предупреждающим сообщением предназначен статический метод `warning()`. Формат метода:

```
warning(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
        buttons=StandardButton.Ok][, defaultButton=StandardButton.NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать выводимые стандартные кнопки (элементы перечисления, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `warning()` возвращает числовое обозначение нажатой кнопки:

```
result = QtWidgets.QMessageBox.warning(window, "Текст заголовка",
                  "Действие может быть опасным. Продолжить?",
                  buttons=QtWidgets.QMessageBox.StandardButton.Yes |
                          QtWidgets.QMessageBox.StandardButton.No |
                          QtWidgets.QMessageBox.StandardButton.Cancel,
                  defaultButton=QtWidgets.QMessageBox.StandardButton.Cancel)
```

Результат выполнения этого кода показан на рис. 27.3.

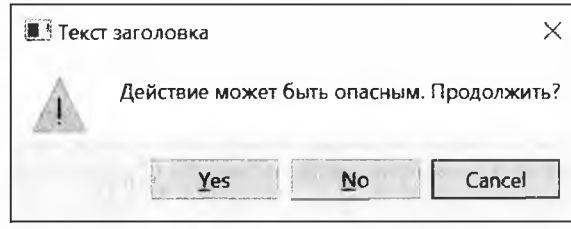


Рис. 27.3. Окно предупреждающего сообщения

27.3.5. Окно критического сообщения

Для вывода окна с критическим сообщением предназначен статический метод `critical()`. Формат метода:

```
critical(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
         buttons=StandardButton.Ok][, defaultButton=StandardButton.NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать выводящиеся на экран стандартные кнопки (элементы перечисления, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `critical()` возвращает числовое обозначение нажатой кнопки:

```
QtWidgets.QMessageBox.critical(window, "Текст заголовка",
        "Программа выполнила недопустимую ошибку и будет закрыта",
        defaultButton=QtWidgets.QMessageBox.StandardButton.Ok)
```

Результат выполнения этого кода показан на рис. 27.4.

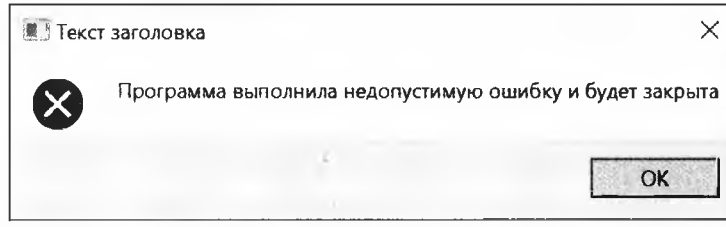


Рис. 27.4. Окно критического сообщения

27.3.6. Окно сведений о программе

Для вывода окна со сведениями о программе и авторских правах ее разработчиков предназначен статический метод `about()`. Формат метода:

```
about(<Родитель>, <Текст заголовка>, <Текст сообщения>)
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Слева от текста сообщения отображается значок приложения (см. *разд. 19.8*), если он был установлен:

```
QtWidgets.QMessageBox.about(window, "Текст заголовка", "Описание программы")
```

27.3.7. Окно сведений о фреймворке Qt

Для вывода окна с описанием используемой версии фреймворка Qt предназначен статический метод `aboutQt()`. Формат метода:

```
aboutQt(<Родитель>[, title=""])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. В параметре `title` можно указать текст, выводимый в заголовке окна. Если параметр не указан, то выводится заголовок **About Qt**:

```
QtWidgets.QMessageBox.aboutQt(window, title="О фреймворке Qt")
```

Результат выполнения этого кода показан на рис. 27.5.

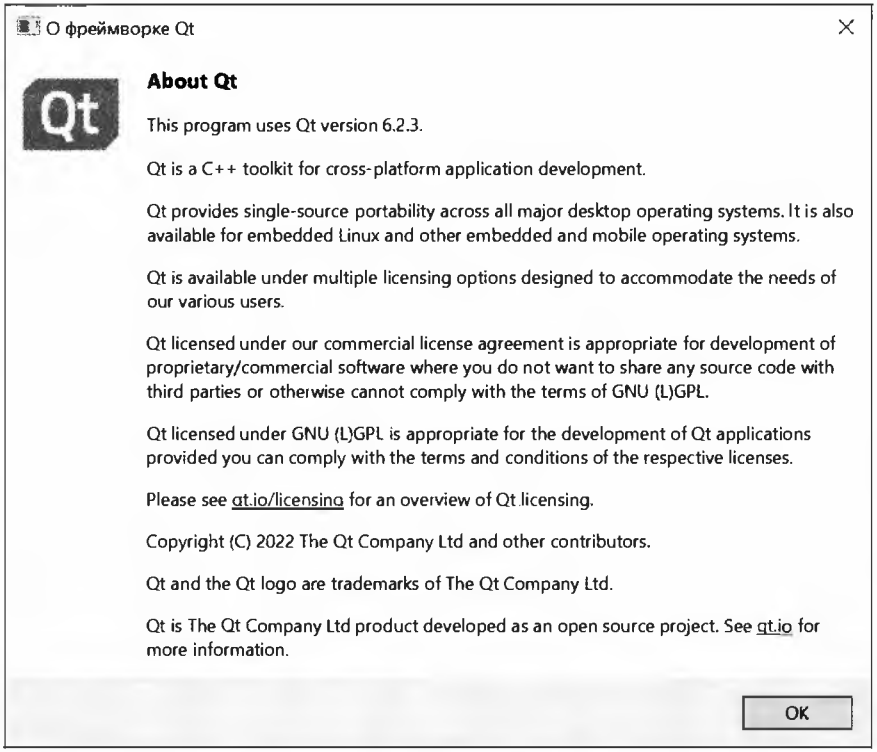


Рис. 27.5. Окно сведений о фреймворке Qt

27.4. Класс *QInputDialog*

Класс `QInputDialog` представляет модальные диалоговые окна для ввода различных данных. Иерархия наследования для этого класса выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QInputDialog
```

Формат конструктора класса `QInputDialog`:

```
QInputDialog([parent=None][, ][flags=0])
```

Если в параметре `parent` указана ссылка на родительское окно, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `flags` задает тип окна (см. *разд. 19.1.1*).

После создания объекта класса следует вызвать метод `exec()`, чтобы отобразить окно. Метод возвращает код возврата в виде одного из следующих элементов перечисления `DialogCode` из класса `QDialog`: `Accepted` или `Rejected`.

Вот пример отображения диалогового окна и обработки статуса внутри обработчика нажатия кнопки:

```
def on_clicked():
    dialog = QtWidgets.QInputDialog(window)
    result = dialog.exec()
    if result == QtWidgets.QDialog.DialogCode.Accepted:
```

```
        print("Нажата кнопка OK")
        # Здесь получаем данные из диалогового окна
    else:
        print("Нажата кнопка Cancel")
```

27.4.1. Основные методы и сигналы

Класс `QInputDialog` наследует все методы базовых классов и добавляет к ним следующие собственные методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qinputdialog.html**):

- `setLabelText(<Текст>)` — задает текст, отображаемый над полем ввода;
- `setOkButtonText(<Текст>)` — задает текст надписи для кнопки **OK**:

  ```
  dialog.setOkButtonText("&Принять")
  ```

- `setCancelButtonText(<Текст>)` — задает текст надписи для кнопки **Cancel**:

  ```
  dialog.setCancelButtonText("&Вернуться")
  ```

- `setInputMode(<Режим>)` — задает режим ввода данных. В качестве параметра указываются следующие элементы перечисления `InputMode` из класса `QInputDialog`:
  - `TextInput` — ввод текста;
  - `IntInput` — ввод целого числа;
  - `DoubleInput` — ввод вещественного числа;
- `setTextEchoMode(<Режим>)` — задает режим отображения текста в поле ввода. Могут быть указаны следующие элементы перечисления `EchoMode` из класса `QLineEdit`:
  - `Normal` — показывать вводимые символы;
  - `NoEcho` — не показывать вводимые символы;
  - `Password` — вместо символов выводить звездочки (*);
  - `PasswordEchoOnEdit` — показывать символы при вводе, а при потере фокуса вместо них отображать звездочки;
- `setTextValue(<Текст>)` — задает текст по умолчанию, отображаемый в поле ввода;
- `textValue()` — возвращает текст, занесенный в поле ввода;
- `setIntValue(<Значение>)` — задает целочисленное значение при использовании режима `IntInput`;
- `intValue()` — возвращает целочисленное значение, введенное в поле, при использовании режима `IntInput`;
- `setIntRange(<Минимум>, <Максимум>)`, `setIntMinimum(<Минимум>)` и `setIntMaximum(<Максимум>)` — задают диапазон допустимых целочисленных значений при использовании режима `IntInput`;
- `setIntStep(<Шаг>)` — задает шаг приращения значения при нажатии кнопок со стрелками в правой части поля при использовании режима `IntInput`;
- `setDoubleValue(<Значение>)` — задает вещественное значение при использовании режима `DoubleInput`;

- `doubleValue()` — возвращает вещественное значение, введенное в поле, при использовании режима `DoubleInput`;
- `setDoubleRange(<Минимум>, <Максимум>)`, `setDoubleMinimum(<Минимум>)` и `setDoubleMaximum(<Максимум>)` — задают диапазон допустимых вещественных значений при использовании режима `DoubleInput`;
- `setDoubleDecimals(<Количество>)` — задает количество цифр после десятичной точки при использовании режима `DoubleInput`;
- `setDoubleStep(<Шаг>)` — задает шаг приращения значения при нажатии кнопок со стрелками в правой части поля при использовании режима `DoubleInput`;
- `setComboBoxItems(<Список строк>)` — задает пункты, которые будут присутствовать в раскрывающемся списке, выводящемся в этом случае вместо обычного поля ввода. Набор пунктов указывается в виде списка строк;
- `setComboBoxEditable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь сможет ввести произвольное значение в раскрывающийся список, как если бы он был обычным полем ввода;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, заданная в первом параметре опция будет установлена, а если `False` — сброшена. Опции указываются в виде следующих элементов перечисления `InputDialogOption` из класса `QInputDialog`:
  - `NoButtons` — не выводить кнопки **OK** и **Cancel**;
  - `UseListViewForComboBoxItems` — для отображения списка строк будет использоваться класс `QListView` (обычный список), а не `QComboBox` (раскрывающийся список);
  - `UsePlainTextEditForTextInput` — вместо класса `QLineEdit` (поле ввода) будет использован класс `QPlainTextEdit` (область редактирования для обычного текста);
- `setOptions(<Опции>)` — сразу устанавливает несколько опций (см. описание метода `setOption()`).

Класс `QInputDialog` поддерживает сигналы:

- `textValueChanged(<Значение>)` — генерируется при изменении значения в текстовом поле. Внутри обработчика через параметр доступно новое значение в виде строки. Сигнал генерируется при использовании режима `TextInput`;
- `textValueSelected(<Значение>)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение в виде строки. Сигнал генерируется при использовании режима `TextInput`;
- `intValueChanged(<Значение>)` — генерируется при изменении значения в поле. Внутри обработчика через параметр доступно новое значение в виде целого числа. Сигнал генерируется при использовании режима `IntInput`;
- `intValueSelected(<Значение>)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение в виде целого числа. Сигнал генерируется при использовании режима `IntInput`;
- `doubleValueChanged(<Значение>)` — генерируется при изменении значения в поле. Внутри обработчика через параметр доступно новое значение в виде вещественного числа. Сигнал генерируется при использовании режима `DoubleInput`;

- doubleValueSelected(<Значение>) — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение в виде вещественного числа. Сигнал генерируется при использовании режима DoubleInput.

27.4.2. Окно для ввода строки

Класс QInputDialog также поддерживает несколько статических методов, реализующих типовые диалоговые окна.

Окно для ввода обычной строки или пароля выводится вызовом статического метода getText(). Формат метода:

```
getText(<Родитель>, <Текст заголовка>, <Текст подсказки>[,
        echo=EchoMode.Normal][, text=""][, flags=0][,
        inputMethodHints=InputMethodHint.ImhNone])
```

В параметре <Родитель> указывается ссылка на родительское окно или значение None. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не экрана. Необязательный параметр echo задает режим отображения текста в поле (см. описание метода setTextEchoMode() в *разд. 27.4.1*). Параметр text устанавливает значение поля по умолчанию, а в параметре flags можно указать тип окна.

Параметр inputMethodHints указывает дополнительные опции поля ввода, присутствующего в выведенном окне. В качестве его значения можно указать один из многочисленных элементов перечисления InputMethodHint из модуля QtCore.Qt, список которых приведен на странице **https://doc.qt.io/qt-6/qt.html#InputMethodHint-enum**, или их комбинацию через оператор |.

Метод getText() возвращает кортеж из двух элементов: (<Значение>, <Статус>). Через первый элемент доступно введенное значение, а через второй — значение True, если была нажата кнопка **OK**, или значение False, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка закрытия в заголовке окна. Пример:

```
s, ok = QtWidgets.QInputDialog.getText(window, "Это заголовок окна",
                  "Это текст подсказки", text="Значение по умолчанию")
if ok:
    print("Введено значение:", s)
```

Результат выполнения этого кода показан на рис. 27.6.

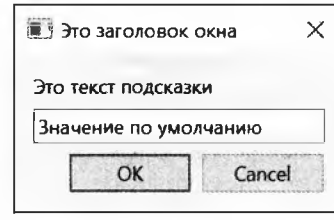


Рис. 27.6. Окно для ввода строки

27.4.3. Окно для ввода целого числа

Окно для ввода целого числа реализуется с помощью статического метода getInt(). Формат метода:

```
getInt(<Родитель>, <Текст заголовка>, <Текст подсказки>[, value=0][,
       min=-2147483647][, max=2147483647][, step=1][, flags=0])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `value` устанавливает значение поля по умолчанию. Параметр `min` задает минимальное значение, параметр `max` — максимальное значение, а параметр `step` — шаг приращения. Параметр `flags` позволяет указать тип окна. Методы возвращают кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй — значение `True`, если была нажата кнопка **OK**, или значение `False`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка закрытия в заголовке окна. Пример:

```
n, ok = QtWidgets.QInputDialog.getInt(window, "Это заголовок окна",
                                      "Это текст подсказки",
                                      value=50, min=0, max=100, step=2)
if ok:
    print("Введено значение:", n)
```

Результат выполнения этого кода показан на рис. 27.7.

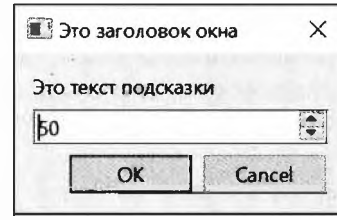


Рис. 27.7. Окно для ввода целого числа

27.4.4. Окно для ввода вещественного числа

Окно для ввода вещественного числа реализуется с помощью статического метода `getDouble()`. Формат метода:

```
getDouble(<Родитель>, <Текст заголовка>, <Текст подсказки>[, value=0][,
   min=-2147483647][, max=2147483647][, decimals=1][, step=1][, flags=0])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `value` устанавливает значение поля по умолчанию. Параметр `min` задает минимальное значение, параметр `max` — максимальное значение, параметр `decimals` — количество цифр после десятичной точки, а параметр `step` — шаг приращения. Параметр `flags` позволяет указать тип окна. Метод возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй элемент — значение `True`, если была нажата кнопка **OK**, или значение `False`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка закрытия в заголовке окна. Пример:

```
n, ok = QtWidgets.QInputDialog.getDouble(window, "Это заголовок окна",
                  "Это текст подсказки", value=50.0, min=0.0, max=100.0,
                  decimals=2, step=2.0)
```

```
if ok:
    print("Введено значение:", n)
```

27.4.5. Окно для выбора пункта из списка

Окно для выбора пункта из списка выводится статическим методом `getItem()`. Формат метода:

```
getItem(<Родитель>, <Текст заголовка>, <Текст подсказки>, <Список строк>[,
        current=0][, editable=True][, flags=0][,
        inputMethodHints=InputMethodHint.ImhNone])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `current` устанавливает индекс пункта, выбранного по умолчанию. Если в параметре `editable` указано значение `True`, пользователь может ввести произвольное значение в список вручную. Параметр `flags` позволяет указать тип окна.

Параметр `inputMethodHints` указывает дополнительные опции списка, который будет присутствовать в выведенном окне. Его имеет смысл указывать только в том случае, если параметру `editable` дано значение `True`. В качестве его значения можно указать один из многочисленных элементов перечисления `InputMethodHint` из модуля `QtCore.Qt`, список которых приведен на странице **https://doc.qt.io/qt-6/qt.html#InputMethodHint-enum**, или их комбинацию через оператор `|`.

Метод возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступен текст выбранного пункта, а через второй — значение `True`, если была нажата кнопка **OK**, или значение `False`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка закрытия в заголовке окна. Пример:

```
s, ok = QtWidgets.QInputDialog.getItem(window, "Это заголовок окна",
        "Это текст подсказки", ["Пункт 1", "Пункт 2", "Пункт 3"],
        current=1, editable=False)
if ok:
    print("Текст выбранного пункта:", s)
```

Результат выполнения этого кода показан на рис. 27.8.

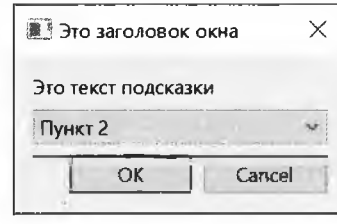


Рис. 27.8. Окно для выбора пункта из списка

27.4.6. Окно для ввода большого текста

Окно для ввода большого фрагмента обычного текста выводится статическим методом `getMultiLineText()`. Формат метода:

```
getMultiLineText(<Родитель>, <Текст заголовка>, <Текст подсказки>[,
            text=""][, flags=0][, inputMethodHints=InputMethodHint.ImhNone])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `text` устанавливает значение, выводимое в области редактирования по умолчанию, а в параметре `flags` можно указать тип окна.

Параметр `inputMethodHints` указывает дополнительные опции области редактирования, которое будет присутствовать в выведенном окне. В качестве его значения можно указать один из элементов перечисления `InputMethodHint` или их комбинацию через оператор `|`.

Метод `getMultiLineText()` возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй — значение `True`, если была нажата кнопка **OK**, или значение `False`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка закрытия в заголовке окна. Пример:

```
s, ok = QtWidgets.QInputDialog.getMultiLineText(window, "Это заголовок окна",
                  "Это текст подсказки", text = "Текст\nТекст\nТекст")
if ok:
    print("Текст выбранного пункта:", s)
```

Результат выполнения этого кода показан на рис. 27.9.

Рис. 27.9. Окно для ввода большого фрагмента текста

27.5. Класс *QFileDialog*

Класс `QFileDialog` реализует модальные диалоговые окна для выбора файла или каталога. Иерархия наследования для него выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QFileDialog
```

Форматы конструктора класса `QFileDialog`:

```
QFileDialog(<Родитель>, <Тип окна>)
QFileDialog([parent=None][, caption=""][, directory=""][, filter=""])
```

Если в параметрах <Родитель> и parent указана ссылка на родительское окно, диалоговое окно будет центрироваться относительно родительского окна, а не экрана. Параметр <Тип окна> задает тип окна (см. *разд. 19.1.1*). Необязательный параметр caption задает заголовок окна, параметр directory — начальный каталог, а параметр filter — фильтр для отбора файлов, которые будут выведены в диалоговом окне (например, фильтр "Images (*.png *.jpg)" задаст вывод только файлов с расширениями png и jpg).

После создания объекта класса следует вызвать метод exec(), чтобы вывести диалоговое окно на экран. Метод возвращает код возврата в виде значения элементов Accepted или Rejected перечисления DialogCode из класса QDialog.

27.5.1. Основные методы и сигналы

Класс QFileDialog наследует все методы базовых классов и определяет следующие собственные методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qfiledialog.html**):

- setAcceptMode(<Тип>) — задает тип окна. В качестве параметра указываются следующие элементы перечисления AcceptMode из класса QFileDialog:
  - AcceptOpen — окно для открытия файла (по умолчанию);
  - AcceptSave — окно для сохранения файла;
- setViewMode(<Режим>) — задает режим вывода списка файлов в окне. В качестве параметра указываются следующие элементы перечисления ViewMode из класса QFileDialog:
  - Detail — отображается подробная информация о файлах;
  - List — отображаются только значки и имена файлов;
- setFileMode(<Тип>) — задает тип элементов, которые пользователь может выбрать в окне. В качестве параметра указываются следующие элементы перечисления FileMode из класса QFileDialog:
  - AnyFile — любой файл, независимо от того, существует он или нет;
  - ExistingFile — только существующий файл;
  - Directory — каталог;
  - ExistingFiles — произвольное количество существующих файлов. Несколько файлов можно выбрать, удерживая нажатой клавишу <Ctrl>;
- setOption(<Опция>[, on=True]) — если во втором параметре указано значение True, указанная в первом параметре опция будет установлена, а если False — сброшена. В первом параметре можно указать следующие элементы перечисления Option из класса QFileDialog (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qfiledialog.html#Option-enum**):
  - ShowDirsOnly — отображать только имена каталогов. Опция работает лишь при использовании типа выбираемых элементов Directory;
  - DontConfirmOverwrite — не спрашивать разрешения на перезапись существующего файла;
  - ReadOnly — режим только для чтения;
  - HideNameFilterDetails — скрывает детали фильтра;

- `setOptions(<Опции>)` — устанавливает сразу несколько опций;
- `setDirectory()` — задает отображаемый каталог. Форматы метода:

```
setDirectory(<Путь>)
setDirectory(<Каталог QDir>)
```

- `directory()` — возвращает объект класса `QDir` с путем к отображаемому каталогу;
- `setDirectoryUrl(<Путь QUrl>)` — задает отображаемый каталог в виде объекта класса `QUrl` из модуля `QtCore`:

```
dialog.setDirectoryUrl(QtCore.QUrl.fromLocalFile("C:\\book"))
```

- `directoryUrl()` — возвращает объект класса `QUrl` с путем к отображаемому каталогу;
- `setNameFilter(<Фильтр>)` — устанавливает фильтр. Чтобы установить несколько фильтров, необходимо указать их через две точки с запятой, например:

```
dialog.setNameFilter("All (*);;Images (*.png *.jpg)")
```

- `setNameFilters(<Список фильтров>)` — устанавливает сразу несколько фильтров:

```
dialog.setNameFilters(["All (*)", "Images (*.png *.jpg)"])
```

- `selectFile(<Имя файла>)` — выбирает указанный файл;
- `selectUrl(<Путь QUrl>)` — выбирает файл, путь к которому указан в виде объекта класса `QUrl`;
- `selectedFiles()` — возвращает список с выбранными файлами;
- `selectedUrls()` — возвращает список объектов класса `QUrl`, представляющих выбранные файлы;
- `setDefaultSuffix(<Расширение>)` — задает расширение, которое добавляется к файлу без явно указанного расширения;
- `setHistory(<Список>)` — задает список истории;
- `setSidebarUrls(<Список каталогов QUrl>)` — задает список каталогов, отображаемый на боковой панели окна:

```
dialog.setSidebarUrls([QtCore.QUrl.fromLocalFile("C:\\book"),
                       QtCore.QUrl.fromLocalFile("C:\\book\\eclipse")])
```

- `setLabelText(<Тип надписи>, <Текст>)` — позволяет изменить текст указанной надписи. В первом параметре указываются следующие элементы перечисления `DialogLabel` из класса `QFileDialog`:
  - `LookIn` — надпись слева от списка с каталогами;
  - `FileName` — надпись слева от поля с именем файла;
  - `FileType` — надпись слева от списка с типами файлов;
  - `Accept` — надпись на кнопке, нажатие которой приведет к подтверждению действия (по умолчанию **Open** или **Save**);
  - `Reject` — надпись на кнопке, нажатие которой приведет к отказу от действия (по умолчанию **Cancel**);
- `saveState()` — возвращает объект класса `QByteArray` с текущими параметрами диалогового окна;

- `restoreState(<Объект QByteArray>)` — восстанавливает параметры диалогового окна из заданного объекта и возвращает `True`, если восстановление прошло успешно, и `False` — в противном случае.

Класс `QFileDialog` поддерживает следующие сигналы:

- `currentChanged(<Путь>)` — генерируется при выборе другого файла. Внутри обработчика через параметр доступен путь к выбранному файлу в виде строки;
- `currentUrlChanged(<Путь QUrl>)` — генерируется при выборе другого файла. Внутри обработчика через параметр доступен путь к выбранному файлу;
- `directoryEntered(<Путь>)` — генерируется при выборе другого каталога. Внутри обработчика через параметр доступен путь к выбранному каталогу в виде строки;
- `directoryUrlEntered(<Путь QUrl>)` — генерируется при выборе другого каталога. Внутри обработчика через параметр доступен путь к выбранному каталогу;
- `fileSelected(<Путь>)` — генерируется при выборе файла и подтверждении выбора. Внутри обработчика через параметр доступен путь к выбранному файлу в виде строки;
- `filesSelected(<Список путей>)` — генерируется при выборе нескольких файлов и подтверждении выбора. Внутри обработчика через параметр доступен список с путями к выбранным файлам, заданными в виде строк;
- `urlSelected(<Путь QUrl>)` — генерируется при выборе файла и подтверждении выбора. Внутри обработчика через параметр доступен путь к выбранному файлу;
- `urlsSelected(<Список путей QUrl>)` — генерируется при выборе нескольких файлов и подтверждении выбора. Внутри обработчика через параметр доступен список с путями к выбранным файлам;
- `filterSelected(<Фильтр>)` — генерируется при выборе другого фильтра. Внутри обработчика через параметр доступен выбранный фильтр в виде строки.

27.5.2. Окно для выбора каталога

Помимо рассмотренных методов, класс `QFileDialog` поддерживает несколько статических методов, реализующих типовые диалоговые окна.

Окно для выбора каталога реализуется с помощью статических методов `getExistingDirectory()` и `getExistingDirectoryUrl()`. Форматы методов:

```
getExistingDirectory([parent=None][, caption=""][, directory=""][,
                                    options=Options.ShowDirsOnly])
getExistingDirectoryUrl([parent=None][, caption=""][, directory=QUrl()][,
                                      options=Options.ShowDirsOnly])
```

В параметре `parent` указывается ссылка на родительское окно или значение `None`. Необязательный параметр `directory` задает текущий каталог, а параметр `options` устанавливает опции (см. описание метода `setOption()` в *разд. 27.5.1*). Метод `getExistingDirectory()` возвращает выбранный каталог или пустую строку, а метод `getExistingDirectoryUrl()` — объект класса `QUrl` с выбранным путем или пустой объект. Пример:

```
dirName = QtWidgets.QFileDialog.getExistingDirectory(parent=window,
                                 directory=QtCore.QDir.currentPath())
```

```
dir = QtWidgets.QFileDialog.getExistingDirectoryUrl(parent=window,
                directory=QtCore.QUrl.fromLocalFile(QtCore.QDir.currentPath()))
dirName = dir.toLocalFile()
```

Результат выполнения этого кода показан на рис. 27.10.

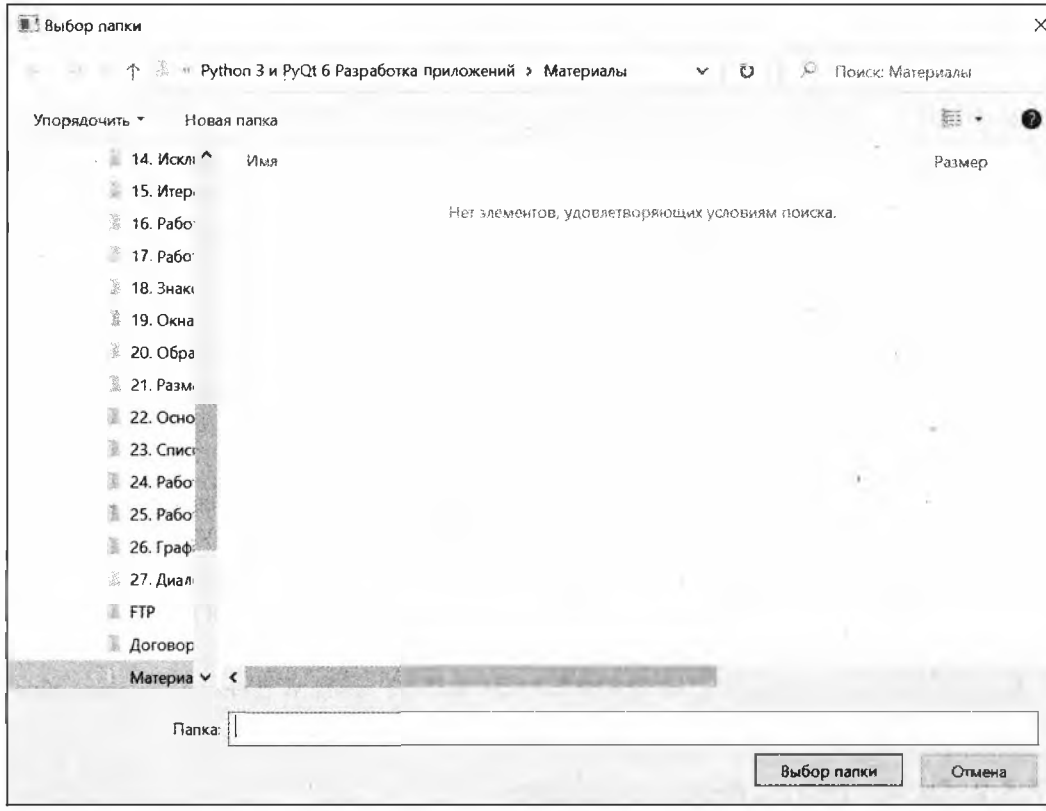


Рис. 27.10. Окно для выбора каталога

27.5.3. Окно для открытия файлов

Окно для открытия одного файла реализуется с помощью статических методов `getOpenFileName()` и `getOpenFileUrl()`. Форматы методов:

```
getOpenFileName([parent=None][, caption=""][, directory=""][, filter=""][,
                                              initialFilter=''][, options=0])
getOpenFileUrl([parent=None][, caption=""][, directory=QUrl()][, filter=""][,
                                             initialFilter=''][, options=0])
```

В параметре `parent` указывается ссылка на родительское окно или значение `None`. Необязательный параметр `caption` задает текст заголовка окна, параметр `directory` — текущий каталог, параметр `filter` — фильтр, параметр `initialFilter` — фильтр, который будет выбран изначально, а параметр `options` устанавливает опции (см. описание метода `setOption()` в *разд. 27.5.1*). Метод `getOpenFileName()` возвращает кортеж из двух элементов: первым элементом будет выбранный файл или пустая строка, вторым — выбранный

фильтр. Метод `getOpenFileUrl()` также возвращает кортеж из двух элементов: первый — объект класса `QUrl` с путем выбранного файла или пустой объект, второй — выбранный фильтр. Примеры:

```
file = QtWidgets.QFileDialog.getOpenFileName(parent=window,
                caption="Заголовок окна", directory="c:\\python310",
                filter="All (*);;Exes (*.exe *.dll)",
                initialFilter="Exes (*.exe *.dll)")
fileName = file[0]
file = QtWidgets.QFileDialog.getOpenFileUrl(parent=window,
                caption="Заголовок окна",
                directory=QtCore.QUrl("file:///c:\\python310"),
                filter="All (*);;Exes (*.exe *.dll)",
                initialFilter="Exes (*.exe *.dll)")
fileName = file[0].toLocalFile()
```

Результат выполнения кода из последнего примера показан на рис. 27.11.

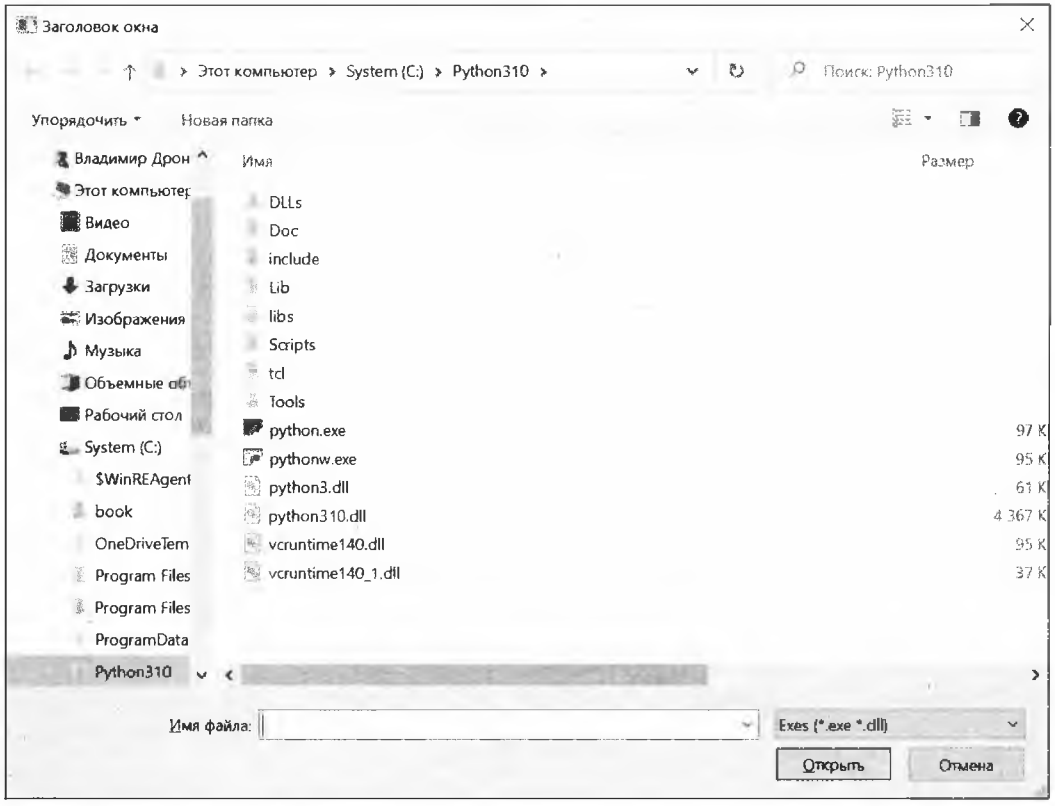


Рис. 27.11. Окно для открытия файла

Окно для открытия произвольного количества файлов реализуется с помощью статических методов `getOpenFileNames()` и `getOpenFileUrls()`. Форматы методов:

```
getOpenFileNames([parent=None][, caption=""][, directory=""][, filter=""][,
                                                initialFilter=''][, options=0])
```

```
getOpenFileUrls([parent=None][, caption=""][, directory=QUrl()][,filter=""][,
                                                initialFilter=''][, options=0])
```

Метод `getOpenFileNames()` возвращает кортеж из двух элементов: первый — список с путями к выбранным файлам или пустой список, второй — выбранный фильтр. Метод `getOpenFileUrls()` также возвращает кортеж из двух элементов: первый — список объектов класса `QUrl` с путями к выбранным файлам или пустой список, второй — выбранный фильтр. Примеры:

```
arr = QtWidgets.QFileDialog.getOpenFileNames(parent=window,
                caption="Заголовок окна", directory="c:\\python310",
                filter="All (*);;Exes (*.exe *.dll)",
                initialFilter="Exes (*.exe *.dll)")
files = arr[0]
arr = QtWidgets.QFileDialog.getOpenFileUrls(parent=window,
                caption="Заголовок окна",
                directory=QtCore.QUrl("file:///c:\\python310"),
                filter="All (*);;Exes (*.exe *.dll)",
                initialFilter="Exes (*.exe *.dll)")
files = list(a.toLocalFile() for a in arr[0])
```

27.5.4. Окно для сохранения файла

Окно для сохранения файла реализуется статическими методами `getSaveFileName()` и `getSaveFileUrl()`. Форматы методов:

```
getSaveFileName([parent=None][, caption=""][, directory=""][, filter=""][,
                                                initialFilter=''][, options=0])
getSaveFileUrl([parent=None][, caption=""][, directory=QUrl()][,filter=""][,
                                               initialFilter=''][, options=0])
```

В параметре `parent` указывается ссылка на родительское окно или значение `None`. Необязательный параметр `caption` задает текст заголовка окна, параметр `directory` — текущий каталог, параметр `filter` — фильтр, параметр `initialFilter` — фильтр, который будет выбран изначально, а параметр `options` устанавливает опции (см. описание метода `setOption()` в *разд. 27.5.1*). Метод `getSaveFileName()` возвращает кортеж из двух элементов: первым элементом будет выбранный файл или пустая строка, вторым — выбранный фильтр. Метод `getSaveFileUrl()` также возвращает кортеж из двух элементов: первый — объект класса `QUrl` с путем выбранного файла или пустой объект, второй — выбранный фильтр. Примеры:

```
f = QtWidgets.QFileDialog.getSaveFileName(parent=window,
                caption="Заголовок окна", directory=QtCore.QDir.currentPath(),
                filter="All (*);;Python Code (*.py *.pyw)")
fileName = f[0]
f = QtWidgets.QFileDialog.getSaveFileUrl(parent=window,
                caption="Заголовок окна",
                directory=QtCore.QUrl("file:///" + QtCore.QDir.currentPath()),
                filter="All (*);;Python Code (*.py *.pyw)")
fileName = f[0].toLocalFile()
```

Результат выполнения кода из последнего примера показан на рис. 27.12.

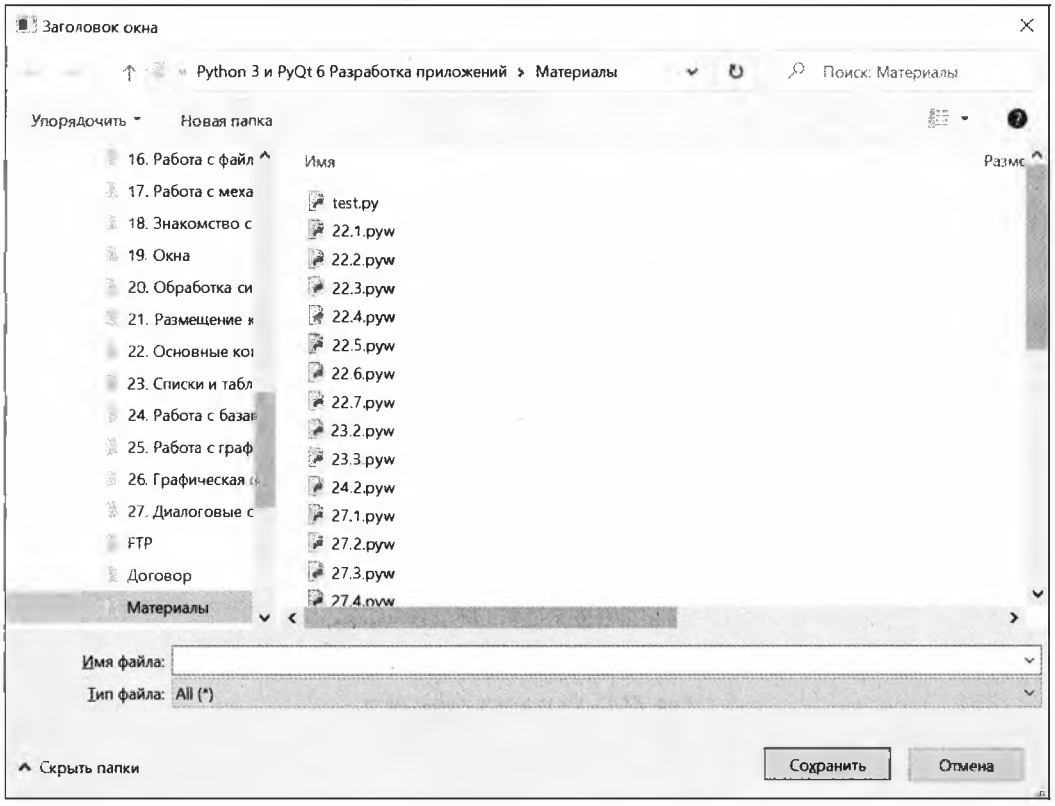


Рис. 27.12. Окно для сохранения файла

27.6. Окно для выбора цвета

Окно для выбора цвета (рис. 27.13) реализуется с помощью статического метода `getColor()` класса `QColorDialog`. Формат метода:

```
getColor([initial=GlobalColor.white][, parent=None][, title=""][, options=0])
```

Параметр `initial` задает начальный цвет. В параметре `parent` указывается ссылка на родительское окно или значение `None`. Параметр `title` позволяет указать заголовок окна. В параметре `options` могут быть указаны следующие элементы (или их комбинация) перечисления `ColorDialogOption` из класса `QColorDialog`:

- `ShowAlphaChannel` — пользователь может выбрать значение прозрачности;
- `NoButtons` — кнопки **OK** и **Cancel** не отображаются;
- `DontUseNativeDialog` — использовать встроенное в библиотеку Qt диалоговое окно выбора цвета вместо системного.

Метод возвращает объект класса `QColor`, представляющий выбранный цвет. Если пользователь нажмет кнопку **Cancel**, возвращенный объект будет невалидным. Пример:

```
color = QtWidgets.QColorDialog.getColor(initial=QtGui.QColor("#ff0000"),
        parent=window, title="Заголовок окна",
        options=QtWidgets.QColorDialog.ColorDialogOption.ShowAlphaChannel)
```

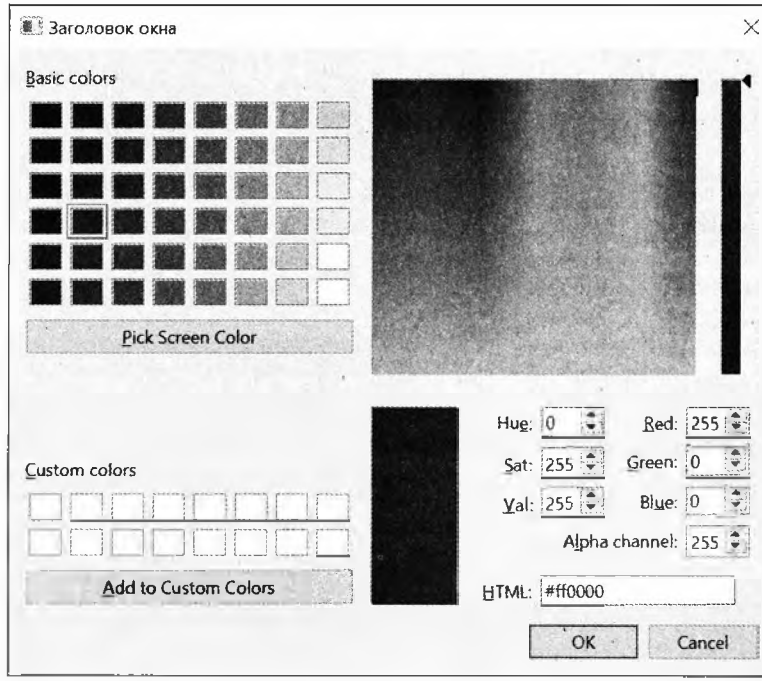


Рис. 27.13. Окно для выбора цвета

```
if color.isValid():
    print(color.red(), color.green(), color.blue(), color.alpha())
```

27.7. Окно для выбора шрифта

Окно для выбора шрифта реализуется с помощью статического метода `getFont()` класса `QFontDialog`. Форматы метода:

```
getFont([parent=None])
getFont(<Шрифт QFont>[, parent=None][, caption=""][, options=0])
```

Первый параметр во втором формате задает начальный шрифт. В параметре `parent` указывается ссылка на родительское окно или значение `None`. Параметр `caption` позволяет указать заголовок окна. В параметре `options` могут быть указаны следующие элементы перечисления `FontDialogOption` из класса `QFontDialog` или их комбинация:

- `NoButtons` — кнопки **OK** и **Cancel** не отображаются;
- `DontUseNativeDialog` — использовать встроенное в библиотеку Qt диалоговое окно выбора шрифта вместо системного;
- `ScalableFonts` — показать только масштабируемые шрифты;
- `NonScalableFonts` — показать только немасштабируемые шрифты;
- `MonospacedFonts` — показать только моноширинные шрифты;
- `ProportionalFonts` — показать только пропорциональные шрифты.

Метод возвращает кортеж из двух элементов: (<Выбранный шрифт>, <Статус>). Если второй элемент содержит значение True, первый элемент хранит объект класса QFont с выбранным шрифтом. Пример:

```
(font, ok) = QtWidgets.QFontDialog.getFont(QtGui.QFont("Tahoma", 16),
                                parent=window, caption="Заголовок окна")
if ok:
    print(font.family(), font.pointSize(), font.weight(),
          font.italic(), font.underline())
```

Результат выполнения этого кода показан на рис. 27.14.

