Коллекция лучших техник программирования

Рецепты Python

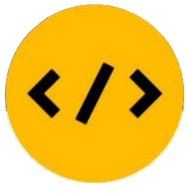
Юн Цуй

MANNING

Python How-To

63 Techniques to Improve Your Python Code

YONG CUI



MANNING
SHELTER ISLAND

Рецепты Python

Коллекция лучших техник программирования

ЮН ЦУЙ

ПИТЕР®

Санкт-Петербург • Москва • Минск

2024

ББК 32.973.2-018.1
УДК 004.43
Ю49

Юн Цуй

Ю49 Рецепты Python. Коллекция лучших техник программирования. — СПб.: Питер, 2024. — 512 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2156-4

В «Рецептах Python» используется простой, но эффективный метод освоения 63-х базовых навыков программирования на Python. Сначала формулируется вопрос, например «Как найти элементы в последовательности?» Затем приводится базовое решение на чистом понятном коде. Далее исследуются другие интересные подходы, такие как поиск подстрок или пользовательские классы. Перед переходом к следующему вопросу полученные навыки закрепляются с помощью решения задач.

Автор рассматривает все языковые средства, необходимые для уверенного владения Python. По ходу знакомства с книгой вы изучите лучшие приемы написания питонического кода. В освоении каждого инструмента помогут конкретные рекомендации и рисунки. Многочисленные перекрестные ссылки указывают на возможность повторного использования рассматриваемых средств и концепций в различных контекстах.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1617299742 англ.

ISBN 978-5-4461-2156-4

Краткое содержание

Часть 3
Определение классов

Часть 4
Операции с объектами и файлами

Часть 5
Защита кодовой базы

Часть 6
Построение веб-приложения

Оглавление

Часть 3
Определение классов

Посвящаю своей жене Тинтин Гу,
которая сидела рядом со мной поздними вечерами,
пока я писал эту книгу.

Предисловие

Пожалуй, мы — самое везучее поколение в истории человечества. Мы живем не в период неолита и не в индустриальный век, мы вошли в информационную эпоху. Современные информационные технологии, особенно компьютеры и сети, полностью изменили жизнь человека. Мы можем перелететь из своего родного города в другое место, находящееся за тысячи километров, менее чем за полдня. В случае необходимости мы можем записаться на прием к врачу по смартфону и побеседовать с ним по видеосвязи. У нас есть возможность заказать почти любой товар в интернет-магазине и получить его в считаные дни и даже часы.

Изменения в нашей жизни сопровождались накоплением громадных объемов данных, особенно в последние двадцать лет. Работа по обработке и анализу данных привела к появлению новой межотраслевой дисциплины — data science. Будучи ученым-бихевиористом, я проводил много времени за работой с данными: можно сказать, что я применял теорию data science к исследованиям в области поведенческой психологии. Однако для обработки таких объемов данных не обойтись ручкой и листом бумаги. Вместо этого я писал на замечательном языке программирования Python код для анализа данных и применения статистических моделей.

Как программист-самоучка я знаю, что изучить Python или любой другой язык программирования достаточно сложно, и не только из-за того, что освоение всех средств (и умение правильно выбирать их в той или иной ситуации) занимает массу времени. Сейчас доступно слишком много учебных ресурсов — онлайн-курсов, учебных видеороликов, статей в блогах и, разумеется, книг. Как выбрать те, которые лучше всего подходят для вас?

Я столкнулся с этим вопросом в начале своего изучения Python. За прошедшие годы я пробовал пользоваться разными ресурсами и обнаружил, что лучшими источниками информации являются книги, потому что они содержат четко структурированный материал, а это позволяет овладеть языком на более глубоком уровне. В процессе обучения вы сами выбираете наиболее подходящий для вас темп проработки материала — при необходимости можно замедлить чтение,

чтобы усвоить более сложные темы. Да и к стоящей на полке книге всегда легко обратиться при возникновении каких-либо вопросов.

Многие книги о Python, представленные на рынке, написаны либо для начинающих (с подробным описанием базовых возможностей языка), либо для опытных пользователей (с рассказом о специализированных методах, которые не столь универсальны). Бесспорно, некоторые из этих книг превосходны. Но приняв во внимание график роста мастерства с накоплением опыта, я счел, что среди этих книг не хватает предназначенной для тех, кто в своем освоении Python переходит с начального уровня на средний. Именно на этой решающей стадии у изучающих формируются правильные навыки работы с кодом и понимание того, какие средства Python лучше использовать в конкретном контексте. Что же касается материала, я решил, что в этой будущей книге должны рассматриваться общие проблемы программирования, которые большинство читателей смогут связать со своей работой независимо от того, применяют ли они Python для веб-программирования или в data science. Иначе говоря, такая книга пригодится многим читателям, потому что она предоставляет информацию, не привязанную к конкретной предметной области.

Я написал эту книгу, чтобы заполнить пробел между руководствами для начинающих и изданиями для опытных программистов. Надеюсь, вы узнаете из нее что-то новое для себя.

Благодарности

Хочу поблагодарить своих наставников, доктора Пола Чинчирипини (Dr. Paul Cinciripini) и доктора Джейсона Робинсона (Dr. Jason Robinson) из Андерсоновского онкологического центра Университета Техаса; они поддерживали меня, пока я учился использовать Python как язык для нашей аналитической работы, которая в конечном итоге и привела к появлению этой книги.

Также хочу поблагодарить команду издательства Manning: издателя Маржана Бейса (Marjan Bace) за руководство редакционной коллегией и технологической группой; соиздателя Майкла Стивенса (Michael Stephens), предложившего мне написать эту книгу; старшего ведущего редактора Марину Майклз (Marina Michaels) за координацию и редактирование; Рене ван ден Берг (René van den Berg) за научное редактирование; Уолтера Александера (Walter Alexander) и Игнасио Торреса (Ignacio Torres) за рецензирование кода; Александара Драгосавлевича (Aleksandar Dragosavljević) за организацию коллегиального рецензирования; весь персонал — за усердную работу по форматированию книги.

С благодарностью перечисляю рецензентов, предоставивших полезные отзывы: Алексей Знаменский (Alexei Znamensky), Алексей Выскубов (Alexey Vyskubov), Ариэль Андрес (Ariel Andres), Брент Бойлан (Brent Boylan), Крис Колосивски (Chris Kolosiwsky), Кристофер Карделл (Christopher Kardell), Кристофер Виллануэва (Christopher Villanueva), Клаудиу Шиллер (Claudiu Schiller), Клиффор Турбер (Clifford Thurber), Дирк Гомес (Dirk Gomez), Ганеш Сваминатан (Ganesh Swaminathan), Георгиос Думас (Georgios Doumas), Джеральд Мэк (Gerald Mack), Грегори Граймс (Gregory Grimes), Игорь Дудченко (Igor Dudchenko), Иябо Синдику (Iyabo Sindiku), Джеймс Мэтлок (James Matlock), Джеффри М. Смит (Jeffrey M. Smith), Джош Макадамс (Josh McAdams), Кирти Шетти (Keerthi Shetty), Ларри Кай (Larry Cai), Луис Алойа (Louis Aloia), Маркус Гезелл (Marcus Geselle), Мэри Энн Тигезен (Mary Anne Thygesen), Майк Баран (Mike Baran), Нинослав Черкез (Ninoslav Cerkez), Оливер Кортен (Oliver Korten), Пьерджиорджио Фаралья (Piergiorgio Faraglia), Радхакришна М. В. (Radhakrishna M. V.), Раджиндер Ядав (Rajinder Yadav), Рэймонд Чун (Raymond Cheung), Роберт Веннер (Robert Wenner), Шанкар Свами (Shankar Swamy), Шрирам Мачарла (Sriram Macharla), Гири С. Сваминатан (Giri S. Swaminathan), Стивен Эррера (Steven Herrera) и Витош К. Дойнов (Vitosh K. Doynov). Их предложения помогли улучшить эту книгу.

О книге

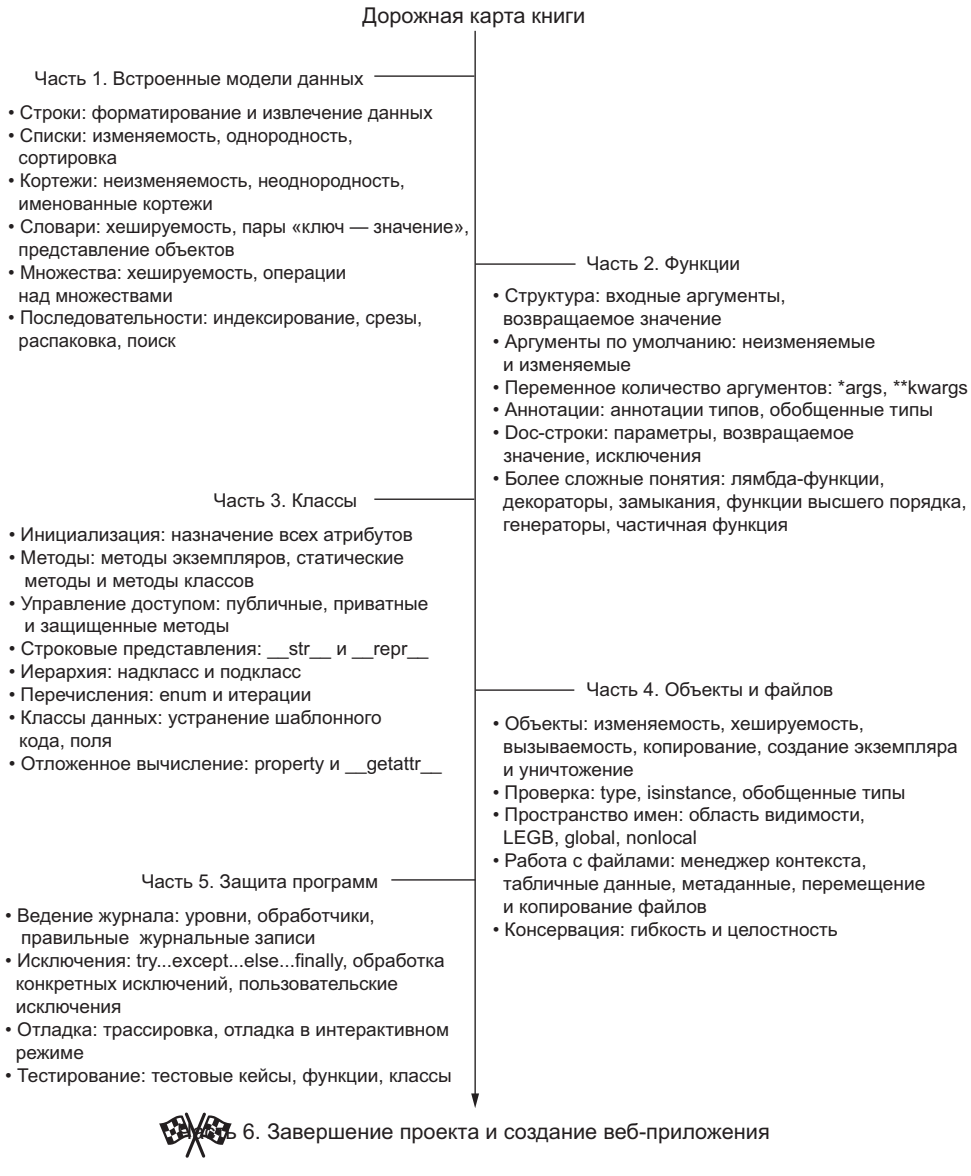
В этой книге я сосредоточился на основных приемах работы с Python, не привязывая их к какой-либо конкретной специализации. Хотя существует множество пакетов Python для разных специализированных областей (например, для data science или для веб-программирования), все эти пакеты строятся на основе базовых средств Python. Какие бы специализированные пакеты Python вы ни использовали в работе, также необходимо хорошо владеть основными навыками — скажем, правильно выбирать модели данных и писать структурированные функции и классы. С этими навыками вы сможете легко пользоваться любыми специализированными пакетами.

ДЛЯ КОГО ЭТА КНИГА

Если какое-то время вы изучали Python самостоятельно, но ваши знания языка представляются вам недостаточно упорядоченными, то вы, вероятно, находитесь в процессе перехода с начального уровня на промежуточный. Эта книга написана специально для вас, потому что вам нужно подкрепить свои знания Python и обобщить их в структурированной форме. В каждой главе этой книги я выделяю несколько тем для решения общих проблем, с которыми вы можете столкнуться в процессе работы. Впрочем, изложение этих тем не ограничивается решением конкретной проблемы — материал помещается в более широкий контекст, чтобы показать, почему и насколько эта тема важна при работе над любым проектом. Таким образом, вы не ограничиваетесь конкретными приемами решения отдельных задач, а работаете над своим проектом, параллельно осваивая применение этих приемов.

СТРУКТУРА КНИГИ

Книга состоит из шести частей, названия которых представлены на следующей схеме («Дорожной карте»). В части 1 (главы 2–5) изучаются встроенные модели данных, включая строки, списки и словари. Эти модели данных являются структурными элементами любого проекта. В части 2 (главы 6 и 7) рассказано



о том, как определяются функции. Функции считаются неотъемлемой частью любого проекта, потому что они обеспечивают обработку данных для получения нужного вывода. В части 3 (главы 8 и 9) вы научитесь правильно определять пользовательские классы. Вместо применения встроенных классов мы определяем пользовательские классы для более эффективного моделирования данных

в проекте. В части 4 (главы 10 и 11) представлены основы использования объектов и управления файлами. Часть 5 (главы 12 и 13) посвящена различным средствам повышения надежности программ, включая ведение журнала, обработку исключений и тестирование. В части 6 (глава 14) все полученные знания синтезируются для построения веб-приложения — проекта, который служит учебной основой для материала всех остальных глав.

Я рекомендую во время работы над книгой сразу воспроизводить все приведенные примеры на компьютере. Это позволит вам быстрее освоить синтаксис Python и основные приемы программирования. Я загрузил весь исходный код на GitHub, и мой общедоступный репозиторий доступен по адресу https://github.com/ycui1/python_how_to. Однако приводя в книге какой-либо код, я также привожу все необходимые пояснения и результаты, так что ничего страшного, если при чтении книги у вас под рукой не будет компьютера.

Если вы намерены воспроизводить примеры на компьютере, не имеет значения, какая на нем установлена операционная система. Windows, macOS и Linux — подойдет любая, потому что Python является кросс-платформенным языком программирования. Так как в книге я сосредоточусь на важнейших приемах и методах, которые устоялись в последних выпусках Python, не так важно, работаете ли вы на Python 3.8 или более ранней версии. Тем не менее, чтобы извлечь максимум пользы из книги, я рекомендую установить Python версии 3.10 и выше.

О КОДЕ В КНИГЕ

Книга содержит множество примеров исходного кода, размещенного как в нумерованных листингах, так и в тексте. В обоих случаях исходный код форматируется моноширинным шрифтом, в отличие от обычного текста. Иногда для кода также применяется **полужирный шрифт**, чтобы выделить фрагменты, изменившиеся по сравнению с предыдущими шагами, например, при добавлении новой функциональности в существующую строку кода.

Во многих случаях оригинальная версия исходного кода переформатирована; я добавил разрывы строк и изменил отступы, чтобы код помещался на странице. Иногда в листинги включаются маркеры продолжения строк (➥). Также из исходного кода удалены комментарии, если код описывается в тексте. Многие листинги сопровождаются аннотациями, поясняющими важные понятия.

Исполняемые фрагменты кода можно загрузить из электронной версии книги на liveBook: https://livebook.manning.com/book/python-how-to. Полный код примеров книги доступен для загрузки на сайтах издательства Manning (https://www.manning.com/books/python-how-to) и GitHub (https://github.com/ycuil/python_how_to).

ФОРУМ LIVEBOOK

Приобретая книгу «Рецепты Python», вы получаете бесплатный доступ к веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу https://livebook.manning.com/book/python-how-to/discussion. Информация о форумах Manning и правилах поведения на них размещена на https://livebook.manning.com/#!/discussion.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы он не терял интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

ДРУГИЕ ИСТОЧНИКИ ИНФОРМАЦИИ

Официальную документацию, включая учебники и справочники, можно найти по адресу https://docs.python.org/3. Автор книги, доктор Юн Цуй, регулярно ведет блоги о Python и сопутствующих темах data science на платформе Medium (https://medium.com/@yongcui01).

Об авторе

Доктор Юн Цуй — ученый, проработавший в области биомедицины более пятнадцати лет. Его исследовательская работа была посвящена разработке мобильных приложений медицинского назначения для поведенческой психотерапии на языках Swift и Kotlin. Его любимый язык Python стал основным средством для анализа данных, машинного обучения и разработки исследовательского инструментария. В свободное время он публикует в блогах посты по различным техническим темам, включая мобильную разработку, программирование на языке Python и искусственный интеллект.

Иллюстрация на обложке

Иллюстрация, помещенная на обложку второго издания книги и озаглавленная «Paysanne des environs de Soleure», или «Крестьянка из окрестностей Золотурна», взята из вышедшего в 1788 году каталога национальных костюмов, составленного Жаком Грассе де Сен-Совером (Jacques Grasset de Saint-Sauveur). Каждая иллюстрация этого каталога тщательно прорисована и раскрашена от руки.

В прежние времена по одежде человека можно было легко определить, где он живет и какова его профессия или положение в обществе. Издательство Manning приветствует изобретательность и инициативность — качества, присущие индустрии IT, — и в знак этого размещает на обложках изображения, которые демонстрируют богатое разнообразие региональных культур, запечатленное на старинных иллюстрациях.

От издательства

На протяжении всей книги автор рассматривает сквозной пример: приложение для управления задачами (таск-менеджер). Названия задач в этом примере (такие, как Laundry — стирка, Homework — домашнее задание, Pay bills — оплатить счета и т. д.) мы не переводили, так как они являются составной частью исходного кода. На понимание излагаемого материала это не влияет!

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Разработка стратегии прагматичного обучения

В этой главе

- ✓ Что такое «прагматичный»
- ✓ На что способен Python
- ✓ Когда стоит подумать об использовании альтернативных языков
- ✓ Чему может научить эта книга

Python — замечательный язык программирования. Распространение с открытым кодом, универсальность, независимость от платформы способствовали формированию огромного сообщества разработчиков и невероятной экосистемы, включающей десятки тысяч находящихся в свободном доступе библиотек для веб-разработки, машинного обучения (МО), data science и многих других областей. Надеюсь, вы согласитесь с тем, что умение писать код на Python важно, но умение создавать по-настоящему эффективные, безопасные и простые в сопровождении приложения дает вам поистине огромное преимущество. Эта книга поможет вам перейти с уровня начинающего программиста Python на уровень уверенного владения языком.

При практической работе в экосистеме Python мы применяем инструменты, предназначенные для конкретной предметной области, например веб-фреймворки или библиотеки для машинного обучения. Эффективное применение этих

инструментов — нетривиальное дело, для которого потребуется хорошее владение базовыми навыками программирования на Python: обработкой текста, работой со структурированными данными, созданием потоков управления и операций с файлами. Python-разработчики могут создавать разные решения для одних и тех же задач. Среди нескольких решений, как правило, можно найти наилучшее: наиболее компактное, или удобочитаемое, или эффективное. Все эти свойства часто объединяются под словосочетанием «питонический код»: имеется в виду идиоматический стиль программирования, овладеть которым стремятся все Python-разработчики. В этой книге рассказывается о том, как писать питонический код для решения различных задач программирования.

Python предоставляет разработчику столько возможностей, что было бы нереально или неразумно пытаться изучить их все по одной книге. Поэтому при выборе материала для своей книги я остановился на прагматичном подходе, а именно на обучении читателя тем основным навыкам, которые наиболее вероятно пригодятся в реальных проектах. Не менее важным я считаю регулярно обращаться к тому, как писать код, учитывая его удобочитаемость и простоту сопровождения. Это поможет вам выработать хороший стиль кодинга, который, я уверен, будет высоко оценен и вами, и вашими коллегам по команде.

> **ПРИМЕЧАНИЕ** В книге часто встречаются такие врезки. Многие из них содержат полезные советы относительно удобочитаемости и простоты сопровождения кода. Не пропускайте их!

1.1. О ПОЛЬЗЕ ПРАГМАТИЧНОГО ПОДХОДА

Мы программируем с определенной целью — для построения веб-сайта, обучения моделей МО или анализа данных. Какой бы ни была наша цель, нам нужно действовать прагматично, поскольку мы пишем код для решения реальных задач. Поэтому следует четко сформулировать свои цели, прежде чем изучать программирование с нуля или работать над повышением своей квалификации где-то в середине карьеры. Но даже если вы пока не уверены в том, чего именно хотите добиться при помощи Python, базовые средства Python универсально полезны. Освоив эти базовые средства, вы сможете применять их с любыми предметно-ориентированными инструментами Python.

Цель «стать прагматичным программистом» означает, что вам следует сосредоточиться на наиболее полезных приемах. Впрочем, освоение этих навыков — лишь первая веха на долгом пути, а вашей долгосрочной целью должно быть написание удобочитаемого кода, который не только работает, но и прост в сопровождении.

1.1.1. Написание удобочитаемого кода Python

Я разработчик, который фанатично стремится к удобочитаемости кода. Написание кода можно сравнить с разговором на обычном человеческом языке. Когда вы говорите на каком-нибудь языке, вы хотите, чтобы другие вас понимали? Если вы согласны, то, вероятно, согласитесь и с тем, что наш код должен быть понятным для других. Вопрос о том, обладают ли читатели нашего кода достаточной технической квалификацией для его понимания, находится вне зоны нашей ответственности. От нас зависит лишь то, как мы пишем код, — насколько удобочитаемым он получается. Попробуйте ответить на несколько простых вопросов:

- *Указывают ли имена ваших переменных на их назначение*? Никто не похвалит код, в котором полно переменных с именами вида `var0`, `temp_var` или `x`.
- *Указывают ли сигнатуры ваших функций на то, что делают эти функции*? Люди приходят в замешательство при виде функций с именами вида `do_data(data)` или `run_step1()`.
- *Насколько последовательно вы распределяете свой код по файлам*? Люди ожидают, что разные файлы одного типа используют сходную структуру. Например, размещаете ли вы инструкции `import` в начале файлов?
- *Хранятся ли конкретные файлы в правильных папках в структуре вашего проекта*? С увеличением масштаба проекта следует создавать отдельные папки для взаимосвязанных файлов.

Все эти вопросы относятся к области удобочитаемости. Не следует вспоминать о ней лишь время от времени, мы должны помнить про удобочитаемость кода постоянно, на протяжении всего проекта. Причина проста: к совершенству ведет правильная практика. Будучи нейробиологом по образованию, я точно знаю, как работает мозг, когда дело доходит до поведенческого обучения. Мы тренируем нейронную сеть нашего мозга, практикуясь в улучшении читабельности и постоянно задавая себе вопросы для самоконтроля. В конце концов вы обучите мозг распознавать хорошую практику программирования и начнете писать удобочитаемый, простой в сопровождении код, даже не задумываясь об этом.

1.1.2. Думайте о опровождаемости еще до написания кода

В отдельных случаях мы создаем код, предназначенный для одноразового использования. Когда мы пишем скрипт, нам почти всегда удается убедить себя в том, что он не предназначается для повторного применения, а значит, не нужно беспокоиться о создании понятных имен переменных, о правильном структурировании кода или рефакторинге функций и моделей данных

(не говоря уже том, что мы не пишем комментарии или оставляем устаревшие). Но сколько раз выяснялось, что тот же скрипт приходится использовать на следующей неделе и даже на следующий день? Вероятно, такое случалось с каждым из нас.

В предыдущем абзаце описана проблема простоты сопровождения (сопровождаемости) в малом масштабе. В данном случае она влияет только на вашу собственную производительность на коротком отрезке времени. Но если вы работаете в командной среде, сложности, создаваемые отдельными участниками, накапливаются и превращаются в крупномасштабные проблемы с сопровождаемостью. Участники команды не следуют одинаковым правилам выбора имен переменных, функций и файлов. В коде сплошь и рядом встречаются закомментированные фрагменты. Повсюду оставлены неактуальные комментарии.

Чтобы разрешить проблемы с сопровождаемостью на более поздних стадиях проектов, следует сформировать правильные установки еще во время обучения программированию. Ниже приводятся некоторые вопросы, которые стоит учитывать для выработки правильного отношения к простоте сопровождения в долгосрочной перспективе.

- *Свободен ли ваш код от устаревших комментариев и закомментированных участков*? Если ответ будет отрицательным, обновите или удалите их! Такие комментарии хуже их полного отсутствия, потому что они могут содержать противоречивую информацию.
- *Значительное ли место в коде занимает дублирование*? Если ответ будет положительным, вероятно, код требует рефакторинга. В программировании нередко упоминается практическое правило DRY (Don't Repeat Yourself, то есть «не повторяйтесь»). Удалив дубликаты из кода, вы получите один совместно используемый блок, что снизит риск ошибок по сравнению с внесением изменений во все повторяющиеся фрагменты.
- *Используете ли вы такие системы контроля версий, как Git*? Если ответ будет отрицательным, изучите расширения или плагины для вашей интегрированной среды разработки (IDE). Для Python часто используются IDE PyCharm и Visual Studio Code. Во многих IDE имеются интегрированные средства контроля версий, которые заметно упрощают управление версиями.

Каждый прагматичный программист Python должен взять на вооружение эти приемы, упрощающие сопровождение. Ведь почти все инструменты Python распространяются с открытым кодом и быстро развиваются. Следовательно, учет будущей сопровождаемости должен занимать центральное место в любом реальном проекте. В этой книге мы постараемся затронуть вопросы реализации практик сопровождения при повседневном программировании на Python. Помните, что удобочитаемость кода является ключевым фактором

долгосрочной простоты сопровождения. Если вы сосредоточитесь на написании удобочитаемого кода, это будет способствовать улучшению сопровождаемости вашей кодовой базы.

1.2. ЧТО PYTHON ДЕЛАЕТ ХОРОШО — ИЛИ НЕ ХУЖЕ, ЧЕМ ДРУГИЕ ЯЗЫКИ

Растущая популярность Python обусловлена характеристиками самого языка. Хотя ни одна из этих характеристик не уникальна для Python, именно их гармоничное сочетание объясняет повсеместное его распространение. Ниже приведена краткая сводка важнейших характеристик Python.

- *Кросс-платформенность* — Python работает на всех основных платформах, включая Windows, Linux и MacOS. Любой код Python, который вы напишете на своей платформе, будет работать на других компьютерах без ограничений, связанных с различиями между платформами.
- *Выразительность и удобочитаемость* — синтаксис Python проще синтаксиса многих других языков. Выразительный, понятный стиль программирования широко распространен среди питонистов. Вы увидите, что хорошо написанный код Python легко читается, как хорошо написанный текст.
- *Быстрота построения прототипов* — благодаря простоте синтаксиса код Python обычно получается более компактным, чем код, написанный на других языках. Следовательно, для построения работоспособного прототипа на Python требуется меньше работы, чем при использовании других языков.
- *Автономность* — после того, как вы установите Python на своем компьютере, он будет готов к использованию сразу же после «распаковки». Базовый установочный пакет Python содержит все основные библиотеки, необходимые для выполнения любых задач повседневного программирования.
- *Открытый код, бесплатное распространение и расширяемость* — хотя Python работает автономно, это не мешает вам создавать и использовать собственные пакеты. Если другой разработчик опубликовал какой-либо нужный вам пакет, вы сможете установить его однострочной командой, не беспокоясь о лицензии или оплате подписки.

Эти ключевые характеристики привлекли многих программистов, в результате чего сформировалось огромное сообщество разработчиков. Модель открытого кода Python позволяет заинтересованным пользователям вносить свой вклад в язык и экосистему в целом. В табл. 1.1 суммируются сведения о некоторых важных предметных областях и средствах, предоставляемых для них Python. Приведенный список далеко не полон, и вам предстоит самостоятельно исследовать, какие средства предлагает Python для деятельности в интересующей вас области.

Таблица 1.1. Обзор предметно-ориентированных инструментов Python

Предметная область	Инструмент	Краткая характеристика
Веб-разработка	Flask	Микрофреймворк для веб-разработки; хорошо подходит для построения облегченных веб-приложений; гибкий механизм расширения для сторонней функциональности
	Django	Полнофункциональный веб-фреймворк; хорошо подходит для построения веб-приложений, управляемых базами данных; обладает высокой масштабируемостью для корпоративных решений
	FastAPI	Веб-фреймворк для построения прикладных интерфейсов (API); проверка и преобразование данных; автоматическое генерирование веб-интерфейсов API
	Streamlit	Веб-фреймворк для простого построения приложений, ориентированных на данные; популярен среди специалистов data science и МО-инженеров
Data science	NumPy	Специализируется на обработке больших многомерных массивов; высокая вычислительная эффективность; интегрирован во многие библиотеки
	pandas	Гибкий пакет для обработки двумерных данных, сходных с электронными таблицами; широкий набор средств для работы с данными
	statsmodels	Популярный пакет статистических вычислений: линейная регрессия, корреляция, байесовские модели, анализ выживаемости
	Matplotlib	Объектно-ориентированная парадигма для построения гистограмм, точечных диаграмм, круговых диаграмм и других стандартных диаграмм с широкими возможностями настройки
	Seaborn	Простая в использовании библиотека визуализации для создания привлекательной графики; высокоуровневый API на базе Matplotlib
Машинное обучение	Scikit-learn	Широкий диапазон средств предварительной обработки для построения МО-моделей; реализация распространенных алгоритмов МО
	TensorFlow	Фреймворк с высокоуровневым и низкоуровневым API; система визуализации Tensor board; хорошо подходит для построения сложных нейронных сетей
	Keras	Высокоуровневые API для построения нейронных сетей; простота использования; хорошо подходит для построения низкопроизводительных моделей
	PyTorch	Фреймворк для построения нейронных сетей; более интуитивный стиль программирования, чем у TensorFlow; хорошо подходит для построения сложных нейронных сетей
	FastAI	Высокоуровневые API для построения нейронных сетей на базе PyTorch; простота использования

Фреймворки, библиотеки, пакеты и модули

При обсуждении инструментария используются тесно связанные термины — *фреймворки*, *библиотеки*, *пакеты* и *модули*. В разных языках могут использоваться термины с несколько различающимися значениями. Стоит разобраться в том, какой смысл вкладывает в эти термины большинство программистов Python.

Термин «*фреймворк*» является самым широким. Фреймворки предоставляют полный набор функциональных средств, специально спроектированных для выполнения определенной работы на высоком уровне (например, веб-разработки).

Библиотеки являются структурными элементами фреймворков и состоят из пакетов. Предоставляемая библиотеками функциональность разрешает пользователям не разбираться в подробностях работы с используемыми пакетами.

Пакеты предоставляют конкретную функциональность. Точнее говоря, пакеты состоят из модулей, и каждый модуль представляет собой набор тесно связанных структур данных и функций в одном файле (например, файле .py).

1.3. ЧЕГО PYTHON НЕ ДЕЛАЕТ ИЛИ ЧТО ДЕЛАЕТ НЕДОСТАТОЧНО ХОРОШО

Все имеет свои ограничения, есть они и у возможностей Python. Существует много всего, чего Python делать не может или, по крайней мере, не может делать так же хорошо, как альтернативные ему средства. Хотя некоторые люди пытаются «протолкнуть» Python как якобы используемый для всех целей язык, признаем, что на данный момент возможности Python в двух важных областях ограниченны:

- *Мобильные приложения* — в мобильную эпоху у всех есть смартфоны, а приложения используются практически во всех аспектах жизни: для банковских операций, покупок в интернете, заботы о здоровье, общения и, конечно, для игр. К сожалению, на сегодня хороших фреймворков Python для разработки приложений для смартфонов пока нет, несмотря на существование Kivy и BeeWare. Если вы работаете в сфере мобильной разработки, в качестве альтернативы стоит рассмотреть более развитые средства, такие как Swift для приложений iOS или Kotlin для приложений Android. Прагматичный программист выбирает язык, который позволяет создавать продукты, нравящиеся пользователю.
- *Низкоуровневая разработка* — при разработке программных продуктов, взаимодействующих напрямую с оборудованием, Python оказывается не лучшим вариантом. Из-за интерпретируемой природы Python общая скорость выполнения недостаточно высока для разработки низкоуровневых продуктов (скажем, драйверов устройств), требующих моментальной реакции. Если вас интересует

низкоуровневая разработка, вам стоит обратить внимание на альтернативные языки, которые успешнее взаимодействуют с аппаратным уровнем. Например, C и C++ хорошо подходят для разработки драйверов устройств.

1.4. О ЧЕМ ВЫ УЗНАЕТЕ ИЗ КНИГИ

Мы немного поговорили о том, что значит быть прагматичным программистом. Теперь обсудим, как прийти к этой цели. В ходе написания программ вы неизбежно столкнетесь с новыми проблемами из области программирования. В книге определены методы программирования, нужные для решения задач, с которыми вы с большой вероятностью встретитесь на практике.

1.4.1. Ориентация на предметно-независимые знания

Все вещи прямо или косвенно связаны друг с другом, это относится и к знанию Python. В контексте Python эта связь проиллюстрирована на рис. 1.1. На концептуальном уровне функциональность Python и его практическое применение можно представить как три взаимосвязанные сущности.

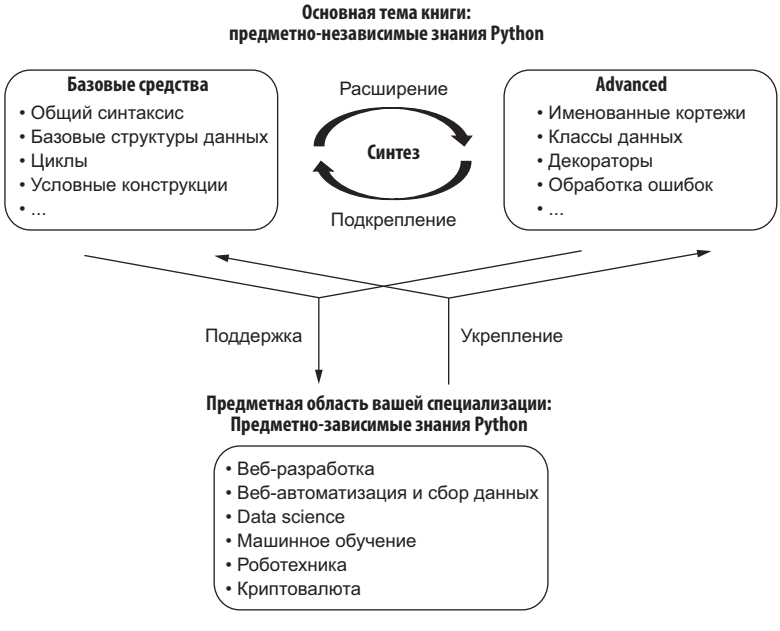


Рис. 1.1. Отношения между предметно-независимыми и предметно-зависимыми областями знаний Python. К предметно-независимым знаниям относятся базовые и расширенные средства Python, тесно связанные друг с другом. Совместно они формируют основу для предметно-зависимых знаний в разных предметных областях

Большинство людей изучают Python, чтобы применять язык для решения задач предметной области, в которой они работают. Для этого необходимы *предметно-зависимые* знания Python, например навыки веб-разработки и анализа данных. Необходимым условием успешного выполнения работы являются базовые навыки Python, а именно *предметно-независимые* знания. Даже если ваши должностные обязанности меняются или развиваются со временем, вы всегда можете применить базовые навыки Python в новой роли.

В этой книге мы сосредоточимся на предметно-независимых знаниях Python. Чтобы упростить процесс обучения, разделим предметно-независимые знания Python условно на два компонента: базовые и расширенные.

Что касается первого компонента, необходимо знать базовые структуры данных и операции с ними. Также нужно уметь вычислять результаты условий для построения инструкций `if...else`... Для выполнения многократно повторяющихся операций можно воспользоваться циклами `for` и `while`, а для повторного использования блоков кода переработать их в функции и классы. Овладения этими основами будет достаточно для написания полезного кода Python, который решает ваши рабочие задачи. Когда вы освоите большую часть базовых навыков, можно переходить к расширенным.

Расширенные навыки позволяют писать еще лучший код, более эффективный и использующий универсальные возможности Python. Чтобы оценить его универсальность, рассмотрим простой пример. При переборе объекта `list` в цикле `for` часто требуется вывести не только сам элемент, но и его позицию:

```
prime_numbers = [2, 3, 5]

# Желательный вывод:
Prime Number #1: 2
Prime Number #2: 3
Prime Number #3: 5
```

Ограничиваясь только базовыми возможностями, мы придем к приведенному ниже решению. В этом решении создается объект `range`, который позволяет извлечь индекс (с начинающейся с 0 нумерацией) для получения позиционной информации. Для вывода применяется конкатенация строк:

```
for num_i in range(len(prime_numbers)):
    num_pos = num_i + 1
    num = prime_numbers[num_i]
    print("Prime Number #" + str(num_pos) + ": " + str(num))
```

Но после прочтения этой книги вы станете более опытным пользователем Python и получите следующее решение, более элегантное и питоническое:

```
for num_pos, num in enumerate(prime_numbers, start=1):
    print(f"Prime Number #{num_pos}: {num}")
```

В этом решении используются три приема: распаковка кортежа для получения `num_pos` и `num` (раздел 4.4), создание объекта `enumerate` (раздел 5.3) и форматирование вывода с использованием f-строк (раздел 2.1). Я не стану подробно описывать эти приемы здесь, потому что они будут рассмотрены в соответствующих разделах. Однако этот пример показывает, чему посвящена данная книга, — *применению различных навыков для создания питонических решений*.

Кроме этих приемов, вы научитесь применять расширенные концепции функций (например, декораторы и замыкания). При определении классов вам будет ясно, как организовать их совместную работу для минимизации объема кода и сокращения риска ошибок. А когда программа будет завершена, вы уже будете знать, как протестировать ваш код, чтобы он был готов к эксплуатации.

Вся эта книга — о синтезе предметно-независимых знаний Python. Вы освоите не только практически полезные расширенные возможности, но и базовые средства Python и фундаментальные концепции программирования (в случаях, где они применимы). Ключевую роль в этом играет термин «*синтез*», что и показано в разделе 1.4.2.

1.4.2. Решение задач посредством синтеза

Начинающие программисты нередко оказываются в сложной ситуации: вроде бы они знают множество разнообразных методов, но не представляют, когда и как использовать их для решения задач. Для каждого рассматриваемого в книге приема я покажу, как он работает автономно, а также продемонстрирую его применение совместно с другими методами. Надеюсь, вскоре вы начнете понимать, как из этих разнородных компонентов строится бесконечное множество новых программ.

Необходимо сделать принципиальное замечание по поводу изучения и синтеза различных приемов: будьте готовы к тому, что путь постижения программирования нелинеен. Ведь и технические возможности Python тесно связаны друг с другом: хотя мы и сосредоточимся на изучении промежуточных и расширенных средств Python, их невозможно полностью изолировать от базовых тем. Вы заметите, что я часто отмечаю базовые средства или намеренно возвращаюсь к уже рассмотренным темам.

1.4.3. Изучение навыков в контексте

Как упоминалось ранее, в книге основное внимание уделяется изучению навыков, основанных на предметно-независимых знаниях Python. Определение «предметно-независимый» означает, что навыки, рассмотренные в книге, применимы к любой области, в которой вы хотите использовать Python. Впрочем, без примеров вряд ли можно что-либо изучить. Многие средства, описанные в книге, будут продемонстрированы на примере текущего проекта, который послужит единым контекстом для обсуждения конкретных навыков. Если какие-то навыки

вами уже освоены, можете переходить к подразделу «Обсуждение», в котором разбираются некоторые ключевые аспекты рассмотренной темы.

Забегая вперед, сообщу, что общим проектом станет веб-приложение для управления задачами (таск-менеджер). В этом приложении пользователь должен выполнять различные операции, включая добавление, редактирование и удаление задач. В нем задействовано все, что может быть реализовано на «чистом» Python: модели данных, функции, классы и вообще все, что обычно встречается в подобных приложениях. Сразу скажу, что целью вовсе не является построение идеально работающего совершенного приложения. В процессе работы над веб-приложением вы будете изучать важнейшие средства Python, чтобы потом применить предметно-независимые знания к вашим рабочим проектам.

ИТОГИ

- Очень важно сформировать стратегию прагматичного обучения. Сосредоточившись на предметно-независимых аспектах Python, вы сможете подготовиться к любой профессиональной роли, связанной с программированием на Python.
- Python — язык программирования общего назначения, распространяемый с открытым кодом. Вокруг него сформировалось огромное сообщество разработчиков, которые создают и распространяют пакеты Python.
- Python конкурентоспособен во многих областях, включая веб-разработку, data science и машинное обучение. В каждой области существуют свои фреймворки и пакеты Python, которыми вы сможете пользоваться.
- Возможности Python не безграничны. Если вы планируете разрабатывать мобильные приложения или низкоуровневые драйверы устройств, используйте Swift, Kotlin, Java, C, C++, Rust или любой другой подходящий для этого язык.
- В книге проводится различие между *предметно-независимыми* и *предметно-зависимыми* знаниями Python. При этом на первое место ставится изучение предметно-независимых знаний Python.
- Путь изучения программирования не линеен. Хотя в книге будут рассматриваться расширенные средства, я также буду часто упоминать о базовых. Кроме того, вы столкнетесь с некоторыми сложными темами, изучение которых требует движения по восходящей спирали.
- Важнейшим рецептом для изучения Python (или любого другого языка программирования) является синтез отдельных технических навыков для формирования их разностороннего набора. В процессе синтеза вы будете изучать язык с прагматичной точки зрения, зная, какие методы подойдут для решаемой вами проблемы.

Часть 1

Использование встроенных моделей данных

Мы создаем приложения для решения неких повседневных задач. Интернет-магазины предназначены для того, чтобы люди покупали одежду и книги, не выходя из дому. Программы управления кадрами — чтобы компании могли управлять своими сотрудниками. Программы для работы с текстом дают возможность редактировать документы. С точки зрения разработки, какие бы задачи ни решало приложение, необходимо извлечь и обработать информацию об этих задачах. В программировании для моделирования разных видов информации в приложениях (например, описаний продуктов и сотрудников) должны использоваться соответствующие структуры данных. Они обеспечивают нас стандартизированными способами представления реальных сущностей в приложениях, позволяя применять конкретные правила, структуры и реализации для решения бизнес-задач.

В этой части мы сосредоточимся на использовании встроенных моделей данных, включая строки, списки, кортежи, словари и множества. Кроме того, вы освоите приемы, общие для разных типов структур, в том числе для данных, сходных с последовательностями, и итерируемых объектов.

2 Обработка и форматирование строк

В этой главе

- ✓ Использование f-строк для интерполяции выражений и применения форматирования
- ✓ Преобразование строк в другие типы данных
- ✓ Объединение и разбиение строк
- ✓ Применение регулярных выражений для расширенной обработки строк

Текстовая информация — самая важная форма данных практически в любом приложении. Текстовые данные, как и числовые, можно сохранять в текстовых файлах, а чтение из таких файлов требует обработки строк. На веб-сайте интернет-магазина, например, текст используется для вывода описания товаров. Машинное обучение сейчас в моде, и вы, возможно, уже слышали об одной из его специализаций — *обработке естественных языков*, направленной на извлечение информации из текстов. Из-за повсеместного использования строк обработка текстов становится неизбежным шагом подготовки данных в таких сценариях. В контексте нашего таск-менеджера атрибуты задачи нужно преобразовать в текстовые данные, чтобы они могли отображаться интерфейсной частью веб-приложения. При получении данных от интерфейсной части приложения необходимо преобразовать эти строки к правильному типу (например, к целому) для дальнейшей обработки.

Необходимость правильной обработки и форматирования строк возникает во множестве реальных ситуаций. В этой главе будут рассмотрены некоторые распространенные задачи, встречающиеся при обработке текста.

2.1. КАК ИСПОЛЬЗОВАТЬ F-СТРОКИ ДЛЯ ИНТЕРПОЛЯЦИИ И ФОРМАТИРОВАНИЯ

В языке Python предусмотрены разные способы форматирования текстовых строк. Один из популярных способов основан на использовании f-строк, позволяющих встраивать выражения в строковый литерал. Можно использовать и другие методы форматирования, но решения с f-строками лучше читаются, следовательно, f-строки становятся предпочтительным вариантом при подготовке строк для вывода.

> **ОБРАТИТЕ ВНИМАНИЕ** F-строки впервые появились в Python 3.6. Префиксом f-строки может быть как символ f, так и символ F (от слова formatted — отформатированный). Строковый литерал представляет собой последовательность символов, заключенную в одинарные или двойные кавычки.

При использовании строк для вывода часто приходится иметь дело с нестроковыми данными, например целыми числами и числами с плавающей точкой. Допустим, таск-менеджер требует создания строкового вывода с использованием существующих переменных:

```
# Существующие переменные
name = "Homework"
urgency = 5

# Желательный вывод:
Name: Homework; Urgency Level: 5
```

В этом разделе вы научитесь использовать f-строки для интерполяции нестроковых данных и представления строк в нужном формате. Вы увидите, что при форматировании строк с использованием существующих строк и других разновидностей переменных f-строки являются лучшим решением с точки зрения удобочитаемости.

2.1.1. Форматирование строк до появления f-строк

Класс `str` работает с текстовыми данными через свои экземпляры, которые мы будем называть *строковыми переменными*. Помимо строковых переменных, в текстовую информацию также часто включаются такие типы данных, как целые числа и числа с плавающей точкой. Теоретически мы могли бы преобразовать нестроковые данные в строки и соединить их для получения нужного текстового вывода, как показано в листинге 2.1.

Листинг 2.1. Создание строкового вывода с применением конкатенации строк

```
task = "Name: " + name + "; Urgency Level: " + str(urgency)

print(task)
# Вывод: Name: Homework; Urgency Level: 5
```

В коде создания переменной `task` скрываются две потенциальные проблемы. Во-первых, он выглядит громоздко и не особенно хорошо читается, так как мы имеем дело с несколькими разными строками, каждая из которых заключена в кавычки. Во-вторых, `urgency` необходимо преобразовать из `int` в `str`, прежде чем значение может быть объединено с другими строками, что дополнительно усложняет операцию конкатенации.

Старые методы форматирования строк

До появления f-строк были доступны два решения. Первое решение записывается в классической форме языка C со знаком %, а во втором используется метод `format`.

Следующий фрагмент демонстрирует эти два решения:

```
task1 = "Name: %s; Urgency Level: %d" % (name, urgency)

task2 = "Name: {}; Urgency Level: {}".format(name, urgency)
```

Знак % отделяет строковый литерал от объекта кортежа

Решение в стиле C использует символ % в строковом литерале для обозначения форматируемой переменной, за которой следует знак % и кортеж соответствующих переменных. Решение с методом `format` используется примерно так же. Вместо знаков % в литерале в нем применяются фигурные скобки как маркеры строковой интерполяции, а соответствующие переменные перечисляются в методе `format`.

Следует заметить, что оба способа все еще поддерживаются в Python, но они считаются устаревшими и вам практически не придется пользоваться ими. По этой причине я не стану подробно на них останавливаться. Важно знать, что все, что они позволяют сделать, можно сделать с помощью f-строк — более наглядного механизма строковой интерполяции и форматирования, — как будет показано в разделе 2.1.2.

ОСНОВНЫЕ ПОНЯТИЯ Методами обычно называются функции, определяемые в классах. В данном случае функция `format` определяется в классе `str`, поэтому этот метод вызывается для экземпляров `str`.

2.1.2. Использование f-строк для интерполяции переменных

Форматирование строк часто подразумевает объединение строковых литералов и переменных разных типов (например, целых чисел и строк). При интеграции переменных в f-строку можно интерполировать эти переменные, чтобы они автоматически преобразовывались в строки нужного вида. В этом разделе описаны разные варианты интерполяции распространенных типов данных с использованием f-строк. Для начала рассмотрим, как использовать f-строки при создании вывода, представленного в листинге 2.1.

```
task_f = f"Name: {name}; Urgency Level: {urgency}"

assert task == task_f == "Name: Homework; Urgency Level: 5"
```

В этом примере переменная `task_f` создается с применением f-строк. Главное, на что следует обратить внимание, — фигурные скобки, в которые заключаются интерполируемые переменные. Так как f-строки интегрируют механизм строковой интерполяции, они также называются *интерполируемыми строковыми литералами*.

> **ОСНОВНЫЕ ПОНЯТИЯ** Термин «*строковая интерполяция*» (string interpolation) не относится к специфике Python, этот механизм присутствует в большинстве основных современных языков (таких, как JavaScript, Swift и C#). В общем случае он предоставляет более компактный и удобочитаемый синтаксис для создания отформатированных строк, чем конкатенация строк и альтернативные способы их форматирования.

Инструкция assert

Ключевое слово Python `assert` используется для проверок (ассертов), которые вычисляют заданное условие. Если результат равен `True`, программа продолжает выполняться. Если же результат равен `False`, выполнение прерывается, а программа выдает ошибку `AssertionError`.

В этой книге я буду использовать инструкцию `assert` для демонстрации эквивалентности переменных, задействованных в сравнении. В особом случае, когда проверяемая переменная относится к логическому типу, с технической точки зрения лучше использовать `assert true_var` и `assert not false_var`. Но чтобы более наглядно отразить логическое значение переменной, я предпочел использовать `assert true_var == True` и `assert false_var == False`.

Вы уже видели, как f-строки интерполируют строки и целочисленные переменные. Как насчет других типов, например, списков `list` и кортежей `tuple`?

Эти типы также поддерживаются f-строками, как показывает следующий фрагмент:

```
tasks = ["homework", "laundry"]
assert f"Tasks: {tasks}" == "Tasks: ['homework', 'laundry']"

task_hwk = ("Homework", "Complete physics work")
assert f"Task: {task_hwk}" == "Task: ('Homework', 'Complete physics work')"

task = {"name": "Laundry", "urgency": 3}
assert f"Task: {task}" == "Task: {'name': 'Laundry', 'urgency': 3}"
```

Интерполирует объект списка

Интерполирует объект кортежа

Интерполирует объект словаря

ЗАБЕГАЯ ВПЕРЕД F-строки также поддерживают экземпляры пользовательских классов. Когда в главе 8 мы займемся созданием пользовательских классов, мы еще вернемся к тому, как строковая интерполяция работает с пользовательскими экземплярами (раздел 8.4).

2.1.3. Использование f-строк для интерполяции выражений

Мы рассмотрели, как в f-строках интерполируются переменные. На более общем уровне f-строки также могут интерполировать выражения, что избавляет вас от необходимости создания промежуточных переменных. Например, при создании строкового вывода можно обратиться к элементу объекта `dict` или использовать результат вызова функции. В таких распространенных ситуациях можно включить эти выражения в f-строки, как показывает следующий фрагмент кода:

```
tasks = ["homework", "laundry", "grocery shopping"]
assert f"First Task: {tasks[0]}" == 'First Task: homework'

task_name = "grocery shopping"
assert f"Task Name: {task_name.title()}" == 'Task Name: Grocery Shopping'

number = 5
assert f"Square: {number*number}" == 'Square: 25'
```

Обращается к элементу списка

Вызывает функцию

Прямые вычисления

Эти выражения заключаются в фигурные скобки, чтобы f-строки напрямую вычислили их для получения нужного строкового вывода: `{tasks[0]}` -> `"homework"`; `{task_name.title()}` -> `"Grocery Shopping"`; `{number*number}` -> `25`.

В тексте часто встречается термин «*выражение*» (expression), относящийся к числу ключевых понятий программирования. Начинающие программисты могут путать его со связанным понятием — *инструкцией* (statement). Выражение обычно представляет собой одну строку кода (хотя может занимать несколько строк при заключении в тройные кавычки), результатом вычисления которой является значение объекта, например строки или экземпляра нестандартного класса. Из этого определения легко выводится, что переменные являются разновидностями выражений.

С другой стороны, инструкции не создают никакого значения или объекта. Цель инструкции заключается в выполнении некоторого действия. Например, ключевое слово `assert` создает проверочную инструкцию (или команду), которая проверяет выполнение некоторого условия, прежде чем процесс продолжится. Мы не пытаемся получить логическое значение `True` или `False`, а проверяем условие. Рисунок 2.1 показывает, чем выражения отличаются от инструкций.

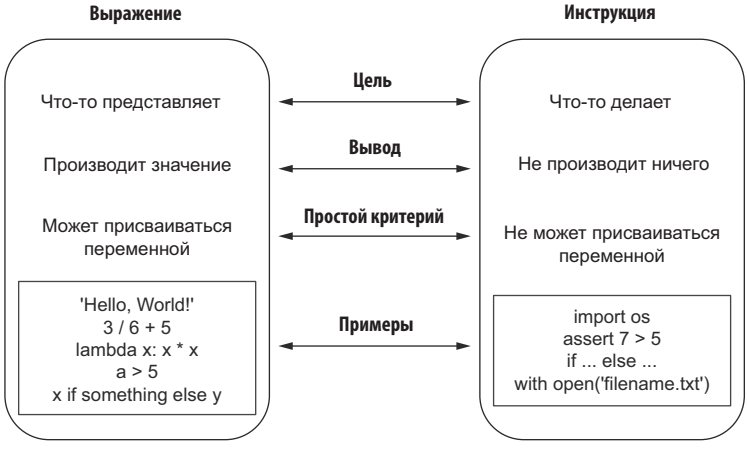


Рис. 2.1. Различия между выражениями и инструкциями. Выражения представляют некие вычисления, в результате которых получается значение или объект, тогда как инструкция выполняет конкретные действия и не может вычисляться для получения значения

Хотя f-строки поддерживают интерполяцию выражений, эту возможность следует использовать с осторожностью, потому что любые сложные выражения в f-строках ухудшают читаемость вашего кода. Следующий пример демонстрирует злоупотребление f-строками, использующими сложные выражения:

```
summary_text = f"Your Average Score: {sum([95, 98, 97, 96, 97, 93]) /
➥ len([95, 98, 97, 96, 97, 93])}."
```

Существует хороший практический критерий оценки удобочитаемости вашего кода — определите, сколько времени потребуется читателю, чтобы разобраться в нем. Чтобы понять, что происходит в приведенном выше фрагменте, читателю понадобятся десятки секунд. Сравните со следующей переработанной версией:

```
scores = [95, 98, 97, 96, 97, 93]

total_score = sum(scores)
subject_count = len(scores)
average_score = total_score / subject_count

summary_text = f"Your Average Score: {average_score}."
```

В этой версии заслуживает внимания ряд обстоятельств. Во-первых, оценки сохраняются в объекте `list`, что позволяет избавиться от дублирования данных. Во-вторых, вычисления разбиты на несколько шагов, при этом каждый шаг представляет собой более простое вычисление. В-третьих, ключевым фактором для улучшения удобочитаемости становится использование на каждом шаге содержательного имени, обозначающего результат вычислений. Такой код хорошо читается без единого комментария, все понятно само по себе.

УДОБОЧИТАЕМОСТЬ Создайте необходимые промежуточные переменные с содержательными именами, чтобы наглядно обозначить каждый шаг ваших операций. Для простых операций вам даже не придется писать комментарии, потому что содержательные имена описывают смысл каждой операции.

2.1.4. Применение спецификаторов для форматирования f-строк

Правильное форматирование текстовых данных (например, выравнивание) играет ключевую роль в передаче информации. Так как f-строки проектировались для форматирования строк, они предоставляют возможность задать *спецификатор формата* (начинающийся с двоеточия) для применения дополнительных правил форматирования к выражению в фигурных скобках (рис. 2.2). В этом разделе вы узнаете, как применять спецификаторы для форматирования f-строк.

При интерполяции можно воспользоваться спецификатором формата — необязательным компонентом, который определяет, как должна форматироваться интерполированная строка выражения. F-строка может получать разные виды спецификаторов формата. Рассмотрим самые полезные их разновидности, начиная с выравнивания текста.

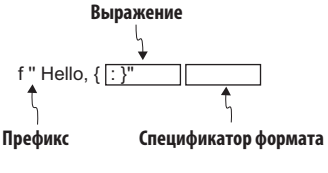


Рис. 2.2. Компоненты f-строки. Выражение — первая и обязательная часть. Сначала вычисляется выражение, затем создается соответствующая строка. Вторая часть — спецификатор формата — не является обязательной

Выравнивание строк для формирования визуальной структуры

Одним из способов повышения эффективности передачи информации является ее структурированная организация; это в полной мере относится к представлению текстовых данных. Как показано на рис. 2.3, сценарий B благодаря

применению более организованной структуры с выравниванием столбцов предоставляет информацию нагляднее, чем сценарий А.

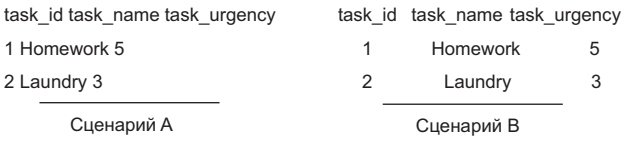


Рис. 2.3. Наглядность представления при организованной структуре текста (сценарий В) по сравнению с выравниванием по левому краю, используемым по умолчанию (сценарий А)

Для выравнивания текста в f-строках используются три символа: `<`, `>` и `^`, включающие выравнивание текста по левому краю, по правому краю и по центру соответственно. Если вы начнете путаться в том, какой символ что делает, посмотрите, в каком направлении указывает стрелка: например, если она направлена влево, то текст выравнивается по левому краю.