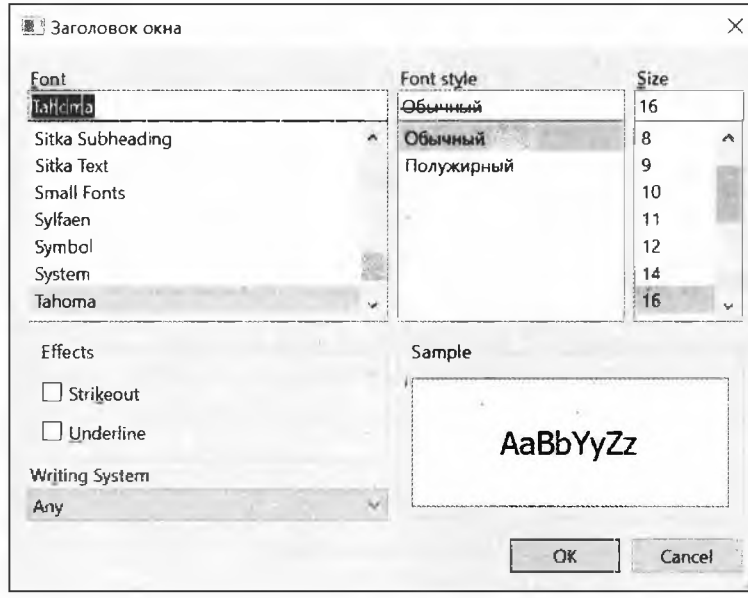


Рис. 27.14. Окно для выбора шрифта

27.8. Окно для вывода сообщения об ошибке

Класс QErrorMessage реализует немодальное диалоговое окно с сообщением об ошибке (рис. 27.15). Окно содержит надпись с собственно сообщением и флажок. Если пользователь снимает флажок, окно больше отображаться не будет. Иерархия наследования для класса QErrorMessage выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QErrorMessage
```

Формат конструктора класса QErrorMessage:

```
QErrorMessage([parent=None])
```

Для отображения окна предназначен метод showMessage(). Форматы его вызова:

```
showMessage(<Текст сообщения>)
showMessage(<Текст сообщения>, <Тип>)
```

Если пользователь установил флажок и тип сообщения не был задан, то все последующие сообщения об ошибках не будут выводиться на экран. Если же был указан тип сообщения

(в виде строки), то не будут выводиться лишь сообщения того же типа. Оба формата метода являются слотами. Пример:

```
ems = QtWidgets.QErrorMessage(parent=window)
ems.showMessage("Сообщение об ошибке")
ems.showMessage("Сообщение об ошибке типа warning", "warning")
```

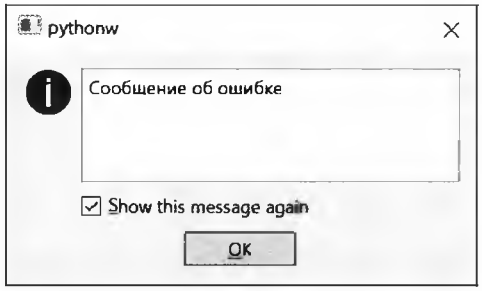


Рис. 27.15. Окно для вывода сообщения об ошибке

27.9. Окно с индикатором хода процесса

Класс `QProgressDialog` реализует диалоговое окно с индикатором хода процесса и кнопкой **Cancel** (рис. 27.16). Иерархия наследования для класса `QProgressDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QProgressDialog
```

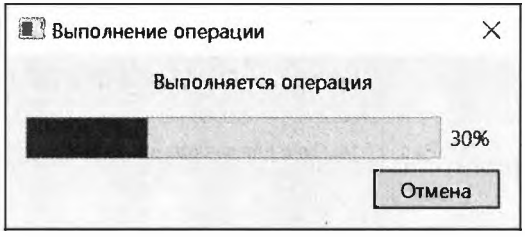


Рис. 27.16. Окно с индикатором хода процесса

Форматы конструктора класса `QProgressDialog`:

```
QProgressDialog([parent=None][, flags=0])
QProgressDialog(<Текст над индикатором>, <Текст на кнопке Cancel>,
                <Минимум>, <Максимум>[, parent=None][, flags=0])
```

Значения минимума и максимума должны быть заданы в виде целых чисел. Если в параметре `parent` указана ссылка на родительское окно, диалоговое окно будет центрироваться относительно родительского окна, а не экрана. Параметр `flags` задает тип окна (см. *разд. 19.1.1*).

Класс `QProgressDialog` наследует все методы базовых классов и дополнительно определяет следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qprogressdialog.html**):

- `setValue(<Значение>)` — задает новое значение индикатора, которое должно представлять собой целое число. Если диалоговое окно является модальным, при установке зна-

чения автоматически вызывается метод `processEvents()` объекта программы. Метод является слотом;

- `value()` — возвращает текущее значение индикатора в виде целого числа;
- `setLabelText(<Текст>)` — задает текст надписи, выводимой над индикатором. Метод является слотом;
- `setCancelButtonText(<Текст>)` — задает текст, выводимый на кнопке **Cancel**. Метод является слотом;
- `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное значения в виде целых чисел. Если оба значения равны нулю, то внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Методы являются слотами;
- `setMinimumDuration(<Значение>)` — устанавливает промежуток времени в миллисекундах перед отображением окна (задается целым числом — значение по умолчанию 4000). Окно может быть отображено ранее этого срока при установке значения индикатора. Метод является слотом;
- `reset()` — сбрасывает значение индикатора. Метод является слотом;
- `cancel()` — имитирует нажатие кнопки **Cancel**. Метод является слотом;
- `setLabel(<Надпись QLabel>)` — заменяет объект надписи на заданный;
- `setBar(<Индикатор QProgressBar>)` — заменяет объект индикатора на заданный;
- `setCancelButton(<Кнопка QPushButton>)` — заменяет объект кнопки **Cancel** на заданный;
- `setAutoClose(<Флаг>)` — если в качестве параметра указано значение `True`, при сбросе значения окно скрывается;
- `setAutoReset(<Флаг>)` — если в качестве параметра указано значение `True`, при достижении максимального значения будет автоматически произведен сброс;
- `wasCanceled()` — возвращает значение `True`, если была нажата кнопка **Cancel**.

Класс `QProgressDialog` поддерживает сигнал `canceled()`, который генерируется при нажатии кнопки **Cancel**.

27.10. Создание многостраничного мастера

PyQt позволяет создать *мастер* — диалоговое окно, в котором при нажатии кнопок **Back** (Назад) и **Next** (Далее) последовательно или в произвольном порядке отображаются различные страницы. Класс `QWizard` реализует контейнер для страниц, а отдельная страница описывается классом `QWizardPage`.

27.10.1. Класс *QWizard*

Класс `QWizard` реализует сам мастер — набор страниц, отображаемых последовательно или в произвольном порядке. Иерархия наследования для него выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QWizard
```

Формат конструктора класса `QWizard`:

```
QWizard([parent=None][, flags=0])
```

Помимо унаследованных методов, класс QWizard поддерживает следующие (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qwizard.html**):

- addPage(<Страница QWizardPage>) — добавляет заданную страницу в конец мастера и возвращает ее целочисленный идентификатор;
- setPage(<Идентификатор>, <Страница QWizardPage>) — добавляет заданную страницу в позицию, обозначенную указанным идентификатором;
- removePage(<Идентификатор>) — удаляет страницу с указанным идентификатором;
- page(<Идентификатор>) — возвращает ссылку на страницу (объект класса QWizardPage), соответствующую указанному идентификатору, или значение None, если такой страницы не существует;
- pageIds() — возвращает список с идентификаторами страниц;
- currentId() — возвращает идентификатор активной страницы;
- currentPage() — возвращает ссылку на активную страницу (объект класса QWizardPage) или None, если страницы не существует (что может случиться, если мастер еще не выведен на экран));
- setStartId(<Идентификатор>) — задает идентификатор начальной страницы. По умолчанию начальной станет страница с наименьшим идентификатором;
- startId() — возвращает идентификатор начальной страницы или -1, если страниц в мастере нет;
- visitedIds() — возвращает список с идентификаторами посещенных страниц или пустой список;
- hasVisitedPage(<Идентификатор>) — возвращает значение True, если страница с указанным идентификатором была посещена, и False — в противном случае;
- back() — имитирует нажатие кнопки **Back**. Метод является слотом;
- next() — имитирует нажатие кнопки **Next**. Метод является слотом;
- restart() — перезапускает мастер. Метод является слотом;
- nextId(self) — этот метод следует переопределить в классе, наследующем класс QWizard, если необходимо изменить порядок отображения страниц. Метод вызывается при нажатии кнопки **Next** и должен возвращать идентификатор следующей страницы или значение -1, если текущая страница — последняя;
- initializePage(self, <Идентификатор>) — этот метод следует переопределить в классе, наследующем QWizard, если необходимо производить настройку свойств компонентов на очередной странице на основе данных, введенных на предыдущих страницах. Метод вызывается при нажатии кнопки **Next** на активной странице, но до отображения следующей страницы. Если установлена опция IndependentPages, метод вызывается только при первом отображении страницы;
- cleanupPage(self, <Идентификатор>) — этот метод следует переопределить в классе, наследующем QWizard, если необходимо выполнять какие-либо завершающие действия перед переходом на предыдущую страницу после нажатия кнопки **Back**. Метод вызывается при нажатии кнопки **Back** на активной странице, но до отображения предыдущей страницы. Если установлена опция IndependentPages, метод не вызывается;

- `validateCurrentPage(self)` — этот метод следует переопределить в классе, наследующем `QWizard`, если необходимо проверять данные, введенные на активной странице. Метод вызывается при нажатии кнопки **Next** или **Finish**. Метод должен вернуть значение `True`, если данные корректны, и `False` — в противном случае. Если метод возвращает значение `False`, переход на следующую страницу не производится;
- `setField(<Свойство>, <Значение>)` — устанавливает значение указанного свойства. Создание свойства производится с помощью метода `registerField()` класса `QWizardPage`. Используя метод `setField()`, можно изменять значения компонентов, расположенных на разных страницах мастера;
- `field(<Свойство>)` — возвращает значение указанного свойства. Используя этот метод, также можно получить значения компонентов, расположенных на разных страницах мастера;
- `setWizardStyle(<Стиль>)` — задает стилевое оформление мастера. В качестве параметра указываются следующие элементы перечисления `WizardStyle` из класса `QWizard`:
  - `ClassicStyle` — классическое в стиле Windows;
  - `ModernStyle` — современное в стиле Windows;
  - `MacStyle` — в стиле macOS;
  - `AeroStyle` — в стиле Windows Aero;
- `setTitleFormat(<Формат>)` — задает формат отображения названия страницы. В качестве параметра указываются следующие элементы перечисления `TextFormat` из модуля `QtCore.Qt`:
  - `PlainText` — простой текст;
  - `RichText` — текст, отформатированный HTML-тегами;
  - `AutoText` — автоматическое определение (режим по умолчанию). Если текст содержит HTML-теги, то используется режим `RichText`, в противном случае — режим `PlainText`;
  - `MarkdownText` — текст, отформатированный Markdown-тегами;
- `setSubTitleFormat(<Формат>)` — задает формат отображения описания страницы. Допустимые значения см. в описании метода `setTitleFormat()`;
- `setButton(<Роль>, <Кнопка QAbstractButton>)` — добавляет кнопку для указанной роли. В первом параметре указываются следующие элементы перечисления `WizardButton` из класса `QWizard`:
  - `BackButton` — кнопка **Back**;
  - `NextButton` — кнопка **Next**;
  - `CommitButton` — кнопка **Commit**;
  - `FinishButton` — кнопка **Finish**;
  - `CancelButton` — кнопка **Cancel** (если установлена опция `NoCancelButton`, кнопка не отображается);
  - `HelpButton` — кнопка **Help** (чтобы отобразить кнопку, необходимо установить опцию `HaveHelpButton`);
  - `CustomButton1` — первая пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию `HaveCustomButton1`);

- CustomButton2 — вторая пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию HaveCustomButton2);
- CustomButton3 — третья пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию HaveCustomButton3);

- button(<Роль WizardButton>) — возвращает ссылку на кнопку с указанной ролью;
- setButtonText(<Роль WizardButton>, <Текст надписи>) — устанавливает текст надписи для кнопки с указанной ролью;
- buttonText(<Роль WizardButton>) — возвращает текст надписи кнопки с указанной ролью;
- setButtonLayout(<Список с ролями WizardButton>) — задает порядок отображения кнопок. В качестве параметра указывается список с ролями кнопок. Список также может содержать значение элемента Stretch перечисления WizardButton из класса QWizard, который создает растягивающееся свободное пространство между кнопками;
- setPixmap(<Роль>, <Изображение QPixmap>) — добавляет изображение для указанной роли. В первом параметре указываются следующие элементы перечисления WizardPixmap из класса QWizard:
  - WatermarkPixmap — изображение, которое занимает всю левую сторону страницы при использовании стилей ClassicStyle и ModernStyle;
  - LogoPixmap — небольшое изображение, отображаемое в правой части заголовка при использовании стилей ClassicStyle и ModernStyle;
  - BannerPixmap — фоновое изображение, отображаемое в заголовке страницы при использовании стиля ModernStyle;
  - BackgroundPixmap — фоновое изображение при использовании стиля MacStyle;
- setOption(<Опция>[, on=True]) — если во втором параметре указано значение True, заданная в первом параметре опция будет установлена, а если указано значение False — сброшена. В первом параметре указываются следующие элементы перечисления WizardOption из класса QWizard:
  - IndependentPages — страницы не зависят друг от друга. В этом случае метод initializePage() будет вызван только при первом отображении страницы, а метод cleanupPage() не вызывается;
  - IgnoreSubTitles — не отображать описания страниц в заголовке;
  - ExtendedWatermarkPixmap — изображение с ролью WatermarkPixmap будет занимать всю левую сторону страницы вплоть до нижнего края окна;
  - NoDefaultButton — не делать кнопки **Next** и **Finish** кнопками по умолчанию;
  - NoBackButtonOnStartPage — не отображать кнопку **Back** на стартовой странице;
  - NoBackButtonOnLastPage — не отображать кнопку **Back** на последней странице;
  - DisabledBackButtonOnLastPage — сделать кнопку **Back** неактивной на последней странице;
  - HaveNextButtonOnLastPage — показать неактивную кнопку **Next** на последней странице;
  - HaveFinishButtonOnEarlyPages — показать неактивную кнопку **Finish** на непоследних страницах;

- NoCancelButton — не отображать кнопку **Cancel**;
- CancelButtonOnLeft — поместить кнопку **Cancel** слева от кнопки **Back** (по умолчанию кнопка **Cancel** расположена справа от кнопок **Next** и **Finish**);
- HaveHelpButton — показать кнопку **Help**;
- HelpButtonOnRight — поместить кнопку **Help** у правого края окна;
- HaveCustomButton1 — показать пользовательскую кнопку с ролью CustomButton1;
- HaveCustomButton2 — показать пользовательскую кнопку с ролью CustomButton2;
- HaveCustomButton3 — показать пользовательскую кнопку с ролью CustomButton3;
- NoCancelButtonOnLastPage — не показывать кнопку **Cancel** на последней странице;

- setOptions(<Опции>) — устанавливает сразу несколько опций;
- setSideWidget(<Компонент>) — устанавливает заданный компонент в левую часть мастера. Если используется стиль ClassicStyle или ModernStyle, компонент будет выведен выше изображения с ролью WatermarkPixmap.

Класс QWizard поддерживает следующие сигналы:

- currentIdChanged(<Идентификатор>) — генерируется, когда активной становится другая страница. Внутри обработчика через параметр доступен целочисленный идентификатор новой активной страницы;
- customButtonClicked(<Роль>) — генерируется при нажатии кнопок с ролями CustomButton1, CustomButton2 и CustomButton3. В параметре доступна целочисленная роль нажатой кнопки;
- helpRequested() — генерируется при нажатии кнопки **Help**;
- pageAdded(<Идентификатор>) — генерируется при добавлении страницы. В параметре передается целочисленный идентификатор добавленной страницы;
- pageRemoved(<Идентификатор>) — генерируется при удалении страницы. В параметре передается целочисленный идентификатор удаленной страницы.

27.10.2. Класс *QWizardPage*

Класс QWizardPage описывает одну страницу в мастере. Его иерархия наследования такова:

```
(QObject, QPaintDevice) — QWidget — QWizardPage
```

Формат конструктора класса QWizardPage:

```
QWizardPage([parent=None])
```

Класс QWizardPage наследует все методы базовых классов и, помимо них, поддерживает также следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qwizardpage.html**):

- wizard() — возвращает ссылку на мастер (объект класса QWizard) или значение None, если страница не была добавлена в какой-либо мастер;
- setTitle(<Текст>) — задает название страницы;
- title() — возвращает название страницы;
- setSubTitle(<Текст>) — задает описание страницы;

- `subTitle()` — возвращает описание страницы;
- `setButtonText(<Роль>, <Текст надписи>)` — устанавливает текст надписи для кнопки с указанной ролью (допустимые значения параметра `<Роль>` см. в описании метода `setButton()` класса `QWizard` в *разд. 27.10.1*);
- `buttonText(<Роль>)` — возвращает текст надписи кнопки с указанной ролью;
- `setPixmap(<Роль>, <Изображение QPixmap>)` — добавляет изображение для указанной роли (допустимые значения параметра `<Роль>` см. в описании метода `setPixmap()` класса `QWizard` в *разд. 27.10.1*);
- `registerField()` — регистрирует свойство, с помощью которого можно получить доступ к значению компонента с любой страницы мастера. Формат метода:

  ```
  registerField(<Свойство>, <Компонент>[, property=None][, changedSignal=0])
  ```

  В параметре `<Свойство>` указывается произвольное имя свойства в виде строки. Если в конце строки указать символ `*`, компонент должен обязательно иметь значение (например, в поле должно быть введено какое-либо значение), иначе кнопки **Next** и **Finish** будут недоступны. Во втором параметре указывается ссылка на компонент, связываемый со свойством. После регистрации свойства изменить значение связанного с ним компонента позволяет метод `setField()`, а получить значение — метод `field()`.

  В параметре `property` может быть указано свойство компонента для получения и изменения значения, а в параметре `changedSignal` — сигнал, генерируемый при изменении значения.

  Назначить эти параметры для определенного класса позволяет также метод `setDefaultProperty(<Имя класса>, property, changedSignal)` класса `QWizard`. Для стандартных классов по умолчанию используются следующие свойства и сигналы (табл. 27.1).

Таблица 27.1. *Свойства и сигналы, используемые по умолчанию для стандартных классов*

Класс	Свойство	Сигнал
QAbstractButton	checked	toggled()
QAbstractSlider	value	valueChanged()
QComboBox	currentIndex	currentIndexChanged()
QDateTimeEdit	dateTime	dateTimeChanged()
QLineEdit	text	textChanged()
QListWidget	currentRow	currentRowChanged()
QSpinBox	value	valueChanged()

- `setField(<Свойство>, <Значение>)` — устанавливает значение указанного свойства. С помощью этого метода можно изменять значения компонентов, расположенных на разных страницах мастера;
- `field(<Свойство>)` — возвращает значение указанного свойства. С помощью этого метода можно получать значения компонентов, расположенных на разных страницах мастера;

- `setFinalPage(<Флаг>)` — если в качестве параметра указано значение `True`, текущая страница станет последней, и на ней будет отображаться кнопка **Finish**;
- `isFinalPage()` — возвращает значение `True`, если текущая страница является последней, и `False` — в противном случае;
- `setCommitPage(<Флаг>)` — если в качестве параметра указано значение `True`, на странице будет отображаться кнопка **Commit**;
- `isCommitPage()` — возвращает значение `True`, если на странице отображается кнопка **Commit**, и `False` — в противном случае;
- `isComplete(self)` — этот метод вызывается, чтобы определить, должны ли кнопки **Next** и **Finish** быть доступными (метод возвращает значение `True`) или недоступными (метод возвращает значение `False`). Метод можно переопределить в классе, наследующем класс `QWizardPage`, и реализовать собственную проверку правильности ввода данных. При изменении возвращаемого значения необходимо генерировать сигнал `completeChanged()`;
- `nextId(self)` — этот метод следует переопределить в классе, наследующем `QWizardPage`, если необходимо изменить порядок отображения страниц. Метод вызывается при нажатии кнопки **Next**. Метод должен возвращать идентификатор следующей страницы или значение `-1`, если текущая страница — последняя;
- `initializePage(self)` — этот метод следует переопределить в классе, наследующем `QWizardPage`, если необходимо производить настройку свойств компонентов на текущей странице на основе данных, введенных на предыдущих страницах. Метод вызывается при нажатии кнопки **Next** на предыдущей странице, но до отображения текущей страницы. Если установлена опция `IndependentPages`, метод вызывается только при первом отображении страницы;
- `cleanupPage(self)` — этот метод следует переопределить в классе, наследующем `QWizardPage`, если необходимо выполнять какие-либо завершающие действия на текущей странице перед переходом на предыдущую при нажатии кнопки **Back**. Метод вызывается при нажатии кнопки **Back** на текущей странице, но до отображения предыдущей страницы. Если установлена опция `IndependentPages`, метод не вызывается;
- `validatePage(self)` — этот метод следует переопределить в классе, наследующем `QWizardPage`, если необходимо производить проверку данных, введенных на текущей странице. Метод вызывается при нажатии кнопки **Next** или **Finish**. Метод должен вернуть значение `True`, если данные корректны, и `False` — в противном случае. Если метод возвращает значение `False`, переход на следующую страницу не производится.

ГЛАВА 28

Создание SDI- и MDI-программ

- *SDI-программа* (Single Document Interface, однодокументный интерфейс) — может открыть лишь один документ. Типичные примеры: Блокнот, WordPad и Paint, поставляемые в составе Windows.
- *MDI-программа* (Multiple Document Interface, многодокументный интерфейс) — позволяет открыть произвольное количество документов, каждый — в отдельном *вложенном окне*. Примером приложения такого рода является программа Adobe Photoshop.

Все рассмотренные в этой главе классы объявлены в модуле `QtWidgets`, если не указано иное.

28.1. Главное окно программы

И SDI-, и MDI-программы имеют одно *главное окно*, в котором отображается один открытый документ или вложенные окна с открытыми документами. Главное окно содержит меню, панели инструментов, прикрепляемые панели, центральный компонент и строку состояния.

Главное окно реализуется классом `QMainWindow`. Иерархия наследования для него выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMainWindow
```

Конструктор класса `QMainWindow` имеет следующий формат:

```
QMainWindow([parent=None][, flags=0])
```

В параметре `parent` указывается ссылка на родительское окно. Доступные значения параметра `flags` мы рассматривали в *разд. 19.1.1*.

Чтобы создать SDI-программу, следует с помощью метода `setCentralWidget()` класса `QMainWindow` установить в главном окне какой-либо центральный компонент, отображающий содержимое открытого документа.

Класс `QMainWindow` наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только интересующие нас — полный их список можно найти на странице **https://doc.qt.io/qt-6/qmainwindow.html**):

- `setCentralWidget(<Компонент>)` — делает указанный компонент центральным в главном окне;

- `centralWidget()` — возвращает ссылку на центральный компонент или значение `None`, если таковой не был установлен;
- `setMenuBar(<Меню QMenuBar>)` — устанавливает в главном окне заданное главное меню вместо стандартного, формируемого самим PyQt;
- `menuBar()` — возвращает ссылку на главное меню (объект класса `QMenuBar`);
- `setMenuWidget(<Меню QWidget>)` — устанавливает в главном окне заданное главное меню вместо стандартного. Используется, если в главном окне требуется создать какое-либо специфическое меню (например, не обычное горизонтальное, а вертикальное);
- `menuWidget()` — возвращает ссылку на главное меню (объект класса `QWidget` или производного от него);
- `createPopupMenu()` — создает контекстное меню с пунктами, позволяющими отобразить или скрыть панели инструментов и прикрепляемые панели, и возвращает ссылку на это меню (объект класса `QMenu`). Контекстное меню по умолчанию отображается при щелчке правой кнопкой мыши в области меню, панели инструментов или прикрепляемых панелей. Переопределив этот метод, можно реализовать собственное контекстное меню;
- `setStatusBar(<Строка состояния QStatusBar>)` — устанавливает в главном окне заданную строку состояния вместо стандартной;
- `statusBar()` — возвращает ссылку на строку состояния (объект класса `QStatusBar`);
- `addToolBar()` — добавляет панель инструментов. Форматы метода:

```
addToolBar(<Панель инструментов QToolBar>)
addToolBar(<Область>, <Панель инструментов QToolBar>)
addToolBar(<Название панели>)
```

  Первый формат добавляет панель инструментов в верхнюю часть окна. Второй формат позволяет задать местоположение панели. В качестве параметра `<Область>` могут быть указаны следующие элементы перечисления `ToolBarArea` из модуля `QtCore.Qt`: `LeftToolBarArea` (слева), `RightToolBarArea` (справа), `TopToolBarArea` (сверху) или `BottomToolBarArea` (снизу). Третий формат создает панель инструментов с указанным именем, добавляет ее в верхнюю область окна и возвращает ссылку на представляющий ее объект класса `QToolBar`;
- `insertToolBar(<Панель QToolBar 1>, <Панель QToolBar 2>)` — добавляет панель `<QToolBar 2>` перед панелью `<QToolBar 1>`;
- `removeToolBar(<Панель инструментов QToolBar>)` — удаляет панель инструментов из окна и скрывает ее. При этом объект панели инструментов не удаляется и далее может быть добавлен в другое место;
- `toolBarArea(<Панель инструментов QToolBar>)` — возвращает местоположение указанной панели инструментов в виде элементов `LeftToolBarArea` (слева), `RightToolBarArea` (справа), `TopToolBarArea` (сверху), `BottomToolBarArea` (снизу) или `NoToolBarArea` (положение не определено) перечисления `ToolBarArea` из модуля `QtCore.Qt`;
- `setToolButtonStyle(<Стиль>)` — задает стиль кнопок на панелях инструментов. В качестве параметра указываются следующие элементы перечисления `ToolButtonStyle` из модуля `QtCore.Qt`:
  - `ToolButtonIconOnly` — только значок;
  - `ToolButtonTextOnly` — только текст;

- `ToolButtonTextBesideIcon` — текст справа от значка;
- `ToolButtonTextUnderIcon` — текст под значком;
- `ToolButtonFollowStyle` — зависит от используемого стиля;

- `toolButtonStyle()` — возвращает стиль кнопок на панелях инструментов;
- `setIconSize(<Размеры QSize>)` — задает размеры значков на кнопках в панелях инструментов;
- `iconSize()` — возвращает размеры значков (объект класса `QSize`) на кнопках в панелях инструментов;
- `setAnimated(<Флаг>)` — если в качестве параметра указано значение `False`, вставка панелей инструментов и прикрепляемых панелей в новое место по окончании перемещения будет производиться без анимации. Метод является слотом;
- `addToolBarBreak([area=ToolBarArea.TopToolBarArea])` — вставляет разрыв в указанное параметром `area` место после всех добавленных ранее панелей. По умолчанию панели добавляются друг за другом на одной строке. С помощью этого метода можно поместить панели инструментов на двух и более строках;
- `insertToolBarBreak(<Панель инструментов QToolBar>)` — вставляет разрыв перед указанной панелью инструментов;
- `removeToolBarBreak(<Панель инструментов QToolBar>)` — удаляет разрыв перед указанной панелью инструментов;
- `toolBarBreak(<Панель инструментов QToolBar>)` — возвращает значение `True`, если перед указанной панелью инструментов существует разрыв, и `False` — в противном случае;
- `addDockWidget()` — добавляет прикрепляемую панель. Форматы метода:

```
addDockWidget(<Область>, <Панель QDockWidget>)
addDockWidget(<Область>, <Панель QDockWidget>, <Ориентация>)
```

  Первый формат добавляет прикрепляемую панель в указанную область окна. В качестве параметра `<Область>` могут быть указаны следующие элементы перечисления `DockWidgetArea` из модуля `QtCore.Qt`: `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху) или `BottomDockWidgetArea` (снизу). Второй формат позволяет дополнительно указать ориентацию в виде следующих элементов перечисления `Orientation` из модуля `QtCore.Qt`: `Horizontal` или `Vertical`. Если указан элемент `Horizontal`, добавляемая панель будет расположена справа от ранее добавленной панели, а если `Vertical` — снизу;

- `removeDockWidget(<Панель QDockWidget>)` — удаляет заданную прикрепляемую панель из окна и скрывает ее. При этом объект панели не удаляется и впоследствии может быть добавлен в другую область;
- `dockWidgetArea(<Панель QDockWidget>)` — возвращает местоположение указанной прикрепляемой панели в виде элементов `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху), `BottomDockWidgetArea` (снизу) или `NoDockWidgetArea` (положение не определено) перечисления `DockWidgetArea` из модуля `QtCore.Qt`;
- `setDockOptions(<Опции>)` — устанавливает опции для прикрепляемых панелей. Значение по умолчанию: `AnimatedDocks | AllowTabbedDocks`. В качестве значения указывается комбинация (через оператор `|`) следующих элементов перечисления `DockOption` из класса `QMainWindow`:

- AnimatedDocks — вставка панелей в новое место по окончании перемещения будет производиться с анимацией;
- AllowNestedDocks — в любую область можно будет поместить несколько панелей;
- AllowTabbedDocks — панели накладываются одна на другую. Для переключения между накладывающимися панелями будет использоваться панель с вкладками;
- ForceTabbedDocks — панели всегда накладываются друг на друга. При этом опция AllowNestedDocks игнорируется;
- VerticalTabs — заголовки вкладок у накладывающихся панелей будут отображаться с внешнего края области (если область справа, то и заголовки вкладок справа, если область слева, то и заголовки вкладок слева, если область сверху, то и заголовки вкладок сверху, если область снизу, то и заголовки вкладок снизу). По умолчанию заголовки вкладок отображаются снизу. Опция AllowTabbedDocks должна быть установлена;
- GroupedDragging — при перетаскивании любой панели все остальные панели, наложенные на нее, будут перетаскиваться вместе с ней. Опция AllowTabbedDocks должна быть установлена;

> ***Внимание!***
>
> Опции необходимо устанавливать до добавления прикрепляемых панелей. Исключением являются опции AnimatedDocks и VerticalTabs.

- dockOptions() — возвращает комбинацию установленных опций прикрепляемых панелей;
- setDockNestingEnabled(<Флаг>) — если указано значение True, опция AllowNestedDocks будет установлена, а если указано значение False — сброшена. Метод является слотом;
- isDockNestingEnabled() — возвращает значение True, если опция AllowNestedDocks установлена, и False — в противном случае;
- setTabPosition(<Область DockWidgetArea>, <Позиция>) — задает позицию отображения заголовков вкладок для указанной области. По умолчанию заголовки вкладок отображаются снизу. В качестве параметра <Позиция> могут быть указаны следующие элементы перечисления TabPosition из класса QTabWidget: North (сверху), South (снизу), West (слева) и East (справа);
- tabPosition(<Область DockWidgetArea>) — возвращает позицию отображения заголовков вкладок прикрепляемых панелей для указанной области;
- setTabShape(<Форма>) — задает форму углов ярлыков вкладок в виде одного из следующих элементов перечисления TabShape из класса QTabWidget: Rounded (скругленные углы — значение по умолчанию) или Triangular (треугольная форма);
- tabShape() — возвращает форму углов ярлыков вкладок;
- setCorner(<Угол>, <Область DockWidgetArea>) — закрепляет указанный угол за заданной областью. По умолчанию верхние углы закреплены за верхней областью, а нижние — за нижней областью. В параметре <Угол> указываются следующие элементы перечисления Corner из модуля QtCore.Qt:
  - TopLeftCorner — левый верхний угол;
  - TopRightCorner — правый верхний угол;

- `BottomLeftCorner` — левый нижний угол;
- `BottomRightCorner` — правый нижний угол;

- `corner(<Угол Corner>)` — возвращает область, за которой закреплен указанный угол;
- `splitDockWidget()` — разделяет область, занимаемую панелью `<QDockWidget 1>`, помещает панель `<QDockWidget 1>` в первую часть области, а панель `<QDockWidget 2>` — во вторую. Формат метода:

  ```
  splitDockWidget(<Панель QDockWidget 1>, <Панель QDockWidget 2>, <Ориентация>)
  ```

  В качестве параметра `<Ориентация>` могут быть указаны следующие элементы перечисления `Orientation` из модуля `QtCore.Qt`: `Horizontal` или `Vertical`. Если указан элемент `Horizontal`, то панель `<QDockWidget 2>` будет расположена справа, а если `Vertical` — то снизу. Если панель `<QDockWidget 1>` расположена на вкладке, панель `<QDockWidget 2>` будет добавлена на новую вкладку, и разделения области при этом не произойдет;
- `tabifyDockWidget(<Панель QDockWidget 1>, <Панель QDockWidget 2>)` — размещает панель `<QdockWidget 2>` над панелью `<QDockWidget 1>`, создавая таким образом область с вкладками;
- `tabifiedDockWidgets(<Панель QDockWidget>)` — возвращает список ссылок на панели (объекты класса `QDockWidget`), которые расположены на других вкладках в области панели, указанной в качестве параметра;
- `resizeDocks()` — задает новые размеры у панелей из списка, указанного в первом параметре, при этом первый размер из указанных в `<Списке размеров>` будет применен к первой панели, второй — ко второй панели и т. д. Формат метода:

  ```
  resizeDocks(<Список панелей QDockWidget>, <Список размеров>, <Ориентация>)
  ```

  Если третьим параметром передан элемент `Horizontal` перечисления `Orientation` из модуля `QtCore.Qt`, `<Список размеров>` задаст значения ширины панелей, если указан элемент `Vertical` — высоты. Размеры панелей, не указанных в первом списке, будут изменены таким образом, чтобы вместиться в оставшееся свободное пространство. Размеры самого окна при этом не изменяются;
- `saveState([version=0])` — возвращает объект класса `QByteArray` с размерами и положением всех панелей инструментов и прикрепляемых панелей.

  Обратите внимание, что все панели инструментов и прикрепляемые панели должны иметь уникальные объектные имена. Задать объектное имя можно с помощью метода `setObjectName(<Имя в виде строки>)` класса `QObject`, например:

  ```
  self.dw = QtWidgets.QDockWidget("MyDockWidget1")
  self.dw.setObjectName("MyDockWidget1")
  ```

- `restoreState(<Объект QByteArray>[, version=0])` — восстанавливает размеры и местоположение всех панелей инструментов и прикрепляемых панелей, ранее сохраненные вызовом метода `saveState()`.

  Чтобы сохранить размеры окна, следует воспользоваться методом `saveGeometry()` класса `QWidget`. Метод возвращает объект класса `QByteArray` с размерами окна. Чтобы восстановить размеры окна, достаточно вызвать метод `restoreGeometry(<Объект QByteArray>)`;
- `restoreDockWidget(<Панель QDockWidget>)` — восстанавливает состояние указанной панели, если она была создана после вызова метода `restoreState()`. Метод возвращает значение `True`, если состояние панели успешно восстановлено, и `False` — в противном случае.

Класс QMainWindow поддерживает полезный сигнал tabifiedDockWidgetActivated(<Панель QDockWidget>), генерируемый при активизации вкладки щелчком на ее заголовке. Активизированная вкладка передается обработчику с единственным параметром. Остальные, менее полезные сигналы описаны на странице **https://doc.qt.io/qt-6/qmainwindow.html**.

28.2. Меню и действия

Главное меню позволяет компактно поместить в главное окно множество команд, объединенных в логические группы. Оно представляет собой горизонтальную панель (реализуемую классом QMenuBar), на которой расположены отдельные меню (реализуются с помощью класса QMenu) верхнего уровня.

Каждое меню содержит произвольное количество *действий* (класс QAction), каждое из которых, в зависимости от его типа, представляет отдельный пункт, разделитель или вложенное меню. Действие, представляющее пункт, содержит текстовое название, необязательные значок и флажок, превращающий его в переключатель. Действие, представляющее вложенное меню, содержит название, собственно меню и необязательный значок. Действие-разделитель ничего не содержит.

Контекстное меню обычно отображается при щелчке правой кнопкой мыши в области какого-либо компонента. Оно также реализуется классом QMenu.

28.2.1. Класс *QMenuBar*

Класс QMenuBar описывает само главное меню. Оно реализовано в главном окне программы изначально. Получить ссылку на него можно вызовом метода menuBar() класса QMainWindow. Установить свое главное меню позволяет метод setMenuBar(<Главное меню QMenuBar>). Иерархия наследования класса QMenuBar такова:

```
(QObject, QPaintDevice) — QWidget — QMenuBar
```

Конструктор класса QMenuBar имеет следующий формат:

```
QMenuBar([parent=None])
```

В параметре parent указывается ссылка на родительское окно.

Класс QMenuBar наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только интересующие нас — полный их список можно найти на странице **https://doc.qt.io/qt-6/qmenubar.html**):

- addMenu(<Меню QMenu>) — добавляет заданное меню в текущее главное меню и возвращает связанное с добавленным меню действие (объект класса QAction);
- addMenu([<Значок QIcon>,]<Название>) — создает меню, добавляет его в главное меню и возвращает ссылку на созданное меню (объект класса QMenu). Внутри текста названия символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В первом параметре можно указать значок, который будет выведен левее названия меню;
- insertMenu(<Действие QAction>, <Меню QMenu>) — добавляет заданное меню перед указанным действием. Метод возвращает действие (объект класса QAction), соответствующее добавленному меню;
- addAction() — добавляет в главное меню действие, непосредственно заданное или созданное на основе заданных названия и обработчика. Форматы метода:

```
addAction(<Действие QAction>)
addAction(<Название>)                    -> QAction
addAction(<Название>, <Обработчик>)      -> QAction
```

Второй и третий форматы возвращают созданное действие;

- `addSeparator()` — добавляет в главное меню разделитель и возвращает представляющее его действие;
- `insertSeparator(<Действие QAction>)` — добавляет перед указанным действием разделитель и возвращает связанное с ним действие;
- `clear()` — удаляет все действия из главного меню;
- `setActiveAction(<Действие QAction>)` — делает активным указанное действие;
- `activeAction()` — возвращает активное действие или значение `None`;
- `setDefaultUp(<Флаг>)` — если в качестве параметра указано значение `True`, отдельные меню будут открываться над главным меню, а не под ним;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `False`, главное меню будет скрыто. Значение `True` отображает панель меню.

Класс `QMenuBar` поддерживает следующие сигналы:

- `hovered(<Действие QAction>)` — генерируется при наведении курсора мыши на какое-либо действие. Последнее передается обработчику с параметром;
- `triggered(<Действие QAction>)` — генерируется при выборе какого-либо действия. Последнее передается обработчику с параметром.

28.2.2. Класс *QMenu*

Класс `QMenu` реализует отдельное меню в главном меню, а также вложенное и контекстное меню. Его иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMenu
```

Форматы конструктора класса `QMenu`:

```
QMenu([parent=None])
QMenu(<Название>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент. Внутри текста в параметре `<Название>` символ `&`, указанный перед буквой или цифрой, задает клавишу быстрого доступа. Чтобы вывести сам символ `&`, необходимо его удвоить.

Помимо унаследованных из базовых классов, класс `QMenu` поддерживает ряд своих методов (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qmenu.html**):

- `addAction()` — добавляет в меню действие, непосредственно заданное или созданное на основе заданных параметров. Форматы метода:

```
addAction(<Действие QAction>)
addAction(<Название>)                                          -> QAction
addAction(<Значок QIcon>, <Название>)                          -> QAction
addAction(<Название>, <Обработчик>[, shortcut=0])              -> QAction
addAction(<Значок QIcon>, <Название>, <Обработчик>[, shortcut=0]) -> QAction
```

Внутри текста названия символ &, указанный перед буквой или цифрой, задает клавишу быстрого доступа. Эта клавиша сработает только в том случае, если меню, в котором находится пункт, является активным.

Параметр shortcut задает комбинацию «горячих» клавиш, нажатие которых аналогично выбору пункта в меню. Нажатие комбинации «горячих» клавиш сработает даже в том случае, если меню не является активным. Примеры:

```
QtGui.QKeySequence("Ctrl+R")
QtGui.QKeySequence(QtCore.Qt.Key.Key_F5))
QtGui.QKeySequence.fromString("Ctrl+R")
```

Все форматы, за исключением первого, возвращают созданное действие;

- addSeparator() — добавляет в меню действие-разделитель и возвращает ссылку на него;
- addSection() — добавляет в меню действие-*секцию*, которая выглядит как разделитель с заголовком и, возможно, значком, и возвращает ссылку на созданное действие. Форматы метода:

```
addSection(<Заголовок>)
addSection(<Заголовок>, <Значок QIcon>)
```

- addMenu(<Меню QMenu>) — добавляет заданное меню в качестве вложенного и возвращает представляющее его действие (объект класса QAction);
- addMenu([<Значок QIcon>,]<Название>) — создает вложенное меню, добавляет его в текущее меню и возвращает ссылку на созданное меню (объект класса QMenu);
- insertMenu(<Действие QAction>, <Меню QMenu>) — добавляет заданное меню в качестве вложенного перед указанным действием. Метод возвращает действие, соответствующее добавленному меню;
- insertSeparator(<Действие QAction>) — добавляет действие-разделитель перед указанным действием и возвращает ссылку на добавленное действие;
- insertSection() — добавляет действие-секцию перед указанным действием и возвращает ссылку на добавленное действие. Форматы метода:

```
insertSection(<Действие QAction>, <Заголовок>)
insertSection(<Действие QAction>, <Заголовок>, <Значок QIcon>)
```

- clear() — удаляет все действия из меню;
- isEmpty() — возвращает значение True, если меню не содержит видимых пунктов, и False — в противном случае;
- menuAction() — возвращает действие, связанное с текущим меню;
- setTitle(<Название>) — задает название меню;
- title() — возвращает название меню;
- setIcon(<Значок QIcon>) — задает значок меню;
- icon() — возвращает значок меню (объект класса QIcon);
- setActiveAction(<Действие QAction>) — делает активным пункт, соответствующий указанному действию;
- activeAction() — возвращает активный пункт (объект класса QAction) или значение None;
- setDefaultAction(<Действие QAction>) — задает пункт по умолчанию;

- `defaultAction()` — возвращает пункт по умолчанию (объект класса `QAction`) или значение `None`;
- `setTearOffEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, в начало меню добавляется пункт с пунктирной линией, с помощью нажатия на который можно оторвать меню от главного меню и сделать его плавающим (отображаемым в отдельном окне, которое можно разместить в любой части экрана);
- `isTearOffEnabled()` — возвращает значение `True`, если меню может быть плавающим, и `False` — в противном случае;
- `isTearOffMenuVisible()` — возвращает значение `True`, если меню является плавающим, и `False` — в противном случае;
- `showTearOffMenu([<Координаты QPoint>])` — принудительно превращает меню, для которого была указана такая возможность, в плавающее и помещает его либо по заданным координатам, либо, если параметр не указан, в точке, где расположен курсор мыши;
- `hideTearOffMenu()` — скрывает плавающее меню;
- `setSeparatorsCollapsible(<Флаг>)` — если в качестве параметра указано значение `True`, вместо нескольких разделителей, идущих подряд, будет отображаться один. Кроме того, разделители, расположенные по краям меню, также будут скрыты;
- `setToolTipsVisible(<Флаг>)` — если передано значение `True`, заданная для меню всплывающая подсказка будет отображаться на экране, если `False` — не будет (поведение по умолчанию). Текст всплывающей подсказки у меню можно задать вызовом метода `setToolTip(<Текст>)`, унаследованным от класса `QWidget`;
- `popup(<Координаты QPoint>[, <Действие QAction>])` — отображает меню по указанным глобальным координатам. Если указан второй параметр, меню отображается таким образом, чтобы по координатам был расположен указанный пункт меню;
- `exec([<Координаты QPoint>][, ][action=None])` — отображает меню по указанным глобальным координатам и возвращает действие, соответствующее выбранному пункту, или значение `None`, если ни один пункт не был выбран (например, была нажата клавиша <Esc>). Если в параметре `action` указано действие (объект класса `QAction`), меню отображается таким образом, чтобы по координатам был расположен соответствующий пункт;
- `exec()` — статический, выводит меню, содержащее действия из указанного списка, по заданным глобальным координатам и возвращает действие, соответствующее выбранному пункту, или значение `None`, если пункт не выбран (например, была нажата клавиша <Esc>). Формат метода:

```
exec(<Список с действиями QAction>, <Координаты QPoint>[, at=None][,
                                                          parent=None])
```

  Если в параметре `at` указано действие (объект класса `QAction`), меню отображается таким образом, чтобы по координатам был расположен соответствующий пункт меню. В параметре `parent` можно указать ссылку на родительский компонент.

Класс `QMenu` поддерживает следующие сигналы:

- `hovered(<Действие QAction>)` — генерируется при наведении курсора мыши на какое-либо действие, которое передается в обработчик в параметре;
- `triggered(<Действие QAction>)` — генерируется при выборе в меню какого-либо действия, которое передается в обработчик в параметре;

- aboutToShow() — генерируется перед отображением меню;
- aboutToHide() — генерируется перед скрытием меню.

28.2.3. Контекстное меню компонента

Чтобы создать у компонента контекстное меню, можно воспользоваться следующими методами класса QWidget:

- addAction(<Действие QAction>) — добавляет указанное действие в конец контекстного меню;
- addActions(<Список с действиями QAction>) — добавляет действия из указанного списка в конец контекстного меню;
- insertAction(<Действие QAction 1>, <Действие QAction 2>) — добавляет <Действие 2> перед <Действием 1>;
- insertActions(<Действие QAction>, <Список с действиями QAction>) — добавляет действия из указанного списка перед заданным действием;
- actions() — возвращает список с действиями (объекты класса QAction), содержащимися в контекстном меню;
- removeAction(<Действие QAction>) — удаляет указанное действие из меню.

Перед добавлением действий в контекстное меню компонента с помощью методов addAction() и addActions() следует вызвать у компонента метод setContectMenuPolicy(<Режим>), задающий режим формирования и вывода контекстного меню, передав ему в качестве параметра элемент ActionsContextMenu перечисления ContextMenuPolicy из модуля QtCore.Qt. Таким образом компоненту будет указано формировать пункты контекстного меню на основе заданных действий. Пример:

```
class MyComponent(QWidget):
    def __init__(self):
        . . .
        self.setContextMenuPolicy(
                            QtCore.Qt.ContextMenuPolicy.ActionsContextMenu)
        # Создаем действия action1, action2 и action3
        self.addAction(self.action1)
        self.addAction(self.action2)
        self.addAction(self.action3)
        . . .
```

Созданное таким образом контекстное меню будет выводиться автоматически при щелчке на компоненте правой кнопкой мыши.

Другой способ вывести контекстное меню — переопределение у компонента метода contextMenuEvent(self, <event>). Этот метод будет автоматически вызываться при щелчке правой кнопкой мыши в области компонента. Внутри метода через параметр <event> доступен объект класса QContextMenuEvent, содержащий дополнительную информацию о событии. Класс QContextMenuEvent поддерживает следующие основные методы:

- x() и y() — возвращают координаты по осям X и Y соответственно в пределах области компонента;
- pos() — возвращает объект класса QPoint с целочисленными координатами в пределах области компонента;

- `globalX()` и `globalY()` — возвращают координаты по осям `X` и `Y` соответственно в пределах экрана;
- `globalPos()` — возвращает объект класса `QPoint` с координатами в пределах экрана.

В методе следует явно создать меню, добавить в него необходимые действия и вывести вызовом метода `exec()` с передачей ему результата выполнения метода `globalPos()`:

```
class MyComponent(QWidget):
    . . .
    def contextMenuEvent(self, event):
        self.context_menu = QtWidgets.QMenu(parent=self)
        self.context_menu.addAction(self.action1)
        self.context_menu.addAction(self.action2)
        self.context_menu.addAction(self.action3)
        self.context_menu.exec(event.globalPos())
```

Контекстное меню можно создавать и заранее (например, в теле конструктора).