Для определения режима выравнивания текста в спецификаторе формата используется синтаксис `f"{expr:x<n}"`, где `expr` — интерполированное выражение, `x` — символ-заполнитель для выравнивания (если не указан, по умолчанию используется пробел), `<` — признак выравнивания по левому краю и `n` — целочисленный интервал, до которого расширяется вывод. Следующий листинг показывает, как создать две записи с выровненными полями для получения более наглядного вывода.

Листинг 2.2. Применение спецификаторов формата в f-строках

```
task_ids = [1, 2, 3]
task_names = ['Do homework', 'Laundry', 'Pay bills']
task_urgencies = [5, 3, 4]

for i in range(3):
    print(f'{task_ids[i]:^12}{task_names[i]:^12}{task_urgencies[i]:^12}')

# Выводятся следующие строки:
     1      Do homework      5
     2        Laundry        3
     3       Pay bills       4
```

Применяет спецификаторы формата к выражениям

В этом листинге стоит обратить внимание на то, что применение одного спецификатора формата ко всем выражениям приводит к повторению. Если в вашем коде встречаются повторения, скорее всего, вы нарушаете принцип DRY (Don't Repeat Yourself, то есть «не повторяйтесь»), а это указывает на необходимость рефакторинга.

Принцип DRY и рефакторинг

В программировании разработчики руководствуются рядом принципов. К их числу относится знаменитый принцип DRY. Если в вашей программе встречается повторяющийся код, скорее всего, следует провести рефакторинг для устранения таких повторений. Некоторые IDE, в частности PyCharm, включают средства автоматического выявления повторений. Пользуйтесь ими для улучшения своих программ.

Говоря о рефакторинге, я имею в виду меры по обновлению существующего кода для улучшения его структуры и, как следствие, повышения удобства сопровождения. Рефакторинг не добавляет новые возможности в вашу программу; вместо этого он изменяет структуру существующего кода без добавления каких-либо изменений в его поведение. Примеры рефакторинга будут приводиться в книге там, где его применение уместно.

Если появятся новые требования к выравниванию текста, в листинге 2.2 код придется обновлять в трех местах; это неудобно и повышает риск ошибок. В результате рефакторинга мы хотим получить механизм использования переменной в качестве спецификатора формата. В листинге 2.3 приведено возможное решение, в котором выделяется повторяющаяся часть — спецификатор формата. Если же пойти немного дальше по пути рефакторинга, можно определить функцию, которая будет получать спецификатор формата в параметре; это позволит экспериментировать с разными спецификаторами формата. Чтобы улучшить удобочитаемость кода, мы создадим отдельные переменные для данных задачи.

Листинг 2.3. Функция, получающая произвольный спецификатор формата

```
def create_formatted_records(fmt):
    for i in range(3):
        task_id = task_ids[i]
        name = task_names[i]
        urgency = task_urgencies[i]
        print(f'{task_id:{fmt}}{name:{fmt}}{urgency:{fmt}}')
```

В листинге 2.3 стоит обратить внимание на то, что спецификатор формата `fmt` заключен в фигурные скобки внутри внешних фигурных скобок. Python знает, как заменить `{fmt}` правильным спецификатором формата. Опробуем эту функцию с разными спецификаторами формата:

```
>>> create_formatted_records('^15')
       1         Do homework         5
       2           Laundry           3
       3          Pay bills          4
>>> create_formatted_records('^18')
        1            Do homework            5
        2              Laundry              3
        3             Pay bills             4
```

Как видите, полученный в результате рефакторинга код позволяет назначить произвольный спецификатор формата, и эта гибкость подчеркивает преимущества рефакторинга. Когда спецификаторы формата используются для выравнивания, текст образует четко различимые столбцы. Тем самым создаются визуальные границы для разделения разных информационных полей.

> **СОПРОВОЖДАЕМОСТЬ** Мы постоянно ищем возможности для рефакторинга своего кода, обычно на «локальном» уровне. Локальная оптимизация может показаться чем-то незначительным, но эти мелкие усовершенствования накапливаются и определяют общее удобство сопровождения проекта.

Для заполнения в приведенном примере используются пробелы, но можно использовать и другие символы. Выбор символов зависит от того, способствуют ли они наглядному выделению информации. В табл. 2.1 приведены примеры использования разных заполнителей и режимов выравнивания.

Таблица 2.1. Спецификаторы формата f-строк для выравнивания текста

F-строка	Вывод	Описание
`f"{task:*>10}"`[a]	`"**homework"`	Выравнивание по правому краю, заполнитель *
`f"{task:*<10}"`	`"homework**"`	Выравнивание по левому краю, заполнитель *
`f"{task:*^10}"`	`"*homework*"`	Выравнивание по центру, заполнитель *
`f"{task:^10}"`	`" homework "`	Выравнивание по центру, заполнитель — пробел

[a] Задача task определяется как строковая переменная: `task = "homework"`

Форматирование чисел

Числа — неотъемлемые источники информации, которые часто включаются в текстовый материал. Существуют различные формы числовых значений: большие целые числа, числа с плавающей точкой, проценты и т. д. В этом разделе вы узнаете, как f-строки представляют числовые значения со спецификаторами формата, упрощающими восприятие информации.

Множество простых чисел бесконечно. Обычный поиск в Google показывает, что наименьшее простое число, превышающее 1 миллиард, равно 1 000 000 007. При выводе такого большого числа желательно разделять группы цифр, и чаще всего после каждых трех цифр вставляется запятая. Для назначения разделителей групп разрядов в целых числах в f-строке используется спецификатор формата `xd`, где `x` — разделитель, а `d` — спецификатор формата для целых чисел:

```
large_prime_number = 1000000007

print(f"Use commas: {large_prime_number:,d}")
# Вывод: Use commas: 1,000,000,007
```

Числа с плавающей точкой, как и дробные числа вообще, встречаются почти в каждом научном или инженерном отчете. Как и следовало ожидать, у f-строк существуют спецификаторы формата, которые позволяют форматировать дробные числа в удобочитаемом виде. Рассмотрим следующие примеры:

```
decimal_number = 1.23456
print(f"Two digits: {decimal_number:.2f}")
# Вывод: Two digits: 1.23
print(f"Four digits: {decimal_number:.4f}")
# Вывод: Four digits: 1.2346
```

Если для целых чисел использовался спецификатор формата `d`, то для дробных значений используется спецификатор `f`. Хотя спецификатор `f` может использоваться автономно, чаще указывается, сколько цифр должно выводиться в дробной части: `.2` для вывода двух цифр, `.4` — для четырех цифр, и т. д.

По аналогии с использованием `f` для дробных чисел, можно воспользоваться спецификатором формата `e` для экспоненциальной (научной) записи. Пример использования этой возможности:

```
sci_number = 0.00000000412733

print(f"Sci notation: {sci_number:e}")
# Вывод: Sci notation: 4.1227330e-09

print(f"Sci notation: {sci_number:.2e}")
# Вывод: Sci notation: 4.13e-09
```

Другая распространенная форма числовых значений — проценты. При выводе процентов используется спецификатор формата `%`. Как и в случае со спецификаторами `e` и `f`, мы можем использовать спецификатор `%` сам по себе или с указанием точности (например, `.2` для вывода двух знаков в дробной части):

```
pct_number = 0.179323

print(f"Percentage: {pct_number:%}")
# Вывод: Percentage: 17.932300%

print(f"Percentage two digits: {pct_number:.2%}")
# Вывод: Percentage two digits: 17.93%
```

Помимо этих спецификаторов, f-строки поддерживают и другие. В табл. 2.2 приведены популярные спецификаторы, применяемые к f-строкам при работе с числами.

Таблица 2.2. Популярные спецификаторы при форматировании чисел с использованием f-строк

Числовой тип	F-строка	Вывод	Описание
`int`	`f"{number:b}"`	`"1111"`	Двоичный формат (запись по основанию 2)
	`f"{number:c}"`	`"\x0f"`	Представление целого числа в Юникоде
	`f"{number:d}"`	`"15"`	Десятичный формат (запись по основанию 10)
	`f"{number:o}"`	`"17"`	Восьмеричный формат (запись по основанию 8)
	`f"{number:x}"`	`"f"`	Шестнадцатеричный формат (запись по основанию 16)
`float`	`f"{point:.2e}"`	`"1.23e+00"`	Научная запись
	`f"{point:.2f}"`	`"1.23"`	Запись с фиксированной точкой и двумя цифрами в дробной части
	`f"{point:.2g}"`	`"1.23"`	Общий формат с автоматическим применением e или f
	`f"{point:.2%}"`	`"123.45%"`	Проценты с точностью 2 знака[a]

[a] Мы определяем `number` как целочисленную переменную (`number = 15`), а `point` — как переменную с плавающей точкой (`point = 1.2345`). Обратите внимание: часть `.2` в спецификаторе формата для `float` не является обязательной. При использовании записи `.3` будет использоваться точность с тремя цифрами в дробной части.

2.1.5. Обсуждение

Хотя с прямой интерполяцией выражений f-строками код становится более чистым, избегайте использования сложных выражений в f-строках — они могут запутать читателей вашего кода. Если выражения слишком сложные, создайте промежуточные переменные с содержательными именами.

В Python все еще поддерживаются традиционные способы в стиле C и с использованием `format`, но реальной необходимости в их изучении нет (впрочем, они могут встретиться вам в старом коде). Каждый раз, когда вам потребуется создать строковый вывод, используйте f-строки. И не забывайте о выравнивании текста и форматировании числовых значений — это сделает текстовый вывод более понятным.

2.1.6. Задача

Джеймс работает в IT-отделе компании оптовой торговли и готовит шаблон для ценников. Допустим, данные товара сохраняются в объекте `dict`: `{"name": "Vacuum", "price": 130.675}`. Как Джеймсу записать f-строку, если нужно, чтобы в ценнике выводилась строка `Vacuum: {130.68}`? Обратите внимание: цена должна выводиться с точностью до двух знаков, а вывод включает фигурные скобки — символы, используемые для строковой интерполяции в f-строках.

> **ПОДСКАЗКА** Фигурные скобки являются специальными символами в f-строках. Если строковый литерал включает специальные символы, необходимо экранировать их, чтобы они не рассматривались как специальные символы. Для экранирования фигурных скобок включите дополнительную фигурную скобку: {{ означает {, а }} означает }.

2.2. КАК ПРЕОБРАЗОВАТЬ СТРОКИ ДЛЯ ПОЛУЧЕНИЯ ПРЕДСТАВЛЯЕМЫХ ДАННЫХ

Хотя внешне строки являются текстовыми данными, фактически они могут представлять целые числа, словари и другие типы данных. Например, встроенная функция `input` предоставляет простейший способ получения пользовательского ввода с консоли Python:

```
>>> age = input("Please enter your age: ")
Please enter your age: 35
>>> type(age)  ◄—— Проверяет тип переменной
<class 'str'>
```

Как показывает приведенный фрагмент кода, ввод пользователя представляет собой строку. Допустим, вы хотите проверить, что пользователю исполнилось не менее 18 лет. Казалось бы, достаточно выполнить следующий фрагмент:

```
>>> age > 18
# ERROR: TypeError: '>' not supported between instances of 'str' and 'int'
```

К сожалению, сравнение не работает, потому что `age` является строкой, а строку нельзя сравнивать с целым числом. В более широком смысле многие ситуации требуют преобразования строк в списки, словари и другие применимые типы данных. Такое преобразование крайне важно для последующей обработки данных. В этом разделе вы узнаете, как проверить типы данных, представляемые строками, и как правильно преобразовывать строки к нужному типу данных.

2.2.1. Проверка строк на представление алфавитно-цифровых значений

В языке Python строки могут содержать любую последовательность символов, которые вводятся с клавиатуры. Одна из типичных задач при работе с ними — проверка того, содержит ли строка только алфавитно-цифровые символы. В этом разделе рассматриваются различные способы проверки символов строки.

Допустим, таск-менеджер требует, чтобы пользователи вводили свое имя, которое должно состоять из алфавитно-цифровых символов. Для реализации этой функциональности можно воспользоваться методом `isalnum`, проверяющим, что строка состоит только из символов `a-z`, `A-Z` и `0-9`. Несколько примеров:

```
bad_username0 = "123!@#"
assert bad_username0.isalnum() == False

bad_username1 = "abc..."
assert bad_username1.isalnum() == False

good_username = "1a2b3c"
assert good_username.isalnum() == True
```

Предположим, название создаваемой задачи должно состоять только из букв. Для проверки этого можно воспользоваться методом `isalpha`, который возвращает `True` или `False`. Как вы, вероятно, уже заметили, все `is`-методы возвращают логические значения:

```
assert "Homework".isalpha() == True

assert "Homework123".isalpha() == False
```

Аналогичным образом при помощи метода `isnumeric` можно проверить, являются ли все символы в строке цифровыми символами:

```
assert "123".isnumeric() == True

assert "a123".isnumeric() == False
```

Стоит обсудить несколько подводных камней, на которые можно натолкнуться при использовании метода `isnumeric` для проверки того, представляет ли строка числовое значение:

- Строки, представляющие числа с плавающей точкой, не проходят проверку `isnumeric`. Казалось бы, для строк, содержащих действительные числовые значения, вызов этого метода должен возвращать `True`. К сожалению, это не так:

  ```
  assert "3.5".isnumeric() == False
  ```

- Строки, представляющие отрицательные числа, не проходят проверку `isnumeric`. Вероятно, это тоже противоречит нашим интуитивным представлениям, как в следующем примере:

```
assert "-2".isnumeric() == False
```

- Для пустых строк `isnumeric` возвращает `False`. Пожалуй, интерпретация пустых строк как нецифровых значений соответствует нашим ожиданиям. При преобразовании строк в целые числа следует понимать, что происходит.

Чтобы избежать всех этих проблем, запомните, что строка выдает значение `True` при проверке методом `isnumeric` только в том случае, если все символы непустой строки являются цифровыми. К категории цифровых символов не относятся точка или знак «минус». По этой причине метод `isnumeric` интерпретирует числа с плавающей точкой и отрицательные числа как `False`.

Чем различаются isnumeric, isdigit и isdecimal

Методы `isdigit` и `isdecimal`, родственные с `isnumeric`, часто используются для проверки того, содержат ли строки только цифры или символы десятичных чисел. Казалось бы, их имена означают одно и то же, и в большинстве случаев — например, для строки `"123"` — они возвращают одинаковые логические результаты. Однако существуют нюансы, из-за которых они возвращают разные значения для некоторых строк, особенно если числовые строки не содержат арабские цифры.

По определению эти три метода связаны следующими отношениями в контексте жесткости проверки: `isdecimal` < `isdigit` < `isnumeric`. Если вы боитесь запутаться в этих методах, лучше использовать наиболее общий метод `isnumeric`.

Напомним, что кроме описанных методов `is...` для проверки числового содержимого у строк Python имеются другие методы `is...` для выполнения других проверок, например `islower` и `isupper`. И хотя в этой книге другие методы `is...` не рассматриваются, вам стоит ознакомиться с ними.

ОБРАТИТЕ ВНИМАНИЕ В категории `is...` присутствует интересный метод `isidentifier`. Он проверяет, является ли строка допустимым идентификатором, который может использоваться в качестве имени переменной, функции или объекта.

2.2.2. Преобразование строк в числа

В предыдущем разделе вы узнали, как проверить, представляет ли строка положительное целое число. Однако вообще-то не существует простого способа определить, представляет ли строка числовое значение, особенно если это число

с плавающей точкой или отрицательное число. Преобразования строк в числа важны, потому что со строками невозможно выполнять числовые вычисления — например, сравнить `age` с `18`. Поэтому во многих случаях для последующей обработки требуется получить числовое значение, представляемое строкой. В этом разделе вы научитесь преобразовывать строки в числа — этот процесс также называется *приведением типа* (casting).

> **ОСНОВНЫЕ ПОНЯТИЯ** В программировании *приведением типа* называется процесс преобразования типа данных к другому типу данных (например, преобразования строки в целое число).

Для работы с числовыми значениями обычно используются типы `float` и `int`. Для создания экземпляров этих типов на базе строк используется синтаксис `float("string")` и `int("string")`. Python вычисляет объекты строк, чтобы преобразовать их в правильный объект `float` или `int`, если это возможно.

Если вы предполагаете, что строка представляет число с плавающей точкой, передайте ее встроенному конструктору `float`. В следующих примерах все преобразованные числа относятся к типу `float`, хотя строка представляет целое число:

```
>>> float("3.25")
3.25
>>> float("-2")  ◄—— Создается float, хотя строка содержит целое число
-2.0
```

> **ОСНОВНЫЕ ПОНЯТИЯ** *Конструктором* (constructor) называется особая разновидность функций, которые создают экземпляр (объект) класса. За дополнительной информацией по этой теме обращайтесь к главе 8. Здесь мы используем конструкторы `float` и `int` для создания объектов типов `float` и `int` соответственно.

Если вы ожидаете, что строка содержит целое число, используйте встроенный конструктор `int`:

```
>>> int("-5")
-5
>>> int("123")
123
```

Операции преобразования завершаются успешно, если эти строки содержат предполагаемые числовые значения. Но в противном случае попытка преобразования приводит к ошибке, а ваша программа аварийно завершается, как показано в следующем фрагменте:

```
>>> float("3.5a")
# ERROR: ValueError: could not convert string to float: '3.5a'

>>> int("one")
# ERROR: ValueError: invalid literal for int() with base 10: 'one'
```

Чтобы программа не завершалась из-за этой ошибки, используйте конструкцию `try...except...` для обработки исключения. Хотя я не буду подробно обсуждать эту тему здесь, следующий листинг продемонстрирует такое применение. Данная возможность рассматривается в главе 12 (раздел 12.3).

Листинг 2.4. Преобразование чисел в строки

```
def cast_number(number_str):
    try:
        casted_number = float(number_str)
    except ValueError:
        print(f"Couldn't cast {repr(number_str)} to a number")
    else:
        print(f"Casting {repr(number_str)} to {casted_number}")
```

Функция `repr` создает строку в формате с кавычками

```
# Использование функции с консоли
>>> cast_number("1.5")
Casting '1.5' to 1.5
>>> cast_number("2.3a")
Couldn't cast '2.3a' to a number
```

2.2.3. Вычисление строк для получения представляемых данных

Кроме числовых значений, в приложениях также используются текстовые данные, представляющие данные других типов, например списки и кортежи. Так, в веб-приложении данные часто вводятся в текстовом формате — скажем, строка «`[1, 2, 3]`» представляет объект `list`. Так как в данном случае используется тип данных `str`, к текстовым данным не могут применяться никакие методы `list`; иначе говоря, методы `list` могут вызываться только для объектов `list`. В таком случае требуется преобразование данных. В этом разделе вы узнаете, как получить из строк другие виды данных помимо чисел.

В предыдущем разделе вы научились использовать конструкторы `float` и `int` для преобразования строк и получения числовых значений. Впрочем, решение, когда вызывается конструктор с объектом строки, работает не всегда. Возьмем три распространенных типа данных — `list`, `tuple` и `dict`; в следующем фрагменте кода они представлены в строковом виде:

```
numbers_list_str = "[1, 2]"
numbers_tuple_str = "(1, 2)"
numbers_dict_str = "{1:'one', 2: 'two'}"
```

Если вы попытаетесь передать строки соответствующим конструкторам, происходит нечто неожиданное:

```
>>> list(numbers_list_str)
['[', '1', ',', ' ', '2', ']']

>>> tuple(numbers_tuple_str)
```

Экземпляры списков и кортежей могут создаваться на базе строк

```
('(', '1', ',', ' ', '2', ')')
```

```
>>> dict(numbers_dict_str)
# ERROR: ValueError: dictionary update sequence element #0 has length 1; 2 is
➥ required
```

Хотя конструкторы `list` и `tuple` создают объекты `list` и `tuple`, интерпретируя строки как итерируемые объекты, созданные объекты содержат не те данные, которые можно было бы ожидать от этих строк. Дело в том, что строка представляет собой итерируемый объект, состоящий из символов. Когда вы включаете строку в конструктор `list`, ее символы становятся элементами созданного объекта `list`. То же самое происходит с конструктором `tuple`.

> **ОСНОВНЫЕ ПОНЯТИЯ** *Итерируемыми объектами* (iterables) называются объекты, предоставляющие возможность последовательного перебора элементов. Строки, списки и кортежи — типичные примеры итерируемых объектов. За дополнительной информацией об итерируемых объектах обращайтесь к главе 5.

Чтобы обойти это непредсказуемое поведение, используйте встроенную функцию `eval`. Функция получает строку так, словно она была введена с консоли, и возвращает вычисленный результат:

```
assert eval(numbers_list_str) == [1, 2]
```

```
assert eval(numbers_tuple_str) == (1, 2)
```

```
assert eval(numbers_dict_str) == {1: 'one', 2: 'two'}
```

При вычислении этих строк вы получаете данные, представляемые этими строками. Такое преобразование полезно, потому что текст часто используется как формат обмена данными. Преимущество `eval` заключается в том, что результат вычисления переданного текста гарантированно будет таким, как если бы вы выполнили тот же текст как код с консоли.

Будьте осторожны с eval и exec

Использование `eval` лучше ограничить доверенными источниками данных, потому что `eval` вычисляет строку так, как если бы код являлся частью программы. Эта проблема продемонстрирована в следующем фрагменте. Вычисление некорректного кода приводит к ошибке SyntaxError, которая может вызвать аварийное завершение программы:

```
>>> eval("[1, 2")
...(пропущенные строки)
SyntaxError: unexpected EOF while parsing
```

Также существует другая встроенная функция `exec`, похожая на `eval`. Функция `exec` может выполнить строку так, как если бы эта строка была частью программы. Самое заметное различие между `exec` и `eval` заключается в том, что `eval` вычисляет и возвращает выражение, а `exec` может принимать выражения и инструкции (например, `if...else...`), но ничего не возвращает. Хотя обе функции наделяют приложение возможностью динамического генерирования кода, их некорректное использование создает угрозу для работоспособности приложения и даже компьютера. Например, функции `exec` можно передать строку `"os.system('rm -rf *')"`, что приведет к удалению всех папок и файлов на вашем компьютере.

Поэтому если ваше приложение должно обрабатывать строки динамически с использованием `eval` и `exec`, будьте очень внимательны. В качестве альтернативы `eval` можно рассмотреть модуль `ast` из стандартной библиотеки; функция `literal_eval` этого модуля выполняет безопасное вычисление строк.

Если у вас возникнут сомнения в безопасности источника данных вашего приложения, я рекомендую разобрать строки самостоятельно. Например, если потребуется получить из строки объект `list` с целыми числами, можно удалить квадратные скобки и разбить строку для воссоздания объекта `list`. Ниже приведен простейший пример такого рода. В нем используются некоторые средства (в частности, разбиение строк и списковые включения), которые будут рассмотрены позднее (разделы 2.3 и 5.2):

```
list_str = "[1, 2, 3, 4]"
stripped_str = list_str.strip("[]")
number_list = [int(x) for x in stripped_str.split(",")]

print(number_list)
# Вывод: [1, 2, 3, 4]
```

СОПРОВОЖДАЕМОСТЬ Применение `eval` без проверки целостности объекта строки может привести к ошибкам и даже катастрофическим последствиям. Будьте очень осторожны, когда вам приходится пользоваться этим способом.

2.2.4. Обсуждение

При использовании конструктора `float` или `int` для получения фактических числовых значений, представляемых строкой, рассмотрите возможность применения `try...except`.... Успех преобразования никогда не гарантирован, а в случае неудачи программа завершится аварийно, если исключение не будет обработано. Будьте осторожны с применением `eval` для получения представляемых данных, потому что ненадежные источники опасны для программы. Если вы озабочены безопасностью данных, стоит подумать о самостоятельном парсинге данных или

об использовании таких проверенных средств, как модуль `ast`. Если вы работаете с собственными данными (например, пишете сценарий их обработки), для получения данных можно пользоваться `eval`.

2.2.5. Задача

В начале этого раздела вы узнали, как получить входные данные от пользователя при помощи функции `input`. Мэри — школьная учительница, которая хочет написать простую демонстрационную программу для своих учеников. Допустим, программа должна запрашивать температуру по Цельсию с консоли Python. Как ей написать программу, чтобы та удовлетворяла приведенным ниже требованиям? `x` — значение, введенное пользователем:

- Если температура < 10 градусов, программа выводит сообщение: «`You entered x degrees. It's cold!`» (Вы ввели x градусов. Холодно!)
- Если температура лежит в диапазоне от 10 до 25 градусов, программа выводит сообщение «`You entered x degrees. It's cool!`» (Вы ввели x градусов. Прохладно!)
- Если температура > 25 градусов, программа выводит сообщение «`You entered x degrees. It's hot!`» (Вы ввели x градусов. Жарко!)
- Значение `x` должно выводиться с точностью до одного знака в дробной части. Например, если пользователь вводит `15.75`, значение должно выводиться в виде `15.8`.

  ПОДСКАЗКА Введенную строку необходимо преобразовать в число с плавающей точкой, прежде чем сравнивать ее с другими числами. Для создания строкового вывода используйте f-строки. И не забывайте о спецификаторах формата!

2.3. КАК ОБЪЕДИНЯТЬ И РАЗБИВАТЬ СТРОКИ

Строки не всегда хранятся в том формате, который вам нужен. В ряде случаев отдельные строки представляют разрозненные фрагменты взаимосвязанной информации, которые необходимо объединить в одну строку. Допустим, пользователь вводит несколько строк с названиями фруктов, которые ему нравятся. Эти строки можно объединить в одну строку со списком предпочтений пользователя:

```
# Исходный ввод
fruit0 = "apple"
fruit1 = "banana"
fruit2 = "orange"

# Нужный вывод
liked_fruits = "apple, banana, orange"
```

В других ситуациях требуется разбить строку и создать несколько строк из ее частей. Скажем, пользователь вводит перечень стран, в которых он побывал, в одной строке. Требуется создать список с названиями стран:

```
# Исходный ввод
visited_countries = "United States, China, France, Canada"

# Нужный вывод
countries = ["United States", "China", "France", "Canada"]
```

Эти два сценария дают вполне реалистичные примеры основных операций обработки строк, с которыми вы можете столкнуться в реальном проекте. В этом разделе мы рассмотрим базовую функциональность объединения и разбиения строк.

2.3.1. Объединение строк с пробельными символами

Для объединения строк используется оператор конкатенации — символ +, представленный в листинге 2.1. При наличии нескольких строковых литералов их можно объединять, если они разделяются пробельными символами (whitespaces): пробелами, табуляциями и символами новой строки. В этом разделе вы узнаете, как объединяются строки, разделенные пробельными символами.

Допустим, в приложении используются разные конфигурации, определяющие стиль отображения информации. Конфигурации оформлены в виде строковых литералов, и эти отдельные настройки объединяются автоматически:

```
style_settings = "font-size=large, " "font=Arial, " "color=black, "
➥ "align=center"

print(style_settings)
# Вывод: font-size=large, font=Arial, color=black, align=center
```

Автоматическая конкатенация может выполняться только между строковыми литералами. Этот метод не применяется к строковым переменным или комбинациям строковых литералов и переменных. F-строки также поддерживают автоматическую конкатенацию. Эта возможность особенно полезна при построении длинных f-строк с разбиением строковых литералов на отдельные строки кода для ясности:

```
settings = {"font_size": "large", "font": "Arial", "color":
➥ "black", "align": "center"}

styles = f"font-size={settings['font_size']}, " \
         f"font={settings['font']}, " \
         f"color={settings['color']}, " \
         f"align={settings['align']}"
```

Обратный слеш — символ продолжения строки

УДОБОЧИТАЕМОСТЬ Если в программе используется длинная строка, рассмотрите возможность разбиения ее на несколько частей, каждая из которых представляет осмысленную подстроку. Если такие подстроки разделяются пробельными символами, они могут объединяться автоматически.

2.3.2. Объединение строк без ограничителей

Объединение строк, разделенных пробелами, может сбивать с толку, потому что границы (пробелы) между строковыми литералами несколько усложняют зрительное восприятие отдельных строк. Более того, такие ограничители не могут встречаться внутри строковых литералов, что создает дополнительные затруднения. В общем случае идеальным вариантом становится объединение строк с произвольными ограничителями. В этом разделе вы узнаете, как это делается.

Продолжим пример со стилевым оформлением. Для выполнения конкатенации отдельных строк можно воспользоваться методом `join`:

```
style_settings = ["font-size=large", "font=Arial", "color=black",
➥ "align=center"]
merged_style = ", ".join(style_settings)

print(merged_style)
# Вывод: font-size=large, font=Arial, color=black, align=center
```

Метод `join` получает список строк. Элементы этого списка соединяются последовательно со строкой-ограничителем, использованной при вызове метода. Хотя в данном случае используется объект списка, в более общем варианте это может быть произвольный итерируемый объект, например кортеж или множество.

`str.join` или `list.join`

Откровенно говоря, когда я только начал изучать Python, вызов метода `"separator".join(the_list)` несколько озадачил меня, потому что мне казалось, что объект `list` должен находиться перед спецификатором. Более того, в другом популярном языке JavaScript класс `Array` (аналог `list` в Python) содержит метод `join`, который строит строку с ограничителями из своих элементов. Применяя эту логику, можно ожидать, что объекты `list` в Python тоже будет содержать метод `join`.

К сожалению, это не так. Метод `join` имеется у строк Python. Возникают расхождения между ожиданиями и фактической реализацией. Позднее я осознал, что запомнить правильную сигнатуру метода проще всего так: я хочу использовать конкретный разделитель для объединения всех элементов объекта `list`.

> Но когда вы больше узнаете о Python, становится понятно, что выбранный создателями Python вариант с оформлением `join` в методе строки был чрезвычайно удачным. Объединять с разделителем можно не только элементы списка, `join` также может использоваться с кортежами, множествами, словарями, объектами отображений и любыми другими итерируемыми объектами. Если бы в Python метод `join` был оформлен как метод списка, то для включения аналогичной функциональности в другие итерируемые объекты Python пришлось бы реализовать `join` для каждого типа итерируемого объекта, а это нарушает принцип DRY!

По сравнению с прямой конкатенацией метод `join` лучше читается, так как строки-компоненты являются отдельными элементами и читателю кода проще увидеть, что именно объединяется. Важным является следующее дополнительное преимущество `join`: вы можете динамически обрабатывать элементы объекта `list`.

Допустим, вы хотите создать в таск-менеджере строку с перечнем задач, которые необходимо выполнить за неделю. Изначально задачи хранятся в виде строк в элементах списка. Объединяя эти строки, можно сгенерировать список и отображать его на рабочем столе как напоминание:

```
tasks = ["Homework", "Grocery", "Laundry", "Museum Trip", "Buy Furniture"]
note = ", ".join(tasks)

print("Remaining Tasks:", note)
# Вывод : Remaining Tasks: Homework, Grocery, Laundry, Museum Trip, Buy
➥ Furniture
```

После того как пользователь выполнит часть задач, эти задачи удаляются из списка:

```
tasks.remove("Buy Furniture")
tasks.remove("Homework")
```

После удаления задач вы все еще можете воспользоваться методом `join` для создания необходимой строки:

```
print("Remaining Tasks: ", ", ".join(tasks))
# Вывод: Remaining Tasks:  Grocery, Laundry, Museum Trip
```

В этом примере продемонстрированы сценарии использования со списками строк, которые могут изменяться динамически. С появлением дополнительных задач можно включать их в объект `list` и заново генерировать обновленную строку методом `join`.

2.3.3. Разбиение строк для создания списка

Для сохранения и передачи данных часто используются текстовые файлы. Например, табличные данные можно сохранить в текстовом файле, чтобы каждая строка представляла отдельную запись. При чтении текстового файла каждой записи данных соответствует одна строка, содержащая несколько подстрок, а каждая подстрока представляет значение некоторого поля. Чтобы обработать данные, необходимо разбить строку и извлечь эти значения в отдельные подстроки. В этом разделе рассматриваются различные темы, связанные с разбиением строк.

Предположим, у нас есть текстовый файл с именем `"task_data.txt"`, в котором хранятся описания задач. Каждая строка представляет информацию по отдельной задаче, как показано в следующем фрагменте: идентификатор, название и уровень важности. Так как чтение данных из файла рассматривается далее, в главе 11, будем считать, что вы уже прочитали текстовые данные и сохранили их в многострочной строке в тройных кавычках:

```
task_data = """1001,Homework,5
1002,Laundry,3
1003,Grocery,4"""
```

> **ОБРАТИТЕ ВНИМАНИЕ** Для создания строк в тройных кавычках, охватывающих несколько физических строк, могут использоваться как одинарные, так и двойные кавычки. Синтаксис тройных кавычек также поддерживается для многострочных f-строк.

Для обработки таких строк мы воспользуемся методом `split`, который ищет заданные ограничители и разбивает по ним строку. Возможное решение приведено в листинге 2.5.

Листинг 2.5. Обработка текстовых данных с разбиением строк

```
processed_tasks = []
for data_line in task_data.split("\n"):
    processed_task = data_line.split(",")  <-- Разбивает текст каждой строки
    processed_tasks.append(processed_task)

print(processed_tasks)
# Выводит следующую строку:
[['1001', 'Homework', '5'], ['1002', 'Laundry', '3'], ['1003', 'Grocery', '4']]
```

У метода `split` есть ограничение: он позволяет задать только один разделитель, что может привести к проблемам при разбиении строк с разными разделителями. Допустим, в текстовом файле в качестве разделителей используются как запятые, так и символы подчеркивания. Упрощая задачу, примем, что между словами

используется только один разделитель. Для демонстрации возьмем следующую строку данных: `messy_data = "process,messy_data_mixed,separators"`.

Эта проблема часто встречается в реальной жизни, когда мы работаем с данными, не прошедшими обработку. В таких случаях приходится искать решения на программном уровне, потому что текстовый файл может содержать многие тысячи записей. Очевидно, метод `split` с такими записями работать не будет, так как задать можно только один разделитель. Рассмотрим альтернативные решения.

1. Последовательное применение разделителей.

   а) Строки разбиваются по запятым, в результате разбиения создается список.

   б) Проверяем, содержат ли элементы списка символы подчеркивания. Если их нет, значит, элемент готов к дальнейшей обработке. Если они есть, выполняется второе разбиение по подчеркиваниям:

```
separated_words0 = []
for word in messy_data.split(","):
    if word.find("_") < 0:
        separated_words0.append(word)
    else:
        separated_words0.extend(word.split("_"))
```

Если совпадение не найдено, результат будет равен –1

Метод extend присоединяет все элементы разделенных строк

2. Консолидация разделителей.

   Так как мы знаем, что существуют всего два возможных разделителя, один разделитель можно преобразовать в другой. Это позволит вызвать метод `split` всего один раз для завершения нужной операции:

```
consolidated = messy_data.replace(",", "_")
separated_words1 = consolidated.split("_")
```

Использует метод replace для замены подстроки

Эти два решения просты и прямолинейны. При хорошем знании базовых операций со строками и списками они окажутся идеальными в случае, если быстродействие не критично, потому что они потребуют нескольких проходов для проверки разделителей. А если вам приходится иметь дело с несколькими разделителями? В таком случае операции потребуют больших затрат вычислительных ресурсов.

Существует ли более производительное решение? Да, существует. Регулярные выражения разрабатывались специально для более сложных операций поиска по шаблону и замены (разделы 2.4 и 2.5).

ОСНОВНЫЕ ПОНЯТИЯ *Регулярные выражения* — последовательности символов, описывающие условия поиска. Их часто сокращают как *regex* (от regular expressions).

2.3.4. Обсуждение

Выбор между конкатенацией строк, f-строками или `join` должен определяться для каждого конкретного случая. Главный критерий — удобочитаемость кода. Если вам нужно объединить небольшое количество строк, используйте операторы конкатенации. Если строк много, рассмотрите возможность применения f-строк для группировки взаимосвязанных строк. Метод `join` особенно полезен для объединения отдельных строк, сохраненных в итерируемом объекте.

Кроме `split`, есть также метод `rsplit` со сходной функциональностью. Единственное его отличие связано со способом задания параметра `maxsplit`, определяющего максимальное количество элементов, которые могут быть созданы в результате разбиения. В разделе 2.3.5 методы `split` и `rsplit` рассматриваются более подробно.

2.3.5. Задача

Ниже приведены сигнатуры методов `split` и `rsplit`. Оба метода получают аргументы с разделителем и максимальным количеством создаваемых элементов. Напишите несколько строк, с которыми эти методы будут работать одинаково или по-разному.

```
str.split(separator, maxsplit)
str.rsplit(separator, maxsplit)
```

> **ПОДСКАЗКА** В общем случае оба метода работают одинаково. Различия проявляются тогда, когда количество элементов, полученных в результате разбиения, оказывается выше максимального.

2.4. КАКИЕ ВОЗМОЖНОСТИ ПРЕДОСТАВЛЯЮТ РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Класс `str` в Python содержит такие полезные методы для поиска подстрок, как `find` и `rfind`. Впрочем, во многих практических ситуациях их возможности оказываются недостаточными, особенно когда речь заходит о сложном поиске по шаблону. В таких случаях стоит подумать об использовании регулярных выражений. В предыдущем разделе я упоминал о том, что регулярные выражения могут применяться для разбиения строк по нескольким видам разделителей, — в таких ситуациях трудно обойтись собственными методами `str`. Решение с помощью регулярных выражений может выглядеть так:

```
import re

regex = re.compile(r"[,_]")  ◄—— Компилирует регулярное выражение
separated_words2 = regex.split(messy_data)
```

Отметим, что строка для завершения разбиения обходится только один раз. С несколькими разделителями регулярные выражения работают намного эффективнее двух других решений (раздел 2.3.3), требующих нескольких проходов по строке. Благодаря своей гибкости и производительности регулярные выражения становятся незаменимым средством нетривиальной обработки строк. В этом разделе я буду использовать поиск по строке для объяснения механизмов работы регулярных выражений.

ОБРАТИТЕ ВНИМАНИЕ Регулярные выражения не привязаны к конкретному языку. Они поддерживаются всеми распространенными языками программирования, хотя синтаксис их реализации может иметь некоторые различия. Однако сами регулярные выражения остаются практически неизменными; можно считать, что в разных языках программирования используются разные диалекты регулярных выражений.

2.4.1. Работа с регулярными выражениями в Python

Изучение регулярных выражений стоит начать с картины в целом: с модуля и его базового синтаксиса. В этом разделе приводится самый общий обзор регулярных выражений в Python.

Стандартная библиотека Python включает модуль `re` с функциональностью, связанной с регулярными выражениями. Существуют два варианта использования этого модуля. Первый вариант относится к аспектам объектно-ориентированного программирования (ООП) в Python. Применяя парадигму ООП к регулярным выражениям (рис. 2.4), вы выполняете операции с объектами `Pattern`. В этом варианте вы сначала создаете объект `Pattern`, компилируя заданный строковый шаблон. Затем объект `Pattern` используется для поиска частей текста, соответствующих шаблону.

ОСНОВНЫЕ ПОНЯТИЯ *ООП* (*объектно-ориентированное программирование*) — модель программной архитектуры, в которой центральное место занимают данные и объекты (вместо функций и процедур).

Следующий фрагмент кода показывает, как применить парадигму ООП для поиска по шаблону с применением регулярных выражений:

```
import re

regex = re.compile("do")  <-- Создает шаблон
regex.pattern  <-- Обращается к атрибутам
regex.search("do homework")  | Использует
regex.findall("don't do that")  | методы
```

В другом стиле применяется функциональный подход. Вместо того чтобы создавать объект `Pattern`, мы вызываем функции непосредственно из модуля.

При вызове функции задается шаблон, а также строка, к которой этот шаблон применяется:

```
import re

re.search("pattern", "the string to be searched")
re.findall("pattern", "the string to be searched")
```

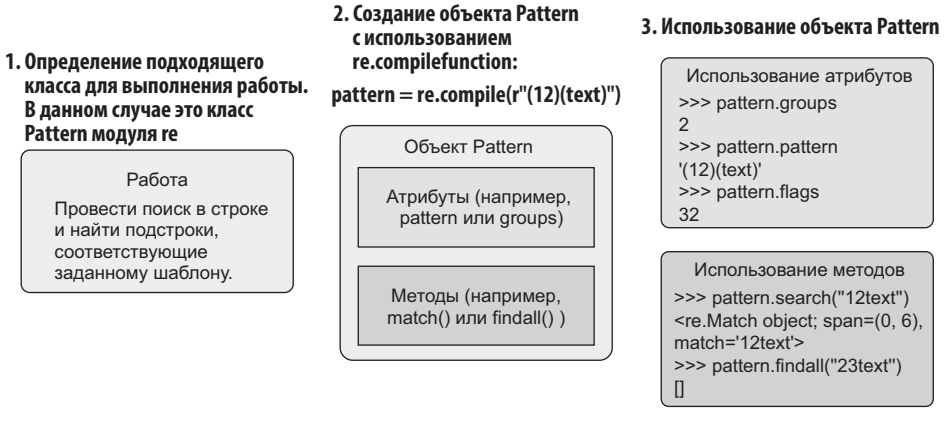


Рис. 2.4. Применение общей модели ООП при поиске по шаблону. В модели ООП разработчик сначала выбирает подходящий класс для решения задачи. В данном случае используется класс Pattern из модуля re. На втором шаге создается экземпляр (объект) класса. В парадигме ООП объект состоит из атрибутов, для обращения к которым используется точечная запись, и методов, вызываемых с круглыми скобками. На третьем шаге используется созданный объект Pattern (например, программа обращается к его атрибутам или вызывает его методы)

Во внутренней реализации при вызове `re.search` Python сам создает объект `Pattern` и вызывает метод `search` для шаблона. Таким образом, использование модуля для вызова функций становится удобным способом работы с регулярными выражениями. Однако следует учитывать, что при создании объекта `Pattern` функцией `compile` откомпилированный шаблон кэшируется, а многократное использование шаблона становится более эффективным, потому что шаблон не нужно компилировать повторно.

> **ОСНОВНЫЕ ПОНЯТИЯ** Механизм кэширования используется в программировании (и компьютерных вычислениях вообще) для временного хранения данных, чтобы будущие запросы этих данных обслуживались быстрее.

С другой стороны, с функциональным подходом шаблон создается «на ходу», и преимущества от повышения эффективности за счет кэширования теряются. Поэтому если вы собираетесь использовать шаблон однократно, о различиях этих двух подходов можно не задумываться.

2.4.2. Определение шаблона в виде необработанной строки

Мощь регулярных выражений проявляется прежде всего в компактности шаблона, описывающего широкий диапазон возможностей. Для создания шаблона часто приходится использовать необработанные (raw) строки, например строковый литерал с префиксом r (`r"шаблон"`). В этом разделе вы увидите, почему необходимо использовать необработанные строки для построения шаблонов регулярных выражений.

В регулярных выражениях `\d` обозначает любую цифру, а `\w` — символ слова в Юникоде. Это примеры специальных символов в регулярных выражениях, а префикс `\` указывает, что эти символы имеют особый смысл помимо того, чем они кажутся на первый взгляд. В строках Python символ `\` также используется для обозначения специальных символов, например `\t` для табуляции, `\n` для новой строки или `\\` для обратного слеша.

Когда эти обозначения объединяются в строках, появляются шаблоны довольно странного вида. Допустим, вы хотите найти подстроку `\task`. Следует заметить, что `\t` в данном случае является литералом; эти два символа действительно обозначают символ обратного слеша и букву `t`, а не символ табуляции. Чтобы Python мог искать подстроку `\task`, необходимо использовать обозначение `\\task`. Ситуация дополнительно усложняется тем, что при создании такого шаблона оба символа `\` должны экранироваться, из-за чего при поиске `\task` необходимо использовать четыре символа обратного слеша (`\\\\task`). Вам это кажется странным? Возьмем следующий код:

```
task_pattern = re.compile("\\\\task")
texts = ["\task", "\\task", "\\\task", "\\\\task"]
for text in texts:
    print(f"Match {text!r}: {task_pattern.match(text)}")

# Выводимые строки:
Match '\task': None
Match '\\task': <re.Match object; span=(0, 5), match='\\task'>
Match '\\\task': None
Match '\\\\task': None
```

Как видно из вывода, шаблон совпадает только с `"\\task"`. Это вполне логично: два последовательных обратных слеша `\\` интерпретируются как литерал `\`, и строка фактически превращается в `"\task"`, что совпадает с искомым шаблоном.

Очевидно, лишние символы `\` создают путаницу. Для решения этой проблемы мы используем синтаксис необработанных строк, чтобы Python не обрабатывал лишние символы `\`. Как и в случае с `f`-строками, строка снабжается специальным префиксом `r`, чтобы обычный строковый литерал превратился в необработанную

строку. Применяя необработанные строки к шаблону, мы получаем следующее решение:

```
task_pattern_r = re.compile(r"\\task")
texts = ["\task", "\\task", "\\\task", "\\\\task"]
for text in texts:
    print(f"Match {text!r}: {task_pattern_r.match(text)}")
```

```
# Выводимые строки:
Match '\task': None
Match '\\task': <re.Match object; span=(0, 5), match='\\task'>
Match '\\\task': None
Match '\\\\task': None
```

Как видно из вывода, необработанная строка определяет более понятный шаблон, чем литерал обычной строки с четырьмя последовательными символами \. Легко представить, что при построении более сложного шаблона для обозначения специальных символов потребуется еще больше символов \. Без необработанных строк ваши шаблоны превратятся в головоломку. Таким образом, для создания шаблонов регулярных выражений всегда рекомендуется применять необработанные строки.

> **УДОБОЧИТАЕМОСТЬ** Использование необработанных строк для построения шаблонов избавляет вас от необходимости экранировать специальный символ \ и делает шаблон более понятным.

2.4.3. Основы синтаксиса регулярных выражений

Синтаксис регулярных выражений ставит в тупик многих программистов. Как упоминалось в разделе 2.4, регулярные выражения представляют собой отдельный язык с собственным синтаксисом. К счастью, Python адаптирует бóльшую часть синтаксиса регулярных выражений. В этом разделе рассматриваются важнейшие компоненты шаблонов.

Границы строк

При работе со строками иногда имеет значение, что строка начинается или заканчивается определенным шаблоном. В подобных ситуациях для обозначения привязки к границам строк используются *якорные символы* (boundary anchors), как показано в следующем фрагменте:

```
^hi         начинается с подстроки hi
task$       заканчивается подстрокой task
^hi task$   начинается и заканчивается подстрокой "hi task"
            (точное совпадение)
```

Символ ^ означает, что совпадение шаблона должно находиться в начале строки, а символ $ — в конце. Ниже приведены примеры использования некоторых якорей:

```
re.search(r"^hi", "hi Python")
# Вывод: <re.Match object; span=(0, 2), match='hi'>

re.search(r"task$", "do the task")
# Вывод: <re.Match object; span=(7, 11), match='task'>

re.search(r"^hi task$", "hi task")
# Вывод: <re.Match object; span=(0, 7), match='hi task'>

re.search(r"^hi task$", "hi Python task")
# Вывод: None (omitted output in an interactive console)
```

Возможно, вам известны методы `startswith` и `endswith` класса `str`, которые работают в простых случаях. Но для более сложных ситуаций, например если потребуется найти строку, начинающуюся с одного или нескольких вхождений `h`, за которыми следует `i`, — метод `startswith` не подходит, потому что приходится учитывать возможности `hi`, `hhi`, `hhhi` и т. д. В таких ситуациях регулярные выражения чрезвычайно полезны.

> **СОПРОВОЖДАЕМОСТЬ** Прежде чем обращаться ко всей мощи регулярных выражений, стоит подумать, не удастся ли обойтись более простым решением, например с startswith или endswith. Такие решения получаются более простыми и надежными.

Квантификаторы

В предыдущем разделе упоминалась проблема поиска переменного количества символов, для чего требуется создавать шаблоны с возможностью задания количества вхождений. Для решения этой проблемы в регулярных выражениях существует категория так называемых квантификаторов. В эту категорию входят следующие специальные символы:

```
hi?       за h следует ноль или одно вхождение i
hi*       за h следует ноль или несколько вхождений i
hi+       за h следует одно или несколько вхождений i
hi{3}     за h следует iii
hi{1,3}   за h следует i, ii или iii
hi{2,}    за h следует 2 или более вхождения i
```

Как показано в этом фрагменте, существуют четыре общих квантификатора: `?` — 0 или 1, `*` — 0 и более, `+` — 1 и более, `{}` — диапазон. Отметим важный момент: поиск совпадений для `?`, `*` и `+` является максимальным («жадным»), то есть при поиске ищется совпадение максимально возможной длины. Чтобы изменить это поведение по умолчанию, к квантификаторам можно присоединить суффикс `?`:

```
test_string = "h hi hii hiii hiiii"
test_patterns = [r"hi?", r"hi*", r"hi+", r"hi{3}", r"hi{2,3}", r"hi{2,}",
                 r"hi??", r"hi*?", r"hi+?", r"hi{2,}?"]
```

```
for pattern in test_patterns:
print(f"{pattern: <9}-->  {re.findall(pattern, test_string)}")

# Выводимые строки:
hi?       --->  ['h', 'hi', 'hi', 'hi', 'hi']
hi*       --->  ['h', 'hi', 'hii', 'hiii', 'hiiii']
hi+       --->  ['hi', 'hii', 'hiii', 'hiiii']
hi{3}     --->  ['hiii', 'hiii']
hi{2,3}   --->  ['hii', 'hiii', 'hiii']
hi{2,}    --->  ['hii', 'hiii', 'hiiii']
hi??      --->  ['h', 'h', 'h', 'h', 'h']
hi*?      --->  ['h', 'h', 'h', 'h', 'h']
hi+?      --->  ['hi', 'hi', 'hi', 'hi']
hi{2,}?   --->  ['hii', 'hii', 'hii']
```

Результаты поиска соответствуют ожидаемым. Обратите внимание на использование в нескольких последних шаблонах суффикса `?`, при наличии которого шаблон совпадает не с самой длинной, а с самой короткой возможной последовательностью.

Символьные классы и множества

Гибкость регулярных выражений обусловлена простотой использования нескольких символов для обозначения различных возможностей. При описании необработанных строк в разделе 2.4.2 я упоминал о том, что последовательность `\d` может представлять любую цифру. В регулярных выражениях также могут задаваться другие символьные множества. Ниже перечислены лишь самые распространенные из них:

```
\d      любая десятичная цифра
\D      любой символ, не являющийся десятичной цифрой
\s      любой пробельный символ, включая пробел, \t, \n, \r, \f, \v
\S      любой символ, не являющийся пробельным
\w      любой символ слова: алфавитно-цифровой или символ подчеркивания
\W      любой символ, не являющийся символом слова
.       любой символ, кроме новой строки
[]      множество перечисленных символов
```

Заслуживают особого внимания некоторые моменты, относящиеся к использованию `[]` с символьными множествами:

- Во множествах можно перечислять отдельные символы. `[abcxyz]` совпадает с любым из этих шести символов, а `[0z]` совпадает с `"0"` и `"z"`.
- Допускается включение интервалов символов. Так, `[a-z]` совпадает с любым символом от `"a"` до `"z"`, а `[A-Z]` совпадает с любым символом от `"A"` до `"Z"`.
- Также допускается перечисление нескольких интервалов. `[a-dw-z]` совпадает с любым символом от `"a"` до `"d"`, и от `"w"` до `"z"`.

Чтобы запомнить, что делает каждый символ, рассмотрим несколько конкретных примеров, как в следующем фрагменте кода:

```
test_text = "#1$2m_ M\t"
patterns = ["\d", "\D", "\s", "\S", "\w", "\W", ".", "[lmn]"]
for pattern in patterns:
    print(f"{pattern: <9}--->  {re.findall(pattern, test_text)}")

# Выводимые строки:
\d       --->  ['1', '2']
\D       --->  ['#', '$', 'm', '_', ' ', 'M', '\t']
\s       --->  [' ', '\t']
\S       --->  ['#', '1', '$', '2', 'm', '_', 'M']
\w       --->  ['1', '2', 'm', '_', 'M']
\W       --->  ['#', '$', ' ', '\t']
.        --->  ['#', '1', '$', '2', 'm', '_', ' ', 'M', '\t']
[lmn]    --->  ['m']
```

Инвертированные классы — своего рода отрицания существующих символьных классов. Например, `\d` совпадает с любой цифрой, а `\D` совпадает с любым символом, который *не является* цифрой. Использование тех же букв в верхнем регистре поможет запомнить смысл инвертированных классов. Впрочем, ключ к освоению регулярных выражений — практика!

Логические операторы

В регулярных выражениях, как и в других языках программирования, существуют логические операции, которые используются для определения шаблонов. Самыми популярными из этих операций являются:

```
a|b       a или b
(abc)     abc как группа
[^a]      любой символ, кроме a
```

Круглые скобки используются для перечисления символов, которые должны присутствовать для совпадения, а знак ^ создает символьное множество инвертированием заданного множества. Например, чтобы найти любой символ, отличный от `s`, используйте запись `[^s]`. Вот несколько примеров:

```
re.findall(r"a|b", "a c d d b ab")
# Вывод: ['a', 'b', 'a', 'b']

re.findall(r"a|b", "c d d b")
# Вывод: ['b']

re.findall(r"(abc)", "ab bc abc ac")
# Вывод: ['abc']

re.findall(r"(abc)", "ab bc ac")
# Вывод: []

re.findall(r"[^a]", "abcde")
# Вывод: ['b', 'c', 'd', 'e']
```

2.4.4. Анализ совпадений

После того как вы освоите построение шаблонов, возникает задача поиска всех совпадений, как было сделано с методом `findall` (раздел 2.4.3). Метод `findall` оказывается наиболее удобным тогда, когда текст имеет небольшую длину и вы можете легко определить, где находятся совпадения. В реальных проектах часто приходится иметь дело с большими объемами текста, так что простой вывод описания совпадений пользы не принесет. Однако необходимо знать, где находятся совпадения, в каком именно тексте. Для таких ситуаций существуют объекты `Match`. В этом разделе вы научитесь работать с совпадениями.

Создание объектов Match

Методы `match` и `search` часто используются для поиска по шаблону. Различаются они прежде всего тем, где ведется поиск. Метод `match` проверяет, существует ли совпадение в начале строки; метод `search` сканирует строку, пока не найдет совпадение (если оно существует). Несмотря на это различие, оба метода возвращают объект `Match` при обнаружении совпадения. Для знакомства с объектами `Match` возьмем пример с вызовом метода `search`:

```
match = re.search(r"(\w\d)+", "xyza2b1c3dd")
print(match)
# Вывод: <re.Match object; span=(3, 9), match='a2b1c3'>
```

Ключевые атрибуты объекта `Match` — совпадение в строке и диапазон символов. Для их получения можно воспользоваться соответствующими методами `group`, `span`, `start` и `end`, как показано в следующем листинге.

Листинг 2.6. Методы объекта Match

```
print("matched:", match.group())
# Вывод: matched: a2b1c3

print("span:", match.span())
# Вывод: span: (3, 9)

print(f"start: {match.start()} & end: {match.end()}")
# Вывод: start: 3 & end: 9
```

При использовании регулярных выражений некоторые операции выполняются только в том случае, если в тексте найдено совпадение. Для упрощения программирования объект `Match` интерпретируется как `True` в условных инструкциях. В общем случае использование `Match` выглядит так:

```
match = re.match("pattern", "string to match")
if match:
    print("do something with the matched")
else:
    print("found no matches")
```

УДОБОЧИТАЕМОСТЬ При использовании `if...else...` с регулярными выражениями можно включить объект `Match` прямо в условие `if`, так как объект `Match` интерпретируется как `True`.

Работа с группами

Вам может показаться странным получение информации вызовом методов вместо атрибутов: `match.span()` вместо `match.span`. Если вас заинтересовало, почему это так, значит, у вас хорошее чувство принципов ООП. Я согласен с тем, что с позиций ООП данные было бы естественно представлять атрибутами. Однако эта функциональность реализована вызовами методов, потому что поиск по шаблону может привести к нахождению нескольких групп. Внимательно присмотревшись к листингу 2.6, вы увидите, что для получения строки совпадения используется метод `group`. Хотите узнать, когда совпадение может состоять из нескольких групп? Рассмотрим пример:

```
match = re.match(r"(\w+), (\w+)", "Homework, urgent; today")
print(match)
# Вывод: <re.Match object; span=(0, 16), match='Homework, urgent'>

match.groups()
# Вывод: ('Homework', 'urgent')

match.group(0)
# Вывод: 'Homework, urgent'

match.group(1)
# Вывод: 'Homework'

match.group(2)
# Вывод: 'urgent'
```

В этом шаблоне определены две группы (заключенные в круглые скобки), каждая из которых ищет один или несколько символов слов, разделенных запятой и пробелом. Как упоминалось ранее, поиск совпадения ведется в максимальном режиме, поэтому самым длинным возможным совпадением оказывается `'Homework, urgent'`. Это совпадение создает группы, соответствующие тем, которые определены в шаблоне.

По умолчанию группа 0 соответствует полному совпадению. Дальнейшие группы определяются группами внутри шаблона. Так как в результате поиска могут присутствовать совпадения для нескольких групп, для получения информации каждой группы лучше использовать методы вместо атрибутов, которые не могут получать аргументы. Аналогичная группировка также применима к `span`:

```
match.span(0)
# Вывод: (0, 16)

match.span(1)
# Вывод: (0, 8)
```

```
match.span(2)
# Вывод: (10, 16)
```

2.4.5. Часто используемые методы

Чтобы эффективно использовать регулярные выражения в проектах, необходимо знать, какая функциональность нам доступна. В табл. 2.3 перечислены ключевые методы, для каждого из которых приводится пояснительный пример.

Таблица 2.3. Часто используемые методы регулярных выражений

Метод	Пример	Совпадение/ возвращаемое значение
`search`: возвращает объект Match, если совпадение найдено в любой позиции строки	`re.search(r"\d+", "ab12xy")` `re.search(r"\d+", "abxy")`	`'12'` `None`
`match`: возвращает Match только в том случае, если совпадение находится в начале строки	`re.match(r"\d+", "ab12xy")` `re.match(r"\d+", "12abxy")`	`None` `'12'`
`findall`: возвращает список строк, совпадающих с шаблоном. Если шаблон состоит из нескольких групп, то элементы списка представляют собой кортежи	`re.findall(r"h[ie]\w", "hi hey hello")` `re.findall(r"(h|H)(i|e)", "Hey hello")`	`['hey', 'hel']` `[('H', 'e'), ('h', 'e')]`
`finditer`: возвращает итератор*, поставляющий объекты Match	`re.finditer(r"(h|H)(i|e)", "hi Hey hello")`	`Итератор`
`split`: разбивает строку по шаблону	`re.split(r"\d+", 'a1b2c3d4e')`	`['a', 'b', 'c', 'd', 'e']`
`sub`: создает строку с заменой совпадений другой строкой	`re.sub(r"\D", "-", '123,456_789')`	`'123-456-789'`

* Итератором называется объект, содержимое которого можно перебирать (например, в цикле `for`). Итераторы рассматриваются в главе 5.

Подчеркну ключевые моменты, касающиеся использования методов из табл. 2.3.

- Методы `search` и `match` определяют один объект `Match`. Главное различие заключается в том, что `match` привязывается к началу строки, а `search` сканирует строку, и совпадение в середине строки также считается допустимым.
- Когда вы пытаетесь найти все совпадения, метод `findall` возвращает все совпадения без информации об их местонахождении. Поэтому обычно используется `finditer`. Метод возвращает итератор, который генерирует

каждый объект `Match` с более содержательной информацией о совпадении (например, его местонахождение).

- Метод `split` разбивает строку по всем совпавшим шаблонам. Также можно задать максимальное количество возможных разбиений.
- Имя метода `sub` происходит от «substitute» (подстановка), и этот метод используется для замены любого найденного совпадения заданным заменителем. В нетривиальном варианте можно задать вместо строкового литерала функцию, которая получает в аргументе объект `Match` для производства требуемой замены.

2.4.6. Обсуждение

Ключевые этапы использования регулярных выражений: (1) создание шаблона, (2) поиск совпадений, (3) обработка совпадений. Эти этапы должны базироваться на четком понимании конкретных требований к производимой обработке текста. Попробуйте взглянуть на шаблон на более высоком уровне обобщения. Должен ли он содержать якоря, квантификаторы или символьные множества? Затем перейдите на уровень синтаксиса этих категорий. Будьте готовы к тому, что ваш шаблон не будет работать так, как вы ожидаете. Шаблон необходимо протестировать поиском совпадений на подмножестве текстов. Почти всегда обнаруживаются какие-нибудь граничные случаи, которые окажутся для вас неожиданными. Убедитесь в том, что эти редкие случаи учтены в шаблоне, прежде чем запускать свой код в продакшен.

2.4.7. Задача

Джерри — студент магистратуры. Один из его проектов требует извлечения данных из текста. Допустим, текстовые данные имеют вид `"abc_,abc__,abc,,__abc_,_abc"`, где `abc` — искомые значения данных. Иначе говоря, значения данных разделяются одним или несколькими разделителями. Как использовать регулярные выражения для извлечения данных?

> **ПОДСКАЗКА** Если вам понадобится создать шаблон с переменным количеством символов, не забывайте о квантификаторах.

2.5. КАК ИСПОЛЬЗОВАТЬ РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ ДЛЯ ОБРАБОТКИ ТЕКСТА

Регулярные выражения — не самая простая для понимания тема, потому что она связана с созданием общих шаблонов, которые могут подходить для множества разных ситуаций. В большинстве случаев шаблон выглядит абстрактно и приводит в замешательство неопытных программистов. Не огорчайтесь, если

что-то покажется вам непонятным; освоение регулярных выражений требует времени. Разобравшись в них, вы поймете, насколько это мощный инструмент для обработки текстовых данных.

В качестве примера возьмем приложение для управления задачами (таск-менеджер). Допустим, у вас имеется текст, приведенный в листинге 2.7. Этот текст содержит данные, восстановленные после сбоя базы данных, с множеством допустимых записей задач, но к сожалению, в данных местами встречается случайный текст.

Листинг 2.7. Текстовые данные для обработки

```
text_data = """101, Homework; Complete physics and math
some random nonsense
102, Laundry; Wash all the clothes today
54, random; record
103, Museum; All about Egypt
1234, random; record
Another random record"""  ◄——  Тройные кавычки для многострочных строк
```

Вам поручено извлечь все действительные записи из текстовых данных, а все недействительные удалить. Предположим, текст состоит из нескольких тысяч строк, так что обработать его вручную нереально. Для выполнения работы необходимо использовать общее решение с поиском по шаблону — именно для таких целей создавались регулярные выражения. В этом разделе будут рассмотрены ключевые этапы решения задачи.

2.5.1. Создание шаблона для поиска совпадений

Строка, приведенная в листинге 2.7, напоминает о важности стандартной операции при работе с текстами — *очистке данных*. Часто нужные данные в тексте чередуются с ненужными. Вам необходимо реализовать программное решение с использованием регулярных выражений, которое оставит только нужные данные. В этом разделе рассматривается первый шаг — создание шаблона.

Внимательно изучив необработанные данные, вы замечаете, что действительные записи состоят из трех групп: идентификатора задачи из трех цифр, названия задачи и описания задачи. Первые две группы разделяются запятой, а две последние — символом ; (точка с запятой). На основании этой информации можно построить следующий шаблон, в котором подробно описывается каждый компонент:

```
r"(\d{3}), (\w+); (.+)"

(\d{3}):   группа из 3 цифр
, :        строковые литералы, запятая и пробел
(\w+):     группа из одного или нескольких символов слова
; :        строковые литералы, точка с запятой и пробел
(.+):      группа из одного или нескольких символов слова
```

Применяя этот шаблон к текстовым данным, легко представить, как будет выглядеть результат. На этой стадии не беспокойтесь об обработке совпадений, потому что сейчас нужно просто убедиться в том, что шаблон работает как предполагалось. После того как вы несколько раз протестируете и измените шаблон, выполните следующий код, прежде чем остановиться на окончательной версии шаблона:


```
regex = re.compile(r"(\d{3}), (\w+); (.+)")
for line in text_data.split("\n"):
    match = regex.match(line)
    if match:
        print(f"{'Matched:':<12}{match.group()}")
    else:
        print(f"{'No Match:':<12}{line}")
```

```
# Выводимые строки:
Matched:    101, Homework; Complete physics and math
No Match:   some random nonsense
Matched:    102, Laundry; Wash all the clothes today
No Match:   54, random; record
Matched:    103, Museum; All about Egypt
No Match:   1234, random; record
No Match:   Another random record
```

Как упоминалось в разделе 2.4.4, важная особенность объекта `Match` заключается в том, что при его вычислении возвращается значение `True`; это позволяет нам работать с объектом `Match` только в том случае, если он был создан при вызове метода `match`. Вывод показывает, что из объектов совпадений можно извлечь нормальные записи. С другой стороны, там, где совпадения не найдены, записи действительно некорректны.

2.5.2. Извлечение данных из совпадений

Поскольку шаблон работает так, как предполагалось, пришло время извлечь данные и подготовить их к дальнейшей обработке. Выражаясь конкретнее, требуется сохранить каждую запись (идентификатор, название и описание) в объекте кортежа `tuple`, а затем объединить объекты `tuple` в объект `list`.

При построении шаблона вы определили три разные группы для каждого из трех полей данных задачи. Эти группы позволяют обращаться к отдельным совпадениям каждой группы. Использование групп продемонстрировано в листинге 2.8.

Как показано в листинге 2.8, мы используем метод `group` и обращаемся к трем найденным группам по номерам: группа 1 для идентификатора, группа 2 для названия, группа 3 для описания. Попутно заметим, что если параметр метода `group` не указан, будет получено все совпадение по всем группам (раздел 2.4.4).

Листинг 2.8. Извлечение данных из отдельных групп

```
regex = re.compile(r"(\d{3}), (\w+); (.+)")
tasks = []
for line in text_data.split("\n"):
    match = regex.match(line)
    if match:
        task = (match.group(1), match.group(2), match.group(3))
        tasks.append(task)

print(tasks)
# Выводимые строки:
[('101', 'Homework', 'Complete physics and math'),
➥ ('102', 'Laundry', 'Wash all the clothes today'),
➥ ('103', 'Museum', 'All about Egypt')]
```

Создает кортеж из нескольких групп

В нашем примере шаблон состоит из трех групп, но с усложнением структуры записей приходится иметь дело с большим количеством групп. Обращение к группам по номерам повышает риск ошибок; слишком легко ошибиться на единицу, что может привести к неожиданным последствиям.

Нет ли более удобного решения? Этот вопрос подводит нас к обсуждению в разделе 2.5.3.

2.5.3. Использование именованных групп для обработки текста

В общем случае текст предоставляет больше семантической информации, чем числа. Если обращения к группам по номерам создают путаницу, нельзя ли использовать текст для ссылок на группы? К счастью, в Python поддерживаются так называемые *именованные группы*. По сути, они позволяют назначить имя группе так, чтобы к ней можно было обращаться по имени при последующей обработке.

Для присвоения имен группам используется синтаксис `(?P<имя_группы>шаблон)`, назначающий группе шаблона `имя_группы`. Имя должно быть действительным идентификатором Python. После этого вы сможете воспользоваться функциональностью именованных групп для обновления кода из листинга 2.8, как показано в листинге 2.9.

Листинг 2.9. Использование именованных групп для извлечения данных

```
regex = re.compile(r"(?P<task_id>\d{3}), (?P<task_title>\w+);
     (?P<task_desc>.+)")
tasks = []
for line in text_data.split("\n"):
    match = regex.match(line)
    if match:
        task = (match.group('task_id'), match.group('task_title'),
        ➥  match.group('task_desc'))
        tasks.append(task)
```

В этом фрагменте кода трем группам присваиваются имена `task_id`, `task_title` и `task_desc`, ясно обозначающие данные каждой группы. Далее, вместо того чтобы передавать целое число методу `group`, мы указываем имя группы. По сравнению с реализацией из листинга 2.8, именованные группы из листинга 2.9 упрощают чтение кода; что еще более важно, они снижают риск случайного обращения не к той группе, особенно если шаблон содержит много групп.

> **СОПРОВОЖДАЕМОСТЬ** Всегда используйте содержательные идентификаторы для назначения имен переменных или любых объектов. Такой подход не только упрощает чтение кода, но и снижает количество возможных ошибок: чтобы понять, с какими данными вы имеете дело, достаточно взглянуть на их имена.

Хотя мы используем метод `group` для обращения к отдельным элементам из найденных групп, именованные группы предоставляют другой способ получения данных — метод `groupdict`. Для первого найденного совпадения результат может выглядеть так:

```
>>> match.groupdict()
{'task_id': '101', 'task_title': 'Homework', 'task_desc':
➥ 'Complete physics and math'}
```

Если вы предпочтете использовать этот объект `dict` для обработки данных, это улучшит удобочитаемость кода.

2.5.4. Обсуждение

Первым шагом при использовании регулярных выражений является определение того, чего вы хотите добиться, и создание соответствующего шаблона. Не стоит с одержимостью пытаться построить правильный шаблон с первой попытки. Шаблон необходимо опробовать на тексте; возможно, для нахождения правильного шаблона потребуется несколько циклов проб и ошибок (рис. 2.5).

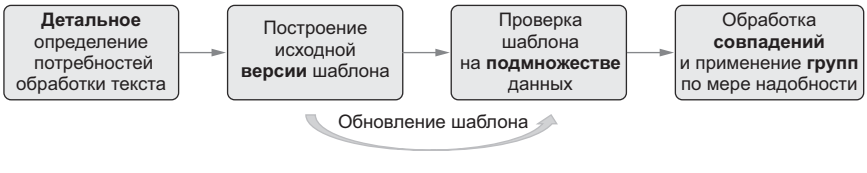


Рис. 2.5. Общая схема использования регулярных выражений при обработке текста

Если вы работаете с большим количеством групп, определенных в шаблоне, я рекомендую использовать именованные группы, так как при указании имени группы вы однозначно сообщаете читателю кода, какие данные в ней содержатся. Кроме того, в дальнейшем и вам будет удобнее обращаться к группам по содержательным именам.

2.5.5. Задача

При обработке текстовых данных с извлечением записей мы разбиваем текст на отдельные строки. Предположим, каждая строка либо содержит одну действительную запись, либо не содержит ни одной. Сможете ли вы найти шаблон, который обрабатывает весь текст без разбиения данных на множество строк?

> **ПОДСКАЗКА** Каждая строка завершается символом новой строки (`\n`). Включите этот символ в свой шаблон.

ИТОГИ

- F-строки предоставляют компактные средства интерполяции переменных и выражений.
- Выравнивание текста в f-строке упрощает восприятие информации за счет создания визуальных границ для разных данных.
- F-строки хорошо подходят для форматированных чисел, в частности для научной записи и для ограниченного количества знаков в дробной части.
- Для строк Python существуют методы `isalnum`, `isnumeric` и много других `is`-методов. Вы можете использовать их для определения природы содержимого строки.
- Все данные Python, включая целые числа и списки, могут представляться в строковом виде (например, при передаче по интернету данных в строковом формате). При преобразовании этих строк к исходному типу данных мы вычисляем их, чтобы в дальнейшем применить к результатам методы конкретных типов.
- Для объединения небольшого количества строк можно воспользоваться знаками конкатенации. Но при работе с множеством строк лучше выбрать метод `join`.
- Метод `split` предназначен для разбиения строк. Он является полезным средством обработки данных, а также основой для обработки текстовых файлов, содержащих данные с разделителями. Хотя для этих задач существуют встроенные модули (например, `csv`), хорошее знание основ сыграет важную роль при самостоятельном написании сценариев.
- Ключом к использованию регулярных выражений становится построение шаблона, соответствующего вашим потребностям. При построении шаблона следует мыслить на высоком уровне обобщения. Вот некоторые важные вопросы, на которые следует ответить: нужно ли использовать группы? А как насчет якорей, символьных множеств или квантификаторов?
- Именованные группы упрощают обращение к конкретной информации, когда регулярные выражения используются для обработки сложных текстовых данных.

3 Встроенные контейнеры данных

В этой главе

- ✓ Преимущества списков перед кортежами и кортежей перед списками
- ✓ Сортировка списков, состоящих из составных типов данных
- ✓ Использование именованных кортежей как модели контейнера данных
- ✓ Обращение к данным словаря
- ✓ Хешируемость и ее последствия для словарей и множеств
- ✓ Применение операций над множествами при работе с данными других типов

Python как язык программирования общего назначения предоставляет диапазон встроенных типов данных для разных целей, включая типы коллекций. Типы коллекций служат контейнерами для хранения целых чисел, строк, экземпляров пользовательских классов и других видов объектов. В любом проекте нам приходится работать с несколькими объектами одновременно, а в таких ситуациях часто возникает необходимость в контейнерах данных.

В каждом современном языке программирования контейнеры являются одной из базовых моделей данных, что подчеркивает их важность как структурных элементов для любого программного проекта. В главе 14 показано, как при построении таск-менеджера контейнеры данных будут использоваться для разных целей: списки `list`, например, для хранения экземпляров пользовательского класса `Task` (глава 8). В этой главе рассматриваются самые распространенные встроенные контейнеры данных, включая списки, кортежи, словари и множества. Я не пытаюсь привести здесь исчерпывающий обзор всей функциональности, относящейся к этим моделям данных. Мы сосредоточимся на фундаментальных темах, особенно важных для наших проектов.

ОСНОВНЫЕ ПОНЯТИЯ *Контейнеры данных*, например списки и кортежи, представляют собой объекты, которые содержат другие объекты. Строки и целые числа, наоборот, не являются контейнерами данных, так как они не содержат другие объекты.

3.1. КАК ВЫБРАТЬ МЕЖДУ СПИСКОМ И КОРТЕЖЕМ

Списки и кортежи часто рассматриваются вместе, что объясняется их сходством как контейнеров данных. И те и другие могут хранить упорядоченные наборы объектов, к которым можно обращаться по индексам. Во многих случаях они используются взаимозаменяемо, но в некоторых ситуациях приходится выбирать один из двух видов контейнеров. Допустим, вам нужен контейнер данных для хранения информации об операциях с банковским счетом. Что использовать — список или кортеж? Или другой пример: какой контейнер лучше подойдет для вывода информации об операции транзакции (скажем, суммы и даты) — список или кортеж?

Существует множество подобных сценариев, в которых оба варианта представляются возможными, но в конечном итоге мы выбираем один из них. В этом разделе рассматриваются ключевые различия между списками и кортежами, влияющие на выбор одного из контейнеров.

3.1.1. Кортежи для неизменяемых данных, списки — для изменяемых данных

Принципиальное различие между списками и кортежами — *изменяемость* (mutability). Списки являются изменяемыми в том смысле, что данные объекта `list` можно изменить: вы можете присоединить новые элементы к концу списка, вставить элементы в середину, изменить и удалить элементы. Для обеспечения изменяемости Python предоставляет набор методов в классе `list`: `append`, `extend`, `remove` и т. д., которые следует знать. Эти методы перечислены на рис. 3.1.

Рис. 3.1. Основные операции со списками как изменяемыми объектами

> **ОБРАТИТЕ ВНИМАНИЕ** Метод списков `remove` удаляет только первый подходящий элемент. Если удаляемый элемент отсутствует в списке, происходит ошибка `ValueError`.

В отличие от списков, кортежи неизменяемы; изменить данные объекта `tuple` невозможно. Чтобы обеспечить неизменяемость и предотвратить возможную путаницу, Python не предоставляет методов для изменения объектов `tuple`. Изменить элементы кортежа синтаксически возможно, но такое действие приводит к исключению: вызов несуществующего метода приводит к ошибке `AttributeError`, а повторное присваивание элементу кортежа — к ошибке `TypeError`, как показано в листинге 3.1.

Листинг 3.1. Неизменяемость объектов tuple

```
integers_tuple = (1, 2, 3)
integers_tuple.append(4)
# ERROR: AttributeError: 'tuple' object has no attribute 'append'

integers_tuple[0] = 'zero'
# ERROR: TypeError: 'tuple' object does not support item assignment
```

Пытается использовать несуществующий метод для объекта кортежа

Пытается присвоить новое значение элементу кортежа

Из-за разницы в изменяемости следует использовать списки, а не кортежи, если вы намерены обновлять данные. В таск-менеджере задачи хранятся в списках, потому что мы добавляем новые или удаляем старые задачи. Если хранимые данные не изменяются, лучше использовать неизменяемые кортежи. Метаданные задачи в таск-менеджере (например, время создания и пользователь) могут

храниться в кортежах, потому что эти метаданные остаются фиксированными. И хотя списки могут применяться везде, где применяются кортежи, есть ряд случаев, когда мы отдаем предпочтение кортежам перед списками:

- *Предотвращение неожиданных изменений* в данных. Попытка изменения данных кортежа приводит к ошибке `AttributeError` или `TypeError` (листинг 3.1).
- *Явное выражение намерения сохранить данные в неизменном виде*. Мы используем для хранения информации о задаче `(creation_time, user)` вместо `[creation_time, user]`, показывая, что эти два значения фиксируются в программе.
- *Более эффективное расходование памяти кортежами по сравнению со списками*. Если список и кортеж содержат одинаковые данные, список занимает больше памяти, чем кортеж. Большие затраты памяти для списков обусловлены необходимостью поддерживать изменяемость. Таким образом, в ситуациях, требующих хранения множества экземпляров, мы предпочитаем использовать кортежи ради более экономного расходования памяти.

  ОБРАТИТЕ ВНИМАНИЕ Чтобы узнать, сколько памяти расходуется для хранения объекта, вызовите `__sizeof__`.

3.1.2. Кортежи для разнородных, списки для однородных данных

В списках и кортежах могут храниться данные произвольных типов. Если элементы относятся к разным типам или все элементы содержат один тип, но с различающейся по смыслу информацией, они называются *разнородными* (heterogeneous) с точки зрения семантики. Возьмем объект из реального мира — допустим, коробку. Информация, относящаяся к коробке, может включать размер, материал и цвет. Такая информация является разнородной, так как она представляет разные аспекты характеристик коробки.

Если элементы относятся к одному типу — или, точнее говоря, данные относятся к информации одного типа, — они называются *однородными* (homorogeneous). Например, при переезде вещи могут быть упакованы по коробкам. Эти коробки являются однородными, потому что они представляют объекты одного вида.

В списках и кортежах могут содержаться как однородные, так и разнородные данные. Означает ли это, что они не отличаются друг от друга? Разумеется, главным отличительным признаком является требование изменяемости данных (см. раздел 3.1.1). Но если нам не нужно в первую очередь заботиться об изменяемости, то выбор должен определяться однородностью данных.

Рассмотрим более конкретный пример с таск-менеджером. В разделе 3.1.1 я упоминал о том, что с точки зрения изменяемости данных для хранения метаданных пользователя лучше использовать кортеж `(creation_time, user)`, так

как он содержит информацию о том, когда была создана задача и кто ее создал, в отдельных элементах. Иногда кортежи называются структурными, так как каждый элемент содержит независимую информацию, которая вносит свой вклад в объект `tuple`. Поэтому кортежи как структура данных предпочтительны для хранения семантически разнородных данных.

С другой стороны, данные, хранящиеся в списках, семантически однородны. В таск-менеджере задачи принадлежат одной семантической категории, следовательно, для хранения задач должны использоваться списки. По умолчанию задачи можно хранить упорядоченными в порядке возрастания времени создания. Как показано на рис. 3.2, список рассматривается как линейная структура данных для хранения однородных элементов.

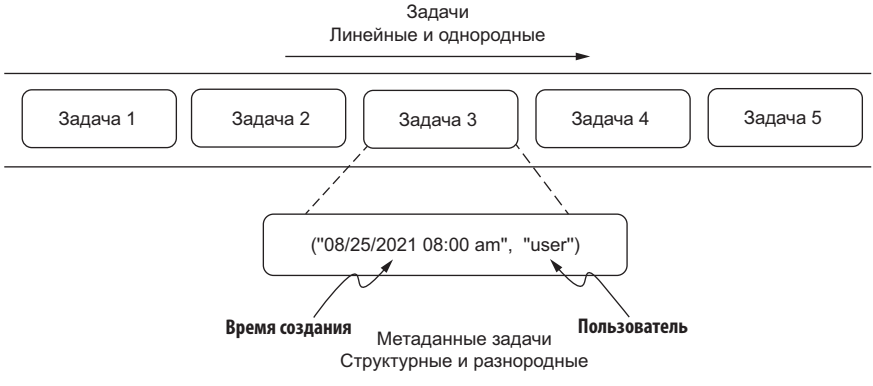


Рис. 3.2. Однородность элементов списка и разнородность элементов кортежа. Списки часто используются для хранения данных одного вида, такие данные называются *однородными*. На диаграмме список используется для хранения набора задач. Кортежи же часто используются для хранения данных с разным смыслом, такие данные называются *разнородными*. Как показано на диаграмме, кортеж используется для хранения метаданных задачи — фиксированных разнородных атрибутов

3.1.3. Обсуждение

С точки зрения удобочитаемости использование кортежей для хранения данных явно показывает читателю кода, что данные не изменяются. С точки зрения простоты сопровождения мы предпочитаем кортежи для предотвращения любых случайных изменений в данных, если они должны оставаться неизменными.

Следует заметить, что неизменяемость кортежей не препятствует изменению данных их элементов. Если кортеж содержит списки, например `numbers = ([1, 2], [1, 2])`, вы можете изменять внутренние списки — скажем, добавить элемент в первый список `list (numbers[0].append(3))`. Такая операция допустима; хотя

мы изменяем содержимое внутреннего объекта, ссылка на объект остается той же. Как показано в главе 10, следует различать сам объект и ссылку на него.

3.1.4. Задача

Зоя работает в компании, занимающейся разработкой картографических приложений. Она создает приложение, работающее с географическими точками. Для каждой точки хранится название, описание и координаты (широта и долгота). Какой контейнер использовать для хранения серии точек, посещенных пользователем, — список или кортеж? Для каждой точки ей также понадобится модель данных для хранения координат. Где лучше хранить широту и долготу — в списке или в кортеже?

> **ПОДСКАЗКА** Подумайте, являются ли хранимые данные изменяемыми и/или однородными, — это поможет вам в принятии решения.

3.2. КАК СОРТИРОВАТЬ СПИСКИ СЛОЖНЫХ ДАННЫХ С ПОМОЩЬЮ СПЕЦИАЛИЗИРОВАННЫХ ФУНКЦИЙ

Списки являются последовательностями (см. главу 4), в которых порядок определяется порядком вставки. Так как списки изменяемы, их содержимое часто выстраивается в порядке, отличном от порядка исходной вставки. Допустим, ваш проект содержит объект `list` с перечнем задач для заданного дня, как показано в листинге 3.2.

Листинг 3.2. Объект list, содержащий внутренние объекты dict

```
tasks = [
    {'title': 'Laundry', 'desc': 'Wash clothes', 'urgency': 3},
    {'title': 'Homework', 'desc': 'Physics + Math', 'urgency': 5},
    {'title': 'Museum', 'desc': 'Egyptian things', 'urgency': 2}
]
```

Предположим, что задачи выводятся с упорядочением по времени создания или степени срочности. Как вы вскоре узнаете, при сортировке таких списков словарей происходит ошибка `TypeError`, потому что Python не умеет сравнивать словари:

```
tasks.sort()
# ERROR: TypeError: '<' not supported between instances of 'dict' and 'dict'
```

В этом разделе вы научитесь сортировать списки, в том числе состоящие из сложных данных (например, словари вместо целых чисел и строк) со специальными (кастомными) требованиями.

3.2.1. Сортировка списков в порядке по умолчанию

Так как сортировка списков является достаточно распространенной задачей, в Python существует встроенное средство для ее выполнения — метод `sort`. В следующем листинге представлены простые примеры использования `sort`.

Листинг 3.3. Сортировка списков методом sort

```
numbers = [12, 4, 1, 3, 7, 5, 9, 8]
numbers.sort()  <-- Сортирует числа на месте
print(numbers)
# Вывод: [1, 3, 4, 5, 7, 8, 9, 12]

names = ['Danny', 'Aaron', 'Zack', 'Jennifer', 'Mike', 'David']
names.sort(reverse=True)  <-- Сортирует строки на месте, но в обратном порядке
print(names)
# Вывод: ['Zack', 'Mike', 'Jennifer', 'David', 'Danny', 'Aaron']

mixed = [3, 1, 2, 'John',  ['c', 'd'], ['a', 'b']]
mixed.sort()
# ERROR: TypeError: '<' not supported between instances of 'str' and 'int'
```

Обратите внимание: сортировка выполняется на месте (in place); это означает, что сортировка изменяет порядок исходного списка вместо того, чтобы создавать новый список. Так как сортировка выполняется на месте, `sort` возвращает `None`. Таким образом, в интерактивной консоли Python после выполнения `numbers.sort()` вы не увидите никакого вывода, потому что `None` автоматически опускается в консольном выводе. Следует обратить внимание и на то, что по умолчанию используется сортировка по возрастанию. Если передать в параметре `reverse` значение `True`, то список будет отсортирован по убыванию.

> **ОСНОВНЫЕ ПОНЯТИЯ** Когда мы говорим, что какие-то операции выполняются с объектом «на месте» (in place), это означает, что процесс изменяет сам объект. Метод `sort` изменяет объект `list` на месте.

Похоже, Python не умеет сортировать списки с разными типами данных. В листинге 3.3 для списка, содержащего целые числа, строки и списки, была выдана ошибка `TypeError`, потому что по умолчанию Python не умеет сравнивать объекты разных типов. Можно ли как-то приказать Python сравнить эти объекты? Ответ приводится в разделе 3.2.2.

3.2.2. Использование встроенной функции как ключа сортировки

Кроме `reverse`, метод `sort` получает параметр `key`. Как следует из его имени, этот параметр предоставляет ключ для сортировки. Точнее говоря, `key` должна быть присвоена функция, которая генерирует значение ключа для каждого элемента списка. Сгенерированные значения используются для сравнения, и их порядок определяет порядок элементов списка.

ОБРАТИТЕ ВНИМАНИЕ Параметр `key` получает не только метод `sort`. У ряда других функций, таких как `max` и `min`, тоже имеется параметр `key`. То, что вы узнаете в этом разделе, применимо и к этим функциям.

Как упоминалось в конце раздела 3.2.1, Python не умеет сравнивать целые числа, строки и списки. Примечательно, что сравнивать строки Python умеет. Таким образом, стратегия сортировки данных разных типов может быть основана на преобразовании их в строки с указанием параметра `key`:

```
mixed = [3, 1, 2, 'John',  ['c', 'd'], ['a', 'b']]
mixed.sort(key=str)

print(mixed)
# Вывод: [1, 2, 3, 'John', ['a', 'b'], ['c', 'd']]
```

В коде в качестве аргумента `key` используется функция `str` (строго говоря, это конструктор класса; см. раздел 10.5), преобразующая каждый элемент в строку. Python сортирует эти строки `['3', '1', '2', 'John', "['c', 'd']", "['a', 'b']"]`, получая в результате `['1', '2', '3', 'John', "['a', 'b']", "['c', 'd']"]`. Каждая преобразованная строка ассоциируется со своим исходным объектом, и Python выводит отсортированный список.

3.2.3. Использование нестандартных функций для более сложных задач сортировки

В разделе 3.2.2 обсуждается, как использовать `key` для сортировки списка объектов разных типов, но этот пример слишком тривиален для использования в реальном проекте. В листинге 3.2 наш таск-менеджер содержит объект `list`, состоящий из объектов `dict`. В этом разделе вы научитесь сортировать объекты `list` такого рода.

Хотя мы можем присвоить `str` аргументу `key`, чтобы эти объекты `dict` можно было сравнивать, отсортированный список получается не таким, как нужно: объекты не упорядочиваются по степеням срочности. Чтобы решить эту проблему, можно создать специальную функцию и присвоить ее аргументу `key`, как показано в листинге 3.4.

Листинг 3.4. Сортировка задач с назначением аргумента key

```
def using_urgency_level(task):
    return task['urgency']

tasks.sort(key=using_urgency_level, reverse=True)
print(tasks)

# Выводимые строки (порядок изменен для удобства чтения):
[{'title': 'Homework', 'desc': 'Physics + Math', 'urgency': 5},
{'title': 'Laundry', 'desc': 'Wash clothes', 'urgency': 3},
{'title': 'Museum', 'desc': 'Egyptian things', 'urgency': 2}]
```

Каждый элемент списка передается функции `using_urgency_level`. Важно отметить, что функция `key` должна получать ровно один параметр, который соответствует каждому элементу объекта `list`. Функция извлекает степени срочности задач в соответствии с правилами выполняемой сортировки. На рис. 3.3 изображена высокоуровневая схема процесса сортировки.

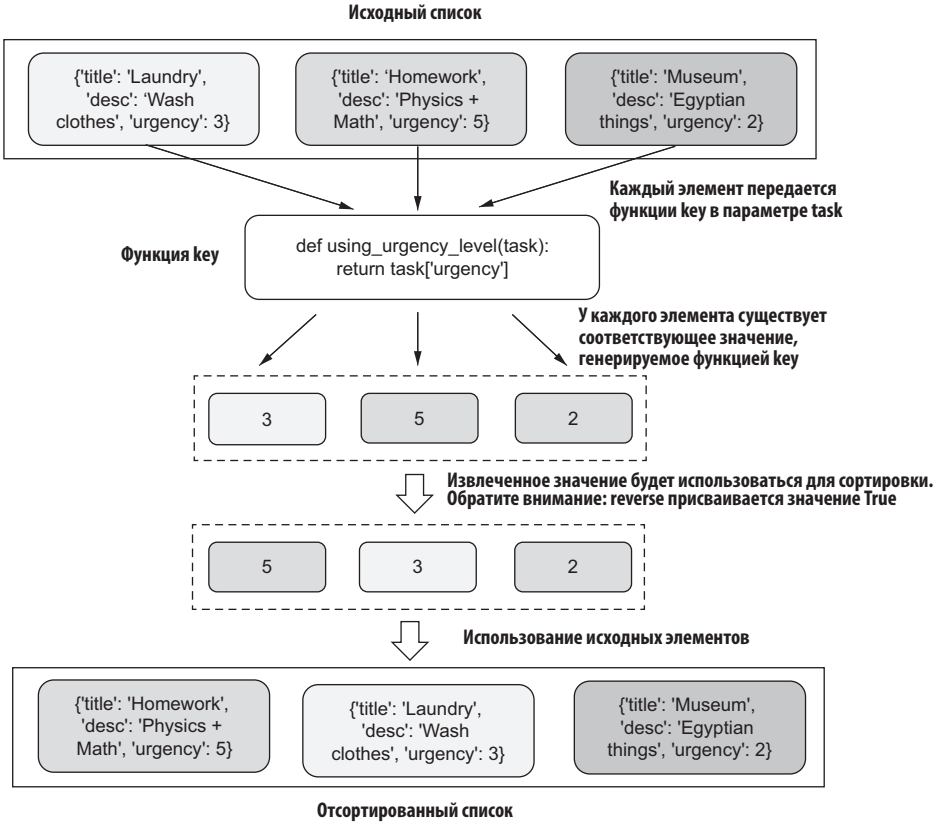


Рис. 3.3. В процессе сортировки используется функция key. Функция key преобразует каждый элемент списка в соответствующее значение. Генерируемые значения будут использоваться как промежуточные ключи для сортировки списка. После сортировки исходные элементы выводятся в порядке, определяемом промежуточными ключами

ЗАБЕГАЯ ВПЕРЕД Значение `key` можно задать в виде лямбда-функции — анонимной функции, которая создается при помощи ключевого слова `lambda`. Чтобы получить тот же результат сортировки с лямбда-функцией, можно использовать вызов `tasks.sort(key=lambda x: x['urgency'], reverse=True)`. Лямбда-функции рассматриваются в разделе 7.1.

3.2.4. Обсуждение

Метод `sort` работает только со списками, потому что он является методом экземпляров списков. При сортировке других типов данных контейнеров, таких как кортежи, множества и словари, нужно использовать метод `sorted`, который получает произвольный итерируемый объект и возвращает отсортированный список. Для `sorted` также можно назначить пользовательскую функцию сортировки. Помните, что функция, назначаемая аргументу `key`, должна получать ровно один параметр. Если функция аргумента `key` выполняет несложную операцию, рассмотрите возможность использования лямбда-функции (см. раздел 7.1).

3.2.5. Задача

В этой главе вы научились сортировать задачи по степеням срочности, как показано в листинге 3.4. Сможете ли вы построить решение для упорядочения задач по длине описания? Чем длиннее описание, тем выше ранг задачи.

ПОДСКАЗКА Нестандартная сортировка требует использования параметра `key` функции `sort`. Встроенная функция `len` может проверить длину строки.

3.3. КАК ПОСТРОИТЬ ОБЛЕГЧЕННУЮ МОДЕЛЬ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ИМЕНОВАННЫХ КОРТЕЖЕЙ

В любом проекте центральное место занимают данные. Если вы разрабатываете приложение для социальных сетей, данными становятся сведения о пользователях и их подключениях. Если вы строите веб-сайт электронной коммерции, данными будет информация о товарах и клиентах. Если вы создаете модель машинного обучения, данные состоят из признаков и целей. В нашем таск-менеджере необходимо построить механизм обработки данных, относящихся к задачам.

При наличии опыта объектно-ориентированного программирования (ООП) вашим решением на интуитивном уровне будет, вероятно, создание специальных классов для управления данными. Но написание класса — нетривиальная задача (лучшие практики создания классов описаны в главе 8). Возможно, с ростом сложности приложений вам придется создать несколько классов для обработки разных аспектов потока данных. Для более простых моделей данных именованные кортежи представляются идеальным решением, особенно если вы стремитесь в первую очередь к созданию облегченной модели, доступной в использовании и способной хранить данные с небольшой нагрузкой на память.

3.3.1. Альтернативные модели данных

Прежде чем переходить к построению модели данных с использованием именованных кортежей, важно ознакомиться со всеми доступными нам возможностями. В этом разделе будут рассмотрены по меньшей мере четыре средства управления данными: списки, кортежи, словари и пользовательские классы.

Обращаясь к контексту нашего приложения, допустим, что каждая задача в нем описывается несколькими информационными фрагментами, которыми вам необходимо управлять: названием, описанием и степенью срочности. Листинг 3.5 показывает, как выглядят модели данных на базе `list`, `tuple` и `dict`.

Листинг 3.5. Использование встроенных моделей данных для управления данными

```
task_list = ['Laundry', 'Wash clothes', 3]  <-- Использует список

task_tuple = ('Laundry', 'Wash clothes', 3)  <-- Использует кортеж

task_dict = {'title': 'Laundry', 'desc': 'Wash clothes', 'urgency': 3}
                                                  Использует словарь
```

Как показано в листинге 3.5, эти фрагменты информации хранятся в отдельных элементах в `list` и `tuple` и в парах «ключ — значение» в `dict`. Помимо использования встроенных классов, мы можем создать собственный класс для хранения данных. Заготовка пользовательского класса приведена в следующем фрагменте кода (не беспокойтесь, если вы еще не знакомы с определением пользовательских классов; эта тема рассматривается в главе 8):

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency

task_class = Task('Laundry', 'Wash clothes', 3)
```

И хотя каждый из этих подходов допустим в некоторых ситуациях, различные недостатки делают их не самым лучшим решением для нашей бизнес-цели — облегченной модели хранения данных.

Списки изменяемы, из-за чего они подвержены намеренным и случайным модификациям. В разделе 3.1 также указано, что списки обычно используются для хранения однородных данных; применять их для хранения разнородных данных не рекомендуется. Хотя кортежи неизменяемы и мы можем не беспокоиться о модификациях данных, для получения атрибута (скажем, названия) приходится прибегать к распаковке (раздел 4.4) либо индексированию (раздел 4.2). Ни один из этих способов не отличается прямолинейностью.

Списки и кортежи не предоставляют метаинформацию о хранящихся в них данных. Коллега, не знакомый с приложением, не получит никакой информации о модели данных при просмотре вашего кода. По сравнению со списками и кортежами словари предоставляют метаинформацию, так как ключи сообщают смысл данных. Но для получения этих атрибутов необходимо использовать соответствующие ключи (например, `task_dict['title']`). Если вы допустите ошибку в тексте ключа или пропустите кавычку, произойдет ошибка `KeyError` или `SyntaxError`.

Списки, кортежи и словари являются общими типами, и они не связаны со знаниями о специфике модели данных. Таким образом, современные среды интегрированной разработки (IDE), такие как PyCharm и Visual Studio Code, не предоставляют полезных подсказок автозаполнения для таких структур данных, что снижает эффективность программирования. Этот недостаток можно преодолеть созданием пользовательского класса. При создании экземпляра `Task` после ввода экземпляра и точки ваша IDE автоматически откроет список доступных атрибутов (таких, как `title` и `desc`). Автозаполнение ускоряет написание кода.

> **ОСНОВНЫЕ ПОНЯТИЯ** IDE предоставляет разностороннюю функциональность (например, подсказки автозаполнения и анализ кода в реальном времени), облегчая разработку программного обеспечения.

Впрочем, решение с разработкой пользовательского класса тоже может создать ряд затруднений:

- разработка пользовательского класса потребует значительного объема шаблонного кода, а для такой простой модели данных реализация полноценного пользовательского класса — это уже перебор;
- затратами памяти не стоит пренебрегать, особенно если вам приходится иметь дело со множеством экземпляров.

Каждый экземпляр пользовательского класса расходует больше памяти, чем экземпляр именованного кортежа, как будет сказано в разделе 3.3.2. По мере развития проекта также должна расширяться функциональность модели данных; мы переместим облегченную модель данных в полноценный пользовательский класс (глава 8).

3.3.2. Создание именованных кортежей для хранения данных

Как следует из названия, именованный кортеж (named tuple) является одним из видов кортежей. Именованные кортежи отличаются тем, что с их элементами связываются имена. В отличие от обычных кортежей, к элементам которых вы обращаетесь по индексам, именованные кортежи поддерживают точечную запись

(dot notation), то есть механизм обращения к элементам аналогичен обращениям к экземплярам пользовательских классов. Следующий пример демонстрирует эту возможность:

```
from collections import namedtuple
Task = namedtuple('Task', 'title desc urgency')
task_nt = Task('Laundry', 'Wash clothes', 3)
assert task_nt.title == 'Laundry'
assert task_nt.desc == 'Wash clothes'
```

Создает класс именованного кортежа

Создает экземпляр именованного кортежа

Обращается к атрибутам экземпляра

Обратите внимание на некоторые важные особенности именованных кортежей.

- Экземпляр именованного кортежа позволяет обращаться к своим атрибутам с помощью точечной записи. Решение не только быстрее программируется благодаря подсказкам автозавершения, но и лучше читается из-за наглядной схемы обращения к атрибутам.
- `Namedtuple` — фабричная функция в модуле `collections`. Так как это фабричная функция, при ее вызове возвращается новый класс или объект нового экземпляра. В данном случае вы получаете класс `Task`.
- В функции `namedtuple` мы указываем имя класса и его атрибуты для класса. Следует заметить, что атрибуты модели данных могут задаваться либо в виде отдельной строки (с пробелами или запятыми в качестве разделителей), либо в виде объекта `list` (рис. 3.4).

```
Task = namedtuple('Task', 'title, desc, urgency')

Task = namedtuple('Task', ['title', 'desc', 'urgency'])
```

УДОБОЧИТАЕМОСТЬ Соблюдайте соглашения об именах классов в Python и используйте форму «верблюжьего регистра»: `ИмяКласса`. Если имя состоит из нескольких слов, первая буква каждого слова должна быть прописной (например, `TaskUser`).

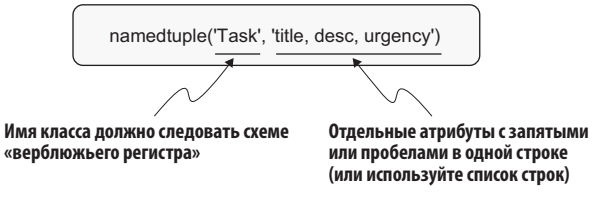


Рис. 3.4. Создание именованного кортежа. Имя класса должно следовать схеме «верблюжьего регистра», а атрибуты должны задаваться одной строкой или списком строк

УДОБОЧИТАЕМОСТЬ Атрибуты задаются одной строкой с пробелами или запятыми. Такой код проще пишется и читается.

Теперь вы знакомы с именованными кортежами и можете использовать класс `Task` для обработки данных, используемых в приложении. Для простоты допустим, что источником данных является объект строки, полученный через программный интерфейс (API):

```
task_data = '''Laundry,Wash clothes,3
Homework,Physics + Math,5
Museum,Epyptian things,2'''
```

ОСНОВНЫЕ ПОНЯТИЯ API определяет набор средств построения и интеграции разных компонентов — как программных, так и аппаратных. Термином API часто обозначаются различные функции, которые могут вызываться вашим приложением для получения данных из другого источника.

Возможное решение для преобразования текстовых данных в экземпляры `Task` выглядит так:

```
for task_text in task_data.split('\n'):
    title, desc, urgency = task_text.split(',')
    task_nt = Task(title, desc, int(urgency))
    print(f"--> {task_nt}")
```

Разбивает текстовые данные на строки

Разбивает текстовые данные по запятым

```
# output the following lines
--> Task(title='Laundry', desc='Wash clothes', urgency=3)
--> Task(title='Homework', desc='Physics + Math', urgency=5)
--> Task(title='Museum', desc='Epyptian things', urgency=2)
```

В этом решении используются некоторые описанные выше приемы, включая разбиение строк и f-строки. Кроме того, оно показывает, что небольшие улучшения в сумме сказываются на работоспособности целого. На следующем шаге можно воспользоваться методом `_make` класса именованного кортежа, который отображает итерируемый объект (список, создаваемый `split`, является итерируемым объектом; итерируемые объекты подробно рассматриваются в главе 5) на именованный кортеж. Обновленное решение выглядит так:

```
for task_text in task_data.split('\n'):
    task_nt = Task._make(task_text.split(','))
```

ЗАБЕГАЯ ВПЕРЕД Методы классов рассматриваются в разделе 8.2.

В отличие от пользовательских классов, экземпляры которых содержат представления `dict` в виде `__dict__`, именованные кортежи не имеют встроенных представлений `dict`, благодаря чему именованные кортежи предоставляют облегченную модель данных с минимальными затратами памяти. Именованные кортежи способны обеспечить значительную экономию памяти, если вам требуется создать тысячи экземпляров.

Любознательным читателям рекомендую обращаться на официальный веб-сайт Python (https://docs.python.org/3/library/collections.html) за информацией о других

возможностях именованных кортежей, например о создании нового именованного кортежа на базе существующего с заменой значений полей и проверкой значений полей по умолчанию.

3.3.3. Обсуждение

По сравнению со встроенными типами (такими, как списки, кортежи и словари) и пользовательскими классами именованные кортежи предоставляют более общую облегченную модель для хранения данных, в основном ориентированную на обращение только для чтения. Например, `pandas` — популярная библиотека data science для Python — позволяет обратиться к каждой строке модели данных `DataFrame` в виде именованного кортежа.

> **ОБРАТИТЕ ВНИМАНИЕ** Многие аналитики данных используют pandas в своей повседневной работе при обработке данных. Ключевая структура данных библиотеки `DataFrame` представляет данные в форме электронной таблицы.

Так как именованные кортежи представляют новый тип, используйте для них содержательное имя, начинающееся с прописной буквы, как в других пользовательских классах. Сделайте так, чтобы класс именованного кортежа был по возможности очевиден. Код создания класса именованного кортежа рекомендуется размещать в начале модуля. В конце концов, код состоит всего из одной строки, и он не должен затеряться.

> **СОПРОВОЖДАЕМОСТЬ** Разместите код создания класса именованного кортежа в заметном месте, например в начале модуля. Код занимает всего одну строку, но он очень важен: в нем создается новый класс.

3.3.4. Задача

Допустим, в таск-менеджере требуется обновить именованный кортеж `Task(title='Laundry', desc='Wash clothes', urgency=3)`, установив для него степень срочности (`urgency`) равной `4`. Можно ли изменить степень напрямую? Если нет, то как это можно сделать?

> **ПОДСКАЗКА** Именованный кортеж представляет собой объект кортежа, поэтому он неизменяем и прямое изменение хранимых в нем данных недопустимо.

3.4. КАК ОБРАЩАТЬСЯ К КЛЮЧАМ, ЗНАЧЕНИЯМ И ЭЛЕМЕНТАМ СЛОВАРЕЙ

Наиболее часто используемыми типами данных являются `int`, `float`, `bool`, `str`, `list`, `tuple`, `set` и `dict`. Первые четыре типа называются примитивными, потому что они становятся структурными элементами других типов данных. Остальные

четыре типа являются контейнерами данных (рис. 3.5). Тип `dict` отличается от `list`, `tuple` и `set` прежде всего тем, что он содержит пары «ключ — значение» вместо отдельных объектов. Использование пар «ключ — значение» означает, что в словаре хранятся две категории информации.

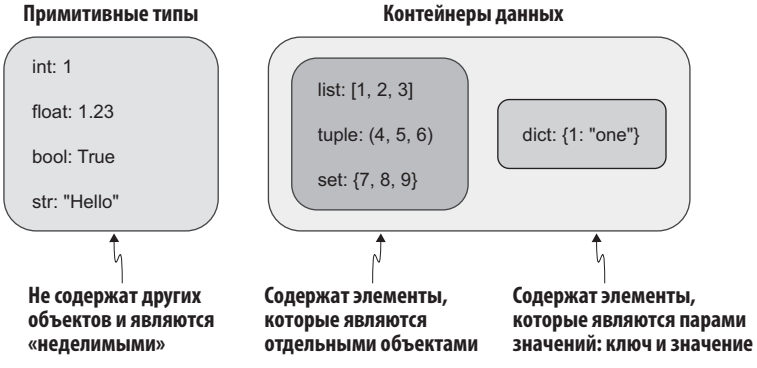


Рис. 3.5. Стандартные модели данных в Python, включая примитивные типы и контейнеры данных

Допустим, у вас есть словарь для хранения степеней срочности некоторых задач в таск-менеджере. Объект `dict` содержит две группы информации: названия (ключи) и степени сложности (значения):

```
urgencies = {"Laundry": 3, "Homework": 5, "Museum": 2}
```

При включении словарей в проект часто возникает необходимость в обращении к хранимым данным — ключам, значениям и парам «ключ — значение». В этом разделе мы исследуем различные способы обращения к данным. Так как словари часто используются в реальных проектах, умение обращаться к данным словаря исключительно важно для использования этого эффективного типа данных.

3.4.1. Прямое использование объектов динамических представлений (keys, values и items)

Помимо обращений к отдельным парам «ключ — значение» в словаре (например, `urgencies["Laundry"]`), Python предоставляет три базовых метода для получения данных, хранимых в словаре во всех парах: `keys`, `values` и `items` для обращения к ключам, значениям и парам «ключ — значение» соответственно. Рассмотрим простые примеры их использования:

```
urgencies = {"Laundry": 3, "Homework": 5, "Museum": 2}
urgen_keys = urgencies.keys()
urgen_values = urgencies.values()
urgen_items = urgencies.items()
print(urgen_keys, urgen_values, urgen_items, sep="\n")
```

```
# Выводимые строки:
dict_keys(['Laundry', 'Homework', 'Museum'])
dict_values([3, 5, 2])
dict_items([('Laundry', 3), ('Homework', 5), ('Museum', 2)])
```

Часто разработчики предполагают, что создаваемые этими методами объекты (`keys`, `values` и `items`) являются объектами `list`. Но это не так. Создаются объекты `dict_keys`, `dict_values` и `dict_items` соответственно. У этих типов данных есть одна особенность: все они являются *объектами динамических представлений* (dynamic view objects). Если вы знакомы с терминологией баз данных, вероятно, вы слышали о *представлениях* (views) — виртуальных результатах, вычисляемых или собираемых динамически по данным из базы.

> **ОБРАТИТЕ ВНИМАНИЕ** Представления — результаты запросов, хранимых в базе данных. При обновлении соответствующих данных представления также обновляются.

Объекты представлений словарей, как и представления в базе данных, являются динамическими, они автоматически обновляются с изменением объекта `dict`. Таким образом, каждый раз, когда вы изменяете хранящиеся в объекте `dict` пары «ключ — значение», эти объекты представлений тоже обновляются. Проследите за этим эффектом:

```
urgencies["Grocery Shopping"] = 4

print(urgen_keys)
# Вывод: dict_keys(['Laundry', 'Homework', 'Museum', 'Grocery])

print(urgen_values)
# Вывод: dict_values([3, 5, 2, 4])

print(urgen_items)
# Вывод: dict_items([('Laundry', 3), ('Homework', 5), ('Museum', 2),
➥  ('Grocery, 4)])
```

Такая динамичность чрезвычайно удобна при обращении к данным словаря, потому что данные идеально синхронизируются с объектом `dict`. С другой стороны, следующий пример, в котором объект `view` не используется, является антипаттерном:

```
urgencies = {"Laundry": 3, "Homework": 5, "Museum": 2}

urgen_keys_list = list(urgencies.keys())
print(urgen_keys_list)
# Вывод: ['Laundry', 'Homework', 'Museum']

urgencies["Grocery"] = 4
print(urgen_keys_list)
# Вывод: ['Laundry', 'Homework', 'Museum']
```

Для ключей создается список `list`. После обновления словаря `list` остается тем же и не синхронизируется с объектом `dict`. Поэтому вы можете столкнуться

с неожиданными ошибками (например, попытками обращения к удаленным элементам), если будете использовать для отслеживания ключей словаря список вместо объекта представления `dict_keys`.

> **СОПРОВОЖДАЕМОСТЬ** Всегда используйте объекты `view` для обращения к данным `dict`, потому что объекты представлений являются динамическими — они автоматически обновляются при обновлении данных словаря.

3.4.2. Будьте осторожны с исключением KeyError

В разделе 3.4.1 описаны три способа обращения ко всем ключам и/или значениям в словаре. Однако в большинстве случаев требуется обратиться к одному значению в индексной записи, в которой ключ записывается в квадратных скобках:

```
assert urgencies["Laundry"] == 3
assert urgencies["Homework"] == 5
```

> **ОСНОВНЫЕ ПОНЯТИЯ** Индексная запись (subscript notation) — стандартный механизм обращения к данным в типах данных коллекций. При работе с объектами `dict` при индексной записи ключи для обращения к соответствующим значениям указываются в квадратных скобках.

Главное преимущество этого метода — прямолинейность. Если вы работали со словарями в других языках, такой подход вам, возможно, уже знаком. Тогда для вас будет естественно использовать эту возможность при обращении к элементам словаря. Но если вы не будете внимательны с ключом, могут произойти непредвиденные ошибки, что продемонстрировано в следующем фрагменте:

```
urgencies["Homeworks"]
# ERROR: KeyError: 'Homeworks'
```

При обращении к ключу, отсутствующему в словаре, происходит исключение `KeyError`. Если в программе происходит такое исключение, которое не будет обработано в инструкции `try...except...` (раздел 12.3), программа аварийно завершается. Конечно, никто не хочет, чтобы его программы аварийно завершались, поэтому для предотвращения ошибки стоит познакомиться с другими решениями.

3.4.3. Предотвращение KeyError с предварительной проверкой: непитонический способ

Так как мы знаем, что исключения `KeyError` происходят только при отсутствии ключа в объекте словаря, проверим существование ключа перед получением связанного с ним значения, как в следующем примере:

```
if "Homework" in urgencies:  ◄—— Проверяет, присутствует ли ключ в словаре
    urgency = urgencies["Homework"]
else:
    urgency = "N/A"
```

Такое решение помогает предотвратить исключение `KeyError`, но оно получается громоздким и непитоническим, потому что питонический код должен быть компактным. Здесь мы обращаемся только к одному элементу. А если мы будем обращаться к нескольким элементам? Этот блок кода придется повторять, что приведет к отвлекающему дублированию в кодовой базе. Дублирование должно напомнить о принципе DRY — следует провести рефакторинг кода, чтобы устранить ненужные повторения. Рассмотрим следующий код:

```
def retrieve_urgency(task_title):
    if task_title in urgencies:
        urgency = urgencies[task_title]
    else:
        urgency = "N/A"
    return urgency
```

После проведения рефакторинга можно получить степень срочности задачи, не переживая из-за возможности исключения `KeyError`:

```
retrieve_urgency("Homework")
# Вывод: 5

retrieve_urgency("Homeworks")
# Вывод: 'N/A'
```

Функция `retrieve_urgency` хорошо подходит для получения степени срочности задачи, но она жестко запрограммирована, включая объект `dict` (`urgencies`) и конкретную семантику (`urgency`). Если вам потребуется обратиться к данным другого объекта `dict`, необходимо определить аналогичную функцию для предотвращения `KeyError`.

Чем больше объектов словарей, тем больше функций придется создать. Не замечаете ли вы повторения на более высоком уровне? Первопроходцы Python уже рассмотрели эту проблему и создали встроенную функцию — метод `get`, описанный в разделе 3.4.4.

3.4.4. Использование метода get для обращения к элементу словаря

Так как `get` является методом класса `dict`, его можно вызвать для любого объекта `dict`; при этом следует указать ключ и значение по умолчанию, если ключ не существует. Если аргумент по умолчанию не задан, Python использует `None`. Некоторые примеры представлены в следующем фрагменте:

```
urgencies.get("Homework")
# Вывод: 5

urgencies.get("Homeworks", "N/A")
# Вывод: 'N/A'

urgencies.get("Homeworks")
# Вывод: None (автоматически скрывается в интерактивной консоли)
```

Преимущество метода `get` заключается в том, что он не выдает ошибку `KeyError` при отсутствии ключа в словаре. Еще важнее то, что он позволяет задать значение по умолчанию. Вы можете использовать `get` при получении значений из словарей, но я предпочитаю индексную запись, которая мне кажется более понятной.

Впрочем, в некоторых ситуациях вызов `get` предпочтительнее индексной записи. Одна из таких ситуаций — необходимость обработки переменного числа ключевых аргументов (`**kwargs`) в определении функции. Применение `**kwargs` будет рассматриваться в разделе 6.4, а пока достаточно запомнить, что `kwargs` — объект `dict`, используемый в функции, и что эти параметры обычно не являются обязательными. Допустим, вы строите пакет Python для сообщества Python, и в этот пакет входит следующая функция:

```
def calculate_something(arg0, arg1, **kwargs):
    kwarg0 = kwargs.get("kwarg0", 0)
    kwarg1 = kwargs.get("kwarg1", "normal")
    kwarg2 = kwargs.get("kwarg2", [])
    kwarg3 = kwargs.get("kwarg3", "text")
    # ... и т.д.