28.2.4. Класс *QAction*

Класс `QAction` из модуля `QtGui` описывает действие. Добавленное в меню, оно преобразуется в пункт, добавленное на панель инструментов — в кнопку. Один и тот же объект действия допускается добавлять в несколько мест (например, и в меню, и на панель инструментов), что позволяет управлять видимостью и доступностью действия централизованно. Иерархия наследования для класса `QAction` следующая:

```
QObject — QAction
```

Форматы конструктора класса `QAction`:

```
QAction([parent=None])
QAction(<Название>[, parent=None])
QAction(<Значок QIcon>, <Название>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент или значение `None`. Внутри текста названия символ `&`, указанный перед буквой или цифрой, задает клавишу быстрого доступа. Чтобы вывести сам символ `&`, необходимо его удвоить.

Класс `QAction` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qaction.html**):

- `setText(<Название>)` — задает название действия. Внутри текста названия символ `&`, указанный перед буквой или цифрой, задает клавишу быстрого доступа. Нажатие клавиши быстрого доступа сработает только в том случае, если меню, в котором находится пункт, является активным;
- `text()` — возвращает название действия;
- `setIcon(<Значок QIcon>)` — устанавливает значок;
- `icon()` — возвращает значок (объект класса `QIcon`);
- `setIconVisibleInMenu(<Флаг>)` — если в качестве параметра указано значение `False`, значок отображаться не будет;
- `setSeparator(<Флаг>)` — если в качестве параметра указано значение `True`, действие станет разделителем;

- `isSeparator()` — возвращает значение `True`, если действие является разделителем, и `False` — в противном случае;
- `setShortcut(<Объект QKeySequence>)` — задает комбинацию «горячих» клавиш. Нажатие комбинации «горячих» клавиш по умолчанию сработает даже в том случае, если меню не является активным. Вот примеры указания значения:

```
"Ctrl+O"
QtGui.QKeySequence("Ctrl+O")
QtGui.QKeySequence.fromString("Ctrl+O")
QtGui.QKeySequence.Open
```

- `setShortcuts()` — позволяет задать сразу несколько комбинаций «горячих» клавиш. Форматы метода:

```
setShortcuts(<Список с объектами QKeySequence>)
setShortcuts(<Стандартная комбинация клавиш>)
```

  В параметре `<Стандартная комбинация клавиш>` указываются элементы перечисления `StandardKey` из класса `QKeySequence` (например, `Open`, `Close`, `Copy`, `Cut` и т. д. — полный их список можно найти в документации по классу `QKeySequence`);
- `setShortcutContext(<Область>)` — задает область действия комбинации «горячих» клавиш. В качестве параметра указываются следующие элементы перечисления `ShortcutContext` из модуля `QtCore.Qt`:
  - `WidgetShortcut` — комбинация доступна, если родительский компонент имеет фокус;
  - `WidgetWithChildrenShortcut` — комбинация доступна, если фокус имеет родительский компонент или любой дочерний компонент;
  - `WindowShortcut` — комбинация доступна, если окно, в котором расположен компонент, является активным;
  - `ApplicationShortcut` — комбинация доступна, если любое окно программы является активным;
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `toolTip()` — возвращает текст всплывающей подсказки;
- `setWhatsThis(<Текст>)` — задает текст расширенной подсказки;
- `whatsThis()` — возвращает текст расширенной подсказки;
- `setStatusTip(<Текст>)` — задает текст, который будет отображаться в строке состояния при наведении курсора мыши на действие;
- `statusTip()` — возвращает текст для строки состояния;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, действие является переключателем, который может находиться в двух состояниях: установленном и неустановленном;
- `isCheckable()` — возвращает значение `True`, если действие является переключателем, и `False` — в противном случае;
- `setChecked(<Флаг>)` — если в качестве параметра указано значение `True`, действие-переключатель будет находиться в установленном состоянии. Метод является слотом;
- `isChecked()` — возвращает значение `True`, если действие-переключатель находится в установленном состоянии, и `False` — в противном случае;

- `setDisabled(<Флаг>)` — если в качестве параметра указано значение `True`, действие станет недоступным. Значение `False` делает действие вновь доступным. Метод является слотом;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `False`, действие станет недоступным. Значение `True` делает действие вновь доступным. Метод является слотом;
- `isEnabled()` — возвращает значение `True`, если действие доступно, и `False` — в противном случае;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `False`, действие будет скрыто. Значение `True` вновь выводит действие на экран. Метод является слотом;
- `isVisible()` — возвращает значение `False`, если действие скрыто, и `True` — в противном случае;
- `setFont(<Шрифт QFont>)` — устанавливает шрифт для текста;
- `font()` — возвращает объект класса `QFont` с текущими параметрами шрифта;
- `setAutoRepeat(<Флаг>)` — если в качестве параметра указано значение `True` (значение по умолчанию), действие будет повторяться, пока удерживается нажатой комбинация «горячих» клавиш;
- `setPriority(<Приоритет>)` — задает приоритет действия. В качестве параметра указываются элементы `LowPriority` (низкий), `NormalPriority` (нормальный) или `HighPriority` (высокий) перечисления `Priority` из класса `QAction`;
- `priority()` — возвращает текущий приоритет действия;
- `setData(<Данные>)` — позволяет сохранить пользовательские данные любого типа в объекте действия;
- `data()` — возвращает пользовательские данные, сохраненные ранее с помощью метода `setData()`, или значение `None`;
- `setActionGroup(<Группа QActionGroup>)` — добавляет действие в указанную группу;
- `actionGroup()` — возвращает ссылку на группу (объект класса `QActionGroup`) или значение `None`;
- `showStatusText([None])` — отправляет событие типа `Type.StatusTip` указанному в параметре компоненту и возвращает значение `True`, если событие успешно отправлено. Если компонент не указан, событие посылается родителю действия.

  Чтобы обработать событие, необходимо наследовать класс компонента и переопределить метод `event(self, <event>)`. При событии `Type.StatusTip` через параметр `<event>` доступен объект класса `QStatusTipEvent`. Получить текст для строки состояния можно через метод `tip()` объекта события. Вот пример обработки события в классе, наследующем класс `QLabel`:

```
def event(self, e):
    if e.type() == QtCore.QEvent.Type.StatusTip:
        self.setText(e.tip())
        return True
    return QtWidgets.QLabel.event(self, e)
```

- `hover()` — посылает сигнал `hovered()`. Метод является слотом;

- `toggle()` — изменяет состояние действия-переключателя на противоположное. Метод является слотом;
- `trigger()` — посылает сигнал `triggered()`. Метод является слотом.

Класс `QAction` поддерживает следующие сигналы:

- `changed()` — генерируется при изменении действия;
- `hovered()` — генерируется при наведении курсора мыши на действие;
- `toggled(<Состояние>)` — генерируется при изменении состояния действия-переключателя. Внутри обработчика через параметр доступно текущее состояние в виде логической величины;
- `triggered(<Состояние переключателя>)` — генерируется при выборе пункта меню, нажатии кнопки на панели инструментов, нажатии комбинации клавиш или вызове метода `trigger()`. В параметре доступно текущее состояние действия-переключателя в виде логической величины — у обычных действий значение параметра всегда равно `False`.

Код, который должен выполняться при выборе действия, обычно помещается в обработчик сигнала `triggered()` или `toggled()`.

28.2.5. Объединение действий-переключателей в группу

Класс `QActionGroup` представляет группу, объединяющую произвольное количество действий-переключателей. По умолчанию внутри группы может быть установлен только один переключатель — при попытке установить другой переключатель ранее установленный будет сброшен. Иерархия наследования у этого класса выглядит так:

```
QObject — QActionGroup
```

Конструктор класса `QActionGroup` имеет следующий формат:

```
QActionGroup(<Родитель>)
```

В параметре `<Родитель>` указывается ссылка на родительский компонент. После создания объекта группы он может быть указан в качестве родителя при создании объектов действия — в этом случае действия автоматически добавятся в группу.

Класс `QActionGroup` поддерживает следующие основные методы:

- `addAction()` — добавляет в группу действие, указанное непосредственно или вновь созданное на основе заданных названия и значка. Метод возвращает добавленное действие. Форматы метода:

```
addAction(<Действие QAction>)
addAction(<Название>)
addAction(<Значок QIcon>, <Название>)
```

- `removeAction(<Действие QAction>)` — удаляет указанное действие из группы;
- `actions()` — возвращает список с действиями (объектами класса `QAction`), которые были добавлены в группу, или пустой список;
- `checkedAction()` — возвращает ссылку (объект класса `QAction`) на установленное действие-переключатель внутри группы при использовании эксклюзивного режима или значение `None`;

- `setExclusive(<Флаг>)` — если в качестве параметра указано значение `False`, внутри группы может быть установлено произвольное количество действий-переключателей. Значение `True` возвращает эксклюзивный режим, при котором может быть установлено только одно действие-переключатель (поведение по умолчанию). Метод является слотом;
- `setExclusionPolicy(<Режим>)` — задает режим работы группы. В качестве параметра указывается один из следующих элементов перечисления `ExclusionPolicy` из класса `QActionGroup`:
  - `None` — может быть установлено произвольное количество действий-переключателей (аналогично вызову метода `setExclusive()` с передачей ему значения `False`);
  - `Exclusive` — может быть установлено лишь одно действие-переключатель (поведение по умолчанию);
  - `ExclusiveOptional` — может быть установлено лишь одно действие-переключатель или ни одного;
- `setDisabled(<Флаг>)` — если в качестве параметра указано значение `True`, все действия в группе станут недоступными. Значение `False` делает действия вновь доступными. Метод является слотом;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `False`, все действия в группе станут недоступными. Значение `True` делает действия вновь доступными. Метод является слотом;
- `isEnabled()` — возвращает значение `True`, если действия в группе доступны, и `False` — в противном случае;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `False`, все действия в группе будут скрыты. Значение `True` вновь выводит действия на экран. Метод является слотом;
- `isVisible()` — возвращает значение `False`, если действия группы скрыты, и `True` — в противном случае.

Класс `QActionGroup` поддерживает сигналы:

- `hovered(<Действие QAction>)` — генерируется при наведении курсора мыши на действие внутри группы. Внутри обработчика через параметр доступна ссылка на это действие;
- `triggered(<Действие QAction>)` — генерируется при выборе пункта меню, нажатии кнопки на панели инструментов или комбинации клавиш. Внутри обработчика через параметр доступна ссылка на действие.

28.3. Панели инструментов

Панели инструментов предназначены для отображения часто используемых команд. Добавить панель инструментов в главное окно позволяют методы `addToolBar()` и `insertToolBar()` класса `QMainWindow`. Один из форматов метода `addToolBar()` позволяет указать область, к которой будет изначально прикреплена панель. По умолчанию с помощью мыши пользователь может переместить панель в другую область окна или отобразить в отдельном окне. Ограничить перечень областей, к которым можно прикрепить панель, позволяет метод `setAllowedAreas()` класса `QToolBar`, а запретить отображение панели в отдельном окне — метод `setFloatable()`.

28.3.1. Класс *QToolBar*

Класс `QToolBar` реализует панель инструментов, которую можно перемещать с помощью мыши. Иерархия наследования для него такова:

```
(QObject, QPaintDevice) — QWidget — QToolBar
```

Форматы конструктора класса `QToolBar`:

```
QToolBar([parent=None])
QToolBar(<Название>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент, а в параметре `<Название>` — название панели, которое отображается в контекстном меню при щелчке правой кнопкой мыши в области меню, области панелей инструментов или на заголовке прикрепляемых панелей. С помощью контекстного меню можно скрыть или отобразить панель инструментов.

Класс `QToolBar` наследует все методы базовых классов и поддерживает следующие собственные методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qtoolbar.html**):

- `addAction()` — добавляет на панель инструментов действие, указанное непосредственно или вновь созданное на основе заданных параметров. Форматы метода:

  ```
  addAction(<Действие QAction>)
  addAction(<Название>)                                -> QAction
  addAction(<Значок QIcon>, <Название>)                -> QAction
  addAction(<Название>, <Обработчик>)                  -> QAction
  addAction(<Значок QIcon>, <Название>, <Обработчик>) -> QAction
  ```

  Все форматы, кроме первого, возвращают ссылку на добавленное действие;

- `addSeparator()` — добавляет действие-разделитель и возвращает ссылку на него;
- `insertSeparator(<Действие QAction>)` — добавляет действие-разделитель перед указанным действием и возвращает ссылку на добавленное действие;
- `addWidget(<Компонент)` — добавляет заданный компонент (например, раскрывающийся список). Метод возвращает действие, представляющее добавленный компонент;
- `insertWidget(<Действие QAction>, <Компонент>)` — добавляет заданный компонент перед указанным действием и возвращает действие, представляющее добавленный компонент;
- `widgetForAction(<Действие QAction>)` — возвращает ссылку на компонент, который связан с указанным действием;
- `clear()` — удаляет все действия из панели инструментов;
- `setOrientation(<Ориентация>)` — задает ориентацию панели в виде следующих элементов перечисления `Orientation` из модуля `QtCore.Qt`:
  - `Horizontal` — горизонтальная (значение по умолчанию);
  - `Vertical` — вертикальная;
- `setAllowedAreas(<Области>)` — задает области, к которым можно прикрепить панель инструментов. В качестве параметра указываются следующие элементы (или их комбинация через оператор `|`) перечисления `ToolBarArea` из модуля `QtCore.Qt`: `LeftToolBarArea` (слева), `RightToolBarArea` (справа), `TopToolBarArea` (сверху), `BottomToolBarArea` (снизу),

AllToolBarAreas (все области — значение по умолчанию) или NoToolBarArea (ни одна область);

- setMovable(<Флаг>) — если в качестве параметра указано значение False, будет невозможно перемещать панель с помощью мыши. Значение True вновь разрешает перемещение панели;
- isMovable() — возвращает значение True, если панель можно перемещать с помощью мыши, и False — в противном случае;
- setFloatable(<Флаг>) — если в качестве параметра указано значение False, панель нельзя будет вынести в отдельное окно. Значение True вновь разрешает выносить панель в окно;
- isFloatable() — возвращает значение True, если панель можно вынести в отдельное окно, и False — в противном случае;
- isFloating() — возвращает значение True, если панель в настоящий момент вынесена в отдельное окно, и False — в противном случае;
- setToolButtonStyle(<Стиль>) — задает стиль кнопок на панели инструментов. Метод является слотом. В качестве параметра указываются следующие элементы перечисления ToolButtonStyle из модуля QtCore.Qt:
  - ToolButtonIconOnly — только значок;
  - ToolButtonTextOnly — только текст;
  - ToolButtonTextBesideIcon — текст справа от значка;
  - ToolButtonTextUnderIcon — текст под значком;
  - ToolButtonFollowStyle — зависит от используемого стиля;
- toolButtonStyle() — возвращает стиль кнопок на панели инструментов;
- setIconSize(<Размеры QSize>) — задает размеры значков. Метод является слотом;
- iconSize() — возвращает размеры значков (объект класса QSize);
- toggleViewAction() — возвращает действие (объект класса QAction), с помощью которого можно скрыть или отобразить панель.

Класс QToolBar поддерживает следующие сигналы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qtoolbar.html**):

- actionTriggered(<Действие QAction>) — генерируется при нажатии кнопки на панели. Внутри обработчика через параметр доступно действие, связанное с этой кнопкой;
- visibilityChanged(<Флаг>) — генерируется при изменении видимости панели. Внутри обработчика через параметр доступно значение True, если панель видима, и False — если скрыта;
- topLevelChanged(<Флаг>) — генерируется при изменении состояния панели. Внутри обработчика через параметр доступно значение True, если панель вынесена в отдельное окно, и False — если прикреплена к области.

28.3.2. Класс *QToolButton*

При добавлении на панель инструментов какого-либо действия автоматически создается кнопка, представляемая классом QToolButton. Получить ссылку на кнопку позволяет метод

widgetForAction() класса QToolBar. Иерархия наследования для класса QToolButton выглядит так:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QToolButton
```

Конструктор класса QToolButton имеет следующий формат:

```
QToolButton([parent=None])
```

В параметре parent указывается ссылка на родительский компонент.

Класс QToolButton, помимо методов базовых классов, поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qtoolbutton.html**):

- setDefaultAction(<Действие QAction>) — связывает указанное действие с кнопкой. Метод является слотом;
- defaultAction() — возвращает ссылку на действие (объект класса QAction), связанное с кнопкой;
- setToolButtonStyle(<Стиль>) — задает стиль кнопки. Метод является слотом. Допустимые значения параметра приведены в описании метода setToolButtonStyle() (см. *разд. 28.3.1*);
- toolButtonStyle() — возвращает стиль кнопки;
- setMenu(<Меню QMenu>) — добавляет к кнопке заданное меню;
- menu() — возвращает ссылку на меню (объект класса QMenu), добавленное к кнопке, или значение None;
- showMenu() — отображает меню, связанное с кнопкой. Метод является слотом;
- setPopupMode(<Режим>) — задает режим отображения меню, связанного с кнопкой. В качестве параметра указываются следующие элементы перечисления ToolButtonPopupMode из класса QToolButton:
  - DelayedPopup — меню отображается при удержании кнопки, нажатой в течение некоторого промежутка времени;
  - MenuButtonPopup — справа от кнопки отображается кнопка со стрелкой, нажатие которой приводит к немедленному открытию меню;
  - InstantPopup — нажатие кнопки приводит к немедленному открытию меню. Сигнал triggered() при этом не генерируется;
- popupMode() — возвращает режим отображения меню, связанного с кнопкой;
- setArrowType(<Тип значка>) — устанавливает у кнопки, связанной с меню, значок в виде стрелки. В качестве параметра задаются элементы NoArrow (нет стрелки — значение по умолчанию), UpArrow (стрелка вверх), DownArrow (стрелка вниз), LeftArrow (стрелка влево) или RightArrow (стрелка вправо) перечисления ArrowType из модуля QtCore.Qt;
- setAutoRaise(<Флаг>) — если в качестве параметра указано значение False, кнопка будет отображаться с рамкой. По умолчанию кнопка сливается с фоном, а при наведении указателя мыши становится выпуклой.

Класс QToolButton поддерживает сигнал triggered(<Действие QAction>), который генерируется при нажатии кнопки или комбинации клавиш, а также при выборе пункта в связанном меню. Внутри обработчика через параметр доступно соответствующее действие.

28.4. Прикрепляемые панели

Если необходимо вывести на экран компоненты, занимающие много места (например, таблицу или иерархический список), можно воспользоваться прикрепляемыми панелями. Прикрепляемые панели реализуются с помощью класса `QDockWidget`. Его иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDockWidget
```

Форматы конструктора класса `QDockWidget`:

```
QDockWidget([parent=None][, flags=0])
QDockWidget(<Название>[, parent=None][, flags=0])
```

В параметре `<Название>` задается название панели, отображаемое в заголовке панели и в контекстном меню при щелчке правой кнопкой мыши в области меню, области панелей инструментов или на заголовке прикрепляемых панелей. С помощью контекстного меню можно скрыть или отобразить прикрепляемую панель. В параметре `parent` указывается ссылка на родительское окно. Доступные значения параметра `flags` мы рассматривали в *разд. 19.1.1*.

Класс `QDockWidget` наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только интересующие нас — полный их список можно найти на странице **https://doc.qt.io/qt-6/qdockwidget.html**):

- `setWidget(<Компонент>)` — устанавливает компонент, который будет отображаться на прикрепляемой панели;
- `widget()` — возвращает ссылку на компонент, расположенный на панели;
- `setTitleBarWidget(<Компонент>)` — указывает компонент, отображаемый в заголовке панели;
- `titleBarWidget()` — возвращает ссылку на компонент, расположенный в заголовке панели, или значение `None`, если компонент не был установлен;
- `setAllowedAreas(<Области>)` — задает области, к которым можно прикрепить панель. В качестве параметра указываются следующие элементы (или их комбинация через оператор `|`) перечисления `ToolBarArea` из модуля `QtCore.Qt`: `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху), `BottomDockWidgetArea` (снизу) или `AllDockWidgetAreas` (все области — значение по умолчанию) или `NoToolBarArea` (ни одна область);
- `setFloating(<Флаг>)` — если в качестве параметра указано значение `True`, панель отобразится в отдельном окне, а если указано значение `False`, она будет прикреплена к какой-либо области;
- `isFloating()` — возвращает значение `True`, если панель отображается в отдельном окне, и `False` — в противном случае;
- `setFeatures(<Свойства>)` — устанавливает свойства панели. В качестве параметра указывается комбинация (через оператор `|`) следующих элементов перечисления `DockWidgetFeature` из класса `QDockWidget`:
  - `DockWidgetClosable` — панель можно закрыть;
  - `DockWidgetMovable` — панель можно перемещать мышью;
  - `DockWidgetFloatable` — панель можно вынести в отдельное окно;

- DockWidgetVerticalTitleBar — заголовок панели отображается с левой стороны, а не сверху;
- NoDockWidgetFeatures — панель нельзя закрыть, переместить и вынести в отдельное окно;

- features() — возвращает комбинацию установленных свойств панели;
- toggleViewAction() — возвращает действие (объект класса QAction), с помощью которого можно скрыть или отобразить панель.

Класс QDockWidget поддерживает следующие полезные сигналы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qdockwidget.html**):

- dockLocationChanged(<Область ToolBarArea>) — генерируется при переносе панели в другую область. Внутри обработчика через параметр доступен идентификатор области, на которую была перенесена панель;
- visibilityChanged(<Флаг>) — генерируется при изменении видимости панели. Внутри обработчика через параметр доступно значение True, если панель видима, и False — если скрыта;
- topLevelChanged(<Флаг>) — генерируется при изменении состояния панели. Внутри обработчика через параметр доступно значение True, если панель вынесена в отдельное окно, и False — если прикреплена к области.

28.5. Строка состояния

Класс QStatusBar реализует строку состояния, в которой можно выводить необходимые сообщения. Помимо текстовой информации, туда можно добавить различные компоненты — например, индикатор хода выполнения процесса. Строка состояния состоит из трех секций:

- *секция для временных сообщений.* Реализована по умолчанию. В эту секцию, в частности, при наведении курсора мыши на пункт меню или кнопку на панели инструментов выводятся сообщения, заданные у действия с помощью метода setStatusTip(). Вывести пользовательское сообщение во временную секцию можно с помощью метода showMessage();
- *обычная секция.* При выводе временного сообщения содержимое обычной секции скрывается. Чтобы отображать сообщения в этой секции, необходимо предварительно добавить туда компоненты вызовом метода addWidget() или insertWidget(). Добавленные компоненты выравниваются по левой стороне строки состояния;
- *постоянная секция.* При выводе временного сообщения содержимое постоянной секции не скрывается. Чтобы отображать там сообщения, необходимо предварительно добавить туда компоненты методом addPermanentWidget() или insertPermanentWidget(). Добавленные компоненты выравниваются по правой стороне строки состояния.

Получить ссылку на строку состояния, установленную в главном окне, позволяет метод statusBar() класса QMainWindow, а установить пользовательскую панель вместо стандартной можно с помощью метода setStatusBar(<Строка состояния QStatusBar>). Иерархия наследования для класса QStatusBar выглядит так:

```
(QObject, QPaintDevice) — QWidget — QStatusBar
```

Формат конструктора класса `QStatusBar`:

```
QStatusBar([parent=None])
```

В параметре `parent` указывается ссылка на родительское окно.

Класс `QStatusBar` наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qstatusbar.html**):

- `showMessage(<Текст>[, msecs=0])` — выводит временное сообщение. Во втором параметре можно указать время показа сообщения в миллисекундах — если указано значение 0, то сообщение показывается, пока не будет выведено новое сообщение или вызван метод `clearMessage()`. Метод является слотом;
- `currentMessage()` — возвращает временное сообщение, отображаемое в текущий момент;
- `clearMessage()` — удаляет временное сообщение. Метод является слотом;
- `addWidget(<Компонент>[, stretch=0])` — добавляет указанный компонент в конец обычной секции. В параметре `stretch` может быть указан фактор растяжения;
- `insertWidget(<Индекс>, <Компонент>[, stretch=0])` — добавляет заданный компонент в позицию обычной секции с указанным индексом и возвращает индекс позиции. В параметре `stretch` может быть указан фактор растяжения;
- `addPermanentWidget(<Компонент>[, stretch=0])` — добавляет указанный компонент в конец постоянной секции. В параметре `stretch` может быть указан фактор растяжения;
- `insertPermanentWidget(<Индекс>, <Компонент>[, stretch=0])` — добавляет заданный компонент в позицию постоянной секции с указанным индексом и возвращает индекс позиции. В параметре `stretch` может быть указан фактор растяжения;
- `removeWidget(<Компонент>)` — удаляет компонент из обычной или постоянной секции. При этом сам компонент не удаляется, а только скрывается, лишается родителя и в дальнейшем может быть помещен в другое место;
- `setSizeGripEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, в правом нижнем углу строки состояния будет отображаться маркер изменения размера. Значение `False` скрывает маркер.

Класс `QStatusBar` поддерживает сигнал `messageChanged(<Сообщение>)`, генерируемый при изменении текста во временной секции. Внутри обработчика через параметр доступно новое сообщение или пустая строка.

28.6. MDI-программы

Чтобы создать MDI-программу, следует в качестве центрального компонента главного окна установить компонент `QMdiArea` вызовом метода `setCentralWidget()` класса `QMainWindow`. Отдельное окно внутри MDI-области представляется классом `QMdiSubWindow`.

28.6.1. Класс *QMdiArea*

Класс `QMdiArea` реализует MDI-область, внутри которой располагаются вложенные окна (объекты класса `QMdiSubWindow`). Иерархия наследования для класса `QMdiArea` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea — QMdiArea
```

Конструктор класса `QMdiArea` имеет следующий формат:

```
QMdiArea([parent=None])
```

В параметре `parent` указывается ссылка на родительское окно.

Класс `QMdiArea` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-6/qmdiarea.html**):

- `addSubWindow(<Компонент>[, flags=0])` — создает вложенное окно с флагами `flags` (см. *разд. 19.1.1*), добавляет в него заданный компонент и возвращает ссылку на созданное окно (объект класса `QMdiSubWindow`):

  ```
  w = MyWidget()
  sWindow = self.mdi_area.addSubWindow(w)
  sWindow.setAttribute(QtCore.Qt.WidgetAttribute.WA_DeleteOnClose)
  # ... Задаем параметры окна
  sWindow.show()
  ```

  Чтобы окно автоматически удалялось при закрытии, необходимо установить у него флаг `WA_DeleteOnClose`, а чтобы отобразить окно — вызвать метод `show()`.

  В первом параметре этого метода также можно указать ссылку на существующее вложенное окно (объект класса `QMdiSubWindow`):

  ```
  w = MyWidget()
  sWindow = QtWidgets.QMdiSubWindow()
  self.mdi_area.addSubWindow(sWindow)
  sWindow.setAttribute(QtCore.Qt.WidgetAttribute.WA_DeleteOnClose)
  # ... Производим настройку свойств окна
  sWindow.setWidget(w)
  sWindow.show()
  ```

  Кроме того, можно добавить вложенное окно в MDI-область, указав последнюю в качестве родителя при создании объекта вложенного окна:

  ```
  sWindow = QtWidgets.QMdiSubWindow(self.mdi_area)
  ```

- `activeSubWindow()` — возвращает ссылку на активное вложенное окно (объект класса `QMdiSubWindow`) или значение `None`;
- `currentSubWindow()` — возвращает ссылку на текущее вложенное окно (объект класса `QMdiSubWindow`) или значение `None`. Результат выполнения этого метода аналогичен таковому у метода `activeSubWindow()`, если MDI-область находится в активном окне;
- `subWindowList([order=WindowOrder.CreationOrder])` — возвращает список со ссылками на все вложенные окна (объекты класса `QMdiSubWindow`), добавленные в MDI-область, или пустой список. В параметре `order` указываются следующие элементы перечисления `WindowOrder` из класса `QMdiArea`:
  - `CreationOrder` — окна в списке указываются в порядке их создания;
  - `StackingOrder` — окна в списке указываются в порядке размещения их на экране. Последний элемент в списке будет содержать ссылку на самое верхнее окно, а первый — ссылку на самое нижнее окно;
  - `ActivationHistoryOrder` — окна в списке указываются в порядке истории получения ими фокуса. Последний элемент в списке будет содержать ссылку на окно, получившее фокус последним;

- `removeSubWindow(<Компонент>)` — удаляет из MDI-области вложенное окно, содержащее указанный компонент. В параметре также можно определить подлежащее удалению вложенное окно (объект класса `QMdiSubWindow`);
- `setActiveSubWindow(<Вложенное окно>)` — делает указанное вложенное окно активным. Если задать значение `None`, все вложенные окна станут неактивными. Метод является слотом;
- `setActivationOrder(<Порядок WindowOrder>)` — задает порядок передачи фокуса при использовании методов `activatePreviousSubWindow()`, `activateNextSubWindow()` и др. В параметре указываются те же элементы, что и в параметре метода `subWindowList()`;
- `activationOrder()` — возвращает порядок передачи фокуса;
- `activateNextSubWindow()` — делает активным следующее вложенное окно. Метод является слотом. Порядок передачи фокуса устанавливается с помощью метода `setActivationOrder()`;
- `activatePreviousSubWindow()` — делает активным предыдущее вложенное окно. Метод является слотом. Порядок передачи фокуса устанавливается с помощью метода `setActivationOrder()`;
- `closeActiveSubWindow()` — закрывает активное вложенное окно. Метод является слотом;
- `closeAllSubWindows()` — закрывает все вложенные окна. Метод является слотом;
- `cascadeSubWindows()` — выстраивает вложенные окна в виде стопки. Метод является слотом;
- `tileSubWindows()` — выстраивает вложенные окна в виде мозаики. Метод является слотом;
- `setViewMode(<Режим>)` — задает режим отображения вложенных окон. В параметре `<Режим>` указываются следующие элементы перечисления `ViewMode` из класса `QMdiArea`:
  - `SubWindowView` — в виде собственно окон (по умолчанию);
  - `TabbedView` — в виде отдельных вкладок на панели с вкладками;
- `viewMode()` — возвращает режим отображения вложенных окон;
- `setTabPosition(<Позиция>)` — задает позицию отображения заголовков вкладок при использовании режима `TabbedView`. В качестве параметра могут быть указаны следующие элементы перечисления `TabPosition` из класса `QTabWidget`: `North` (сверху — значение по умолчанию), `South` (снизу), `West` (слева) или `East` (справа);
- `tabPosition()` — возвращает позицию отображения заголовков вкладок при использовании режима `TabbedView`;
- `setTabShape(<Форма>)` — задает форму углов ярлыков вкладок при использовании режима `TabbedView`. Могут быть указаны следующие элементы перечисления `TabShape` из класса `QTabWidget`: `Rounded` (скругленные углы — значение по умолчанию) или `Triangular` (треугольная форма);
- `tabShape()` — возвращает форму углов ярлыков вкладок в области заголовка при использовании режима `TabbedView`;
- `setTabsMovable(<Флаг>)` — если передано значение `True`, вкладки на панели при использовании режима `TabbedView` можно перемещать мышью;

- setTabsClosable(<Флаг>) — если передано значение True, вкладки на панели при использовании режима TabbedView можно закрывать щелчком на расположенной в заголовке вкладки кнопке закрытия;
- setBackground(<Кисть QBrush>) — задает кисть для заполнения фона MDI-области;
- setOption(<Опция>[, on=True]) — если во втором параметре указано значение True, заданная в первом параметре опция будет установлена, если False — сброшена. В параметре <Опция> может быть указан единственный поддерживаемый в настоящее время элемент DontMaximizeSubWindowOnActivation перечисления AreaOption из класса QMdiArea, в случае установки запрещающий автоматически разворачивать следующее вложенное окно при переключении на него из уже развернутого окна;
- testOption(<Опция AreaOption>) — возвращает значение True, если указанная опция установлена, и False — в противном случае.

Класс QMdiArea поддерживает сигнал subWindowActivated(<Вложенное окно>), генерируемый при активизации другого вложенного окна. В обработчике через параметр доступна ссылка на активизированное вложенное окно или значение None.

28.6.2. Класс *QMdiSubWindow*

Класс QMdiSubWindow реализует вложенное окно, отображаемое внутри MDI-области. Иерархия наследования для него выглядит следующим образом:

```
(QObject, QPaintDevice) — QWidget — QMdiSubWindow
```

Формат конструктора класса QMdiSubWindow:

```
QMdiSubWindow([parent=None][, flags=0])
```

В параметре parent указывается ссылка на родительское окно. Доступные значения параметра flags мы рассматривали в *разд. 19.1.1*.

Класс QMdiSubWindow наследует все методы базовых классов и дополнительно поддерживает следующие методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qmdisubwindow.html**):

- setWidget(<Компонент>) — помещает заданный компонент в окно;
- widget() — возвращает ссылку на помещенный в окно компонент;
- mdiArea() — возвращает ссылку на MDI-область (объект класса QMdiArea) или значение None;
- setSystemMenu(<Меню QMenu>) — устанавливает указанное меню у окна в качестве системного;
- systemMenu() — возвращает ссылку на системное меню окна (объект класса QMenu) или значение None;
- showSystemMenu() — открывает системное меню окна. Метод является слотом;
- setKeyboardSingleStep(<Значение>) — устанавливает шаг изменения размера окна или его положения с помощью клавиш со стрелками. Чтобы изменить размеры окна или его положение с клавиатуры, необходимо в системном меню окна выбрать пункт **Переместить** или **Размер**. Значение по умолчанию — 5;
- setKeyboardPageStep(<Значение>) — устанавливает шаг изменения размера окна или его положения с помощью клавиш со стрелками при удержании нажатой клавиши <Shift>. Значение по умолчанию — 20;

- `showShaded()` — сворачивает содержимое окна, оставляя только заголовок. Метод является слотом;
- `isShaded()` — возвращает значение `True`, если отображается только заголовок окна, и `False` — в противном случае;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, заданная в первом параметре опция будет установлена, а если `False` — сброшена. В параметре `<Опция>` могут быть указаны следующие элементы перечисления `SubWindowOption` из класса `QMdiSubWindow`:
  - `RubberBandResize` — если опция установлена, при изменении размеров окна станут изменяться размеры вспомогательного компонента, а не самого окна. По окончании изменения размеров будут изменены размеры окна;
  - `RubberBandMove` — если опция установлена, при изменении положения окна станет перемещаться вспомогательный компонент, а не само окно. По окончании перемещения будет изменено положение окна;
- `testOption(<Опция SubWindowOption>)` — возвращает значение `True`, если указанная опция установлена, и `False` — в противном случае.

Класс `QMdiSubWindow` поддерживает следующие сигналы:

- `aboutToActivate()` — генерируется перед активацией вложенного окна;
- `windowStateChanged(<Старое состояние>, <Новое состояние>)` — генерируется при изменении состояния окна. Внутри обработчика через параметры доступны обозначения старого и нового состояний в виде элементов перечисления `WindowState` из модуля `QtCore.Qt`.

28.7. Добавление значка программы в область уведомлений

Класс `QSystemTrayIcon` позволяет добавить значок в область уведомлений, расположенную в правой части панели задач Windows. Иерархия наследования для этого класса выглядит так:

```
QObject — QSystemTrayIcon
```

Форматы конструктора класса `QSystemTrayIcon`:

```
QSystemTrayIcon([parent=None])
QSystemTrayIcon(<Значок QIcon>[, parent=None])
```

В параметре `parent` указывается ссылка на родительское окно.

Класс `QSystemTrayIcon` поддерживает следующие основные методы:

- `isSystemTrayAvailable()` — статический, возвращает значение `True`, если область уведомлений доступна, и `False` — в противном случае;
- `setIcon(<Значок QIcon>)` — устанавливает заданный значок;
- `icon()` — возвращает значок (объект класса `QIcon`);
- `setContextMenu(<Меню QMenu>)` — устанавливает заданное меню у значка в качестве контекстного, отображаемого при щелчке правой кнопкой мыши;

- `contextMenu()` — возвращает ссылку на контекстное меню значка (объект класса `QMenu`);
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `toolTip()` — возвращает текст всплывающей подсказки;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `True`, значок будет выведен на экран, а если `False` — то скрыт. Метод является слотом;
- `show()` — отображает значок. Метод является слотом;
- `hide()` — скрывает значок. Метод является слотом;
- `isVisible()` — возвращает значение `True`, если значок присутствует на экране, и `False` — в противном случае;
- `geometry()` — возвращает объект класса `QRect` с размерами и координатами значка на экране;
- `supportsMessages()` — статический, возвращает значение `True`, если поддерживается вывод сообщений, и `False` — в противном случае;
- `showMessage()` — выводит сообщение в области уведомлений. Форматы метода:

```
showMessage(<Заголовок>, <Текст сообщения>[, icon=MessageIcon.Information][,
                                                msecs=10000])
showMessage(<Заголовок>, <Текст сообщения>, <Значок QIcon>[, msecs=10000])
```

  В первом формате необязательный параметр `icon` задает значок, который отображается слева от заголовка сообщения. В качестве значения можно указать следующие элементы перечисления `MessageIcon` из класса `QSystemTrayIcon`: `NoIcon` (без значка), `Information`, `Warning` или `Critical`.

  Во втором формате третьим параметром можно указать произвольный значок.

  Необязательный параметр `msecs` задает промежуток времени, в течение которого сообщение будет присутствовать на экране. Обратите внимание, что сообщение может не показываться вообще, кроме того, значение параметра `msecs` в некоторых операционных системах игнорируется.

  Метод является слотом.

Класс `QSystemTrayIcon` поддерживает следующие сигналы:

- `activated(<Причина>)` — генерируется при щелчке мышью на значке. Внутри обработчика через параметр доступна причина в виде значения одного из следующих элементов перечисления `ActivationReason` из класса `QSystemTrayIcon`:
  - `Unknown` — неизвестная причина;
  - `Context` — нажата правая кнопка мыши;
  - `DoubleClick` — двойной щелчок мышью;
  - `Trigger` — нажата левая кнопка мыши;
  - `MiddleClick` — нажата средняя кнопка или колесико мыши;
- `messageClicked()` — генерируется при щелчке мышью на сообщении.

ГЛАВА 29

Мультимедиа

Библиотека PyQt предоставляет развитые средства воспроизведения звука и видео. Для этого она задействует мультимедийную подсистему Windows Media Foundation, входящую в состав Windows. Следовательно, все форматы, поддерживаемые операционной системой, будут гарантированно поддерживаться и PyQt.

Все описанные в этой главе классы объявлены в модуле `QtMultimedia`, если не указано иное.

29.1. Воспроизведение мультимедиа

29.1.1. Мультимедийный проигрыватель

Класс `QMediaPlayer` реализует полнофункциональный мультимедийный проигрыватель, позволяющий воспроизводить как звук, так и видео из файлов и сетевых источников. Единственный его недостаток заключается в том, что он не предоставляет никакого интерфейса для управления воспроизведением: ни кнопок запуска, остановки и паузы, ни регулятора громкости, ни индикатора текущей позиции воспроизведения. Все это придется делать самостоятельно.

Иерархия наследования для класса `QMediaPlayer` выглядит так:

```
QObject - QMediaPlayer
```

Формат вызова конструктора:

```
QMediaPlayer([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Класс `QMediaPlayer` поддерживает следующие наиболее полезные для нас методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qmediaplayer.html**):

- `setAudioOutput(<Звуковой выход>)` — соединяет мультимедийный проигрыватель с заданным звуковым выходом (объект класса `QAudioOutput`, описываемого в *разд. 29.1.2*);
- `audioOutput()` — возвращает звуковой выход, с которым в настоящий момент соединен текущий проигрыватель, в виде объекта класса `QAudioOutput`;
- `setVideoOutput(<Видеопанель>)` — задает видеопанель, в которой будет выводиться видео (объект класса `QVideoWidget`, описываемого в *разд. 29.1.4*). Используется лишь при воспроизведении видео;

- videoOutput() — возвращает видеопанель, с которой в настоящий момент соединен текущий проигрыватель, в виде объекта класса QVideoWidget;
- setSource(<Путь QUrl>) — задает путь к источнику (локальному мультимедийному файлу или сетевому ресурсу), который будет воспроизводиться. Если передать «пустой» объект класса QUrl, заданный ранее источник будет закрыт. Пример задания источника-файла:

```
mp = QtMultimedia.QMediaPlayer()
mc = QtCore.QUrl.fromLocalFile(r"c:\media\song1.mp3")
mp.setSource(mc)
```

  Пример задания сетевого источника:

```
mp.setSource(QtCore.QUrl(r"https://www.someradio.ru/audio.mp3"))
```

  Метод является слотом;
- source() — возвращает заданный путь к источнику (объект класса QUrl);
- play() — начинает или возобновляет воспроизведение мультимедиа. Метод является слотом;
- pause() — приостанавливает воспроизведение мультимедиа. Метод является слотом;
- stop() — останавливает воспроизведение мультимедиа и перемещает позицию воспроизведения на начало. Метод является слотом;
- playbackState() — возвращает текущее состояние проигрывателя в виде значения одного из следующих элементов перечисления PlaybackState из класса QMediaPlayer:
  - StoppedState — воспроизведение либо остановлено, либо еще не начиналось;
  - PlayingState — идет воспроизведение;
  - PausedState — воспроизведение приостановлено;
- duration() — возвращает продолжительность воспроизведения мультимедиа в миллисекундах;
- position() — возвращает текущую позицию воспроизведения мультимедиа в миллисекундах;
- isSeekable() — возвращает True, если имеется возможность изменить позицию воспроизведения, и False, если источник этого не поддерживает;
- setPosition(<Позиция>) — задает текущую позицию воспроизведения в миллисекундах. Метод является слотом;
- setPlaybackRate(<Скорость>) — задает скорость воспроизведения мультимедиа. Значение параметра должно представлять собой вещественное число, умножаемое на стандартную скорость воспроизведения, чтобы получить ее новую величину. Значение 1.0 задает стандартную скорость, меньшее 1.0 — уменьшенную скорость, а большее 1.0 — увеличенную. Метод является слотом;
- playbackRate() — возвращает текущую скорость воспроизведения мультимедиа;
- setLoops(<Количество>) — задает количество повторений воспроизводящегося мультимедиа в виде целого числа. По умолчанию источник воспроизводится один раз. Если указать число -1, источник будет воспроизводиться бесконечно;
- loops() — возвращает количество повторений воспроизведения мультимедиа в виде целого числа;

- `mediaStatus()` — возвращает состояние источника в виде значения одного из следующих элементов перечисления `MediaStatus` из класса `QMediaPlayer`:
  - `NoMedia` — источник отсутствует;
  - `LoadingMedia` — идет загрузка источника;
  - `LoadedMedia` — источник загружен;
  - `StalledMedia` — воспроизведение приостановлено из-за опустошения внутреннего буфера проигрывателя или невозможности загрузить следующую порцию данных;
  - `BufferingMedia` — идет заполнение данными внутреннего буфера проигрывателя, однако данных достаточно для продолжения воспроизведения;
  - `BufferedMedia` — внутренний буфер заполнен, идет воспроизведение;
  - `EndOfMedia` — воспроизведение источника закончено;
  - `InvalidMedia` — источник не может быть воспроизведен (неподдерживаемый формат мультимедийных данных, поврежденный файл и т. п.);
- `hasAudio()` — возвращает `True`, если источник содержит звук, и `False` — в противном случае. Обычно используется при воспроизведении видеороликов, чтобы узнать, имеют ли они звуковое сопровождение;
- `hasVideo()` — возвращает `True`, если источник содержит видео, и `False` — в противном случае;
- `bufferProgress()` — возвращает процент заполнения внутреннего буфера проигрывателя в виде вещественного числа от `0.0` до `1.0`;
- `error()` — возвращает обозначение ошибки, возникшей при воспроизведении мультимедиа, в виде одного из следующих элементов перечисления `Error` из класса `QMediaPlayer`:
  - `NoError` — никакой ошибки не возникло;
  - `ResourceError` — некорректный путь или интернет-адрес источника;
  - `FormatError` — неподдерживаемый формат звука или видео;
  - `NetworkError` — сетевая ошибка;
  - `AccessDeniedError` — недостаточно прав для загрузки источника;
- `errorString()` — возвращает текстовое описание ошибки;
- `metaData()` — возвращает метаданные воспроизводящегося в текущий момент источника в виде объекта класса `QMediaMetaData` (описан в *разд. 29.1.3*);
- `audioTracks()` — возвращает список со всеми звуковыми дорожками, имеющимися в воспроизводящемся источнике и представленными объектами класса `QMediaMetaData`;
- `videoTracks()` — возвращает список со всеми видеодорожками, имеющимися в воспроизводящемся источнике и представленными объектами класса `QMediaMetaData`;
- `subtitleTracks()` — возвращает список со всеми дорожками субтитров, имеющимися в воспроизводящемся источнике и представленными объектами класса `QMediaMetaData`;
- `setActiveAudioTrack(<Индекс>)` — выбирает для воспроизведения звуковую дорожку с указанным индексом;
- `activeAudioTrack()` — возвращает индекс воспроизводящейся в текущий момент звуковой дорожки;

- `setActiveVideoTrack(<Индекс>)` — выбирает для воспроизведения видеодорожку с указанным индексом;
- `activeVideoTrack()` — возвращает индекс воспроизводящейся в текущий момент видеодорожки;
- `setActiveSubtitleTrack(<Индекс>)` — выбирает для воспроизведения дорожку субтитров с указанным индексом;
- `activeSubtitleTrack()` — возвращает индекс воспроизводящейся в текущий момент дорожки субтитров.

Класс `QMediaPlayer` поддерживает большое количество сигналов, из которых для нас представляют интерес лишь приведенные далее (полный их список можно найти на странице **https://doc.qt.io/qt-6/qmediaplayer.html**):

- `sourceChanged(<Путь QUrl>)` — генерируется при указании другого источника. Путь к новому источнику передается в параметре;
- `durationChanged(<Продолжительность>)` — генерируется при первом получении или изменении продолжительности источника. Новое значение продолжительности, заданное целым числом в миллисекундах, передается обработчику в параметре;
- `playbackStateChanged(<Состояние PlaybackState>)` — генерируется при изменении состояния проигрывателя. В параметре обработчику передается новое состояние;
- `positionChanged(<Позиция>)` — генерируется при изменении позиции воспроизведения. В параметре обработчику передается новая позиция в виде целого числа в миллисекундах;
- `mediaStatusChanged(<Состояние MediaStatus>)` — генерируется при изменении состояния источника. Новое обозначение состояния передается в параметре;
- `playbackRateChanged(<Скорость>)` — генерируется при изменении скорости воспроизведения. В параметре передается новое значение скорости воспроизведения, представленное вещественным числом;
- `bufferProgressChanged(<Процент>)` — генерируется при изменении процента заполнения внутреннего буфера проигрывателя. Новая величина процента заполнения в виде вещественного числа от `0.0` до `1.0` передается в параметре;
- `errorOccured(<Ошибка Error>, <Описание>)` — генерируется при возникновении ошибки. В параметрах передаются обозначение возникшей ошибки и ее строковое описание;
- `metaDataChanged()` — генерируется при изменении метаданных ролика.

29.1.2. Звуковой выход. Воспроизведение звука

Класс `QAudioOutput` представляет звуковой выход, через который выполняется вывод звука. Иерархия наследования для этого класса:

```
QObject - QAudioOutput
```

Формат вызова конструктора:

```
QAudioOutput([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Звуковой выход, представляемый этим классом, изначально выводит звук на устройство вывода звука, помеченное в настройках операционной системы как используемое по умол-

чанию. Однако звуковому выходу можно указать выводить звук на любое другое доступное устройство.

Выяснить, какие устройства вывода звука существуют на текущей платформе, позволяют следующие статические методы класса `QMediaDevices`:

- `audioOutputs()` — возвращает список доступных устройств вывода звука, представляемых объектами класса `QAudioDevice` (описан далее):

```
>>> dlist = QtMultimedia.QMediaDevices.audioOutputs()
>>> for d in dlist: print(d.description())
...
    Динамики (2- High Definition Audio Device)
    PHL 276E8V (NVIDIA High Definition Audio)
    Цифровое аудио (S/PDIF) (2- High Definition Audio Device)
    Цифровое аудио (S/PDIF) (2- High Definition Audio Device)
```

- `defaultAudioOutput()` — возвращает устройство вывода звука, помеченное как используемое по умолчанию (объект класса `QAudioDevice`):

```
>>> QtMultimedia.QMediaDevices.defaultAudioOutput().description()
    'Динамики (2- High Definition Audio Device)'
```

Класс `QAudioDevice` представляет доступное на текущей платформе устройство для вывода звука. Метод `description()` этого класса возвращает строковое название текущего устройства вывода звука. Полное описание класса `QAudioDevice` приведено на странице **https://doc.qt.io/qt-6/qaudiodevice.html**.

Класс `QAudioOutput` поддерживает следующие полезные методы:

- `setVolume(<Громкость>)` — задает громкость звука в виде вещественного числа от `0.0` (звук отключен) до `1.0` (максимальная громкость). Метод является слотом;
- `volume()` — возвращает громкость звука в виде вещественного числа от `0.0` до `1.0`;
- `setMuted(<Флаг>)` — если передать значение `True`, звук будет отключен. Значение `False` восстанавливает прежнюю громкость. Метод является слотом;
- `isMuted()` — возвращает `True`, если звук в настоящее время отключен, и `False` — в противном случае;
- `setDevice(<Устройство QAudioDevice>)` — назначает текущему звуковому выходу заданное устройство вывода звука:

```
ao = QtMultimedia.QAudioOutput(parent=self)
dlist = QtMultimedia.QMediaDevices.audioOutputs()
ao.setDevice(dlist[1])
```

- `device()` — возвращает устройство вывода звука, назначенное у текущего звукового выхода (объект класса `QAudioDevice`).

Еще этот класс поддерживает такие сигналы:

- `volumeChanged(<Громкость>)` — генерируется при изменении громкости. В параметре передается новое значение громкости в виде вещественного числа от `0.0` до `1.0`;
- `mutedChanged(<Флаг>)` — генерируется при отключении или, напротив, включении звука. В параметре передается значение `True`, если звук отключен, или `False`, если вновь включен;

- `deviceChanged()` — генерируется при указании у звукового выхода другого устройства вывода звука.

Соединить проигрыватель со звуковым выходом можно вызовом метода `setAudioOutput(<Звуковой выход QAudioOutput>)` класса `QMediaPlayer`.

Простейший аудиопроигрыватель (листинг 29.1) содержит кнопку для открытия звукового файла, кнопки пуска, паузы и остановки воспроизведения, регулятор текущей позиции воспроизведения, регулятор громкости и кнопку временного отключения звука (рис. 29.1).

Рис. 29.1. Аудиопроигрыватель

Листинг 29.1. Аудиопроигрыватель

```
from PyQt6 import QtCore, QtWidgets, QtMultimedia
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Аудиопроигрыватель")
        # Создаем устройство вывода звука
        self.aOutput = QtMultimedia.QAudioOutput(parent=self)
        self.aOutput.setVolume(0.5)
        # Создаем сам проигрыватель
        self.mplPlayer = QtMultimedia.QMediaPlayer(parent=self)
        self.mplPlayer.setAudioOutput(self.aOutput)
        self.mplPlayer.mediaStatusChanged.connect(self.initPlayer)
        self.mplPlayer.playbackStateChanged.connect(self.setPlayerState)
        vbox = QtWidgets.QVBoxLayout()
        # Создаем кнопку открытия файла
        btnOpen = QtWidgets.QPushButton("&Открыть файл...")
        btnOpen.clicked.connect(self.openFile)
        vbox.addWidget(btnOpen)
        # Создаем компоненты для управления воспроизведением.
        # Делаем их изначально недоступными
        self.sldPosition = QtWidgets.QSlider(QtCore.Qt.Orientation.Horizontal)
        self.sldPosition.setMinimum(0)
        self.sldPosition.valueChanged.connect(self.mplPlayer.setPosition)
        self.mplPlayer.positionChanged.connect(self.sldPosition.setValue)
```

```
        self.sldPosition.setEnabled(False)
        vbox.addWidget(self.sldPosition)
        hbox = QtWidgets.QHBoxLayout()
        self.btnPlay = QtWidgets.QPushButton("&Пуск")
        self.btnPlay.clicked.connect(self.mplPlayer.play)
        self.btnPlay.setEnabled(False)
        hbox.addWidget(self.btnPlay)
        self.btnPause = QtWidgets.QPushButton("П&ауза")
        self.btnPause.clicked.connect(self.mplPlayer.pause)
        self.btnPause.setEnabled(False)
        hbox.addWidget(self.btnPause)
        self.btnStop = QtWidgets.QPushButton("&Стоп")
        self.btnStop.clicked.connect(self.mplPlayer.stop)
        self.btnStop.setEnabled(False)
        hbox.addWidget(self.btnStop)
        vbox.addLayout(hbox)
        # Создаем компоненты для управления громкостью
        hbox = QtWidgets.QHBoxLayout()
        lblVolume = QtWidgets.QLabel("&Громкость")
        hbox.addWidget(lblVolume)
        self.sldVolume = QtWidgets.QSlider(QtCore.Qt.Orientation.Horizontal)
        self.sldVolume.setRange(0, 100)
        self.sldVolume.setTickPosition(
                        QtWidgets.QSlider.TickPosition.TicksAbove)
        self.sldVolume.setTickInterval(10)
        self.sldVolume.setValue(50)
        lblVolume.setBuddy(self.sldVolume)
        self.sldVolume.valueChanged.connect(self.setVolume)
        hbox.addWidget(self.sldVolume)
        btnMute = QtWidgets.QPushButton("&Тихо!")
        btnMute.setCheckable(True)
        btnMute.toggled.connect(self.aOutput.setMuted)
        hbox.addWidget(btnMute)
        vbox.addLayout(hbox)
        self.setLayout(vbox)
        self.resize(300, 100)

    # Для открытия файла используем метод getOpenFileUrl() класса
    # QFileDialog, т. к. для указания источника у проигрывателя
    # нам нужен путь к файлу, заданный в виде объекта класса QUrl
    def openFile(self):
        file = QtWidgets.QFileDialog.getOpenFileUrl(parent=self,
                        caption="Выберите звуковой файл",
                        filter="Звуковые файлы (*.mp3 *.ac3)")
        if file[1]:
            self.mplPlayer.setSource(file[0])

    def initPlayer(self, state):
        match state:
```

```
            case QtMultimedia.QMediaPlayer.MediaStatus.LoadedMedia:
                # После загрузки файла подготавливаем проигрыватель
                # для его воспроизведения
                self.mplPlayer.stop()
                self.btnPlay.setEnabled(True)
                self.btnPause.setEnabled(False)
                self.sldPosition.setEnabled(True)
                self.sldPosition.setMaximum(self.mplPlayer.duration())
            case QtMultimedia.QMediaPlayer.MediaStatus.EndOfMedia:
                # По окончании воспроизведения файла возвращаем
                # проигрыватель в изначальное состояние
                self.mplPlayer.stop()
                self.sldPosition.setValue(0)
                self.sldPosition.setEnabled(False)
                self.btnPlay.setEnabled(False)
                self.btnPause.setEnabled(False)
                self.btnStop.setEnabled(False)
            case QtMultimedia.QMediaPlayer.MediaStatus.NoMedia | \
                 QtMultimedia.QMediaPlayer.MediaStatus.InvalidMedia:
                # Если файл не был загружен, отключаем компоненты,
                # управляющие воспроизведением
                self.sldPosition.setValue(0)
                self.sldPosition.setEnabled(False)
                self.btnPlay.setEnabled(False)
                self.btnPause.setEnabled(False)
                self.btnStop.setEnabled(False)

    # В зависимости от того, воспроизводится ли файл, поставлен
    # ли он на паузу или остановлен, делаем соответствующие кнопки
    # доступными или недоступными
    def setPlayerState(self, state):
        match state:
            case QtMultimedia.QMediaPlayer.PlaybackState.StoppedState:
                self.sldPosition.setValue(0)
                self.btnPlay.setEnabled(True)
                self.btnPause.setEnabled(False)
                self.btnStop.setEnabled(False)
            case QtMultimedia.QMediaPlayer.PlaybackState.PlayingState:
                self.btnPlay.setEnabled(False)
                self.btnPause.setEnabled(True)
                self.btnStop.setEnabled(True)
            case QtMultimedia.QMediaPlayer.PlaybackState.PausedState:
                self.btnPlay.setEnabled(True)
                self.btnPause.setEnabled(False)
                self.btnStop.setEnabled(True)

    def setVolume(self, value):
        self.aOutput.setVolume(value / 100)
```

```
    # При закрытии окна останавливаем воспроизведение
    def closeEvent(self, e):
        self.mplPlayer.stop()
        e.accept()
        QtWidgets.QWidget.closeEvent(self, e)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec())
```

29.1.3. Метаданные мультимедийного источника

Класс `QMediaMetaData` представляет метаданные воспроизводящегося мультимедийного источника, его отдельной звуковой, видеодорожки или дорожки субтитров. Он поддерживает следующие полезные методы:

- `value(<Ключ>)` — возвращает метаданные, относящиеся к заданному ключу, или `None`, если метаданных с таким ключом в источнике нет. Ключ задается в виде одного из указанных далее элементов перечисления `Key` из класса `QMediaMetaData` (рассмотрены наиболее часто используемые элементы — полный их список можно найти на странице **https://doc.qt.io/qt-6/qmediametadata.html**). Если не указано обратное, возвращаемый методом результат представляет собой строку, а слово «список» означает обычный список Python (объект класса `list`).
  - `Title` — название произведения;
  - `Author` — список авторов произведения;
  - `Comment` — примечание;
  - `Description` — описание;
  - `Genre` — список жанров, к которым относится произведение;
  - `Date` — дата выпуска произведения в виде объекта класса `QDate`;
  - `Language` — язык произведения в виде элемента перечисления `Language` из класса `QLocale` (модуль `QtCore`);
  - `Copyright` — авторские права создателей произведения;
  - `FileFormat` — формат файла источника в виде элемента перечисления `FileFormat` из класса `QMediaFormat`;
  - `Duration` — продолжительность воспроизведения источника в миллисекундах, заданная целым числом;
  - `AudioCodec` — обозначение кодека, с помощью которого кодировался звук, в виде элемента перечисления `AudioCodec` из класса `QMediaFormat`;
  - `AudioBitRate` — битрейт звука в виде целого числа, измеряемый в битах в секунду;
  - `AlbumTitle` — название альбома, в который входит произведение;
  - `AlbumArtist` — автор альбома;
  - `ContributingArtist` — список исполнителей, занятых в записи альбома;
  - `Composer` — список композиторов, произведения которых входят в альбом;

- `LeadPerformer` — список ведущих исполнителей, занятых в записи альбома;
- `TrackNumber` — порядковый номер произведения в альбоме в виде целого числа;
- `VideoCodec` — обозначение кодека, с помощью которого кодировалось видео, в виде элемента перечисления `VideoCodec` из класса `QMediaFormat`;
- `Resolution` — размер видео в виде объекта класса `QSize`;
- `VideoFrameRate` — частота кадров видео в виде вещественного числа, измеряемая в кадрах в секунду;
- `VideoBitRate` — битрейт видео в виде целого числа, измеряемый в битах в секунду.

Следует отметить, что в источнике совсем не обязательно будут присутствовать все эти данные.

Получить метаданные, относящиеся к какому-либо ключу, также можно, обратившись к объекту класса `QMediaMetaData` как к словарю:

```
md = mediaplayer.metaData()
media_title = md[QtMultimedia.QMediaMetaData.Key.Title]
```

- `stringValue(<Ключ>)` — аналогичен `value()`, только всегда возвращает результат в виде строки;
- `keys()` — возвращает список ключей, содержащихся в текущем объекте.

> ***Внимание!***
>
> При вызове метода `keys()` происходит аварийное завершение программы. Вероятно, это следствие ошибки, которая присутствует во фреймворке Qt и будет исправлена в его будущих версиях.

- `isEmpty()` — возвращает `True`, если текущий объект не содержит метаданных, и `False` — в противном случае.

Если требуется лишь получить метаданные из загруженного мультимедийного источника, соединять проигрыватель со звуковым выходом необязательно.

В листинге 29.2 приведен код служебной утилиты, выводящей метаданные, которые хранятся в загруженном мультимедийном файле. В настоящий момент она выводит сведения об открытом в ней аудиофайле (рис. 29.2).

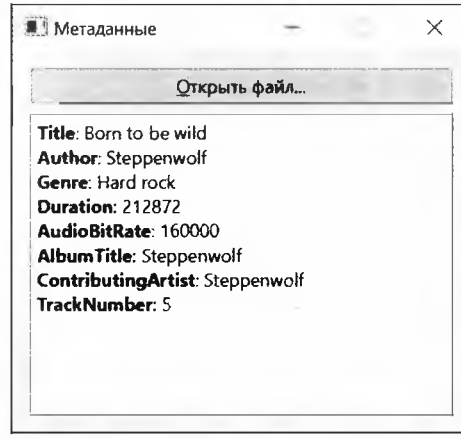


Рис. 29.2. Утилита, выводящая метаданные мультимедийного файла

Листинг 29.2. Вывод метаданных мультимедийного файла

```
from PyQt6 import QtCore, QtWidgets, QtMultimedia
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Метаданные")
        self.mplPlayer = QtMultimedia.QMediaPlayer(parent=self)
        self.mplPlayer.metaDataChanged.connect(self.showMetadata)
        vbox = QtWidgets.QVBoxLayout()
        btnOpen = QtWidgets.QPushButton("&Открыть файл...")
        btnOpen.clicked.connect(self.openFile)
        vbox.addWidget(btnOpen)
        # Создаем доступную только для чтения область редактирования,
        # в которую будет выводиться результат
        self.txtOutput = QtWidgets.QTextEdit()
        self.txtOutput.setReadOnly(True)
        vbox.addWidget(self.txtOutput)
        self.setLayout(vbox)
        self.resize(300, 250)

    def openFile(self):
        file = QtWidgets.QFileDialog.getOpenFileUrl(parent=self,
                        caption="Выберите звуковой файл",
                        filter="Звуковые файлы (*.mp3 *.ac3)")
        if file[1]:
            self.mplPlayer.setSource(file[0])

    def showMetadata(self):
        # Как только метаданные будут получены...
        # ...очищаем область редактирования...
        self.txtOutput.clear()
        # ...извлекаем объект с метаданными...
        md = self.mplPlayer.metaData()
        s = ""
        # ...перебираем их в цикле...
        for k in list(QtMultimedia.QMediaMetaData.Key):
            v = md.value(k)
            # ...проверяем, действительно ли существует значение
            # с таким ключом...
            if v:
                # ...если значение представляет собой список,
                # преобразуем его в строку...
                if v is list:
                    v = ", ".join(v)
```

```
                # ...формируем на основе значений текст...
                s += "<strong>" + k.name + "</strong>: " + str(v) + "<br>"
        # ...и выводим его в область редактирования
        self.txtOutput.setHtml(s)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec())
```

29.1.4. Видеопанель. Воспроизведение видео

Класс `QVideoWidget`, определенный в модуле `QtMultimediaWidgets`, представляет видеопанель, которая выводит воспроизводимое видео.

Иерархия наследования класса `QVideoWidget` выглядит следующим образом:

```
(QObject, QPaintDevice) — QWidget — QVideoWidget
```

А формат вызова его конструктора таков:

```
QVideoWidget([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Методов класс `QVideoWidget` поддерживает относительно немного:

- `setFullScreen(<Флаг>)` — если передать значение `True`, воспроизводящееся видео займет весь экран. Если передать значение `False`, оно займет лишь саму панель. Метод является слотом;
- `isFullScreen()` — возвращает `True`, если воспроизводящееся видео занимает весь экран, и `False`, если оно выводится лишь в самой панели;
- `setAspectRatioMode(<Режим>)` — задает режим управления соотношением сторон при масштабировании видео. В качестве результата указывается один из следующих элементов перечисления `AspectRatioMode` из модуля `QtCore.Qt`:
  - `IgnoreAspectRatio` — соотношение сторон при масштабировании видео не соблюдается, в результате чего видео может выводиться сжатым или вытянутым;
  - `KeepAspectRatio` — соотношение сторон соблюдается. Видео масштабируется так, чтобы полностью вместиться в панель или экран, но при этом какая-то часть панели или экрана может оказаться незанятой;
  - `KeepAspectRatioByExpanding` — соотношение сторон соблюдается. Видео масштабируется так, чтобы полностью занять панель (экран), но при этом какая-то часть видео может выйти за ее границы.

  Метод является слотом;
- `aspectRatioMode()` — возвращает обозначение режима управления соотношением сторон при масштабировании видео.

Класс `QVideoWidget` поддерживает следующие сигналы:

- `fullScreenChanged(<Флаг>)` — генерируется при переключении видео из обычного в полноэкранный режим или из полноэкранного в обычный. Через параметр передается значение `True`, если видео выводится на весь экран, или `False` — в противном случае;

- `aspectRatioModeChanged(<Режим AspectRatioMode>)` — генерируется при изменении режима управления соотношением сторон у видео. Через параметр доступно новое обозначение режима.

Задать видеопанель у мультимедийного проигрывателя можно вызовом метода `setVideoOutput(<Видеопанель QVideoWidget>)` класса `QMediaPlayer`.

Мы можем превратить написанную ранее программу аудиопроигрывателя в инструмент для воспроизведения видео (рис. 29.3). Для этого достаточно внести в ее код некоторые изменения (листинг 29.3) — добавленные и исправленные строки выделены здесь полужирным шрифтом.

Рис. 29.3. Видеопроигрыватель

Листинг 29.3. Видеопроигрыватель

```
from PyQt6 import QtCore, QtWidgets, QtMultimedia, QtMultimediaWidgets
. . .
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        . . .
        self.setWindowTitle("Видеопроигрыватель")
        . . .
        vbox.addWidget(btnOpen)
        vwg = QtMultimediaWidgets.QVideoWidget()
        vwg.setAspectRatioMode(QtCore.Qt.AspectRatioMode.KeepAspectRatio)
        self.mplPlayer.setVideoOutput(vwg)
        vbox.addWidget(vwg)
        self.sldPosition = QtWidgets.QSlider(QtCore.Qt.Orientation.Horizontal)
        . . .
        self.setLayout(vbox)
        self.resize(300, 300)
    . . .
```

```
def openFile(self):
    file = QtWidgets.QFileDialog.getOpenFileUrl(parent=self,
                    caption="Выберите видеофайл",
                    filter="Видеофайлы (*.avi *.mp4)")
    if file[1]:
        self.mplPlayer.setSource(file[0])
```

29.2. Запись мультимедиа

29.2.1. Транспорт

Класс `QMediaCaptureSession` представляет *транспорт* — механизм, который передает звук и видео, полученные с камеры и микрофона, кодировщику, непосредственно кодирующему мультимедиа и записывающему его в указанный файл. Иерархия наследования этого класса:

```
QObject - QMediaCaptureSession
```

Формат конструктора класса:

```
QMediaCaptureSession([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Класс `QMediaCaptureSession` поддерживает следующие методы:

- `setAudioInput(<Звуковой вход>)` — соединяет текущий транспорт с заданным звуковым входом (объект класса `QAudioInput`, описываемого в *разд. 29.2.2*);
- `setRecorder(<Кодировщик>)` — задает кодировщик звука и видео (объект класса `QMediaRecorder`, описываемого в *разд. 29.2.3*);
- `setCamera(<Камера>)` — соединяет транспорт с заданной камерой, с которой будет записываться видео (объект класса `QCamera`, описываемого в *разд. 29.2.5*);
- `setImageCapture(<Кодировщик>)` — задает кодировщик, записывающий статичные изображения (объект класса `QImageCapture`, описываемого в *разд. 29.2.6*);
- `setAudioOutput(<Устройство>)` — соединяет транспорт с заданным устройством вывода звука (объект класса `QAudioOutput`, описанного в *разд. 29.1.1*). Это устройство будет использоваться для прослушивания записываемого звука в целях контроля;
- `setVideoOutput(<Видеопанель QVideoWidget>)` — соединяет транспорт с заданной видеопанелью (см. *разд. 29.1.2*). Она будет использоваться для просмотра записываемого видео в целях контроля;
- `audioInput()` — возвращает устройство ввода звука, с которым соединен текущий транспорт, в виде объекта класса `QAudioInput`;
- `recorder()` — возвращает заданный у транспорта кодировщик звука и видео в виде объекта класса `QMediaRecorder`;
- `camera()` — возвращает камеру, с которой соединен текущий транспорт, в виде объекта класса `QCamera`;
- `imageCapture()` — возвращает кодировщик статичных изображений, заданный у транспорта, в виде объекта класса `QImageCapture`;
- `audioOutput()` — возвращает устройство вывода звука, с которым соединен текущий транспорт, в виде объекта класса `QAudioOutput`;

- `videoOutput()` — возвращает видеопанель, с которой соединен текущий транспорт, в виде объекта класса `QVideoWidget`.

Класс `QMediaCaptureSession` также поддерживает несколько не очень полезных для нас сигналов, описанных на странице **https://doc.qt.io/qt-6/qmediacapturesession.html**.

29.2.2. Звуковой вход

Класс `QAudioInput` служит для получения записываемого звука и полностью аналогичен классу `QAudioOutput`, описанному в *разд. 29.1.2*.

Звуковой вход изначально захватывает звук с устройства ввода, помеченного в настройках операционной системы как используемое по умолчанию. Однако можно указать ему захватывать звук с любого другого устройства ввода, доступного на текущей платформе.

Выяснить, какие устройства ввода звука существуют на текущей платформе, позволяют следующие статические методы класса `QMediaDevices`:

- `audioInputs()` — возвращает список доступных устройств ввода звука, представляемых объектами класса `QAudioDevice` (см. *разд. 29.1.2*):

```
>>> dlist = QtMultimedia.QMediaDevices.audioInputs()
>>> for d in dlist: print(d.description())
...
    Микрофон (SG330)
    Головной телефон (RB-HX220B Hands-Free AG Audio)
```

- `defaultAudioInput()` — возвращает устройство ввода звука, помеченное как используемое по умолчанию (объект класса `QAudioDevice`):

```
>>> QtMultimedia.QMediaDevices.defaultAudioInput().description()
    'Микрофон (SG330)'
```

Назначить у звукового входа другое устройство ввода звука можно вызовом метода `setDevice(<Звуковой вход QAudioDevice>)` класса `QAudioInput`:

```
ai = QtMultimedia.QAudioInput(parent=self)
dlist = QtMultimedia.QMediaDevices.audioInputs()
ai.setDevice(dlist[1])
```

Соединить транспорт со звуковым входом можно вызовом метода `setAudioInput(<Звуковой вход QAudioInput>)` класса `QMediaCaptureSession`.

29.2.3. Кодировщик звука и видео

Класс `QMediaRecorder` представляет кодировщик, кодирующий получаемые от транспорта звук и видео и записывающий их в заданный файл. Его иерархия наследования выглядит так:

```
QObject — QMediaRecorder
```

Формат конструктора этого класса:

```
QAudioRecorder([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Класс поддерживает следующие полезные методы (полный их список приведен на странице **https://doc.qt.io/qt-6/qmediarecorder.html**):

- `setOutputLocation(<Путь QUrl>)` — задает путь к файлу, в котором будет сохраняться записываемое мультимедиа;
- `setMediaFormat(<Формат>)` — задает формат результирующего файла, алгоритмы кодирования звука и видео в виде объекта класса `QMediaFormat` (будет описан далее);
- `setEncodingMode(<Режим>)` — задает режим кодирования в виде одного из следующих элементов перечисления `EncodingMode` из класса `QMediaRecorder`:
  - `ConstantQualityEncoding` — кодирование с постоянным качеством (реальный битрейт может немного отличаться от указанного);
  - `ConstantBitRateEncoding` — с постоянным битрейтом;
  - `AverageBitRateEncoding` — с переменным битрейтом (реальный битрейт может варьироваться относительно указанного);
  - `TwoPassEncoding` — кодирование в два прохода;
- `setQuality(<Качество>)` — задает качество результирующего мультимедиа в виде одного из следующих элементов перечисления `Quality` из класса `QMediaRecorder`: `VeryLowQuality` (очень низкое качество), `LowQuality` (низкое), `NormalQuality` (обычное — значение по умолчанию), `HighQuality` (высокое) и `VeryHighQuality` (очень высокое). Этот параметр принимается во внимание, если установлен режим кодирования `ConstantQualityEncoder`;
- `setAudioBitRate(<Битрейт>)` — задает битрейт кодируемого звука в виде целого числа в бит/с. Этот параметр принимается во внимание, если установлен режим кодирования, отличный от `ConstantQualityEncoder`;
- `setAudioChannelCount(<Количество>)` — задает количество каналов кодируемого звука. Если задать число `-1`, кодировщик сам выберет оптимальное количество каналов;
- `setAudioSampleRate(<Частота>)` — задает частоту дискретизации кодируемого звука в виде целого числа в герцах (Гц). Если задать число `-1`, кодировщик сам выберет оптимальную частоту дискретизации;
- `setVideoBitRate(<Битрейт>)` — задает битрейт кодируемого видео в виде целого числа в бит/с;
- `setVideoResolution()` — задает размеры изображения у кодируемого видео в пикселах. Форматы метода:

  ```
  setVideoResolution(<Размеры QSize>)
  setVideoResolution(<Ширина>, <Высота>)
  ```

  Если при вызове в первом формате передать методу объект класса `QSize` с нулевыми размерами, кодировщик сам выберет оптимальные размеры видео;
- `setVideoFrameRate(<Частота>)` — задает частоту кадров кодируемого видео в герцах (Гц). Если задать `0`, кодировщик сам выберет оптимальную частоту кадров;
- `isAvailable()` — возвращает `True`, если кодировщик готов к работе, и `False` — в противном случае;
- `record()` — запускает или возобновляет запись мультимедиа. Метод является слотом;
- `pause()` — приостанавливает запись. Метод является слотом;

- `stop()` — останавливает запись. Метод является слотом;
- `recorderState()` — возвращает текущее состояние кодировщика в виде значения одного из следующих элементов перечисления `RecorderState` из класса `QMediaRecorder`:
  - `StoppedState` — запись мультимедиа не выполняется;
  - `RecordingState` — идет запись;
  - `PausedState` — запись приостановлена;
- `duration()` — возвращает продолжительность записанного мультимедиа в виде целого числа в миллисекундах;
- `error()` — возвращает обозначение возникшей при записи мультимедиа ошибке в виде одного из следующих элементов перечисления `Error` из класса `QMediaRecorder`:
  - `NoError` — ошибки не возникло;
  - `ResourceError` — устройство захвата звука или видео отсутствует или недоступно;
  - `FormatError` — заданный формат файла, алгоритм кодирования звука или видео не поддерживается на текущей платформе;
  - `OutOfSpaceError` — не хватает места на диске для сохранения результирующего файла;
  - `LocationNotWritable` — местоположение результирующего файла не доступно для записи;
- `errorString()` — возвращает строковое описание возникшей ошибки.

Класс `QMediaRecorder` поддерживает следующие полезные сигналы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qmediarecorder.html**):

- `durationChanged(<Продолжительность>)` — генерируется при изменении продолжительности записанного мультимедиа. В параметре передается новое значение продолжительности, заданное целым числом в миллисекундах;
- `recorderStateChanged(<Состояние RecorderState>)` — генерируется при изменении состояния записи мультимедиа. В параметре передается обозначение нового состояния;
- `errorOccured(<Ошибка Error>, <Описание>)` — генерируется при возникновении ошибки. В параметрах передаются обозначение возникшей ошибки и ее строковое описание;
- `actualLocationChanged(<Путь QUrl>)` — генерируется при указании другого файла для сохранения записанного мультимедиа. В параметре передается путь к новому файлу.

Соединить транспорт с кодировщиком можно вызовом метода `setRecorder(<Кодировщик QMediaRecorder>)` класса `QMediaCaptureSession`.

29.2.4. Указание форматов кодирования. Запись звука

Класс `QMediaFormat` служит для указания формата файла, в котором будет сохраняться записываемое мультимедиа, и алгоритмов кодирования звука и видео. Форматы конструктора этого класса:

```
QMediaFormat([<Формат файла>=FileFormat.UnspecifiedFormat])
QMediaFormat(<Исходный объект QMediaFormat>)
```

В первом формате вызова формат файла задается в виде одного из следующих элементов перечисления `FileFormat` из класса `QMediaFormat`: `WMA`, `AAC`, `Matroska`, `WMV`, `MP3`, `Wave`, `Ogg`,

`MPEG4`, `AVI`, `QuickTime`, `WebM`, `Mpeg4Audio`, `FLAC` или `UnspecifiedFormat` (формат не задан — значение по умолчанию).

Второй формат вызова конструктора создает копию указанного объекта.

Класс поддерживает следующие полезные методы (их полный список приведен на странице **https://doc.qt.io/qt-6/qmediaformat.html**):

- `supportedFileFormats(<Операция>)` — возвращает список обозначений форматов файлов, с которыми можно выполнить заданную операцию. Параметр `<Операция>` указывается в виде следующего элемента перечисления `ConversionMode` из класса `QmediaFormat`: `Encode` (кодирование, т. е. запись) или `Decode` (декодирование, т. е. воспроизведение). Пример получения форматов файлов, запись в которые поддерживается на компьютере одного из авторов:

```
>>> mf = QtMultimedia.QMediaFormat()
>>> flist = mf.supportedFileFormats(
...                     QtMultimedia.QMediaFormat.ConversionMode.Encode)
>>> for f in flist: print(f.name, end=" ")
...
    Mpeg4Audio MP3 AAC WMA WMV MPEG4 FLAC
```

- `supportedAudioCodecs(<Операция ConversionMode>)` — возвращает список обозначений алгоритмов кодирования звука, с которыми можно выполнить заданную операцию. Обозначения алгоритмов представляются следующими элементами перечисления `AudioCodec` из класса `QMediaFormat`: `WMA`, `AC3`, `AAC`, `ALAC`, `DolbyTrueHD`, `EAC3`, `MP3`, `Wave`, `Vorbis`, `FLAC`, `Opus` или `Unspecified` (алгоритм не задан). Пример:

```
>>> flist = mf.supportedAudioCodecs(
...                     QtMultimedia.QMediaFormat.ConversionMode.Encode)
>>> for f in flist: print(f.name, end=" ")
...
    FLAC AAC MP3 AC3 WMA ALAC
```

- `supportedVideoCodecs(<Операция ConversionMode>)` — возвращает список обозначений алгоритмов кодирования видео, с которыми можно выполнить заданную операцию. Обозначения алгоритмов представляются следующими элементами перечисления `VideoCodec` из класса `QMediaFormat`: `VP8`, `MPEG2`, `MPEG1`, `WMV`, `H265`, `H264`, `MPEG4`, `AV1`, `MotionJPEG`, `VP9`, `Theora` или `Unspecified` (алгоритм не указан). Пример:

```
>>> flist = mf.supportedVideoCodecs(
...                     QtMultimedia.QMediaFormat.ConversionMode.Encode)
>>> for f in flist: print(f.name, end=" ")
...
    H264 WMV
```

- `setFileFormat(<Формат FileFormat>)` — задает формат файла;
- `setAudioCodec(<Алгоритм AudioCodec>)` — задает алгоритм кодирования звука;
- `setVideoCodec(<Алгоритм VideoCodec>)` — задает алгоритм кодирования видео.

Напишем программу-диктофон (листинг 29.4), записывающую звук с устройства ввода по умолчанию и сохраняющую его в файле формата MP3 в минимальном качестве (рис. 29.4).

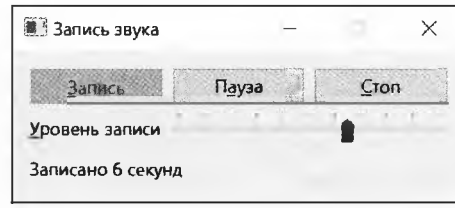


Рис. 29.4. Программа-диктофон

Листинг 29.4. Программа-диктофон

```
from PyQt6 import QtCore, QtWidgets, QtMultimedia
import sys, os

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Запись звука")
        # Создаем транспорт, звуковой вход и кодировщик
        mcs = QtMultimedia.QMediaCaptureSession(parent=self)
        self.aInput = QtMultimedia.QAudioInput(parent=self)
        self.aInput.setVolume(1.0)
        mcs.setAudioInput(self.aInput)
        self.ardRecorder = QtMultimedia.QMediaRecorder(parent=self)
        mcs.setRecorder(self.ardRecorder)
        # Звук будет сохраняться в файле record.mp3, находящемся
        # в той же папке, где хранится программа
        fn = QtCore.QUrl.fromLocalFile(os.path.abspath("record.mp3"))
        self.ardRecorder.setOutputLocation(fn)
        # Указываем формат файла MP3
        mf = QtMultimedia.QMediaFormat(
                        QtMultimedia.QMediaFormat.FileFormat.MP3)
        mf.setAudioCodec(QtMultimedia.QMediaFormat.AudioCodec.MP3)
        self.ardRecorder.setMediaFormat(mf)
        # Задаем параметры кодирования звука
        self.ardRecorder.setQuality(
                QtMultimedia.QMediaRecorder.Quality.LowQuality)
        self.ardRecorder.setEncodingMode(
            QtMultimedia.QMediaRecorder.EncodingMode.ConstantQualityEncoding)
        self.ardRecorder.setAudioChannelCount(1)
        self.ardRecorder.setAudioSampleRate(-1)
        self.ardRecorder.recorderStateChanged.connect(self.initRecorder)
        self.ardRecorder.durationChanged.connect(self.showDuration)
        # Создаем компоненты для запуска, приостановки и остановки
        # записи звука и регулирования его уровня
        vbox = QtWidgets.QVBoxLayout()
        hbox = QtWidgets.QHBoxLayout()
```

```
        self.btnRecord = QtWidgets.QPushButton("&Запись")
        self.btnRecord.clicked.connect(self.ardRecorder.record)
        hbox.addWidget(self.btnRecord)
        self.btnPause = QtWidgets.QPushButton("П&ауза")
        self.btnPause.clicked.connect(self.ardRecorder.pause)
        self.btnPause.setEnabled(False)
        hbox.addWidget(self.btnPause)
        self.btnStop = QtWidgets.QPushButton("&Стоп")
        self.btnStop.clicked.connect(self.ardRecorder.stop)
        self.btnStop.setEnabled(False)
        hbox.addWidget(self.btnStop)
        vbox.addLayout(hbox)
        hbox = QtWidgets.QHBoxLayout()
        lblVolume = QtWidgets.QLabel("&Уровень записи")
        hbox.addWidget(lblVolume)
        self.sldVolume = QtWidgets.QSlider(QtCore.Qt.Orientation.Horizontal)
        self.sldVolume.setRange(0, 100)
        self.sldVolume.setTickPosition(
                        QtWidgets.QSlider.TickPosition.TicksAbove)
        self.sldVolume.setTickInterval(10)
        self.sldVolume.setValue(100)
        lblVolume.setBuddy(self.sldVolume)
        self.sldVolume.valueChanged.connect(self.setVolume)
        hbox.addWidget(self.sldVolume)
        vbox.addLayout(hbox)
        # Создаем надпись, в которую будет выводиться состояние программы
        self.lblStatus = QtWidgets.QLabel("Готово")
        vbox.addWidget(self.lblStatus)
        self.setLayout(vbox)
        self.resize(300, 100)

# В зависимости от состояния записи звука делаем нужные
# кнопки доступными или, напротив, недоступными и выводим
# соответствующий текст в надписи
def initRecorder(self, state):
    match state:
        case QtMultimedia.QMediaRecorder.RecorderState.RecordingState:
            self.btnRecord.setEnabled(False)
            self.btnPause.setEnabled(True)
            self.btnStop.setEnabled(True)
            self.lblStatus.setText("Запись")
        case QtMultimedia.QMediaRecorder.RecorderState.PausedState:
            self.btnRecord.setEnabled(True)
            self.btnPause.setEnabled(False)
            self.lblStatus.setText("Пауза")
        case QtMultimedia.QMediaRecorder.RecorderState.StoppedState:
            self.btnRecord.setEnabled(True)
            self.btnPause.setEnabled(False)
            self.btnStop.setEnabled(False)
            self.lblStatus.setText("Готово")
```

```
    # Выводим продолжительность записанного звука
    def showDuration(self, duration):
        self.lblStatus.setText("Записано " + str(duration // 1000) +
                               " секунд")

    def setVolume(self, value):
        self.aInput.setVolume(value / 100)

    # При закрытии окна останавливаем запись
    def closeEvent(self, e):
        self.ardRecorder.stop()
        e.accept()
        QtWidgets.QWidget.closeEvent(self, e)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec())
```

29.2.5. Камера. Запись видео

Класс `QCamera` представляет виртуальную камеру, соединенную с указанным устройством захвата видео (например, с веб-камерой). Иерархия наследования этого класса:

```
QObject — QCamera
```

Форматы конструктора этого класса:

```
QCamera([parent=None])
QCamera(<Местоположение камеры>[, parent=None])
QCamera(<Устройство захвата видео>[, parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Первый формат создает камеру, которая будет захватывать видео с устройства, указанного в настройках операционной системы как используемое по умолчанию.

Второй формат позволяет выбрать устройство захвата видео, задав его местоположение. Последнее указывается в виде одного из следующих элементов перечисления `Position` из класса `QCameraDevice`:

- `UnspecifiedPosition` — местоположение устройства не указано, и будет использовано устройство по умолчанию;
- `BackFace` — устройство захвата видео, расположенное сзади (тыловая камера);
- `FrontFace` — устройство захвата видео, расположенное спереди (фронтальная камера).

Пример:

```
cam = QtMultimedia QCamera(QtMultimedia.QCameraDevice.Position.FrontFace)
```

Третий формат позволяет указать конкретное устройство, представленное объектом класса `QCameraDevice`.

Выяснить, какие устройства захвата видео доступны на текущей платформе, позволяют следующие статические методы класса `QMediaDevices`:

- `videoInputs()` — возвращает список доступных устройств захвата видео, представляемых объектами класса `QCameraDevice` (будет описан далее):

```
>>> dlist = QtMultimedia.QMediaDevices.videoInputs()
>>> for d in dlist: print(d.description())
...
    SG330
```

- `defaultVideoInput()` — возвращает устройство захвата видео, помеченное как используемое по умолчанию (объект класса `QCameraDevice`):

```
>>> QtMultimedia.QMediaDevices.defaultVideoInput().description()
    'SG330'
```

Пример:

```
dlist = QtMultimedia.QMediaDevices.videoInputs()
cam = QtMultimedia.QCamera(dlist[1])
```

Назначить у камеры другое устройство захвата видео можно вызовом метода `setCameraDevice(<Камера QCameraDevice>)` класса `QCamera`:

```
cam = QtMultimedia.QCamera(parent=self)
dlist = QtMultimedia.QMediaDevices.videoInputs()
cam.setCameraDevice(dlist[1])
```

Класс `QCameraDevice` представляет доступное на текущей платформе устройство для захвата видео. Метод `description()` этого класса возвращает строковое название текущего устройства. Полное описание класса `QCameraDevice` приведено на странице **https://doc.qt.io/qt-6/qcameradevice.html**.

Класс `QCamera` поддерживает много методов, позволяющих, в частности, узнать характеристики устройства захвата видео и указать некоторые его параметры, и ряд сигналов. Полное описание этого класса приведено на странице **https://doc.qt.io/qt-6/qcamera.html**.

Соединить транспорт с камерой можно вызовом метода `setCamera(<Камера QCamera>)` класса `QMediaCaptureSession`.

После присоединения камеры к транспорту необходимо запустить захват видео этой камерой, вызвав метод `start()` класса `QCamera`.

Напишем простую программу для записи видео (листинг 29.5), которая станет сохранять записанное видео в минимальном качестве в файле формата MPEG4, кодировать звук алгоритмом AAC, а видео — алгоритмом H264 (рис. 29.5). Ради простоты не будем реализовывать в ней регулировку уровня записи звука.

Рис. 29.5. Программа для записи видео

Листинг 29.5. Программа для записи видео

```
from PyQt6 import QtCore, QtWidgets, QtMultimedia, QtMultimediaWidgets
import sys, os

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Запись видео")
        # Создаем транспорт, звуковой вход, камеру и кодировщик
        mcs = QtMultimedia.QMediaCaptureSession(parent=self)
        self.aInput = QtMultimedia.QAudioInput(parent=self)
        self.aInput.setVolume(1.0)
        mcs.setAudioInput(self.aInput)
        cam = QtMultimedia.QCamera(parent=self)
        mcs.setCamera(cam)
        self.ardRecorder = QtMultimedia.QMediaRecorder(parent=self)
        mcs.setRecorder(self.ardRecorder)
        # Запускаем захват видео созданной ранее камерой
        cam.start()
        # Видео будет сохраняться в файле movie.mp4, находящемся
        # в той же папке, где хранится программа
        fn = QtCore.QUrl.fromLocalFile(os.path.abspath("movie.mp4"))
        self.ardRecorder.setOutputLocation(fn)
        mf = QtMultimedia.QMediaFormat(
                        QtMultimedia.QMediaFormat.FileFormat.MPEG4)
        mf.setAudioCodec(QtMultimedia.QMediaFormat.AudioCodec.AAC)
        mf.setVideoCodec(QtMultimedia.QMediaFormat.VideoCodec.H264)
        self.ardRecorder.setMediaFormat(mf)
        # Задаем параметры кодирования звука и видео
        self.ardRecorder.setQuality(
                QtMultimedia.QMediaRecorder.Quality.LowQuality)
        self.ardRecorder.setEncodingMode(
            QtMultimedia.QMediaRecorder.EncodingMode.ConstantQualityEncoding)
        self.ardRecorder.setAudioChannelCount(1)
        self.ardRecorder.setAudioSampleRate(-1)
        self.ardRecorder.setVideoResolution(QtCore.QSize())
        self.ardRecorder.recorderStateChanged.connect(self.initRecorder)
        self.ardRecorder.durationChanged.connect(self.showDuration)
        vbox = QtWidgets.QVBoxLayout()
        # Создаем контрольную видеопанель и связываем ее с транспортом
        vwg = QtMultimediaWidgets.QVideoWidget()
        vwg.setAspectRatioMode(QtCore.Qt.AspectRatioMode.KeepAspectRatio)
        mcs.setVideoOutput(vwg)
        vbox.addWidget(vwg, stretch=1)
        # Создаем компоненты для запуска, приостановки и остановки
        # записи видео
        hbox = QtWidgets.QHBoxLayout()
        self.btnRecord = QtWidgets.QPushButton("&Запись")
```

```
        self.btnRecord.clicked.connect(self.ardRecorder.record)
        hbox.addWidget(self.btnRecord)
        self.btnPause = QtWidgets.QPushButton("П&ауза")
        self.btnPause.clicked.connect(self.ardRecorder.pause)
        self.btnPause.setEnabled(False)
        hbox.addWidget(self.btnPause)
        self.btnStop = QtWidgets.QPushButton("&Стоп")
        self.btnStop.clicked.connect(self.ardRecorder.stop)
        self.btnStop.setEnabled(False)
        hbox.addWidget(self.btnStop)
        vbox.addLayout(hbox)
        vbox.addLayout(hbox)
        # Создаем надпись, в которую будет выводиться состояние программы
        self.lblStatus = QtWidgets.QLabel("Готово")
        vbox.addWidget(self.lblStatus)
        self.setLayout(vbox)
        self.resize(300, 280)

    # В зависимости от состояния записи видео делаем нужные
    # кнопки доступными или, напротив, недоступными и выводим
    # соответствующий текст в надписи
    def initRecorder(self, state):
        match state:
            case QtMultimedia.QMediaRecorder.RecorderState.RecordingState:
                self.btnRecord.setEnabled(False)
                self.btnPause.setEnabled(True)
                self.btnStop.setEnabled(True)
                self.lblStatus.setText("Запись")
            case QtMultimedia.QMediaRecorder.RecorderState.PausedState:
                self.btnRecord.setEnabled(True)
                self.btnPause.setEnabled(False)
                self.lblStatus.setText("Пауза")
            case QtMultimedia.QMediaRecorder.RecorderState.StoppedState:
                self.btnRecord.setEnabled(True)
                self.btnPause.setEnabled(False)
                self.btnStop.setEnabled(False)
                self.lblStatus.setText("Готово")

    # Выводим продолжительность записанного видео
    def showDuration(self, duration):
        self.lblStatus.setText("Записано " + str(duration // 1000) + " секунд")

    # При закрытии окна останавливаем запись
    def closeEvent(self, e):
        self.ardRecorder.stop()
        e.accept()
        QtWidgets.QWidget.closeEvent(self, e)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
```

```
window.show()
sys.exit(app.exec())
```

29.2.6. Кодировщик статичных изображений. Захват фото

Класс `QImageCapture` представляет кодировщик статичных изображений, который получает с камеры отдельные кадры и сохраняет их в графических файлах. Иерархия наследования:

```
QObject — QImageCapture
```

Формат конструктора класса:

```
QImageCapture([parent=self])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Класс `QImageCapture` поддерживает следующие методы (их полный список приведен на странице **https://doc.qt.io/qt-6/qimagecapture.html**):

- `setFileFormat(<Формат>)` — задает формат результирующих файлов. В качестве параметра указывается один из следующих элементов перечисления `FileFormat` из класса `QImageCapture`: `JPEG`, `PNG`, `Tiff`, `WebP` или `UnspecifiedFormat` (формат не указан — значение по умолчанию);
- `setQuality(<Качество>)` — задает качество результирующего изображения в виде одного из следующих элементов перечисления `Quality` из класса `QImageCapture`: `VeryLowQuality` (очень низкое качество), `LowQuality` (низкое), `NormalQuality` (обычное — значение по умолчанию), `HighQuality` (высокое) и `VeryHighQuality` (очень высокое);
- `setResolution()` — задает размеры результирующего изображения в пикселах. Форматы метода:

  ```
  setResolution(<Размеры QSize>)
  setResolution(<Ширина>, <Высота>)
  ```

  Если при вызове в первом формате передать методу объект класса `QSize` с нулевыми размерами, кодировщик сам выберет оптимальные размеры изображения;
- `isAvailable()` — возвращает `True`, если кодировщик готов к работе, и `False` — в противном случае;
- `captureToFile(<Путь к файлу>)` — захватывает статичный кадр и сохраняет его в файле с заданным путем. Метод является слотом.