# Возможные вызовы:
calculate_something(arg0, arg1)
calculate_something(arg0, arg1, kwarg0=5)
calculate_something(arg0, arg1, kwarg0=5, kwarg3="text")
```

В этом примере `calculate_something` получает несколько ключевых аргументов помимо двух позиционных. Чтобы код был более компактным, вам, скорее всего, не захочется перечислять все необязательные ключевые аргументы, если в программе почти всегда используются их значения по умолчанию, — их можно упаковать в словарь `kwargs` в заголовке функции. Вы заметите, что в теле функции `get` используется несколько раз; это позволяет задать значения по умолчанию, если при вызове функции ключи отсутствуют. Соответствующие значения по умолчанию включаются в метод `get`.

3.4.5. Побочный эффект метода setdefault

В качестве альтернативы для метода `get` иногда предлагается метод `setdefault`. Как и `get`, этот метод получает два параметра: ключ и значение по умолчанию. Несколько примеров использования `setdefault`:

```
urgencies = {"Laundry": 3, "Homework": 5, "Museum": 2}
urgencies.setdefault("Homework")
# Вывод: 5

urgencies.setdefault("Homeworks", 0)
# Вывод: 0

urgencies.setdefault("Grocery")
# Вывод: None (автоматически скрывается в интерактивной консоли)
```

Этот фрагмент кода демонстрирует сходство между `setdefault` и `get`. Однако между ними есть одно важное различие: если при вызове `setdefault` ключ отсутствует в словаре, выполняется дополнительная операция (`dict[key] = default_value`):

```
print(urgencies)
# Вывод: {'Laundry': 3, 'Homework': 5, 'Museum': 2, 'Homeworks': 0,
➥  'Grocery': None}
```

Ранее мы вызывали `setdefault` с ключами `"Homework"`, `"Homeworks"` и `"Grocery"`. Так как два последних ключа не присутствовали в `dict` изначально, при этом неявно были выполнены следующие операции:

```
urgencies["Homeworks"] = 0
urgencies["Grocery"] = None
```

Из-за этого побочного эффекта я не рекомендую использовать метод `setdefault`. Его имя создает путаницу (как правило, мы не ожидаем, что при вызове метода, который что-то присваивает, будет возвращено значение), и при вызове выполняется неявная операция (присваивание заданного значения по умолчанию или `None`, если ключ не существует), о которой многие разработчики не знают.

> **СОПРОВОЖДАЕМОСТЬ** Избегайте использования метода `setdefault`, потому что он может задать значение для отсутствующего ключа неожиданным образом. Используйте более явное решение — метод `get`.

3.4.6. Обсуждение

Объекты представлений словаря — превосходно спроектированное решение, динамически отслеживающее ключи, значения и пары «ключ — значение» словаря. Как итерируемые объекты, они могут использоваться в циклах `for` (раздел 5.3), если вам потребуется перебрать данные объекта `dict`.

Вы не обязаны использовать `get` каждый раз, когда вам надо обратиться к связанному с ключом значению. Если вы привыкли к индексной записи, ничто не мешает вам использовать ее. Иногда стоит прибегать именно к ней, потому что любые ошибки должны обнаруживаться в ходе разработки, а выдача ошибок является важнейшим механизмом для выявления любых проблем. Если вы допустите ошибку в имени ключа, то метод `get` может скрыть исключение `KeyError`, предоставив значение по умолчанию.

3.4.7. Задача

Встроенная функция `id` проверяет адрес памяти, занимаемой объектом. Вызов `id("Hello")` вернет адрес объекта `"Hello"`. Можно ли использовать функцию `id` для отслеживания изменений в таком объекте представления словаря, как `dict_keys`? Ожидается, что данные объекта представления будут изменяться

с обновлением объекта `dict`. С другой стороны, адрес памяти объекта представления должен оставаться неизменным.

ПОДСКАЗКА Адрес памяти объекта остается постоянным на протяжении его жизненного цикла. Хотя данные объекта могут изменяться, адрес памяти меняться не должен.

3.5. КОГДА ИСПОЛЬЗОВАТЬ СЛОВАРИ И МНОЖЕСТВА ВМЕСТО СПИСКОВ И КОРТЕЖЕЙ

Мы подробно рассмотрели два контейнера данных: кортежи и списки. В Python нет ограничений относительно типов данных, которые могут в них храниться, и такая гибкость делает их привлекательными моделями данных в любом проекте. В разделе 3.4 упоминается, что контейнер `dict` полезен, потому что в нем хранятся пары «ключ — значение», но как насчет множеств? Кроме того, возможно, вам известно, что в словарях и множествах могут храниться не все типы данных, как показывает следующий листинг.

Листинг 3.6. Неудачная попытка создания объектов dict и set

```
failed_dict = {[0, 2]: "even"}
# ERROR: TypeError: unhashable type: 'list'

failed_set = {{"a": 0}}
# ERROR: TypeError: unhashable type: 'dict'
```

Нехешируемые объекты не могут быть ключами `dict` или элементами `set`. На первый взгляд этот факт кажется недостатком, который снижает полезность этих двух структур данных. Тем не менее для такого решения были веские причины. В этом разделе мы разберемся в том, как ограничения хешируемости способствуют выборке данных с этими двумя структурами и когда их следует применять. Также будет рассмотрена концепция хешируемых и нехешируемых сущностей.

3.5.1. Преимущества постоянной эффективности поиска

В словарях хранятся пары «ключ — значение», и эта схема хранения позволяет получать данные по их ключам. Кроме того, у словарей есть одно важное преимущество: более высокая эффективность выборки конкретных элементов. Так как множества используют тот же базовый механизм хранения, что и словари (хеш-таблицу, см. раздел 3.5.2), они обладают теми же характеристиками — эффективностью поиска элементов. В этом разделе вы узнаете, когда следует отдавать предпочтение словарям или множествам перед списками и кортежами.

Допустим, ваше приложение должно часто выполнять выборку (поиск) элементов. Теоретически для хранения данных можно было бы использовать список или множество. Проведем простой эксперимент для сравнения скорости выборки

случайного элемента из каждого объекта при помощи модулей `timeit` и `random`, как показано в листинге 3.7.

Листинг 3.7. Сравнение скорости выборки данных для list и set

```
from timeit import timeit

for count in [10, 100, 1000, 10000, 100000]:
setup_str = f"""from random import randint; n = {count};
➥ numbers_set = set(range(n));
➥ numbers_list = list(range(n))"""
    stmt_set = "randint(0, n-1) in numbers_set"
    stmt_list = "randint(0, n-1) in numbers_list"
    t_set = timeit(stmt_set, setup=setup_str, number=10000)
    t_list = timeit(stmt_list, setup=setup_str, number=10000)
    print(f"{count: >6}: {t_set:e} vs. {t_list:e}")
```

Строка для настройки хронометражного теста

Строка для проверки присутствия в объекте set

Вычисляет среднее время выполнения

Строка для проверки присутствия в объекте list

ОБРАТИТЕ ВНИМАНИЕ Модуль `timeit`, являющийся частью стандартной библиотеки Python, позволяет проанализировать быстродействие операций, а модуль `random` предоставляет функциональность генерирования случайных чисел. Наличие этих встроенных средств в очередной раз показывает, насколько серьезно подходит Python к предоставлению повседневных инструментов для нашей работы.

В листинге 3.7 цикл `for` используется для перебора нескольких ситуаций с переменным количеством элементов в объектах `list` и `set`. После выполнения кода вы получите следующий результат:

```
    10: 1.108225e-02 vs. 9.955332e-03
   100: 9.514037e-03 vs. 1.533820e-02
  1000: 1.051638e-02 vs. 7.346468e-02
 10000: 1.034654e-02 vs. 6.189157e-01
100000: 1.086105e-02 vs. 6.290399e+00
```

На других компьютерах будут выводиться другие результаты

УДОБОЧИТАЕМОСТЬ Мы использовали f-строки для форматирования строкового вывода. Точнее говоря, мы применили спецификатор формата, определяющий выравнивание текста, чтобы создать визуальную структуру и упростить чтение результатов.

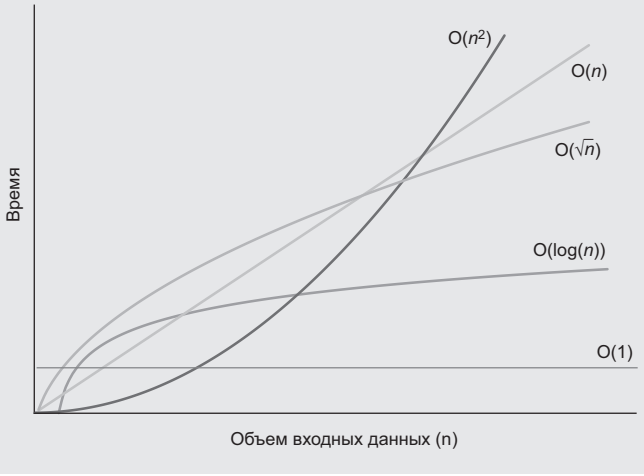
С возрастанием количества элементов в множестве время поиска остается на том же уровне, то есть операция выполняется за постоянное время, что называется временно́й сложностью O(1). Иначе говоря, каким бы большим ни было множество, поиск элемента занимает приблизительно одно и то же время. С другой стороны, приблизительное время поиска возрастает в линейной зависимости от размера

списка. В отличие от множеств, использующих хеш-таблицы для индексирования объектов с хеш-кодами (раздел 3.5.2), списки требуют обхода элементов для проверки их наличия в списке, а время обхода напрямую зависит от количества элементов. Различия во временной сложности подчеркивают преимущества использования множеств вместо списков, если ваши задачи связаны с поиском элементов.

В этом примере объект множества `set` используется для проверки эффективности поиска элементов и демонстрации временной сложности O(1). Такой же эффективностью обладают объекты `dict`, так как они используют тот же базовый механизм хранения — хеш-таблицы. Каждому ключу в объекте `dict` и каждому элементу в объекте `set` соответствует некоторое значение хеш-кода. Но что это означает? Тема рассматривается в разделе 3.5.2.

Сложность алгоритмов

В computer science алгоритмы на концептуальном уровне могут представляться как инструкции для решения задачи, например, сортировки списка или выборки элемента из последовательности. Не все алгоритмы решают задачи с одинаковой скоростью. Для количественной оценки быстродействия используется *временна́я сложность*, описывающая время, необходимое для выполнения алгоритма. Для обозначения временной сложности используется так называемая нотация «O-большое», где в круглых скобках записывается функция от количества элементов, обычно обозначаемое как n. Так, запись O(n) означает, что время, необходимое для выполнения алгоритма, линейно зависит от количества задействованных элементов; $O(n^2)$ означает, что необходимое время находится в квадратичной зависимости от количества элементов; O(1) — что время постоянно и не зависит от количества элементов. Следующий график дает общее представление о разных вариантах временной сложности.



Графики временной сложности разных порядков. Переменная n представляет количество элементов, задействованных в вычислениях

3.5.2. Хешируемость и хеширование

При создании словарей или множеств не хотелось бы столкнуться с исключением `TypeError` (листинг 3.6). Исключение происходит из-за того, что мы пытаемся использовать нехешируемые объекты в качестве ключей словарей или элементов множеств. Как нетрудно догадаться, только хешируемые объекты могут использоваться со словарями или множествами. Но что же означает «хешируемость»? В этом разделе вы узнаете как о хешируемых, так и о нехешируемых объектах.

> **ОСНОВНЫЕ ПОНЯТИЯ** Когда в программе Python возникает ошибка, мы говорим, что произошло исключение — или что программа *выдает* ошибку или исключение.

Хешируемость не является изолированным понятием. Вероятно, вам встречались термины с тем же корнем — хеш-коды, хеширование, хеш-таблицы и хеш-карты. По своей сути все хешируемые объекты используют одну фундаментальную процедуру: хеширование. На рис. 3.6 представлен общий процесс хеширования, в котором в качестве примера используются ключи словарей. Все начинается с необработанных значений данных — четырех строк. Хеш-функция выполняет серию вычислений с применением конкретных алгоритмов и выводит значения (*хеш-коды*), полученные в результате хеширования для необработанных значений данных.

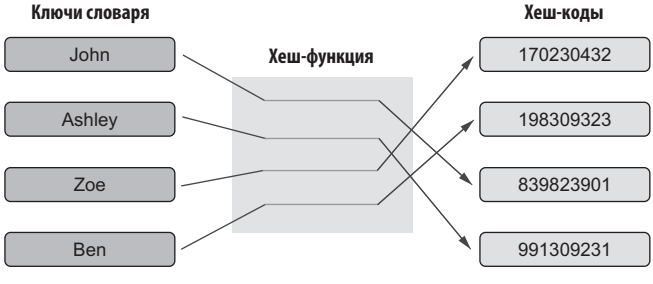


Рис. 3.6. Процесс хеширования с использованием ключей словаря в качестве примера. Хеш-функция обрабатывает ключи словаря и производит целочисленные хеш-коды, однозначно связываемые с каждым из ключей словаря. Предполагается, что разные хеш-функции производят разные хеш-коды

Возьмите на заметку ключевые моменты процесса хеширования.

- *Хеш-функция должна быть вычислительно устойчивой, то есть генерирующей разные хеш-коды для разных объектов.* В редких случаях хеш-функция может производить одинаковые хеш-коды для разных объектов — это явление называется *хеш-коллизией* (hash collision), и оно должно обрабатываться в соответствии с конкретным протоколом.

- *Хеш-функция должна быть последовательной, чтобы для одинаковых объектов всегда генерировались одинаковые хеш-коды.* Когда вы вводите пароль в приложении, этот пароль хешируется и сохраняется в базе данных. При повторной попытке входа введенная строка с паролем хешируется и сравнивается с хранимым хеш-кодом. В этих двух случаях для одного пароля должны генерироваться одинаковые хеш-коды.
- *Для более сложных хеш-функций хеширование выполняется только в одну сторону.* При этом практически невозможно провести восстановление исходных данных по хеш-коду (для этого принимаются соответствующие меры, например введение случайного числа). Необратимость хеширования необходима в области кибербезопасности. Даже если хакер узнает хеш-код пароля, он не сможет определить пароль по хеш-коду (по крайней мере, это будет не слишком легко сделать).

Python реализует хеш-функцию, которая генерирует хеш-коды для своих объектов. Если выражаться точнее, вы можете получить хеш-код объекта при помощи встроенной функции `hash`. Примеры приведены в следующем фрагменте:

```
hash("Hello World!")
# Вывод: 9222343606437197585

hash(100)
# Вывод: 100

hash([1, 2, 3])
# ERROR: TypeError: unhashable type: 'list'
```

У вас значение может быть другим, потому что некоторые хеш-функции зависят от операционной системы

Не каждый объект может сгенерировать хеш-код функцией `hash`. Строки и целые числа являются хешируемыми, но списки нехешируемы. Вы задались вопросом, почему списки нехешируемы или, в более широком смысле, почему словари и множества тоже нехешируемы? Причина проста: эти нехешируемые типы данных являются изменяемыми. Хеш-функция генерирует хеш-код на основании содержимого объекта.

Содержимое изменяемых данных может модифицироваться после создания. Если вам каким-то волшебным образом удастся сделать список хешируемым, то при обновлении списка с изменением содержимого для него должен быть сгенерирован другой хеш-код. Но хеш-функция должна стабильно генерировать один и тот же хеш-код для одного объекта, и в данном случае хеш-код также должен оставаться неизменным для объекта `list`. Очевидно, изменение содержимого `list`, приводящее к изменению хеш-кода, не согласуется с предполагаемой стабильностью хеш-кода для существующего объекта `list` (рис. 3.7).

С другой стороны, у неизменяемых данных (целых чисел, строк и кортежей) содержимое остается постоянным после создания. Постоянство содержимого играет ключевую роль в применении хеш-кода к произвольному объекту. Таким образом, все неизменяемые типы данных являются хешируемыми.

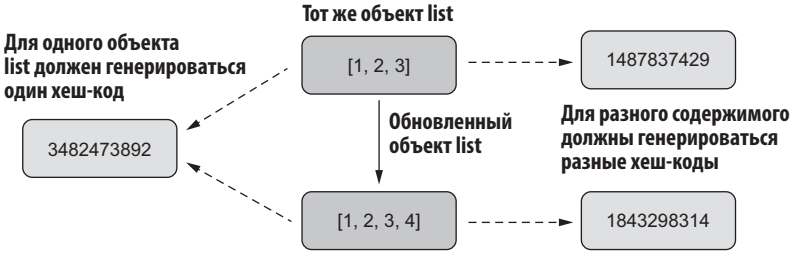


Рис. 3.7. Несовместимость процесса хеширования с изменяемостью объекта. С одной стороны, если объект list является хешируемым, можно ожидать, что список будет генерировать один и тот же хеш-код независимо от содержимого, так как это тот же объект. С другой стороны, после обновления списка для разного содержимого должны генерироваться разные хеш-коды. Эти два сценария несовместимы

Нет ли более прямолинейного способа определить хешируемость объекта без использования функции `hash`? В листинге 3.8 приведено решение. Все выглядит вполне тривиально, не считая использования `Hashable`. Для простоты можно рассматривать `Hashable` как класс, а каждый хешируемый объект — как экземпляр этого класса.

Листинг 3.8. Проверка хешируемости объекта

```
from collections.abc import Hashable

def check_hashability():
    items = [{"a": 1}, [1], {1}, 1, 1.2, "test", (1, 2), True, None]
    for item in items:
        print(f"{str(type(item)): <18} | {isinstance(item, Hashable)}")

print(f"{'Data Type': <18}  {'Hashable'}")
check_hashability()

# Выводимые строки:
Data Type           Hashable
<class 'dict'>     | False
<class 'list'>     | False
<class 'set'>      | False
<class 'int'>      | True
<class 'float'>    | True
<class 'str'>      | True
<class 'tuple'>    | True
<class 'bool'>     | True
<class 'NoneType'> | True
```

Создает список объектов разных типов

isinstance возвращает логическое значение для проверки типа

ОБРАТИТЕ ВНИМАНИЕ Подмодуль `abc` определяет группу абстрактных базовых классов. Он позволяет проверить, предоставляет ли класс определенный интерфейс, например, хешируемость. Упрощенно говоря, он помогает проверить, способен ли объект делать что-то конкретное, скажем, хешироваться.

Как объяснялось выше, изменяемые данные, включая словари, списки и множества, не хешируются. С другой стороны, все остальные неизменяемые типы данных являются хешируемыми. Для встроенных типов данных неизменяемость фактически эквивалентна хешируемости. В табл. 3.1 приводится упорядоченная сводка распространенных типов данных с точки зрения их изменяемости и хешируемости.

Таблица 3.1. Распространенные типы данных как функция хешируемости

Изменяемость	Хешируемость	Типы данных	Может ли использоваться в качестве ключей словаря или элементов множества?
Изменяемые	Нехешируемые	`dict`, `list`, `set`	Нет
Неизменяемые	Хешируемые	`int`, `float`, `str`, `tuple`, `bool`, `NoneType`	Да

В разделе 3.1 была продемонстрирована неизменяемость объектов `tuple`, и вы не сможете присвоить другое значение элементу в объекте `tuple`. Обратите внимание на то, что в табл. 3.1 строки Python также неизменяемы. Об этом свидетельствует невозможность изменения символа или подстроки в строке. Следующий код демонстрирует неизменяемость строк:

```
text = "Hello, World."
text[-1] = "!"
# ERROR: TypeError: 'str' object does not support item assignment
```

Если вам потребуется заменить подстроку, у строк для этого есть метод `replace`. Он создает новую строку, как видно из следующего кода:

```
text.replace(".", "!")
# Вывод: 'Hello, World!'
```

> **ОБРАТИТЕ ВНИМАНИЕ** Мы знаем, что функция `id` может использоваться для проверки адреса памяти объекта — разным объектам должны соответствовать разные адреса памяти. Вы можете сравнить адреса исходной строки и строки, полученной в результате замены.

3.5.3. Обсуждение

Хешируемость — ключевая концепция программирования. Во внутренней реализации Python использует хеш-таблицы в качестве механизма хранения словарей и множеств. Самое важное преимущество хеш-таблиц заключается в том, что выборка данных выполняется с быстродействием O(1), вследствие чего эта модель идеально подходит для ситуаций, требующих быстрого поиска элементов. В том числе объекты `set` часто используются для хранения данных, в которых проверяется присутствие тех или иных значений.

3.5.4. Задача

Дженнифер изучает Python, потому что она собирается строить карьеру в data science. Она узнает, что объект `dict` не может содержать повторяющиеся ключи из-за используемой реализации в форме хеш-таблиц. Допустим, она создает объект `dict`: `numbers = {1: "one", 1.0: "one point one"}`. Как вы думаете, какие значения будет содержать `numbers`?

> **ПОДСКАЗКА** Когда вы намеренно передаете повторяющиеся ключи, последнее значение переопределяет предыдущее при построении объекта `dict`.

3.6. КАК ИСПОЛЬЗОВАТЬ ОПЕРАЦИИ НАД МНОЖЕСТВАМИ ДЛЯ ПРОВЕРКИ ОТНОШЕНИЙ МЕЖДУ СПИСКАМИ

Списки — первый кандидат на хранение однородных данных. Иногда в программе создается несколько списков для хранения похожих элементов и требуется определить отношения между объектами `list`: допустим, вы используете API для загрузки списка акций, рекомендованных инвестиционной компанией. Текущие акции каждого клиента также хранятся в объекте `list`. Для простоты мы начнем со следующих данных:

```
good_stocks = ["AAPL", "GOOG", "AMZN", "NVDA"]
client0 = ["GOOG", "AMZN"]
client1 = ["AMZN", "SNAP"]
```

Одна из специфических функций приложения — проверка того, входят ли все акции клиента в рекомендуемый список. Как решить эту проблему? Для этого можно воспользоваться некоторыми методами `list`. Однако объекты `set` в Python, как и математические множества, поддерживают ряд удобных методов для проверки отношений между объектами `set`. В этом разделе мы рассмотрим уникальные операции класса `set`, и вы научитесь использовать эти операции для решения задач, связанных с отношениями между списками.

3.6.1. Проверка вхождения всех элементов в другой список

Для реализации описанной выше функциональности необходимо ответить на вопрос: как проверить, содержит ли объект `list` все элементы другого объекта `list`? В этом разделе вы узнаете, как использовать для этого операции `set`. Без применения операций `set` начинающий программист предложил бы решение, основанное на переборе объекта `list`. Создадим функцию, которую можно вызвать в любой нужный момент.

Листинг 3.9. Проверка полного вхождения списка в другой список

```
def all_contained_in_recommended(recommended, personal):
    print(f"Is {personal} contained in {recommended}?")
    for stock in personal:
        if stock not in recommended:
            return False
    return True
```

"not in" проверяет, что элемент не содержится в коллекции

СОПРОВОЖДАЕМОСТЬ Если вам нужно разработать общее решение для нескольких похожих сценариев использования, подумайте о создании функции. Когда функциональность вдруг потребуется изменить, достаточно будет внести изменения в одну функцию вместо всех дубликатов, которые делают то же самое.

Логика приведенной в листинге 3.9 функции такова: если мы находим хотя бы один случай, в котором акции не входят в рекомендованный список, мы заключаем, что список клиента не содержится полностью в рекомендованном списке. По этой логике выполняется перебор элементов списка акций клиента. Если какие-либо акции отсутствуют в рекомендованном списке, происходит выход из функции с возвращением `False`; в противном случае после перебора всего списка возвращается `True`. С этой функцией можно протестировать пару примеров:

```
print(all_contained_in_recommended(good_stocks, client0))
# Выводимые строки:
Is ['GOOG', 'AMZN'] contained in ['AAPL', 'GOOG', 'AMZN', 'NVDA']?
True

print(all_contained_in_recommended(good_stocks, client1))
# Выводимые строки:
Is ['AMZN', 'SNAP'] contained in ['AAPL', 'GOOG', 'AMZN', 'NVDA']?
False
```

Оба примера работают так, как ожидалось. Однако есть и другое, более эффективное решение, не требующее создания функции. Еще один важный принцип программирования — «Не изобретайте велосипед». Если существует готовое решение, используйте его. Более эффективное решение применяет операции с объектами `set`:

```
good_stocks_set = set(good_stocks)
contained0 = good_stocks_set.issuperset(client0)
print(f"Is {client0} contained in {good_stocks}? {contained0}")
# Вывод: Is ['GOOG', 'AMZN'] contained in
➥  ['AAPL', 'GOOG', 'AMZN', 'NVDA']? True

contained1 = good_stocks_set.issuperset(client1)
print(f"Is {client1} contained in {good_stocks}? {contained1}")
# Вывод: Is ['AMZN', 'SNAP'] contained in
➥  ['AAPL', 'GOOG', 'AMZN', 'NVDA']? False
```

Создает объект set

Использует метод issuperset

Чтобы использовать метод `issuperset`, мы преобразуем объект `list` с именем `good_stocks` в объект `set` с именем `good_stocks_set`. Мы вызываем метод `issuperset` объекта `good_stocks_set` и передаем в аргументе объект `list` `client0` или `client1`. Как и предполагалось, при этом возвращается нужный результат. Теоретически можно было бы использовать метод `issubset` для реализации этой функциональности, но это потребует создания объектов `set` для списка каждого клиента, что приводит к избыточному дублированию. По этой причине метод `issuperset` эффективнее `issubset` при совместном использовании объекта `set`, который предположительно является надмножеством. В нашем случае это множество рекомендованных акций.

Как видите, решение с `issuperset` получается более компактным, чем решение с пользовательской функцией. К тому же использование встроенной функции вместо пользовательской снижает риск ошибок.

> **СОПРОВОЖДАЕМОСТЬ** Хорошо иметь возможность написать функцию для решения конкретной задачи. Но еще лучше использовать существующие функции, например встроенные!

3.6.2. Проверка вхождения любого элемента списка в другой список

Еще одна распространенная задача, связанная с отношениями между списками, — проверка того, содержит ли список хотя бы один элемент другого списка. Она рассматривается в этом разделе.

Для простоты обсуждения возьмем пример с рекомендуемыми акциями. Допустим, вы хотите проверить, содержит ли список акций клиента какие-либо из рекомендованных акций. Как показано в разделе 3.6.1, эта функциональность предоставляется механизмом перебора (листинг 3.10).

Листинг 3.10. Проверка вхождения любого элемента списка в другой список

```
def contained_any_in_recommended(recommended, personal):
    print(f"Does {personal} contain any in {recommended}?")
    for stock in personal:
        if stock in recommended:
            return True
    return False
```

Логика функции из листинга 3.10 противоположна логике листинга 3.9. Если хотя бы один элемент клиентского списка входит в рекомендуемый список, критерий выполнен и функция возвращает `True`; в противном случае подходящей записи нет и функция возвращает `False`. В следующем фрагменте продемонстрированы два сценария использования:

```
print(contained_any_in_recommended(good_stocks, client0))
# Выводимые строки:
```

```
Does ['GOOG', 'AMZN'] contain any in ['AAPL', 'GOOG', 'AMZN', 'NVDA']?
True

print(contained_any_in_recommended(good_stocks, client1))
# Выводимые строки:
Does ['AMZN', 'SNAP'] contain any in ['AAPL', 'GOOG', 'AMZN', 'NVDA']?
True
```

Вопрос о том, содержит ли список любой из элементов другого списка, фактически сводится к вопросу о том, есть ли перекрытие между списками. К сожалению, встроенных методов для проверки пересечений между двумя объектами `list` не существует, однако такие методы есть у объектов `set`. Одна из ключевых операций `set` вычисляет пересечение между двумя объектами `set`, а это именно то, что там нужно. Решение выглядит так:

```
good_stocks_set & set(client0)
# Вывод: {'AMZN', 'GOOG'}

bool(good_stocks_set & set(client0))
# Вывод: True

good_stocks_set & set(client1)
# Вывод: {'AMZN'}

bool(good_stocks_set & set(client1))
# Вывод: True
```

Оператор пересечения `&` позволяет без труда вычислить пересечение между двумя объектами множеств. Если вы хотите получить логический результат, используйте встроенную функцию `bool`, которая интерпретирует любую непустую коллекцию (как `set` в этом случае) как `True`.

> **ОБРАТИТЕ ВНИМАНИЕ** Функция `bool` представляет собой конструктор, который создает объект `bool` вычислением объекта в круглых скобках. Объекты `set` интерпретируются как `True`, если они содержат хотя бы один элемент.

Кроме использования `&` с двумя объектами `set`, операцию пересечения можно вычислить методом `intersection`. Как и `issuperset`, операция `intersection` удобна тем, что объекты списков `client0` и `client1` можно передать методу напрямую, без предварительного преобразования их в объекты `set`. Следующий фрагмент демонстрирует эту возможность:

```
good_stocks_set.intersection(client0)
# Вывод: {'AMZN', 'GOOG'}

good_stocks_set.intersection(client1)
# Вывод: {'AMZN'}
```

В разделах 3.6.1 и 3.6.2 операции `set` использовались для проверки стандартных отношений между объектами `list`. А теперь немного отступим назад

и рассмотрим более общие операции с объектами `set` в Python, прежде всего связанные с анализом отношений между множествами.

3.6.3. Работа с несколькими объектами set

Как упоминалось в разделе 3.5, объекты `set` лучше использовать в случаях, требующих проверки принадлежности, потому что эта операция выполняется со сложностью O(1). Помимо проверки принадлежности, при наличии нескольких взаимосвязанных объектов `set` может возникнуть необходимость выполнять различные операции между ними. В этом разделе представлены основы операций, выполняемых с несколькими объектами `set`. Самыми распространенными являются четыре операции: объединение (union), пересечение (intersection), симметрическая разность (symmetric difference) и разность (difference) (рис. 3.8).

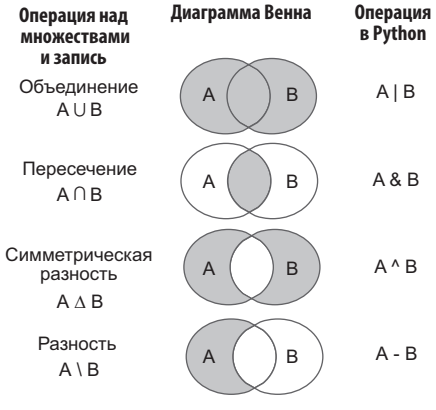


Рис. 3.8. Операции над множествами в Python. На схеме представлены четыре распространенные операции над множествами с соответствующими математическими обозначениями, диаграммами Венна и операциями Python. Объединение состоит из элементов, входящих хотя бы в одно из множеств A или B. Пересечение состоит из элементов, входящих как в множество A, так и в множество B. Симметрическая разность состоит из элементов, входящих только в одно из множеств A и B, но не в оба сразу. Разность состоит из элементов одного множества, не входящих во второе

У всех четырех операций имеются соответствующие специальные операторы, упрощающие синтаксис. В следующем фрагменте представлены все эти операции. Как видно из приведенного примера, эти операции удобны для выбора элементов, удовлетворяющих некоторому критерию, например принадлежащих обоим множествам (пересечение) или хотя бы одному из двух множеств (объединение).

```
tasks_a = {"Homework", "Laundry", "Grocery"}
tasks_b = {"Laundry", "Gaming"}

tasks_a | tasks_b  <-------  Операция объединения (|)
# Вывод: {'Laundry', 'Gaming', 'Homework', 'Grocery'}

tasks_a & tasks_b  <-------  Операция пересечения (&)
# Вывод: {'Laundry'}

tasks_a ^ tasks_b  <-------  Операция симметрической разности (^)
# Вывод: {'Homework', 'Grocery', 'Gaming'}

tasks_a - tasks_b  <-------  Операция разности (-)
# Вывод: {'Homework', 'Grocery'}
```

ОБРАТИТЕ ВНИМАНИЕ Вы можете получить результаты в другом порядке. Элементы, хранящиеся в объекте `set`, не упорядочены, потому что они используют хеш-таблицы и определенного порядка следования элементов в множестве не существует.

Кроме этих операций, создающих множество на базе других множеств, существуют методы `issubset` и `issuperset`, которые проверяют отношения между двумя множествами. `issubset` проверяет, является ли объект, для которого был вызван метод, подмножеством другого множества (в более общем виде это может быть произвольный итерируемый объект), а `issuperset` проверяет обратное условие. Несколько простых примеров:

```
small_set = {1, 2}
large_set = {1, 2, 3, 4}

assert small_set.issubset(large_set) == True
assert small_set.issuperset(large_set) == False

assert large_set.issubset(small_set) == False
assert large_set.issuperset(small_set) == True
```

Вы уже познакомились с четырьмя операциями над множествами (объединение, пересечение, симметрическая разность и разность), соответствующими методами и операторами. Интересно, что у методов `issuperset` и `issubset` тоже существуют операторы. Эти методы и операторы представлены в табл. 3.2.

Таблица 3.2. Операторы множеств с соответствующими методами

Операция над множествами	Оператор	Метод
Объединение	`|`	`union`
Пересечение	`&`	`intersection`
Симметрическая разность	`^`	`symmetric_difference`
Разность	`-`	`difference`
Проверка того, что одно множество является надмножеством для другого	`>=`	`issuperset`
Проверка того, что одно множество является подмножеством для другого	`<=`	`issubset`
Проверка того, что одно множество является строгим надмножеством для другого	`>`	Отсутствует, но реализуется объединением `issuperset` с `!=`
Проверка того, что одно множество является строгим подмножеством для другого	`<`	Отсутствует, но реализуется объединением `issubset` с `!=`

Хотя с операторами код становится более компактным, они работают только с объектами `set`. Напротив, все методы могут получать итерируемые объекты в параметрах, что повышает их гибкость. Если вы работаете с итерируемыми объектами, которые не являются объектами `set`, рассмотрите возможность использовать эти объекты напрямую — это избавит вас от необходимости сначала преобразовывать их в объекты `set`.

УДОБОЧИТАЕМОСТЬ При выполнении операций над множествами отдавайте предпочтение методам — они не только обладают большей гибкостью (так как могут получать любые итерируемые объекты), но и лучше читаются (благодаря понятным именам).

3.6.4. Обсуждение

Объекты `set` считаются предпочтительной моделью данных для хранения уникальных элементов, и Python предоставляет ряд операций для работы с наборами объектов `set` и проверки отношений между ними. Так как у списков нет встроенных методов для проверки отношений между ними, удобнее будет преобразовать списки в множества для определения отношений между исходными объектами `list`.

3.6.5. Задача

При выполнении операции объединения двух множеств генерируется множество, состоящее из всех элементов обоих множеств. В этом отношении операция напоминает операцию OR. А вы знаете, что произойдет, если использовать ключевое слово `or` между двумя множествами? Какой результат вы бы ожидали получить при выполнении операции `{1, 2, 3} or {4, 5, 6}`? Аналогичным образом операцию пересечения можно уподобить операции AND. Как вы думаете, как результат будет получен при выполнении операции `{1, 2, 3} and {4, 5, 6}`?

ПОДСКАЗКА Эти вычисления также называются *ускоренными*. Результатом операции `or` становится первый объект, если этот первый объект интерпретируется как логическое значение `True`; в противном случае результатом становится второй объект. Для операции `and` результатом становится первый объект, если этот первый объект интерпретируется как логическое значение `False`; в противном случае результатом становится второй объект.

ИТОГИ

- Список — изменяемый тип данных, позволяющий добавлять, вставлять, обновлять и удалять элементы, тогда как кортежи являются неизменяемыми, то есть не могут меняться после создания.

- Кроме различий в изменяемости, списки и кортежи различаются по однородности содержащихся в них данных. Списки используются для хранения семантически однородных элементов, и эти элементы образуют линейную упорядоченную последовательность. Кортежи используются для хранения семантически разнородных элементов, образующих структурную последовательность.
- Сортировка по умолчанию может сортировать список только в числовом или лексикографическом порядке; эти возможности весьма ограниченны. Следовательно, вы должны понимать, как передать пользовательскую функцию в аргументе `key` для указания требований к сортировке.
- Если вам нужен простой контейнер данных, рассмотрите возможность использования именованных кортежей, позволяющих создать класс одной строкой кода. Именованные кортежи обладают рядом преимуществ, включая эффективность использования памяти и точечную запись для обращения к атрибутам.
- Для обращения ко всем ключам, значениям или парам «ключ — значение» словаря используйте объекты представлений словаря, потому что они автоматически обновляются и синхронизируются с объектом словаря, на основе которого созданы.
- Только хешируемые объекты могут быть ключами словарей и элементами множеств в Python. К числу популярных хешируемых типов данных относятся `int`, `float`, `str`, `tuple`, `bool` и `NoneType`.
- Поиск элементов в словарях и множествах, в основе которых лежит хеширование, выполняется эффективно с временной сложностью O(1).
- Метод `get` получает элемент словаря без выдачи исключения. Этот метод рекомендуется использовать при работе со словарями, которые были созданы другими людьми.
- При работе с созданными вами словарями можно использовать индексную запись (`dict[key]`), при которой любые ошибки в ключах проявятся сами собой.
- Множество `set` — структура данных, специализирующаяся на работе с имеющими уникальные значения элементами. Между объектами `set` могут выполняться различные операции, включая объединение и разность. Эти операции могут использоваться для проверки отношений между всеми другими типами данных, кроме множеств, например `list`.

4 Работа с последовательностями

В этой главе

- ✓ Использование срезов для получения подпоследовательностей и выполнения операций с ними
- ✓ Комбинации положительных и отрицательных индексов при выборке элементов
- ✓ Поиск элементов в последовательности
- ✓ Распаковка последовательности
- ✓ Модели данных, отличные от списков

В главе 3 вы узнали, как использовать списки и кортежи для хранения данных. У списков и кортежей есть одна общая особенность: элементы следуют в определенном порядке. Эти две структуры данных являются примерами более общего типа данных — *последовательности* (sequence). В Python существуют и другие последовательности, например строки и байты. Модели данных последовательностей принадлежат к числу важнейших структур данных. Причина проста: данные используются для моделирования реальной жизни, в которой полно упорядоченных объектов/событий — очереди, письменность, номера домов и многое другое. Следовательно, эффективная обработка данных последовательностей нужна во всех программных проектах независимо от специализации.

С точки зрения реализации Python структуры данных последовательностей обладают рядом общих характеристик, которые стоит рассматривать вместе. Как выясняется, приемы, на первый взгляд применимые только к одной модели данных (например, распаковка объекта кортежа), подходят ко всем моделям данных последовательностей. Хотя в примерах этой главы я в основном использую списки или строки, не стоит полагать, что рассматриваемые навыки актуальны только для списков и строк.

4.1. КАК ПОЛУЧАТЬ ПОДПОСЛЕДОВАТЕЛЬНОСТИ И ВЫПОЛНЯТЬ НАД НИМИ ОПЕРАЦИИ С ПОМОЩЬЮ СРЕЗОВ

При работе с данными последовательностей может потребоваться получить конкретное подмножество последовательности, которое мы будем называть *подпоследовательностью*. К числу встроенных типов данных относятся распространенные модели данных последовательностей `str`, `list` и `tuple`:

```
# str - последовательность символов:
text = "Hello, World!"

# list - изменяемая последовательность произвольных объектов
fruits = ["apple", "orange", "banana", "strawberry"]

# кортеж - неизменяемая последовательность произвольных объектов
vowels = ("a", "e", "i", "o", "u")
```

Если вы извлекаете подпоследовательность объекта `list`, это называется слайсингом, или нарезкой (slicing). Простейшая форма слайсинга — `list[start:end]`, и в срез включаются все элементы между индексами `start` и `end` (элемент с индексом `end` не включается):

```
assert fruits[1:3] == ["orange", "banana"]
```

В этом разделе мы выйдем за пределы базовой формы среза `list[start:end]`. Мы рассмотрим нетривиальные возможности слайсинга, и вы научитесь пользоваться этими возможностями для получения подпоследовательностей и выполнения операций с ними.

4.1.1. Использование всех возможностей слайсинга

Кроме определения начального и конечного индекса, слайсинг имеет ряд вариаций, предоставляющих разные возможности извлечения подпоследовательностей. Ниже рассматриваются самые важные из них:

- игнорирование начального или конечного индекса;
- допустимость нарушения границ при слайсинге;
- применение смещения при определении среза.

Игнорирование начального или конечного индекса

По умолчанию начальный индекс равен 0, и если вы хотите получить первые n элементов, питонический стиль предписывает опустить начальный индекс и использовать запись `list[:end]`. По умолчанию конечный индекс равен длине списка и выборка при слайсинге не включает конечный индекс, так что для получения последних n элементов списка используется запись `list[start:]`. Понятно, что игнорирование начального или конечного индекса избавляет от лишнего кода и упрощает чтение кода:

```
assert fruits[:3] == ["apple", "orange", "banana"]
assert fruits[1:] == ["orange", "banana", "strawberry"]
```

А если опустить как начальный, так и конечный индекс? Вероятно, вы уже догадались: `list[:]` получает все элементы, то есть копию исходного списка (в разделе 10.3 тема копирования объектов рассматривается более подробно). Следующий фрагмент кода показывает, что запись `[:]` получает все элементы объекта `list`:

```
assert fruits[:] == ["apple", "orange", "banana", "strawberry"]
```

> **УДОБОЧИТАЕМОСТЬ** Код будет проще читаться, если вы опустите начальный или конечный индекс (когда это возможно).

Допустимость нарушения границ при слайсинге

Одна из особенностей слайсинга — допустимость выхода индексов за границы диапазона, так как Python ограничивает срез максимально допустимой границей. У каждого элемента последовательности имеется индекс, обозначающий его позицию. Если вы используете индекс, не совпадающий ни с одним элементом последовательности, возникает исключение `IndexError`, указывающее на выход за пределы допустимого диапазона:

```
fruits[5]
# ERROR: IndexError: list index out of range
```

При этом Python допускает выход индексов, используемых при слайсинге, за пределы допустимого диапазона, например, если индекс не соответствует ни одному элементу последовательности. Следующие примеры демонстрируют эту возможность:

```
numbers = [0, 1, 2, 3, 4, 5]
numbers[:20]  <-- Индекс превышает индекс конечного элемента
# Вывод: [0, 1, 2, 3, 4, 5]

numbers[-10000:2]  <-- Индекс меньше индекса первого элемента
# Вывод: [0, 1, 2]
```

И хотя возможность выхода индексов за пределы допустимого диапазона при слайсинге добавляет некоторую гибкость в получении элементов, я не

рекомендую ею пользоваться, потому что она сбивает с толку читателя кода. Читающий код решит, что в программе допущена ошибка или что программист забыл обновить индексы. В любом случае код теряет свою ясность.

СОПРОВОЖДАЕМОСТЬ При использовании индексов, выходящих за пределы допустимого диапазона, вы только путаете себя и своих коллег.

Применение смещения при слайсинге

При слайсинге можно указать величину смещения (stride), чтобы получить выборку элементов, находящихся на равных расстояниях. В запись слайсинга включается дополнительный параметр `stride`: `list[start:end:stride]`, с которым в срез включается каждый n-й элемент, начиная со `start`, пока не будет достигнут индекс `end`. При использовании смещения (или шага, как его называют некоторые программисты) индексы `start` и `end` можно опустить, Python сам подставит необходимые границы. Вот несколько типичных примеров использования (наглядно представленных на рис. 4.1):

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

assert numbers[2:5:2] == [3, 5]

assert numbers[::3] == [1, 4, 7]

assert numbers[::-1] == [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

С приращением 2 включается каждый второй элемент

С приращением 3 включается каждый третий элемент

С приращением –1 сегментирование начинается справа и двигается влево

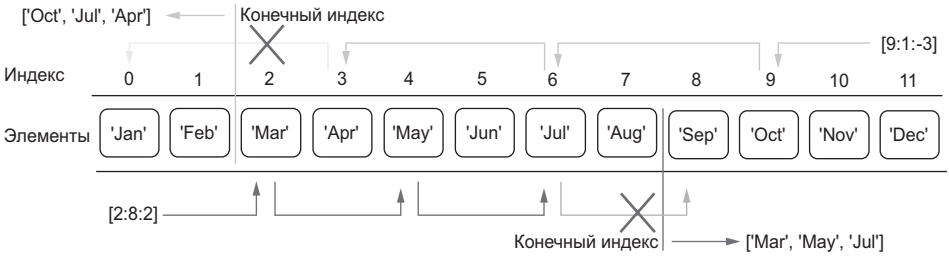


Рис. 4.1. Слайсинг списка с положительным и отрицательным приращением. С положительным приращением слайсинг начинается слева, а с отрицательным — справа

С положительными приращениями все просто. Однако слайсинг также поддерживает отрицательные приращения, которые нередко вызывают затруднения. Один из трюков Python, который известен многим разработчикам, — обратная перестановка элементов списка записью `list[::-1]`, как в предыдущем примере, но многие не понимают, почему это происходит. Дело в том, что с отрицательным приращением слайсинг начинается с правого края и двигается влево. Таким образом, приращение –1 означает непрерывное получение элементов справа

налево. Так как начальный и конечный индекс не указаны, то справа налево нарезается весь список; происходит обратная перестановка. На рис. 4.1 наглядно представлены различия между положительным и отрицательным приращением.

> **СОПРОВОЖДАЕМОСТЬ** При слайсинге избегайте любых отрицательных приращений, кроме –1. Они не очевидны на интуитивном уровне и могут создать путаницу в программе.

И хотя слайсинг поддерживает отрицательные приращения, я не рекомендую использовать эту возможность, потому что она затрудняет чтение кода. Если вы хотите выделить подпоследовательность справа налево, используйте метод `reverse` для обратной перестановки списка на месте (вызов `reverse` изменяет исходный список), а затем выполните операцию в направлении слева направо. Такое решение потребует дополнительной строки кода, но читателю будет намного проще понять смысл слайсинга.

4.1.2. Различия между срезами и интервалами

Во внутренней реализации при получении подпоследовательности создается объект среза `slice`. Иначе говоря, слайсинг списка `list[start:stop:end]` эквивалентен `list[slice(start, stop, step)]`. Но другой класс `range` имеет такую же сигнатуру вызова: `range(start, stop, step)`.

Это сходство сбивает с толку некоторых неопытных разработчиков. В этом разделе я объясню, чем различаются эти два класса.

`slice` и `range` похожи, так как их конструкторы получают аргументы `start`, `stop` и `step` и создают три атрибута, `start`, `stop` и `step`:

```
slice_obj = slice(1, 10, 2)
range_obj = range(1, 10, 2)

slice_obj.start, slice_obj.stop, slice_obj.step
# Вывод: (1, 10, 2)

range_obj.start, range_obj.stop, range_obj.step
# Вывод: (1, 10, 2)
```

Такое сходство может породить путаницу. И все же между срезами `slice` и интервалами `range` существует два важных различия, из-за которых они не могут считаться взаимозаменяемыми. Во-первых, интервалы являются итерируемыми объектами, а срезы — нет. Как следствие, вы можете применять интервалы для создания списка или использовать их в цикле `for`, тогда как срезы в этих операциях использоваться не могут. Следующий фрагмент кода демонстрирует пример:

```
list(range(10))
# Вывод: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

list(slice(10))
# ERROR: TypeError: 'slice' object is not iterable
```

Во-вторых, объект `slice` может использоваться для получения элементов списка или других данных последовательности. В следующем примере мы получаем нечетные числа с объектом `slice`, но с объектом `range` такая операция недопустима:

```
numbers = list(range(10))

odd_slice = slice(1, 10, 2)
numbers[odd_slice]
# Вывод: [1, 3, 5, 7, 9]

odd_range = range(1, 10, 2)
numbers[odd_range]
# ERROR: TypeError: list indices must be integers or slices, not range
```

На рис. 4.2 продемонстрированы различия между объектами `slice` и `range`, а также сходство в их конструкторах и атрибутах.

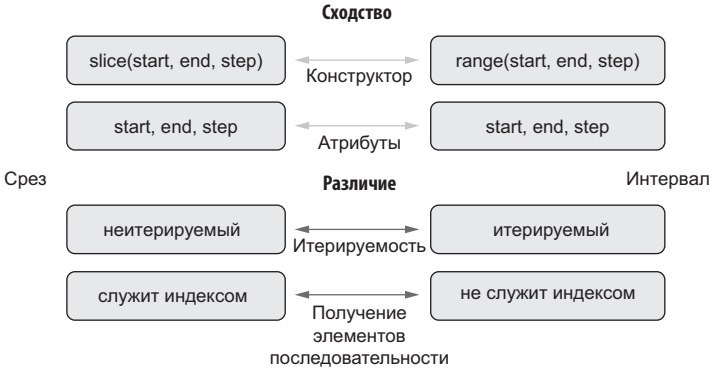


Рис. 4.2. Сходство и различия между объектами slice и range. Объекты обладают сходными сигнатурами конструкторов и атрибутами. Интервалы (но не срезы!) являются итерируемыми объектами, а срезы (но не интервалы!) служат индексами для получения элементов из последовательности

4.1.3. Использование именованных объектов slice для обработки данных последовательностей

В большинстве случаев для получения элементов используется слайсинг с указанием начального и конечного индекса: `list[start:stop]`. Такое решение работает, если данные в последовательности достаточно просты. Но если последовательность содержит более сложные данные, для улучшения удобочитаемости кода следует использовать объекты `slice` с содержательными именами.

Допустим, вы обрабатываете текстовые данные, сгенерированные из внешнего источника для нашего таск-менеджера. Из-за настроек форматирования текстовые данные выглядят примерно так (числа в тексте представляют индексы символов):

```
tasks = """
0....5.............20.........................48......
1001 Laundry        Wash all clothes            3
1002 Museum Visit   Go to the Egypt exhibit     4
1003 Do Homework    Physics and math            5
1004 Go to Gym      Work out for 1 hour         2
"""
```

Обратите внимание на вертикальное выравнивание полей данных в каждой строке. Использование объектов `slice` — лучший практический прием, и в листинге 4.1 приведена его возможная реализация.

Листинг 4.1. Использование именованных срезов при обработке данных

```
task_id = slice(5)
task_title = slice(5, 20)
task_desc = slice(20, 48)
task_urgency = slice(48, 49)

task_lines = tasks.split("\n")[2:-1]

tasks = []
for line in task_lines:
    task = (line[task_id].strip(), line[task_title].strip(),
    ➥ line[task_desc].strip(), line[task_urgency].strip())
    tasks.append(task)

print(tasks)
# Выводимые строки (переформатированы для наглядности):
[('1001', 'Laundry', 'Wash all clothes', '3'),
('1002', 'Museum Visit', 'Go to the Egypt exhibit', '4'),
('1003', 'Do Homework', 'Physics and math', '5'),
('1004', 'Go to Gym', 'Work out for 1 hour', '2')]
```

Использует метод strip для удаления завершающих пробелов

Чтобы разделить идентификатор задачи, название, описание и степень срочности, мы создадим четыре объекта `slice` для извлечения каждой соответствующей подстроки. Технически можно применить слайсинг непосредственно к строке, например, `line[:5]` для названия. Однако имена объектов `slice` ясно показывают, какие данные содержатся в каждом срезе. К тому же с точки зрения простоты сопровождения при изменении форматирования в текстовых файлах (например, вставке дополнительных пробелов между полями данных) будет проще изменить объекты `slice`, обновляя индексы в соответствии с новыми требованиями к форматированию.

СОПРОВОЖДАЕМОСТЬ Именованные срезы легко читаются и ясно показывают, какие данные они представляют.

4.1.4. Операции с элементами списков с применением слайсинга

В разделах с 4.1.1 по 4.1.3 вы узнали о получении подпоследовательностей с использованием слайсинга. Эти операции доступны для всех моделей данных

последовательностей, как для изменяемых (таких, как `list` и `bytearray`), так и для неизменяемых (таких, как `tuple` и `string`), как показано в табл. 4.1.

Изменяемые модели данных последовательностей поддерживают еще один набор операций, которые мы будем называть *операциями над срезами*. В этом разделе вы узнаете, как обрабатывать элементы изменяемой последовательности.

Таблица 4.1. Распространенные модели данных последовательностей и их зависимость от изменяемости

Изменяемость	Типы данных	Допустимость операций над срезами
Изменяемые	`list, bytearray`	Да
Неизменяемые	`str, tuple, range, bytes`	Нет

Если использовать в качестве примера списки, операции над срезами дают возможность манипулировать с подпоследовательностью списка, полученной при помощи объекта `slice`. Операции над срезами позволяют выполнять различные манипуляции с подпоследовательностями, включая замену, расширение, сжатие и удаление. Чтобы заменить подпоследовательность, присвойте полученной подпоследовательности то же количество элементов:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8]
numbers[:3] = [10, 11, 12]
numbers
# Вывод: [10, 11, 12, 3, 4, 5, 6, 7, 8]
```

Чтобы расширить подпоследовательность, присвойте более длинную подпоследовательность исходной подпоследовательности:

```
numbers[3:] = [13, 14, 15, 16, 17, 18, 19, 20]
numbers
# Вывод: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Чтобы сжать подпоследовательность, присвойте более короткую подпоследовательность исходной подпоследовательности:

```
numbers[:5] = [0, 1]
numbers
# Вывод : [0, 1, 15, 16, 17, 18, 19, 20]
```

Заметим, что подпоследовательность не обязана быть непрерывной. Даже если для последовательности задано приращение, это не помешает выполнить замену. Как показано в следующем примере, при задании приращения 2 мы обновляем каждый второй элемент списка заданными элементами:

```
numbers[::2] = [0, 0, 0, 0]
numbers
# Вывод: [0, 1, 0, 16, 0, 18, 0, 20]
```

При необходимости подпоследовательность можно удалить командой `del`. Кроме того, подпоследовательности можно присвоить пустой список, чтобы соответствующие элементы были удалены:

```
numbers = [0, 1, 0, 16, 0, 18, 0, 20]
del numbers[:4]
print(numbers)
# Вывод: [0, 18, 0, 20]

numbers[-2:] = []
print(numbers)
# Вывод: [0, 18]
```

4.1.5. Обсуждение

Если задать отрицательное приращение, слайсинг обрабатывает последовательность справа налево. Так как людям обычно более привычна обработка в порядке слева направо, отрицательное приращение может создавать путаницу, поэтому применять его следует с осторожностью.

При обработке серии последовательностей в согласованном формате, как показано в примере с обработкой текстовых данных (раздел 4.1.3), рекомендуется использовать именованные объекты `slice`. При таком подходе каждое имя ясно показывает, какие данные соответствуют той или иной подпоследовательности, а это улучшает удобочитаемость кода.

4.1.6. Задача

Джейсон изучает Python, чтобы анализировать новости туризма, это входит в круг его интересов в области машинного обучения. На работе ему приходится иметь дело с разнообразными данными последовательностей. Он хочет попробовать применить слайсинг с разными типами последовательностей, такими как строки и кортежи. Поможете ли вы ему подобрать типы данных для генерируемых подпоследовательностей этих типов последовательностей? Как показывает табл. 4.1, интервалы тоже являются разновидностью последовательностей. Пожалуйста, попробуйте создать подпоследовательность для интервала.

> **ПОДСКАЗКА** Генерируемая подпоследовательность должна напоминать «родительскую» последовательность в отношении своего типа.

4.2. КАК ИСПОЛЬЗОВАТЬ ПОЛОЖИТЕЛЬНЫЕ И ОТРИЦАТЕЛЬНЫЕ ИНДЕКСЫ ДЛЯ ПОЛУЧЕНИЯ ЭЛЕМЕНТОВ

У всех объектов последовательностей есть одна общая характеристика: хранимые данные имеют линейный порядок, и каждая точка данных соответствует

конкретному индексу, что позволяет нам применять индексирование для получения данных из последовательности. В большинстве языков программирования отсчет индексов начинается слева. Так как мы знаем, что списки являются типичной моделью данных последовательностей, в качестве примера в этом разделе будет использоваться приведенный ниже объект `list`. В этом объекте `list` хранятся ежемесячные доходы книжного магазина за последний год:

```
revenue_by_month = [95, 100, 80, 93, 92, 110, 102, 88, 96, 98, 115, 120]
```

Допустим, вы хотите получить данные за ноябрь. Как это сделать? В этом разделе вы научитесь использовать положительные и отрицательные индексы для получения данных из последовательностей. Как будет показано, Python также поддерживает отсчет индексов справа (отрицательное индексирование), и вы узнаете, когда следует использовать положительное или отрицательное индексирование.

4.2.1. Положительное индексирование от начала списка

В разделе 4.1 было продемонстрировано использование положительного индексирования в срезах для создания подпоследовательностей. Как и в большинстве других языков, мы получаем отдельные элементы на основании их индексов, начиная с нуля для крайнего левого элемента. В этом разделе вы научитесь использовать положительное индексирование. Разумеется, этот механизм знаком большинству читателей, так что описание будет кратким. Например, с положительным индексированием для получения данных за январь и за второй квартал можно использовать следующие команды:

```
revenue_jan = revenue_by_month[0]
revenue_season2 = revenue_by_month[3:6]
```

Первый элемент имеет индекс 0

Получает четвертый, пятый и шестой элементы с индексами 3, 4 и 5

Что делать, если вы хотите получить элементы в конце списка? Допустим, вам потребовалось получить данные за ноябрь и за четвертый квартал. Первым приходит в голову такое решение:

```
revenue_nov = revenue_by_month[10]
revenue_season4 = revenue_by_month[9:]
```

Ноябрьским данным присвоен индекс 10

Получает данные, начиная с индекса 9 и до последнего элемента

С положительным индексированием и слайсингом для получения нужных элементов нам приходится считать до 10 и 9 соответственно. Конечно, подсчет индексов возможен, если списки состоят из десятков элементов. Но если элементов становится больше, то получение индексов отсчетом от начала повышает риск ошибок. Не существует ли более удобного решения? В следующем разделе рассматривается одно из таких решений: *отрицательное индексирование*.

4.2.2. Отрицательное индексирование от конца списка

Python поддерживает отрицательное индексирование. Считать можно не только слева направо, но и справа налево. В этом разделе вы увидите, как отрицательное индексирование улучшает удобочитаемость при получении элементов, расположенных ближе к концу последовательности.

В типичной ситуации индексы положительного индексирования начинаются с 0 (для первого элемента) и заканчиваются длиной списка, уменьшенной на 1 (для последнего элемента). При отрицательном индексировании `-1` используется для последнего элемента, `-2` — для предпоследнего, `-3` — для предпредпоследнего и т. д. Таким образом, первый элемент имеет отрицательный индекс `-len(list)`. Отрицательное индексирование — замечательное решение, так как оно интуитивно понятно. В повседневной жизни мы привыкли считать от 1 с поправкой на отрицательный знак. На рис. 4.3 представлены как положительные, так и отрицательные индексы для списка.

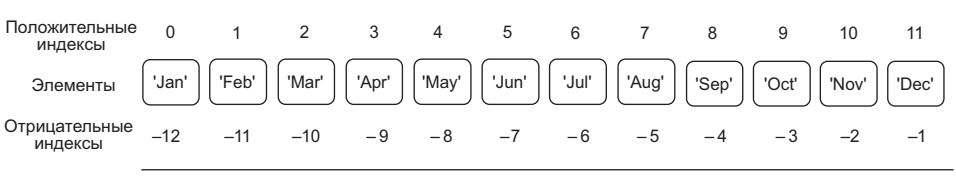


Рис. 4.3. Положительные и отрицательные индексы в списке. Положительные индексы начинают отсчет слева с начальным значением 0, а отрицательные — справа с начальным значением -1

Применим механизм отрицательного индексирования к извлечению дохода за ноябрь и за четвертый квартал:

```
revenue_nov_neg = revenue_by_month[-2]   <-- Ноябрю соответствует индекс –2
assert revenue_nov == revenue_nov_neg

revenue_season4_neg = revenue_by_month[-3:]   <-- Четвертый квартал состоит
assert revenue_season4 == revenue_season4_neg     из трех последних элементов списка
```

Как показывает этот пример, мы получаем такие же результаты, как при положительном индексировании. Однако у отрицательного индексирования есть три преимущества:

- *Экономия времени*. Достаточно отсчитать небольшое количество элементов от конца списка.
- *Прямолинейность*. Элементы отсчитываются справа, а n-й элемент имеет отрицательный индекс $-n$. При этом не нужно вносить поправку из-за отсчета с 0, как для положительных индексов, а достаточно изменить знак числа: `2 -> -2`.

- *Наглядность*. Отрицательный индекс однозначно показывает читателю кода, что элементы читаются от конца списка, и это самое важное преимущество отрицательного индексирования.

Таким образом, когда вы хотите получить элементы, находящиеся ближе к концу последовательности, лучше использовать отрицательное индексирование.

УДОБОЧИТАЕМОСТЬ Когда вы пытаетесь получить элементы в конце последовательности, отрицательные индексы хорошо заметны при чтении кода.

4.2.3. Объединение положительных и отрицательных индексов

Положительные и отрицательные индексы не являются взаимоисключающими. Для каждого элемента последовательности положительный и отрицательный индекс эквивалентны, несмотря на их различающиеся значения, и ссылаются на один и тот же элемент, что позволяет нам объединять обе разновидности индексов там, где это возможно.

Допустим, вы хотите получить элементы из середины списка. При определении сегмента можно использовать положительное индексирование вместе с отрицательным:

```
revenue_middle = revenue_by_month[1:-1]  ◄── Обращается к данным с февраля по ноябрь
print(revenue_middle)

# Вывод: [100, 80, 93, 92, 110, 102, 88, 96, 98, 115]
```

4.2.4. Обсуждение

При получении элементов, расположенных в конце последовательности, следует использовать отрицательные индексы. Возможно, этот раздел показался некоторым читателям скучным, но я решил включить его в книгу, потому что модели последовательностей данных используются во многих проектах. Важно сформировать привычку использовать отрицательные индексы для обозначения последних элементов в последовательности. Это не только упрощает поиск индекса последних элементов, но и явно сигнализирует читателю, что код работает с элементами в конце последовательности. А удобочитаемость играет ключевую роль в любой кодовой базе.

4.2.5. Задача

Джеффри — студент, входящий в школьную команду робототехники. Недавно он узнал о положительном индексировании последовательностей. Он знает, что длина списка может использоваться для вычисления положительного индекса элемента по направлению к концу, и хочет написать код для вычисления дохода за ноябрь; этот код вычисляет длину списка. Сможете ли вы помочь ему?

ПОДСКАЗКА Помните, что положительное индексирование начинается с 0. Следовательно, положительный индекс последнего элемента отличается на 1 от длины последовательности.

4.3. КАК ИСКАТЬ ЭЛЕМЕНТЫ ПОСЛЕДОВАТЕЛЬНОСТИ ПО КРИТЕРИЮ

В разделах 4.1 и 4.2 были описаны такие общие возможности типов последовательностей, как слайсинг и индексирование. При работе с последовательностями иногда возникает необходимость найти какой-то конкретный элемент. Например, при работе с объектом `list`, содержащим задачи, нужно узнать, связана ли какая-либо задача с прохождением опроса. Или надо выяснить, включает ли текстовое описание задачи строку «курсовая». Если говорить в более общем смысле, поиск элемента последовательности по критерию принадлежит к числу распространенных операций, и в этом разделе рассмотрены некоторые подходы к решению этой задачи.

4.3.1. Проверка вхождения элемента

Поиск элемента последовательности по критерию начинается с проверки вхождения элемента. Этой теме посвящен текущий раздел.

Во многих языках программирования, таких как JavaScript, проверка присутствия элемента в последовательности реализуется именованным методом: `list.contains(item)`, `list.includes(item)` или что-нибудь в этом роде. Однако в Python используется другой подход к решению задачи с ключевым словом `in`. Общий синтаксис таков: *элемент* `in` *последовательность*, при этом возвращается логическое значение — признак присутствия или отсутствия элемента в последовательности. Приведу несколько примеров:

```
assert (8 in [1, 2, 3, 4, 5]) == False

assert ('cool' in 'Python is cool') == True

assert (404 in (404, 'Page Not Found')) == True
```

Круглые скобки обязательны. В противном случае сначала будет вычислена проверка равенства ==

== True можно опустить. Я включаю это сравнение для ясности

Функциональность *элемент* `in` *последовательность* удобна тогда, когда вас интересует только факт присутствия конкретного элемента в последовательности. Однако бинарных признаков `True` или `False` недостаточно в ситуации, когда необходимо знать точный индекс элемента. Допустим, вам потребуется узнать индекс искомого (якорного) элемента и получить подпоследовательность, начинающуюся с якоря. В таком случае необходимо использовать метод `index`, описанный в следующем разделе.

4.3.2. Использование метода index для поиска элемента

Еще одна общая характеристика последовательностей — поддержка метода `index`, который возвращает индекс элемента в последовательности. В этом разделе вы научитесь пользоваться методом `index` для нахождения конкретного элемента.

Следующий фрагмент кода демонстрирует примеры использования разных типов последовательностей. Как видно из кода, все последовательности поддерживают метод `index`:

```
[1, 2, 3, 4, 5].index(4)
# Вывод: 3

(404, 'Page Not Found').index('Page Not Found')
# Вывод: 1

'Python is cool'.index('cool')
# Вывод: 10
```

По умолчанию `index` использует положительные индексы, начинающиеся с 0. Если проверяемый элемент входит в последовательность, то все работает так, как ожидалось, и вы получаете индекс элемента.

> **ПРИМЕЧАНИЕ** Если в последовательности присутствуют дубликаты, метод `index` возвращает индекс первого совпавшего элемента.

В вызове метода `index` есть один нюанс, который многие люди не осознают: иногда может оказаться, что элемент отсутствует в последовательности, как, например, в этом случае:

```
[1, 2, 3, 4, 5].index(8)
# ERROR: ValueError: 8 is not in list
```

При выдаче исключения (в данном случае исключение `ValueError`) программа аварийно завершается, если исключение не будет обработано. И хотя обработка исключений будет рассматриваться в главе 12, я приведу краткое решение с конструкцией `try...except...`:

```
def process_item_try(item):
    try:
        item_index = the_list.index(item)
    except ValueError:
        # Действия в случае отсутствия элемента
    # Что-то сделать с item_index
```

Этот фрагмент кода также можно записать по-другому, чтобы выполнить проверку присутствия перед вызовом `index`:

```
def process_item_check_first(item):
    if item in the_list:
        item_index = the_list.index(item)
```

```
        # Что-то сделать с item_index
    else:
        # Действия в случае отсутствия элемента
```

На первый взгляд оба решения делают одно и то же, но я предпочитаю первый вариант, потому что он эффективнее второго. При использовании метода `index` Python приходится обходить последовательность и проверять каждый элемент для выявления совпадения, а эта операция занимает много времени. Аналогичным образом, когда вы узнаете, присутствует ли элемент в последовательности, Python приходится снова обходить последовательность. Следовательно, при использовании решения с `process_item_check_first` затраты времени удваиваются, потому что оно требует двух проходов по сравнению с одним проходом в решении с `process_item_try`. Таким образом, для короткой последовательности подойдет любой вариант, но если последовательность длинная, лучше использовать первое решение.

EAFP против LBYL

В Python всеобщим уважением пользуется принцип EAFP (Easier to Ask for Forgiveness Than Permission, то есть «Проще попросить прощения, чем разрешения»). В этом принципе вы используете `try...except...` в предположении, что все должно сработать как надо. Если что-то пойдет не так, ошибка обрабатывается соответствующим образом (мы «просим прощения»). Также существует другой принцип, обозначаемый LBYL («Look Before You Leap», то есть «Смотри, прежде чем прыгнуть»). Этот принцип более ярко выражен в других языках программирования, таких как C. Согласно этому принципу вы сначала «смотрите» — проверяете условие (вероятно, при помощи инструкции `if`) и только потом «прыгаете» — применяете операцию, если условие выполняется.

4.3.3. Поиск подстроки в строке

Строки как тип последовательностей тоже поддерживают метод `index`, как было показано в разделе 4.3.2. Более того, мы обеспечили обработку возможного исключения `ValueError`, связанного с методом `index`. Однако в отличие от других типов последовательностей, строки содержат два дополнительных метода поиска элементов по критерию: `find` и `rfind`.

Оба метода возвращают индекс искомой подстроки. Они лучше метода `index` тем, что если подстрока не входит в строку, они возвращают `-1` вместо того, чтобы выдавать исключение `ValueError`. Поэтому я рекомендую использовать `find` или `rfind` при поиске подстроки, как показано в следующем примере:

```
def find_string(substr):
    str_index = the_str.find(substr)
    if str_index >= 0:
```

```
        # Что-то сделать с str_index
    else:
        # Действия в случае отсутствия подстроки
```

Заметим, что метод `find` доступен только для строк. Он не может использоваться с другими типами последовательностей, хотя я не вижу никаких технических затруднений в том, чтобы реализовать эту возможность с моделями и других последовательностей, помимо `str`.

ОБРАТИТЕ ВНИМАНИЕ Метод `find` может использоваться только со строками, но не с другими типами последовательностей.

4.3.4. Поиск экземпляра пользовательского класса в списке

Когда проект разрастается, мы начинаем применять пользовательские (кастомные) классы как модели данных. Так, списки используются для хранения нескольких экземпляров пользовательского класса. Весьма вероятно, что вам потребуется выяснить, существует ли в списке конкретный экземпляр. В этом разделе вы узнаете, как найти экземпляр пользовательского класса в списке.

Допустим, в таск-менеджере мы используем объект `list` для хранения задач на день. Код из листинга 4.2 станет отправной точкой для дальнейшей разработки. Упрощенный класс `Task` содержит минимальную реализацию. Для подтверждения работоспособности концепции в список включены четыре экземпляра.

Листинг 4.2. Создание списка объектов пользовательских классов

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

tasks = [
    Task("Laundry", 3),
    Task("Museum", 4),
    Task("Homework", 5),
    Task("Ticket", 2)
]
```

Список этих задач выводится в интерфейсе нашего приложения. Одна из возможных функций приложения может выделять цветом строку с задачей, соответствующей критерию фильтрации, например степени срочности 5. Чтобы реализовать эту функциональность, необходимо знать индекс задачи с нужной степенью срочности. Понятно, что метод `index` использовать не удастся, — задача с нужной степенью срочности неизвестна заранее. Следовательно, придется искать другое решение. Вы можете перебрать весь список для поиска потенциального совпадения. В следующем фрагменте представлено работоспособное решение:

```
needed_urgency = 5
needed_task_index = None

for task_i in range(len(tasks)):
    task = tasks[task_i]
    if task.urgency == needed_urgency:
        needed_task_index = task_i
        break

print(f"Task Index: {needed_task_index}")
# Вывод: Task Index: 2
```

Альтернативное решение представлено в разделе 4.3.6

Использует break для выхода из цикла for

Цикл `for` используется для перебора списка с целью проверки атрибута степени срочности каждого экземпляра и сравнения его с нужной степенью. При обнаружении задачи инструкция `break` (см. раздел 5.4.1) используется для выхода из цикла `for` и завершения поиска. С найденным индексом можно обновить интерфейс приложения и выделить соответствующую строку задачи.

ОСНОВНЫЕ ПОНЯТИЯ Инструкция break моментально выходит из цикла `for`.

4.3.5. Обсуждение

Вызов `index` для последовательности возвращает индекс только первого подходящего элемента, поэтому следует учитывать, что последовательность может содержать другие подходящие элементы. Так как метод `index` выдает исключение `ValueError`, если элемент отсутствует в последовательности, мы можем использовать инструкцию `try...except...` (раздел 12.3) для обработки исключения. И хотя мы можем проверить присутствие конкретного элемента, принцип LBYL требует двух обходов последовательности, что создает лишние затраты времени. А значит, стоит воспользоваться принципом EAFP для улучшения производительности.

СОПРОВОЖДАЕМОСТЬ Старайтесь применять принцип EAFP там, где это возможно, потому что в общем случае он более эффективен, чем LBYL.

4.3.6. Задача

В примере с обнаружением объекта пользовательского класса задача с нужной степенью срочности имела индекс 2; это был объект `Task("Homework", 5)`. Что произойдет при выполнении кода `tasks.index(Task("Homework", 5))`? Получите ли вы в результате индекс `2`?

ПОДСКАЗКА Хотя некоторые объекты вроде бы содержат одинаковые данные, это разные объекты, имеющие разные адреса памяти. Вы можете воспользоваться функцией `id` для объяснения результатов.

4.4. КАК РАСПАКОВАТЬ ПОСЛЕДОВАТЕЛЬНОСТИ

Так как кортежи являются неизменяемыми контейнерами данных, они могут использоваться для хранения множественных объектов, содержимое которых не должно изменяться. Вы уже знаете, как применять индексирование и слайсинг (разделы 4.2 и 4.3) для получения элементов из кортежа по одному или группами.

```
task = (1001, "Laundry", 5)

task_id = task[0]
task_title = task[1]
task_urgency = task[-1]
```

В этом примере три отдельных присваивания используются для создания трех переменных, каждая из которых соответствует одному элементу кортежа `task`. Если объект кортежа содержит больше элементов, придется включать дополнительные присваивания; это рутинная работа, которая загромождает код и затрудняет его чтение. Нет ли более эффективного способа доступа к нескольким элементам через соответствующие переменные?

Задача решается с помощью *распаковки* (unpacking). Применительно к кортежам это обычно называется *распаковкой кортежей*. Суть в том, что создание кортежа для хранения данных можно на концептуальном уровне рассматривать как процесс упаковки информации. Поэтому вполне логично, что обратный процесс — извлечение элементов — называется распаковкой. В этом разделе вы освоите этот важный прием, ориентируясь в первую очередь на объекты кортежей. Однако заметим, что распаковка доступна не только для кортежей; она также поддерживается для любых итерируемых объектов, включая типы последовательностей.

4.4.1. Распаковка коротких последовательностей с однозначным соответствием

Если вы работаете с кортежами, содержащими небольшое количество элементов, и хотите использовать все элементы, используйте однозначную (one-to-one) распаковку, при которой каждый элемент присваивается соответствующей переменной:

```
task = (1001, "Laundry", 5)
task_id, task_title, task_urgency = task

print(task_id, task_title, task_urgency)
# Вывод: 1001 Laundry 5

user_data = ("python_user", 35, "male")
username, age, gender = user_data
print(username, age, gender)
# Вывод: python_user 35 male
```

В этом примере однозначной распаковки одна строка кода использовалась для создания нескольких переменных, соответствующих каждому элементу объекта `tuple`. Обратите внимание: в предыдущих примерах сначала создавались кортежи, это отражает реальную ситуацию, в которой вы получаете объекты `tuple`, созданные в других частях проекта.

С однозначной распаковкой тесно связан прием *множественного присваивания* (multiple assignment), при котором несколько переменных создаются одним оператором присваивания (знак `=`):

```
x0, y0 = (90, 20)
(x1, y1) = 90, 20
(x2, y2) = (90, 20)

assert x0 == x1 == x2 == 90
assert y0 == y1 == y2 == 20
```

В приведенном фрагменте кода продемонстрировано несколько разновидностей множественного присваивания. Несмотря на внешние различия, они делают одно и то же. В правой части создаются объекты `tuple`, а в левой части содержится такое же количество переменных, чтобы элементы распаковывались однозначно. Кроме того, в этих присваиваниях следует обратить внимание на то, что круглые скобки при создании и распаковке кортежей не обязательны. В следующем фрагменте кода приведена перестановка, которая дополняет приведенные ранее примеры со строками:

```
x3, y3 = 90, 20

assert x3 == 90
assert y3 == 20
```

> **УДОБОЧИТАЕМОСТЬ** Используйте множественное присваивание только в том случае, если переменные тесно связаны. Если переменные служат разным целям, то присваивание лучше выполнять в разных строках кода.

4.4.2. Выборка смежных элементов с использованием выражений со звездочкой

В предыдущем разделе для выборки нескольких элементов была использована однозначная распаковка. Этот механизм хорошо подходит для кортежей с малым количеством элементов. В кортежах с множеством элементов нередко требуется загрузить часть элементов в отдельные переменные, а некоторые смежные элементы — в одну переменную. В этом разделе показано, как это делается.

Допустим, вы проводите соревнования по гимнастике, и каждого участника оценивают 8 судей. Чтобы вычислить итоговую оценку игрока, мы отбросим наибольшую и наименьшую оценку, а затем вычислим среднее для шести оставшихся

оценок. Мы сохраняем данные об оценках каждого участника: наименьшую, среднюю, наибольшую и итоговую. Чтобы упростить пример, допустим, что оценки уже были отсортированы по возрастанию. Конечно, для генерирования данных можно воспользоваться индексированием:

```
player_scores = [6.1, 6.5, 6.8, 7.1, 7.3, 7.6, 8.2, 8.9]

lowest0 = player_scores[0]
middles0 = player_scores[1:-1]
highest0 = player_scores[-1]

final0 = sum(middles0) / len(middles0)
```

Вместо механизма распаковки, который рассматривается позже в этом разделе, мы используем несколько строк кода и создаем переменные одну за другой. Такое решение не является питоническим способом создания нескольких переменных по данным последовательностей. К сожалению, при попытке решить проблему применением синтаксиса однозначной распаковки мы сталкиваемся с проблемой:

```
lowest1, middles1, highest1 = player_scores
# ERROR: ValueError: too many values to unpack (expected 3)
```

Смысл сообщения об ошибке вполне ясен: слишком много значений для распаковки. Рассмотрим этот момент повнимательнее. Слева указаны три переменные, так что Python ожидает, что из кортежа будут распакованы три элемента. Но объект `tuple` содержит восемь элементов, что приводит к несоответствию. Как решить проблему? На помощь приходит выражение со звездочкой:

```
lowest2, *middles2, highest2 = player_scores  <-- Использует выражение со звездочкой
final2 = sum(middles2) / len(middles2)

assert lowest0 == lowest2 == player_scores[0]
assert middles0 == middles2 == player_scores[1:-1]
assert highest0 == highest2 == player_scores[-1]
```

Обратите внимание на некоторые характеристики выражений со звездочкой.

- Выражение использует звездочку * в качестве префикса переменной (`*var_name`). Все элементы, не захваченные другими переменными, захватываются этой переменной. В данном случае первый и последний элементы попадают в `lowest2` и `highest2` соответственно. Шесть элементов в середине захватываются `middles2`.
- Выражение со звездочкой строит объект `list` с захваченными элементами независимо от типа данных исходной последовательности. За этим эффектом можно наблюдать при помощи объекта `str`, как показано в следующем фрагменте. Было бы ошибочным полагать, что переменная `b` представляет собой объект `str`, содержащий все символы в середине:

```
a, *b, c = "abcdefg"
assert b == ['b', 'c', 'd', 'e', 'f']
```

- Количество захваченных элементов в объекте `list` может быть равно 0. Если все элементы распакованы с правильным количеством переменных и остается 0 неучтенных элементов, выражение со звездочкой производит пустой список. Посмотрите на этот эффект:

```
first_score, *scores, last_score = [9.1, 8.9]
assert scores == []
```

- В одном присваивании может использоваться только одно выражение со звездочкой. Попытка использования двух выражений со звездочкой является синтаксической ошибкой. Причина проста: выражение со звездочкой должно захватывать все элементы, которые еще не были захвачены, так что при использовании двух выражений со звездочкой невозможно определить, какое из них должно захватывать те или иные элементы:

```
score0, *scores0, *scores1, score1 = [9.1, 8.8, 9.2, 7.7, 8.4]
# ERROR: SyntaxError: multiple starred expressions in assignment
```

4.4.3. Пометка нежелательных элементов подчеркиваниями

Ранее мы выясняли, как распаковать кортеж или список для обращения к одиночным или последовательным элементам. При любой распаковке необходимо предоставить правильное количество переменных (с выражением со звездочкой, если необходимо), соответствующих элементам последовательности. Но некоторые распакованные элементы могут не использоваться в программе. В таком случае следует указать при распаковке символы подчеркивания.

```
def update_status(t_id, t_status):
    # Найти задачу в базе данных по task_id и обновить ее статус
    pass

task = (1001, "Laundry", "Wash clothes", "completed")
task_id, task_title, task_desc, task_status = task

update_status(task_id, task_status)
```

Вспомогательная функция для обновления базы данных

API возвращает кортеж из четырех элементов

Полностью распаковывает объект tuple

В приведенном фрагменте кода мы распаковали объект `tuple` таким образом, что все элементы связываются с соответствующими переменными. Однозначная распаковка передает важный сигнал для читателя кода: далее в программе будет использоваться каждый распакованный элемент. Но как показано в коде, работать мы будем только с идентификатором и статусом задачи.

Таким образом, полная распаковка, включающая присваивание переменных, которые нам не понадобятся, вводит читателя в заблуждение. Чтобы устранить такое искажение, заменим ненужные элементы символами подчеркивания:

```
task_id, _, _, task_status = task
```

Смысл прост: если какие-то переменные нам не понадобятся, не нужно присваивать им содержательные имена. Отмечу несколько особенностей использования символов подчеркивания при распаковке:

- Вы можете использовать столько символов подчеркивания, сколько потребуется. В нашем примере объект `tuple` содержит четыре элемента. Так как для нас представляют интерес только два элемента, при распаковке указываются два символа подчеркивания, а также имена `task_id` и `task_status`.
- Символы подчеркивания являются действительными именами переменных. Их применение не ограничивается распаковкой. У разработчиков Python принято использовать символы подчеркивания для обозначения ненужных переменных. И хотя обозначение показывает, что эти переменные не нужны, к ним можно обратиться при желании. В нашем примере переменная `_` содержит описание задачи, потому что более раннее присваивание названия задачи (первый символ `_`) было перезаписано.
- В выражениях со звездочкой символы `*` и `_` могут объединяться. Пример:

```
task = (1001, "Laundry", "Wash clothes", "completed")
task_id, *_, task_status = task
```

Объединение * с _ в выражении со звездочкой

УДОБОЧИТАЕМОСТЬ При распаковке последовательностей исключайте ненужные элементы при помощи символов подчеркивания; они показывают, что вы не собираетесь использовать эти элементы.

4.4.4. Обсуждение

Распаковка — самый наглядный способ извлечения отдельных или смежных элементов последовательности. Разработчик должен хорошо понимать разные способы распаковки. Выше я показал, как работает распаковка, в основном на примере кортежей. Но механизм распаковки может применяться к любым итерируемым объектам. После того как вы узнаете больше об итерируемых объектах из главы 5, попробуйте использовать механизм распаковки с любыми итерируемыми объектами.

4.4.5. Задача

Дэнни работает над проектом, в котором он применяет распаковку для извлечения данных из объектов `list`. Его данные отличаются тем, что объекты `list` имеют два уровня, скажем, `[1, (2, 3), 4]`. Как в одной строке кода распаковать оба уровня и извлечь четыре числа в четыре переменные?

ПОДСКАЗКА Для создания уровней во время распаковки можно использовать круглые скобки.

4.5. КОГДА СЛЕДУЕТ ВЫБИРАТЬ ДРУГИЕ МОДЕЛИ ДАННЫХ, ПОМИМО СПИСКОВ И КОРТЕЖЕЙ

Несомненно, универсальные возможности списков и кортежей делают их неплохими кандидатами на роль контейнеров данных во многих типичных ситуациях. Но когда вы начинаете заниматься специфическими проектами, выясняется, что списки и кортежи уже не столь идеальны. А значит, следует непредвзято относиться к альтернативным структурам данных, которые во многих практических сценариях оказываются правильным вариантом. В этом разделе рассматриваются некоторые стандартные ситуации с рекомендуемыми альтернативами.

4.5.1. Использование множеств в ситуациях с проверкой принадлежности

Часто требуется узнать, содержит ли контейнер данных конкретный элемент, — эта функциональность называется *проверкой принадлежности* (membership checking). Вы уже знаете, что при работе со списками и кортежами можно воспользоваться либо конструкцией `item in the_list` для проверки принадлежности, либо методом `index` для косвенной проверки того, что список содержит конкретный элемент. Обратите внимание: вариант с `index` менее желателен, потому что при отсутствии элемента в списке выдается исключение `ValueError`.

Хотя списки поддерживают проверку принадлежности, лучше рассмотреть возможность использования множеств, если такая проверка актуальна для вашего приложения. Как показано в разделе 3.5, Python требует, чтобы все элементы множества были уникальными, потому что во внутренней реализации множества реализуются на базе хеш-таблиц. Такая реализация обеспечивает значительный выигрыш в виде постоянного времени поиска, называемого временной сложностью O(1). С другой стороны, время поиска при проверке принадлежности линейно зависит от длины списка, потому что Python приходится обходить последовательность для поиска потенциальных совпадений. Чем больше элементов в списке, тем больше времени занимает обход. Следовательно, если проверка принадлежности выполняется часто, лучше использовать множества.

> **ВОПРОС** А вы помните, что для хранения данных множеств используется реализация на базе хеш-таблиц (раздел 3.5)?

4.5.2. Деки для обеспечения принципа FIFO

В некоторых приложениях доступ к данным должен осуществляться по принципу FIFO (First In, First Out, то есть «первым пришел, первым вышел»). FIFO означает, что элементы, которые были первыми добавлены в последовательность, будут первыми извлечены из нее. В этом разделе представлена более совершенная модель для ситуаций, в которых важен принцип FIFO.

Допустим, вы строите корпоративную чат-систему для общения с клиентами. В рабочее время клиенты оформляют заказы, и мы используем список для отслеживания порядка последовательности заказов. Будет логично сначала связывать со службой поддержки клиентов, оформивших заказ раньше других; это требование представляет принцип FIFO в приложении. В одной из возможных реализаций используются списки, как показано в листинге 4.3.

Листинг 4.3. Построение системы клиентской очереди на базе списков

```
clients = list()

def check_in(client):
    clients.append(client)
    print(f"in: New client {client} joined the queue.")

def connect_to_associate(associate):
    if clients:
        client_to_connect = clients.pop(0)
        print(f"out: Remove {client_to_connect}, connecting to
➥ {associate}.")
    else:
        print("No more clients are waiting.")
```

Присоединяет новый элемент в конец списка

Проверяет, присутствуют ли элементы в списке. Если список пуст, вызов pop приводит к ошибке IndexError

Удаляет первый элемент из списка

В этом фрагменте функция `check_in` добавляет нового клиента в конец очереди — объекта `list` с именем `clients`. Когда сотрудник поддержки освобождается, с ним связывается первый клиент в очереди. Для получения первого клиента используется метод `pop` объектов `list`. Этот метод не только возвращает первый элемент списка, но и удаляет его.

Удаление элемента в начале объекта `list` заслуживает особого внимания. Во внутренней реализации Python сдвигает каждый элемент списка, чтобы заполнить освободившуюся память; это затратная операция с временной сложностью O(n). С учетом ее значительной сложности стоит рассмотреть альтернативное решение.

Тип данных `deque` представляет двустороннюю очередь, или *дек* (deque — double-ended queue). Благодаря двустороннему режиму дек поддерживает вставку и удаление с обоих концов, что делает его идеальным типом данных для реализации системы управления клиентами, требующей принципа FIFO. Как упоминалось ранее, вызов метода `pop` для объекта `list` является затратной операцией как по времени, так и по памяти. С другой стороны, так как дек открыт с обоих концов, удаление крайнего левого элемента из дека становится вычислительно тривиальной операцией. На рис. 4.4 сравниваются списки и деки.

Проведем прямое сравнение между списками и деками для этой операции. Ниже приведено упрощенное решение для удаления первого элемента из очереди ожидания. Обратите внимание на использование лямбда-функции в листинге 4.4 (глава 7).

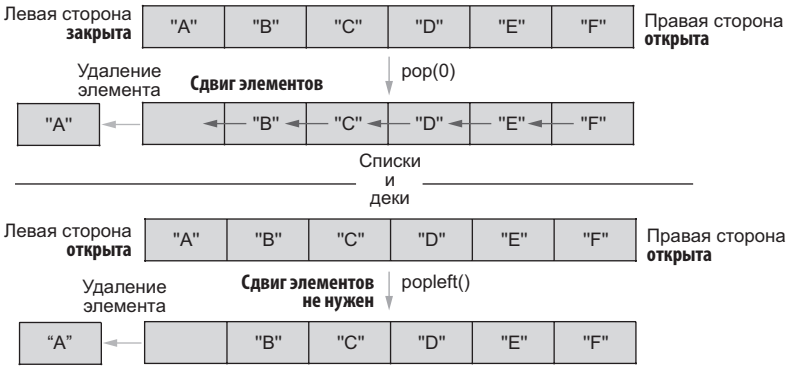


Рис. 4.4. Удаление первого элемента в списке и деке. Удаление крайнего левого элемента в списке требует сдвига всех остальных элементов, из-за чего оно обладает временной сложностью O(n), тогда как удаление крайнего левого элемента в деке не требует никаких действий с остальными элементами, из-за чего оно становится операцией со сложностью O(1)

Листинг 4.4. Сравнение быстродействия деков и списков

```
from collections import deque
from timeit import timeit

def time_fifo_testing(n):
    integer_l = list(range(n))
    integer_d = deque(range(n))
    t_l = timeit(lambda : integer_l.pop(0), number=n)
    t_d = timeit(lambda : integer_d.popleft(), number=n)
    return f"{n: >9} list: {t_l:.6e} | deque: {t_d:.6e}"

numbers = (100, 1000, 10000, 100000)
for number in numbers:
    print(time_fifo_testing(number))

# Вывод выглядит примерно так:
      100 list: 6.470000e-05 | deque: 3.790000e-05
     1000 list: 7.637000e-04 | deque: 3.435000e-04
    10000 list: 1.805050e-02 | deque: 2.134700e-03
   100000 list: 1.641030e+00 | deque: 1.336000e-02
```

Тип данных deque доступен в модуле collections стандартной библиотеки

Функция timeit вычисляет среднее время выполнения выражения

Метод popleft извлекает первый элемент в начале дека

Выигрыш по быстродействию в этом тривиальном примере, использующем деки вместо списков, составляет два порядка при 100 000 элементов. В корпоративных приложениях такая оптимизация по одному аспекту может оказаться критичным для улучшения общего впечатления от приложения у пользователя. Важно заметить, что использование типа `deque` не требует никаких сложных реализаций. Так почему бы не извлечь пользу из повышения быстродействия без каких-либо затрат, просто за счет выбора встроенного типа данных? В листинге 4.5 приведена обновленная реализация на базе дека.

Листинг 4.5. Реализация системы клиентской очереди на базе дека

```
from collections import deque

clients = deque()

def check_in(client):
    clients.append(client)
    print(f"in: New client {client} joined the queue.")

def connect_to_associate(associate):
    if clients:
        client_to_connect = clients.popleft()
        print(f"out: Remove {client_to_connect}, connecting to
➥   {associate}.")
    else:
        print("No more clients are waiting.")
```

4.5.3. Обработка многомерных данных средствами NumPy и Pandas

До настоящего момента мы занимались линейными последовательными структурами данных, такими как списки, кортежи и строки. Однако в реальной жизни данные могут иметь многомерный формат — это относится, например, к графике и видео. Скажем, графические изображения могут представляться в математическом виде как три слоя (красный, зеленый и синий) двумерных сеток пикселей. Попытка использовать базовые модели данных для представления данных высокой размерности оборачивается сущим кошмаром. К счастью, модель распространения с открытым кодом, выбранная для Python, способствовала разработке множества сторонних библиотек и пакетов для обработки многомерных крупномасштабных наборов данных. Таким образом, вместо списков стоит поискать альтернативы, разработанные специально для задач с высокой вычислительной нагрузкой.

Например, если вам приходится работать с большими объемами числовых данных, рассмотрите возможность использования массивов NumPy — базового типа данных, реализованного в пакете NumPy. Заметим, что в пакете доступно множество сопутствующих операций: реструктуризация данных, преобразования и различные арифметические операции.

Если вам нужно работать с данными в стиле электронных таблиц со смешанными типами данных (строками, датами, числами и т. д.), обратитесь к `DataFrame` — одному из основных типов данных, реализованных в пакетах pandas. Если вы занимаетесь машинным обучением, вам придется работать с *тензорами* — одним из важнейших типов данных во фреймворках машинного обучения, таких как TensorFlow и PyTorch. Если ваше приложение обрабатывает большие объемы многомерных данных, особенно в виде числовых значений, присмотритесь к этим сторонним библиотекам со специализированными типами данных и методами, упрощающими вашу жизнь.

4.5.4. Обсуждение

Списки и кортежи — полезные типы последовательностей для хранения упорядоченных элементов. Однако теперь вам также известны основные альтернативные модели данных. Конечно, приведенный выше перечень таких моделей ни в коей мере не полон. Я всего лишь хочу пояснить, что следует без предвзятости подходить к выбору модели данных. Решение должно определяться конкретными бизнес-потребностями.

> **СОПРОВОЖДАЕМОСТЬ** Всегда выбирайте подходящую модель данных для своих целей. Использование неподходящей модели значительно усложнит сопровождение вашего проекта.

Подведем итог: выбор модели данных должен происходить с учетом специфики потребностей для конкретных компонентов вашего приложения. Иначе говоря, ваше приложение может использовать сколько угодно разных моделей данных, при этом каждая модель выбирается под конкретную цель. На рис. 4.5 изображена общая схема выбора модели в зависимости от конкретных потребностей.

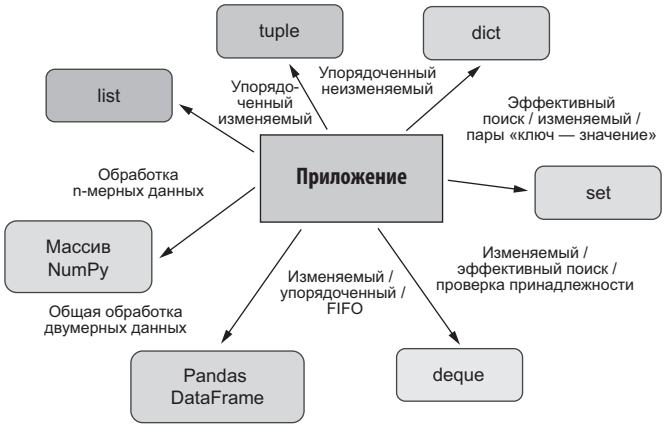


Рис. 4.5. Выбор моделей данных зависит от конкретных потребностей компонентов приложения

4.5.5. Задача

Эмма — начинающий аналитик данных, и она еще только изучает Python. Она понимает, что может использовать списки для хранения одномерных данных, например список `list` с числами. Но в ее проектах есть списки, встроенные в другие объекты списков для хранения двумерных данных, которые образуют электронную таблицу из четырех строк и трех столбцов:

```
numbers = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

Как ей поступить, если потребуется умножить каждый элемент на 3? Понятно, что в такой реализации это утомительная и рутинная работа. Сможете ли вы предложить другую модель данных, которая лучше подходит для данной цели?

ПОДСКАЗКА Тип array в NumPy специализирован для выполнения операций с многомерными числовыми данными.

ИТОГИ

- Слайсинг, или нарезка, может использоваться для получения подпоследовательностей. Для среза можно указать начальный индекс, конечный индекс и приращение. Учтите, что существуют разные варианты слайсинга, в том числе без указания начального или конечного индекса.
- Срезы используются для создания подпоследовательностей из данных последовательностей, а интервалы — для перебора в заданном интервале с заданным приращением.
- Данные последовательностей включают изменяемые и неизменяемые типы. С изменяемыми типами (как `list` и `bytearray`) можно выполнять операции замены, расширения, сжатия и удаления подпоследовательностей.
- В последовательности каждому элементу соответствует индекс, обозначающий его позицию. Положительные индексы начинаются с 0 (левый элемент) и увеличиваются слева направо с приращением 1, а отрицательные индексы начинаются с –1 (правый элемент) и увеличиваются справа налево.
- Чтобы ваш код лучше читался, старайтесь использовать положительные индексы для обращения к элементам в начале последовательности и отрицательные индексы для элементов в конце последовательности.
- Вы должны знать разные способы проверки присутствия элемента в последовательности, а также учитывать ограничения метода `index`. Для поиска подстроки в строке следует использовать метод `find` или `rfind`. Для экземпляров пользовательских классов применяйте перебор для проверки каждого элемента на возможное совпадение.
- Распаковка кортежей — важный механизм извлечения элементов из объектов `tuple`. Он доступен для всех типов последовательностей и других видов итерируемых объектов. Но вы должны знать другие особенности распаковки, включая использование символов подчеркивания и выражений со звездочкой.
- Списки не универсальны. Вам стоит ознакомиться с альтернативными структурами данных, которые могут лучше подходить для конкретных бизнес-задач, такими как массивы NumPy для обработки многомерных числовых данных.

5 Итерируемые объекты и перебор

В этой главе

- ✓ Итерируемые объекты и итераторы
- ✓ Создание контейнеров данных с использованием итерируемых объектов
- ✓ Использование списков, словарей и включений множеств для создания экземпляров
- ✓ Усовершенствование перебора в циклах `for`
- ✓ Инструкции `continue`, `break` и `else` в циклах `for` и `while`

В предыдущих главах неоднократно упоминались итерируемые объекты (iterables), и вам уже известно, что к ним относятся списки, кортежи и многие другие встроенные типы данных. Однако я не привел явное определение концепции итерируемых объектов. Мы говорили, что эти типы данных являются итерируемыми объектами, но не объясняли почему. В этой главе вы получите ответ на этот вопрос и узнаете, как реализовать самые распространенные модели данных (списки и словари) на базе других итерируемых объектов с использованием конструкторов и включений.

Одним из важнейших механизмов Python и других языков программирования является выполнение повторяющихся действий в циклах `for` (или циклах `while`,

хотя циклы `for` встречаются чаще). При каждой итерации одни и те же операции применяются к каждому элементу итерируемых объектов. Существуют различные способы повышения быстродействия циклов `for` с применением встроенных функций (например, `enumerate` и `zip`) и необязательных инструкций, включая `break` и `continue`. Все эти темы рассматриваются в данной главе.

5.1. КАК СОЗДАВАТЬ РАСПРОСТРАНЕННЫЕ КОНТЕЙНЕРЫ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ИТЕРИРУЕМЫХ ОБЪЕКТОВ

Итерируемые объекты вам уже знакомы. В главе 2 рассматривались важные приемы обработки строк, а строки представляют собой итерируемые объекты, состоящие из символов. В главе 3 рассматривались встроенные контейнеры данных, включая списки, кортежи, множества и словари; все они являются итерируемыми объектами, состоящими из отдельных элементов (или пар «ключ — значение»). В главе 4 были описаны общие методы последовательностей, а все типы последовательностей являются итерируемыми объектами. Как видите, итерируемые объекты очень часто встречаются в Python.

В действительности итерируемые объекты являются важным базовым типом, на основе которого реализуются многие встроенные типы данных. Рассмотрим следующий сценарий: в таск-менеджере существуют два отдельных источника данных — для идентификаторов и для названий задач. Требуется создать объект `dict`, состоящий из пар «идентификатор — название»:

```
id_numbers = [101, 102, 103]
titles = ["Laundry", "Homework", "Soccer"]

desired_output = {101: "Laundry", 102: "Homework", 103: "Soccer"}
```

Для получения желаемого результата неопытный программист предложит использовать цикл `for`:

```
desired_output = {}
for item_i in range(len(id_numbers)):
    desired_output[id_numbers[item_i]] = titles[item_i]
```

В более эффективном (на первый взгляд) решении используются включения словарей (раздел 5.2) и функция `zip`:

```
desired_output = {key: value for key, value in zip(id_numbers, titles)}
```

Тем не менее эти решения не идеальны, потому что в них не учитывается тот факт, что `dict`, как и многие встроенные контейнеры данных, напрямую получает итерируемые объекты при создании экземпляров. В этом разделе мы сначала разберемся, что представляют собой итерируемые объекты, а затем перейдем

к обсуждению одного ключевого приема — создания экземпляров распространенных встроенных контейнеров данных на базе итерируемых объектов.

Экземпляры, конструкторы и конструирование

В языках объектно-ориентированного программирования (ООП), включая Python, важнейшими моделями данных являются классы — как встроенные классы (`list`, `dict` и `tuple`), так и пользовательские классы, которые мы создаем в собственных проектах. При создании объекта, принадлежащего классу, например объекта `dict` – `num_dict = dict(one=1, two=2)`, мы говорим о создании *экземпляра* класса; таким образом, `num_dict` является экземпляром класса `dict`. Концепция создания экземпляров применима и к пользовательским классам.

В процессе инстанцирования функция `dict` используется для создания объекта `dict`; такая функция, создающая экземпляры класса, называется *конструктором*. Как вам, вероятно, известно, для пользовательских классов конструктором является определяемая вами функция `__init__`. Так как конструктор используется для создания экземпляров, процесс создания экземпляра называется конструированием.

ПРИМЕЧАНИЕ Создание экземпляров более подробно рассматривается в главе 8.

5.1.1. Итерируемые объекты и итераторы

Использование итерируемых объектов не является обособленной темой, она тесно связана с важным понятием итераторов. Итератор — специальный тип данных, при помощи которого можно получить каждый из входящих в него элементов, для чего используется процесс *перебора*, или *итерации*. Ключевая связь между итерируемыми объектами и итераторами заключается в том, что все итерируемые объекты преобразуются в итераторы, прежде чем с ними можно будет выполнять операции перебора.

Во внутренней реализации всю работу выполняют две функции: `iter` и `next`. На рис. 5.1 показано, как организована совместная работа итерируемых объектов и итераторов при переборе, состоящая из трех фаз:

1. Создание итератора на базе итерируемого объекта при помощи функции `iter`. Итераторы спроектированы для выполнения перебора элементов итерируемого объекта.
2. Генерирование элементов с использованием `next`. При вызове `next` для итератора вы получаете следующий элемент, если он доступен.

3. Остановка перебора исключением `StopIteration`. Если доступных элементов не осталось, вызов `next` приводит к исключению `StopIteration`.

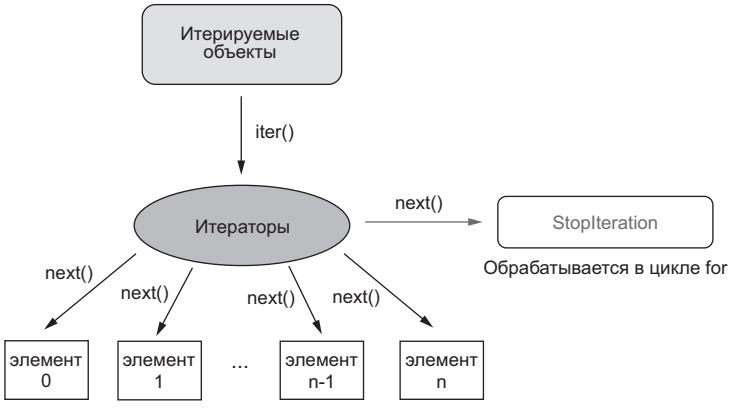


Рис. 5.1. Схема перебора с использованием итераторов. Итераторы создаются функцией iter на базе итерируемых объектов. Итераторы используют функцию next для получения следующего элемента, если он доступен. Когда итератор переберет все свои элементы, выдается исключение StopIteration

Чтобы продемонстрировать процесс перебора, рассмотрим распространенный итерируемый объект — объект `list`, на основе которого мы создаем итератор вызовом `iter`:

```
tasks = ["task0", "task1", "task2"]

tasks_iterator = iter(tasks)

tasks_iterator
# Вывод: <list_iterator object at 0x000001F232ACEE50>
```

На вашем компьютере адрес памяти будет другим

Мы начнем с объекта `list` с именем `tasks` и создадим итератор `list_iterator` вызовом функции `iter`. Можно использовать функцию `next` для получения элементов итератора один за другим:

```
next(tasks_iterator)
# Вывод: 'task0'

next(tasks_iterator)
# Вывод: 'task1'

next(tasks_iterator)
# Вывод: 'task2'

next(tasks_iterator)
# ERROR: StopIteration
```

Как видите, при каждом вызове `next` для итератора мы получаем следующий элемент, пока не будут исчерпаны все элементы и не произойдет исключение `StopIteration`.

Это описание вызовов `iter` и `next` дает чисто механическое представление о процессе перебора. В коде нам почти не приходится создавать итераторы самостоятельно — Python незаметно выполняет всю черную работу за нас. Возьмем цикл `for` — самую распространенную форму использования итерируемых объектов и итераторов, как в следующем примере:

```
for task in tasks:
    print(task)

# Выводимые строки:
task0
task1
task2
```

Мы используем список `tasks` прямо в цикле `for`, не беспокоясь о создании итератора, так как он будет автоматически создан Python. Важно также, что вместо выдачи исключения `StopIteration` при исчерпании итератора списка происходит безопасный выход из цикла `for`, так как исключение обрабатывается само.

5.1.2. Проверка итерируемости

Чтобы эффективнее использовать итерируемые объекты в коде, важно знать, какие типы данных являются итерируемыми (помимо уже рассмотренных `str`, `list`, `tuple`, `dict` и `set`). В этом разделе вы узнаете, как определить, является ли конкретный объект итерируемым.

С практической точки зрения любой тип данных, который может использоваться в цикле `for`, является итерируемым. Как формально проверить объект на итерируемость? Из предыдущего раздела можно сделать вывод, что если объект может быть преобразован в итератор при помощи функции `iter`, он является итерируемым. Следующий фрагмент показывает, как объекты (объект `int` и объект `list`) по-разному ведут себя в отношении итерируемости:

```
iter(5)
# ERROR: TypeError: 'int' object is not iterable

iter([1, 2, 3])
# Вывод: <list_iterator object at 0x000001F232A44700>
```

> **ОСНОВНЫЕ ПОНЯТИЯ** Термином «итерируемость» обозначается характеристика объекта, который может быть преобразован в итератор для перебора.

В дополнение к проверке итерируемости объекта следует знать о том, какие популярные типы данных являются итерируемыми помимо `str`, `list`, `tuple`, `dict` и `set`. Используя `iter` для определения итерируемости, можно прийти

к решению, приведенному в следующем листинге. В главе 12 конструкция `try...except...` рассматривается более подробно.

Листинг 5.1. Проверка того, является ли объект итерируемым

```
def is_iterable(obj):
    try:
        _ = iter(obj)
    except TypeError:
        print(type(obj), "is not an iterable")
    else:
        print(type(obj), "is an iterable")

is_iterable(5)
# Вывод: <class 'int'> is not an iterable

is_iterable([1, 2, 3])
# Вывод: <class 'list'> is an iterable
```

Символ подчеркивания показывает, что возвращаемый результат не будет использоваться

Секция else выполняется при отсутствии исключения TypeError

В листинге 5.1 для проверки итерируемости объекта мы пытаемся вызвать функцию `iter` непосредственно для объекта. Если вызов завершается успехом, то объект является итерируемым, в случае неудачи — не является. Используя функцию `is_iterable`, можно выполнить проверку для серии встроенных объектов, чтобы определить, какие типы данных являются итерируемыми. В табл. 5.1 перечислены распространенные встроенные итерируемые объекты.

Таблица 5.1. Часто используемые встроенные итерируемые объекты с примерами

Тип данных	Пример кода	Тип итератора
`str`	`"Hello"`	`str_iterator`
`list`	`[1, 2, 3]`	`list_iterator`
`tuple`	`(1, 2, 3)`	`tuple_iterator`
`dict`	`{"one": 1, "two": 2}`	`dict_keyiterator`[a]
`set`	`{1, 2, 3}`	`set_iterator`
`range`	`range(3)`	`range_iterator`
`map`	`map(int, ["1", "2"])`	`map`
`zip`	`zip([1, 2], [2, 3])`	`zip`
`filter`	`filter(bool, [1, None])`	`filter`
`enumerate`	`enumerate([1, 2, 3])`	`enumerator`
`reversed`	`reversed("Hello")`	`reversed`

[a] При переборе `dict` по умолчанию перебираются ключи. Следующие две операции эквивалентны: `for key in dict` и `for key in dict.keys()`. Также можно перебирать значения и элементы объекта `dict`. За дополнительной информацией обращайтесь к разделу 5.3.7.

В табл. 5.1 встречаются отдельные типы данных, которые ранее еще не рассматривались, такие как `map` и `zip`. В разделе 5.1.3 рассматриваются некоторые из этих итерируемых типов.

5.1.3. Использование итерируемых объектов для создания встроенных контейнеров данных

В главе 2 рассматривались типы данных коллекций, также называемые контейнерами данных: списки, множества, кортежи, словари и т. д. В простых ситуациях можно использовать соответствующие литеральные формы для создания данных, если они содержат небольшое количество элементов.

В листинге 5.2 мы создаем несколько контейнеров данных без использования их конструкторов. Вместо этого указываются данные с их специфическими синтаксическими требованиями: квадратные скобки для объектов `list`, фигурные скобки для объектов `set`. Такой подход называется *созданием экземпляров на базе литералов*.

Листинг 5.2. Использование литералов для создания экземпляров

```
list_obj = [1, 2, 3]

tuple_obj = (404, "Connection Error")

dict_obj = {"one": 1, "two": 2}

set_obj = {1, 2, 3}
```

Однако если вам нужно создать данные для контейнера с множеством элементов, литералы оказываются не такими удобными. Следует заметить, что каждый из этих типов данных коллекций имеет свои конструкторы, которые используют соответствующие имена классов, и они могут получать итерируемые объекты для создания новых объектов коллекций. В листинге 5.3 показано, как это делается.

Листинг 5.3. Использование итерируемых объектов для создания экземпляров

```
integers_list = list(range(10))  <-- Вызывает конструктор list
assert integers_list == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

integers_tuple = tuple(integers_list)  <-- Вызывает конструктор tuple
assert integers_tuple == (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

dict_items = [("zero", 0), ("one", 1), ("two", 2)]
integers_dict = dict(dict_items)  <-- Вызывает конструктор dict
assert integers_dict == {'zero': 0, 'one': 1, 'two': 2}

even_numbers = (-2, 4, 0, 2, 4, 2)
unique_evens = set(even_numbers)  <-- Вызывает конструктор set
assert unique_evens == {0, 2, 4, -2}
```

Как показано в листинге 5.3, конструкторы `list`, `tuple`, `dict` и `set` могут создавать объекты из соответствующего итерируемого объекта. Создание контейнеров на базе итерируемых объектов часто встречается в реальных проектах, имеющих дело со многими разновидностями итерируемых объектов, данные которых связаны между собой. Такая возможность часто используется для создания новых данных из существующих итерируемых объектов.

> **ВОПРОС** *Строки* являются итерируемыми объектами, состоящими из символов. Допустим, имеется следующий объект `str`: `letters = "ABCDE"`. Как лучше всего создать из `letters` список символов `["A", "B", "C", "D", "E"]`?

Предположим, в проекте используется объект `list` со строками, при этом каждая строка представляет число с плавающей точкой: `numbers_str = ["1.23", "4.56", "7.89"]`. Для проведения вычислений строки преобразуются в числа с плавающей точкой. Преобразование можно выполнить с использованием контейнера `map`, который применяет функцию к каждому элементу итерируемого объекта и создает итератор для `map`:

```
numbers_str = ["1.23", "4.56", "7.89"]

numbers_float = list(map(float, numbers_str))

assert numbers_float == [1.23, 4.56, 7.89]
```

В приведенном примере кода функция `map` применяет встроенную функцию `float` (точнее, это конструктор `float`) к каждой строке, а конструктор `list` получает созданный итератор `map` для создания объекта `list`, содержащего числа с плавающей точкой.

> **ЗАБЕГАЯ ВПЕРЕД** Функция `map` является функцией высшего порядка, которая получает функцию в аргументе. За дополнительной информацией обращайтесь к разделу 7.2.

Среди других контейнеров данных конструктор типа `dict` занимает особое место, так как он требует, чтобы каждый элемент итерируемого объекта состоял из двух парных компонентов: ключа и значения. Кроме списков кортежей, каждый из которых состоит из двух элементов, объекты `dict` обычно создаются на базе существующих итерируемых объектов при помощи функции `zip`, производящей слияние двух итерируемых объектов. Перед вами тот же сценарий, который уже упоминался выше: создание объекта `dict` из двух объектов `list`. Решение выглядит так:

```
zipped_tasks = dict(zip(id_numbers, titles))

assert zipped_tasks == {101: "Laundry", 102: "Homework", 103: "Soccer"}
```

Функция `zip` соединяет `id_numbers` и `titles` в одном ряду и образует итератор `zip`, который генерирует элементы, включающие один элемент из каждого итерируемого объекта. На рис. 5.2 показано, как работает функция `zip`.

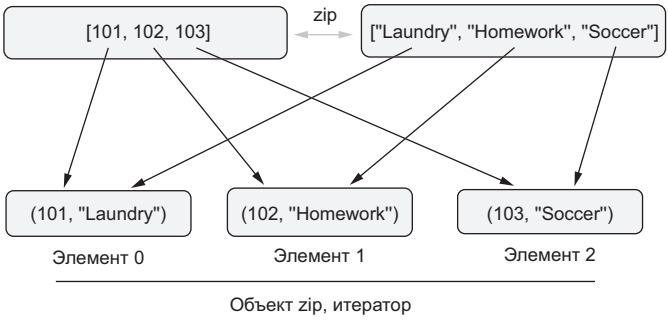


Рис. 5.2. Функция zip создает итератор из нескольких итерируемых объектов. В данном примере используются два итерируемых объекта. Функция zip соединяет объекты, находящиеся в соответствующих позициях двух итерируемых объектов. Обратите внимание: порядок итерируемых объектов, используемых функцией zip, важен, потому что в создаваемых кортежах элементы следуют в порядке, совпадающем с порядком итерируемых объектов

На рис. 5.2 два итерируемых объекта используются для создания итератора — объекта `zip`, который генерирует кортежи, состоящие из двух элементов. Двухэлементные кортежи — именно то, что нужно конструктору `dict`; первый элемент становится ключом, а второй — соответствующим ему значением. В реальных проектах функция `zip` часто применяется для создания объектов `dict`.

5.1.4. Обсуждение

Помимо стандартной библиотеки Python, итерируемые объекты широко используются в сторонних библиотеках. Например, `ndarray` в NumPy и `Series` в библиотеке pandas могут получать итерируемый объект для создания экземпляра. Если ваша работа связана с data science, эта возможность упростит преобразование данных между разными типами итерируемых объектов.

Такие разные zip

Функция `zip` соединяет два и более итерируемых объекта, каждый из которых вносит один компонент в элементы итератора `zip`. В большинстве случаев функция `zip` используется с двумя итерируемыми объектами.

Вероятно, термин «zip» также встречался вам в контексте сжатия файлов. В Python функциональность сжатия и распаковки файлов предоставляется модулем `zipfile`.

5.1.5. Задача

Ава, перспективный финансовый аналитик, изучает Python. Ей очень нравится функция `zip`, соединяющая несколько итераторов, и ей хотелось бы узнать, как `zip` работает с несколькими итерируемыми объектами. Помогите ей написать код для тестирования `zip` с тремя итерируемыми объектами.

Количество элементов в разных итерируемых объектах тоже различается. Определите, что происходит при попытке соединения итерируемых объектов с разным количеством элементов функцией `zip`.

> **ПОДСКАЗКА** Два итерируемых объекта образуют двухэлементные кортежи после соединения. Если один итерируемый объект короче другого, то более короткий итерируемый объект не сможет предоставить компонент, когда его элементы будут полностью израсходованы.

5.2. ЧТО ТАКОЕ ВКЛЮЧЕНИЯ СПИСКОВ, СЛОВАРЕЙ И МНОЖЕСТВ

Если спросить программиста Python среднего уровня, какое языковое средство можно считать наиболее питоническим, скорее всего, он ответит, что это *списковые включения* (list comprehension) — компактные средства создания объектов `list`. Следующий пример показывает, как выглядит списковое включение:

```
numbers = [1, 2, 3, 4]
squares = [x * x for x in numbers]
assert squares == [1, 4, 9, 16]
```

Как видите, списковые включения не похожи на литералы, так как их элементы не перечисляются напрямую, но они также не похожи на конструирование, потому что они не вызывают `list`. Списковые включения — питоническая конструкция, которой вы будете часто пользоваться. Определение «питонический» подразумевает компактность и удобочитаемость. Кроме списковых включений, также доступны включения словарей и множеств для создания объектов `dict` и `set` соответственно. В следующем разделе рассматриваются средства включения, а также некоторые потенциальные проблемы, которых вам следует избегать.

5.2.1. Создание списков из итерируемых объектов с использованием списковых включений

Мы используем разные виды итерируемых объектов для хранения разнообразных данных. Часто в проектах данные требуется преобразовать в объект `list`. В этом разделе вы научитесь преобразовывать итерируемые объекты в объекты `list` с использованием списковых включений. Допустим, в нашем таск-менеджере имеется объект `list` с экземплярами класса `Task`, как показано в листинге 5.4.

Листинг 5.4. Создание списка экземпляров пользовательского класса

```
from collections import namedtuple

Task = namedtuple("Task", "title, description, urgency")

tasks = [
    Task("Homework", "Physics and math", 5),
    Task("Laundry", "Wash clothes", 3),
    Task("Museum", "Egypt exhibit", 4)
]
```

Пользовательский класс использует именованные кортежи

Списковое включение или map

Мы используем списковое включение для создания объекта списка из существующего итерируемого объекта, а именно: тот же объект `list` можно создать конструктором `list` в сочетании с функцией `map`. Например, для построения списка названий в нашем примере можно воспользоваться следующим альтернативным решением:

```
def get_title(task):
    return task.title

titles = list(map(get_titles, tasks))
```

Как показано в разделе 7.1, также можно применить лямбда-выражение, чтобы избавиться от необходимости создавать функцию `get_title`: `titles = list(map(lambda x: x.title, tasks))`.

Как видите, решение с использованием `list` и `map` для создания объекта `list` получается более пространным, чем списковое включение; как следствие, оно в общем случае хуже читается. Я рекомендую использовать списковое включение вместо решения с `map`. Тем не менее некоторые разработчики предпочитают решение с `map`, потому что оно относится к стилю, называемому функциональным программированием. Этот стиль ориентируется на написание и использование функций, вместо того чтобы ориентироваться на объекты, как в языках ООП.

НАПОМИНАНИЕ Именованный кортеж — облегченная модель данных, используемая для хранения данных и поддержки точечной записи. За дополнительной информацией обращайтесь к разделу 3.3.

В нашем приложении понадобится объект `list` для хранения названий всех задач. Неопытный программист, ничего не знающий о списковых включениях, может предложить следующее решение:

```
task_titles = []
for task in tasks:
    task_titles.append(task.title)

assert task_titles == ['Homework', 'Laundry', 'Museum']
```

Программа перебирает элементы `tasks` в цикле `for` и получает их атрибуты `title`, которые присоединяются к списку `task_titles`. Такое решение работает, но в Python оно не слишком эффективно. Лучший подход основан на использовании спискового включения: `[выражение for элемент in итерируемый_объект]`, в котором `выражение` — конкретная операция, использующая каждый элемент итерируемого объекта. Выражения вычисляются и становятся элементами созданного списка. Следующий фрагмент кода демонстрирует использование спискового включения для извлечения названий задач:

```
titles = [task.title for task in tasks]
assert titles == ['Homework', 'Laundry', 'Museum']
```

Как показывает этот пример, при использовании спискового включения мы создаем объект `list` с нужными данными. Пример подчеркивает самое важное преимущество использования списковых включений: компактность. Использовать цикл `for` не обязательно, а вся операция занимает всего одну строку кода. И хотя этот прием может озадачить некоторых новичков, когда у вас появится больше опыта работы на Python, вы увидите, что списковые включения не только компактны, но и хорошо читаются.

5.2.2. Создание словарей из итерируемых объектов с использованием словарных включений

Словарь `dict` — еще один ключевой тип данных контейнера в Python. Как и в случае с объектами `list`, объекты `dict` тоже можно создавать с использованием включений: эта конструкция называется словарным включением. В этом разделе я кратко опишу словарные включения, так как их синтаксис лишь незначительно отличается от синтаксиса списковых включений. Принцип остается неизменным: словарные включения предоставляют компактный механизм создания объекта `dict` из существующего итерируемого объекта.

Так как словари состоят из пар «ключ — значение», словарное включение состоит из двух выражений, разделенных двоеточием: `{выражение_ключ: выражение_значение for элемент in итерируемый_объект}`, где `выражение_ключ` при вычислении дает ключ, а `выражение_значение` при вычислении дает соответствующее значение. Другое отличие в синтаксисе — использование фигурных скобок в словарных включениях в отличие от квадратных скобок в списковых включениях.

Возьмем за отправную точку тот же объект `list` с именем `tasks` и допустим, что нашему приложению нужен объект `dict`, у которого названия являются ключами, а описания — значениями. Следующий фрагмент показывает, как решить эту задачу с циклом `for` и словарным включением; вы можете непосредственно сравнить эти два решения по удобочитаемости:

```
title_dict0 = {}
for task in tasks:
    title_dict0[task.title] = task.description
```

```
title_dict1 = {task.title: task.description for task in tasks}

assert title_dict0 == title_dict1
```

По сравнению с непитоническим подходом с `for` словарные включения намного более компактны. Для опытных пользователей Python они также лучше читаются, так как вы можете сразу сказать, что названия становятся ключами, а описания — значениями. Такая ясность становится еще одним преимуществом включения как компактного метода создания контейнеров данных в Python.

5.2.3. Создание множеств из итерируемых объектов с использованием включений множеств

В разделе 3.5 вы узнали, что объекты `set` становятся идеальной моделью данных в тех случаях, когда нас интересует проверка принадлежности. Поэтому часто требуется создавать объекты `set`, полученные в результате преобразования других итерируемых объектов. Такое преобразование можно реализовать при помощи включения множества `{выражение for элемент in итерируемый_объект}`, в котором `выражение` при вычислении дает элементы множества. В этом разделе вы узнаете о включениях множеств (set comprehension).

> **НАПОМИНАНИЕ** Так как лежащая в основе реализация использует хеш-таблицы, поиск элемента в объекте множества выполняется за постоянное время; эта особенность называется *временно́й сложностью O(1)*.

Включения множеств используют фигурные скобки вместо квадратных. Рассматривая все три способа включения, можно заметить, что в них используются те же знаки, что и в соответствующих литеральных формах: `[]` для `list`, `{:}` для `dict` и `{}` для `set`. Чтобы не запутаться в том, какие скобки используются в том или ином случае, вспомните их литеральные формы.

Следующий фрагмент кода демонстрирует компактность включения множества для создания объекта `set` из итерируемого объекта в сравнении с циклом `for`. Мы используем `task.title` для получения названия каждой задачи, которое направляется в создаваемый объект `set`:

```
title_set0 = set()
for task in tasks:
    title_set0.add(task.title)
```

Для создания пустого множества необходимо использовать конструктор set, так как не существует литеральной формы для пустого множества

```
title_set1 = {task.title for task in tasks}

assert title_set0 == title_set1 == {'Homework', 'Laundry', 'Museum'}
```

Стоит заметить, что, как и конструктор `set` (пример: `set([1, 1, 2, 2, 3, 3]) = {1, 2, 3}`), включение множества автоматически удаляет дубликаты, потому

что в объектах множеств могут храниться только уникальные элементы (это объясняется используемой реализацией с хешированием). Иначе говоря, объекты с одинаковыми значениями (а следовательно, с одинаковыми хеш-кодами — вспомните концепцию согласованности хеш-функций) могут содержаться в объекте `set` только в одном экземпляре, как показано в следующем примере:

```
numbers = [-3, -2, -1, 0, 1, 2, 3]

squares = {x*x for x in numbers}

assert squares == {0, 9, 4, 1}  ◄——  Элементы в объекте set не упорядочены
```

5.2.4. Применение фильтрующего условия

При переборе итерируемого объекта иногда требуется проверить, удовлетворяет ли элемент некоторому критерию, прежде чем выполнять с ним операции. В этом разделе вы узнаете, как применять условие фильтрации во включениях.

Допустим, что для списка `tasks` требуется сгенерировать список, содержащий только названия задач со степенью срочности более 3. В таком случае итерируемый объект можно отфильтровать инструкцией `if`. Неопытный программист, не знающий о списковых включениях, может воспользоваться обычным циклом `for` и предложить следующее решение:

```
filtered_titles0 = []
for task in tasks:
    if task.urgency > 3:
        filtered_titles0.append(task.title)

assert filtered_titles0 == ['Homework', 'Museum']
```

В цикле `for` при каждой итерации проверяется степень срочности задачи, и задача включается в список только в том случае, если она проходит проверку. Однако в питоническом решении инструкция `if` интегрируется прямо в списковое включение: `[выражение for элемент in итерируемый_объект if условие]`. А конкретно после итерируемого объекта присоединяется инструкция `if` для фильтрации нужных элементов:

```
filtered_titles1 = [task.title for task in tasks if task.urgency > 3]

assert filtered_titles0 == filtered_titles1
```

И хотя соответствующие примеры здесь не приводятся, инструкция `if` также может использоваться в словарных включениях и включениях множеств для исключения ненужных элементов при создании объектов `dict` и `set`. Если вас интересует эта тема, опробуйте такую возможность.

5.2.5. Встроенные циклы for

При работе с вложенными данными иногда требуется получить все элементы всех уровней вложенных структур. Например, объект `list` с именем `tasks` представляет уровень данных, а каждый элемент определяет дополнительный уровень данных, так как каждая задача хранит собственные данные. В этом разделе вы узнаете, как использовать встроенные циклы `for` для получения внутренних элементов вложенных данных.

Для контраста начнем с непитонического подхода. Вероятно, вы знаете, что при использовании циклов `for` для перебора можно встроить цикл `for` в другой цикл `for`:

```
flattened_items0 = []
for task in tasks:
    for item in task:
        flattened_items0.append(item)

assert flattened_items0 == ['Homework', 'Physics and math', 5,
➥ 'Laundry', 'Wash clothes', 3, 'Museum', 'Egypt exhibit', 4]
```

Эта операция со встроенными циклами `for` допустима, потому что `tasks` является списком экземпляров `Task`, а каждый экземпляр `Task` представляет именованный кортеж — другую разновидность итерируемых объектов. Та же операция поддерживается списковыми включениями. А это означает, что списковое включение может содержать встроенные циклы `for`. Посмотрите, как это работает:

```
flattened_items1 = [item for task in tasks for item in task]

assert flattened_items0 == flattened_items1
```

В этом коде первый цикл `for` извлекает каждую задачу из списка `tasks`, а второй цикл `for` извлекает каждый элемент объекта `task`. Этот синтаксис может смутить некоторых начинающих из-за двух циклов `for`. Я рекомендую читать код так, как если бы вы имели дело с обычными встроенными циклами `for`. Первый цикл `for` — внешний, а второй — внутренний: `[выражение for итерируемый_объект in итерируемые_объекты for элемент in итерируемый_объект]`.

Теоретически количество встроенных циклов `for` может быть сколь угодно большим. Однако с точки зрения удобочитаемости я бы не рекомендовал использовать включения более чем с двумя уровнями циклов `for`.

> **УДОБОЧИТАЕМОСТЬ** Не используйте больше двух уровней циклов `for`. Списковое включение с тремя и более уровнями циклов `for` слишком трудно прочитать.

5.2.6. Обсуждение

В разделе 5.2 рассказано об использовании включения списков, словарей и множеств как компактного механизма создания объектов `list`, `dict` и `set` соответственно. На рис. 5.3 приведена краткая сводка этих средств.

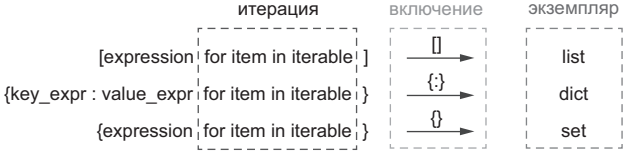


Рис. 5.3. Общие формы включений списков, словарей и множеств. Каждое включение перебирает итерируемый объект, использует специфический синтаксис для включения и создает соответствующие экземпляры

Вы должны хорошо представлять себе, в каких ситуациях следует использовать включения. Если вы начинаете с итерируемого объекта и хотите создать экземпляр класса `list`, `dict` или `set`, этот сценарий, пожалуй, будет наилучшим для использования включений. Знаете, почему я говорю «пожалуй»? Потому что есть несколько исключений.

Во-первых, если вам не нужно оперировать с элементами в итерируемом объекте, можно использовать их конструкторы напрямую. Вы начинаете с объекта `list`, `numbers = [1, 1, 2, 3]`, и, допустим, хотите создать объект `set`. Хотя использование включения множества не является ошибкой, не стоит применять его: `{x for x in numbers}`. Вместо этого лучше использовать конструктор множества, так как он получает итерируемый объект напрямую и создает объект `set`: `set(numbers)`.

Во-вторых, когда включение требует сложных выражений или глубоко вложенных циклов `for`, лучше использовать традиционное решение с циклом `for`. Допустим, имеется следующее включение:

```
styles = ['long-sleeve', 'v-neck']
colors = ['white', 'black']
sizes = ['L', 'S']

options = [' '.join([style, color, size]) for style in styles
➥ for color in colors for size in sizes]
```

Нельзя сказать, что этот код плохо читается, но вы должны постараться сделать так, чтобы ваш код был еще более удобочитаемым для большинства пользователей. Альтернатива выглядит так:

```
options = []
for style in styles:
    for color in colors:
        for size in sizes:
            option = ' '.join([style, color, size])
            options.append(option)
```

По сравнению с предыдущим это решение содержит чуть больше строк кода, но оно четко отражает разные уровни циклов `for`, благодаря чему проще читается и оказывается более понятным.

5.2.7. Задача

Лукас изучает Python для своей дипломной работы в области физики. Он понимает, что включения списков, словарей и множеств используют квадратные и фигурные скобки. Его интересует, что может делать конструкция (`выражение for элемент in итерируемый_объект`). Так как в выражении используются круглые скобки, применяемые для создания кортежей, не будет ли эта конструкция определять включение кортежа? Попробуйте выполнить эту конструкцию и расскажите Лукасу, что из этого получится.

> **ПОДСКАЗКА** Если бы включения кортежа работали, то я бы их уже описал. Для проверки природы объекта можно воспользоваться функцией `type`. Создаваемый объект рассматривается в разделе 7.4.

5.3. КАК УЛУЧШИТЬ ЦИКЛЫ FOR С ИСПОЛЬЗОВАНИЕМ ВСТРОЕННЫХ ФУНКЦИЙ

В своих проектах мы обычно рассчитываем на представление данных в упорядоченном виде. Допустим, на интернет-форуме названия сообщений должны выводиться слева, а авторы — справа. А чтобы вывести чек в удобном виде, следует перечислить позиции одну за другой с указанием соответствующей цены. Можно сказать, что в каждом проекте используется структурированная информация, а всеобщая потребность в хранении этой информации оправдывает реализацию в Python различных видов итерируемых объектов с разными характеристиками.

Для структурированной информации: сообщений, упорядоченных элементов или любых данных, актуальных для ваших проектов, — в большинстве случаев используются однородные данные, а применяется обычно одна и та же операция. Если одна и та же операция применяется к итерируемому объекту, лучше использовать цикл `for` в следующей форме (с которой вы должны быть знакомы):

```
for item in iterable:
    # Та же операция
```

Эта базовая форма станет хорошей отправной точкой на пути к решению задач, связанных с перебором. Однако Python предоставляет и другие средства, которые улучшают работу циклов. В этом разделе рассматриваются питонические реализации для типичных практических ситуаций. Сначала я привожу непитоническое решение, а затем мы исследуем другое, более эффективное. Напоследок я кратко рассмотрю разные функции и практические приемы.

5.3.1. Перечисление элементов и enumerate

Многие итерируемые объекты, такие как списки и кортежи, представляют собой последовательности. Каждый элемент характеризуется соответствующим индексом — его позицией в последовательности. Часто возникает необходимость в использовании информации о позиции элемента вместе с данными элемента. В этом разделе рассматривается решение такой задачи, оно называется *перечислением* (enumeration).

Допустим, в таск-менеджере хранится список экземпляров класса `Task`. Для простоты класс `Task` реализуется через именованные кортежи, как показано в листинге 5.5.

Листинг 5.5. Создание списка экземпляров пользовательского класса

```
from collections import namedtuple

Task = namedtuple("Task", "title description urgency")
tasks = [
    Task("Homework", "Physics and math", 5),
    Task("Laundry", "Wash clothes", 3),
    Task("Museum", "Egypt exhibit", 4)
]
```

Требуется вывести эти задачи в виде нумерованного списка:

```
Task 1: task1_title task1_description task1_urgency
Task 2: task2_title task2_description task2_urgency
Task 3: task3_title task3_description task3_urgency
```

В этом выводе не хватает только индекса каждой задачи в `tasks`. Таким образом, можно прийти к следующему решению:

```
for task_i in range(len(tasks)):
    task = tasks[task_i]
    task_counter = task_i + 1
    print(f"Task {task_counter}: {task.title:<10}
    ➥ {task.description:<18} {task.urgency}")

# Выводимые строки:
Task 1: Homework   Physics and math   5
Task 2: Laundry    Wash clothes       3
Task 3: Museum     Egypt exhibit      4
```

УДОБОЧИТАЕМОСТЬ В f-строках (раздел 2.1.4) для форматирования интерполируемых строк используются спецификаторы формата, среди них спецификаторы выравнивания текста, приведенные в этом примере. Структурное выравнивание упрощает чтение строкового вывода.

Решение создает объект `range` с длиной `tasks`. Заметим, что если конструктору `range` передается только один аргумент (`len(tasks)`, то есть `3`), он интерпретируется как параметр `stop`; таким образом, объект `range` состоит из индексов `0`, `1` и `2`.

И хотя такое решение работает в описанной ситуации, существует другое, более питоническое решение. В нем используется функция `enumerate`, которая получает все элементы и генерирует счетчик для каждого элемента:

```
for task_i, task in enumerate(tasks, start=1):
    print(f"Task {task_i}: {task.title:<10}
    ➥ {task.description:<18} {task.urgency}")
```

Функция `enumerate` получает итерируемый объект и создает итератор типа `enumerate` (табл. 5.1). Этот итератор генерирует объект `tuple (item_counter, item)`, содержащий счетчик (item_counter) и элемент (item) из исходного итерируемого объекта. Также заметим, что функция `enumerate` получает необязательный аргумент `start`, который позволяет указать номер первого элемента. В нашем случае отсчет должен начинаться с 1, поэтому в вызов `enumerate` включается аргумент `start=1`.

НАПОМИНАНИЕ В этом решении используется распаковка кортежей (раздел 4.4). Каждый элемент итератора `enumerate` представляет собой объект `tuple`. Однозначная распаковка создает две переменные, `task_i` и `task`, для одновременного обращения к счетчику и элементу.

5.3.2. Обратная перестановка элементов функцией reversed

В этом разделе мы начинаем с того же итерируемого объекта: списка `tasks` из раздела 5.3.1. Однако на этот раз требуется вывести задачи в обратном порядке, сохранив исходные данные для других целей. Когда вы сталкиваетесь с такой задачей, первым побуждением может быть получение элементов в обратном порядке, от последнего к первому. Это может привести к следующему решению:

```
for task_i in range(len(tasks)):
    task = tasks[-(task_i + 1)]
    print(f"Task: {task}")

# Выводимые строки:
Task: Task(title='Museum', description='Egypt exhibit', urgency=4)
Task: Task(title='Laundry', description='Wash clothes', urgency=3)
Task: Task(title='Homework', description='Physics and math', urgency=5)
```

Такое решение создает объект `range` с длиной, соответствующей длине `tasks`. Обратите внимание на применение отрицательного индексирования (раздел 4.2) для получения элементов в порядке, обратном порядку исходного списка. Так как отрицательное индексирование начинается с `-1` для последнего элемента, необходимо увеличить `task_i` на 1 перед сменой знака индекса. Как видите, понять, как создать нужные отрицательные индексы из положительных индексов, не так просто. В этом конкретном сценарии в питоническом решении используется функция `reversed`:

```
for task in reversed(tasks):
    print(f"Task: {task}")
```

Функция `reversed` получает объект данных последовательности и возвращает объект с элементами, следующими в обратном порядке. Следует заметить, что инвертированный объект представляет собой итератор, который генерирует элементы в обратном порядке относительно исходного объекта `list`. По сравнению с непитоническим решением в этом случае не приходится иметь дело с какими-либо индексами. Вместо этого `task` используется для прямых обращений к элементам, генерируемым итератором `reversed`. Такой прямой доступ без преобразований индексов нагляден и хорошо читается.

5.3.3. Соединение итерируемых объектов с использованием zip

Если есть несколько итерируемых объектов для хранения разных видов информации об одном объекте, нужно иметь возможность выполнять операции, которые требуют информации из всех итерируемых объектов. В таком случае эти итерируемые объекты приходится каким-то образом соединять. В этом разделе рассматривается соединение итерируемых объектов функцией `zip`.

Описание этой операции может запутать читателя. Я поясню ее конкретным примером. Кроме объекта `list` с именем `tasks`, наше приложение содержит два объекта `list` — `dates` с датами выполнения задач и `locations` с местами, в которых должны выполняться задачи:

```
dates = ["May 5, 2022", "May 9, 2022", "May 11, 2022"]

locations = ["School", "Home", "Downtown"]
```

Требуется вывести для пользователей следующую информацию: название каждой задачи, дату и место выполнения (location). Когда вы сталкиваетесь с подобными задачами, приходит на ум, что эти итерируемые объекты содержат разные аспекты одних и тех же элементов. Можно заметить, что у этих итерируемых объектов есть одна особенность: информация с заданным индексом относится к одной и той же задаче. Таким образом, можно прийти к следующему решению:

```
for task_i in range(len(tasks)):
    task = tasks[task_i]
    date = dates[task_i]
    location = locations[task_i]
    print(f"{task.title}: by {date} at {location}")

# Выводимые строки:
Homework: by May 5, 2022 at School
Laundry: by May 9, 2022 at Home
Museum: by May 11, 2022 at Downtown
```

Так как мы знаем, что индексы представляют собой согласованные элементы, которые позволяют ссылаться на одни и те же задачи по всем итерируемым объектам, мы создаем объект `range` для получения индексов. Но в разделе 5.1.3 была описана функция `zip`, которая может использоваться для соединения двух итерируемых объектов при создании объекта `dict`. Как там упоминалось, объект `zip` представляет собой итератор, который генерирует объекты `tuple`, полученные в результате слияния совмещаемых итерируемых объектов. Решение, в котором используется функция `zip`, выглядит так:

```
for task, date, location in zip(tasks, dates, locations):
    print(f"{task.title}: by {date} at {location}")
```

Функция `zip` получает несколько итерируемых объектов (в данном случае три) и совмещает их параллельно. Объект `zip`, создаваемый при этом вызове функции, является итератором, и он генерирует объект `tuple` из трех элементов, вносимых отдельными итерируемыми объектами. Как и в случае с объектом `enumerate`, однозначная распаковка кортежа используется для одновременного создания переменных `task`, `date` и `location`, что значительно улучшает компактность и удобочитаемость вашего кода (вы скоро привыкнете к этой конструкции, и она станет вполне понятной).

Также может оказаться, что итерируемые объекты содержат разное количество элементов. По умолчанию функция `zip` прекращает соединение после исчерпания итерируемого объекта с наименьшим количеством элементов. Но если вы хотите, чтобы соединение продолжалось по длине итерируемого объекта с наибольшим количеством элементов, используйте функцию `zip_longest` из модуля `itertools`.

Соединение итерируемых объектов с разным количеством элементов

Для демонстрации работы `zip` я использовал итерируемые объекты одинаковой длины. Но что произойдет при соединении итерируемых объектов с разным количеством элементов?

По умолчанию функция `zip` прекращает соединение после исчерпания итерируемого объекта с наименьшим количеством элементов. Например, при вы-

полнении вызова `zip(range(3), range(4))` вы получите только три объекта `tuple`. Иногда бывает нужно проследить за тем, чтобы итерируемые объекты содержали одинаковое количество элементов. Для обеспечения согласованности длин в Python 3.10 появился необязательный параметр `strict`; если он равен `True`, то количество элементов должно быть одинаковым. Заметим, что по умолчанию параметр `strict` равен `False`, так что приведенные примеры использования `zip` будут работать. Выпуск новой версии программного продукта, сохраняющей работоспособность созданного в старой версии кода, называется *обратной совместимостью*.

В некоторых случаях бывает нужно, чтобы соединение продолжалось до исчерпания итерируемого объекта с наибольшим количеством элементов. В таких случаях стоит воспользоваться функцией `zip_longest` из модуля `itertools` стандартной библиотеки Python. Ее использование продемонстрировано в следующем фрагменте. Как видите, при исчерпании более коротких итерируемых объектов Python использует значение `None` для соединения с оставшимися элементами более длинных итерируемых объектов:

```
>>> from itertools import zip_longest
>>> list(zip_longest(range(3), range(4), range(5)))
[(0, 0, 0), (1, 1, 1), (2, 2, 2), (None, 3, 3), (None, None, 4)]
```

5.3.4. Сцепление нескольких итерируемых объектов функцией chain

В функции `zip` итерируемые объекты располагаются рядом друг с другом (параллельно) перед соединением соответствующих элементов. Но возможна и другая ситуация: у вас есть несколько итерируемых объектов, которые необходимо соединить так, чтобы элементы можно было извлекать последовательно. Иначе говоря, итерируемые объекты должны использоваться последовательно, а не параллельно. В этом разделе будет рассмотрена операция, называемая *сцеплением* итерируемых объектов (chaining of the iterables). Допустим, в дополнение к списку `tasks` имеется другой объект `list`, в котором хранятся недавно завершенные задачи:

```
completed_tasks = [
    Task("Toaster", "Clean the toaster", 2),
    Task("Camera", "Export photos", 4),
    Task("Floor", "Mop the floor", 3)
]
```

Требуется вывести все названия завершенных и незавершенных задач. В такой ситуации можно создать объект `list`, который соединяет `tasks` и `completed_tasks`, в результате будет получено следующее решение:

```
all_tasks = tasks + completed_tasks
for task in all_tasks:
```

```
    print(task.title)

# Выводимые строки:
Homework
Laundry
Museum
Toaster
Camera
Floor
```

Такое решение работает, но оно требует создания промежуточного объекта `list`. И хотя при небольшом размере объекта `list` этот факт обычно остается незамеченным, при работе с несколькими большими объектами `list` затраты памяти могут создать проблемы. Более питоническое решение требует использования функции `chain`:

```
from itertools import chain

for task in chain(tasks, completed_tasks):
    print(task.title)
```

Функция `chain`, как и функция `zip_longest`, доступна в модуле `itertools`. `chain` получает несколько итерируемых объектов и создает итератор, объединяющий все элементы из этих итерируемых объектов. Таким образом, как `zip`, так и `chain` могут получать несколько итерируемых объектов и соединять их разными способами. Различия представлены на рис. 5.4.

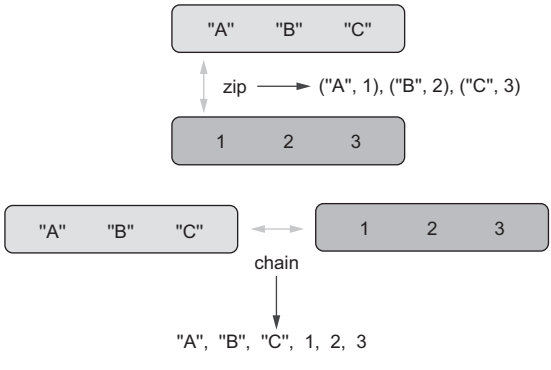


Рис. 5.4. Обе функции, zip и chain, получают несколько итерируемых объектов. Функция zip параллельно соединяет итерируемые объекты по каждому индексу, а функция chain соединяет итерируемые объекты последовательно. Итератор zip генерирует объекты tuple с элементами, которые берутся из каждого итерируемого объекта. Итератор chain генерирует элементы последовательно из всех итерируемых объектов. На диаграмме для примера используются два итерируемых объекта, но обе функции могут получать более двух итерируемых объектов

Иначе говоря, перебор по многим итерируемым объектам обрабатывается через итератор `chain`, который не добавляет лишних затрат памяти, обусловленных созданием промежуточного объекта `list` в непитоническом решении.

5.3.5. Фильтрация итерируемого объекта функцией filter

Итерируемый объект состоит из многих элементов. Однако в некоторых случаях требуется работать с подмножеством элементов, удовлетворяющих нашим потребностям. В этом разделе рассматривается возможность фильтрации итерируемых объектов функцией `filter`.

Допустим, требуется вывести информацию о задачах со степенью срочности больше 3. Как упоминалось в описании включений (раздел 5.2.4), условие фильтрации можно применить в цикле `for`:

```
for task in tasks:
    if task.urgency > 3:
        print(task)

# Выводимые строки:
Task(title='Homework', description='Physics and math', urgency=5)
Task(title='Museum', description='Egypt exhibit', urgency=4)
```

Надо сказать, что это абсолютно нормальное решение, и если вы нашли его самостоятельно, я могу вас только поздравить. Но чуть более эффективное решение, питоническое или нет (решайте сами, обратившись к разделу 5.3.6), использует функцию `filter`:

```
for task in filter(lambda x: x.urgency > 3, tasks):
    print(task)
```

Функция `filter` получает функцию, которая применяется к элементам итерируемого объекта. Каждый элемент вычисляется функцией: если он равен `True`, то элемент остается, а если `False` — исключается. В нашем примере используется лямбда-функция, а `x` обозначает элемент из итерируемого объекта. И хотя лямбда-функции уже встречались вам при обсуждении сортировки в разделе 3.2, они подробно рассматриваются в разделе 7.1. А пока считайте, что лямбда-функция — обычная функция, которая возвращает значение из выражения; в нашем случае это признак того, что степень срочности задачи превышает 3.

5.3.6. Обсуждение

При использовании `reversed` создается итератор, который содержит те же элементы, что и итерируемый объект, но в обратном порядке. Не путайте `reversed` с методом `reverse`, который переставляет объект `list` на месте. Изменение на месте подразумевает, что метод изменяет исходный объект `list` и возвращает

None. А значит, следующий код выполняться не будет! Аналогичное различие существует между sorted и sort. В первом случае создается отсортированный объект list, совместимый с циклом for. Во втором случае возвращается None, и этот вариант несовместим с циклом for:

```
tasks = ["task1", "task2", "task3"]
for task in tasks.reverse():
    pass
```

Начиная с Python 3.10, у zip появился необязательный параметр strict. Присваивая strict значения True, вы требуете, чтобы длины итерируемых объектов были одинаковыми; в противном случае zip останавливается при исчерпании самого короткого итерируемого объекта. Как будет показано в разделе 6.1, определение значения по умолчанию для параметра позволяет опустить соответствующий атрибут при вызове функции. Самое важное следствие заключается в том, что в старой кодовой базе любой вызов функции zip, например zip(list0, list1), будет работать даже в том случае, если вы обновите свою копию Python до версии 3.10. Функция будет интерпретироваться как zip(list0, list1, strict=False); при этом не требуется, чтобы итерируемые объекты содержали одинаковое количество элементов, как старая функция zip до выхода Python 3.10. Это решение обеспечивает обратную совместимость.

> **СОПРОВОЖДАЕМОСТЬ** При включении новых возможностей в существующую кодовую базу желательно сохранять обратную совместимость, чтобы вам не пришлось возвращаться и исправлять код, в котором использовалась старая функциональность.

За более мощными средствами перебора обращайтесь к модулю itertools, в котором предоставляется различная функциональность, связанная с итерациями, — не только zip_longest и chain. Например, range представляет собой итерируемый объект, который генерирует целые, но не дробные числа. При этом в itertools присутствует функция count, которая создает итератор для генерирования значений, разделенных интервалами постоянной величины, включая дробные.

Что касается функции filter, некоторые разработчики предпочитают filter как питоническую реализацию. Но на мой взгляд, функция filter не имеет сколько-нибудь заметных преимуществ перед инструкцией if. Мне инструкция if кажется более наглядной, так как критическая логическая операция (проверка условия) размещается в отдельной строке кода. Решайте сами, что использовать — filter или инструкцию if.

5.3.7. Задача

В разделе 3.4 вы научились пользоваться функциями keys(), values() и items() для обращения к ключам словаря, его значениям или парам «ключ — значение».

Все ли полученные результаты являются итерируемыми объектами? Если потребуется перебрать пары «ключ — значение», как это лучше сделать?

ПОДСКАЗКА Функция `items` возвращает пары «ключ — значение» в виде объектов `tuple`, и вы можете воспользоваться распаковкой кортежей для получения ключа и значения из каждого объекта `tuple`.

5.4. КАК ИСПОЛЬЗОВАТЬ НЕОБЯЗАТЕЛЬНЫЕ ИНСТРУКЦИИ В ЦИКЛАХ FOR И WHILE

Ранее мы говорили о том, как циклы `for` помогают выполнять повторяющиеся рабочие операции за счет перебора итерируемых объектов. Кроме циклов `for`, для выполнения повторяющихся действий мы часто используем другую важную управляющую конструкцию — циклы `while`. Если вы еще не знакомы с циклами `while`, взгляните на приведенные ниже примеры. Фактически вы указываете условие после ключевого слова `while`, а код вычисляет это условие в каждой итерации. Если результат равен `True`, то код в теле цикла выполняется; в приведенном примере он выполняется, когда значение `n` равно `1` или `2`. Если результат равен `False`, то программа выходит из цикла `while`; в примере `n` содержит `3` после завершения цикла `while`:

```
n = 1
while n < 3:
    print(f"n's value: {n}")
    n += 1
print(f"n's value after while loop: {n}")

# Выводимые строки:
n's value: 1
n's value: 2
n's value after while loop: 3
```

Эти управляющие конструкции выполняют код из тела цикла во время итерации. Но вам не всегда нужно завершать итерации для всех элементов. Допустим, у вас имеется список задач, которые должны быть завершены в течение недели; вы собираетесь в первую очередь заняться неотложными задачами и поэтому хотите найти первую задачу со степенью срочности 5.

Листинг 5.6. Поиск неотложной задачи с созданием списка

```
from collections import namedtuple

Task = namedtuple("Task", "title, description, urgency")

tasks = [
    Task("Toaster", "Clean the toaster", 2),
```

Задача Task создается с помощью именованных кортежей

```
    Task("Camera", "Export photos", 4),
    Task("Homework", "Physics and math", 5),
    Task("Floor", "Mop the floor", 3),
    Task("Internet", "Upgrade plan", 5),
    Task("Laundry", "Wash clothes", 3),
    Task("Museum", "Egypt exhibit", 4),
    Task("Utility", "Pay bills", 5)
]
```

Если вы попытаетесь решить эту задачу с использованием цикла `for`, можно прийти к следующему решению:

```
first_urgent_task0 = None
for counter, task in enumerate(tasks, 1):
    print(f"---checking task {counter}: {task.title}")
    if (task.urgency == 5) and (first_urgent_task0 is None):
        first_urgent_task0 = task

print(f"***first urgent task: {first_urgent_task0}")

# Выводимые строки:
---checking task 1: Toaster
---checking task 2: Camera
---checking task 3: Homework
---checking task 4: Floor
---checking task 5: Internet
---checking task 6: Laundry
---checking task 7: Museum
---checking task 8: Utility
***first urgent task: Task("Homework", "Physics and math", 5)
```

Значение присваивается, если задача является неотложной, а значение first_urgent_task0 не задано

НАПОМИНАНИЕ Функция `enumerate` создает счетчик для итерируемого объекта.

Как видите, цикл `for` перебирает весь объект `list`, прежде чем завершить необходимую работу. Просматривая список, можно заметить, что искомая задача находится в самом начале; будет крайне неэффективно ожидать завершения итерации, пока в цикле `for` не будет закончен перебор всего объекта `list` не будет перебран. Почему бы не выйти из цикла `for` после обнаружения искомой задачи? К счастью, поведение перебора по умолчанию можно изменить при помощи двух необязательных инструкций: `break` и `continue`. Кроме этих двух инструкций, в Python существует уникальная возможность, позволяющая использовать инструкцию `else` с циклами `for` и `while`.

В этом разделе рассматриваются данные инструкции. На практических примерах я покажу, как они улучшают удобочитаемость и эффективность ваших циклов `for` и `while`.

5.4.1. Выход из циклов с помощью инструкции break

В описанной ситуации требуется механизм выхода из цикла `for` до завершения перебора всего итерируемого объекта. Для этого используется инструкция `break`, которая останавливает перебор и немедленно передает управление за пределы цикла. В этом разделе вы научитесь пользоваться данной командой. Начнем с простого примера, показывающего, как `break` работает с технической точки зрения:

```
for number in range(5):
    print(f"Number: {number}")
    if number == 2:
        print("Breaking at 2")
        break

# Выводимые строки:
Number: 0
Number: 1
Number: 2
Breaking at 2
```

Как видите, выполнение цикла `for` прекращается, когда значение `number` равно `2`; пример показывает, что команда `break` немедленно прерывает цикл `for`. Если у вас возникнут вопросы, в цикле `while` команда `break` работает точно так же:

```
number = 0
while number < 100:
    if number == 2:
        print("Breaking at 2")
        break
    else:
        number += 1
        print(f"Number: {number}")

# Выводимые строки:
Number: 1
Number: 2
Breaking at 2
```

Объединяя эти два примера, можно заметить общую закономерность использования: `break` размещается в инструкции `if` для проверки конкретного условия. В ходе итерации результат вычисления условия может измениться, и если он дает результат `True`, то `break` может немедленно прервать выполнение цикла. Я использовал циклы `for` в разных случаях. На рис. 5.5 представлена логика работы `break` в цикле `while`.

Следующим шагом станет решение практической задачи, описанной выше. Нам нужно найти только первую неотложную задачу, а получение всех неотложных задач занимает больше времени из-за перебора всего списка. Более эффективное решение использует команду `break`, как показано в листинге 5.7.

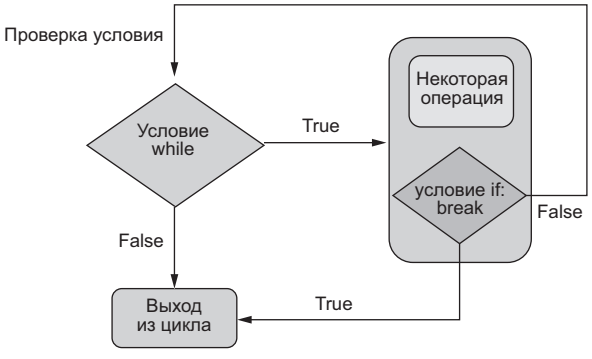


Рис. 5.5. Логика работы break в цикле while. Условие в заголовке while вычисляется при каждой итерации. Если при вычислении будет получен результат True, то выполнение передается в тело цикла while. Где-то в теле цикла размещается инструкция if, внутри которой размещается break. Если условие истинно, то выполняется команда break, а цикл while завершается. Если условие ложно, то управление передается в заголовок цикла while, где снова вычисляется условие, в зависимости от которого цикл завершается или продолжается

Листинг 5.7. Поиск неотложной задачи с использованием break

```
first_urgent_task1 = None  <-- Присваивает исходное значение

for task in tasks:
    if task.urgency == 5:
        first_urgent_task1 = task
        break

assert first_urgent_task0 == first_urgent_task1
```

В листинге 5.7 цикл `for` перебирает задачи и проверяет степень срочности для каждой задачи. При нахождении неотложной задачи итерация немедленно завершается, потому что мы получили необходимую информацию; любая дополнительная операция означает лишние затраты времени.

СОПРОВОЖДАЕМОСТЬ Переменной `first_urgent_task1` необходимо присвоить исходное значение, я присваиваю ей `None`. Если этого не сделать, то значение `first_urgent_task1` будет задаваться только в теле инструкции `if`. Может оказаться, что неотложных задач нет, и в этом случае переменная `first_urgent_task1` останется неинициализированной. При попытке присвоить значение переменной, которая не была инициализирована, происходит аварийное завершение приложения.

5.4.2. Пропуск итерации с помощью инструкции continue

Когда вы работаете с итерируемым объектом, может потребоваться применять операции только к некоторым элементам, удовлетворяющим конкретным

критериям. Ранее вы узнали, что итерируемый объект можно фильтровать (раздел 5.3.5). Но также возможен другой вариант — пропускать операции для элементов, не удовлетворяющих критерию; такой код может оказаться более удобочитаемым. В этом разделе вы научитесь использовать `continue` для пропуска итераций конкретных элементов.

Команда `continue`, как и `break`, изменяет стандартное поведение перебора — она пропускает текущую итерацию и переходит к следующей. Простой цикл `for` демонстрирует действие `continue`:

```
for number in range(5):
    if number < 3:
        continue
    print(f"Number: {number}")

# Выводимые строки:
Number: 3
Number: 4
```

Для каждой из первых трех итераций с `0`, `1` и `2` условие `if` дает результат `True`; команда `continue` выполняется, управление переходит в следующую итерацию и никакой текст не выводится. Пока `number` не достигнет значения `3`, `continue` выполняться не будет, так что итерация переходит к вызову функции `print`. На рис. 5.6 представлена логика работы `continue`.

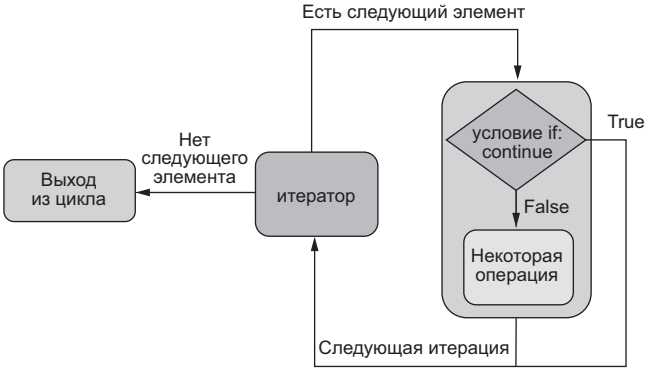


Рис. 5.6. Логика работы continue в цикле for. Когда итератор генерирует элемент, выполняется код в теле цикла for. В теле цикла вычисляется условие инструкции if. Если условие истинно, то выполняется команда continue, а цикл переходит к следующей итерации. Если условие ложно, то управление передается другим операциям, пока не начнется следующая итерация. Перебор прекращается, когда итератор исчерпает все элементы и цикл for завершится

Рассмотрим более практичный пример. Допустим, вы хотите применить серию операций к неотложным задачам. Следующая реализация без использования

continue приведена просто для демонстрации. Учтите, что код в листинге 5.8 выполняться не будет, так как методы do_something не определены.

Листинг 5.8. Применение нескольких операций к элементам при выполнении условия

```
for task in tasks:
    if task.urgency > 4:
        result0 = task.do_something0()
        result1 = task.do_something1()
        if (result0 >= 0) and (result1 == "Hello"):
            task.do_something2()
            task.do_something3()
            task.do_something4()
```

В этом примере функции применяются только к неотложным задачам. Иначе говоря, нам не нужно применять любые функции к задачам, степень срочности которых не превышает 4. В такой ситуации можно рассмотреть альтернативную реализацию с continue, приведенную в листинге 5.9.

Листинг 5.9. Пропуск итерации при выполнении условия

```
for task in tasks:
    if task.urgency <= 4:
        continue
    result0 = task.do_something0()
    result1 = task.do_something1()
    if (result0 < 0) or (result1 != "Hello"):
        continue
    task.do_something2()
    task.do_something3()
    task.do_something4()
```

Сравнив реализации из листингов 5.8 и 5.9, можно увидеть, что главным различием становится использование противоположных условий в двух местах. Существуют ли более принципиальные различия? С точки зрения быстродействия различий нет, но реализации могут различаться по удобочитаемости. Если серия операций должна применяться к элементам, удовлетворяющим критерию, то решение с обратным критерием и командой continue обычно лучше читается (рис. 5.7).

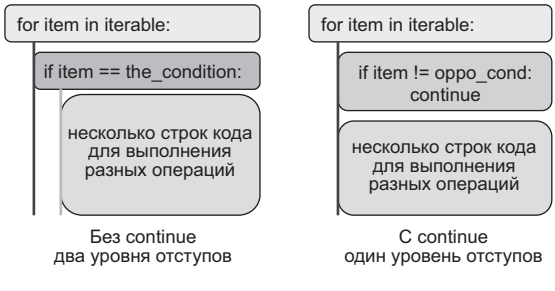


Рис. 5.7. Сокращение уровней отступов с командой continue. Без continue в цикле for потребуются два уровня отступов. С обратным условием тому же циклу for потребуется только один уровень отступов

Как видно из листинга 5.7, `continue` сокращает количество необходимых уровней отступов. Таким образом, наш код лучше читается благодаря удалению кода с большим уровнем вложенности. Сравните листинги 5.7 и 5.8.

5.4.3. Использование else в циклах for и while

Вы уже знаете, что инструкция `else` может использоваться вместе с инструкцией `if`. По сути, инструкция `if...else...` создает в программе логическое разветвление в зависимости от результата проверки условия. Если условие истинно, то выполняются операции в секции `if`; в противном случае выполняются операции в секции `else`. Важно, что операции в этих двух инструкциях являются взаимоисключающими, то есть выполнен будет только один из двух вариантов.

В большинстве языков программирования инструкция `else` существует только в инструкции `if...else...`. Python необычен в этом отношении, он позволяет использовать `else` в циклах `for` и `while`. Учтите, что добавление `else` в цикл `for` или `while` не относится к распространенной практике и может привести в замешательство многих программистов Python, особенно начинающих. Хотя инструкцию `else` следует применять в циклах `for` и `while` с осторожностью, полезно знать и использовать эти возможности в подходящих ситуациях. Такие ситуации рассматриваются в этом разделе.

Использование else в цикле for

Если вы присоединяете инструкцию `else` к циклу `for`, она имеет следующую структуру:

```
for item in iterable:
    # операции
else:
    # другие операции
```

В отличие от взаимоисключающего характера выполнения `if` и `else` в инструкциях `if...else...`, инструкция `else` не определяет альтернативную ветвь для цикла `for` (или его итераций). Инструкция `else` выполняется только один раз после завершения перебора, но пропускается, если перебор завершается командой `break`. Это правило продемонстрировано в листинге 5.10.

Листинг 5.10. Как работает инструкция for… else

```
def show_for_else_rule(breaking_number):
    for number in range(2):
        print(f"Iteration: {number}")
        if number == breaking_number:
            print(f"Break: {number}; Skip the else statement")
            break
    else:
        print("Running the else statement")
```

```
    print("Outside the for...else...")

show_for_else_rule(1)
# Выводимые строки
Iteration: 0
Iteration: 1
Break: 1; Skip the else statement
Outside the for...else...

show_for_else_rule(3)
# Выводимые строки:
Iteration: 0
Iteration: 1
Running the else statement
Outside the for...else...
```

Как видите, выполнение инструкции `else` зависит от того, выполняется ли команда `break`. В двух словах, выполнение `break` -> пропуск `else`, а без `break` -> выполнение `else`. Таким образом, если перебор не включает команды `break`, не присоединяйте `else`, потому что она будет выполнена в любом случае. Иначе говоря, инструкцию `for...else...` следует использовать в том случае, если вы точно знаете, что в процессе перебора будет участвовать команда `break`.

Рассмотрим практический пример. Допустим, имеется список задач, и вы хотите найти первую задачу с нужной степенью срочности. Возможное решение с `for... else...` представлено в листинге 5.11.

Листинг 5.11. Практический пример использования инструкции for… else

```
def locate_task(urgency_level):
    for task in tasks:
        if task.urgency == urgency_level:
            working_task = task
            break
    else:
        working_task = None
    print(f"Working Task: {working_task}")

locate_task(1)
# Вывод: Working Task: None

locate_task(4)
# Вывод: Working Task: Task(title='Camera',
➥ description='Export photos', urgency=4)
```

Если при переборе в листинге 5.11 обнаруживается задача с нужной степенью срочности, цикл прерывается таким образом, что инструкция `else` будет пропущена. Но если все итерации будут завершены без срабатывания команды `break` (например, если у всех задач степень срочности равна 1), будет выполнена инструкция `else` и вы получите вывод с `None` вместо описания задачи.

Использование else в цикле while

Инструкция `else` также может присоединяться к циклам `while`. При этом образуется следующая структура:

```
while the_condition:
    # операции
else:
    # другие операции
```

Инструкция `while...else...` выполняется по тем же правилам, что и `for...else...` : выполнение `break` -> пропуск `else`, без `break` -> выполнение `else`. На рис. 5.8 представлена логика циклов `for` и `while`.

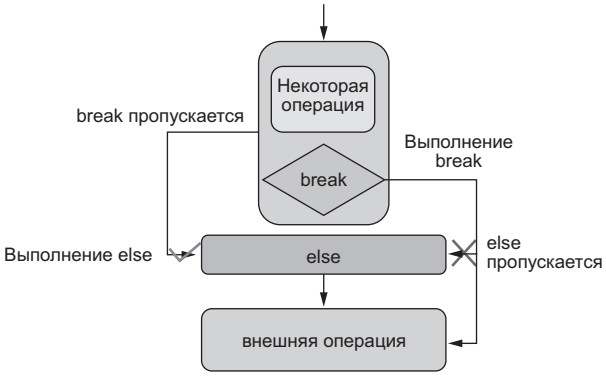


Рис. 5.8. Логика работы else в циклах for и while. Если в цикле выполняется команда break, то цикл немедленно завершается, а инструкция else пропускается. Если цикл завершается нормально без команды break, то инструкция else выполняется

Для примера допустим, что вы хотите отдохнуть после завершения серии задач в каждом сеансе. Чтобы работа была эффективной, мы устанавливаем в качестве порога отдыха сумму степеней срочности всех завершенных задач. В листинге 5.12 приведена возможная реализация с инструкцией `while...else...`.

Листинг 5.12. Практический пример использования инструкции while… else…

```
def complete_tasks_with_break(resting_threshold):
    completed_urgency_levels = 0
    while tasks:  <-- Если список не пуст, то он интерпретируется как True
        if completed_urgency_levels > resting_threshold:
            print("Coffee break now!")
            break
        next_task = tasks.pop()  <-- pop удаляет и возвращает последний элемент объекта list
        print(f"Completed: {next_task}")
        completed_urgency_levels += next_task.urgency
    else:
        print("Party! Completed all the tasks.")
```

```
tasks = [
    Task("Toaster", "Clean the toaster", 2),
    Task("Camera", "Export photos", 4),
    Task("Homework", "Physics and math", 5),
    Task("Floor", "Mop the floor", 3),
    Task("Internet", "Upgrade plan", 5)
]

complete_tasks_with_break(7)
# Выводимые строки:
Completed: Task(title='Internet', description='Upgrade plan', urgency=5)
Completed: Task(title='Floor', description='Mop the floor', urgency=3)
Coffee break now!

complete_tasks_with_break(6)
# Выводимые строки:
Completed: Task(title='Homework', description='Physics and math', urgency=5)
Completed: Task(title='Camera', description='Export photos', urgency=4)
Coffee break now!

complete_tasks_with_break(5)
# Выводимые строки:
Completed: Task(title='Toaster', description='Clean the toaster', urgency=2)
Party! Completed all the tasks.
```

> **УДОБОЧИТАЕМОСТЬ** При проверке того, не пуст ли контейнер данных или объект последовательности (например, `str`, `list` или `dict`), желательно использовать сам объект, скажем, `if tasks` и `while tasks`. Если объект tasks содержит элементы, то он интерпретируется как `True`. Непитоническое или плохо читаемое решение основано на проверке длины таких объектов, например, `if len(tasks) > 0`.

Первые два вызова функции `complete_tasks_with_break` включают выполнение команды `break`, и инструкция `else` в этих случаях пропускается. А вот третий вызов завершает итерации без выполнения `break`, поэтому инструкция `else` будет выполнена.

5.4.4. Обсуждение

Следует четко понимать, в каких случаях стоит использовать для итераций ту или иную разновидность цикла (`for` или `while`). Используйте циклы `for`, если у вас изначально имеется итерируемый объект, а количество итераций зависит от количества элементов, которые могут генерироваться итерируемым объектом. Напротив, используйте циклы `while`, если вы не уверены в том, сколько итераций будет выполнено, так как цикл `while` постоянно проверяет конкретное условие для определения того, когда цикл должен завершиться.

Старайтесь не использовать `else` с циклами `for` и `while`, потому что с этой практикой знакомы далеко не все и она может привести в замешательство многих программистов. Я не рекомендую пользоваться этой возможностью в вашей кодовой базе и описываю ее только на случай, если она будет применяться другими программистами.

СОПРОВОЖДАЕМОСТЬ Избегайте использования else с циклами for и while, потому что это может сбить с толку читателей кода.

5.4.5. Задача

В листинге 5.7 переменной `first_urgent_task1` присваивается исходное значение `None`. Как упоминалось ранее, присвоить исходное значение важно, потому что нет гарантий, что неотложная задача будет найдена в другом наборе задач. Допустим, вы не присвоили исходное значение и использовали список задач, не содержащий ни одной неотложной задачи. Посмотрим, что произойдет при попытке обращения к переменной `first_urgent_task1`.

ПОДСКАЗКА Если переменной не было присвоено значение, Python не сможет определить, какой смысл вы вкладываете в эту переменную.

ИТОГИ

- Итерируемые объекты преобразуются в итераторы при помощи функции `iter`. Итераторы — объекты данных, которые могут последовательно выдавать свои элементы при использовании функции `next`.
- Распространенные контейнеры данных, такие как `list`, `dict`, `set` и `tuple`, могут получать итерируемые объекты, чтобы создавать экземпляры с использованием соответствующих конструкторов. Каждый раз, когда у вас имеются итерируемые объекты любого типа и вы хотите создать контейнеры данных из существующих итерируемых объектов, сначала рассмотрите возможность использования этих конструкторов.
- Списковые включения, словарные включения и включения множеств представляют собой компактные способы создания объектов `list`, `dict` и `set` соответственно. Они устраняют необходимость использования обычных циклов `for` для создания экземпляров. Впрочем, если вы не оперируете элементами, для создания экземпляра проще напрямую использовать конструктор.
- Циклы `for` используются для выполнения итераций с итерируемыми объектами. Они предоставляют фундаментальную возможность применения

одних и тех же операций к группе элементов, хранящихся в итерируемом объекте. Чтобы циклы `for` были более эффективными, следует знать расширенные операции с существующими итерируемыми объектами — `enumerate`, `reversed`, `zip`, `chain` и `filter`. Функция `chain` из этого набора является частью модуля `itertools`, содержащего дополнительные расширенные операции с итерациями.

- Как циклы `for`, так и циклы `while` могут включать три необязательные инструкции: `break`, `continue` и `else`. Инструкция `break` немедленно выходит из цикла, `continue` пропускает текущую итерацию, а `else` выполняется в том случае, если цикл не был прерван инструкцией `break`. Вы должны знать, в каких ситуациях уместно использовать эти инструкции.

Часть 2
Определение функций

В части 1 вы научились пользоваться встроенными моделями данных для представления реальных задач в ваших приложениях. Впрочем, преобразование реальных задач в абстрактные модели данных станет лишь первым шагом при построении приложения. Данные можно сравнить с сырьем; чтобы преобразовать сырье в продукт, необходимо использовать соответствующее оборудование и следовать конкретному технологическому протоколу. В наших приложениях роль оборудования отводится функциям, а алгоритмы функций определяют протокол. Понятно, что сырье (данные) невозможно обработать, если у вас нет необходимого оборудования и протокола (функций и подробностей их реализации). В этой части книги рассматриваются различные приемы написания функций — движущей силы, лежащей в основе обработки данных в любом приложении.

6 Определение дружественных к пользователю функций

В этой главе

- ✓ Выбор аргументов по умолчанию для функции
- ✓ Выбор и использование возвращаемого значения для функции
- ✓ Применение аннотаций типов к параметрам и возвращаемому значению
- ✓ Определение функций с переменным количеством позиционных и ключевых аргументов
- ✓ Создание doc-строк для функции

В предыдущих главах вам уже встречались примеры функций. Вообще говоря, независимо от назначения приложения мы всегда определяем в нем функции для выполнения разных операций, например вычисления и форматирования строк. При работе в командах часто приходится определять функции, которые позволяют коллегам повторно использовать ваш код. Публикуемый пакет Python должен содержать четко определенные функции, предназначенные для пользователей, такие как встроенные функции из стандартной библиотеки Python. Следовательно, очень важно определять функции, дружественные к пользователю; даже если вы работаете не в команде, применять их вам самим должно быть легко.

Говоря о *дружественных* функциях, я имею в виду функции с понятным смыслом, с правильными аннотациями типов для аргументов и удобные в использовании, возможно, также имеющие аргументы по умолчанию. Если смысл функции вполне очевиден, пользователь сможет сам найти для нее необходимую справочную информацию (обычно в форме doc-строк).

В этой главе рассматриваются основные приемы для создания дружественных к пользователю функций. Когда мы займемся построением таск-менеджера в главе 14, мы рассмотрим практическое применение всех этих приемов, подчеркивающее важную роль функций в любом проекте.

6.1. КАК ОПРЕДЕЛИТЬ АРГУМЕНТЫ ПО УМОЛЧАНИЮ ДЛЯ УПРОЩЕНИЯ ВЫЗОВА ФУНКЦИЙ

В зависимости от конкретных требований функции могут не получать ни одного или получать несколько аргументов. Функцию с меньшим количеством аргументов проще вызывать; в идеале функция проще всего вызывается, если она вообще не получает аргументов. Если функция содержит несколько аргументов, их количество при вызове можно уменьшить, определяя для некоторых из них значения по умолчанию.

Самое большое преимущество аргументов по умолчанию — *удобство*. Не нужно задавать параметры, если аргументы по умолчанию содержат именно те значения, которые вам нужны. А поскольку функции необходима *гибкость*, всегда можно переопределить значения по умолчанию, задав подходящие аргументы.

6.1.1. Вызов функций с аргументами по умолчанию

Определение значений по умолчанию — распространенный прием для упрощения вызова функций, часто встречающийся в стандартной библиотеке Python. В этом разделе приведены примеры, которые помогут вам приобрести практический опыт вызова функций с аргументами по умолчанию.

Хотя аргументы по умолчанию не обсуждались явно в предшествующих главах, мы уже неоднократно пользовались этой возможностью. Например, в разделе 3.2 рассматривается использование метода `sort` объектов `list`, как в следующем фрагменте кода:

```
numbers = [4, 5, 7, 2]

numbers.sort()

assert numbers == [2, 4, 5, 7]
```

Если вам потребуется отсортировать список `numbers` по убыванию, вызовите метод `sort` и передайте параметр `reverse`:

```
numbers.sort(reverse=True)

assert numbers == [7, 5, 4, 2]
```

Взгляните на заголовок метода `sort`: `sort(*, key=one, reverse=False)`. Обратите внимание: для параметров `key` и `reverse` определены значения по умолчанию: `None` и `False`. Значения по умолчанию для таких параметров часто называются *аргументами по умолчанию*.

> **ОБРАТИТЕ ВНИМАНИЕ** Звездочка в заголовке метода `sort` требует, чтобы все аргументы, следующие за ней, передавались с указанием имен параметров, например `numbers .sort(reverse=True)`. С другой стороны, вызов `numbers.sort(True)` считается недействительным. Такой способ передачи называется *передачей ключевых аргументов*. За дополнительной информацией обращайтесь к разделу 6.4.1.

Когда разработчики основной функциональности Python определяли метод `sort`, они учитывали, что при сортировке объекта `list` в большинстве случаев используется лексикографический или числовой порядок и элементы обычно упорядочиваются по возрастанию, поэтому в качестве аргументов по умолчанию для параметров `key` и `reverse` передаются `None` и `False`. При использовании `sort` для объекта `list` обычно используется форма `the_list.sort()`, которая интерпретируется как `the_list.sort(key=None, reverse=False)` из-за аргументов по умолчанию, заданных в определении функции.

6.1.2. Определение функций с аргументами по умолчанию

Функции с аргументами по умолчанию отличаются не только удобством вызова, но и гибкостью с поддержкой разных сценариев использования. В этом разделе рассматривается общий процесс определения функций с аргументами по умолчанию.

При исходном определении функция обычно имеет одну конкретную цель, для чего ей передается один или несколько аргументов. Допустим, когда пользователь завершает задачу в таск-менеджере, ее статус должен обновиться. Для этого можно определить функцию `complete_task`. Вообще говоря, эта функция должна определяться как метод экземпляра (раздел 8.2). Здесь я определяю ее за пределами класса `Task` только для простоты вызова:

```
class Task:  <-- Определяет пользовательский класс
    def __init__(self, title, description, urgency):
        self.title = title
        self.description = description
        self.urgency = urgency

def complete_task(task):
```

```
    task.status = "completed"
    print(f"{task.title}'s status: completed")

task = Task("Homework", "Physics and math", 5)
complete_task(task)
# Вывод: Homework's status: completed
```

ЗАБЕГАЯ ВПЕРЕД Мы используем пользовательский класс вместо модели данных на базе именованных кортежей. Пользовательский класс предоставляет возможность гибкого изменения атрибутов экземпляра, чего нельзя сделать с моделью именованного кортежа (раздел 3.3). Определение пользовательских классов рассматривается в главе 8.

Когда пользователь завершает свою задачу, ее статус меняется на `"completed"` (завершено); это именно то, что делает функция. Позднее выясняется, что нужно дать пользователю возможность добавить заметку о завершении задачи, то есть расширить возможности функции. С появлением дополнительной возможности функция преображается таким образом:

```
def complete_task(task, note):
    task.status = "completed"
    task.note = note
    print(f"{task.title}'s status: completed; note: {note}")
```

После обновления этой функции проявляются две проблемы. Во-первых, нам приходится обновлять старый код при вызове `complete_task(task)`, так как в нем отсутствует аргумент. Во-вторых, в большинстве других мест статус обновляется без ввода заметок:

```
# Сценарий 1
complete_task(task1, "")

# Сценарий 2
complete_task(task2, "")

# Сценарий 3
complete_task(task3, "")
```

Как видите, в типичном варианте вызова функции вместо текста заметки передается пустая строка, что напоминает нам о принципе DRY («не повторяйтесь»): если в программе что-то повторяется, скорее всего, код нуждается в рефакторинге. В данном примере аргументу `note` в большинстве случаев присваивается пустая строка — идеальная ситуация для определения аргумента по умолчанию, чтобы его значение могло выбираться автоматически в большинстве сценариев использования:

```
def complete_task(task, note=""):
    task.status = "completed"
    task.note = note
    print(f"{task.title}'s status: completed; note: {note}")
```

После обновления функции, если при вызове не нужно задавать текст заметки, достаточно выполнить команду

```
complete_task(task)
```

Помимо удобной возможности пропуска аргумента при вызове, важным оказывается то, что обновление функции не нарушает работоспособность старого кода, в котором функция вызывается только с аргументом `task`. Благодаря аргументу по умолчанию в обновленном определении функции вызов `complete_task(task)` в старом коде автоматически интерпретируется как `complete_task(task, "")`.

> **СОПРОВОЖДАЕМОСТЬ** При обновлении функций лучше поддерживать одну и ту же сигнатуру вызова, чтобы существующий код продолжал работать в неизмененном виде.

На рис. 6.1 изображена общая логика процесса эволюции функции с расширением ее функциональности за счет использования аргументов по умолчанию. На диаграмме определяются две роли: разработчик API (программного интерфейса приложения), который определяет функцию, и потребитель API, который использует функцию при построении приложения. В зависимости от размера команды эти роли могут быть закреплены за разными людьми или за одним и тем же человеком. В небольших проектах нельзя исключать, что обе роли будут принадлежать одному человеку.

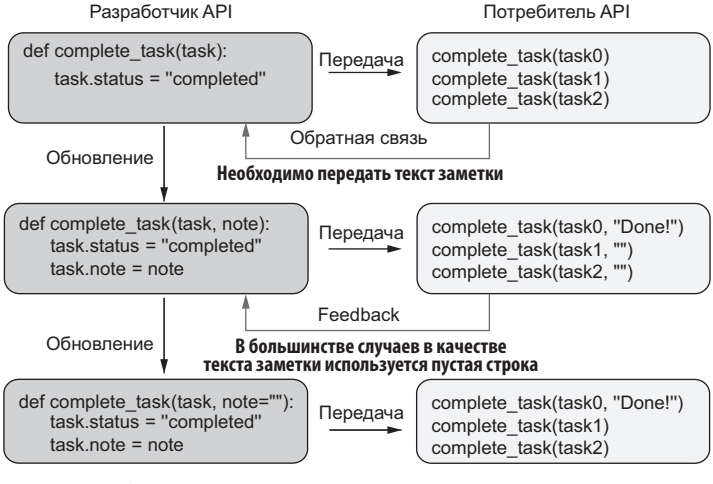


Рис. 6.1. Общий процесс создания функции с аргументами по умолчанию. Когда разработчики API получают обратную связь от потребителей, они добавляют необходимый параметр, чтобы потребитель мог указать текст заметки. Позднее потребители осознают, что постоянно передавать пустую строку слишком утомительно, и просят разработчика изменить API. Разработчик использует аргументы по умолчанию для обновления API, что отменяет необходимость в задании параметра note, если в аргументе должна передаваться пустая строка

С точки зрения потребителя, вызов функции с аргументами по умолчанию позволяет не указывать аргументы, в этом случае им автоматически присваиваются заранее заданные значения. С точки зрения разработчика, когда вы упрощаете вызов определенной функции, сокращение количества параметров снижает вероятность случайной ошибки. Тем самым вы улучшаете опыт взаимодействия потребителя и пользователя в двух отношениях:

- Вы предоставляете расширенную функциональность в существующей функции. Функция становится более гибкой с несколькими вариантами функциональности.
- Вы обеспечиваете работоспособность существующего кода, в котором используется старая сигнатура вызова. Отсутствующий аргумент будет интерпретирован со значением по умолчанию.

6.1.3. Предотвращение проблем с назначением аргументов по умолчанию для изменяемых параметров

В разделе 6.1.2 объясняется, зачем мы определяем аргументы по умолчанию и как эволюционируют функции, использующие аргументы по умолчанию. В наших примерах использовался аргумент по умолчанию с типом `str`. Как упоминалось в главе 3, строки неизменяемы.

Другую категорию моделей данных составляют изменяемые объекты, например списки и словари. В этом разделе вы узнаете, как назначать аргументы по умолчанию для изменяемых параметров.

> **Как правильно: аргументы или параметры?**
>
> Может показаться, что термины «аргументы» и «параметры» являются синонимами для обозначения переменных, используемых в функции. Тем не менее между ними существуют смысловые различия. Когда вы определяете функцию, переменные, указанные в ее заголовке, называются *параметрами*. При вызове функций используемые переменные называются *аргументами*.

Допустим, при завершении задачи существует необязательная возможность добавить ее в группу отслеживаемых задач. Начать можно со следующей рабочей версии:

```
def complete_task(task, grouped_tasks=[]):
    task.status = "completed"
    grouped_tasks.append(task.title)  <-- Для простоты используем только title
    return grouped_tasks
```

Параметру `grouped_tasks` в качестве аргумента по умолчанию присваивается пустой объект `list`. Предполагается, что если функция вызывается без указания

аргумента `grouped_tasks`, будет создан пустой объект `list`. Результат продемонстрирован в листинге 6.1.

Листинг 6.1. Использование функций с изменяемыми аргументами по умолчанию

```
task0 = Task("Homework", "Physics and math", 5)
task1 = Task("Fishing", "Fishing at the lake", 3)

work_tasks = complete_task(task0)
play_tasks = complete_task(task1)

print("Work Tasks:", work_tasks)
print("Play Tasks:", play_tasks)

# Выводимые строки:
Work Tasks: ['Homework', 'Fishing']
Play Tasks: ['Homework', 'Fishing']
```

Как показано в листинге 6.1, для каждого вызова функции `complete_task` без указания `grouped_tasks` мы хотим получить новый объект `list`, содержащий завершенную задачу. Но как ни странно, оба объекта `list` содержат одинаковые элементы, хотя ожидалось, что `work_tasks` и `play_tasks` будут содержать `['Homework']` и `['Fishing']` соответственно. Внимательно присмотревшись к двум объектам `list`, вы обнаружите, что это один и тот же объект:

```
assert work_tasks == play_tasks
assert work_tasks is play_tasks
```

is проверяет, что две переменные относятся к одному объекту

Это происходит из-за того, что Python вычисляет функцию при ее определении. У вычисления имеется побочный эффект: любые изменяемые аргументы по умолчанию создаются во время вычисления и становятся частью функции. В нашем примере объект `list` создается при вычислении функции. Теперь этот конкретный объект `list` используется в качестве аргумента `grouped_tasks` при каждом вызове функции без указания аргумента `grouped_tasks`, как показывает код в листинге 6.2.

Листинг 6.2. Использование единого изменяемого объекта, определенного в функции

```
def append_task(task, tasks=[]):
    tasks.append(task)
    print(f"Tasks: {tasks}; id: {id(tasks)}")

append_task.__defaults__
# Вывод: ([],)

id(append_task.__defaults__[0])
```

Функция id возвращает адрес памяти, однозначно идентифицирующий объект

__defaults__ получает объекты по умолчанию, связанные с функцией

```
# Вывод: 4356663616

append_task("Homework")
# Вывод: Tasks: ['Homework']; id: 4356663616

append_task("Laundry")
# Вывод: Tasks: ['Homework', 'Laundry']; id: 4356663616

append_task.__defaults__
# Вывод: (['Homework', 'Laundry'],)
```

В листинге 6.2 встроенная функция `id` используется для проверки адреса памяти объекта. Если операции выполняются с одним и тем же объектом, функция `id` возвращает один и тот же адрес памяти. Как видите, при вызове функции без указания аргумента `tasks` вы получаете один и тот же объект, созданный при определении функции.

CPython и функция id

Написанный вами код Python выполняется на вашем компьютере (машине). Заметим, что код Python не взаимодействует с машиной напрямую. Перед выполнением он должен быть откомпилирован в байт-код. Существуют разные реализации для компиляции кода Python, самая распространенная называется CPython. Это исходная реализация Python, которую можно загрузить с официального сайта Python. Другие реализации, такие как Jython, компилируют код Python в байт-код Java.

В CPython функция `id` возвращает адрес памяти объекта на текущий момент. Таким образом, если вы будете выполнять функцию `id` в одном коде в разное время или на разных машинах, можно ожидать, что адреса памяти будут разными. Заметим, что в других реализациях Python для функции `id` могут использоваться другие идентификаторы.

Если `[]` или `list()` не могут использоваться в качестве значения по умолчанию для параметра `list`, то как быть? Означает ли это, что для изменяемого параметра нельзя задать значение по умолчанию? Нет, не означает. На практике принято использовать `None` в качестве аргумента по умолчанию для изменяемых параметров. Эта схема представлена в листинге 6.3.

Листинг 6.3. Использование None в качестве значения по умолчанию для изменяемых параметров

```
def complete_task(task, grouped_tasks=None):
    task.status = "completed"
    if grouped_tasks is None:  <-- Сравнивая объект с None, используйте is вместо ==
        grouped_tasks = []
    grouped_tasks.append(task.title)
```

```
    return grouped_tasks

complete_task.__defaults__
# Вывод: (None,)
```

Как видите, аргумент по умолчанию для функции равен `None`. В теле функции мы сравниваем аргумент `grouped_tasks` с `None`, и если условие истинно, то создаем новый объект `list`. Каждый раз, когда эта функция вызывается без указания аргументов `grouped_tasks`, функция создает новый объект `list`, что и требовалось.

СОПРОВОЖДАЕМОСТЬ Если вы задаете аргумент по умолчанию для изменяемого параметра функции, присвойте ему `None`.

6.1.4. Обсуждение

Назначение аргументов по умолчанию в определениях функции часто встречается в стандартной библиотеке Python. Кроме метода `sort`, многие встроенные функции (в частности, `sorted` и `print`) включают аргументы по умолчанию. С такими аргументами эти функции проще вызывать; кроме того, они обеспечивают гибкость при передаче разных аргументов. Следует помнить о различиях между изменяемыми и неизменяемыми параметрами. Если задать неправильный аргумент по умолчанию для изменяемого параметра, это может привести к появлению скрытых ошибок в кодовой базе.

6.1.5. Задача

Кори преподает программирование на Python в колледже. Он хочет показать студентам, что аргументы по умолчанию вычисляются при определении функции, а не при ее вызове. Как бы вы предложили ему подкрепить это утверждение?

ПОДСКАЗКА Создайте временную метку для проверки того, что происходит во время определения и вызовов функции. Следующий фрагмент позволяет получить временную метку:

```
from datetime import datetime
timestamp = datetime.today()
```

6.2. КАК ОПРЕДЕЛИТЬ И ИСПОЛЬЗОВАТЬ ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ ПРИ ВЫЗОВАХ ФУНКЦИЙ

Функции определяются для выполнения конкретной операции. Мы вызываем их с передачей нужных аргументов — входных данных функций. Завершая свою

операцию, функции возвращают значение — результат их работы. К настоящему моменту вы уже должны понимать, какую важную роль играют функции в ваших приложениях; следовательно, очень важно уметь работать не только с входными данными (например, задавать аргументы по умолчанию, раздел 6.1), но и с результатом вызова функций. В этом разделе мы сосредоточимся на том, как задать возвращаемое значение функции и как его использовать.

6.2.1. Явное и неявное возвращение значения

В наших примерах использовались многие встроенные и пользовательские функции. Некоторые функции возвращают значение; другие, как кажется, этого не делают. В этом разделе я покажу, что любая функция Python возвращает значение, хотя иногда и неявно.

Встроенная функция `sum` предназначена для суммирования элементов итерируемого объекта. Неудивительно, что возвращаемым значением становится вычисленная сумма:

```
numbers = list(range(5))

sum_numbers = sum(numbers)

print(f"Sum of {numbers} is {sum_numbers}")
# Вывод: Sum of [0, 1, 2, 3, 4] is 10
```

В разделе 3.2 вы научились пользоваться функцией `sort` для упорядочения элементов объекта `list`. Следует заметить, что метод `sort` сортирует объект `list` на месте (in place), а это означает, что сортировка изменяет исходный объект `list`. Если при этом проверить возвращаемое значение `sort`, вы увидите, что оно равно `None`:

```
primes = [5, 7, 2, 3, 11]

sort_return_value = primes.sort()

print(f"Return value of sort: {sort_return_value}")
# Вывод: Return value of sort: None
```

Эти два примера подтверждают, что каждая функция возвращает значение. Следует четко осознавать, что именно она возвращает: `None` или что-то еще. Не стоит полагаться на интуицию, потому что вы можете допустить глупые ошибки при построении цепочки вызовов методов. Следующий проблемный фрагмент пытается отсортировать список `primes` и присоединить `13` в конец списка:

```
primes.sort().append(13)
```

> **ВОПРОС** А вы знаете, почему этот код не работает? Проверьте, что возвращает `sort`.

6.2.2. Определение функций, возвращающих 0, 1 или несколько значений

Чтобы понять, как функции возвращают значения, лучше всего рассмотреть несколько примеров с разными вариантами поведения. В общем случае функция может возвращать ноль значений, одно или несколько.

Возвращение нуля значений

Строго говоря, вы не сможете определить функцию, которая не возвращает значения. Как упоминалось в разделе 6.2.1, каждая функция имеет возвращаемое значение, явное или неявное. При определении функции, которая ничего не возвращает, она все равно возвращает `None`. Рассмотрим следующий пример:

```
def append_task(task, grouped_tasks):
    grouped_tasks.append(task)

appended_no_return = append_task("Homework", [])

print(f"Appended: {appended_no_return}")
# Вывод: Appended: None
```

Как видите, в определении функции нет инструкции `return`. Но если проверить возвращаемое значение, выясняется, что `appended_no_return` содержит `None`. Этот результат соответствует обсуждению в разделе 6.2.1. На рис. 6.2 изображена общая схема определения функции без явного возвращения переменной.

Рис. 6.2. Неявное возвращение значения функцией. Если функция не содержит инструкции return, это эквивалентно тому, что функция возвращает None

> **ВОПРОС** Какое значение вернет функция, содержащая только инструкцию `return`?

Возвращение одного значения

Чаще всего функции все же возвращают значение. Вы уже знаете, что функции обычно проектируются для выполнения конкретной операции. Как правило, операция должна иметь одно выходное значение, и этот факт устраняет любую неоднозначность относительно предназначения функции. Таким образом, в большинстве случаев надо стремиться к тому, чтобы ваши функции возвращали только одно значение.

Вкратце рассмотрим процесс присваивания переменной возвращаемого значения функции — самой распространенной формы вызова функции. Пример:

```
def say_hello(person):
    hello = f"Hello, {person}!"
    return hello

greeting = say_hello("Rocky")
```

В этом фрагменте кода продемонстрирован типичный сценарий использования: вызов функции `say_hello` и присваивание ее возвращаемого значения переменной `greeting`. Вы точно понимаете, что в этом случае происходит «за кулисами»? Если понимаете, то можете переходить к следующему разделу, а если нет, взгляните на рис. 6.3.

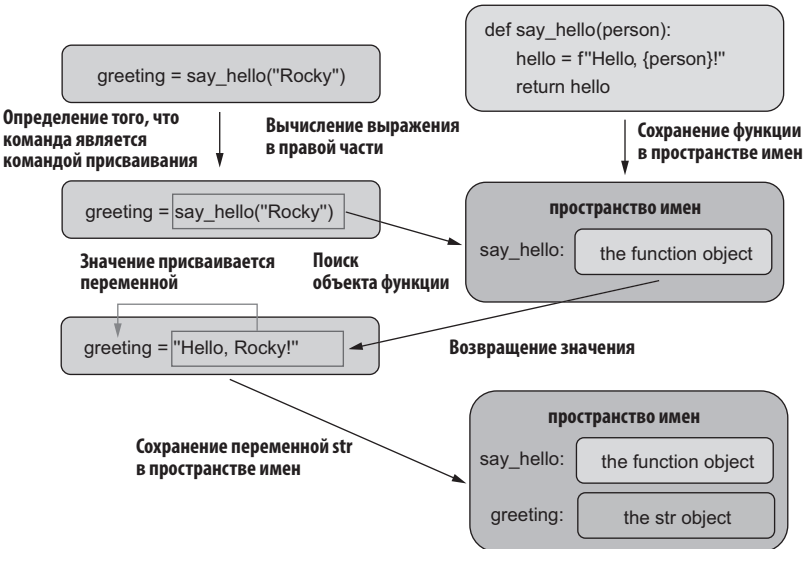


Рис. 6.3. Процесс создания переменной на основе вызова функции. Когда вы определяете функцию, она сохраняется в пространстве имен. При вызове функции Python ищет эту функцию в пространстве имен и вызывает ее с переданными аргументами. После завершения вызова функции возвращаемое значение присваивается переменной. После присваивания новая переменная загружается в то же пространство имен, чтобы ее можно было найти при последующих обращениях

ОСНОВНЫЕ ПОНЯТИЯ *Пространство имен* (namespace) представляет собой набор определенных переменных, которые можно искать и использовать. Пространство имен можно рассматривать как объект словаря: идентификаторы (такие, как имя функции) являются ключами, а соответствующие объекты — значениями. Пространства имен подробно рассматриваются в разделе 10.2.

При создании переменной на основе вызова функции используется инструкция присваивания. Инструкция присваивания вычисляет выражение в правой части; в нашем случае это вызов функции, которая ищется в текущем пространстве имен. После завершения определенной в функции операции в результате выполнения возвращается значение, которое присваивается переменной `greeting`.

Возвращение нескольких значений

Если в функции выполняются сложные операции, они могут генерировать два и более объекта, и все эти объекты понадобятся для последующей обработки. В таком случае стоит рассмотреть возможность возвращения всех этих объектов в выходных данных функции.

Как вам, вероятно, известно, ученые обычно ищут среднее значение и стандартное отклонение для всех измерений в эксперименте. Допустим, вы определяете функцию, которая поможет ученым справиться с этой задачей. Возможное решение приведено в листинге 6.4.

Листинг 6.4. Возвращение нескольких значений из функции

```
from statistics import mean, stdev

def generate_stats(measures):
    measure_mean = mean(measures)
    measure_std = stdev(measures)
    return measure_mean, measure_std
```

Функция `generate_stats` возвращает среднее значение и стандартное отклонение одновременно, что упрощает кодовую базу. Непитоническое решение могло бы использовать две разные функции, каждая из которых возвращает только одно значение:

```
def calculate_mean(measures):
    measure_mean = mean(measures)
    return measure_mean

def calculate_std(measures):
    measure_std = stdev(measures)
    return measure_std
```

В листинге 6.4 значения `measure_mean` и `measure_std` тесно связаны, так как они образуют статистические характеристики этих экспериментальных метрик; таким образом, листинг является допустимым примером возвращения двух значений из функции.

С другой стороны, если вы пытаетесь вернуть два значения, никак не связанных между собой, то скорее всего, ваша функция состоит из смешанных операций, которые служат разным целям. Плохо определенная функция представлена в следующем фрагменте кода:

```
def process_data(measures):
    formatted_measures = [f"{x} mg/L" for x in measures]
    measure_mean = mean(measures)
    return formatted_measures, measure_mean
```

Как видите, в функции `process_data` возвращаемые значения не связаны друг с другом. Когда другие люди будут пользоваться этой функцией, им будет трудно разобраться в том, какой результат возвращается при ее вызове, потому что функция решает две разные задачи: форматирование характеристик и вычисление среднего значения. Более удобочитаемое решение будет определять отдельные функции для разных целей. Еще важнее сделать так, чтобы имена этих функций очевидно отражали их предназначение:

```
def format_measures(measures):
    formatted_measures = [f"{x} mg/L" for x in measures]
    return formatted_measures

def calculate_mean(measures):
    measure_mean = mean(measures)
    return measure_mean
```

СОПРОВОЖДАЕМОСТЬ Хорошо написанная функция должна иметь только одну цель. Если вам кажется, что вы «экономите» несколько строк кода за счет объединения функций, служащих разным целям, на самом деле вы усложняете чтение и использование кода, а также усложняете работу для себя и своих коллег.

6.2.3. Использование нескольких значений, возвращаемых при вызове функции

Если функция возвращает `None` или любое другое одиночное значение, использовать такой результат несложно. Но в некоторых случаях функция может возвращать несколько значений. В этом разделе речь пойдет о том, как использовать множественные значения, возвращаемые при вызове функции.

Хотя ранее я говорил о возможности определить функцию, возвращающую несколько значений, в действительности любая функция может вернуть только один объект. Взгляните на пример использования функции `generate_stats`, определенной в листинге 6.4:

```
measures = [5.6, 7.0, 5.7, 5.8, 4.3, 5.2]

measures_stats = generate_stats(measures)

print(type(measures_stats), measures_stats)
# Вывод: <class 'tuple'> (5.6, 0.8786353054595518)
```

Функция type проверяет тип данных объекта

Возвращаемым значением при вызове `generate_stats` является объект `tuple`, хотя может показаться, что в определении функции возвращаются два значения.

Эти два значения упакованы в один объект `tuple`. Другими словами, даже если вам кажется, что в определении функции возвращаются несколько значений, в действительности возвращается только одна переменная — объект `tuple`, состоящий из этих значений. Как упоминалось при описании распаковки кортежей (раздел 4.4), при создании объектов кортежей круглые скобки не обязательны.

Для работы со значениями, возвращаемыми функцией, можно воспользоваться методом распаковки кортежа — этот компактный питонический способ обращения к отдельным компонентам возвращенного объекта `tuple` продемонстрирован в листинге 6.5.

Листинг 6.5. Распаковка возвращаемого объекта tuple

```
m_mean, m_std = generate_stats(measures)

print(f"Mean: {m_mean}; SD: {m_std}")
# Вывод: Mean: 5.6; SD: 0.8786353054595518
```

ВОПРОС Что делать, если из результатов вызова функции `generate_stats` вас интересует только среднее значение?

6.2.4. Обсуждение

Каждая функция должна служить только одной цели, так что возвращение только одного значения можно считать наиболее естественной формой вывода. И хотя функция способна возвращать сколько угодно значений, увлекаться не стоит, потому что пользователям функции будет трудно разобраться в том, для чего каждое значение используется. На практике лучше ограничиться одним возвращаемым значением на функцию. В некоторых случаях можно возвращать от 2 до 4 значений, но 5 и более значений показывают, что с функцией что-то не так: например, она служит сразу нескольким целям.

6.2.5. Задача

Зоя продолжает работу над своим картографическим приложением (раздел 3.1.4). Она определяет несколько функций, возвращающих широту и долготу заданного места:

```
def locate_me():
    # Поиск текущего местоположения пользователя
    return latitude0, longitude0

def locate_home():
    # Поиск места жительства пользователя
    return latitude1, longitude1

def locate_work():
    # Поиск места работы пользователя
    return latitude2, longitude2
```

Видя закономерность повторения возвращаемых значений, вы понимаете, что этот код можно подвергнуть рефакторингу. Что следует сделать, чтобы функции возвращали одно значение?

ПОДСКАЗКА Именованные кортежи (раздел 3.3) — упрощенная модель, которая может использоваться для хранения данных.

6.3. КАК ИСПОЛЬЗОВАТЬ АННОТАЦИИ ТИПОВ ДЛЯ НАПИСАНИЯ ПОНЯТНЫХ ФУНКЦИЙ

При определении функций Python не заставляет вас определять типы аргументов и возвращаемого значения. В большинстве случаев функции получают только конкретные типы данных. Взгляните на функцию из листинга 6.4:

```
def generate_stats(measures):
    pass
```

Если пользователи не знают, с какими типами данных им придется работать, они могут вызвать функцию следующим образом:

```
generate_stats({"measure0": 7.9, "measure1": 6.8, "measure2": 7.0})
```

Этот вызов функции не работает, потому что функция предполагает, что аргумент `measures` представляет собой объект `list` или `tuple`. Таким образом, для снижения риска некорректного использования кода стоит рассмотреть возможность использования аннотаций типов в определениях функций. *Аннотации типов* (type hints) сообщают пользователю, какие аргументы получает функция и какие значения она возвращают; с ними функция становится более понятной. В следующем разделе вы узнаете, как сделать функцию более дружественной к пользователю при помощи аннотаций типов.

6.3.1. Определение аннотаций типов для переменных

В главах 1–5 вы узнали о таких распространенных моделях данных, как `str`, `list`, `tuple` и `dict`. Определяя переменную некоторого типа, вы не задаете для нее тип данных. Однако также можно указать тип данных переменной, который станет основой для применения аннотаций типов для функций. В этом разделе будут рассмотрены важнейшие навыки определения аннотаций типов для переменных. Простой пример создания переменной `int`:

```
number = 1
print(type(number))
# Вывод: <class 'int'>
```

ОБРАТИТЕ ВНИМАНИЕ Встроенная функция `type` сообщает информацию о типе объекта.

Как и ожидалось, переменная `number` имеет тип данных `int`. Если вы решите присвоить `number` другое значение (например `string`), на языке Python это можно сделать так:

```
number = "one"

print(type(number))
# Вывод: <class 'str'>
```

В этом фрагменте кода мы присваиваем строковый литерал переменной `number`, вследствие чего она приобретает тип данных `str`. Иначе говоря, мы работаем с той же переменной `number`, но ее тип данных преобразовался из `int` в `str` простым присваиванием. В терминологии программирования Python называется языком с *динамической типизацией* — тип переменной может измениться после ее создания.

Некоторые языки программирования не позволяют изменять тип переменной после ее определения; это языки со *статической типизацией*. Язык Swift, рекомендуемый для разработки приложений для iPhone и других систем семейства Apple, является языком со статической типизацией. В Swift невозможно присвоить строковое значение переменной, которая была изначально определена как целочисленная. Если переменная относится к конкретному типу, то значение другого типа не может быть использовано для повторного присваивания, как показано в листинге 6.6.

Листинг 6.6. Пример статической типизации в Swift

```
var number = 1

number = "one"
error: cannot assign value of type 'String' to type 'Int'
```

И хотя Python относится к языкам с динамической типизацией, вы можете определить аннотации типа для создаваемых в нем переменных. Эта возможность появилась в Python 3.6. Чтобы определить аннотацию типа, поставьте двоеточие (`:`) после ее имени, а затем укажите тип переменной. Приведу несколько примеров:

```
number: int = 3

name: str = "John"

primes: list = [1, 2, 3]
```

Важно знать, что аннотации типов не делают Python языком со статической типизацией и не обеспечивают принудительного соблюдения типа переменной (если вас интересует, зачем тогда нужны аннотации типов, читайте следующий раздел). При этом вы можете присвоить значение другого типа переменной,

созданной с аннотацией типа, и выполнить следующие две строки кода без малейших проблем:

```
numbers: tuple = (1, 2, 3)
numbers = [1, 2, 3]
```

6.3.2. Использование аннотаций типов в определениях функций

В разделе 6.3.1 вы научились определять аннотации типов для отдельных переменных. В этом разделе такая возможность будет применена в определении функции, чтобы продемонстрировать преимущества определения функций с аннотациями типов.

Использование аннотаций типов в определении функции почти не отличается от их использования при создании переменных, за исключением определения аннотации для возвращаемого значения. Пример в следующем листинге (измененная версия функции `generate_stats` из листинга 6.4) демонстрирует, как аннотации типов работают с функциями.

Листинг 6.7. Использование аннотаций типов в функциях

```
from statistics import mean, stdev

def generate_stats(measures: list) -> tuple:
    measure_mean = mean(measures)
    measure_std = stdev(measures)
    return measure_mean, measure_std
```

Добавление аннотаций типов к параметрам функции почти не отличается от создания переменных, и в обоих случаях используется форма `параметр: тип_данных`. С добавлением аннотаций типов к возвращаемому значению дело обстоит иначе, потому что в заголовке функции нет переменной, представляющей возвращаемое значение. Вместо этого для обозначения типа возвращаемого значения используется конструкция `-> тип_данных`. Использование аннотаций типов в определениях функций объясняется двумя основными причинами:

- *Аннотации типов ясно показывают, какие параметры получает функция и что она возвращает.* Например, вызов `help(generate_stats)` позволит вам увидеть сигнатуру функции и правильно использовать ее:

  ```
  >>> help(generate_stats)
  Help on function generatate_stats in module __main__:

  generate_stats(measures: list) -> tuple
  ```

- *Аннотации типов повышают эффективность программирования*, так как разработчик может проверить правильность типов в процессе работы. Эти

преимущества не столь очевидны, если вы работаете в консоли или текстовом редакторе, потому что ведущие интегрированные среды разработки (IDE) для языка Python способны анализировать код в реальном времени — либо на уровне встроенной функциональности, либо при установке плагинов (также называемых расширениями).

Допустим, вы определяете функцию, которая получает целые числа, и задаете эти требования при помощи аннотаций типов. На рис. 6.4 показано, как анализ кода может привести к отображению осмысленных контекстных меню, способствующих повышению качества кода.

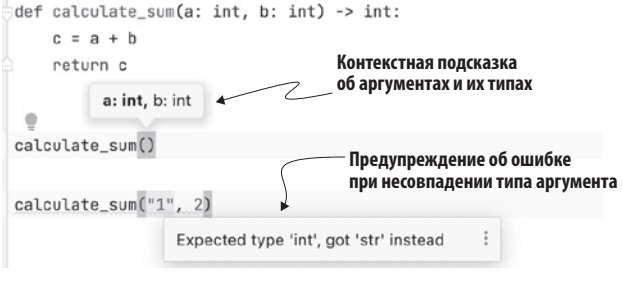


Рис. 6.4. Контекстные меню отображаются для функций с аннотациями типов в редакторе Python PyCharm. При вызове функции в контекстном меню отображаются аргументы и соответствующие им типы. Когда вы вызываете функцию с неправильными типами, в контекстном меню выводится информация о несовместимых типах

6.3.3. Нетривиальное применение аннотаций типов в определениях функций

В разделе 6.3.2 вы узнали о синтаксисе аннотаций типов в определениях функций. Впрочем, в некоторых ситуациях базовых возможностей оказывается недостаточно. В этом разделе рассматриваются нетривиальные возможности аннотаций типов:

- аргументы со значениями по умолчанию;
- пользовательские классы;
- объекты-контейнеры;
- множественные типы данных.

Использование аргументов со значениями по умолчанию

Ранее я объяснял, как задать значение по умолчанию для параметра в определении функции. Если вы объединяете эту возможность с аннотациями типов, то

все, что вам необходимо помнить, это порядок: сначала аннотация типа, затем значение по умолчанию. Пример приведен в следующем фрагменте:

```
def calculate_product(a: int, b: int, multiplier: int = 1) -> int:
    c = a * b * multiplier
    return c
```

Параметр `multiplier` имеет значение по умолчанию `1` и тип `int`. Обратите внимание: пробелы, используемые при задании значения по умолчанию и типа параметра, необходимы, потому что они упрощают чтение кода. Точнее говоря, пробелы необходимы до и после типа, а также до и после знака `=`.

УДОБОЧИТАЕМОСТЬ Пробелы и пустые строки должны присутствовать во многих местах. Они улучшают удобочитаемость кода за счет визуальной изоляции разных компонентов.

Работа с пользовательскими классами

С ростом проекта появляются новые классы для управления данными. Эти классы представляют собой новые типы, и их можно использовать так же, как мы используем встроенные типы данных — `int`, `tuple` и `dict`. Листинг 6.8 показывает, как включать пользовательские классы в определения функций с рекомендациями типов.

Листинг 6.8. Использование аннотаций типов с пользовательскими классами

```
from collections import namedtuple
Task = namedtuple("Task", "title description urgency")
class User:
    pass  <-------  Инструкция pass используется как заполнитель
def assign_task(pending_task: Task, user: User):
    pass
```

ОБРАТИТЕ ВНИМАНИЕ Инструкция `pass` используется в тех случаях, когда для выполнения синтаксических требований необходимо присутствие кода. Это заполнитель, который не делает ничего. В теле определения функции должен присутствовать код реализации класса. Однако в данном случае можно передать `pass`, чтобы определение класса считалось действительным.

Как видно из листинга 6.8, мы определяем два класса: `Task` (с использованием именованного кортежа) и `User` (с использованием типичного определения класса). Когда эти классы будут определены, их можно немедленно использовать в программе. Python знает, что эти классы являются типами и могут использоваться для обозначения типов аргументов в определении функции.

Работа с объектами-контейнерами

Ранее вы узнали, что некоторые встроенные типы данных, такие как `list` и `tuple`, называются контейнерами, потому что они могут содержать другие объекты. Когда речь заходит об аннотациях типов для таких контейнеров, можно заметить, что определение типа для самого контейнера не всегда имеет смысл. Допустим, у вас имеется функция для завершения нескольких задач, как показано в листинге 6.9.

Листинг 6.9. Аннотации типов с использованием типа контейнера

```
def complete_tasks(tasks: list):
    for task in tasks:
        pass
```

Определение функции показывает, что аргумент `tasks` является объектом `list`, но не указывает, какие объекты в этом списке находятся. Таким образом, пользователь может использовать список объектов `str` или список объектов `Task`:

```
complete_tasks(["Laundry", "Museum"])
complete_tasks([Task("Laundry", "Wash clothes", 5),
➥ Task("Museum", "Egyptian exhibit", 4)])
```

Вообще говоря, при добавлении в функцию конкретных операций можно обеспечить ее совместимость с объектами `str` или `Task`, но для пользователя было бы удобнее сделать более конкретным аргумент `tasks`. Список объектов `str` или список объектов `Task`? В листинге 6.10 приведена измененная версия этой функции.

Листинг 6.10. Аннотации типов с контейнером, имеющим конкретный тип содержимого

```
def complete_tasks_hinted(tasks: list[Task]):
    for task in tasks:
        pass
```

> **ОБРАТИТЕ ВНИМАНИЕ** Функциональность аннотаций типов развивалась в последних версиях Python. Если вы используете не самую новую версию, некоторые функции могут оказаться недоступными.

Можно не ограничиваться `list`, а поставить после `list` квадратные скобки с ожидаемым типом данных объектов, содержащихся в контейнере. В нашем случае предполагается, что объект `list` будет содержать объекты `Task`, но не объекты `str`. С этим изменением IDE выдаст предупреждение о попытке использования объекта `list` с несовместимым типом данных, например строкой (рис. 6.5).

Наряду с `list` самыми распространенными типами данных контейнеров являются `dict`, `tuple` и `set`. В табл. 6.1 приведена сводка аннотаций типов для содержимого контейнера.

Рис. 6.5. Вывод предупреждения в том случае, если контейнер содержит объекты несовместимого типа данных. Снимок экрана был сделан в Python IDE PyCharm. IDE проводит анализ кода в реальном времени, поэтому при задании аргумента, расходящегося с аннотациями типов, IDE выдает контекстное меню с предупреждением

Таблица 6.1. Аннотации типов для распространенных встроенных объектов контейнеров

Тип контейнера	Примеры кода	Объяснение
`list`	`list[str]` `list[int]`	Список `list` объектов `str` Список `list` объектов `int`
`tuple`	`tuple[float, int]` `tuple[float, ...]`	Кортеж `tuple`, содержащий объект `float` и объект `int` Кортеж `tuple`, содержащий несколько объектов `float`
`dict`	`dict[int, str]` `dict[int, list[int]]`	Словарь `dict`, в котором ключами являются объекты `int`, а значениями — объекты `str` Словарь `dict`, в котором ключами являются объекты `int`, а значениями — объекты `list`, содержащие объекты `int`
`set`	`set[int]` `set[str]`	Множество `set`, содержащее объекты `int` Множество `set`, содержащее объекты `str`

Передача нескольких типов данных

Функция может получать разные типы данных в конкретном параметре. В листинге 6.5 параметр `measures` функции `generate_stats` содержит список чисел. Но эта функция будет работать точно так же и при использовании кортежа чисел. В таком случае следует использовать аннотации типов для обозначения того, что параметр может относиться к нескольким разным типам, как показано в листинге 6.11.

Листинг 6.11. Определение нескольких типов

```
from statistics import mean, stdev

def generate_stats(measures: list[float] | tuple[float, ...])
➥   -> tuple[float, float]:
```

```
    measure_mean = mean(measures)
    measure_std = stdev(measures)
    return measure_mean, measure_std
```

Чтобы указать несколько возможных типов для параметра, перечислите их, разделяя вертикальной чертой (|). Если типов больше двух, то и символов | потребуется больше:

```
para0: int | float | str | list
```

6.3.4. Обсуждение

В ранних версиях Python аннотации типов не поддерживались, эта функциональность появилась позднее. Серьезным дополнением стандартной библиотеки Python стал модуль `typing` для расширенных аннотаций типов. То, что вы узнали в этой главе, подготовит вас к освоению новых возможностей модуля `typing`. Чтобы немного упростить вашу задачу, привожу следующий код. Он показывает, как прояснить смысл аннотаций типов при помощи модуля `typing`, так как этот модуль поддерживает высокоуровневую информацию типов (например, `Sequence` представляет любые типы данных последовательностей):

```
from statistics import mean, stdev
from typing import Sequence

def generate_stats(measures: Sequence[float]) -> tuple[float, float]:
    measure_mean = mean(measures)
    measure_std = stdev(measures)
    return measure_mean, measure_std
```

6.3.5. Задача

Эндрю строит пакет Python для обработки финансовых данных. Он использует аннотации типов в пакете, чтобы упростить жизнь пользователей. Как ему записать аннотации типов, если параметр функции может быть списком `list` с элементами `int` или списком `list` с элементами `str`?

> **ПОДСКАЗКА** Вертикальная черта означает операцию «или», которая не используется между аннотациями типов. Иначе говоря, она может использоваться *внутри* аннотации типа: например, `set[int | str]`.

6.4. КАК ПОВЫСИТЬ ГИБКОСТЬ ФУНКЦИИ ПРИ ПОМОЩИ *ARGS И **KWARGS

Определяя функции, мы хотим использовать их для решения конкретных задач. При вызове этих функций мы передаем соответствующие аргументы, чтобы они

могли выполнять необходимые операции. До настоящего момента все функции, которые мы определяли, получали заранее заданное количество аргументов, но иногда этого оказывается недостаточно. Возьмем заголовок встроенной функции `print`:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

На первый взгляд кажется, что функция `print` получает пять аргументов, причем последние четыре имеют значения по умолчанию. Но как вы, вероятно, заметили, при использовании `print` можно вывести любое количество объектов, что показано в листинге 6.12.

Листинг 6.12. Использование встроенной функции print

```
word = "Hello"
numbers = [1, 2, 3]
prime_number = 11

print(word, numbers, prime_number)
# Вывод: Hello [1, 2, 3] 11
```

Почему же функция `print` может получать любое количество объектов? Дело в символе * перед параметром `objects`, который обозначает переменное (ноль и более) количество позиционных аргументов. Такой способ передачи параметров обычно обозначается `*args`. С использованием `*args` функция `print` становится достаточно гибкой для передачи любого количества объектов. Также существует похожий способ определения переменного количества ключевых аргументов, который обозначается `**kwargs`. В следующем разделе вы научитесь использовать `*args` и `**kwargs` для определения функций, обладающих повышенной гибкостью. Кроме того, мы представим ряд ключевых концепций, относящихся к категориям аргументов.

6.4.1. Позиционные и ключевые аргументы

Возможно, вы уже заметили, что при вызове функций в круглых скобках аргументы в одних случаях перечисляются напрямую, а в других перед ними указываются идентификаторы. Для обозначения этих двух типов аргументов используются разные термины.

Аргументы, с которыми связываются идентификаторы, называются *ключевыми аргументами*; идентификаторы используются в теле функции для ссылок на такие аргументы. Если с аргументами идентификаторы не связаны, они называются позиционными аргументами. Иначе говоря, Python обрабатывает эти аргументы на основании их позиции в последовательности, указанной при определении функции. Чтобы понять различия между ключевыми и позиционными аргументами, рассмотрим простую функцию:

```
def multiply_numbers(a, b):
    return a * b
```

Для типичной функции, такой как `multiply_numbers`, значения параметров могут задаваться как позиционными, так и ключевыми аргументами. На рис. 6.6 показаны некоторые способы вызова этой функции с двумя параметрами.

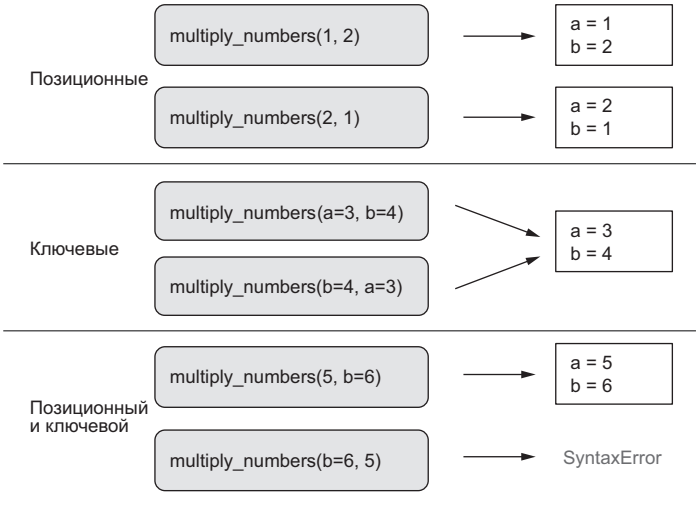


Рис. 6.6. Использование позиционных и ключевых аргументов при вызове функций. Если перед аргументом указывается идентификатор, то это ключевой аргумент. Если перед аргументом нет идентификатора, то такой аргумент называется позиционным

Из примеров, изображенных на рис. 6.6, можно сделать некоторые выводы, относящиеся к использованию позиционных и ключевых аргументов:

- При использовании позиционных аргументов их порядок имеет значение. Аргументы сопоставляются с исходными параметрами из заголовка функции.
- При использовании ключевых аргументов их порядок значения не имеет. Аргументы определяются передаваемыми идентификаторами.
- Если позиционные аргументы используются вместе с ключевыми, все позиционные аргументы должны предшествовать ключевым. В противном случае будет выдана ошибка `SyntaxError`.

Итак, вы знаете, чем позиционные аргументы отличаются от ключевых, и мы можем перейти к определению переменного количества позиционных и ключевых аргументов.

Только позиционные и только ключевые аргументы

На рис. 6.6 показано, что при вызове функции аргументы могут задаваться как позиционные или как ключевые. Таким образом, при вызове функции с аргументами Python использует определенный порядок для идентификации аргументов по определению функции. Если аргументы являются ключевыми, Python сопоставляет их с соответствующими параметрами в определении. Если аргументы являются позиционными, Python обрабатывает их в соответствии с их позициями. В общем случае способ задания аргументов (как позиционных, так и ключевых) может быть любым.

Существуют два более сложных варианта задания аргументов: *только позиционные* (positional-only) аргументы могут задаваться только позиционно, а *только ключевые* (key-only) аргументы могут задаваться лишь с идентификаторами. Напомню, что заголовок метода `sort` выглядит так: `sort(*, key=None, reverse=False)`. Знак `*` указывает на то, что следующие за ним аргументы могут задаваться только как ключевые.

Определяя аргументы как только ключевые, вы заставляете пользователей своего кода использовать ключевые аргументы, чтобы они точно знали, значения каких параметров они задают. Используйте эту возможность, если вы хотите, чтобы некоторые аргументы задавались только как ключевые.

Что касается только позиционных аргументов, взгляните на функцию `sum`: `sum(iterable, /, start=0)`. Знак `/` указывает, что предшествующие ему аргументы могут задаваться только как позиционные. Эта возможность полезна, но вам нечасто придется задавать в коде аргументы, которые могут использоваться только как позиционные.

6.4.2. Получение переменного количества позиционных аргументов

В функции `print` (листинг 6.12) обозначение `*objects` позволяет выводить произвольное количество объектов, что делает функцию более гибкой. В этом разделе вы научитесь определять функции, получающие переменное количество позиционных аргументов.

Для удобства обсуждения я начну с простой функции, целью которой является преобразование произвольного количества объектов в соответствующие строковые представления.

Вы не знаете заранее, сколько объектов будет передано при вызове функции. А значит, ваша функция должна быть гибкой — такой же, как функция `print`. Ее код выглядит так:

```
def stringify(*items):
    print(f"got {items} in {type(items)}")
    return [str(item) for item in items]
```

Использование *args как кортежа

В заголовке функции запись `*items` используется для обозначения того, что функция может получать переменное количество позиционных аргументов. Для этого перед именем аргумента ставится символ * (звездочка). Теперь вы знаете, что с таким заголовком пользователь может вызвать функцию с произвольным числом позиционных аргументов, и возникает следующий вопрос: как использовать эти позиционные аргументы в теле функции?

Так как мы включили строку кода для вывода аргументов `print(f"got {items} in {type(items)}")`, можно вызвать функцию `stringify` для проверки содержимого `items`:

```
>>> stringify(1, "two", None)
got (1, 'two', None) in <class 'tuple'>
['1', 'two', 'None']  ◄——— Возвращаемое значение функции выводится на консоль
```

Из вывода следует, что все позиционные аргументы упакованы в кортеж `tuple` с именем `items`. Таким образом, к `items` можно применить любые методы, относящиеся к кортежам. В данном примере для перебора объекта `items` используется списковое включение.

Размещение *args в последнем позиционном аргументе

Если вы ожидаете, что пользователь вызовет функцию, которая кроме `*args` получает другие позиционные аргументы, разместите `*args` в конце. Рассмотрим измененную версию функции `stringify`:

```
def stringify_a(item0, *items):
    print(item0, items)
```

При вызове `stringify_a` Python знает, как нужно разобрать позиционные аргументы. Первый аргумент направляется в `item0`, а остальные аргументы направляются в `items`:

```
>>> stringify_a(0)
0; ()
>>> stringify_a(0, 1)
0; (1,)
```

Очевидно, функция `stringify_a` работает. Теперь рассмотрим нерабочую модификацию:

```
def stringify_b(*items, item0):
    print(item0, items)
```

При вызове `stringify_b` с позиционными аргументами Python не может определить, какому параметру соответствует тот или иной аргумент. `items` обозначает любое количество позиционных аргументов, и Python не знает, где следует остановиться, как в следующем примере:

```
stringify_b(0, 1)
# ERROR: TypeError: stringify_b() missing 1 required keyword-only argument:
➥ 'item0'
```

При вызове `stringify_b` только с позиционными аргументами возникает ошибка `TypeError`, а в сообщении об ошибке говорится, что при вызове пропущен только ключевой аргумент `item0`. А значит, `stringify_b` можно будет использовать при передаче `items` как ключевого аргумента:

```
>>> stringify_b(0, item0=1)
1 (0,)
```

Хотя вызов функции работает, изначально мы намеревались определить функцию, которая может вызываться только с позиционными аргументами. При таком предположении следует помнить, что `*args` следует размещать в конце списка позиционных аргументов.

6.4.3. Получение переменного количества ключевых аргументов

В разделе 6.4.2 вы узнали, как написать функцию, получающую произвольное количество позиционных аргументов. Также можно определить функцию, которая получает любое количество ключевых аргументов. Для переменного количества ключевых аргументов используется обозначение `**kwargs`. В этом разделе вы узнаете о `**kwargs`.

Для удобства обсуждения мы начнем с простой функции, в которой применяется `**kwargs`. Затем, используя функцию в качестве примера, приведем несколько замечаний по поводу `**kwargs`:

```
def create_report(name, **grades):
    print(f"got {grades} in {type(grades)}")
    report_items = [f"***** Report Begin for {name} *****"]
    for subject, grade in grades.items():
        report_items.append(f"### {subject}: {grade}")
    report_items.append(f"***** Report End for {name} *****")
    print("\n".join(report_items))
```

Использование **kwargs в качестве словаря

Мы знаем, что переменное число позиционных аргументов упаковывается в объект `tuple`. Аналогичным образом переменное число ключевых аргументов

упаковывается в объект `dict`. Чтобы убедиться в этом, вызовем функцию `create_report`:

```
create_report("John", math=100, phys=98, bio=95)
# Выводимые строки:
got {'math': 100, 'phys': 98, 'bio': 95} in <class 'dict'>
***** Report Begin for John *****
### math: 100
### phys: 98
### bio: 95
***** Report End for John *****
```

Из вывода следует, что эти ключевые аргументы образуют объект `dict`. С объектом `dict` можно использовать методы, применимые к `dict`. В этом примере все пары «ключ — значение» перебираются с использованием `items`.

Размещение **kwargs в последнем параметре

При использовании `**kwargs` в функции следует помнить правило синтаксиса, согласно которому обозначение `**kwargs` должно размещаться после всех остальных параметров. Также стоит вспомнить, что позиционные аргументы должны размещаться до всех ключевых аргументов. На рис. 6.7 представлен общий порядок таких аргументов.

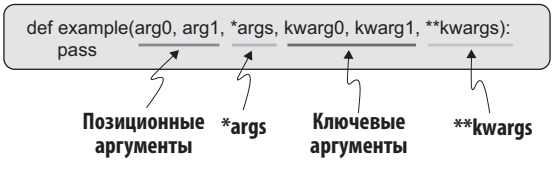


Рис. 6.7. Порядок размещения позиционных и ключевых аргументов в определении функции. В общем случае позиционные аргументы должны предшествовать ключевым аргументам. *args должен быть последним позиционным аргументом, а **kwargs — последним ключевым аргументом

6.4.4. Обсуждение

Несмотря на то что `*args` и `**kwargs` повышают гибкость определяемых функций, они менее явно передают пользователям информацию о применимых параметрах. А значит, злоупотреблять этой возможностью не стоит. Рассматривать идею использования `*args` и `**kwargs` следует, только если вы не знаете, сколько позиционных или ключевых аргументов будет получать функция. В общем случае в определении функции лучше использовать позиционные или ключевые аргументы с явно заданными именами, потому что имена аргументов должны выражать смысл соответствующих параметров.

6.4.5. Задача

Продолжим историю о Кори, преподающем программирование на Python в колледже. Студенты знают, что функция с `**kwargs` получает переменное количество ключевых аргументов, как в следующем примере:

```
def example(**kwargs):
    pass
```

Чтобы протестировать их знания о вызове функций, Кори создает список возможных способов вызова функции из этого примера:

```
example(a=1, b=2)
example(1, 2)
example(2a=1, 2b=2)
example()
```

Если бы вы были на месте студента, как бы вы определили, какой способ допустим, а какой нет? Из-за чего некоторые вызовы становятся недопустимыми?

> **ПОДСКАЗКА** Ключевые аргументы используют идентификаторы. В языке Python в отношении идентификаторов действуют специальные правила: например, они не могут начинаться с цифры.

6.5. КАК ПРАВИЛЬНО НАПИСАТЬ DOC-СТРОКУ ДЛЯ ФУНКЦИИ

Сталкиваясь с новой функцией, вы обычно обращаетесь к документации, чтобы понять, как использовать эту функцию. Например, при помощи встроенной функции `isinstance` можно проверить, принадлежит ли объект к конкретному типу. Но вы не знаете, как вызвать эту функцию. Можно ли где-то получить эту информацию (не считая поиска в интернете)? Да, можно — при помощи встроенной функции `help`, как показано в следующем листинге.

Листинг 6.13. Получение doc-строки с помощью функции help

```
>>> help(isinstance)
Help on built-in function isinstance in module builtins:

isinstance(obj, class_or_tuple, /)
Return whether an object is an instance of a class or
➥  of a subclass thereof.