  Вызов этого метода лишь запускает процесс захвата и сохранения статичного изображения. После этого в течение какого-то времени кодировщик будет обрабатывать захваченный кадр, и попытка выполнить захват нового кадра вызовом метода `captureToFile()` приведет к возникновению ошибки с типом `Error.NotReadyError`.

  Как только файл с захваченным кадром будет сохранен, будет сгенерирован сигнал `imageSaved()`;
- `capture()` — захватывает статичный кадр и сохраняет его в памяти в виде объекта класса `QImage`. Это изображение впоследствии можно получить в обработчике сигнала `imageCaptured()`. Метод является слотом
- `isReadyForCapture()` — возвращает `True`, если кодировщик в текущий момент готов к захвату очередного статичного кадра, и `False` — если он еще обрабатывает кадр, захваченный ранее;

- `error()` — возвращает обозначение возникшей при захвате кадра ошибки в виде одного из следующих элементов перечисления `Error` из класса `QImageCapture`:
  - `NoError` — ошибки не возникло;
  - `NotReadyError` — кодировщик не готов к работе, поскольку уже выполняет захват и обработку статичного кадра;
  - `ResourceError` — устройство захвата видео отсутствует или недоступно;
  - `FormatError` — заданный формат файла не поддерживается на текущей платформе;
  - `OutOfSpaceError` — не хватает места на диске для сохранения результирующего файла;
  - `NotSupportedFeatureError` — выбранное устройство захвата видео не поддерживает захват статичных изображений;
- `errorString()` — возвращает строковое описание возникшей ошибки.

Класс `QImageCapture` поддерживает следующие полезные сигналы (их полный список приведен на странице **https://doc.qt.io/qt-6/qimagecapture.html**):

- `imageCaptured(<Идентификатор>, <Изображение QImage>)` — генерируется сразу после захвата статичного изображения и перед его сохранением в файле (если был выполнен вызовом метода `captureToFile()`). Через параметры передаются целочисленный идентификатор захваченного изображения и само это изображение;
- `imageSaved(<Идентификатор>, <Путь к файлу>)` — генерируется сразу после сохранения захваченного кадра в файле. Через параметры передаются целочисленный идентификатор захваченного изображения и путь к созданному файлу;
- `readyForCaptureChanged(<Флаг>)` — генерируется при изменении состояния готовности кодировщика к захвату кадра. В параметре передается значение `True`, если кодировщик готов к захвату кадра, и `False` — если не готов;
- `errorOccured(<Ошибка Error>, <Описание>)` — генерируется при возникновении ошибки. В параметрах передаются обозначение возникшей ошибки и ее строковое описание.

Присоединить кодировщик статичных изображений к транспорту можно вызовом метода `setImageCapture(<Кодировщик QImageCapture>)` класса `QMediaCaptureSession`.

В листинге 29.6 приведен код программы-«фотоаппарата», которая захватывает статичные кадры и сохраняет их в файлах формата JPEG, чьи имена представляют собой последовательно увеличивающиеся целые числа.

Листинг 29.6. Программа-«фотоаппарат»

```
from PyQt6 import QtCore, QtWidgets, QtMultimedia, QtMultimediaWidgets
import sys, os

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        # Создаем счетчик захваченных изображений
        self.counter = 0
        self.setWindowTitle("Захват фото")
```

```
        # Создаем транспорт, камеру и кодировщик
        mcs = QtMultimedia.QMediaCaptureSession(parent=self)
        cam = QtMultimedia.QCamera(parent=self)
        mcs.setCamera(cam)
        self.icCapture = QtMultimedia.QImageCapture(parent=self)
        mcs.setImageCapture(self.icCapture)
        cam.start()
        # Задаем параметры сохранения изображений
        self.icCapture.setFileFormat(
                        QtMultimedia.QImageCapture.FileFormat.JPEG)
        self.icCapture.setQuality(
                        QtMultimedia.QImageCapture.Quality.HighQuality)
        self.icCapture.setResolution(QtCore.QSize())
        vbox = QtWidgets.QVBoxLayout()
        # Создаем контрольную видеопанель и связываем ее с транспортом
        vwg = QtMultimediaWidgets.QVideoWidget()
        vwg.setAspectRatioMode(QtCore.Qt.AspectRatioMode.KeepAspectRatio)
        mcs.setVideoOutput(vwg)
        vbox.addWidget(vwg, stretch=1)
        # Создаем кнопку для захвата изображения
        self.btnCapture = QtWidgets.QPushButton("&Сделать фото")
        self.btnCapture.clicked.connect(self.captureImage)
        self.icCapture.readyForCaptureChanged.connect(
                        self.btnCapture.setEnabled)
        vbox.addWidget(self.btnCapture)
        self.setLayout(vbox)
        self.resize(320, 240)

    def captureImage(self):
        self.counter += 1
        self.icCapture.captureToFile(
               os.path.abspath(str(self.counter) + ".jpg"))

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec())
```

29.3. Воспроизведение звуковых эффектов

Звуковой эффект — это короткий звук, сообщающий пользователю о наступлении какого-либо события (например, об окончании выполнения длительной операции).

Класс `QSoundEffect` представляет звуковой эффект. Его иерархия наследования:

```
QObject — QSoundEffect
```

Конструктор этого класса имеет следующие форматы вызова:

```
QSoundEffect([parent=None])
QSoundEffect(<Устройство вывода звука QAudioDevice>[, parent=None])
```

Первый формат соединяет создаваемый эффект с устройством для вывода звука, используемым по умолчанию. Второй формат позволяет явно указать используемое устройство вывода звука.

В параметре `parent` может быть указана ссылка на родительский компонент.

Класс `QSoundEffect` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qsoundeffect.html**):

- `setSource(<Путь QUrl>)` — задает файл-источник воспроизводимого звука;
- `setVolume(<Громкость>)` — задает громкость воспроизводимого звука в виде вещественного числа от `0.0` (звук отключен) до `1.0` (максимальная громкость — значение по умолчанию);
- `setLoopCount(<Количество>)` — задает количество повторений звука при воспроизведении. Значение `0` или `1` предписывает воспроизвести звук однократно, а значение `-2` — воспроизводить его бесконечное количество раз;
- `setMuted(<Флаг>)` — если передать значение `True`, звук при воспроизведении будет отключен. Значение `False` вновь включает звук;
- `setAudioDevice(<Устройство вывода звука QAudioDevice>)` — связывает текущий эффект с заданным устройством вывода звука;
- `play()` — запускает воспроизведение звука. Метод является слотом;
- `stop()` — останавливает воспроизведение звука. Метод является слотом;
- `isLoaded()` — возвращает `True`, если звук загружен из указанного в методе `setSource()` файла, и `False` — в противном случае;
- `isPlaying()` — возвращает `True`, если звук в текущий момент воспроизводится, и `False` — в противном случае;
- `loopsRemaining()` — возвращает количество оставшихся повторов воспроизведения звука или значение `-2`, если было задано бесконечное воспроизведение;
- `volume()` — возвращает текущее значение громкости звука в виде вещественного числа от `0.0` до `1.0`;
- `isMuted()` — возвращает `True`, если воспроизводящийся звук в текущий момент отключен, и `False` — в противном случае;
- `status()` — возвращает текущее состояние звука в виде значения одного из следующих элементов перечисления `Status` из класса `QSoundEffect`:
  - `Null` — источник звука не задан;
  - `Loading` — идет загрузка звука из файла-источника;
  - `Ready` — звук загружен и готов к воспроизведению;
  - `Error` — в процессе загрузки звука возникла ошибка;
- `supportedMimeTypes()` — статический, возвращает список MIME-типов поддерживаемых форматов звука:

```
>>> for f in QtMultimedia.QSoundEffect.supportedMimeTypes():
...     print(f, end=" ")
...
    audio/x-wav audio/wav audio/wave audio/x-pn-wav
```

Класс `QSoundEffect` поддерживает следующие полезные сигналы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qsoundeffect.html**):

- `statusChanged()` — генерируется при изменении состояния звука;
- `loadedChanged()` — генерируется при изменении состояния загрузки звука из файла-источника;
- `playingChanged()` — генерируется при изменении состояния воспроизведения звука;
- `loopsRemainingChanged()` — генерируется при каждом повторении воспроизводимого звука;
- `volumeChanged()` — генерируется при изменении громкости звука;
- `mutedChanged()` — генерируется при отключении звука или, наоборот, восстановлении его громкости вызовом метода `setMuted()`.

Примером к этому разделу станет небольшая программа, воспроизводящая звук, в зависимости от выбора пользователя: единожды, десять раз или бесконечно (листинг 29.7).

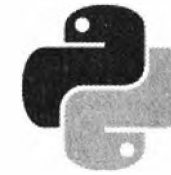
Листинг 29.7. Воспроизведение звуковых эффектов

```
from PyQt6 import QtCore, QtWidgets, QtMultimedia
import sys, os

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Звуковые эффекты")
        # Инициализируем подсистему вывода звуковых эффектов
        self.sndEffect = QtMultimedia.QSoundEffect()
        # Задаем файл-источник
        fn = QtCore.QUrl.fromLocalFile(os.path.abspath("effect.wav"))
        self.sndEffect.setSource(fn)
        self.sndEffect.loopsRemainingChanged.connect(self.showCount)
        self.sndEffect.playingChanged.connect(self.clearCount)
        vbox = QtWidgets.QVBoxLayout()
        # Создаем кнопки для запуска воспроизведения звука
        lblPlay = QtWidgets.QLabel("Воспроизвести...")
        vbox.addWidget(lblPlay)
        btnOnce = QtWidgets.QPushButton("...&один раз")
        btnOnce.clicked.connect(self.playOnce)
        vbox.addWidget(btnOnce)
        btnTen = QtWidgets.QPushButton("...&десять раз")
        btnTen.clicked.connect(self.playTen)
        vbox.addWidget(btnTen)
        btnInfinite = QtWidgets.QPushButton(
                              "...&бесконечное количество раз")
        btnInfinite.clicked.connect(self.playInfinite)
        vbox.addWidget(btnInfinite)
        btnStop = QtWidgets.QPushButton("&Стоп")
```

```
        btnStop.clicked.connect(self.sndEffect.stop)
        vbox.addWidget(btnStop)
        self.lblStatus = QtWidgets.QLabel("")
        vbox.addWidget(self.lblStatus)
        self.setLayout(vbox)
        self.resize(300, 100)

    def playOnce(self):
        self.sndEffect.setLoopCount(1)
        self.sndEffect.play()

    def playTen(self):
        self.sndEffect.setLoopCount(10)
        self.sndEffect.play()

    def playInfinite(self):
        self.sndEffect.setLoopCount(-2)
        self.sndEffect.play()

    # Выводим количество повторений воспроизведения эффекта
    def showCount(self):
        self.lblStatus.setText("Воспроизведено " +
                               str(self.sndEffect.loopCount() -
                               self.sndEffect.loopsRemaining()) + " раз")

    # Если воспроизведение закончено, очищаем выведенное ранее
    # количество повторений эффекта
    def clearCount(self):
        if not self.sndEffect.isPlaying():
            self.lblStatus.setText("")

    def closeEvent(self, e):
        self.sndEffect.stop()
        e.accept()
        QtWidgets.QWidget.closeEvent(self, e)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec())
```

Примечание

Помимо описанных здесь возможностей, PyQt поддерживает доступ к устройствам воспроизведения звука на низком уровне, а также получение непосредственно массива звуковых и видеоданных с целью их анализа и обработки. Соответствующие программные инструменты описаны в документации по этой библиотеке.



ГЛАВА 30

Печать документов

PyQt поддерживает ряд развитых средств, позволяющих выполнить печать документов, их предварительный просмотр и экспорт в формат PDF.

Все описанные в этой главе классы определены в модуле `QtPrintSupport`, если не указано иное.

30.1. Основные средства печати

30.1.1. Класс *QPrinter*

Класс `QPrinter` представляет принтер, связанный с указанным физическим устройством печати (например, физическим принтером). Его иерархия наследования такова:

```
QPaintDevice - QPagedPaintDevice - QPrinter
```

Конструктор класса `QPrinter` поддерживает следующие форматы вызова:

```
QPrinter([mode=PrinterMode.ScreenResolution])
QPrinter(<Устройство печати>[, mode=PrinterMode.ScreenResolution])
```

Первый формат создает принтер, связанный с физическим устройством печати, которое помечено в настройках операционной системы как используемое по умолчанию. В параметре `mode` может быть указан режим работы принтера, заданный в виде одного из следующих элементов перечисления `PrinterMode` из класса `QPrinter`:

- `ScreenResolution` — печать в разрешении экрана. Позволяет вывести документ максимально быстро, но в худшем качестве;
- `HighResolution` — печать в разрешении принтера. Печать выполняется качественнее, но медленнее.

Второй формат создает принтер, связанный с указанным устройством печати. Последнее задается объектом класса `QPrinterInfo`, речь о котором пойдет далее.

Поскольку класс `QPrinter` является производным от класса `QPagedPaintDevice`, который, в свою очередь, наследует рассмотренный в *главе 25* класс `QPaintDevice`, для вывода документов на печать используются методы последнего.

Класс `QPrinter` поддерживает очень много методов, из которых мы рассмотрим лишь наиболее полезные (полный их список можно найти на страницах **https://doc.qt.io/qt-6/qprinter.html** и **https://doc.qt.io/qt-6/qpaintdevice.html**).

- `setPrinterName(<Имя принтера>)` — соединяет текущий принтер с устройством печати с заданным в виде строки именем. Если передать пустую строку, будет выполнено соединение со встроенной в PyQt подсистемой вывода документов в формате PDF;
- `printerName()` — возвращает строку с именем устройства печати, с которым соединен текущий принтер;
- `setOutputFileName(<Путь к файлу>)` — задает путь к файлу, в который будет выведен печатаемый документ (вместо печати на бумаге). Если файл имеет расширение pdf, будет выполнен вывод в формате PDF. В противном случае файл будет сохранен в формате, установленном в вызове метода `setOutputFormat()`. Чтобы отключить вывод документа в файл, следует вызвать этот метод, передав ему в качестве параметра пустую строку;
- `outputFileName()` — возвращает путь к файлу, в который будет выведен документ вместо печати на бумаге;
- `setOutputFormat(<Формат вывода>)` — задает формат вывода документа при записи его в файл. В качестве параметра передается значение одного из следующих элементов перечисления `OutputFormat` из класса `QPrinter`:
  - `NativeFormat` — внутренний формат устройства печати. Этот режим автоматически устанавливается при вызове метода `setPrinterName()` с указанием имени существующего устройства печати. Если текущий принтер подключен к подсистеме вывода в формате PDF, будет выполнено соединение с устройством печати по умолчанию;
  - `PdfFormat` — формат PDF. Этот режим автоматически устанавливается при вызове метода `setPrinterName()` с указанием пустой строки;
- `outputFormat()` — возвращает обозначение формата вывода документа в виде элемента перечисления `OutputFormat`;
- `isValid()` — возвращает `True`, если связанное с принтером устройство печати действительно установлено в системе и готово к работе, и `False` — в противном случае.
- `setPageSize(<Размер>)` — задает размер страницы в виде объекта класса `QPageSize` из модуля `QtGui` (будет описан далее). Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае. Пример задания размера бумаги A4:

```
ps = QtGui.QPageSize(QtGui.QPageSize.QPageSizeId.A4)
printer.setPageSize(ps)
```

- `setPageOrientation(<Ориентация>)` — задает ориентацию страницы в виде элемента `Portrait` (портретная) или `Landscape` (ландшафтная) перечисления `Orientation` из класса `QPageLayout` (модуль `QtGui`). Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setPageMargins(<Отступы QMarginsF>[, units=Unit.Millimeter])` — задает отступы от краев страницы. Параметр `units` указывает единицу измерения отступов в виде одного из следующих элементов перечисления `Unit` из класса `QPageLayout`:
  - `Millimeter` — миллиметры;
  - `Point` — пункты;
  - `Inch` — дюймы;
  - `Pica` — пики;

- Didot — дидо (0,375 мм);
- Cicero — цицеро (4,5 мм).

Возвращает True, если операция увенчалась успехом, и False — в противном случае. Вот пример задания отступов в 1 пункт со всех сторон страницы:

```
m = QtCore.QMargins(1, 1, 1, 1)
printer.setPageMargins(m, units=QtGui.QPageLayout.Unit.Point)
```

- setPageLayout(<Параметры>) — задает сразу все параметры страницы (размер, ориентацию и отступы от краев) в виде объекта класса QPageLayout (будет описан далее). Возвращает True, если операция увенчалась успехом, и False — в противном случае;
- pageLayout() — возвращает параметры страницы в виде объекта класса QPageLayout;
- setCopyCount(<Количество>) — задает количество копий печатаемого документа;
- copyCount() — возвращает количество копий печатаемого документа;
- setCollateCopies(<Флаг>) — если передано значение True, каждая копия документа будет отпечатана полностью, прежде чем начнется печать следующей копии. Если передать значение False, сначала будут отпечатаны все копии первой страницы, потом все копии второй и т. д.;
- collateCopies() — возвращает True, если каждая копия документа печатается полностью, и False — в противном случае;
- setDuplex(<Режим>) — задает режим двусторонней печати в виде одного из следующих элементов перечисления DuplexMode из класса QPrinter:
  - DuplexNone — односторонняя печать;
  - DuplexAuto — двусторонняя печать с автоматическим определением стороны листа, вокруг которой следует его перевернуть;
  - DuplexLongSide — двусторонняя печать с переворачиванием листа вокруг длинной стороны;
  - DuplexShortSide — двусторонняя печать с переворачиванием листа вокруг короткой стороны;
- duplex() — возвращает обозначение заданного для принтера режима двусторонней печати в виде элемента перечисления DuplexMode;
- setPrintRange(<Диапазон>) — задает диапазон печати документа в виде одного из следующих элементов перечисления PrintRange из класса QPrinter:
  - AllPages — печатать все страницы;
  - Selection — печатать только выделенный фрагмент;
  - PageRange — печатать только заданный диапазон страниц;
  - CurrentPage — печатать только текущую страницу;
- setFromTo(<Первая страница>, <Последняя страница>) — задает номера первой и последней страниц в печатаемом диапазоне;
- fromPage() — возвращает номер первой страницы из печатаемого диапазона;
- toPage() — возвращает номер последней страницы из печатаемого диапазона;
- setColorMode(<Режим>) — задает режим вывода цвета в виде элемента Color (цветной) или Grayscale (черно-белый) перечисления ColorMode из класса QPrinter;

- `colorMode()` — возвращает обозначение режима вывода цвета в виде элемента перечисления `ColorMode`;
- `setResolution(<Разрешение>)` — задает разрешение для принтера в виде целого числа в точках на дюйм. Если указано неподдерживаемое значение разрешения, будет выставлено разрешение, наиболее близкое к заданному;
- `resolution()` — возвращает заданное для принтера разрешение;
- `setPaperSource(<Источник>)` — задает источник бумаги в виде значения одного из следующих элементов перечисления `PaperSource` из класса `QPrinter` (приведены лишь наиболее часто используемые источники — полный их список можно найти на странице **https://doc.qt.io/qt-6/qprinter.html#PaperSource-enum**):
  - `OnlyOne` — единственный лоток принтера или лоток, используемый по умолчанию;
  - `Lower` — нижний лоток;
  - `Middle` — средний лоток;
  - `Manual` — лоток для ручной подачи бумаги;
  - `Envelope` — лоток для конвертов;
  - `EnvelopeManual` — лоток для ручной подачи конвертов;
  - `Auto` — автоматический выбор источника;
- `paperSource()` — возвращает обозначение источника бумаги в виде элемента перечисления `PaperSource`;
- `setFontEmbeddingEnabled(<Флаг>)` — если передано значение `True`, в создаваемый документ PDF будут внедрены все использованные в его тексте шрифты, если передано значение `False` — не будут;
- `supportsMultipleCopies()` — возвращает `True`, если принтер сам способен напечатать несколько копий документа, и `False` — в противном случае;
- `supportedResolutions()` — возвращает список разрешений, поддерживаемых принтером и измеряемых в точках на дюйм:

```
>>> printer = QtPrintSupport.QPrinter()
>>> for f in printer.supportedResolutions(): print(f, end=" ")
...
    600
```

> ***Внимание!***
>
> Для некоторых принтеров метод `supportResolutions()` по какой-то причине выводит пустой список.

30.1.2. Вывод на печать

Процесс вывода документа на печать средствами PyQt делится на следующие этапы:

1. Создание принтера (объекта класса `QPrinter`) и задание его параметров:

```
printer = QtPrintSupport.QPrinter()
```

2. Создание объекта класса `QPainter`:

```
painter = QtGui.QPainter()
```

3. Вызов метода `begin(<Устройство QPaintDevice>)` у созданного ранее объекта класса `QPainter` с передачей ему в качестве параметра только что созданного принтера:

   ```
   painter.begin(printer)
   ```

   Метод `begin()` инициирует процесс вывода графики на принтер. Он возвращает `True`, если инициализация прошла успешно, и `False` — в противном случае.

4. Вывод содержимого печатаемого документа в созданный ранее объект класса `QPainter` средствами этого класса (см. *главу 25*).

5. В случае необходимости начать вывод новой страницы — вызов метода `newPage()` класса `QPrinter`:

   ```
   printer.newPage()
   ```

   Метод `newPage()` подготавливает принтер к выводу новой страницы и возвращает `True`, если подготовка увенчалась успехом, и `False` — в противном случае.

6. По окончании вывода документа — вызов метода `end()` класса `QPainter`:

   ```
   painter.end()
   ```

   Этот метод завершает рисование графики и возвращает `True`, если вывод графики был закончен успешно, и `False` — в противном случае. После его вызова выполняется собственно печать документа.

Метод `abort()` класса `QPrinter` прерывает процесс печати и возвращает `True`, если печать успешно прервана, и `False` — в противном случае.

Перед началом вывода графики понадобится определить размеры всей страницы принтера, размеры области на ней, доступной для рисования, и некоторые другие параметры, касающиеся этих размеров. Для этого применяются следующие методы класса `QPrinter`:

- `pageRect(<Единица измерения>)` — возвращает размеры области на странице, доступной для печати, в заданных единицах измерения в виде объекта класса `QRectF`. Единицы измерения задаются в виде одного из следующих элементов перечисления `Unit` из класса `QPrinter`:
  - `Millimeter` — миллиметры;
  - `Point` — пункты;
  - `Inch` — дюймы;
  - `Pica` — пики;
  - `Didot` — дидо (0,375 мм);
  - `Cicero` — цицеро (4,5 мм);
  - `DevicePixel` — пикселы;
- `paperRect(<Единица измерения Unit>)` — возвращает размеры страницы целиком в заданных единицах измерения в виде объекта класса `QRectF`.

Начало координат находится в левом верхнем углу области, отведенной под печать. Горизонтальная координатная ось направлена направо, а вертикальная — вниз.

Напишем код, который будет выводить на установленное по умолчанию устройство печати документ из двух страниц (листинг 30.1). Первую страницу мы обведем точечной синей

рамкой, а по ее центру расположим надпись «QPrinter». Вторая страница будет отпечатана в ландшафтной ориентации, и ее полностью займет графическое изображение.

Листинг 30.1. Использование класса QPrinter

```
from PyQt6 import QtCore, QtWidgets, QtGui, QtPrintSupport
import sys
app = QtWidgets.QApplication(sys.argv)
# Создаем принтер
printer = QtPrintSupport.QPrinter()
# Для целей отладки лучше выводить документ не на принтер,
# а в файл в формате PDF. Чтобы сделать это, достаточно
# раскомментировать следующую строку кода:
# printer.setOutputFileName("output.pdf")
# Создаем поверхность рисования и привязываем принтер к ней
painter = QtGui.QPainter()
painter.begin(printer)
# Рисуем рамку вокруг страницы
pen = QtGui.QPen(QtGui.QColor(QtCore.Qt.GlobalColor.blue), 5,
                              style=QtCore.Qt.PenStyle.DotLine)
painter.setPen(pen)
painter.setBrush(QtCore.Qt.BrushStyle.NoBrush)
page_size = printer.pageRect(QtPrintSupport.QPrinter.Unit.DevicePixel)
page_width = int(page_size.width())
page_height = int(page_size.height())
painter.drawRect(0, 0, page_width, page_height)
# Выводим надпись
color = QtGui.QColor(QtCore.Qt.GlobalColor.black)
painter.setPen(QtGui.QPen(color))
painter.setBrush(QtGui.QBrush(color))
font = QtGui.QFont("Verdana", pointSize=42)
painter.setFont(font)
painter.drawText(10, page_height // 2 - 100, page_width - 20,
                 50, QtCore.Qt.AlignmentFlag.AlignCenter |
                     QtCore.Qt.TextFlag.TextDontClip,
                 "QPrinter")
# Изменяем ориентацию страницы. Сделать это нужно перед вызовом
# метода newPage()
printer.setPageOrientation(QtGui.QPageLayout.Orientation.Landscape)
# Переходим на новую страницу
printer.newPage()
# Выводим изображение
page_size = printer.pageRect(QtPrintSupport.QPrinter.Unit.DevicePixel)
page_width = int(page_size.width())
page_height = int(page_size.height())
pixmap = QtGui.QPixmap("img.jpg")
pixmap = pixmap.scaled(page_width, page_height,
         aspectRatioMode=QtCore.Qt.AspectRatioMode.KeepAspectRatio)
painter.drawPixmap(0, 0, pixmap)
painter.end()
```

В листинге 30.2 приведен код класса `PrintList`, реализующий печать списков или содержимого таблиц баз данных в виде полноценного табличного отчета. Этот класс можно использовать для разработки бизнес-приложений.

Листинг 30.2. Класс `PrintList`, выводящий на печать табличные данные

```
from PyQt6 import QtCore, QtGui, QtPrintSupport

class PrintList:
    def __init__(self):
        self.printer = QtPrintSupport.QPrinter()
        self.headerFont = QtGui.QFont("Arial", pointSize=10,
                                      weight=QtGui.QFont.Weight.Bold)
        self.bodyFont = QtGui.QFont("Arial", pointSize=10)
        self.footerFont = QtGui.QFont("Arial", pointSize=9, italic = True)
        self.headerFlags = QtCore.Qt.AlignmentFlag.AlignHCenter | \
                           QtCore.Qt.TextFlag.TextWordWrap
        self.bodyFlags = QtCore.Qt.TextFlag.TextWordWrap
        self.footerFlags = QtCore.Qt.AlignmentFlag.AlignHCenter | \
                           QtCore.Qt.TextFlag.TextWordWrap
        color = QtGui.QColor(QtCore.Qt.GlobalColor.black)
        self.headerPen = QtGui.QPen(color, 2)
        self.bodyPen = QtGui.QPen(color, 1)
        self.margin = 5
        self._resetData()

    def _resetData(self):
        self.headers = None
        self.columnWidths = None
        self.data = None
        self._brush = QtCore.Qt.BrushStyle.NoBrush
        self._currentRowHeight = 0
        self._currentPageHeight = 0
        self._headerRowHeight = 0
        self._footerRowHeight = 0
        self._currentPageNumber = 1
        self._painter = None

    def printData(self):
        self._painter = QtGui.QPainter()
        self._painter.begin(self.printer)
        self._painter.setBrush(self._brush)
        if self._headerRowHeight == 0:
            self._painter.setFont(self.headerFont)
            self._headerRowHeight = self._calculateRowHeight(
                self.columnWidths, self.headers)
        if self._footerRowHeight == 0:
            self._painter.setFont(self.footerFont)
            self._footerRowHeight = self._calculateRowHeight(
                [self.printer.width()], "Страница")
```

```
        for i in range(len(self.data)):
            height = self._calculateRowHeight(self.columnWidths, self.data[i])
            if self._currentPageHeight + height > self.printer.height() - \
               self._footerRowHeight - 2 * self.margin:
                self._printFooterRow()
                self._currentPageHeight = 0
                self._currentPageNumber += 1
                self.printer.newPage()
            if self._currentPageHeight == 0:
                self._painter.setPen(self.headerPen)
                self._painter.setFont(self.headerFont)
                self.printRow(self.columnWidths, self.headers,
                              self._headerRowHeight, self.headerFlags)
                self._painter.setPen(self.bodyPen)
                self._painter.setFont(self.bodyFont)
            self.printRow(self.columnWidths, self.data[i], height,
                          self.bodyFlags)
        self._printFooterRow()
        self._painter.end()
        self._resetData()

    def _calculateRowHeight(self, widths, cellData):
        height = 0
        for i in range(len(widths)):
            r = self._painter.boundingRect(0, 0, widths[i] - 2 *
                    self.margin, 50, QtCore.Qt.TextFlag.TextWordWrap,
                    str(cellData[i]))
            h = r.height() + 2 * self.margin
            if height < h:
                height = h
        return height

    def printRow(self, widths, cellData, height, flags):
        x = 0
        for i in range(len(widths)):
            self._painter.drawText(x + self.margin,
                 self._currentPageHeight + self.margin,
                 widths[i] - self.margin, height - 2 * self.margin,
                 flags, str(cellData[i]))
            self._painter.drawRect(x, self._currentPageHeight,
                                   widths[i], height)
            x += widths[i]
        self._currentPageHeight += height

    def _printFooterRow(self):
        self._painter.setFont(self.footerFont)
        self._painter.drawText(self.margin, self.printer.height() -
             self._footerRowHeight - self.margin, self.printer.width() -
             2 * self.margin, self._footerRowHeight - 2 * self.margin,
             self.footerFlags, "Страница " + str(self._currentPageNumber))
```

Пользоваться этим классом очень просто. Сначала нужно создать его объект, вызвав конструктор следующего формата:

```
PrintList()
```

После чего задать параметры печатаемого табличного отчета, воспользовавшись следующими атрибутами класса `PrintList`:

- `headers` — заголовки для столбцов табличного отчета в виде списка строк;
- `columnWidths` — значения ширины для столбцов, измеряемые в пикселах, в виде списка целочисленных величин;
- `data` — собственно выводимые данные. Они должны представлять собой список значений, выводимых в отдельных ячейках, — каждый из элементов этого списка задает данные для одной строки.

Следующие свойства являются необязательными для указания:

- `printer` — принтер (ссылка на объект класса `QPrinter`), на котором будет печататься отчет. Изначально хранит принтер, используемый по умолчанию;
- `headerFont` — шрифт, используемый для вывода текста «шапки» табличного отчета. По умолчанию — полужирный шрифт Arial размером 10 пунктов;
- `headerPen` — параметры рамки, рисуемой вокруг ячеек «шапки». По умолчанию — черная линия толщиной 2 пиксела;
- `headerFlags` — параметры вывода текста ячеек «шапки». По умолчанию — выравнивание по центру и перенос по словам;
- `bodyFont` — шрифт, используемый для вывода текста обычных строк. По умолчанию — шрифт Arial размером 10 пунктов;
- `bodyPen` — параметры рамки, рисуемой вокруг ячеек обычных строк. По умолчанию — черная линия толщиной 1 пиксел;
- `bodyFlags` — параметры вывода текста ячеек обычных строк. По умолчанию — перенос по словам;
- `footerFont` — шрифт, используемый для вывода текста «поддона» таблицы. По умолчанию — курсивный шрифт Arial размером 9 пунктов;
- `footerFlags` — параметры вывода текста «поддона». По умолчанию — выравнивание по центру и перенос по словам;
- `margin` — величина просвета между рамкой ячейки и ее содержимым. По умолчанию — 5 пикселов.

После задания всех необходимых параметров следует вызвать метод `printData()` класса `PrintList`, который и выполняет печать данных. Впоследствии, пользуясь тем же объектом этого класса, мы можем распечатать другой набор данных.

В листинге 30.3 приведен код тестовой программы, выводящей на экран числа от 1 до 100, их квадраты и кубы. Подразумевается, что код, приведенный в листинге 30.2, сохранен в модуле PrintList.py.

Листинг 30.3. Тестовая программа для проверки класса `PrintList`

```
from PyQt6 import QtWidgets
import sys
import PrintList
```

```
app = QtWidgets.QApplication(sys.argv)
pl = PrintList.PrintList()
# Если требуется вывести документ в файл формата PDF,
# следует раскомментировать эту строку:
# pl.printer.setOutputFileName("output.pdf")
data = []
for b in range(1, 101):
    data.append([b, b ** 2, b ** 3])
pl.data = data
pl.columnWidths = [100, 100, 200]
pl.headers = ["Аргумент", "Квадрат", "Куб"]
pl.printData()
```

В результате мы должны получить табличный документ из трех страниц с тремя столбцами и нумерацией в «поддоне».

30.1.3. Служебные классы

30.1.3.1. Класс *QPageSize*

Класс `QPageSize`, определенный в модуле `QtGui`, описывает размер (или формат) страницы. Для указания размера страницы, который будет использовать принтер, предназначен метод `setPageSize(<Размеры QPageSize>)` класса `QPrinter`.

Форматы конструктора класса:

```
QPageSize()
QPageSize(<Идентификатор формата>)
QPageSize(<Размер QSize>[, name=""][, matchPolicy=SizeMatchPolicy.FuzzyMatch])
QPageSize(<Размер QSizeF>, <Единица измерения>[, name=""][,
                           matchPolicy=SizeMatchPolicy.FuzzyMatch])
QPageSize(<Исходный объект QPageSize>)
```

Первый формат создает «пустой» объект класса, не хранящий данные ни о каком размере страницы.

Второй формат позволяет указать размер сразу, в виде одного из следующих элементов перечисления `PageSizeId` из класса `QPageSize` (приведены только наиболее употребительные размеры — полный их список можно найти на странице **https://doc.qt.io/qt-6/qpagesize.html#PageSizeId-enum**): `A0`, `A1`, `A2`, `A3`, `A4`, `A5`, `Letter`, `Legal`.

Третий и четвертый форматы служат для создания нестандартных форматов. В третьем формате размеры задаются в первом параметре в пунктах. В параметре `name` можно задать имя создаваемого размера бумаги — если он не указан, будет создано имя по умолчанию вида `Custom (<Ширина> x <Высота>)`.

Если заданные размеры близки к размерам какого-либо из стандартных форматов бумаги, будет использован этот формат. В параметре `matchPolicy` можно задать режим подбора стандартного формата бумаги в виде одного из следующих элементов перечисления `SizeMatchPolicy` из класса `QPageSize`:

- `FuzzyMatch` — размеры стандартного формата должны быть близки к заданным в конструкторе размерам в определенных рамках с учетом ориентации;
- `FuzzyOrientationMatch` — размеры стандартного формата должны быть близки к заданным в конструкторе размерам в определенных рамках без учета ориентации;

- `ExactMatch` — размеры стандартного формата должны точно совпадать с заданными в конструкторе.

В четвертом формате вторым параметром должны быть заданы единицы измерения размеров в виде значения одного из следующих элементов перечисления `Unit` из класса `QPageSize`:

- `Millimeter` — миллиметры;
- `Point` — пункты;
- `Inch` — дюймы;
- `Pica` — пики;
- `Didot` — дидо (0,375 мм);
- `Cicero` — цицеро (4,5 мм).

Последний формат создает копию указанного исходного объекта.

Примеры:

```
# Задаем размер бумаги A5
ps = QtGui.QPageSize(QtGui.QPageSize.PageSizeId.A5)
printer.setPageSize(ps)
# Задаем размер бумаги 50 x 30 пунктов с названием "Особый размер"
sz = QtCore.QSize(50, 30)
ps = QtGui.QPageSize(sz, name="Особый размер",
                     matchPolicy=QtGui.QPageSize.SizeMatchPolicy.FuzzyMatch)
printer.setPageSize(ps)
```

Класс `QPageSize` поддерживает следующие полезные методы (полный их список можно найти на странице **https://doc.qt.io/qt-6/qpagesize.html**):

- `size(<Единица измерения Unit>)` и `rect(<Единица измерения Unit>)` — возвращают размеры бумаги в заданных единицах измерения в виде объекта класса `QSizeF` или `QRectF` соответственно;
- `size()` — статический, возвращает размеры бумаги для заданного идентификатора формата в заданных единицах измерения, представленные объектом класса `QSizeF`. Формат метода:

  ```
  size(<Идентификатор формата PageSizeId>, <Единица измерения Unit>)
  ```

  Пример:

  ```
  >>> s = QtGui.QPageSize.size(QtGui.QPageSize.PageSizeId.A0,
  ...                          QtGui.QPageSize.Unit.Millimeter)
  >>> print(s.width(), "x", s.height())
     841.0 x 1189.0
  ```

- `sizePixels(<Разрешение>)` и `rectPixels(<Разрешение>)` — возвращают размеры бумаги для заданного разрешения, которое измеряется в точках на дюйм, в пикселах в виде объекта класса `QSize` или `QRect` соответственно:

  ```
  >>> ps = QtGui.QPageSize(QtGui.QPageSize.PageSizeId.A5)
  >>> s = ps.sizePixels(600)
  >>> print(s.width(), "x", s.height())
      3500 x 4958
  ```

- `sizePixels()` — статический, возвращает размеры бумаги для заданных идентификатора формата и разрешения, которое измеряется в точках на дюйм, в пикселах в виде объекта класса `QSize`. Формат метода:

  ```
  sizePixels(<Идентификатор формата PageSizeId>, <Разрешение>)
  ```

- `sizePoints()` и `rectPoints()` — возвращают размеры бумаги в пунктах в виде объекта класса `QSize` или `QRect` соответственно;
- `sizePoints(<Идентификатор формата PageSizeId>)` — статический, возвращает размеры бумаги для заданного идентификатора формата в пунктах в виде объекта класса `QSize`;
- `swap(<Размер QPageSize>)` — меняет размер бумаги, хранящийся в текущем объекте класса, на заданный в параметре;
- `name()` — возвращает имя размера бумаги;
- `name(<Идентификатор формата PageSizeId>)` — статический, возвращает имя размера бумаги, заданного идентификатором;
- `isEquivalentTo(<Размер QPageSize>)` — возвращает `True`, если размеры текущего формата бумаги равны размерам, переданным в параметре, и `False` — в противном случае.

Также класс `QPageSize` поддерживает операторы сравнения == и `!=`:

```
s1 = QtGui.QPageSize(QtGui.QPageSize.PageSizeId.A4)
s2 = QtGui.QPageSize(QtGui.QPageSize.PageSizeId.A4)
s3 = QtGui.QPageSize(QtGui.QPageSize.PageSizeId.A3)
print(s1 == s2)                              # Выведет: True
print(s1 != s3)                              # Выведет: True
```

30.1.3.2. Класс *QPageLayout*

Класс `QPageLayout`, также определенный в модуле `QtGui`, представляет сразу размеры, ориентацию страницы и величины отступов от ее краев. Передать все эти сведения принтеру позволяет метод `setPageLayout(<Параметры QPageLayout>)` класса `QPrinter`.

Форматы вызова конструктора этого класса:

```
QPageLayout()
QPageLayout(<Размер QPageSize>, <Ориентация Orientation>, <Отступы QMarginsF>[,
                        units=Unit.Point][, minMargins=QMarginsF(0,0,0,0)])
QPageLayout(<Исходный объект QPageLayout>)
```

Первый формат создает «пустой» объект, не хранящий никаких сведений.

Второй формат позволяет задать сразу все необходимые сведения о бумаге. Ориентация бумаги задается в виде элемента `Portrait` (портретная) или `Landscape` (ландшафтная) перечисления `Orientation` из класса `QPageLayout`. Параметр `units` задает единицы измерения размеров в виде значения одного из элементов перечисления `Unit` из класса `QPageLayout` (см. *разд. 30.1.1*). А параметр `minMargins` указывает минимальные отступы от краев страницы, которые может обеспечить принтер. Пример:

```
layout = QtGui.QPageLayout(QtGui.QPageSize(QtGui.QPageSize.PageSizeId.A5),
                           QtGui.QPageLayout.Orientation.Landscape,
                           QtCore.QMarginsF(10, 10, 10, 10),
                           units=QtGui.QPageLayout.Unit.Millimeter)
printer.setPageLayout(layout)
```

Третий формат создает копию указанного исходного объекта.

Наиболее полезные методы класса `QPageLayout` приведены далее (полный их список можно найти на странице **https://doc.qt.io/qt-6/qpagelayout.html**).

- `setPageSize(<Размер QPageSize>[, minMargins=QMarginsF(0, 0, 0, 0)])` — задает размеры бумаги. В необязательном параметре `minMargins` можно указать минимальные величины отступов от краев страницы (объект класса `QMarginsF`);
- `setOrientation(<Ориентация Orientation>)` — задает ориентацию страницы;
- `setMargins(<Отступы QMarginsF>)` — задает величины отступов от краев страницы. Возвращает `True`, если операция прошла успешно, и `False` — в противном случае;
- `setLeftMargin(<Отступ>)`, `setTopMargin(<Отступ>)`, `setRightMargin(<Отступ>)` и `setBottomMargin(<Отступ>)` — задают величины отступов от левого, верхнего, правого и нижнего краев страницы соответственно. Возвращают `True`, если операция прошла успешно, и `False` — в противном случае;
- `setUnits(<Единица измерения Unit>)` — задает единицу измерения размеров;
- `setMinimumMargins(<Отступы QMarginsF>)` — задает минимальные величины отступов от краев страницы, которые может обеспечить принтер;
- `swap(<Параметры QPageLayout>)` — меняет параметры бумаги, хранящиеся в текущем объекте класса, на заданные;
- `fullRect([<Единица измерения Unit>])` — возвращает размеры страницы с учетом ориентации, но без учета отступов в виде объекта класса `QRectF`. Если единица измерения не указана, выведенные значения будут исчисляться в единицах измерения, заданных в конструкторе. Пример:

```
>>> lt = QtGui.QPageLayout(QtGui.QPageSize(QtGui.QPageSize.PageSizeId.A5),
...                        QtGui.QPageLayout.Orientation.Landscape,
...                        QtCore.QMarginsF(10, 10, 10, 10),
...                        units=QtGui.QPageLayout.Unit.Millimeter)
>>> r = lt.fullRect()
>>> print(r.width(), "x", r.height())
    210.0 x 148.0
```

- `fullRectPixels(<Разрешение>)` — возвращает размеры страницы в пикселах с учетом ориентации, но без учета отступов для заданного разрешения, которое измеряется в точках на дюйм. Результат представляет собой объект класса `QRect`;
- `fullRectPoints()` — возвращает размеры страницы в пунктах с учетом ориентации, но без учета отступов в виде объекта класса `QRect`;
- `paintRect([<Единица измерения Unit>])` — возвращает размеры страницы с учетом ориентации и отступов (фактически — доступной для вывода графики области страницы) в виде объекта класса `QRectF`. Если единица измерения не указана, выведенные значения будут исчисляться в единицах измерения, заданных в конструкторе;
- `paintRectPixels(<Разрешение>)` — возвращает размеры страницы в пикселах с учетом ориентации и отступов для заданного разрешения, которое измеряется в точках на дюйм. Результат представляет собой объект класса `QRect`;
- `paintRectPoints()` — возвращает размеры страницы в пунктах с учетом ориентации и отступов в виде объекта класса `QRect`.

Класс `QPageLayout` также поддерживает операторы сравнения == и `!=`.

30.2. Задание параметров принтера и страницы

30.2.1. Класс *QPrintDialog*

Класс `QPrintDialog` реализует функциональность стандартного диалогового окна выбора и настройки принтера (рис. 30.1).

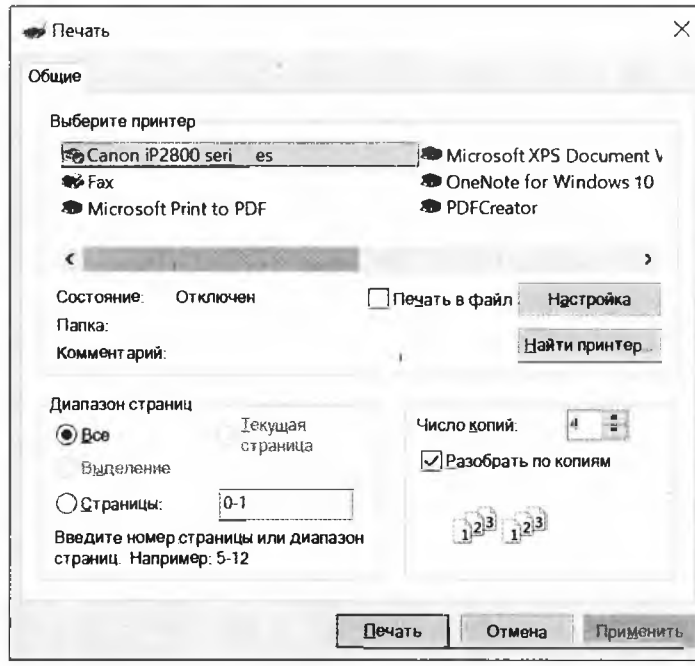


Рис. 30.1. Стандартное диалоговое окно выбора и настройки принтера

Иерархия наследования этого класса:

```
(QObject, QPaintDevice) — QWidget — QDialog - QAbstractPrintDialog - QPrintDialog
```

Конструктор класса `QPrintDialog` имеет следующий формат:

```
QPrintDialog(<Принтер QPrinter>[, parent=None])
```

В первом параметре указывается принтер, настройки которого будут задаваться в диалоговом окне. Необязательный параметр `parent` может быть использован для задания родителя.

Перед выводом диалогового окна можно указать для него значения по умолчанию, воспользовавшись методами класса `QPrinter`, описанными в *разд. 30.1.1*. Например, для указания размера бумаги следует вызвать метод `setPageSize()`, для задания количества копий — `setCopyCount()` и т. д.

Вывести диалоговое окно на экран можно вызовом метода `exec()`.

После закрытия диалогового окна все заданные в нем параметры при необходимости могут быть получены через соответствующие методы класса `QPrinter`. При этом выяснить размер бумаги, ее ориентацию и отступы от краев позволит метод `pageLayout()`, количество копий документа — метод `copyCount()` и т. д. Все эти методы были описаны в *разд. 30.1.1*.

Все настройки, заданные в диалоговом окне, будут применены к принтеру самой библиотекой PyQt. Так, если пользователь выберет другой принтер, печать будет выполнена на выбранном им принтере. А если он укажет напечатать документ в нескольких копиях, все эти копии будут напечатаны PyQt самостоятельно.

Класс `QPrintDialog` поддерживает следующие основные методы (полный их список можно найти на страницах **https://doc.qt.io/qt-6/qabstractprintdialog.html** и **https://doc.qt.io/qt-6/qprintdialog.html**):

- `setOption(<Настройка>[, on=True])` — активизирует указанную в первом параметре настройку принтера, если в параметре `on` задано значение `True`, и сбрасывает, если передано `False`. Настройка задается в виде одного из следующих элементов перечисления `PrintDialogOption` из класса `QPrintDialog` или их комбинации через оператор `|`:
  - `None` — все настройки принтера сброшены;
  - `PrintToFile` — доступна возможность печати в файл;
  - `PrintSelection` — разрешен выбор принтера;
  - `PrintPageRange` — доступно указание диапазона печатаемых страниц;
  - `PrintShowPageSize` — вывод размера страницы и отступов (только если это возможно);
  - `PrintCollateCopies` — доступно указание режима печати копий документов (будет ли каждая копия печататься полностью, или сначала будут отпечатаны все копии первой страницы, потом — копии второй и т. д.);
  - `PrintCurrentPage` — доступно указание печати только текущей страницы;
- `setOptions(<Настройки PrintDialogOptions>)` — позволяет активизировать сразу несколько настроек принтера;
- `testOption(<Настройка PrintDialogOptions>)` — возвращает `True`, если заданная настройка принтера активизирована, и `False` — в противном случае;
- `options()` — возвращает через оператор `|` комбинацию элементов перечисления `PrintDialogOption`, представляющих активизированные настройки;
- `printer()` — возвращает принтер (объект класса `QPrinter`), выбранный в диалоговом окне.

30.2.2. Класс *QPageSetupDialog*

Класс `QPageSetupDialog` реализует работу стандартного диалогового окна установки параметров страницы: размера, ориентации, отступов и др. (рис. 30.2).

Иерархия наследования этого класса:

```
(QObject, QPaintDevice) — QWidget — QDialog — QPageSetupDialog
```

Конструктор класса `QPageSetupDialog` имеет следующие форматы вызова:

```
QPageSetupDialog([parent=None])
QPageSetupDialog(<Принтер QPrinter>[, parent=None])
```

Первый формат создает диалоговое окно, привязанное к используемому по умолчанию устройству печати, второй формат позволяет указать нужный принтер. Необязательный параметр `parent` может быть использован для задания родителя.

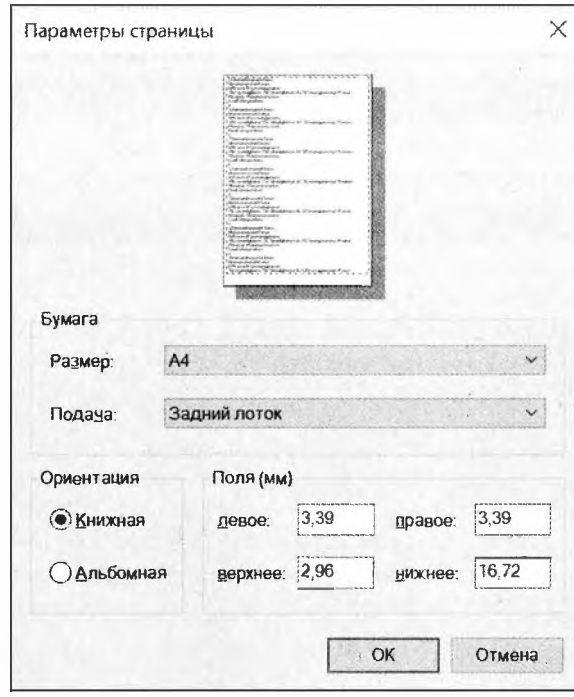


Рис. 30.2. Стандартное диалоговое окно установки параметров страницы

Принципы работы с диалоговым окном параметров страницы такие же, как и у его «коллеги», представляющего параметры принтера (см. *разд. 30.2.1*): мы задаем нужные параметры страницы, вызывая соответствующие методы класса `QPrinter`, и выводим диалоговое окно на экран вызовом метода `exec()`. И, опять же, все заданные в этом диалоговом окне настройки будут применены к принтеру автоматически.

Из поддерживаемых классом `QPageSetupDialog` методов нас может заинтересовать разве что `printer()`. Он возвращает принтер (объект класса `QPrinter`), указанный в вызове конструктора.

Напишем печатающую изображение из указанного графического файла утилиту, позволяющую задать настройки принтера и параметры страницы в соответствующих диалоговых окнах (листинг 30.4).

Листинг 30.4. Использование классов `QPrintDialog` и `QPageSetupDialog`

```
from PyQt6 import QtCore, QtWidgets, QtGui, QtPrintSupport
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Печать изображений")
        self.printer = QtPrintSupport.QPrinter()
```

```
        self.printer.setPageOrientation(
                            QtGui.QPageLayout.Orientation.Landscape)
        self.file = None
        vbox = QtWidgets.QVBoxLayout()
        btnOpen = QtWidgets.QPushButton("&Открыть файл...")
        btnOpen.clicked.connect(self.openFile)
        vbox.addWidget(btnOpen)
        btnPageOptions = QtWidgets.QPushButton("Настройка &страницы...")
        btnPageOptions.clicked.connect(self.showPageOptions)
        vbox.addWidget(btnPageOptions)
        btnPrint = QtWidgets.QPushButton("&Печать...")
        btnPrint.clicked.connect(self.print)
        vbox.addWidget(btnPrint)
        self.setLayout(vbox)
        self.resize(300, 100)

    def openFile(self):
        self.file = QtWidgets.QFileDialog.getOpenFileName(parent=self,
                    caption="Выберите графический файл",
                    filter="Графические файлы (*.bmp *.jpg *.png)")[0]

    def showPageOptions(self):
        pd = QtPrintSupport.QPageSetupDialog(self.printer, parent=self)
        pd.exec()

    def print(self):
        pd = QtPrintSupport.QPrintDialog(self.printer, parent=self)
        pd.setOptions(
         QtPrintSupport.QAbstractPrintDialog.PrintDialogOption.PrintToFile |
         QtPrintSupport.QAbstractPrintDialog.PrintDialogOption.PrintSelection)
        if pd.exec() == QtWidgets.QDialog.DialogCode.Accepted:
            painter = QtGui.QPainter()
            painter.begin(self.printer)
            pixmap = QtGui.QPixmap(self.file)
            pixmap = pixmap.scaled(self.printer.width(), self.printer.height(),
                     aspectRatioMode=QtCore.Qt.AspectRatioMode.KeepAspectRatio)
            painter.drawPixmap(0, 0, pixmap)
            painter.end()

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec())
```

30.3. Предварительный просмотр

30.3.1. Класс *QPrintPreviewDialog*

Класс `QPrintPreviewDialog` выводит документ в диалоговом окне предварительного просмотра (рис. 30.3). Это окно дает возможности перехода со страницы на страницу, масшта-

бирования выведенного документа, указания режима его вывода (постранично, по две страницы и т. п.), задания ориентации страницы, вызова диалогового окна настроек страницы, а также отправки документа на печать.

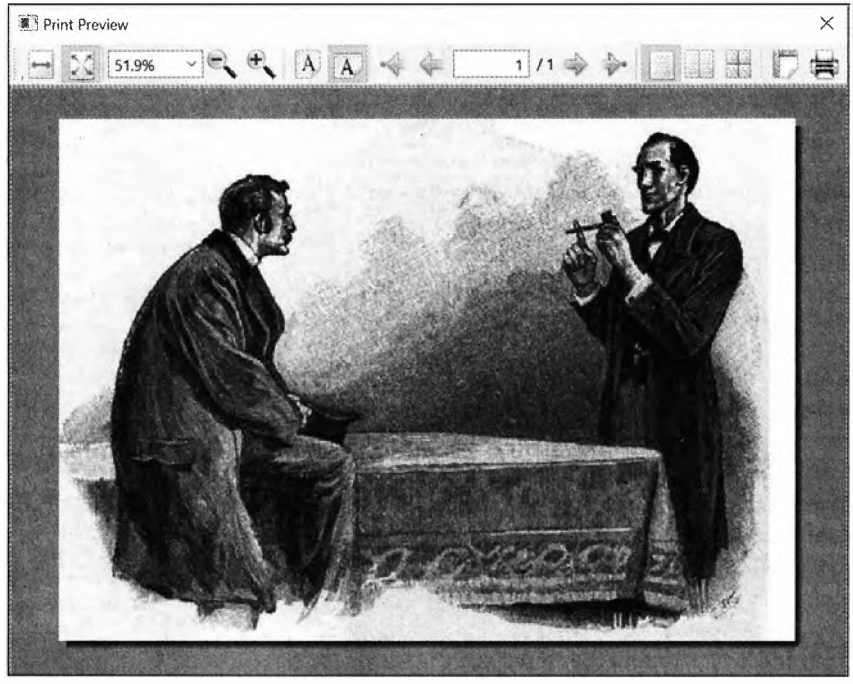


Рис. 30.3. Диалоговое окно предварительного просмотра документа

Иерархия наследования класса `QPrintPreviewDialog`:

```
(QObject, QPaintDevice) — QWidget — QDialog - QPrintPreviewDialog
```

Конструктор класса имеет следующие форматы вызова:

```
QPrintPreviewDialog([parent=None][, ][flags=0])
QPrintPreviewDialog(<Принтер QPrinter>[, parent=None][, flags=0])
```

Первый формат создает диалоговое окно, привязанное к используемому по умолчанию устройству печати, второй формат позволяет указать нужный принтер. Параметр `parent` задает родителя, а параметр `flags` — типа окна (см. *разд. 19.11*).

Вывод документа для предварительного просмотра с помощью класса `QPrintPreviewDialog` выполняется в три этапа:

1. Создание объекта класса `QPrintPreviewDialog`.
2. Назначение сигналу `paintRequested(<Принтер QPrinter>)` этого объекта обработчика, внутри которого и будет выполняться вывод документа. Обработчику в параметре передается указанный в вызове конструктора принтер.
3. Вывод диалогового окна на экран вызовом метода `exec()` класса `QPrintPreviewDialog`.

Переделаем написанную ранее утилиту печати изображений (см. листинг 30.4) таким образом, чтобы дать пользователю возможность просматривать изображения перед печатью. Код новой утилиты приведен в листинге 30.5.

Листинг 30.5. Использование класса QPrintPreviewDialog

```
from PyQt6 import QtCore, QtWidgets, QtGui, QtPrintSupport
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Печать изображений")
        self.printer = QtPrintSupport.QPrinter()
        self.printer.setPageOrientation(
                             QtGui.QPageLayout.Orientation.Landscape)
        self.file = None
        vbox = QtWidgets.QVBoxLayout()
        btnOpen = QtWidgets.QPushButton("&Открыть файл...")
        btnOpen.clicked.connect(self.openFile)
        vbox.addWidget(btnOpen)
        btnPageOptions = QtWidgets.QPushButton("Настройка &страницы...")
        btnPageOptions.clicked.connect(self.showPageOptions)
        vbox.addWidget(btnPageOptions)
        btnPreview = QtWidgets.QPushButton("П&росмотр...")
        btnPreview.clicked.connect(self.preview)
        vbox.addWidget(btnPreview)
        btnPrint = QtWidgets.QPushButton("&Печать...")
        btnPrint.clicked.connect(self.print)
        vbox.addWidget(btnPrint)
        self.setLayout(vbox)
        self.resize(300, 100)

    def openFile(self):
        self.file = QtWidgets.QFileDialog.getOpenFileName(parent=self,
                    caption="Выберите графический файл",
                    filter="Графические файлы (*.bmp *.jpg *.png)")[0]

    def showPageOptions(self):
        pd = QtPrintSupport.QPageSetupDialog(self.printer, parent=self)
        pd.exec()

    def preview(self):
        pp = QtPrintSupport.QPrintPreviewDialog(self.printer, parent=self)
        pp.paintRequested.connect(self._printImage)
        pp.exec()

    def print(self):
        pd = QtPrintSupport.QPrintDialog(self.printer, parent=self)
        pd.setOptions(
         QtPrintSupport.QAbstractPrintDialog.PrintDialogOption.PrintToFile |
         QtPrintSupport.QAbstractPrintDialog.PrintDialogOption.PrintSelection)
```

```
        if pd.exec() == QtWidgets.QDialog.DialogCode.Accepted:
            self._printImage(self.printer)

    def _printImage(self, printer):
        painter = QtGui.QPainter()
        painter.begin(printer)
        pixmap = QtGui.QPixmap(self.file)
        pixmap = pixmap.scaled(printer.width(), printer.height(),
                 aspectRatioMode=QtCore.Qt.AspectRatioMode.KeepAspectRatio)
        painter.drawPixmap(0, 0, pixmap)
        painter.end()

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec())
```

30.3.2. Класс *QPrintPreviewWidget*

Класс `QPrintPreviewWidget` представляет отдельный компонент — панель для предварительного просмотра документа. Он имеет ту же функциональность, что и диалоговое окно, рассмотренное в *разд. 30.3.1*.

Иерархия наследования этого класса:

```
(QObject, QPaintDevice) — QWidget — QPrintPreviewWidget
```

Конструктор класса `QPrintPreviewWidget` имеет следующие форматы вызова:

```
QPrintPreviewWidget([parent=None][, flags=0])
QPrintPreviewWidget(<Принтер QPrinter>[, parent=None][, flags=0])
```

Первый формат создает компонент, привязанный к используемому по умолчанию устройству печати, второй формат позволяет указать нужный принтер. Параметр `parent` задает родителя, а параметр `flags` — тип окна (см. *разд. 19.11*).

Последовательность действий, необходимых для реализации предварительного просмотра с помощью класса `QPrintPreviewWidget`, почти такая же, что и в случае класса `QPrintPreviewDialog`: мы создаем объект класса `QPrintPreviewWidget`, назначаем для его сигнала `paintRequested()` обработчик и пишем в этом обработчике код, который и выполнит вывод документа.

Класс `QPrintPreviewWidget` поддерживает ряд методов, предназначенных для выполнения различных действий над выведенным в панели документом. Рассмотрим их:

- `setOrientation(<Ориентация>)` — задает ориентацию страницы в виде элемента `Portrait` (портретная) или `Landscape` (ландшафтная) перечисления `Orientation` из класса `QPageLayout`. Метод является слотом;
- `setPortraitOrientation()` — задает портретную ориентацию страницы. Метод является слотом;
- `setLandscapeOrientation()` — задает ландшафтную ориентацию страницы. Метод является слотом;

- `orientation()` — возвращает ориентацию страницы в виде элемента перечисления `Orientation`;
- `setViewMode(<Режим просмотра>)` — задает режим просмотра страниц документа в виде одного из следующих элементов перечисления `ViewMode` из класса `QPrintPreviewWidget`:
  - `SinglePageView` — постраничный режим (одновременно выводится только одна страница);
  - `FacingPagesView` — режим просмотра разворота (одновременно отображаются две страницы);
  - `AllPagesView` — выводятся сразу все страницы документа.

  Метод является слотом;
- `setSinglePageViewMode()` — задает постраничный режим просмотра страниц. Метод является слотом;
- `setFacingPagesViewMode()` — задает режим просмотра разворота. Метод является слотом;
- `setAllPagesViewMode()` — задает режим просмотра всех страниц документа. Метод является слотом;
- `viewMode()` — возвращает обозначение режима просмотра страниц в виде элемента перечисления `ViewMode`;
- `setZoomMode(<Режим>)` — задает режим масштабирования страниц при просмотре в виде одного из следующих элементов перечисления `ZoomMode` из класса `QPrintPreviewWidget`:
  - `CustomZoom` — произвольное масштабирование со значением масштаба, заданным методом `setZoomFactor()`;
  - `FitToWidth` — выбирается такое значение масштаба, чтобы страница помещалась в панели по ширине;
  - `FitInView` — выбирается такое значение масштаба, чтобы страница помещалась в панели полностью.

  Метод является слотом;
- `fitToWidth()` — задает такой режим масштабирования, чтобы страница помещалась в панели по ширине. Метод является слотом;
- `fitInView()` — задает такой режим масштабирования, чтобы страница помещалась в панели полностью. Метод является слотом;
- `zoomMode()` — возвращает обозначение режима масштабирования страниц в виде элемента перечисления `ZoomMode`;
- `setZoomFactor(<Масштаб>)` — задает значение масштаба для выводимого в панели документа в виде вещественного числа при условии, что указан режим масштабирования `CustomZoom`. Значение `1.0` задает изначальный масштаб, меньшие значения уменьшают его, а бóльшие — увеличивают. Метод является слотом;
- `zoomIn([<Масштаб>])` — задает новый масштаб документа, увеличивая его. Если масштаб не задан, будет использовано значение `1.1`. Метод является слотом;
- `zoomOut([<Масштаб>])` — задает новый масштаб документа, уменьшая его. Если масштаб не задан, будет использовано значение `1.1`. Метод является слотом;

- `zoomFactor()` — возвращает значение масштаба;
- `pageCount()` — возвращает общее количество страниц в документе;
- `setCurrentPage(<Номер страницы>)` — задает номер страницы, выводимой в панели. Метод является слотом;
- `currentPage()` — возвращает номер страницы, выводящейся в панели в данный момент;
- `updatePreview()` — обновляет содержимое панели. При этом будет снова сгенерирован сигнал `paintRequested()`. Метод является слотом;
- `print()` — печатает документ, отображающийся в панели, с выводом на экран диалогового окна выбора и настройки принтера. Метод является слотом.

В дополнение к `paintRequested()`, класс `QPrintPreviewWidget` поддерживает сигнал `previewChanged()`. Он генерируется при изменении параметров панели просмотра: ориентации страницы, режима просмотра, масштаба и др.

30.4. Класс *QPrinterInfo*: получение сведений об устройстве печати

Класс `QPrinterInfo` позволяет получить сведения как обо всех установленных в системе физических устройствах печати, так и о конкретном устройстве, указанном нами. Форматы его конструктора следующие:

```
QPrinterInfo(<Принтер QPrinter>)
QPrinterInfo(<Исходный объект QPrinterInfo>)
```

Первый формат создает объект со сведениями об устройстве печати, связанном с заданным принтером. Второй формат создает копию указанного исходного объекта.

Класс `QPrinterInfo` поддерживает следующие методы:

- `availablePrinters()` — статический, возвращает список всех установленных в системе устройств печати, представленных объектами класса `QPrinterInfo`;
- `availablePrinterNames()` — статический, возвращает список имен всех установленных в системе устройств печати:

```
>>> l = QtPrintSupport.QPrinterInfo.availablePrinterNames()
>>> for p in l: print(p)
...
    OneNote for Windows 10
    PDFCreator
    Microsoft XPS Document Writer
    Microsoft Print to PDF
    Fax
    Canon iP2800 series
```

- `defaultPrinter()` — статический, возвращает сведения об устройстве печати, используемом по умолчанию, в виде объекта класса `QPrinterInfo`;
- `defaultPrinterName()` — статический, возвращает имя устройства печати, используемого по умолчанию:

```
>>> print(QtPrintSupport.QPrinterInfo.defaultPrinterName())
    Canon iP2800 series
```

- `printerInfo(<Имя устройства печати>)` — статический, возвращает объект класса `QPrinterInfo`, хранящий сведения об устройстве печати с указанным именем. Если такого устройства печати нет, возвращается некорректный, «пустой» объект класса. Пример:

```
>>> p = QtPrintSupport.QPrinterInfo.printerInfo("Canon iP2800 series")
```

- `isNull()` — возвращает `True`, если текущий объект указывает на конкретное установленное в системе устройство печати, и `False`, если он является «пустым»;
- `printerName()` — возвращает имя текущего устройства печати;
- `makeAndModel()` — возвращает развернутое наименование устройства печати с указанием его изготовителя, если таковые сведения имеются в наличии, или просто его имя — в противном случае;
- `description()` — возвращает развернутое описание устройства печати, если таковое имеется в наличии, или просто его имя — в противном случае;
- `location()` — возвращает указание на местоположение устройства печати, если таковое имеется в наличии, или пустую строку — в противном случае;
- `isDefault()` — возвращает `True`, если устройство печати помечено как используемое по умолчанию, и `False` — в противном случае;
- `isRemote()` — возвращает `True`, если это сетевое устройство печати, и `False`, если локальное;
- `state()` — возвращает текущее состояние устройства печати в виде одного из следующих элементов перечисления `PrinterState` из класса `QPrinter`:
  - `Idle` — простаивает;
  - `Active` — идет печать;
  - `Aborted` — печать была прервана;
  - `Error` — возникла ошибка;
- `supportedPageSizes()` — возвращает список поддерживаемых устройством печати размеров бумаги, представляемых объектами класса `QPageSize`:

```
>>> lst = p.supportedPageSizes()
>>> for l in lst: print(l.name())
...
    Letter 22x28 см 8.5"x11"
    Legal 22x36 см 8.5"x14"
    A5
    A4
    . . . Остальной вывод пропущен . . .
```

- `supportsCustomPageSizes()` — возвращает `True`, если устройство печати поддерживает указание произвольного размера бумаги, и `False` — в противном случае;
- `supportedResolutions()` — возвращает список разрешений, поддерживаемых устройством печати и измеряемых в точках на дюйм;
- `supportedDuplexModes()` — возвращает список поддерживаемых устройством печати режимов двусторонней печати. Каждый элемент списка является одним из элементов перечисления `DuplexMode` из класса `QPrinter` (см. *разд. 30.1.1*);

- `supportedColorModes()` — возвращает список поддерживаемых устройством печати режимов вывода цвета. Каждый элемент списка является одним из элементов перечисления `ColorMode` из класса `QPrinter` (см. *разд. 30.1.1*);
- `defaultPageSize()` — возвращает установленный по умолчанию размер бумаги в виде объекта класса `QPageSize`:

```
>>> ps = p.defaultPageSize()
>>> print(p.defaultPageSize().name())
    A4
>>> s = ps.size(QtGui.QPageSize.Millimeter)
>>> print(s.width(),"x", s.height())
210.0 x 297.0
```

- `defaultDuplexMode()` — возвращает заданный по умолчанию режим двусторонней печати в виде элемента перечисления `DuplexMode`;
- `defaultColorMode()` — возвращает заданный по умолчанию режим вывода цвета в виде элемента перечисления `ColorMode`;
- `minimumPhysicalPageSize()` и `maximumPhysicalPageSize()` — возвращают соответственно минимальный и максимальный размеры бумаги, поддерживаемые принтером и представленные объектами класса `QPageSize`:

```
>>> ps = p.maximumPhysicalPageSize()
>>> print(ps.name())
    Custom (1190pt x 1916pt)
>>> s = ps.size(QtGui.QPageSize.Millimeter)
>>> print(s.width(),"x", s.height())
    419.81 x 675.92
```

30.5. Класс *QPdfWriter*: экспорт в формат PDF

В *разд. 30.1.1* упоминалось о возможности сохранения печатаемого документа в файл формата PDF — для этого достаточно вызвать метод `setOutputFileName()`, передав ему в качестве параметра путь к файлу и указав у этого файла расширение pdf. Однако при этом задействуется подсистема печати Windows, что приводит к напрасному расходу системных ресурсов.

Библиотека PyQt предоставляет возможность экспорта документов в формат PDF напрямую, без использования подсистемы печати. Это обеспечивает класс `QPdfWriter`, определенный в модуле `QtGui`. Его иерархия наследования:

```
QPaintDevice - (QObject, QPagedPaintDevice) - QPdfWriter
```

Формат конструктора этого класса:

```
QPdfWriter(<Путь к результирующему PDF-файлу>)
```

Вывод документа в формат PDF выполняется так же, как и его печать (см. *разд. 30.1.2*). Как и `QPrinter`, класс `QPdfWriter` поддерживает метод `newPage()`, подготавливающий следующую страницу документа для вывода.

Класс `QPdfWriter` поддерживает также и следующие полезные методы (полный их список можно найти на страницах **https://doc.qt.io/qt-6/qpdfwriter.html** и **https://doc.qt.io/qt-6/qpagedpaintdevice.html**):

- `setPageSize(<Размер QPageSize>)` — задает размер страницы. Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setPageOrientation(<Ориентация>)` — задает ориентацию страницы в виде значения элемента `Portrait` (портретная) или `Landscape` (ландшафтная) перечисления `Orientation` из класса `QPageLayout`. Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setPageMargins(<Отступы QMarginsF>[, units=Unit.Millimeter])` — задает отступы от краев страницы в заданных единицах измерения. Последние указываются в параметре `units` в виде одного из элементов перечисления `Unit` из класса `QPageLayout` (см. *разд. 30.1.1*). Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setPageLayout(<Параметры QPageLayout>)` — задает сразу все параметры страницы (размеры, ориентацию, отступы от краев страницы и т. п.). Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setResolution(<Разрешение>)` — задает разрешение в виде целого числа в точках на дюйм;
- `resolution()` — возвращает разрешение в виде целого числа в точках на дюйм;
- `setTitle(<Заголовок>)` — задает заголовок документа, который будет записан в файл PDF;
- `setCreator(<Имя создателя>)` — задает имя создателя документа, которое будет записано в файл PDF;
- `setPdfVersion(<Версия>)` — задает версию формата PDF, в которой будет создан документ, в виде одного из следующих элементов перечисления `PdfVersion` из класса `QPdfWriter`: `PdfVersion_1_4`, `PdfVersion_A1b` или `PdfVersion_1_6`.

Возьмем код из листинга 30.1 и немного переделаем его: пусть на первой странице он выводит надпись «QPdfWriter» и выполняет печать в файл формата PDF на бумаге формата A5 (листинг 30.6).

Листинг 30.6. Использование класса `QPdfWriter`

```
from PyQt6 import QtCore, QtWidgets, QtGui, QtPrintSupport
import sys
app = QtWidgets.QApplication(sys.argv)
writer = QtGui.QPdfWriter("output.pdf")
writer.setCreator("Владимир Дронов")
writer.setTitle("Тест")
layout = QtGui.QPageLayout()
layout.setPageSize(QtGui.QPageSize(QtGui.QPageSize.PageSizeId.A5))
layout.setOrientation(QtGui.QPageLayout.Orientation.Portrait)
writer.setPageLayout(layout)
painter = QtGui.QPainter()
painter.begin(writer)
color = QtGui.QColor(QtCore.Qt.GlobalColor.black)
painter.setPen(QtGui.QPen(color))
painter.setBrush(QtGui.QBrush(color))
font = QtGui.QFont("Verdana", pointSize=42)
painter.setFont(font)
```

```
# Получаем размеры страницы в пикселах для текущего экранного разрешения
page_size = layout.paintRectPixels(writer.resolution())
page_width = int(page_size.width())
page_height = int(page_size.height())
painter.drawText(10, page_height // 2 - 50, page_width - 20, 50,
        QtCore.Qt.AlignmentFlag.AlignCenter | QtCore.Qt.TextFlag.TextDontClip,
        "QPdfWriter")
layout.setOrientation(QtGui.QPageLayout.Orientation.Landscape)
writer.setPageLayout(layout)
writer.newPage()
page_size = layout.paintRectPixels(writer.resolution())
page_width = int(page_size.width())
page_height = int(page_size.height())
pixmap = QtGui.QPixmap("img.jpg")
pixmap = pixmap.scaled(page_width, page_height,
         aspectRatioMode=QtCore.Qt.AspectRatioMode.KeepAspectRatio)
painter.drawPixmap(0, 0, pixmap)
painter.end()
```

У класса `QPdfWriter` есть недостаток — создаваемые с его помощью PDF-документы имеют очень большой размер. Так, у авторов после выполнения кода из листинга 30.6 получился файл объемом в 7,1 Мбайт. Вероятно, PyQt создает документы максимально допустимого в формате PDF качества, и понизить его, тем самым уменьшив размер результирующих файлов, невозможно.

ГЛАВА 31

Сохранение настроек программ

PyQt позволяет сохранять настройки программ в едином *хранилище настроек*, в качестве которого могут использоваться как реестр Windows (по умолчанию), так и INI-файл.

Для работы с таким хранилищем предусмотрен удобный класс `QSettings`, определенный в модуле `QtCore`.

31.1. Создание объекта класса *QSettings*

Форматы конструктора класса `QSettings`:

```
QSettings([parent=None])
QSettings(<Уровень>[, parent=None])
QSettings(<Название организации>[, application=""][, parent=None])
QSettings(<Уровень>, <Название организации>[, application=""][, parent=None])
QSettings(<Тип хранилища>, <Уровень>, <Название организации>[,
                                    application=""][, parent=None])
QSettings(<Путь к файлу или ключу реестра>, <Тип хранилища>[, parent=None])
```

Первый формат создает объект, хранящий настройки на уровне текущего пользователя в реестре Windows.

Чтобы настройки успешно сохранились, понадобится задать названия организации и программы, и сделать это нужно еще до создания объекта класса `QSettings`. В этом нам помогут следующие два статических метода класса `QApplication` из модуля `QtWidgets`:

- `setOrganizationName(<Название организации>)` — задает название организации;
- `setApplicationName(<Название программы>)` — задает название программы.

Пример:

```
QtWidgets.QApplication.setOrganizationName("Прохоренок и Дронов")
QtWidgets.QApplication.setApplicationName("Тестовая программа")
settings = QtCore.QSettings()
```

Второй формат позволяет указать уровень хранения настроек в виде одного из следующих элементов перечисления `Scope` из класса `QSettings`:

- `UserScope` — уровень текущего пользователя;
- `SystemScope` — уровень системы.

Эти настройки тоже будут храниться в реестре Windows.

В третьем формате задается <Название организации> — разработчика программы. В необязательном параметре application можно указать название программы. Настройки также будут храниться на уровне текущего пользователя в реестре Windows.

В четвертом формате, помимо названий организации и программы, можно задать уровень хранения настроек в виде элемента перечисления Scope.

Пятый формат позволяет сверх того задать тип хранилища в виде одного из следующих элементов перечисления Format из класса QSettings:

- NativeFormat — реестр Windows: его 32-разрядная копия, если программа работает под управлением 32-разрядной редакции Python, и 64-разрядная — для 64-разрядной редакции языка;
- IniFormat — INI-файл;
- Registry32Format — для 64-разрядных программ — 32-разрядная копия реестра, для 32-разрядных программ аналогичен атрибуту NativeFormat;
- Registry64Format — для 32-разрядных программ, запущенных под 64-разрядной Windows, — 64-разрядная копия реестра. Для 32-разрядных программ, работающих под 32-разрядной Windows, и 64-разрядных программ — аналогичен NativeFormat.

Шестой формат позволяет напрямую указать <Путь к файлу или ключу реестра>, где будут храниться настройки (в частности, этот формат пригодится в случае, если нужно прочитать какие-либо системные настройки). В качестве параметра <Тип хранилища> следует указать один из следующих элементов перечисления Format из класса QSettings:

- NativeFormat — реестр;
- IniFormat — INI-файл.

Класс QSettings поддерживает следующие методы, позволяющие узнать значения параметров, которые были заданы при создании объекта:

- organizationName() — возвращает название организации;
- applicationName() — возвращает название программы;
- scope() — возвращает обозначение уровня хранения настроек в виде элемента перечисления Scope;
- format() — возвращает обозначение формата хранения настроек в виде элемента перечисления Format;
- fileName() — возвращает путь к ключу реестра или INI-файлу с настройками.

31.2. Запись и чтение данных

Создав объект класса QSettings, мы можем приступить к сохранению настроек или чтению их.

31.2.1. Базовые средства записи и чтения данных

Для выполнения простейших операций по записи и чтению данных класс QSettings предоставляет следующие методы:

- setValue(<Имя значения>, <Значение>) — записывает заданное <Значение> под указанным в виде строки <Именем>. <Значение> может быть любого типа;

- `value(<Имя значения>[, defaultValue=None])` — считывает значение с указанным в виде строки `<Именем>` и возвращает его в качестве результата. В необязательном параметре `defaultValue` можно указать значение, которое будет возвращено, если значение с заданным `<Именем>` не будет найдено;
- `remove(<Имя значения>)` — удаляет значение с заданным в виде строки `<Именем>`. Если в качестве параметра была передана пустая строка, будут удалены все значения, что находятся в хранилище;
- `contains(<Имя значения>)` — возвращает `True`, если значение с заданным `<Именем>` существует в хранилище, и `False` — в противном случае;
- `childKeys()` — возвращает список имен имеющихся в хранилище значений;
- `clear()` — удаляет все значения из хранилища;
- `sync()` — выполняет принудительную запись всех выполненных в хранилище изменений в реестр или файл.

В листинге 31.1 приведен код программы, которая записывает в хранилище настроек (поскольку формат хранения не задан, в качестве хранилища выступает реестр) число, строку и объект класса `QSize`. Далее она считывает все сохраненные ранее значения, выводит их на экран и проверяет, присутствует ли в хранилище настроек значение, которое заведомо там не сохранялось. Наконец, она очищает хранилище.

Листинг 31.1. Использование базовых средств хранения настроек

```
from PyQt6 import QtCore, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
settings = QtCore.QSettings("Прохоренок и Дронов", "Тест 1")
v1 = 123
v2 = "Python"
v3 = QtCore.QSize(640, 480)
print(v1, v2, v3, sep=" | ")
print("Сохраняем настройки")
settings.setValue("Значение 2", v2)
settings.setValue("Значение 3", v3)
settings.sync()
print("Считываем настройки")
lv1 = settings.value("Значение 1")
lv2 = settings.value("Значение 2")
lv3 = settings.value("Значение 3")
print(lv1, lv2, lv3, sep=" | ")
if settings.contains("Значение 4"):
    print("Значение 4 в хранилище присутствует")
else:
    print("Значение 4 в хранилище отсутствует")
print("Очищаем хранилище")
settings.clear()
```

В консоли эта программа выведет:

```
123 | Python | PyQt6.QtCore.QSize(640, 480)
Сохраняем настройки
```

```
Считываем настройки
123 | Python | PyQt6.QtCore.QSize(640, 480)
Значение 4 в хранилище отсутствует
Очищаем хранилище
```

Теперь прочитаем из реестра путь к системному каталогу **Документы**, для чего воспользуемся пятым форматом конструктора класса `QSettings` (см. *разд. 31.1*):

```
>>> settings = QtCore.QSettings("HKEY_CURRENT_USER\\Software\\" +
                                "Microsoft\\Windows\\CurrentVersion\\" +
                                "Explorer\\Shell Folders",
                                QtCore.QSettings.Format.NativeFormat)
>>> settings.value("Personal")
    'D:\\Data\\Документы'
```

31.2.2. Группировка сохраняемых значений. Ключи

PyQt позволяет объединять сохраняемые в хранилище настроек значения по какому-либо признаку в особые группы, называемые *ключами*. Каждый ключ способен содержать произвольное количество значений, а разные ключи могут содержать значения с одинаковыми именами. Для создания ключей можно применить два способа: простой и сложный.

Простой способ заключается в том, что в первом параметре метода `setValue()` имя значения предваряется именем ключа, в которое его следует поместить, и отделяется от него прямым слешем:

```
settings.setValue("Ключ 1/Значение 1", v1)
. . .
lv1 = settings.value("Ключ 1/Значение 1")
```

Ключи можно вкладывать внутрь других ключей:

```
settings.setValue("Ключ 2/Вложенный ключ 1/Значение 4", v4)
. . .
lv4 = settings.value("Ключ 2/Вложенный ключ 1/Значение 4")
```

Сложный способ (который на самом деле не так уж и сложен) пригодится, если нужно сохранить в одном ключе сразу несколько значений или прочитать ряд значений из одного и того же ключа. Для его реализации следует выполнить следующие шаги:

1. Вызвать метод `beginGroup(<Имя или путь ключа>)` класса `QSettings`, который предпишет PyQt с этого момента работать с заданным ключом. В качестве параметра методу передается имя ключа, к содержимому которого следует обратиться, или же целый «путь», если нужно создать вложенный ключ. Примеры:

   ```
   settings.beginGroup("Ключ 1")
   settings.beginGroup("Ключ 2/Вложенный ключ 1")
   ```

2. Выполнить запись или чтение нужных значений с помощью методов `setValue()` и `value()` (см. *разд. 31.2.1*). В этом случае значения будут записаны в ключ, указанный в предыдущем вызове метода `beginGroup()`, или же прочитаны из этого ключа.

3. Вызвать метод `endGroup()` класса `QSettings`, который завершит работу с ключом.

Использование методов `beginGroup()` и `endGroup()` имеет ряд особенностей, о которых нам обязательно следует знать.

- Методы `remove()`, `contains()` и `childKeys()`, если их вызовы помещены между вызовами упомянутых ранее методов, действуют только внутри указанного в вызове метода `beginGroup()` ключа. Так, метод `contains()` будет искать значение с указанным именем только в текущем ключе.
- Вызов метода `remove()` с передачей ему в качестве параметра пустой строки удалит все значения, сохраненные в текущем ключе, не затрагивая содержимое других ключей и «корня» хранилища.
- Список, возвращаемый методом `childKeys()`, будет включать не только имена значений, но и имена всех ключей, вложенных в текущий ключ.
- Для проверки существования какого-либо ключа можно использовать метод `contains()`, вызвав его в ключе предыдущего уровня или же в «корне» хранилища, если ключ никуда не вложен.
- Для удаления ключа можно использовать метод `remove()`, который в этом случае вызывается точно так же.

При использовании ключей могут пригодиться три следующих метода класса `QSettings`:

- `group()` — возвращает строку с именем текущего ключа;
- `childGroups()` — возвращает список с именами всех ключей, что имеются в текущем;
- `allKeys()` — возвращает список полных путей ко всем значениям, что имеются в хранилище, включая значения, которые сохранены в ключах.

В листинге 31.2 приведен код программы, окно которой сохраняет свое местоположение при завершении и восстанавливает при запуске. Помимо этого, при нажатии специальной кнопки выполняется сохранение текста, занесенного пользователем в поле ввода. Местоположение окна и введенный текст сохраняются в разных группах.

Листинг 31.2. Использование ключей

```
from PyQt6 import QtCore, QtWidgets
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                                   flags=QtCore.Qt.WindowType.Window)
        self.setWindowTitle("Использование ключей")
        self.settings = QtCore.QSettings("Прохоренок и Дронов",
                                         "Использование ключей")
        vbox = QtWidgets.QVBoxLayout()
        self.txtLine = QtWidgets.QLineEdit(parent=self)
        vbox.addWidget(self.txtLine)
        btnSave = QtWidgets.QPushButton("&Сохранить текст")
        btnSave.clicked.connect(self.saveText)
        vbox.addWidget(btnSave)
        self.setLayout(vbox)
        if self.settings.contains("Окно/Местоположение"):
            self.setGeometry(self.settings.value("Окно/Местоположение"))
```

```
        else:
            self.resize(200, 50)
        if self.settings.contains("Данные/Текст"):
            self.txtLine.setText(self.settings.value("Данные/Текст"))

    def closeEvent(self, evt):
        self.settings.beginGroup("Окно")
        self.settings.setValue("Местоположение", self.geometry())
        self.settings.endGroup()

    def saveText(self):
        self.settings.beginGroup("Данные")
        self.settings.setValue("Текст", self.txtLine.text())
        self.settings.endGroup()

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec())
```

31.2.3. Запись списков

Для записи в хранилище настроек списка каких-либо значений следует выполнить следующую последовательность действий:

1. Вызвать метод `beginWriteArray(<Имя или путь ключа>[, size=-1])` класса `QSettings`. Первым параметром методу передается `<Имя или путь ключа>`, в котором будут сохранены элементы списка. В необязательном параметре `size` можно указать размер списка — если задать значение `-1` или вообще опустить этот параметр, PyQt определит размер сохраняемого списка самостоятельно.
2. Перебрать сохраняемый список в цикле, внутри которого выполнить следующие действия:
   - вызвать метод `setArrayIndex(<Индекс записываемого элемента>)` класса `QSettings`, тем самым указав PyQt, что сейчас будет записан элемент списка с заданным индексом;
   - собственно, записать элемент списка, индекс которого был задан ранее.
3. Закончив запись элементов списка, по завершении выполнения цикла вызвать метод `endArray()` класса `QSettings`. Это послужит сигналом окончания записи списка.

Прочитать список из хранилища настроек можно точно таким же образом, только на *шаге 1* вместо метода `beginWriteArray()` следует вызвать метод `beginReadArray(<Имя или путь ключа>)` того же класса `QSettings`. Метод в качестве результата вернет размер списка, ранее сохраненный методом `beginWriteArray()`. Этот размер понадобится для формирования цикла, который будет выполнять выборку элементов списка.

На *шаге 2*, после вызова метода `setArrayIndex()`, можно пользоваться методами `remove()` и `contains()` для удаления и проверки существования значения соответственно. Эти методы будут действовать только для элемента списка с указанным в вызове метода `setArrayIndex()` индексом.

При использовании описанного здесь подхода в хранилище настроек создается следующая структура ключей и значений:

- ключ, чье имя (путь) было задано в вызове метода `beginWriteArray()`. В этом ключе будут созданы:
  - значение `size` — хранящее размер записанного списка. Этот размер будет возвращен методом `beginReadArray()`;
  - ключ, чье имя совпадает с индексом, заданным очередным вызовом метода `setArrayIndex()` и увеличенным на единицу.

    В этом ключе будут созданы значения, записываемые последующими вызовами метода `setValue()`.

В качестве примера рассмотрим листинг 31.3, в котором приведен код, формирующий список строк, записывающий его в хранилище настроек, считывающий впоследствии и выводящий на экран.

Листинг 31.3. Запись и чтение списков

```
from PyQt6 import QtCore, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
settings = QtCore.QSettings("Прохоренок и Дронов", "Тест 2")
l = ["Python", "Ruby", "PHP", "JavaScript"]
print(l)
print("Сохраняем список")
settings.beginWriteArray("Список")
for i, el in enumerate(l):
    settings.setArrayIndex(i)
    settings.setValue("Элемент", el)
settings.endArray()
settings.sync()
print("Считываем список")
ll = []
lSize = settings.beginReadArray("Список")
for i in range(lSize):
    settings.setArrayIndex(i)
    ll.append(settings.value("Элемент"))
settings.endArray()
print(ll)
settings.clear()
```

В консоли эта программа выведет следующее:

```
['Python', 'Ruby', 'PHP', 'JavaScript']
Сохраняем список
Считываем список
['Python', 'Ruby', 'PHP', 'JavaScript']
```

31.3. Вспомогательные методы класса *QSettings*

Теперь познакомимся с вспомогательными методами класса `QSettings`, которые могут пригодиться в некоторых случаях.

- `status()` — возвращает обозначение состояния, в котором пребывает хранилище настроек после выполнения очередной операции, выраженное в виде одного из следующих элементов перечисления `Status` класса `QSettings`:
  - `NoError` — операция была успешно выполнена, и никаких ошибок не возникло;
  - `AccessError` — возникла ошибка доступа к реестру или INI-файлу (возможно, была произведена попытка чтения недоступного для текущего пользователя ключа реестра, файла или же запись в файл, недоступный для записи);
  - `FormatError` — была выполнена попытка открыть некорректно сформированный INI-файл;
- `isWritable()` — возвращает `True`, если хранилище настроек доступно для записи, и `False` — в противном случае (например, если хранилищем является INI-файл, на запись в который текущий пользователь не имеет прав).

31.4. Где хранятся настройки?

Если местоположение хранилища настроек не указано явно (т. е. используются форматы вызова конструктора класса `QSettings`, отличные от шестого, подробности — в *разд. 31.1*), настройки будут храниться по следующим местоположениям:

- при сохранении в реестр Windows (указаны типы хранилища `NativeFormat`, `Registry32Format` или `Registry64Format`):
  - если выполняется сохранение на уровне текущего пользователя (указан диапазон `UserScope`):
    - когда название программы указано (т. е. задан параметр `application` конструктора) — в ветви `HKEY_CURRENT_USER\Software\<Название организации>\<Название программы>`;
    - когда название программы не указано (т. е. параметр `application` конструктора не задан) — в ветви `HKEY_CURRENT_USER\Software\<Название организации>\OrganizationDefaults` (похоже, что в таком случае PyQt считает, что заданные настройки применяются сразу ко всем программам, разработанным этой организацией);
  - если выполняется сохранение на уровне системы (указан диапазон `SystemScope`):
    - когда название программы указано — в ветви `HKEY_LOCAL_MACHINE\Software\<Название организации>\<Название программы>`;
    - когда название программы не указано — в ветви `HKEY_LOCAL_MACHINE\Software\<Название организации>\OrganizationDefaults`.

  Программы, работающие под управлением 32-разрядной редакции Python на 64-разрядной редакции Windows, если указан тип хранилища, отличный от `Registry64Format`, сохраняют настройки в ветвях `HKEY_LOCAL_MACHINE\Software\WOW6432node\<Название`

организации>\<Название программы> и HKEY_LOCAL_MACHINE\Software\WOW6432node\<Название организации>\OrganizationDefaults соответственно;

- при сохранении в INI-файл (указан тип хранилища IniFormat):
  - если выполняется сохранение на уровне текущего пользователя:
    - когда название программы указано — в файле <каталог пользовательского профиля>\AppData\Roaming\<Название организации>\<Название программы>.ini;
    - когда название программы не указано — в файле <каталог пользовательского профиля>\AppData\Roaming\<Название организации>.ini;
  - если выполняется сохранение на уровне системы:
    - когда название программы указано — в файле <системный диск>:\ProgramData\<Название организации>\<Название программы>.ini;
    - когда название программы не указано — в файле <системный диск>:\ProgramData\<Название организации>.ini.

ГЛАВА 32

Программа «Судоку»

В завершение в качестве примера напишем на языке Python с применением библиотеки PyQt программу «Судоку». Она предназначена для создания и решения головоломок судоку и позволит, помимо всего прочего, сохранять головоломки в файлах, загружать их из файлов и выводить на печать.

Изначально программа была разработана Н. Прохоренком в 2011 году на языке C++. Для этой книги она переписана на Python и несколько усовершенствована В. Дроновым в конце 2017 года.

32.1. Правила судоку

Судоку — традиционная японская числовая головоломка (иногда ее неправильно называют *магическим квадратом*). Далее приведены правила ее решения (полное описание судоку можно найти по интернет-адресу **https://ru.wikipedia.org/wiki/Судоку**).

- Поле судоку представляет собой квадрат, разбитый на 81 ячейку (9 столбцов по 9 строк). Каждые 9 ячеек объединены в группу 3 × 3 — итого получается 9 групп.
- В каждую ячейку поля можно подставить только одну цифру от 1 до 9.
- Ячейки группы не должны содержать одинаковых цифр.
- Одна и та же цифра должна присутствовать в каждой строке и каждом столбце поля только один раз.

На рис. 32.1 приведен пример решенной судоку.

5	4	7	8	6	3	9	2	1
3	8	1	2	9	7	6	5	4
2	9	6	5	1	4	8	3	7
6	2	5	9	3	1	4	7	8
7	3	8	4	2	6	1	9	5
4	1	9	7	5	8	3	6	2
9	7	3	1	8	2	5	4	6
1	6	2	3	4	5	7	8	9
8	5	4	6	7	9	2	1	3

Рис. 32.1. Решенная судоку (группы выделены утолщенными рамками)

Как правило, судоку решают вдвоем: один игрок произвольно расставляет цифры в некоторых ячейках поля, а второй, собственно, решает головоломку.

32.2. Описание программы «Судоку»

Программа «Судоку» выполнена в традиционном ключе, присущем обычным Windows-программам. Ее окно (рис. 32.2) включает в себя главное меню, панель инструментов, основное содержимое — поле судоку и набор кнопок для установки цифр в ячейки, и строку состояния.

В самом поле судоку группы ячеек выделены различными цветами фона: оранжевым и светло-серым. Установленные в них цифры выводятся черным цветом.

Одна из ячеек является *активной* — именно с активной ячейкой осуществляется взаимодействие. Активная ячейка закрашена желтым цветом (на рис. 32.2 это ячейка с цифрой 4 в правом нижнем углу поля). Чтобы сделать какую-либо ячейку активной, следует:

- либо, пользуясь клавишами-стрелками, переместить на нее желтый фокус выделения;
- либо просто щелкнуть на ней мышью.

Чтобы установить в активную ячейку какую-либо цифру, следует:

- либо нажать соответствующую кнопку в наборе, расположенном ниже поля;
- либо нажать соответствующую цифровую клавишу.

Чтобы убрать цифру из активной ячейки, нужно:

- либо нажать кнопку **X**, находящуюся в расположенном под полем наборе;
- либо нажать клавишу пробела, <Backspace> или <Del>.

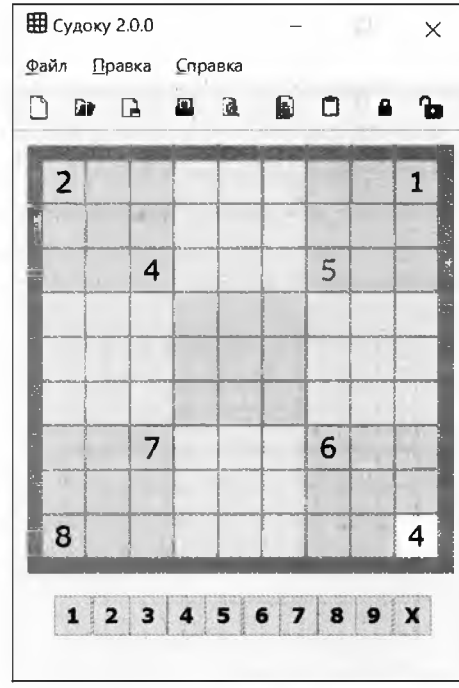


Рис. 32.2. Окно программы «Судоку»

Чтобы случайно не занести в какую-либо ячейку другую цифру, есть возможность заблокировать ее. Для этого следует сделать нужную ячейку активной и нажать клавишу <F2>. В заблокированной ячейке цифра выводится красным шрифтом (на рис. 32.2 заблокирована ячейка с цифрой 5). Отметим, что блокировать можно только ячейки, содержащие цифры.

Снять блокировку с ячейки можно, сделав ее активной и нажав клавишу <F4>.

Главное меню программы содержит следующие пункты:

- меню **Файл**:
  - **Новый** (с ним связана комбинация клавиш <Ctrl>+<N>) — очистка поля;
  - **Открыть** (<Ctrl>+<O>) — загрузка сохраненной ранее головоломки из выбранного пользователем файла;
  - **Сохранить** (<Ctrl>+<S>) — сохранение головоломки в файле с указанным пользователем именем.

    При выборе этого пункта производится сохранение в полном формате — т. е. для каждой ячейки, помимо находящейся в ней цифры, сохраняется признак того, заблокирована ли она;
  - **Сохранить компактно** — сохранение головоломки в файле в компактном формате.

    Компактный формат не предусматривает хранения признака блокировки ячейки. Вследствие этого файл, сохраненный в компактном формате, вдвое меньше полноформатного.

    При открытии файла, сохраненного в компактном формате, производится автоматическая блокировка всех ячеек, содержащих цифры;
  - **Печать** (<Ctrl>+<P>) — вывод головоломки на печать;
  - **Предварительный просмотр** — просмотр головоломки в том виде, в котором она будет выведена на печать;
  - **Параметры страницы** — настройка параметров печатаемой страницы;
  - **Выход** (<Ctrl>+<Q>) — завершение работы программы;
- меню **Правка**:
  - **Копировать** (<Ctrl>+<C>) — копирование головоломки в буфер обмена в полном формате (аналогичном тому, в котором сохраняется файл при выборе пункта **Сохранить** меню **Файл**);
  - **Копировать компактно** — копирование головоломки в буфер обмена в компактном формате (аналогичном тому, в котором сохраняется файл при выборе пункта **Сохранить компактно** меню **Файл**);
  - **Копировать для Excel** — копирование головоломки в буфер обмена в формате, предназначенном для вставки в таблицу Microsoft Excel;
  - **Вставить** (<Ctrl>+<V>) — вставка головоломки, скопированной ранее в любом формате.

    Если головоломка была скопирована в компактном формате, все ячейки, содержащие цифры, будут автоматически заблокированы;
  - **Вставить из Excel** — вставка из буфера обмена головоломки, скопированной из таблицы Microsoft Excel;
  - **Блокировать** (<F2>) — блокировка активной ячейки;

- **Блокировать все** (<F3>) — блокировка всех ячеек. Блокируются только ячейки, содержащие цифры;
- **Разблокировать** (<F4>) — разблокировка активной ячейки;
- **Разблокировать все** (<F5>) — разблокировка всех ячеек;

♦ меню **Справка**:

- **О программе** — вывод окна со сведениями о программе «Судоку»;
- **О Qt** — вывод окна со сведениями о фреймворке Qt.

При наведении курсора мыши на любой пункт меню в строке состояния появляется его развернутое описание.

В панели инструментов присутствуют кнопки (в порядке слева направо): **Новый**, **Открыть**, **Сохранить**, **Печать**, **Предварительный просмотр**, **Копировать**, **Вставить**, **Блокировать все** и **Разблокировать все**. Они выполняют те же действия, что и одноименные пункты меню. При наведении курсора мыши на кнопку панели инструментов в строке состояния появляется ее развернутое описание, а рядом с кнопкой спустя небольшой промежуток времени появляется всплывающая подсказка (на рис. 32.2 курсор мыши наведен на кнопку **Сохранить**).

Сохранение головоломок выполняется в текстовых файлах с расширением svd. Форматы, в которых хранятся головоломки, будут описаны позже.

Окно программы при закрытии сохраняет свое местоположение и впоследствии, после следующего запуска, восстанавливает его.

32.3. Разработка программы

32.3.1. Подготовительные действия

Создадим где-либо на диске каталог sudoku, в котором будут находиться все файлы нашей будущей программы. В этом каталоге создадим два вложенных каталога:

♦ images — для хранения значков, которые будут выводиться в пунктах меню и кнопках панели инструментов, а также значок самой программы;

♦ modules — для хранения файлов с программными модулями Python, которые напишем впоследствии.

Найдем в Интернете значки для представления самой программы, пунктов меню и кнопок панели инструментов. Сохраним их в каталоге images под именами svd.png (значок программы), new.png, open.png, save.png, print.png, preview.png, copy.png, paste.png, lock.png и unlock.png (значки кнопок).

32.3.2. Класс *MyLabel*: ячейка поля судоку

Начнем с создания класса `MyLabel`, который будет представлять отдельную ячейку поля судоку, — 81 такой компонент и составит это поле.

Наш класс должен «уметь» выводить на экран цифру, занесенную в ячейку, отображать обычное, активное и заблокированное состояния, реагировать на щелчки мышью, чтобы сообщить компоненту поля судоку, что та или иная ячейка стала активной. Помимо этого,

ячейка должна иметь возможность принимать фоновый цвет — ведь именно разными цветами фона мы будем выделять группы ячеек на поле.

Визуально ячейка должна иметь размеры 30 × 30 пикселов и выводить цифру, выровненную по середине без отступов от границ компонента.

Большую часть необходимой функциональности мы можем получить, просто сделав класс `MyLabel` производным от класса надписи `QLabel`. Тогда, чтобы вывести на экран цифру, можно просто задать ее в качестве содержимого надписи, воспользовавшись унаследованным методом `setText()`, — также несложно будет указать необходимые размеры и выравнивание. А задать для текста и фона нужные цвета мы сможем, привязав к ячейке таблицу стилей, для чего воспользуемся методом `setStyleSheet()`, опять же, унаследованным.

Теперь подумаем, какие атрибуты должен поддерживать класс `MyLabel`:

- `colorYellow`, `colorOrange`, `colorGrey`, `colorBlack` и `colorRed` — будут хранить RGB-коды соответственно желтого, оранжевого, светло-серого, черного и красного цветов. Поскольку значения этих атрибутов будут одинаковыми для всех объектов класса ячейки, мы сделаем их атрибутами класса (если же сделать их обычными атрибутами, каждый объект будет хранить свой собственный набор этих атрибутов, что приведет к избыточному расходу оперативной памяти);
- `isCellChange` — значение `True` сообщит о том, что ячейка разблокирована, а значение `False` — что она заблокирована;
- `fontColorCurrent` — текущий цвет текста: черный или красный;
- `bgColorDefault` — заданный при создании ячейки цвет фона: оранжевый или светло-серый. Он понадобится, чтобы перевести ячейку из активного в неактивное состояние;
- `bgColorCurrent` — текущий цвет фона: желтый, оранжевый или светло-серый;
- `id` — порядковый номер ячейки. Он понадобится, чтобы при щелчке мышью на ячейке сообщить компоненту поля судоку, какая ячейка стала активной.

Компонент `MyLabel` будет входить в состав компонента поля, который мы напишем чуть позже. Он должен уведомлять компонент поля, когда текущая ячейка становится активной после щелчка мышью. Наилучший способ сделать это — объявить сигнал, который будет генерироваться при щелчке. Обрабатывая этот сигнал, поле всегда будет «в курсе», на какой ячейке был выполнен щелчок.

Мы объявим в ячейке сигнал `cellChangeFocus`. Он будет передавать обработчику единственный параметр целочисленного типа — номер ячейки, на которой пользователь щелкнул мышью.

Теперь определим набор необходимых методов нашего класса и представим в общих чертах, что должен делать каждый из них:

- `mousePressEvent()` — этот метод следует переопределить, чтобы получить возможность обрабатывать щелчки мышью. Внутри него мы будем генерировать сигнал `cellChangeFocus`;
- `showColorCurrent()` — задаст для ячейки цвета текста и фона, взятые из атрибутов `fontColorCurrent` и `bgColorCurrent` соответственно, и тем самым обновит визуальное состояние ячейки, или, как говорят программисты, перерисует ее. Этот метод мы будем вызывать после перевода ячейки из неактивного состояния в активное, из разблокированного — в заблокированное и наоборот;
- `setCellFocus()` — переведет ячейку из неактивного состояния в активное. Сделать это очень просто — мы занесем в атрибут `bgColorCurrent` код желтого цвета (который хра-

нится в атрибуте класса colorYellow) и вызовем метод showColorCurrent(), чтобы обновить визуальное представление ячейки;

- clearCellFocus() — переведет ячейку из активного состояния в неактивное. Мы занесем в атрибут bgColorCurrent значение цвета фона, взятое из атрибута bgColorDefault (он, как мы помним, хранит код изначального цвета фона), и вызовем метод showColorCurrent(), который перерисует ячейку;
- setCellBlock() — переведет ячейку из разблокированного состояния в заблокированное. Здесь мы занесем в атрибут isCellChange значение False, тем самым указывая, что ячейка заблокирована, присвоим атрибуту fontColorCurrent код красного цвета, хранящийся в атрибуте класса colorRed, и перерисуем ячейку вызовом метода showColorCurrent();
- clearCellBlock() — переведет ячейку из заблокированного состояния в разблокированное. Для этого достаточно присвоить атрибуту isCellChange значение True, занести в атрибут fontColorCurrent значение черного цвета из атрибута объекта colorBlack и не забыть перерисовать ячейку вызовом метода showColorCurrent();
- setNewText() — занесет в ячейку новое число, переданное ему в качестве единственного параметра в виде строки. Здесь нужно предварительно проверить, не заблокирована ли ячейка (не хранится ли в атрибуте isCellChange значение False), и уже потом вызывать унаследованный от класса QLabel метод setText();
- конструктор должен принимать в качестве обязательных параметров номер создаваемой ячейки (он будет занесен в атрибут id), цвет ее фона (его мы присвоим атрибутам bgColorDefault и bgColorCurrent), а также необязательный параметр родителя parent. Еще он должен присвоить атрибуту isCellChange значение True — вновь созданная ячейка изначально должна быть разблокирована. Напоследок он перерисует ячейку, вызвав метод showColorCurrent().

Код класса MyLabel относительно невелик, несложен и полностью приведен в листинге 32.1. Его следует сохранить в модуле mylabel.pyw в каталоге modules.

> ***Электронный архив***
>
> Напомним, что файлы с кодами разрабатываемой в этой главе программы находятся в папке *sudoku* сопровождающего книгу электронного архива (см. *приложение*).

Листинг 32.1. Класс MyLabel

```
from PyQt6 import QtCore, QtWidgets

class MyLabel(QtWidgets.QLabel):
    colorYellow = "#FFFF90"
    colorOrange = "#F5D8C1"
    colorGrey = "#E8E8E8"
    colorBlack = "#000000"
    colorRed = "#D77A38"

    changeCellFocus = QtCore.pyqtSignal(int)

    def __init__(self, id, bgColor, parent=None):
        QtWidgets.QLabel.__init__(self, parent)
```

```
        self.setAlignment(QtCore.Qt.AlignmentFlag.AlignCenter)
        self.setFixedSize(30, 30)
        self.setMargin(0)
        self.setText("")
        if id < 0 or id > 80:
            id = 0
        self.id = id
        self.isCellChange = True
        self.fontColorCurrent = self.colorBlack
        self.bgColorDefault = bgColor
        self.bgColorCurrent = bgColor
        self.showColorCurrent()

    def mousePressEvent(self, evt):
        self.changeCellFocus.emit(self.id)
        QtWidgets.QLabel.mousePressEvent(self, evt)

    def showColorCurrent(self):
        self.setStyleSheet("background-color:" + self.bgColorCurrent +
                           ";color:" + self.fontColorCurrent + ";")

    def setCellFocus(self):
        self.bgColorCurrent = self.colorYellow
        self.showColorCurrent()

    def clearCellFocus(self):
        self.bgColorCurrent = self.bgColorDefault
        self.showColorCurrent()

    def setCellBlock(self):
        self.isCellChange = False
        self.fontColorCurrent = self.colorRed
        self.showColorCurrent()

    def clearCellBlock(self):
        self.isCellChange = True
        self.fontColorCurrent = self.colorBlack
        self.showColorCurrent()

    def setNewText(self, text):
        if self.isCellChange:
            self.setText(text)
```

Отметим пару чисто технических деталей:

- в методе `mousePressEvent()`, выполнив все необходимые действия, а именно сгенерировав сигнал `changeCellFocus`, мы в обязательном порядке вызываем тот же метод суперкласса. Если этого не сделать, возможны проблемы;
- в методе `showColorCurrent()` мы вызовом метода `setStyleSheet()` привязываем к нашему компоненту таблицу стилей, которая установит для ячейки цвета текста и фона (прочие параметры для компонента мы установим в таблице стилей, привязанной к компоненту основного окна, которую создадим позже).

32.3.3. Класс *Widget*: поле судоку

Класс `Widget` представит само поле судоку, составленное из 81 компонента `MyLabel`, написанного ранее, и набора из 10 обычных кнопок, с помощью которых пользователь будет вставлять цифры в ячейки.

Сразу после создания компонент `Widget` должен вывести на экран поле судоку и набор кнопок. Он даст пользователю возможность делать ячейки активными с помощью клавиш-стрелок (активизация посредством щелчка мышью уже реализована нами в классе `MyLabel`), устанавливать в ячейки цифры либо клавишами, либо кнопками из набора и удалять цифры, опять же, клавишами или специальной кнопкой. Это основная функциональность компонента.

Что касается его дополнительной функциональности, то ее мы будем реализовывать по частям. И сейчас мы сделаем лишь очистку поля, блокировку и разблокировку его ячеек.

Поскольку функциональности, специфической для какого-либо имеющегося в библиотеке компонента, нам не требуется, класс `Widget` сделаем производным от класса `QWidget`.

Необходимый нам набор атрибутов класса очень невелик:

- `cells` — массив ячеек поля судоку — объектов класса `MyLabel`. Мы сохраним этот массив в атрибуте, поскольку в других методах этого класса нам понадобится получать к нему доступ;
- `idCellInFocus` — порядковый номер ячейки, являющейся активной в настоящий момент. Его нужно знать в любом случае — хотя бы для того, чтобы визуально выделить активную ячейку, вызвав у нее метод `setCellFocus()` (см. *разд. 32.3.2*).

Набор методов, поддерживаемый классом `Widget`, будет более объемным (даже с учетом того, что мы еще не реализовали в классе дополнительную функциональность). Поскольку эти методы сложнее таковых у класса `MyLabel`, рассмотрим здесь их вкратце, не углубляясь в технические детали, — полное их описание будет приведено далее, вместе с их кодом:

- `onChangeCellFocus()` — обработчик сигналов `changeCellFocus` всех ячеек, что имеются в поле (упомянутый здесь сигнал, как мы помним, генерируется при щелчке на ячейке мышью). Он получит с единственным параметром номер ячейки, на которой был выполнен щелчок, и активизирует ее;
- `keyPressEvent()` — этот метод следует переопределить, чтобы получить возможность обрабатывать нажатия клавиш клавиатуры. Здесь мы будем, в зависимости от нажатой клавиши, перемещать по полю фокус выделения, ставить цифры в ячейки и очищать их;
- `onBtn<N>Clicked()` (где `<N>` — число от `0` до `8` или буква `X`) — обработчики щелчков на кнопках **1**...**9** и **X**. Они будут ставить в ячейку соответствующую цифру или же очищать ячейку;
- `onClearAllCells()` — очистит поле судоку. Будет вызываться при выборе пункта **Новый** меню **Файл** или нажатии кнопки **Новый** панели инструментов;
- `onBlockCell()` — заблокирует активную ячейку, если она содержит цифру и еще не заблокирована. Будет вызываться при выборе пункта **Заблокировать** меню **Правка**;
- `onBlockCells()` — заблокирует все ячейки, содержащие цифры и не заблокированные. Будет вызываться при выборе пункта **Заблокировать все** меню **Правка** или нажатии кнопки **Заблокировать все** панели инструментов;
- `onClearBlockCell()` — разблокирует активную ячейку, если она заблокирована. Будет вызываться при выборе пункта **Разблокировать** меню **Правка**;

- `onClearBlockCells()` — разблокирует все заблокированные ячейки. Будет вызываться при выборе пункта **Разблокировать все** меню **Правка** или нажатии кнопки **Разблокировать все** панели инструментов;
- конструктор, который создаст все необходимые компоненты и выполнит привязку обработчиков к сигналам.

Весь код класса `Widget` мы сохраним в модуле widget.pyw в каталоге modules. Поскольку этот код весьма велик, мы рассмотрим его по частям.

32.3.3.1. Конструктор класса *Widget*

Конструктор — самый сложный метод класса `Widget`. Поэтому мы не станем приводить полный листинг конструктора, а рассмотрим его по фрагментам.

```
from PyQt6 import QtCore, QtGui, QtWidgets
```

Помимо модулей `QtCore` и `QtWidgets`, которые понадобятся нам сейчас, мы импортируем модуль `QtGui`. Объявленные в нем классы пригодятся позже, при реализации печати.

```
from modules.mylabel import MyLabel
```

Не забываем импортировать из модуля mylabel.pyw, что хранится в каталоге modules, класс `MyLabel`, представляющий отдельную ячейку и написанный нами ранее.

```
class Widget(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFocusPolicy(QtCore.Qt.FocusPolicy.StrongFocus)
```

По умолчанию объект класса `QWidget` или производного от него класса не может принимать фокус ввода. Чтобы дать ему возможность принимать фокус ввода при щелчке мышью и переходе нажатием клавиши <Tab>, мы вызовем у него метод `setFocusPolicy()`, передав ему в качестве параметра элемент `StrongFocus` перечисления `FocusPolicy`.

```
        vBoxMain = QtWidgets.QVBoxLayout()
```

Поскольку само поле и набор кнопок будут располагаться друг над другом, мы используем для их размещения контейнер `QVBoxLayout`.

```
        frame1 = QtWidgets.QFrame()
        frame1.setStyleSheet(
            "background-color:#9AA6A7;border:1px solid #9AA6A7;")
```

Поле (которое будет создано контейнером-сеткой `QGridLayout`) поместим в панель с рамкой `QFrame`. У этой панели с помощью таблицы стилей укажем тонкую рамку и фон одинакового темно-серого цвета. Панель с рамкой займет все выделенное под него пространство контейнера, а поле судоку поместится в центре этой панели. В результате этого будет казаться, что поле окружено толстой темно-серой рамкой, что выглядит весьма эффектно.

```
        grid = QtWidgets.QGridLayout()
        grid.setSpacing(0)
```

Создаем сетку `QGridLayout`, которая сформирует само поле.

```
        idColor = (3, 4, 5, 12, 13, 14, 21, 22, 23, 27, 28, 29, 36, 37, 38, 45,
                   46, 47, 33, 34, 35, 42, 43, 44, 51, 52, 53, 57, 58, 59, 66,
                   67, 68, 75, 76, 77)
```

Объявляем массив, хранящий номера ячеек, которые должны быть выделены светло-серым фоном.

```
self.cells = [MyLabel(i, MyLabel.colorGrey if i in idColor else \
                      MyLabel.colorOrange) for i in range(0, 81)]
```

Создаем список из 81 ячейки `MyLabel`, сохранив его в атрибуте `cells` класса `Widget`. Здесь мы используем выражение генератора списка, которое позволит нам радикально упростить код. Если номер создаваемой ячейки имеется в объявленном ранее массиве, задаем для нее светло-серый цвет фона, в противном случае — оранжевый.

```
self.cells[0].setCellFocus()
self.idCellInFocus = 0
```

Делаем активной ячейку с номером 0 и заносим тот же номер в атрибут `idCellInFocus` класса `Widget`. В результате изначально активной станет самая первая ячейка поля.

```
i = 0
for j in range(0, 9):
    for k in range(0, 9):
        grid.addWidget(self.cells[i], j, k)
        i += 1
```

Помещаем все созданные ячейки в сетку.

```
for cell in self.cells:
    cell.changeCellFocus.connect(self.onChangeCellFocus)
```

У всех ячеек задаем для сигнала `changeCellFocus` обработчик — метод `onChangeCellFocus()` класса `Widget`, пока еще не объявленный.

```
frame1.setLayout(grid)
vBoxMain.addWidget(frame1,
                   alignment=QtCore.Qt.AlignmentFlag.AlignHCenter)
```

Помещаем сетку в панель с рамкой и добавляем последнюю в контейнер `VBoxLayout`, указав для нее горизонтальное выравнивание посередине.

```
frame2 = QtWidgets.QFrame()
frame2.setFixedSize(272, 36)
```

Набор кнопок, с помощью которых будет выполняться занесение цифр в ячейки, мы поместим в другую панель с рамкой `QFrame`. Это позволит нам привязать к кнопкам таблицу стилей, задающую для них представление. У панели обязательно зададим фиксированные размеры — иначе их установит сам PyQt согласно своему разумению, которое вряд ли совпадет с нашим.

```
hbox = QtWidgets.QHBoxLayout()
hbox.setSpacing(1)
```

Кнопки будут выстроены по горизонтали, следовательно, наилучший вариант — поместить их в контейнер `QHBoxLayout`.

```
btns = []
for i in range(1, 10):
    btn = QtWidgets.QPushButton(str(i))
    btn.setFixedSize(27, 27)
    btn.setFocusPolicy(QtCore.Qt.FocusPolicy.NoFocus)
    btns.append(btn)
```

Создаем кнопки 1...9, даем им размеры 27 × 27 пикселов и добавляем в специально созданный для этого список. Также для каждой кнопки указываем, что она не должна принимать

фокус ввода (для чего вызовем у нее метод `setFocusPolicy()` с параметром `NoFocus`), — это нужно для того, чтобы поле судоку при нажатии любой из этих кнопок не теряло фокус, и пользователь смог продолжать манипулировать в нем с помощью клавиш.

```
btn = QtWidgets.QPushButton("X")
btn.setFixedSize(27, 27)
btns.append(btn)
```

Таким же образом создаем кнопку **X**, которая уберет цифру из ячейки.

```
for btn in btns:
    hbox.addWidget(btn)
```

Помещаем все кнопки в контейнер `QHBoxLayout`.

```
btns[0].clicked.connect(self.onBtn0Clicked)
btns[1].clicked.connect(self.onBtn1Clicked)
btns[2].clicked.connect(self.onBtn2Clicked)
btns[3].clicked.connect(self.onBtn3Clicked)
btns[4].clicked.connect(self.onBtn4Clicked)
btns[5].clicked.connect(self.onBtn5Clicked)
btns[6].clicked.connect(self.onBtn6Clicked)
btns[7].clicked.connect(self.onBtn7Clicked)
btns[8].clicked.connect(self.onBtn8Clicked)
btns[9].clicked.connect(self.onBtnXClicked)
```

Привязываем к сигналам `clicked` всех этих кнопок соответствующие обработчики — методы класса поля, которые объявим позже.

```
frame2.setLayout(hbox)
vBoxMain.addWidget(frame2,
                   alignment=QtCore.Qt.AlignmentFlag.AlignHCenter)
```

Помещаем контейнер с кнопками в панель с рамкой, а ее — во «всеобъемлющий» контейнер `VBoxLayout`, не забыв указать горизонтальное выравнивание посередине.

```
self.setLayout(vBoxMain)
```

И помещаем этот контейнер в компонент поля.

32.3.3.2. Прочие методы класса *Widget*

Теперь напишем код остальных методов класса `Widget`. Они существенно проще конструктора, и мы можем не рассматривать их по частям.

Листинг 32.2. Метод onChangeCellFocus()

```
def onChangeCellFocus(self, id):
    if self.idCellInFocus != id and not (id < 0 or id > 80):
        self.cells[self.idCellInFocus].clearCellFocus()
        self.idCellInFocus = id
        self.cells[id].setCellFocus()
```

Метод `onChangeCellFocus()` станет обработчиком сигнала `changeCellFocus` ячейки `MyLabel` (листинг 32.2). В качестве единственного параметра он получит номер ячейки, ставшей активной.

Сначала проверим, не совпадает ли полученный номер с тем, что хранится в атрибуте `idCellInFocus` (не щелкнул ли пользователь на активной ячейке), не меньше ли он `0` и не больше ли `80` (т. е. не вышел ли он за диапазон номеров ячеек). Если это так, выполняем операцию по переносу фокуса на ячейку с полученным в параметре номером.

Предварительно нам следует деактивировать ячейку, бывшую активной ранее (номер этой ячейки в настоящий момент хранится в атрибуте `idCellInFocus`). Извлекаем номер, получаем из списка ячеек (он хранится в атрибуте `cells`) саму эту ячейку и переводим ее в неактивное состояние вызовом метода `clearCellFocus()`. Далее заносим полученный с параметром номер в атрибут `idCellInFocus`, тем самым указывая, что ячейка с этим номером сейчас активна, и делаем ее активной, вызвав у нее метод `setCellFocus()`.

Листинг 32.3. Метод `keyPressEvent()`

```
def keyPressEvent(self, evt):
    key = evt.key()
    if key == QtCore.Qt.Key.Key_Up:
        tid = self.idCellInFocus - 9
        if tid < 0:
            tid += 81
        self.onChangeCellFocus(tid)
    elif key == QtCore.Qt.Key.Key_Right:
        tid = self.idCellInFocus + 1
        if tid > 80:
            tid -= 81
        self.onChangeCellFocus(tid)
    elif key == QtCore.Qt.Key.Key_Down:
        tid = self.idCellInFocus + 9
        if tid > 80:
            tid -= 81
        self.onChangeCellFocus(tid)
    elif key == QtCore.Qt.Key.Key_Left:
        tid = self.idCellInFocus - 1
        if tid < 0:
            tid += 81
        self.onChangeCellFocus(tid)
    elif key >= QtCore.Qt.Key.Key_1 and key <= QtCore.Qt.Key.Key_9:
        self.cells[self.idCellInFocus].setNewText(chr(key))
    elif key == QtCore.Qt.Key.Key_Delete or \
         key == QtCore.Qt.Key.Key_Backspace or \
         key == QtCore.Qt.Key.Key_Space:
        self.cells[self.idCellInFocus].setNewText("")
    QtWidgets.QWidget.keyPressEvent(self, evt)
```

Переопределенный метод `keyPressEvent()` будет обрабатывать нажатия клавиш (листинг 32.3). В качестве параметра он получит объект события клавиатуры. Мы вызовем у этого объекта метод `key()`, чтобы получить код нажатой клавиши. После чего начнем последовательно сравнивать его с кодами различных клавиш, чтобы выяснить, какая из них была нажата.

Если была нажата клавиша <↑>, следует сделать активной ячейку, расположенную строкой выше. Мы получим номер этой ячейки, вычтя из номера активной ячейки (он, как мы знаем, хранится в атрибуте `idCellInFocus`) число `9` — т. е. количество ячеек, помещающихся в строке поля. Если получившаяся разность меньше `0` (фокус выделения вышел за пределы верхней границы поля), мы прибавляем к разности `81` (количество ячеек в поле), в результате чего фокус окажется на самой последней строке поля, в том же столбце. И, наконец, вызываем метод `onChangeCellFocus()` класса поля, передав ему результирующий номер ячейки, чтобы сделать ее активной.

Если была нажата клавиша <→>, нужно сделать активной ячейку, расположенную правее. Понятно, что для получения ее номера нам следует прибавить к номеру активной ячейки единицу. Если же полученная сумма оказалась больше `80` (фокус выделения вышел за пределы верхней границы диапазона имеющихся ячеек), мы вычтем из нее то же число `81` — тогда фокус окажется на самой первой ячейке в поле. И не забываем напоследок вызвать метод `onChangeCellFocus()`.

Обработка нажатия клавиш <↓> и <←> производится аналогично. Вы можете сами разобраться, как работает выполняющий ее код.

Если была нажата клавиша <1>...<9>, мы формируем на основе ее кода соответствующий символ, воспользовавшись функцией `chr()`, вызываем у активной ячейки метод `setNewText()` и передаем ему этот символ. Так мы занесем в активную ячейку цифру, соответствующую нажатой клавише.

Случай нажатия клавиш <Backspace>, <Del> или <пробел> — самый простой. Мы вызываем у активной ячейки метод `setNewText()`, передав ему пустую строку, — так мы уберем цифру с ячейки.

В самом конце, какая бы ни была нажата клавиша, мы в обязательном порядке вызываем метод `keyPressEvent()` базового класса. Если этого не сделать, программа может повести себя непредсказуемо.

Листинг 32.4. Методы `onBtn0Clicked()`...`onBtn8Clicked()` и `onBtnXClicked()`

```
    def onBtn0Clicked(self):
        self.cells[self.idCellInFocus].setNewText("1")

    def onBtn1Clicked(self):
        self.cells[self.idCellInFocus].setNewText("2")

    def onBtn2Clicked(self):
        self.cells[self.idCellInFocus].setNewText("3")

    def onBtn3Clicked(self):
        self.cells[self.idCellInFocus].setNewText("4")

    def onBtn4Clicked(self):
        self.cells[self.idCellInFocus].setNewText("5")

    def onBtn5Clicked(self):
        self.cells[self.idCellInFocus].setNewText("6")

    def onBtn6Clicked(self):
        self.cells[self.idCellInFocus].setNewText("7")
```

```
    def onBtn7Clicked(self):
        self.cells[self.idCellInFocus].setNewText("8")

    def onBtn8Clicked(self):
        self.cells[self.idCellInFocus].setNewText("9")

    def onBtnXClicked(self):
        self.cells[self.idCellInFocus].setNewText("")
```

Методы с `onBtn0Clicked()` по `onBtn8Clicked()` и `onBtnXClicked()` занесут в ячейку цифру, соответствующую нажатой кнопке **1**...**9**, или удалят цифру, если была нажата кнопка **X** (листинг 32.4). Как они работают, понятно без дополнительных пояснений.

Листинг 32.5. Метод `onClearAllCells()`

```
    def onClearAllCells(self):
        for cell in self.cells:
            cell.setText("")
            cell.clearCellBlock()
```

Метод `onClearAllCells()` очистит поле судоку (листинг 32.5). В нем мы перебираем все имеющиеся в поле ячейки, каждую очищаем от занесенной в нее цифры и переводим в разблокированное состояние вызовом метода `clearCellBlock()` класса `MyLabel`.

Листинг 32.6. Метод `onBlockCell()`

```
    def onBlockCell(self):
        cell = self.cells[self.idCellInFocus]
        if cell.text() != "":
            if cell.isCellChange:
                cell.setCellBlock()
```

Метод `onBlockCell()` будет блокировать активную ячейку (листинг 32.6). Сначала он проверит, есть ли в ней цифра (получить ее можно вызовом унаследованного от суперкласса `QLabel` метода `text()`), поскольку блокировать можно только ячейки с цифрами. Если в ячейке нет цифры, метод ничего не сделает. В противном случае он проверит, хранится ли в атрибуте `isCellChange` блокируемой ячейки значение `True` (т. е. не заблокирована ли уже эта ячейка), и, если это так, заблокирует ячейку вызовом метода `setCellBlock()` класса `MyLabel`.

В этом методе мы предварительно извлекаем из списка активную ячейку, сохраняем ее в переменной и в дальнейшем используем для доступа к активной ячейке именно эту переменную. Такой подход позволяет несколько повысить быстродействие, поскольку обращение к переменной выполняется быстрее, чем к атрибуту класса или элементу списка.

Листинг 32.7. Метод `onBlockCells()`

```
    def onBlockCells(self):
        for cell in self.cells:
            if cell.text() and cell.isCellChange:
                cell.setCellBlock()
```

Метод `onBlockCells()`, блокирующий все ячейки, выполняет перебор всех ячеек и блокирует любую из них, если она содержит цифру и еще не заблокирована (листинг 32.7).

Листинг 32.8. Метод `onClearBlockCell()`

```
def onClearBlockCell(self):
    cell = self.cells[self.idCellInFocus]
    if not cell.isCellChange:
        cell.clearCellBlock()
```

Метод `onClearBlockCell()`, предназначенный для разблокирования активной ячейки, предварительно проверит, хранится ли в ее атрибуте `isCellChange` значение `False` (т. е. заблокирована ли эта ячейка). И только после этого он вызывает метод `clearCellBlock()` класса `MyLabel`, чтобы разблокировать ячейку (листинг 32.8).

Листинг 32.9. Метод `onClearBlockCells()`

```
def onClearBlockCells(self):
    for cell in self.cells:
        if not cell.isCellChange:
            cell.clearCellBlock()
```

Метод `onClearBlockCells()`, разблокирующий все ячейки поля (листинг 32.9), очень прост, и вы, уважаемые читатели, сами поймете, как он работает.

32.3.4. Класс *MainWindow*: основное окно программы

Класс `MainWindow` представляет основное окно программы «Судоку». Оно включает компонент `Widget` — т. е. поле судоку, главное меню, панель инструментов и строку состояния.

Класс основного окна, в силу его сложности, мы также будем писать по частям. В настоящий момент мы реализуем в нем только часть всех функций программы: операции очистки поля, завершения программы, блокировки и разблокировки ячеек и получения справочных сведений. Мы также реализуем сохранение и восстановление местоположения окна и начнем разработку функции печати. Остальная функциональность будет добавлена позже.

Класс `MainWindow` мы сделаем производным от класса главного окна `MainWindow`. Это позволит нам без проблем разместить в окне поле судоку, создать главное меню, панель инструментов и строку состояния.

Подумаем, какие атрибуты мы объявим в классе основного окна:

- `sudoku` — объект класса `Widget`, представляющий компонент поля судоку. Нам придется работать с этим компонентом в других методах класса `MainWindow`;
- `settings` — объект класса `QSettings`, предназначенный для загрузки и сохранения настроек программы. Мы применим здесь подход, показанный в листинге 31.2, при котором сохранение настроек выполняется в методе `closeEvent()`;
- `printer` — объект класса `QPrinter`, представляющий принтер. Мы сразу же создадим его в конструкторе класса окна, чтобы потом не вносить в код конструктора слишком много правок.

Что касается методов класса `MainWindow`, то мы объявим всего три:

- `closeEvent()` — этот метод следует переопределить, если нужно выполнять какие-либо действия непосредственно перед закрытием окна. Внутри его мы выполним сохранение местоположения окна;
- `aboutInfo()` — выведет на экран стандартное диалоговое окно со сведениями о программе «Судоку»;
- конструктор сформирует интерфейс программы, загрузит и установит сохраненное ранее местоположение окна, создаст принтер, а также привяжет к окну таблицу стилей, которая укажет специфическое оформление для ячеек поля судоку и набора кнопок, который находится ниже поля.

Весь код класса `MainWindow` мы сохраним в модуле mainwindow.pyw в каталоге modules. Его мы также рассмотрим по частям.

32.3.4.1. Конструктор класса *MainWindow*

И в этом случае конструктор — самый сложный метод рассматриваемого класса. Поэтому мы не станем приводить полный листинг конструктора, а тоже рассмотрим его по фрагментам.

```
from PyQt6 import QtCore, QtGui, QtWidgets, QtPrintSupport
```

Не забываем импортировать все нужные модули, включая модуль `QtPrintSupport`.

```
from modules.widget import Widget
```

Импортируем класс поля судоку `Widget` из модуля widget.pyw, который мы сохранили в каталоге modules.

```
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent,
                  flags=QtCore.Qt.WindowType.Window |
                        QtCore.Qt.WindowType.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Судоку 2.0.0")
```

У создаваемого окна указываем флаг `MSWindowsFixedSizeDialogHint`, запрещающий изменение его размеров. Все равно поле судоку у нас имеет фиксированный размер.

```
        self.setStyleSheet(
             "QFrame QPushButton {font-size:10pt;font-family:Verdana;"
             "color:black;font-weight:bold;}"
             "MyLabel {font-size:14pt;font-family:Verdana;"
             "border:1px solid #9AA6A7;}")
```

Указываем у окна таблицу стилей, которая задаст представление для следующих элементов управления:

- у кнопок из набора, расположенного под полем судоку, — черный полужирный шрифт Verdana размером 10 пунктов (так мы сделаем эти кнопки заметнее);
- у ячеек поля судоку — шрифт Verdana размером 14 пунктов и темно-серую сплошную рамку толщиной в 1 пиксел.

```
        self.settings = QtCore.QSettings("Прохоренок и Дронов", "Судоку")
        self.printer = QtPrintSupport.QPrinter()
```

Создаем объекты хранилища настроек и принтера.

```
self.sudoku = Widget()
self.setCentralWidget(self.sudoku)
```

Создаем объект класса `Widget`, сохраняем его в атрибуте `sudoku` и помещаем в окно в качестве центрального.

```
menuBar = self.menuBar()
toolBar = QtWidgets.QToolBar()
```

Получаем уже имеющееся в окне главное меню и создаем панель инструментов.

```
myMenuFile = menuBar.addMenu("&Файл")
```

Создаем меню **Файл**.

```
action = myMenuFile.addAction(QtGui.QIcon(r"images/new.png"),
                    "&Новый", self.sudoku.onClearAllCells,
                    QtGui.QKeySequence("Ctrl+N"))
```

Создаем пункт **Новый** меню **Файл**.

Для создания пунктов меню используем разновидность метода `addAction()` класса `QMenu`, которая в качестве параметров принимает значок, название пункта, обработчик и комбинацию клавиш. На основе всего этого метод формирует действие (объект класса `QAction`), создает связанный с ним пункт меню и возвращает это действие в качестве результата (подробности — в *разд. 28.2.2*). Мы сохраним полученное действие в переменной, чтобы впоследствии создать на панели инструментов связанную с ним кнопку и задать текст подсказки для строки состояния.

В качестве обработчика для этого действия мы указываем метод `onClearAllCells()` компонента поля судоку (класса `Widget`).

```
toolBar.addAction(action)
action.setStatusTip("Создание новой, пустой головоломки")
```

Создаем на основе только что подготовленного действия кнопку **Новый** панели инструментов и задаем для действия текст подсказки, выводимой в строке состояния.

```
myMenuFile.addSeparator()
toolBar.addSeparator()

action = myMenuFile.addAction("&Выход",
                    QtWidgets.QApplication.instance().quit,
                    QtGui.QKeySequence("Ctrl+Q"))
action.setStatusTip("Завершение работы программы")
```

Добавляем в меню и на панель инструментов разделители и точно таким же образом создаем пункт **Выход** меню **Файл**. В качестве обработчика указываем сигнал `quit()` объекта программы.

```
myMenuEdit = menuBar.addMenu("&Правка")

action = myMenuEdit.addAction("&Блокировать",
                    self.sudoku.onBlockCell, QtCore.Qt.Key.Key_F2)
action.setStatusTip("Блокирование активной ячейки")
action = myMenuEdit.addAction(QtGui.QIcon(r"images/lock.png"),
                    "Б&локировать все",
                    self.sudoku.onBlockCells, QtCore.Qt.Key.Key_F3)
```

```
        toolBar.addAction(action)
        action.setStatusTip("Блокирование всех ячеек")

        action = myMenuEdit.addAction("&Разблокировать",
                                      self.sudoku.onClearBlockCell, QtCore.Qt.Key.Key_F4)
        action.setStatusTip("Разблокирование активной ячейки")

        action = myMenuEdit.addAction(QtGui.QIcon(r"images/unlock.png"),
                                      "Р&азблокировать все",
                                      self.sudoku.onClearBlockCells,
                                      QtCore.Qt.Key.Key_F5)
        toolBar.addAction(action)
        action.setStatusTip("Разблокирование всех ячеек")
```

Создаем меню **Правка**, его пункты **Блокировать**, **Блокировать все**, **Разблокировать** и **Разблокировать все** и соответствующие им кнопки панели инструментов. В качестве обработчиков действий указываем соответственно методы `onBlockCell()`, `onBlockCells()`, `onClearBlockCell()` и `onClearBlockCells()` компонента поля судоку.

```
        myMenuAbout = menuBar.addMenu("&Справка")

        action = myMenuAbout.addAction("О &программе...", self.aboutProgram)
        action.setStatusTip("Получение сведений о программе")

        action = myMenuAbout.addAction("О &Qt...", self.aboutQt)
        action.setStatusTip("Получение сведений о фреймворке Qt")
```

Создаем меню **Справка** и его пункты **О программе** и **О Qt**. У действия, связанного с первым пунктом, указываем в качестве обработчика метод `aboutProgram()` класса `MainWindow`, а у действия, связанного с вторым пунктом, — метод `aboutQt()` того же класса. Оба метода мы скоро напишем.

```
        toolBar.setMovable(False)
        toolBar.setFloatable(False)
        self.addToolBar(toolBar)
```

Запрещаем панели инструментов перемещаться внутри области, в которой она находится, и выноситься в отдельное окно (для нашей простой программы это излишне), и добавляем ее в окно. Поскольку в вызове метода `addToolBar()` окна мы не указали область, панель будет помещена в верхнюю часть окна.

```
        statusBar = self.statusBar()
        statusBar.setSizeGripEnabled(False)
        statusBar.showMessage("\"Судоку\" приветствует вас", 20000)
```

Получаем доступ к строке состояния, убираем из нее маркер изменения размера (все равно главное окно имеет фиксированные размеры) и выводим приветственное сообщение, которое будет отображаться в течение 20 секунд.

```
        if self.settings.contains("X") and self.settings.contains("Y"):
            self.move(self.settings.value("X"), self.settings.value("Y"))
```

Проверяем, находятся ли в хранилище настроек значения с именами `X` (горизонтальная координата левого верхнего угла окна) и `Y` (его вертикальная координата), и, если это так,

извлекаем эти значения и позиционируем окно по этим координатам (размеры окна хранить не имеет смысла, поскольку они неизменны).

32.3.4.2. Остальные методы класса *MainWindow*

Осталось рассмотреть три метода класса основного окна `MainWindow`.

Листинг 32.10. Метод `closeEvent()`

```
def closeEvent(self, evt):
    g = self.geometry()
    self.settings.setValue("X", g.left())
    self.settings.setValue("Y", g.top())
```

Метод `closeEvent()` будет автоматически вызван при закрытии окна (листинг 32.10). В нем мы выполняем сохранение текущих координат левого верхнего угла окна.

Листинг 32.11. Методы `aboutProgram()` и `aboutQt()`

```
def aboutInfo(self):
    QtWidgets.QMessageBox.about(self, "О программе",
              "<center>\"Судоку\" v2.0.0<br><br>"
              "Программа для просмотра и редактирования судоку<br><br>"
              "(c) Прохоренок Н.А., Дронов В.А. 2011-2022 гг.")

def aboutQt(self):
    QtWidgets.QMessageBox.aboutQt(self, title="О фреймворке Qt")
```

Метод `aboutProgram()` выведет стандартное окно со сведениями о программе, а метод `aboutQt()` — стандартное окно со сведениями о фреймворке Qt (листинг 32.11).

32.3.5. Главный модуль

Теперь напишем главный модуль, который, собственно, и запустит программу на выполнение. Сохраним его в файле **start.pyw** непосредственно в каталоге **sudoku**. Код главного модуля представлен в листинге 32.12.

Листинг 32.12. Главный модуль

```
from PyQt6 import QtGui, QtWidgets
import sys
from modules.mainwindow import MainWindow
app = QtWidgets.QApplication(sys.argv)
app.setWindowIcon(QtGui.QIcon(r"images/svd.png"))
window = MainWindow()
window.show()
sys.exit(app.exec())
```

Здесь мы создаем объект класса `QApplication`, представляющий программу, указываем у нее значок, подготовленный ранее, создаем объект только что написанного класса `MainWindow`, представляющего основное окно, выводим окно на экран и запускаем программу.

Выполним запуск программы, запустив модуль start.pyw любым знакомым нам способом: щелчком мышью на самом файле или нажатием клавиши <F5> в окне IDLE, в котором открыт этот модуль. Проверим, работает ли активизация ячеек по щелчку мыши и посредством клавиш-стрелок поставим в какие-либо ячейки цифры и удалим их. Попробуем заблокировать и разблокировать ячейки. Наконец, очистим поле судоку, вызовем окна сведений о программе и Qt и закроем программу.

32.3.6. Копирование и вставка головоломок

32.3.6.1. Форматы данных

Сначала следует определиться, в каких форматах головоломки будут копироваться в буфер обмена. В *разд. 32.2* мы решили, что таковых будет три: полный, компактный и предназначенный для Microsoft Excel.

В любом случае данные будут копироваться в виде строки, состоящей из цифр от 0 до 9.

- *Полный формат* — строка длиной 162 символа. Каждая пара цифр, содержащаяся в ней, представляет сведения об одной ячейке:
  - первая цифра — обозначает состояние блокировки ячейки: 0 — разблокирована, 1 — заблокирована;
  - вторая цифра — это, собственно, цифра, которая установлена в ячейке, или 0, если ячейка не имеет цифры.

  Сведения о ячейках записываются последовательно, без каких-либо разделителей: первая пара цифр хранит сведения о ячейке с номером 0, вторая — о ячейке 1, третья — о ячейке 2 и т. д.
- *Компактный формат* — строка длиной 81 символ. Она содержит только цифры, установленные в ячейках. 0 обозначает отсутствие цифры в соответствующей ячейке. Первая цифра соответствует ячейке 0, вторая — ячейке 1 и т. д.
- *Формат для Excel* — более длинная строка. Каждую ячейку представляет одна цифра — та, что установлена в ней (состояние блокировки не сохраняется). Если ячейка не имеет цифры, сохраняется пустая строка. Цифры или пустые строки, соответствующие всем ячейкам одной строки поля судоку, отделяются друг от друга символами табуляции. Наборы цифр или пустых строк, соответствующие отдельным строкам, отделяются друг от друга последовательностями символов возврата каретки и перевода строки. Этой же последовательностью символов завершается сама строка с данными.

Чтобы реализовать копирование и вставку головоломок, нам потребуется добавить новые методы в классы `Widget` и `MainWindow`.

32.3.6.2. Реализация копирования и вставки в классе *Widget*

В классе `Widget` мы объявим четыре новых метода:

- `getDataAllCells()` — возвращает данные о головоломке в полном формате;
- `getDataAllCellsMini()` — возвращает данные о головоломке в компактном формате;
- `getDataAllCellsExcel()` — возвращает данные о головоломке в формате для Excel.

  Эти методы не будут непосредственно помещать данные о головоломке в буфер обмена, а станут лишь формировать их. Занесением готовых данных в буфер обмена займутся методы класса `MainWindow`, которые мы напишем позже.

Методы `getDataAllCells()` и `getDataAllCellsMini()` мы впоследствии используем для подготовки данных, которые будут записываться в файлы;

- `setDataAllCells()` — принимает с единственным параметром данные о головоломке, представленные в полном или компактном формате (формат распознается автоматически), и выполняет их вставку.

  Опять же, этот метод не будет непосредственно извлекать данные из буфера обмена, а станет лишь выполнять вставку данных, извлеченных оттуда специальными методами класса `MainWindow`. Один из этих методов, помимо всего прочего, выполнит преобразование полученных из буфера обмена данных, представленных в формате Excel, в полный формат перед тем, как передать их методу `setDataAllCells()`.

  Позднее мы используем этот метод для вставки данных о головоломке, прочитанных из файла.

Листинг 32.13. Метод `getDataAllCells()`

```
def getDataAllCells(self):
    listAllData = []
    for cell in self.cells:
        listAllData.append("0" if cell.isCellChange else "1")
        s = cell.text()
        listAllData.append(s if len(s) == 1 else "0")
    return "".join(listAllData)
```

Метод `getDataAllCells()` возвращает строку с данными о головоломке в полном формате (листинг 32.13).

Формировать строку с копируемыми данными можно двумя способами. Первый способ заключается в том, что сначала объявляется переменная, хранящая пустую строку, а потом к этой строке постепенно добавляются символы, хранящие сведения о ячейках. Но в таком случае при очередном добавлении символов предыдущая строка останется в оперативной памяти — в результате эти «мусорные» строки станут постепенно накапливаться, засоряя память. Разумеется, рано или поздно они будут удалены особой подсистемой Python, носящей название *сборщика мусора*, но произойдет это только тогда, когда программа будет простаивать.

Поэтому, чтобы избежать «замусоривания» памяти, мы используем второй способ: объявим пустой список, в который будем добавлять строки с символами, представляющие сведения об очередной ячейке, а под конец сформируем строку, составленную из элементов этого списка (для этого мы вызовем у пустой строки метод `join()`, передав ему наш список). Это и будут данные о головоломке, записанные в полном формате.

В самом методе `getDataAllCells()` нет ничего особо сложного. Мы перебираем ячейки поля судоку в цикле. У каждой ячейки выясняем состояние блокировки (оно, как мы помним, хранится в атрибуте `isCellChange` класса `MyLabel`). Если ячейка разблокирована, добавляем в список строку `"0"`, в противном случае — `"1"`. Далее вызовом унаследованного метода `text()` извлекаем текстовое содержимое ячейки, проверяем, равна ли ее длина единице (т. е. установлена ли в ячейку цифра), и, если так, добавляем в список строку с этим содержимым (т. е. с установленной в ячейку цифрой). В противном случае добавляем строку `"0"`. Под конец формируем строку, составленную из элементов списка и представляющую собой данные, которые должны быть помещены в буфер обмена. Эту строку возвращаем в качестве результата.

Листинг 32.14. Метод getDataAllCellsMini()

```
def getDataAllCellsMini(self):
    listAllData = []
    for cell in self.cells:
        s = cell.text()
        listAllData.append(s if len(s) == 1 else "0")
    return "".join(listAllData)
```

Код метода `getDataAllCellsMini()`, возвращающий данные о головоломке в компактном формате (листинг 32.14), работает аналогично. Вы, уважаемые читатели, и сами разберетесь, как.

Листинг 32.15. Метод getDataAllCellsExcel()

```
def getDataAllCellsExcel(self):
    numbers = (9, 18, 27, 36, 45, 54, 63, 72)
    listAllData = [self.cells[0].text()]
    for i in range(1, 81):
        listAllData.append("\r\n" if i in numbers else "\t")
        listAllData.append(self.cells[i].text())
    listAllData.append("\r\n")
    return "".join(listAllData)
```

Метод `getDataAllCellsExcel()`, что станет формировать данные для вставки в Excel, немногим сложнее (листинг 32.15). Мы создаем кортеж, содержащий номера ячеек, перед которыми в результирующую строку вместо символа табуляции нужно вставить возврат каретки и перевод строки, — как видим, это номера первых ячеек в каждой строке, кроме первой. Затем создаем список из единственного элемента — содержимого самой первой ячейки: цифры или пустой строки. Далее в цикле перебираем все ячейки от второй до последней. Если номер очередной ячейки входит в объявленный ранее кортеж (т. е. начинается новая строка поля судоку), добавляем в список возврат каретки и перевод строки, в противном случае — символ табуляции. Далее добавляем в список содержимое ячейки. Наконец, завершаем формируемые данные возвратом каретки и переводом строки и формируем строку, составленную из элементов списка.

Листинг 32.16. Метод setDataAllCells()

```
def setDataAllCells(self, data):
    l = len(data)
    if l == 81:
        for i in range(0, 81):
            if data[i] == "0":
                self.cells[i].setText("")
                self.cells[i].clearCellBlock()
            else:
                self.cells[i].setText(data[i])
                self.cells[i].setCellBlock()
        self.onChangeCellFocus(0)
```

```
        elif l == 162:
            for i in range(0, 162, 2):
                if data[i] == "0":
                    self.cells[i // 2].clearCellBlock()
                else:
                    self.cells[i // 2].setCellBlock()
                self.cells[i // 2].setText("" if data[i + 1] == "0" \
                                           else data[i + 1])
            self.onChangeCellFocus(0)
```

Метод `setDataAllCells()`, вставляющий данные о головоломке в поле судоку (листинг 32.16), будет самым сложным. Ведь сначала он должен выяснять, в каком формате представлены данные — в полном или компактном.

Данные, предназначенные для вставки, метод получает с единственным параметром, и данные эти представлены в виде строки. Следовательно, узнать их формат мы можем, выяснив длину строки с данными:

- если длина строки с данными равна 81 символу, данные представлены в компактном формате. В этом случае мы перебираем все символы в полученной строке. Если очередной символ представляет собой цифру `"0"`, очищаем и разблокируем соответствующую ячейку, в противном случае заносим очередной символ (которым станет цифра) в ячейку и блокируем ее;
- если же длина строки равна 162 символам, данные представлены в полном формате. Мы точно так же перебираем в цикле символы этой строки, но уже через один, извлекая тем самым на каждом проходе каждый четный символ (для чего применим цикл `for i in range(0, 162, 2)`), — этим символом станет обозначение состояния блокировки соответствующей ячейки. Номер соответствующей символу ячейки можем получить, разделив номер очередного символа на `2` нацело (применив оператор `//`). Если извлеченный символ является цифрой `"0"`, разблокируем ячейку, в противном случае — блокируем. Далее извлекаем и проверяем следующий символ строки: если это цифра `"0"`, очищаем ячейку, а если цифра, отличная от `"0"`, заносим ее в ячейку.

В любом случае после вставки данных мы делаем активной самую первую (левую верхнюю) ячейку поля судоку.

32.3.6.3. Реализация копирования и вставки в классе *MainWindow*

В классе `MainWindow` мы внесем дополнения в код конструктора и объявим шесть новых методов:

- `onCopyData()` — копирует в буфер обмен данные в полном формате;
- `onCopyDataMini()` — копирует в буфер обмена данные в компактном формате;
- `onCopyDataExcel()` — копирует в буфер обмена данные в формате для Excel;
- `onPasteData()` — вставляет из буфера обмена данные, представленные в полном или компактном формате, предварительно проверив эти данные на корректность;
- `onPasteDataExcel()` — вставляет из буфера обмена данные, представленные в формате для Excel, предварительно проверив эти данные на корректность;
- `dataErrorMsg()` — выводит сообщение о том, что предназначенные для вставки данные имеют неправильный формат.

Начнем с того, что в самое начало модуля, где находятся выражения импорта, добавим выражение, импортирующее модуль для поддержки регулярных выражений:

```
import re
```

Регулярные выражения очень помогут нам реализовать проверку вставляемых данных на корректность.

Доработаем конструктор. Все необходимые добавления показаны в листинге 32.17 (добавленный код выделен полужирным шрифтом).

Листинг 32.17. Конструктор (дополнения)

```
    def __init__(self, parent=None):
        . . .
        myMenuEdit = menuBar.addMenu("&Правка")

        action = myMenuEdit.addAction(QtGui.QIcon(r"images/copy.png"),
                                      "К&опировать", self.onCopyData,
                                      QtGui.QKeySequence("Ctrl+C"))
        toolBar.addAction(action)
        action.setStatusTip("Копирование головоломки в буфер обмена")

        action = myMenuEdit.addAction("&Копировать компактно",
                                      self.onCopyDataMini)
        action.setStatusTip("Копирование в компактном формате")

        action = myMenuEdit.addAction("Копировать &для Excel",
                                      self.onCopyDataExcel)
        action.setStatusTip("Копирование в формате MS Excel")

        action = myMenuEdit.addAction(QtGui.QIcon(r"images/paste.png"),
                                      "&Вставить", self.onPasteData,
                                      QtGui.QKeySequence("Ctrl+V"))
        toolBar.addAction(action)
        action.setStatusTip("Вставка головоломки из буфера обмена")

        action = myMenuEdit.addAction("Вставить &из Excel",
                                      self.onPasteDataExcel)
        action.setStatusTip("Вставка головоломки из MS Excel")

        myMenuEdit.addSeparator()
        toolBar.addSeparator()

        action = myMenuEdit.addAction("&Блокировать",
                                      self.sudoku.onBlockCell, QtCore.Qt.Key_F2)
        . . .
```

Здесь мы добавляем в начало меню **Правка** пять пунктов: **Копировать**, **Копировать компактно**, **Копировать для Excel**, **Вставить** и **Вставить из Excel**. Указываем для них в качестве обработчиков приведенные ранее методы, а также добавляем нужные кнопки в панель инструментов.

Листинг 32.18. Методы onCopyData(), onCopyDataMini() и onCopyDataExcel()

```
def onCopyData(self):
    QtWidgets.QApplication.clipboard().setText(
        self.sudoku.getDataAllCells())

def onCopyDataMini(self):
    QtWidgets.QApplication.clipboard().setText(
        self.sudoku.getDataAllCellsMini())

def onCopyDataExcel(self):
    QtWidgets.QApplication.clipboard().setText(
        self.sudoku.getDataAllCellsExcel())
```

Методы `onCopyData()`, `onCopyDataMini()` и `onCopyDataExcel()`, копирующие данные, очень просты (листинг 32.18). Они всего лишь помещают в буфер обмена результат, возвращенный соответственно методами `getDataAllCells()`, `getDataAllCellsMini()` и `getDataAllCellsExcel()` класса поля судоку.

Листинг 32.19. Метод onPasteData()

```
def onPasteData(self):
    data = QtWidgets.QApplication.clipboard().text()
    if data:
        if len(data) == 81 or len(data) == 162:
            r = re.compile(r"[^0-9]")
            if not r.match(data):
                self.sudoku.setDataAllCells(data)
                return
    self.dataErrorMsg()
```

Метод `onPasteData()` вставляет данные в полном или компактном формате (листинг 32.19). Перед тем как вызвать метод `setDataAllCells()` класса поля судоку, передав ему данные для вставки, он выполняет их проверку. Сначала он удостоверяется, что данные для вставки вообще есть (не равны пустой строке), потом — что их длина равна 81 или 162 символам. Далее он выполняет последнюю проверку — выясняет, не присутствует ли в строке символ, отличный от цифр 0...9. Для этого он создает регулярное выражение, совпадающее с любым из таких символов (`[^0-9]`), и выполняет поиск в строке с данными посредством быстро выполняющегося метода `match()` (более подробно о работе с регулярными выражениями рассказывалось в *главе 7*).

И только если все проверки выполнены, вызывается метод `setDataAllCells()`, и ему передается строка с данными для вставки. После чего сразу же выполняется выход из метода.

Если же какая-либо проверка завершилась неудачей, будет выполнено самое последнее выражение — вызов метода `dataErrorMsg()`, выводящего сообщение об ошибке. Мы напишем этот метод позже.

Листинг 32.20. Метод onPasteDataExcel()

```
def onPasteDataExcel(self):
    data = QtWidgets.QApplication.clipboard().text()
```

```
        if data:
            data = data.replace("\r", "")
            r = re.compile(r"([0-9]?[\t\n]){81}")
            if r.match(data):
                result = []
                if data[-1] == "\n":
                    data = data[:-1]
                dl = data.split("\n")
                for sl in dl:
                    dli = sl.split("\t")
                    for sli in dli:
                        if len(sli) == 0:
                            result.append("00")
                        else:
                            result.append("0" + sli[0])
                data = "".join(result)
                self.sudoku.setDataAllCells(data)
                return
        self.dataErrorMsg()
```

Метод `onPasteDataExcel()` вставит из буфера обмена данные, представленные в формате для Excel, разумеется, также выполнив необходимые проверки (листинг 32.20). Сначала он убедится, что данные для вставки есть, и для удобства их дальнейшей проверки и обработки удалит из них символы возврата каретки (для этого можно использовать метод `replace()` класса `str`, указав у этого метода первым параметром удаляемый символ, а вторым — пустую строку).

Полученная строка представляет собой набор из строго 81 комбинации двух символов: цифры от 0 до 9, которая может присутствовать в единственном числе или отсутствовать, и символа табуляции или перевода строки. Это правило прекрасно формализуется регулярным выражением `([0-9]?[\t\n]){81}`. Мы сравниваем с ним строку с данными и выполняем дальнейшие манипуляции, только если сравнение выполняется.

Сначала подготавливаем пустой список, в который будем помещать отдельные строки — фрагменты вставляемой головоломки. Удаляем из полученной строки с данными завершающий символ перевода строки, если он там есть. Разбиваем эту строку по символам перевода строки, воспользовавшись методом `split()` класса `str`, и получаем список строк, представляющих отдельные строки поля судоку. Перебираем этот список и каждую из имеющихся в нем строк тем же методом разбиваем по символам табуляции, получив список строк, каждая из которых представляет сведения об одной ячейке. Перебираем этот список. Если очередной его элемент-строка пуст (т. е. ячейка не имеет цифры), добавляем в список строку `"00"`, где первая цифра обозначает, что ячейка не заблокирована, а вторая — отсутствие цифры в ячейке. Если же элемент не пуст, значит, он представляет собой цифру, которую следует занести в ячейку, и мы добавляем в список строку вида `"0<Эта цифра>`. Наконец, объединяем все элементы списка в строку, передаем ее методу `setDataAllCells()` класса поля судоку и выполняем возврат из метода.

Последнее выражение, выполняющееся, если какая-либо проверка из описанных ранее завершилась неудачей, вызовет все тот же метод `dataErrorMsg()`, который выведет сообщение о неправильном формате данных.

Осталось написать этот метод. Его код приведен в листинге 32.21 — как видим, он очень прост.

Листинг 32.21. Метод dataErrorMsg()

```
def dataErrorMsg(self):
    QtWidgets.QMessageBox.information(self, "Судоку",
              "Данные имеют неправильный формат")
```

Запустим программу и проверим, как работает копирование и вставка данных в разных форматах. Проще всего сделать это, занеся цифры в некоторые ячейки поля судоку, выполнив копирование в каком-либо формате, очистив поле и произведя вставку. После этого поле судоку должно выглядеть так же, как перед копированием.

32.3.7. Сохранение и загрузка данных

Сохранять головоломки мы будем в тех же форматах, в каких они копировались в буфер обмена, — это позволит нам использовать написанные в *разд. 32.3.6.2* методы класса `Widget`, выполняющие копирование данных.

Чтобы дать нашей программе возможность сохранять и загружать данные, добавим в класс `MainWindow` следующие методы:

- `onOpenFile()` — загрузит сохраненную в файле головоломку;
- `onSave()` — сохранит головоломку в файл в полном формате;
- `onSaveMini()` — сохранит головоломку в файл в компактном формате;
- `saveSVDFile()` — этот метод станет вызываться обоими предыдущими методами для выполнения собственно сохранения данных в файл. Эти данные он будет получать с единственным параметром.

И внесем исправления в код конструктора — их можно увидеть в листинге 32.22 (добавленный код выделен полужирным шрифтом).

Листинг 32.22. Конструктор (дополнения)

```
def __init__(self, parent=None):
    . . .
    action = myMenuFile.addAction(QtGui.QIcon(r"images/new.png"),
                        "&Новый", self.sudoku.onClearAllCells,
                        QtCore.Qt.CTRL + QtCore.Qt.Key_N)
    toolBar.addAction(action)
    action.setStatusTip("Создание новой, пустой головоломки")

    action = myMenuFile.addAction(QtGui.QIcon(r"images/open.png"),
                        "&Открыть...", self.onOpenFile,
                        QtGui.QKeySequence("Ctrl+O"))
    toolBar.addAction(action)
    action.setStatusTip("Загрузка головоломки из файла")

    action = myMenuFile.addAction(QtGui.QIcon(r"images/save.png"),
                        "Со&хранить...", self.onSave,
                        QtGui.QKeySequence("Ctrl+S"))
```

```
        toolBar.addAction(action)
        action.setStatusTip("Сохранение головоломки в файле")

        action = myMenuFile.addAction("&Сохранить компактно...",
                                      self.onSaveMini)
        action.setStatusTip("Сохранение головоломки в компактном формате")

        myMenuFile.addSeparator()
        toolBar.addSeparator()
        . . .
```

Этот код добавит в меню **Файл**, между пунктом **Новый** и разделителем, пункты **Открыть**, **Сохранить** и **Сохранить компактно**. В качестве обработчиков указаны описанные ранее методы. Также выполняется добавление еще двух кнопок в панель инструментов.

Листинг 32.23. Метод onOpenFile()

```
    def onOpenFile(self):
        fileName = QtWidgets.QFileDialog.getOpenFileName(self,
                   "Выберите файл", QtCore.QDir.homePath(),
                   "Судоку (*.svd)")[0]
        if fileName:
            data = ""
            try:
                with open(fileName, newline="") as f:
                    data = f.read()
            except:
                QtWidgets.QMessageBox.information(self, "Судоку",
                          "Не удалось открыть файл")
                return
            if len(data) > 2:
                if data[-1] == "\n":
                    data = data[:-1]
                if len(data) == 81 or len(data) == 162:
                    r = re.compile(r"[^0-9]")
                    if not r.match(data):
                        self.sudoku.setDataAllCells(data)
                        return
            self.dataErrorMsg()
```

В методе `onOpenFile()`, загружающем данные из файла (листинг 32.23), мы выводим стандартное диалоговое окно открытия файла, указав в качестве начального каталог пользовательского профиля. Если пользователь выбрал файл и нажал кнопку **Открыть**, мы в блоке обработки исключения открываем этот файл для чтения и читаем его содержимое. Если файл прочитать не удалось и было сгенерировано исключение, выводим соответствующее сообщение и выполняем возврат из метода.

Если данные были прочитаны, мы проверяем, имеют ли они длину 81 или 162 символа и не включают ли в себя символы, отличные от цифр 0...9. Если это так, передаем загруженные данные все тому же методу `setDataAllCells()` класса поля судоку и выполняем возврат.

Если же все эти проверки не увенчаются успехом, выполняется последнее выражение, которое вызовет метод `dataErrorMsg()` класса `MainWindow`, написанный нами ранее.

Листинг 32.24. Методы onSave() и onSaveMini()

```
def onSave(self):
    self.saveSVDFile(self.sudoku.getDataAllCells())

def onSaveMini(self):
    self.saveSVDFile(self.sudoku.getDataAllCellsMini())
```

Методы `onSave()` и `onSaveMini()`, сохраняющие данные в файл (листинг 32.24), очень просты — они лишь вызывают метод `saveSVDFile()`, передав ему результат, возвращенный методами соответственно `getDataAllCells()` и `getDataAllCellsMini()` класса поля судоку.

Листинг 32.25. Метод saveSVDFile()

```
def saveSVDFile(self, data):
    fileName = QtWidgets.QFileDialog.getSaveFileName(self,
               "Выберите файл", QtCore.QDir.homePath(),
               "Судоку (*.svd)")[0]
    if fileName:
        try:
            with open(fileName, mode="w", newline="") as f:
                f.write(data)
            self.statusBar().showMessage("Файл сохранен", 10000)
        except:
            QtWidgets.QMessageBox.information(self, "Судоку",
                      "Не удалось сохранить файл")
```

Метод `saveSVDFile()`, непосредственно сохраняющий данные (листинг 32.25), также несложен. Сначала мы выводим стандартное диалоговое окно сохранения файла. Если пользователь задал имя файла для сохранения и нажал кнопку **Сохранить**, открываем файл на запись (если файл с заданным именем отсутствует, он будет создан), записываем в него данные, полученные с параметром, и выводим в строке состояния сообщение об успехе. Открытие файла и запись в него выполняем в блоке обработки исключений — в случае возникновения исключения на экран будет выведено сообщение об ошибке записи.

Запустим программу, поставим в некоторые ячейки цифры, сохраним головоломку в полном формате, очистим поле судоку и загрузим сохраненную головоломку. После чего попробуем сохранить и загрузить головоломку в компактном формате.

32.3.8. Печать и предварительный просмотр

Условимся, что головоломка будет печататься в том же виде, в каком представлена на экране. Ячейки будут иметь размеры 30 × 30 пикселов, темно-серую рамку, оранжевый или светло-серый цвет фона. Цифры в ячейках будут выводиться черным цветом, шрифтом Verdana размером 14 пунктов и выравниваться посередине.

32.3.8.1. Реализация печати в классе *Widget*

Все действия по формированию печатного представления головоломки мы будем выполнять в классе поля судоку `Widget`. Для этого определим в нем метод `print()`, который в качестве единственного параметра получит принтер, на котором должна быть выполнена печать, и который представляется объектом класса `QPrinter`.

Код метода `print()` не очень велик, но требует развернутых пояснений. Мы рассмотрим его по частям.

```
def print(self, printer):
    penText = QtGui.QPen(QtGui.QColor(MyLabel.colorBlack), 1)
    penBorder = QtGui.QPen(QtGui.QColor(QtCore.Qt.GlobalColor.darkGray), 1)
    brushOrange = QtGui.QBrush(QtGui.QColor(MyLabel.colorOrange))
    brushGrey = QtGui.QBrush(QtGui.QColor(MyLabel.colorGrey))
```

Сразу же создаем два пера: для вывода цифр (черное) и рамок ячеек (темно-серое) и две кисти: оранжевую и светло-серую.

```
    painter = QtGui.QPainter()
    painter.begin(printer)
```

Начинаем печать.

```
    painter.setFont(QtGui.QFont("Verdana", pointSize=14))
```

Указываем шрифт для вывода цифр в ячейках.

```
    i = 0
```

Объявляем переменную, в которой будет храниться номер печатаемой в настоящий момент ячейки.

```
    for j in range(0, 9):
```

Запускаем цикл, который станет перебирать числа из диапазона 0...8 включительно. Эти числа будут представлять номера печатаемых строк поля судоку.

```
        for k in range(0, 9):
```

Внутри этого цикла запускаем другой, аналогичный, который будет перебирать номера ячеек текущей строки.

```
            x = k * 30
            y = j * 30
```

Вычисляем координаты левого верхнего угла печатаемой в настоящий момент ячейки. Горизонтальную координату мы можем получить, взяв номер текущей ячейки текущей строки и умножив его на ширину ячейки (30 пикселов). Вертикальная координата вычисляется аналогично на основе номера текущей строки и ее высоты (также 30 пикселов).

```
            painter.setPen(penBorder)
```

Теперь нам нужно вывести сам квадратик, создающий ячейку. Задаем темно-серое перо для печати рамки этого квадратика.

```
            painter.setBrush(brushGrey if \
                    self.cells[i].bgColorDefault == MyLabel.colorGrey \
                    else brushOrange)
```

Если для фона ячейки задан светло-серый цвет, задаем светло-серое перо, в противном случае — оранжевое.

```
painter.drawRect(x, y, 30, 30)
```

Рисуем квадратик.

```
painter.setPen(penText)
```

Задаем черное перо, которым будет выведена цифра.

```
painter.drawText(x, y, 30, 30,
                 QtCore.Qt.AlignmentFlag.AlignCenter,
                 self.cells[i].text())
```

Выводим поверх квадратика цифру — содержимое ячейки.

```
i += 1
```

Увеличиваем значение номера текущей ячейки на единицу, чтобы на следующем проходе цикла напечатать следующую ячейку.

```
painter.end()
```

И завершаем печать.

32.3.8.2. Класс *PreviewDialog*: диалоговое окно предварительного просмотра

Класс `PreviewDialog` реализует функциональность диалогового окна предварительного просмотра головоломки перед печатью (рис. 32.3). Это окно позволит просматривать головоломку в масштабе 1:1, увеличивать, уменьшать масштаб и сбрасывать его к изначальному значению.

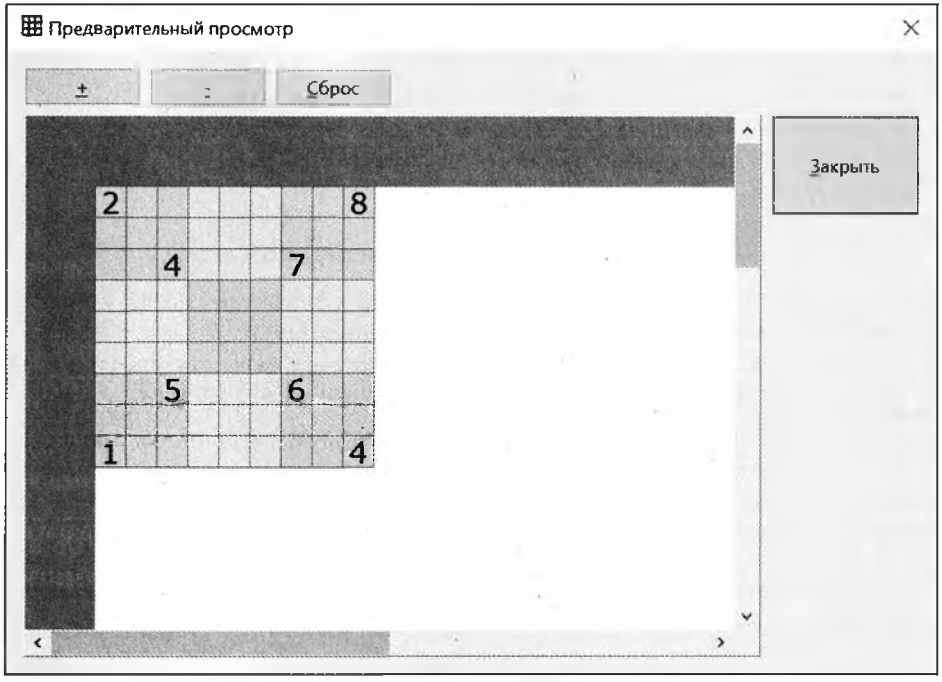


Рис. 32.3. Диалоговое окно предварительного просмотра

Код класса PreviewDialog мы сохраним в модуле previewdialog.pyw в каталоге modules. Он весьма велик и использует примечательные приемы программирования, о которых следует поговорить, поэтому рассмотрим его по частям.

```
from PyQt6 import QtCore, QtWidgets, QtPrintSupport

class PreviewDialog(QtWidgets.QDialog):
```

Окно предварительного просмотра мы делаем подклассом класса Dialog. Это позволит нам без особых проблем сделать размеры окна неизменяемыми, а само окно — модальным.

```
    def __init__(self, parent=None):
        QtWidgets.QDialog.__init__(self, parent)
        self.setWindowTitle("Предварительный просмотр")
        self.resize(600, 400)
        vBox = QtWidgets.QVBoxLayout()
```

Интерфейс окна включает две горизонтальные группы элементов управления, расположенные друг над другом (см. рис. 32.3). Поэтому для размещения групп мы создадим вертикальный контейнер QVBoxlayout.

```
        hBox1 = QtWidgets.QHBoxLayout()
```

Верхняя группа будет содержать три кнопки: для увеличения, уменьшения и сброса масштаба. Поскольку элементы в группе должны располагаться по горизонтали, используем для их расстановки контейнер QHBoxLayout.

```
        btnZoomIn = QtWidgets.QPushButton("&+")
        btnZoomIn.setFocusPolicy(QtCore.Qt.FocusPolicy.NoFocus)
        hBox1.addWidget(btnZoomIn, alignment=QtCore.Qt.AlignmentFlag.AlignLeft)
        btnZoomOut = QtWidgets.QPushButton("&-")
        btnZoomOut.setFocusPolicy(QtCore.Qt.FocusPolicy.NoFocus)
        hBox1.addWidget(btnZoomOut,
                        alignment=QtCore.Qt.AlignmentFlag.AlignLeft)
        btnZoomReset = QtWidgets.QPushButton("&Сброс")
        btnZoomReset.setFocusPolicy(QtCore.Qt.FocusPolicy.NoFocus)
        btnZoomReset.clicked.connect(self.zoomReset)
        hBox1.addWidget(btnZoomReset,
                        alignment=QtCore.Qt.AlignmentFlag.AlignLeft)
```

Создаем все эти три кнопки и добавляем их в контейнер. У каждой кнопки указываем, что она не может принимать фокус ввода, вызвав у нее метод setFocusPolicy() с параметром NoFocus, — таким образом, мы создадим в нашем диалоговом окне подобие панели инструментов. Также у всех трех кнопок указываем выравнивание по левому краю.

У кнопки сброса масштаба мы сразу же указываем в качестве обработчика сигнала clicked метод zoomReset() класса PreviewDialog. У остальных кнопок мы пока не указываем обработчики этого сигнала.

```
        hBox1.addStretch()
```

Добавляем в горизонтальный контейнер растягивающуюся область, чтобы все кнопки оказались прижатыми к левому краю контейнера.

```
        vBox.addLayout(hBox1)
```

Добавляем сам горизонтальный контейнер в вертикальный.

```
hBox2 = QtWidgets.QHBoxLayout()
```

Создаем еще один горизонтальный контейнер, в котором будут выводиться панель предварительного просмотра и кнопка **Закрыть**.

```
self.ppw = QtPrintSupport.QPrintPreviewWidget(parent.printer)
self.ppw.paintRequested.connect(parent.sudoku.print)
hBox2.addWidget(self.ppw)
```

Создаем панель предварительного просмотра (объект класса `QPrintPreviewWidget`) и связываем его сигнал `paintRequested` с методом `print()` компонента поля судоку, иначе эта панель ничего не выведет. Компонент поля судоку хранится в атрибуте `sudoku` основного окна программы, а основное окно мы получим с параметром `parent` конструктора. Напоследок добавляем панель просмотра во второй горизонтальный контейнер.

```
btnZoomIn.clicked.connect(self.ppw.zoomIn)
btnZoomOut.clicked.connect(self.ppw.zoomOut)
```

Создав панель предварительного просмотра, связываем с ее методами `zoomIn()` и `zoomOut()` сигналы `clicked` кнопок увеличения и уменьшения масштаба.

```
box = QtWidgets.QDialogButtonBox(
                QtWidgets.QDialogButtonBox.StandardButton.Close,
                QtCore.Qt.Orientation.Vertical)
```

Создаем контейнер для кнопок, которые обычно выводятся в диалоговом окне, добавляем в него кнопку закрытия и располагаем по вертикали.

```
btnClose = box.button(QtWidgets.QDialogButtonBox.StandardButton.Close)
btnClose.setText("&Закрыть")
btnClose.setFixedSize(96, 64)
btnClose.clicked.connect(self.accept)
```

Получаем только что созданную в контейнере кнопку, задаем у нее надпись **Закрыть**, увеличенные размеры и связываем ее сигнал `clicked` с методом `accept()`, унаследованным нашим диалоговым окном от класса `Dialog`.

```
hBox2.addWidget(box, alignment=QtCore.Qt.AlignmentFlag.AlignRight |
                                QtCore.Qt.AlignmentFlag.AlignTop)
```

Добавляем контейнер с кнопками во второй горизонтальный контейнер, указав выравнивание по правой и верхней границам, — т. е. расположение в верхнем правом углу.

```
vBox.addLayout(hBox2)
self.setLayout(vBox)
```

Добавляем второй горизонтальный контейнер в вертикальный контейнер и помещаем последний в окно.

```
self.zoomReset()
```

Указываем масштаб по умолчанию — 1:1, вызвав метод `zoomReset()` окна.

```
def zoomReset(self):
    self.ppw.setZoomFactor(1)
```

И, не откладывая дела в долгий ящик, определим этот метод.

32.3.8.3. Реализация печати в классе *MainWindow*

Теперь внесем необходимые дополнения в код класса `MainWindow`. Нам понадобится добавить в конструктор код, создающий необходимые пункты меню и кнопки панели инструментов, и три метода:

- `onPrint()` — печатает головоломку;
- `onPreview()` — выводит на экран созданное ранее окно предварительного просмотра;
- `onPageSetup()` — производит настройку параметров страницы.

Поскольку предварительный просмотр у нас будет выполняться в только что созданном окне `PreviewDialog`, нам следует добавить в самое начало кода класса `MainWindow` выражение, выполняющее импорт класса `PreviewDialog`:

```
from modules.previewdialog import PreviewDialog
```

Дополнения, которые следует внести в код конструктора класса `MainWindow`, показаны в листинге 32.26 (добавленный код, как обычно, выделен полужирным шрифтом).

Листинг 32.26. Конструктор (дополнения)

```
    def __init__(self, parent=None):
        . . .
        action = myMenuFile.addAction("&Сохранить компактно...",
                                      self.onSaveMini)
        action.setStatusTip("Сохранение головоломки в компактном формате")

        myMenuFile.addSeparator()
        toolBar.addSeparator()

        action = myMenuFile.addAction(QtGui.QIcon(r"images/print.png"),
                            "&Печать...", self.onPrint,
                            QtGui.QKeySequence("Ctrl+P"))
        toolBar.addAction(action)
        action.setStatusTip("Печать головоломки")

        action = myMenuFile.addAction(QtGui.QIcon(r"images/preview.png"),
                            "П&редварительный просмотр...", self.onPreview)
        toolBar.addAction(action)
        action.setStatusTip("Предварительный просмотр головоломки")

        action = myMenuFile.addAction("П&араметры страницы...",
                            self.onPageSetup)
        action.setStatusTip("Задание параметров страницы")

        myMenuFile.addSeparator()
        toolBar.addSeparator()

        action = myMenuFile.addAction("&Выход",
                            QtWidgets.QApplication.instance().quit,
                            QtGui.QKeySequence("Ctrl+P"))

        . . .
```

Между пунктами главного меню, вызывающими файловые операции, мы вставляем разделитель и пункты **Печать**, **Предварительный просмотр** и **Параметры страницы**. Не забываем добавить соответствующие кнопки на панель инструментов.

Листинг 32.27. Методы `onPrint()`, `onPreview()` и `onPageSetup()`

```
    def onPrint(self):
        pd = QtPrintSupport.QPrintDialog(self.printer, parent=self)
        pd.setOptions(
              QtPrintSupport.QPrintDialog.PrintDialogOption.PrintToFile |
              QtPrintSupport.QPrintDialog.PrintDialogOption.PrintSelection)
        if pd.exec() == QtWidgets.QDialog.DialogCode.Accepted:
            self.sudoku.print(self.printer)

    def onPreview(self):
        pd = PreviewDialog(self)
        pd.exec()

    def onPageSetup(self):
        pd = QtPrintSupport.QPageSetupDialog(self.printer, parent=self)
        pd.exec()
```

Код методов `onPrint()`, `onPreview()` и `onPageSetup()`, производящих печать, предварительный просмотр и настройку страницы, очень прост (листинг 32.27). Единственная деталь, достойная рассмотрения: в диалоговом окне печати мы задаем только возможность указания печати в файл и выбора принтера, на котором будет выполняться печать. Остальные параметры — в частности выбор печатаемых страниц — в нашем случае не нужны.

Запустим программу, создадим какую-либо головоломку и проверим, как работает печать, предварительный просмотр и настройка параметров страницы.

На этом работу над программой «Судоку» можно считать завершенной. Собственно, подошел конец и книге, посвященной замечательному языку Python и не менее замечательной библиотеке PyQt.

Заключение

Вот и закончилось наше путешествие в мир Python 3 и PyQt 6. Материал книги описывает лишь базовые возможности этих замечательных программных платформ. А здесь мы расскажем, где найти дополнительную информацию, чтобы продолжить изучение.

Самыми важными источниками информации является официальный сайт Python **https://www.python.org/** — на нем вы найдете дистрибутив, новости, а также ссылки на все другие ресурсы в Интернете. А на сайте **https://docs.python.org/3/** имеется документация по Python 3, которая обновляется в режиме реального времени.

Описание библиотеки PyQt можно найти на сайте **https://www.riverbankcomputing.com/**, но от него мало толку. Поэтому лучше сразу наведаться на сайт **https://doc.qt.io/**, где приведена полная документация по фреймворку Qt, лежащему в основе PyQt.

Описание всевозможных дополнительных библиотек, расширяющих возможности Python, созданных сторонними разработчиками и доступных для свободного скачивания, вы найдете на сайте **https://pypi.python.org/pypi**.

Следует отметить библиотеку Qt for Python (**https://www.qt.io/qt-for-python**), бывшую PySide, созданную специалистами компании Nokia, а ныне поддерживаемую независимыми разработчиками. Эта библиотека является полным аналогом PyQt и распространяется по лицензии LGPL. Существуют другие библиотеки для создания оконных программ: wxPython (**https://www.wxpython.org/**), PyGObject (**https://pygobject.readthedocs.io/en/latest/index.html**, бывшая PyGTK), PyWin32 (**https://mhammond.github.io/pywin32/**) и pyFLTK (**http://pyfltk.sourceforge.net/**). Обратите внимание и на библиотеку pygame (**http://www.pygame.org/**), позволяющую разрабатывать игры, и на фреймворк Django (**https://www.djangoproject.com/**), с помощью которого можно создавать веб-сайты.

Если в процессе изучения у вас возникнут какие-либо вопросы, вспомните, что в Интернете можно найти решения самых разнообразных проблем, — достаточно лишь набрать свой вопрос в строке запроса того или иного поискового портала (например, **https://www.bing.com/** или **https://www.google.ru/**).

Засим авторы прощаются с вами, уважаемые читатели, и желают успехов в нелегком, но таком увлекательном деле, как программирование!

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив с материалами, сопровождающими книгу, можно скачать с сервера издательства «БХВ» по ссылке **https://zip.bhv.ru/9785977517065.zip** или со страницы книги на сайте **https://bhv.ru**.

Структура архива представлена в табл. П.1.

Таблица П.1. *Структура электронного архива*

Файл или каталог	Описание
!main	Все листинги, представленные в книге
additional	Более 750 дополнительных листингов, демонстрирующих возможности библиотеки PyQt (к *части II* книги)
sudoku	Каталог с исходными текстами программы «Судоку»
readme.txt	Описание электронного архива

Предметный указатель

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Х

Ц

Ч

Э

Python 3 и PyQt 6

Разработка приложений

Быстрое создание программ с графическим интерфейсом

Если вы хотите научиться программировать на языке Python 3 и создавать программы с графическим интерфейсом, эта книга для вас. В первой части книги описан язык Python 3: типы данных, операторы, управляющие инструкции, регулярные выражения, функции, классы, часто используемые модули стандартной библиотеки. Вторая часть книги посвящена библиотеке PyQt 6, позволяющей создавать программы с графическим интерфейсом на языке Python 3. Описаны основные компоненты (кнопки, поля ввода, списки, таблицы, меню, панели инструментов и др.), средства для их размещения в окнах, обработка событий и сигналов, создание многопоточных приложений, работа с базами данных, вывод графики, воспроизведение и запись мультимедиа, вывод документов на печать и экспорт их в формате Adobe PDF и сохранения настроек программ.
Книга содержит большое количество практических примеров, помогающих начать программировать на языке Python самостоятельно. А в конце книги описывается процесс разработки приложения, предназначенного для создания и решения головоломок «судоку». Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания приложений с использованием C, C++, Go, Java, Python, HTML, JavaScript, PHP и MySQL. Автор более 20 книг, в том числе «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «JavaScript и Node.js для веб-разработчиков», «Bootstrap и CSS-препроцессор Sass. Самое необходимое», «Python 3. Самое необходимое», «Qt 6. Разработка оконных приложений на C++» и др.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор около 50 популярных книг по информационным технологиям, в том числе «Django 3.0. Практика создания веб-сайтов на Python», «Laravel 8. Быстрая разработка веб-сайтов на PHP», «HTML и CSS. 25 уроков для начинающих», «JavaScript. 20 уроков для начинающих», «PHP и MySQL. 25 уроков для начинающих», «JavaScript. Дополнительные уроки для начинающих» и «React 17. Разработка веб-приложений на JavaScript».

Примеры из книги можно скачать по ссылке
https://zip.bhv.ru/9785977517065.zip,
а также на странице книги на сайте https://bhv.ru.

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®