A tuple, as in ``isinstance(x, (A, B, ...))``, may be given as the
➥  target to check against. This is equivalent to
➥  ``isinstance(x, A) or isinstance(x, B) or ...`` etc.
```

Как показано в листинге 6.13, функция `help` используется для получения doc-строк для функции `isinstance`. Существует другая, хотя и менее известная возможность: для получения doc-строки функции можно обратиться к ее специальному атрибуту `__doc__`:

```
>>> print(isinstance.__doc__)
Return whether an object is an instance of a class or
➥  of a subclass thereof.
A tuple, as in ``isinstance(x, (A, B, ...))``, may be given as the
➥  target to check against. This is equivalent to
➥  ``isinstance(x, A) or isinstance(x, B) or ...`` etc.
```

На всякий случай напомню, что Python использует doc-строки для обращения к документации функции, класса или модуля, поясняющей их функциональность. В нашем примере просматриваются doc-строки функции `isinstance`, содержащие инструкции по использованию `isinstance`. Важно, что doc-строку легко получить, просто выполнив `help` в консоли Python, и не придется полагаться ни на какие внешние источники информации. В этом разделе вы научитесь создавать doc-строки для функций.

ОСНОВНЫЕ ПОНЯТИЯ *Doc-строка* представляет собой строку, которая документирует модуль, класс, функцию или метод, объясняя, как правильно пользоваться ими.

6.5.1. Базовая структура doc-строки функции

Doc-строка функции охватывает несколько физических строк и размещается под заголовком функции. По общепринятым соглашениям строка заключается в тройные кавычки. Такие строки могут заключаться как в утроенные символы `'`, так и в утроенные символы `"`, важно лишь то, чтобы они были парными. В этом разделе рассматривается базовая структура doc-строк функций.

В такой многострочной строке должны присутствовать три элемента: краткое описание функции, параметры и возвращаемое значение. Если ваша функция может выдавать одно или несколько исключений, их следует перечислить в четвертом элементе. На рис. 6.8 изображена структура doc-строки функции.

Заметим, что Python-программисты не достигли консенсуса в отношении стиля doc-строк. Стиль doc-строки на рис. 6.8 называется *стилем Google*, потому что он официально рекомендован компанией Google. Разные пользователи Python и IDE используют разные стили. PyCharm — одна из самых распространенных IDE для Python — по умолчанию использует для doc-строк так называемый стиль reST (reStructured Text), пример которого представлен на рис. 6.9.

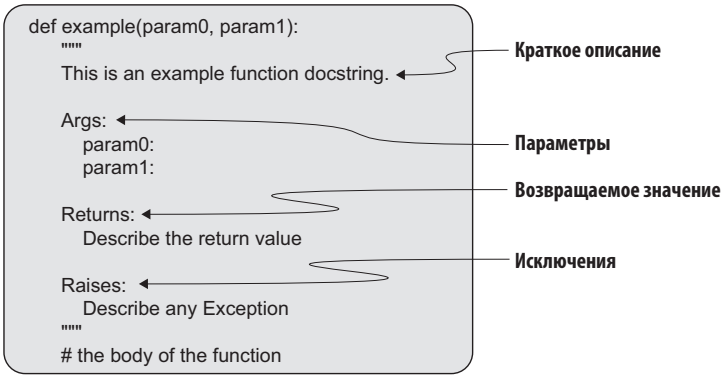


Рис. 6.8. Doc-строка функции в стиле Google. Обязательны три элемента: краткое описание, параметры и возвращаемое значение. Если функция выдает какие-либо исключения, они тоже должны быть перечислены в doc-строке

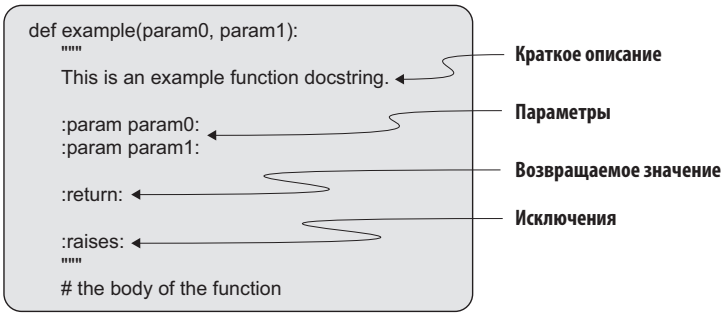


Рис. 6.9. Doc-строка функции в стиле reST, используемом в PyCharm. Ключевые элементы остаются такими же, как у doc-строк в других стилях: краткое описание, параметры, возвращаемое значение и исключения (если они есть)

Хотя Python-программисты обычно сходятся во мнениях о том, какие элементы должны включаться в doc-строку функции, каждый отдельный программист может сам выбрать стиль по своему вкусу или следовать соглашениям компании. В этом разделе мы будем придерживаться стиля reST. В следующих разделах будут рассмотрены правильные способы определения каждого элемента.

СОПРОВОЖДАЕМОСТЬ Важно придерживаться конкретного стиля doc-строк в проекте. Последовательность документирования чрезвычайно важна как для удобочитаемости кода, так и для простоты сопровождения.

6.5.2. Указание действия функции в кратком описании

Первым элементом doc-строки функции является краткое описание функции. Оно должно быть компактным и по возможности занимать только одну строку. Этот элемент предоставляет высокоуровневое описание действия, выполняемого функцией.

Например, в листинге 6.13 мы видим doc-строку для встроенной функции `isinstance`. В ее кратком описании четко обозначено действие, выполняемое функцией: возвращение информации о том, является ли объект экземпляром класса или какого-либо из его подклассов. Стоит придерживаться этого принципа и при создании ваших собственных кратких описаний. Для некоторых простых функций doc-строка может ограничиваться всего одной строкой текста. В таком случае краткое описание составляет всю doc-строку. Следующая простая функция представляет такую ситуацию:

```
def doubler(a):
    """Возвращает число, умноженное на 2"""
    return a * 2
```

6.5.3. Документирование параметров и возвращаемого значения

После краткого описания функции следующим шагом построения doc-строки функции становится документирование каждого параметра, используемого функцией. В стиле reST каждый параметр начинается с `:param`, а разные параметры перечисляются в разных строках. Для каждого параметра должна быть указана следующая информация:

- *Имя параметра* — должно точно совпадать с именем, указанным в заголовке функции.
- *Тип параметра* — какой тип данных вы ожидаете получить в параметре? Укажите его.
- *Описание* — в зависимости от того, насколько интуитивно понятен смысл параметра, предоставьте краткое или более содержательное описание, которое поможет пользователям понять, какую информацию содержит параметр или зачем он нужен (если его назначение неочевидно).
- *Значение по умолчанию* (необязательно) — если параметр имеет значение по умолчанию, укажите его. Если не очевидно, почему конкретное значение было выбрано по умолчанию, необходимо привести краткое обоснование.

В листинге 6.14 эти рекомендации продемонстрированы в действии.

Пример в листинге 6.14 предоставляет необходимую doc-строку для трех параметров, включая имена параметров, их типы и краткие пояснения. Кроме того, поскольку `taking_int` имеет значение по умолчанию, оно тоже упоминается

в doc-строке. Если doc-строка одного параметра занимает несколько физических строк, не забудьте включить отступ во вторую и последующие строки, чтобы разделение параметров было более явным.

Листинг 6.14. Пример doc-строки для простой функции

```
def quotient(dividend, divisor, taking_int=False):
    """
    Вычисляет частное от деления двух чисел.

    :param dividend: int | float, делимое
    :param divisor: int | float, делитель
    :param taking_int: bool, признак получения целой части
    ➥  частного; default: False, вычисляется точный результат деления
    ➥  двух чисел

    :return: float | int, частное от деления
    """
    result = dividend / divisor
    if taking_int:
        result = int(result)
    return result
```

С позиций удобочитаемости мы используем содержательные имена для самой функции (`quotient`) и всех параметров (`dividend`, `divisor` и `taking_int`). Использование содержательных имен играет ключевую роль в определении функции, потому что эти имена могут предоставить интуитивно понятную информацию о функции. При хорошем выборе имен пользователям, вероятно, не придется обращаться к doc-строке, чтобы понять смысл функции.

> **УДОБОЧИТАЕМОСТЬ** Все имена должны быть содержательными, чтобы обеспечить максимальную удобочитаемость кода. В выборе длинных имен нет ничего страшного, потому что функция автозаполнения присутствует во всех основных IDE. После ввода первой пары букв вы сможете выбрать нужное имя.

Другими словами, вашей целью должно быть определение функции, которое будет понятным и удобным для пользователей, а это сведет к минимуму вероятность того, что пользователю придется обращаться к doc-строкам функций. Следует помнить, что doc-строка должна быть резервным источником информации для ваших функций.

Что касается возвращаемого значения функции, doc-строка использует `:return` для обозначения типа и краткого описания возвращаемого значения. Описание должно быть лаконичным и понятным.

6.5.4. Определение возможных исключений

Если ваша функция может выдавать исключения, следует перечислить их в doc-строке, чтобы пользователь знал, каких исключений можно ожидать, и смог избежать или обработать их.

Возьмем функцию `quotient`, которая включает операцию деления `dividend / divisor`. Мы знаем, что результат деления на 0 не определен, и можем легко увидеть, что произойдет при попытке делить на 0:

```
1 / 0
# ERROR: ZeroDivisionError: division by zero
```

Таким образом, это исключение можно указать в doc-строке, как показано в листинге 6.15.

Листинг 6.15. Определение возможных исключений в doc-строке

```
def quotient(dividend, divisor, taking_int=False):
    """
    Вычисляет частное от деления двух чисел.

    :param dividend: int | float, делимое
    :param divisor: int | float, делитель
    :param taking_int: bool, признак получения целой части
    ➥  частного; default: False, вычисляется точный результат деления
    ➥  двух чисел

    :return: float | int, частное от деления
    :raises: ZeroDivisionError, если divisor равен 0
    """
    if divisor == 0:
        raise ZeroDivisionError("division by zero")
    result = dividend / divisor
    if taking_int:
        result = int(result)
    return result
```

Явно выдает исключение ZeroDivisionError

В листинге 6.15 мы явно проверяем, равен ли 0 делитель `divisor`, и если равен, выдаем исключение `ZeroDivisionError`. Заметим, что даже если исключение не выдается явно, оно может выдаваться при вызове вида `quotient(1, 0)`, потому что Python выдает `ZeroDivisionError` там, где это уместно. Здесь это исключение выдается явно, потому что я хочу показать, как исключение, выдаваемое функцией, должно документироваться в doc-строке.

Попутно заметим, что при создании собственных Python-модулей часто приходится определять собственные исключения и явно выдавать их в создаваемых вами функциях. Нестандартные исключения рассматриваются в разделе 12.5.

6.5.5. Обсуждение

Существуют разные стили для создания doc-строк функций. Главное — неизменно придерживаться выбранного типа. Если вы работаете в команде, используйте стиль, согласованный с вашей командой. Если вы пишете функции/модули только для себя, используйте стиль, к которому вы привыкли. Помните, что

в долгосрочной перспективе единство стиля программирования играет ключевую роль для простоты сопровождения любого проекта.

6.5.6. Задача

Джерри привык использовать стиль reST в своих doc-строках, как показано в листинге 6.15. Он перешел в компанию, использующую во всей документации стиль Google. Как будет выглядеть doc-строка, если он перепишет ее в листинге 6.15 с использованием стиля Google?

ПОДСКАЗКА На рис. 6.8 приведена doc-строка, использующая стиль Google.

ИТОГИ

- Рассмотрите возможность определения значений по умолчанию для аргументов, значения которых остаются неизменными для большинства вызовов. Пользователям уже не придется задавать их, используя значения по умолчанию; сокращение числа аргументов упростит чтение вызовов функций.
- Задавая значения по умолчанию для изменяемых аргументов (таких, как `list`), не используйте конструктор `list()`, потому что функция вычисляется в месте определения, включая аргументы по умолчанию. Применение конструктора приведет к тому, что разные вызовы функции будут использовать один и тот же изменяемый объект, а это может вызвать нежелательные побочные эффекты. Чтобы избежать этой проблемы, используйте `None` в качестве значения по умолчанию для изменяемых аргументов.
- У каждой функции Python есть возвращаемое значение — либо явное, либо неявно возвращаемое значение `None`.
- Функция может возвращать несколько значений, образующих один объект кортежа. Вы можете применить механизм распаковки кортежа для получения его отдельных компонентов после вызова функции. Так читателю кода будет проще понять, как вы собираетесь использовать возвращаемое значение.
- Хотя Python является языком с динамической типизацией, вы можете использовать аннотации типов для предоставления полезной информации о типах аргументов и возвращаемого значения функции. При включении аннотаций типов в определение функции вы делаете свою функцию более удобочитаемой, так что пользователям будет проще в ней разобраться. Важно, что современные IDE могут использовать аннотации типов функции и выводить предупреждения в реальном времени, если в качестве аргумента используется объект несовместимого типа.
- При вызове функции часто передаются необходимые аргументы. Если при передаче аргумента указывается идентификатор, такой аргумент называется

ключевым. В свою очередь, аргументы, которые не имеют идентификаторов и интерпретируются на основании своей позиции, называются позиционными. Позиционные аргументы всегда должны предшествовать ключевым аргументам.

- В большинстве случаев желательно использовать фиксированное количество позиционных и ключевых аргументов. Однако в некоторых ситуациях лучше определять функции, получающие переменное число позиционных и/или ключевых аргументов, которые обозначаются `*args` и `**kwargs` соответственно.
- Если ваши функции предназначены для использования другими людьми, необходимо предоставить для них документацию — так называемые doc-строки. Doc-строка функции должна включать краткое описание функции, все параметры, возвращаемое значение и возможные исключения (если они есть).
- Разработчики используют разные стили записи doc-строк. Создавая doc-строки для своих функций, обязательно выберите конкретный стиль doc-строк и последовательно применяйте его в своем коде. Последовательное применение стиля doc-строк упростит разработку и сопровождение кода (достаточно хорошо владеть всего одним стилем); кроме того, другим разработчикам будет проще читать ваш код.

7 Продвинутое использование функций

В этой главе

- ✓ Использование лямбда-функций для небольших операций
- ✓ Работа с функциями высшего порядка
- ✓ Создание и использование декораторов
- ✓ Использование генераторов для получения данных
- ✓ Создание частичных функций

Вероятно, вы уже поняли, что в любом проекте бóльшая часть проводимого за разработкой времени тратится на написание функций. В главе 6 мы сосредоточились на основах написания и использования функций. После рассмотрения этих вопросов вы сможете писать функции для решения своих задач, которые будут удобными для пользователей. Создатели Python понимали важнейшую роль функций в любом проекте, поэтому в языке присутствуют расширенные возможности, которые вы сможете применить, чтобы ваши функции лучше справлялись со своими задачами.

В этой главе вы узнаете о некоторых продвинутых возможностях функций. Может показаться, что некоторые концепции требуют углубленных знаний, однако применять эти практические приемы в повседневной работе оказывается не так уж сложно.

7.1. КАК ОПРЕДЕЛЯЮТСЯ ЛЯМБДА-ФУНКЦИИ

Определяя функцию, вы вводите ключевое слово `def`, а затем указываете имя функции, которое становится ее идентификатором. Мы будем называть такие функции *именованными* (хотя этот термин не является общепринятым).

В Python также можно определить и другую разновидность функций, которым не присваиваются имена. Такие функции называются *анонимными*. В формальной терминологии эти функции известны как *лямбда-функции*. При обсуждении расширенной сортировки с пользовательскими функциями (раздел 3.2) мы использовали пример, в котором лямбда-функция передавалась в параметре `key` метода `sort`:

```
tasks.sort(key=lambda x: x['urgency'], reverse=True)
```

В этом разделе вы узнаете все, что необходимо знать для использования лямбда-функций: их компоненты и основные приемы работы с ними.

> **ОБРАТИТЕ ВНИМАНИЕ** Термин «лямбда-функции» или «лямбда-выражения», который мы используем для анонимных функций, существует не только в Python, но и во многих других языках, в частности в Java. Название происходит от лямбда-исчисления в математике.

7.1.1. Создание лямбда-функций

Вы уже видели примеры лямбда-выражений, но еще не занимались их формальным изучением. Начнем с ключевых элементов, образующих лямбда-функцию.

При создании лямбда-функций не нужно использовать ключевое слово `def` и указывать идентификатор, как для обычных функций. Вместо этого используется ключевое слово `lambda`, которое сообщает об использовании лямбда-функции. На рис. 7.1 представлены компоненты лямбда-функции.

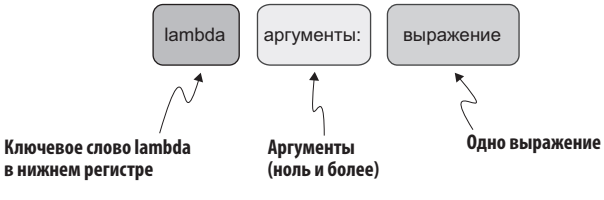


Рис. 7.1. Создание лямбда-функции, состоящей из трех компонентов: ключевого слова lambda, аргументов и выражения

Как показано на рис. 7.1, за ключевым словом `lambda` следуют аргументы и одно выражение, которое использует аргументы для получения значения. Не забудьте, что после аргументов должно следовать двоеточие. Лямбда-функция может иметь

ноль и более аргументов. Даже если лямбда-функция не получает аргументов, двоеточие перед выражением все равно обязательно.

> **ОСНОВНЫЕ ПОНЯТИЯ** *Ключевые слова* — специальные слова, зарезервированные Python для выполнения предварительно определенных операций: `def` для создания функции, `class` для создания класса, `lambda` для создания лямбда-функции и т. д.

В отличие от обычных функций, которые возвращают объект, лямбда-функции не возвращают ничего. При попытке это сделать выдается синтаксическая ошибка:

```
lambda  x: return x * 2
# ERROR: SyntaxError: invalid syntax
```

Ошибка `SyntaxError` ожидаема, потому что лямбда-функции используют выражения вместо инструкций, а `return x * 2` является инструкцией.

> **НАПОМИНАНИЕ** Результатом вычисления выражения является одно значение или объект, тогда как инструкция выполняет некоторое действие и не вычисляется с получением результата.

Итак, теперь вы знаете, как создавать лямбда-функции. Попробуем сделать это:

```
doubler = lambda x: x * 2
```

Эта лямбда-функция умножает число на 2. Для демонстрационных целей лямбда-функция присваивается переменной `doubler`, чтобы эту функцию можно было проанализировать более подробно. Но как будет показано в следующем разделе, присваивать лямбда-функции переменным не считается хорошей практикой. Проверка типа лямбда-функции показывает, что это разновидность функции:

```
print(type(doubler))
# Вывод: <class 'function'>
```

Лямбда-функции по своей сути и являются функциями, поэтому они вызываются как обычные функции. При вызове лямбда-функции указываются необходимые аргументы, как и при вызове обычной функции:

```
>>> doubler(5)
10
>>> doubler(8)
16
```

7.1.2. Использование лямбда-функций для выполнения небольших операций

В разделе 7.1.1 я упоминал, что лямбда-функции не стоит присваивать выражениям. Главная причина заключается в том, что лямбда-функция должна

выполнять небольшую операцию и она используется лишь один раз. В этом разделе мы разберемся, что имеется в виду под «небольшой операцией».

В какой ситуации может понадобиться небольшая операция? Вспомните, как вы учились выполнять сложную сортировку с пользовательской функцией (листинг 3.3) в разделе 3.2.1. Для удобства код снова приводится в листинге 7.1.

Листинг 7.1. Сортировка списка с применением пользовательской функции

```
tasks = [
    {'title': 'Laundry', 'desc': 'Wash clothes', 'urgency': 3},
    {'title': 'Homework', 'desc': 'Physics + Math', 'urgency': 5},
    {'title': 'Museum', 'desc': 'Egyptian things', 'urgency': 2}
]

def using_urgency_level(task):
    return task['urgency']

tasks.sort(key=using_urgency_level, reverse=True)
```

Мы определяем функцию `using_urgency_level` и присваиваем ее аргументу `key` в вызове метода `sort`. Заметим, что эта функция `using_urgency_level` выполняет небольшую операцию по получению значения объекта `dict`. Кроме того, эта функция используется только один раз как аргумент `key` метода `sort`. Используя одноразовую лямбда-функцию в аргументе `key` при вызове `sort`, вы избегаете генерирования лишнего «шума» (явного определения функции), что делает код более чистым. Таким образом, пример становится идеальным сценарием использования лямбда-функции:

```
tasks.sort(key=lambda x: x['urgency'], reverse=True)
```

Эта лямбда-функция получает один параметр, представляющий каждый объект `dict` в объекте `list` как в функции `using_urgency_level`.

> **НАПОМИНАНИЕ** Вызов как обычной функции, так и лямбда-функции является выражением, которое получает входные данные и генерирует результат.

7.1.3. Потенциальные проблемы при использовании лямбда-функций

После первого знакомства с лямбда-функциями возникает впечатление, что это классная возможность для настоящих профессионалов. Название — лямбда! — звучит круто. Лямбда-функция компактна — всего одна строка кода. Многие начинающие Python-программисты недостаточно хорошо знают лямбда-функции, но при этом полагают, что если они начнут пользоваться этой нетривиальной возможностью, то их будут считать профессионалами. Если у вас тоже возникли подобные мысли, есть вероятность, что вы натолкнетесь на один из следующих подводных камней.

Присваивание лямбда-функции переменной

Я уже не единожды упоминал о том, что лямбда-функции не должны присваиваться переменным. Причина в том, что лямбда-функция используется только один раз, как сказано в предыдущем разделе. Однако с точки зрения удобочитаемости присваивание лямбда-функции переменной может показаться удачной идеей — ведь вы можете присвоить переменной содержательное имя и тем самым сообщить читателям кода больше информации о лямбда-функции. Рассмотрим следующий пример:

```
using_urgency_level = lambda x: x['urgency']

tasks.sort(key=using_urgency_level, reverse=True)
```

В этом примере имя `using_urgency_level` используется для обозначения лямбда-функции, и оно дает информацию об алгоритме сортировки. Однако более важная причина избегать присваивания лямбда-функций переменным заключается в том, что при возникновении ошибки в функции это усложнит отладку программы, как показано в листинге 7.2.

Листинг 7.2. KeyError с лямбда-функцией

```
using_urgency_level0 = lambda x: x['urgency0']
tasks.sort(key=using_urgency_level0, reverse=True)
# ERROR:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
KeyError: 'urgency0'
```

Для сравнения применим ту же ошибку (использование ошибочного ключа для обращения к значению) к именованной функции. Следующий листинг показывает, что при этом происходит.

Листинг 7.3. KeyError с именованной функцией

```
def using_urgency_level1(task):
    return task['urgency1']
tasks.sort(key=using_urgency_level1, reverse=True)
# ERROR:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in using_urgency_level1
KeyError: 'urgency1'
```

В листингах 7.2 и 7.3 я выделил самое значимое различие, хотя в обоих фрагментах кода упоминается одна и та же ошибка `KeyError`. При использовании именованной функции сообщение об ошибке ясно показывает, где возникла проблема: в функции `using_urgency_level1`. С другой стороны, при использовании лямбда-функции с неправильным ключом сообщение об ошибке указывает лишь

на то, что какие-то проблемы возникли с функцией `<lambda>`. Такое сообщение об ошибке ничего не говорит о том, где следует исправлять проблему, особенно если лямбда-функция определяется в другом месте.

СОПРОВОЖДАЕМОСТЬ Не присваивайте лямбда-функции переменным, это усложнит отладку кода, если что-то пойдет не так.

Более эффективные альтернативы

Итак, лямбда-функции предназначены для выполнения небольших операций. Типичный практический сценарий — передача лямбда-функций в параметре `key` функций (таких, как `sort`, `sorted` и `max`). Однако в некоторых случаях существуют более эффективные альтернативы.

Допустим, у вас имеется список чисел и вы хотите создать новый объект `list` из трех чисел, упорядоченных по абсолютным значениям. Возможно, у вас получится решение следующего вида:

```
integers = [-4, 3, 7, 0, -6]

sorted(integers, key=lambda x: abs(x))
# Вывод: [0, 3, -4, -6, 7]
```

В лямбда-функциях используется встроенная функция `abs`, которая вычисляет абсолютное значение элемента. Иное, более питоническое решение применяет функцию `abs` непосредственно в параметре `key`:

```
sorted(integers, key=abs)
```

Другой пример: допустим, имеется список кортежей, в котором каждый кортеж содержит оценки студента по трем дисциплинам, и вы хотите узнать, какой объект `tuple` имеет наивысшую суммарную оценку. Рассмотрим следующее решение:

```
scores = [(93, 95, 94), (92, 95, 96), (94, 97, 91), (95, 97, 99)]
max(scores, key=lambda x: x[0] + x[1] + x[2])
# Вывод: (95, 97, 99)
```

В этой лямбда-функции мы при помощи индексирования получаем каждую из трех оценок и прибавляем ее к сумме. Но вы уже знаете, что встроенная функция `sum` может получить любой итерируемый объект для вычисления суммы своих элементов. Следовательно, выгоднее использовать функцию `sum` напрямую. Попутно заметим, что для получения того же результата применяется вызов `max(scores)`. Здесь для выбора максимального элемента я явно включаю значение `key=sum`:

```
max(scores, key=sum)
```

УДОБОЧИТАЕМОСТЬ По возможности используйте вместо лямбда-функций встроенные функции или другие альтернативы. Такие решения обычно получаются более компактными.

7.1.4. Обсуждение

Лямбда-функции должны выполнять небольшие операции, рассчитанные на одноразовое использование, например передачу аргумента `key` для встроенных функций (таких, как `sorted`, `max` и `min`). В частности, лямбда-функции широко используются в сторонних библиотеках, например pandas, популярной библиотеке data science. Так, в pandas можно использовать функцию `apply` для создания новых данных из существующего объекта `DataFrame`. Функция `apply` получает параметр `key`, который определяет, как новые данные должны создаваться на базе существующих данных. Таким образом, лямбда-функции становятся универсальным методом, который может использоваться для определения небольших заданий в контексте извлечения или преобразования данных.

7.1.5. Задача

Студентка Линда изучает Python, чтобы выполнять пакетную обработку снимков и видеороликов. Она знает, что у функций Python есть специальный атрибут с именем `__name__`. Линда пытается обратиться к этому атрибуту с помощью лямбда-функции и именованной функции. Как вы думаете, какие значения она при этом получит?

ПОДСКАЗКА Вернитесь к листингам 7.2 и 7.3 и посмотрите, что говорится в сообщении об ошибке про именованную функцию и лямбда-функцию.

7.2. К КАКИМ ПОСЛЕДСТВИЯМ ПРИВОДИТ ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ КАК ОБЪЕКТОВ

Мы знаем, что Python является языком объектно-ориентированного программирования (ООП). В целом, говоря об объектах, мы обычно имеем в виду объект как сущность, представляющую некоторые данные. В первых пяти главах наше внимание было сосредоточено на темах, относящихся к моделям данных, — `str`, `list`, `tuple`, `dict` и `set`.

Эти классы и соответствующие им экземпляры являются примерами объектов. Важнейшая особенность работы с объектами заключается в том, что представляемыми данными можно оперировать, передавая их функциям. Следующий фрагмент демонстрирует возможность использования экземпляров `int` и `str` в функциях:

```
def add_three(number):
    return number + 3

add_three(7)  <-------  Использует объект int в функции

def greeting_message(person):
    return f"Hello, {person}!"

greeting_message("Zoe")  <-------  Использует объект str в функции
```

Заметим, что в этом разделе упоминалась возможность передачи именованной или лямбда-функции методу `sort`:

```
tasks.sort(key=lambda x: x['urgency'], reverse=True)
```

Возможность передачи функции в аргументе наводит на мысль о том, что лямбда-функции — или функции вообще — представляют некоторые данные по аналогии с другими моделями данных вроде `int` или `str`. Если пойти еще дальше, можно задаться вопросом, не являются ли объектами сами функции. В самом деле, говорят, что *в Python нет ничего, кроме объектов*: функции Python тоже рассматриваются как объекты. В этом разделе демонстрируются самые важные последствия того, что функции являются объектами, а также приводятся некоторые практические примеры.

7.2.1. Хранение функций в контейнерах данных

Мы знаем, что базовые модели данных можно комбинировать друг с другом, и такие комбинации открывают невероятные возможности. В частности, контейнеры данных могут использоваться для хранения практически любых видов моделей данных. Ничто не мешает вам создать список объектов `int`, `str`, `dict` или `set`. В `dict` могут храниться значения типов `int`, `str`, `list` и `dict`. В этом разделе вы узнаете о первом следствии работы с функциями как с объектами, а именно об использовании функций с другими моделями данных. В частности, вы увидите, какие возможности открывает хранение функций в контейнерах данных.

Допустим, у вас имеется программный интерфейс (API), при помощи которого пользователь может отправить список чисел и указать необходимое действие для данных. Для простоты допустим, что действием может быть вычисление среднего, минимума или максимума. Функция API выглядит примерно так:

```
def get_mean(data):
    return "mean of the data"

def get_min(data):
    return "min of the data"

def get_max(data):
    return "max of the data"
```

```
def process_data(data, action):
    if action == "mean":
        processed = get_mean(data)
    elif action == "min":
        processed = get_min(data)
    elif action == "max":
        processed = get_max(data)
    else:
        processed = "error in action"

    return processed
```

В этом фрагменте кода `get_mean`, `get_min` и `get_max` представляют функции, выполняющие соответствующие вычисления. Как нетрудно заметить, тело `process_data` получается довольно громоздким. Если вместо этого сохранить функции в виде значений в объекте `dict`, решение получится более эффективным, как показано в листинге 7.4.

Листинг 7.4. Сохранение функций в объекте dict

```
actions = {"mean": get_mean, "min": get_min, "max": get_max}

def fallback_action(data):
    return "error in action"

def process_data(data, action):
    calculation = actions.get(action, fallback_action)
    processed = calculation(data)
    return processed
```

Резервная функция вызывается в том случае, если не используется ни одно из определенных действий

В листинге 7.4 используется словарь `actions`, в котором хранятся все необходимые действия. Когда пользователь указывает действие, необходимая функция определяется поиском по словарю. Тем самым снимается необходимость в конструкциях `if...elif...else...` с многочисленными разветвлениями. При большем количестве действий вы сможете существенно улучшить удобочитаемость за счет хранения функций в объекте `dict`.

> **УДОБОЧИТАЕМОСТЬ** Сложные структуры в инструкциях `if...elif...else...` усложняют чтение кода. Старайтесь выбирать другие альтернативы там, где это возможно.

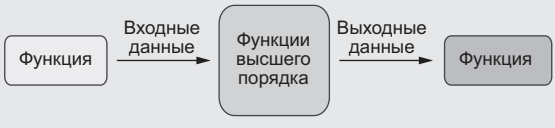
7.2.2. Передача функций в аргументах функций высшего порядка

Второе следствие использования функций как объектов заключается в том, что мы можем применять их как аргументы при вызове других функций. Если функции могут получать другие функции во входных данных (аргументы) или возвращать функции как результаты, мы будем называть их функциями высшего порядка — функциями, работающими на базе других функций.

В этом разделе мы сосредоточимся на достойной внимания функции высшего порядка `map` для демонстрации передачи функций как данных (аргументов) для других функций.

Функции высшего порядка

Функции высшего порядка получают функции в аргументах или возвращают функции как выходные значения, как показано на следующей схеме. Если функция получает одну или несколько функций в параметрах, она называется функцией высшего порядка; если функция возвращает функцию на выходе, она также называется функцией высшего порядка. Если функция делает и то и другое, она, безусловно, считается функцией высшего порядка.



Функции высшего порядка используют функции как аргументы и/или выходное значение

Функции первого порядка являются противоположностью функциям высшего порядка. Заметим, что концепция функций высшего порядка присутствует во многих современных языках, включая JavaScript, Kotlin и Swift. Знание этой концепции принесет пользу даже в том случае, если вам доведется работать на других языках.

В разделе 5.1 я кратко упомянул функцию `map` с сигнатурой вызова `map(func, iterable)`, в которой `func` — объект функции, часто называемый *функцией отображения*. Функция `map` создает итератор `map`, и я уже показывал, как сконструировать объект `list` на базе итератора `map`:

```
numbers_str = ["1.23", "4.56", "7.89"]

numbers = list(map(float, numbers_str))

assert numbers == [1.23, 4.56, 7.89]
```

ОБРАТИТЕ ВНИМАНИЕ Функция `map` может получать более одного итерируемого объекта. При передаче нескольких итерируемых объектов элементы каждого такого объекта передаются функции отображения в зависимости от порядка их перечисления. Как правило, функция `map` использует только один итерируемый объект, так что использование нескольких таких объектов в `map` может вызвать затруднения у неопытных программистов. Пользуйтесь этой возможностью осмотрительно.

С точки зрения написания питонического кода лучше не пытаться создать объект `list` на базе итератора `map`, а вместо этого воспользоваться механизмом спискового включения:

```
numbers_list = [float(x) for x in numbers_str]
```

Однако использование функций высшего порядка относится к стилю функционального программирования, в отличие от свойственного языку Python стиля ООП. В функциональном программировании внимание разработчика сосредоточено на применении и композиции функций, тогда как в ООП мы направляем все усилия на работу с разнообразными объектами. Со списковым включением и выражением-генератором (раздел 7.4) большинство случаев использования, связанных с `map`, можно заменить этими двумя механизмами, которые считаются более питоническими. Так как объект `map` может быть итератором, ничто не мешает использовать его в цикле `for`, если задействованные операции сложны:

```
for number in map(float, numbers_str):
    # операция 1
    # операция 2
    # операция 3
    # операция 4
    ...
```

В этом примере цикл `for` включает ряд операций, которые не подходят для спискового включения. В таком случае следует воспользоваться итератором `map`, который выдает свои элементы по одному, избавляя вас от необходимости создания объекта `list`.

7.2.3. Функции как возвращаемые значения

В предыдущем разделе мы сосредоточились на использовании функций как объектов при передаче их в параметрах таких функций высшего порядка, как `map`. В этом разделе мы займемся третьим аспектом использования функций как объектов. Я покажу, как создать функцию высшего порядка, которая возвращает функцию.

Ключевое слово `def` означает, что вы создаете функцию. Не все знают, что определение функции можно вложить в определение другой функции по следующей общей схеме:

```
def outside(x):
    def inside(y):
        pass
    pass
```

НАПОМИНАНИЕ Инструкция `pass` используется для выполнения синтаксических требований там, где должны присутствовать инструкции.

Допустим, вы хотите создать функцию высшего порядка. С этой новой функцией можно генерировать функции, которые прибавляют заранее определенное число. Применяя предшествующий синтаксис, мы приходим к решению, представленному в листинге 7.5.

Листинг 7.5. Создание функции, возвращающей функцию

```
def increment_maker(number):
    def increment(num0):
        return num0 + number

    return increment
```

УДОБОЧИТАЕМОСТЬ Добавьте пустую строку между внутренней функцией и инструкцией `return` внешней функции, чтобы код лучше читался. Как правило, пробелы и пустые строки служат естественными разделителями между разными логическими компонентами.

Как видно из листинга 7.5, внешняя функция получает параметр `number`. Внутри функции `increment_maker` определяется внутренняя функция `increment`, которая получает другое число (параметр `num0`). В отличие от функций первого порядка, которые возвращают `None` или некоторые данные, функция высшего порядка `increment_maker` возвращает как результат своей работы функцию `increment`. Теперь вы видите, насколько полезна эта функция высшего порядка: она позволяет создать серию функций увеличения значений, как показано в листинге 7.6.

Листинг 7.6. Создание функций с помощью вызова функции высшего порядка

```
increment_one = increment_maker(1)
increment_three = increment_maker(3)
increment_five = increment_maker(5)
increment_ten = increment_maker(10)

increment_one(99), increment_three(88), increment_five(80),
➥ increment_ten(100)
# Вывод: (100, 91, 85, 110)
```

Как показано в листинге 7.6, вы без труда можете создать несколько функций, указывая желаемую величину приращения. При вызове таких функций вы получите ожидаемые результаты.

7.2.4. Обсуждение

Python является ООП-языком, а возможность работы с функциями как с обычными объектами обеспечивает его дополнительную гибкость. Кто-то скажет, что примеры в листингах 7.5 и 7.6 слишком тривиальны, и я полностью согласен. Я использовал простые примеры для наглядного подтверждения самой

концепции. В разделе 7.3 будут рассматриваться декораторы — практический метод, основанный на создании функций высшего порядка.

7.2.5. Задача

В листинге 7.4 функции сохранялись в объекте `dict`. Понимаете ли вы, для чего в дополнение к этим функциям была создана функция `fallback_action`? И почему мы используем метод `get` вместо индексной записи?

> **ПОДСКАЗКА** Невозможно предсказать, как пользователи вызовут определенную вами функцию. Как обработать возможный вызов вида `process_data([1, 2, 3], "maxx")`?

7.3. КАК ПРОВЕРИТЬ БЫСТРОДЕЙСТВИЕ ФУНКЦИЙ С ДЕКОРАТОРАМИ

Функции являются неотъемлемыми компонентами любого приложения. Быстродействие вашего приложения, прежде всего скорость его реакции на действия пользователя, в значительной мере зависит от того, насколько быстро функции справляются с обработкой данных. Из-за этого в процессе разработки часто возникает необходимость в измерении скорости работы функций. Наивное решение могло бы выглядеть так:

Листинг 7.7. Измерение быстродействия функции

```
import random
import time

def example_func0():
print("--- example_func0 starts")
start_t = time.time()
    random_delay = random.randint(1, 5) * 0.1
    time.sleep(random_delay)
end_t = time.time()
print(f"*** example_func0 ends; used time: {end_t - start_t:.2f} s")

def example_func1():
print("--- example_func1 starts")
start_t = time.time()
    random_delay = random.randint(6, 10) * 0.1
    time.sleep(random_delay)
end_t = time.time()
    print(f"*** example_func1 ends; used time: {end_t - start_t:.2f} s")
```

Внедряет случайную задержку (0.1–0.5 секунды) для моделирования реальных операций

Внедряет случайную задержку (0.6–1 секунду) для моделирования реальных операций

В листинге 7.7 мы вычисляем разность во времени между моментами запуска функции и ее завершения, чтобы узнать, сколько времени заняло выполнение. Остается вызвать функцию и проверить ее быстродействие.

```
example_func0()
# Выводимые строки:
--- example_func0 starts
*** example_func0 ends; used time: 0.20 s

example_func1()
# Выводимые строки:
--- example_func1 starts
*** example_func1 ends; used time: 0.70 s
```

> **УДОБОЧИТАЕМОСТЬ** Если вы предполагаете, что будет выводиться множество строк, содержащих похожий текст, стоит выбрать специальный префикс и использовать его при выводе. Префиксы служат наглядными визуальными ориентирами.

Вряд ли вам придется наблюдать в приложении только за одной-двумя функциями. Скорее придется отслеживать быстродействие десятков или сотен функций. Было бы слишком утомительно добавлять в листинг 7.7 соответствующие строки (выделенные жирным шрифтом) для всех этих функций. Вспомним принцип DRY («не повторяйтесь»): если в программе есть существенные повторения, это верный признак того, что необходимо провести рефакторинг кода. В этом разделе я покажу, как использовать декораторы для решения подобных задач — применения общего действия к разным функциям.

7.3.1. Декорирование функции для вывода ее быстродействия

Я уже несколько раз упоминал о декораторах, но не объяснял, что означает этот термин. *Декораторы* — функции, предоставляющие дополнительную функциональность декорируемым функциям. Важно заметить, что декораторы не изменяют то, как работают декорируемые функции. В этом разделе мы построим декоратор для отслеживания быстродействия функции.

Прежде чем объяснять, как все работает, приведу пример кода. Просмотрите функцию `logging_time` и начните чтение кода со строки `@logging_time` в листинге 7.8.

Листинг 7.8. Использование декоратора для отслеживания быстродействия функций

```
import random
import time

def logging_time(func):
    def logger(*args, **kwargs):
        print(f"--- {func.__name__} starts")
```

```
        start_t = time.time()
        value_returned = func(*args, **kwargs)
        end_t = time.time()
        print(f"*** {func.__name__} ends; used time: {end_t - start_t:.2f} s")
        return value_returned

    return logger

@logging_time
def example_func2():
    random_delay = random.randint(3, 5) * 0.1
    time.sleep(random_delay)

example_func2()
# Выводятся следующие две строки:
--- example_func2 starts
*** example_func2 ends; used time: 0.40 s
```

Как видно из листинга 7.8, при вызове функции `example_func2` вы получаете вывод, в котором указывается ее быстродействие. Однако в теле `example_func2` нет кода, который бы это делал. Как же `example_func2` выводит информацию о скорости выполнения?

Волшебство кроется в строке `@logging_time` непосредственно перед заголовком `example_func2`. Этот специальный синтаксис предназначен для декорирования; он означает, что определенная ниже функция будет декорироваться функцией-декоратором `logging_time`. Функцию-декоратор можно применить к любым функциям на ваше усмотрение, как в следующем примере:

```
@logging_time
def example_func3():
    pass

@logging_time
def example_func4():
    pass

@logging_time
def example_func5():
    pass
```

СОПРОВОЖДАЕМОСТЬ Декораторы выделяют общую вспомогательную функциональность, которая может использоваться многими функциями. Вам необходимо сопровождать только функции-декораторы, а не все отдельные декорируемые функции.

Вы уже видели, что функция-декоратор может применяться к разным функциям для выполнения общей функциональности. Но при этом мы еще не обсудили, как же устроен декоратор. Этой теме посвящен следующий раздел.

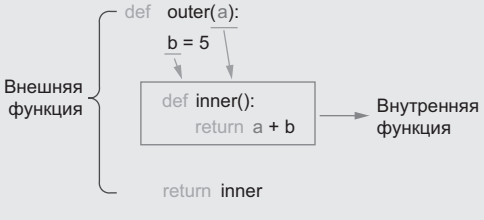
7.3.2. Строение функции-декоратора

В разделе 7.2 я упоминал о том, что декоратор является разновидностью функции высшего порядка. Как видно из листинга 7.8, функция `logging_time` является декоратором — разновидностью замыкания (дополнительная информация приведена во врезке). В данном разделе мы воспользуемся этим примером для анализа декоратора и выделения его ключевых элементов.

Что стоит за декораторами: замыкания

Декораторы являются разновидностями замыканий. На общем уровне замыкания представляют собой расширенную программную концепцию, присутствующую во многих современных языках, включая Kotlin, Swift и, конечно, Python. *Замыканием* (closure) называется внутренняя функция, которая создается и возвращается из внешней функции. При этом внутренняя функция должна использовать переменные из области видимости внешней функции — этот механизм называется *нелокальным связыванием переменной* (nonlocal variable binding).

Вероятно, вы заметили целый ряд новых терминов, включая области видимости и нелокальное связывание переменной. Для полного объяснения этой концепции потребовался бы отдельный раздел. В то же время это довольно важная тема, которая поможет вам понять некоторые сопутствующие методы, особенно декораторы. По этой причине я предоставляю диаграмму с основными компонентами замыканий. Следует помнить, что практические применения замыкания, такие как декораторы, могут использоваться без полного понимания замыканий, так что не огорчайтесь, если концепция покажется вам непонятной.



Функция высшего порядка, которая возвращает функцию

На этой схеме следует обратить внимание на три аспекта: в теле внешней функции создается внутренняя функция; внутренняя функция использует параметры, принадлежащие внешней функции; внешняя функция возвращает внутреннюю функцию как результат своей работы.

Создавая функцию вызовом внешней функции, вы создаете замыкание. Просмотрев замыкание, вы увидите, что это действительно внутренняя функция, созданная во внешней, и ее тоже можно вызвать:

```
>>> closure = outer(100)
>>> closure
<function outer.<locals>.inner at 0x7f89a812d5a0>
>>> closure()
105
```

Замыкание можно проанализировать на более глубоком уровне. Например, можно проверить, какие переменные связываются замыканием:

```
>>> closure.__closure__[0].cell_contents
100
>>> closure.__closure__[1].cell_contents
5
```

Основные элементы структуры: функция, генерирующая замыкание

Если на время отвлечься от подробностей реализации функции `logging_time`, основная структура выглядит так:

```
def logging_time_backbone(func):
    def logger(*args, **kwargs):
        # Тело подробно рассматривается позднее
        pass

    return logger
```

Напомним, что эта структура представляет функцию высшего порядка, то есть получающую функцию на входе или возвращающую функцию на выходе. По сути, декоратор обрабатывает функцию, и этот процесс называется *декорированием*. Но что происходит в процессе декорирования «за кулисами»? Чтобы продемонстрировать используемый механизм, начнем со следующего фрагмента кода:

```
def before_deco():
    pass

after_deco = logging_time(before_deco)

after_deco()
# Выводимые строки:
--- before_deco starts:
*** before_deco ends; used time: 0.00 s
```

Интересно, что вызов функции `after_deco` приводит к такому же выводу данных быстродействия, как у всех предыдущих функций с декоратором `@logging_time`. Вернувшись на шаг назад, вы увидите, что функция `after_deco` создается вызовом функции-декоратора `logging_time` с передачей функции `before_deco`.

Итак, как вы, возможно, уже поняли, декорирование представляет собой процесс создания замыкания через передачу внешней функции декоратору. Процесс изображен на рис. 7.2.

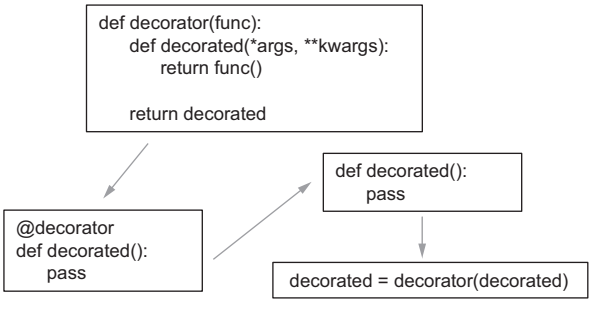


Рис. 7.2. Применение декоратора — это создание замыкания из функции-декоратора. Функция-декоратор представляет собой функцию высшего уровня, она получает функцию (которая должна быть декорирована) и возвращает функцию (декорированную функцию, замыкание). Заметим, что в инструкции присваивания можно использовать то же имя переменной. Интерпретатор Python вычисляет правую часть и присваивает вычисленное значение левой части. Так как имя остается неизменным, старое значение переменной заменяется новым

*args и **kwargs во внутренней функции

В разделе 6.4 вы ознакомились с концепциями `*args` и `**kwargs` и увидели, как с их помощью сделать так, чтобы пользователи могли передавать любое количество позиционных и ключевых аргументов соответственно. Причина для использования `*args` и `**kwargs` во внутренней функции остается прежней: декоратор должен быть совместим со всеми функциями независимо от их сигнатур вызова.

Чтобы продемонстрировать необходимость `*args` и `**kwargs`, рассмотрим декоратор, в котором они не используются, и выясним, с какими проблемами при этом можно столкнуться. Для простоты примем, что декоратор `monitor` сообщает, когда была вызвана функция:

```
def monitor(func):
    def monitored():
        print(f"*** {func.__name__} is called")
        func()

    return monitored
```

Если применить этот декоратор к функции, которая не получает параметров, все работает нормально:

```
@monitor
def example0():
    pass
example0()
# Вывод: *** example0 is called
```

Но если применить этот декоратор к функции, получающей один или несколько параметров, происходит ошибка `TypeError`:

```
@monitor
def example1(param0):
    pass

example1("a string")
# ERROR: TypeError: monitor.<locals>.monitored() takes 0 positional
➥ arguments but 1 was given
```

Сообщение об ошибке указывает, где кроется проблема. В четвертой строке функции-декоратора `monitor` декорируемая функция вызывается в форме `func()` без передачи параметров! Но декорируемая функция `example1` рассчитывает получить один позиционный аргумент. Как нетрудно догадаться, такая несовместимость значительно ограничивает возможности применения декораторов. Таким образом, для обеспечения максимальной гибкости декораторов важно включить во внутреннюю функцию `*args` и `**kwargs`, потому что созданная внутренняя функция будет декорированной, а использование `*args` и `**kwargs` обеспечивает совместимость внутренней функции с любой сигнатурой вызова.

СОПРОВОЖДАЕМОСТЬ Используйте `*args` и `**kwargs` во внутренней функции декоратора для обеспечения максимальной гибкости декоратора.

Инструкция return во внутренней функции

В разделе 6.2 упоминалось о том, что каждая функция Python возвращает значение — неявно в виде `None` или явно. Поэтому при определении внутренней функции не забудьте добавить инструкцию `return`. Точнее говоря, возвращаемым значением должно быть значение, которое вы получаете при вызове декорируемой функции.

Заметим, что при выборе места для размещения инструкции `return` необходима осторожность. Как вам, вероятно, известно, код после инструкции `return` выполняться не может, потому что `return` завершает выполнение текущей функции и передает управление в точку вызова. А значит, если вы хотите применить операции после вызова декорируемой функции, возвращаемое значение необходимо сохранить во временной переменной. После дополнительных операций эта переменная возвращается функцией. Именно это было сделано в функции `logging_time` в листинге 7.8. Схема изображена на рис. 7.3.

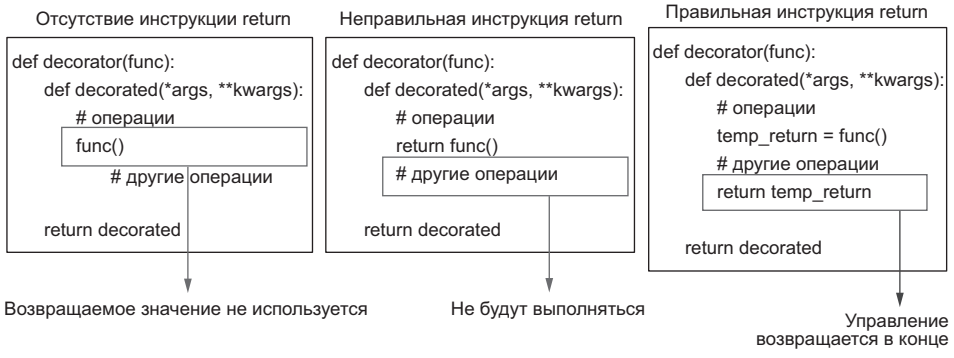


Рис. 7.3. Инстркция return размещается в конце внутренней функции. Во-первых, не забудьте добавить return. В противном случае вы измените поведение декорируемой функции, так как ожидаемое возвращаемое значение пропадет во внутренней функции. Во-вторых, возвращаемое значение должно размещаться в конце внутренней функции, а не в середине

7.3.3. Упаковка для передачи метаданных декорируемой функции

В предыдущих разделах я представил все основные возможности декораторов и описал процедуру создания декоратора `logging_time`, чтобы измерить продолжительность выполнения любой декорируемой функции. Но процесс декорирования может привести к потере метаданных декорируемой функции, например ее doc-строки. В этом разделе вы узнаете, как сохранить метаданные декорируемой функции. Прежде чем переходить к решению, просмотрите следующий код и подумайте, какие потенциальные проблемы могут возникнуть после декорирования:

```
def say_hi(person):
    """Greet someone"""
    print(f"Hi, {person}")

@logging_time
def say_hello(person):
    """Greet someone"""
    print(f"Hello, {person}")

print(say_hi.__doc__, say_hi.__name__, sep="; ")
# Вывод: Greet someone; say_hi

print(say_hello.__doc__, say_hello.__name__, sep="; ")
# Вывод: None; logger
```

Как следует из этого кода, без декорирования мы получаем doc-строку функции `say_hi` с помощью обращения к ее атрибуту `__doc__`. С другой стороны, после

декорирования doc-строка `say_hello` становится недоступной. Аналогичным образом декорирование изменяет имя функции (доступное через атрибут `__name__`). Эти атрибуты функций, включая `__doc__` и `__name__` (называемые метаданными функции), зависят от процесса декорирования. Почему? Подумайте, прежде чем читать дальше.

ПОДСКАЗКА Декорирование преобразует исходную функцию в замыкание, то есть внутреннюю функцию, создаваемую из декоратора.

При определении функции без декоратора идентификатор (имя функции) представляет определенную функцию и связанные с ней операции. С другой стороны, при определении функции с декоратором декорируемая функция — это не просто функция, как кажется на первый взгляд. Внутренняя функция создается и возвращается функцией-декоратором и является *замыканием*. Таким образом, обращение к атрибуту `__doc__` функции `say_hello` эквивалентно обращению к атрибуту `__doc__` внутренней функции `logging_time`, то есть `logger`. Чтобы убедиться в этом, проведите эксперимент и добавьте doc-строку во внутреннюю функцию:

```
def logging_time_doc(func):
    def logger(*args, **kwargs):
        """Log the time"""
        print(f"--- {func.__name__} starts")
        start_t = time.time()
        value_returned = func(*args, **kwargs)
        end_t = time.time()
        print(f"*** {func.__name__} ends; used time:
        ➥ {end_t - start_t:.2f} s")
        return value_returned

    return logger

@logging_time_doc
def example_doc():
    """Example function"""
    pass

print(example_doc.__doc__)
# Вывод: Log the time
```

Вывод подтверждает наш прогноз — это действительно doc-строка внутренней функции декоратора. Если применить этот декоратор к нескольким функциям, то все декорируемые функции будут использовать одну и ту же doc-строку и имя, соответствующее внутренней функции! Конечно, это неприемлемо. К счастью, Python предоставляет решение: можно использовать декоратор `wraps` из модуля `functools`, который обеспечит сохранение правильных метаданных в декорируемой функции. Эффект от его применения продемонстрирован в листинге 7.9.

Листинг 7.9. Упаковка декорируемой функции

```
import functools

def logging_time_wraps(func):
@functools.wraps(func)
    def logger(*args, **kwargs):
        """Log the time"""
        print(f"--- {func.__name__} starts")
        start_t = time.time()
        value_returned = func(*args, **kwargs)
        end_t = time.time()
        print(f"*** {func.__name__} ends; used time:
        ➥ {end_t - start_t:.2f} s")
        return value_returned

    return logger

@logging_time_wraps
def example_wraps():
    """Example function"""
    pass

print(example_wraps.__doc__, example_wraps.__name__, sep="; ")
# Вывод: Example function; example_wraps
```

Такой подход вполне допустим, так как декораторы являются функциями высшего порядка и могут получать сколько угодно функций в аргументах. Вообще говоря, эта возможность — передача параметров декораторам — относится к более высокому уровню, и обычно можно обойтись без нее. Но мне хотелось завершить этот раздел чем-то необычным!

> **СОПРОВОЖДАЕМОСТЬ** Не забудьте использовать декоратор `wraps` для сохранения метаданных декорируемой функции, в частности doc-строки и имени.

7.3.4. Обсуждение

Вероятно, материал этого раздела является самым сложным из всего, что рассматривалось до настоящего момента. Однако после его изучения вы можете с полным правом гордиться собой — мы разобрали довольно сложные концепции и построили полезный декоратор, который пригодится при оптимизации. Теперь вы знаете, что собой представляет замыкание и почему декоратор рассматривается как практическое применение замыканий. При определении декоратора важно использовать декоратор `wraps` для переноса метаданных декорируемой функции.

7.3.5. Задача

Майк — веб-разработчик, выбравший Python основным языком. Для работы ему потребовалось определить несколько декораторов, которые могут получать

аргументы. Помогите ему написать функцию-декоратор (допустим, `logging_time_app`), которая получает аргумент. Декоратор делает то же, что и декоратор `logging_time`. Параметр содержит строку с именем приложения, которая становится префиксом для всех строк, выводимых функцией `print`. При использовании декоратора мы хотим добиться следующего эффекта:

```
@logging_time_app("Task Tracker")
def example_app():
    pass

example_app()
# Выводимые строки:
Task Tracker --- example_app starts
Task Tracker *** example_app ends; used time: 0.00 s
```

ПОДСКАЗКА 1 Когда параметр используется в `@decorator(param)`, сначала вызывается функция высшего порядка `decorator` с параметром `param`, а затем эта функция возвращает другой декоратор — допустим, с именем `true_decorator`. Затем `true_decorator` применяется к декорируемой функции так, как если бы мы использовали `@true_decorator`.

ПОДСКАЗКА 2 Не бойтесь создавать функцию высшего порядка в другой функции высшего порядка, если обе функции высшего порядка являются декораторами!

7.4. КАК ИСПОЛЬЗОВАТЬ ФУНКЦИИ-ГЕНЕРАТОРЫ В КАЧЕСТВЕ ПОСТАВЩИКА ДАННЫХ, ЭФФЕКТИВНО РАСХОДУЮЩЕГО ПАМЯТЬ

В любом приложении главная роль отводится данным. С развитием дисциплин data science и машинного обучения многие пользователи стали использовать Python для обработки громадных объемов данных — гигабайтов и более. При работе с такими объемами данных их загрузка в память может занимать минуты и даже часы. Если обработка данных состоит из нескольких фаз, каждая фаза может занимать много времени, а сбой в любой фазе усложнит отладку. Помимо увеличения времени ожидания, самым, пожалуй, серьезным ограничением становится тот факт, что на некоторых компьютерах просто не хватит памяти для обработки таких объемов.

Рассмотрим простой пример с большим объемом данных. Замечу, что я мог бы использовать еще большее число, но это усложнило бы запуск примера на обычном компьютере, поэтому я ограничился умеренно большим числом. Допустим, требуется вычислить сумму квадратов чисел от 1 до 1 000 000. В типичном решении мы создаем объект `list` для хранения чисел, а затем вычисляем их сумму:

```
upper_limit = 1_000_000

squares_list = [x*x for x in range(1, upper_limit + 1)]

sum_list = sum(squares_list)
```

Конечный индекс не используется, его необходимо скорректировать на 1

ВОПРОС Сможете ли вы написать функцию, декорированную `logging_time`, для определения затрат времени на выполнение суммирования?

При выполнении этого кода вы заметите, что получение результата требует определенных временны́х затрат. Также обратите внимание, что объект занимает значительный объем памяти:

```
print(squares_list.__sizeof__())

# Вывод: 8448712
```

На других компьютерах могут быть получены другие результаты из-за различий в механизмах хранения данных

В этом разделе вы научитесь пользоваться функциями-генераторами для получения необходимых данных без лишних затрат памяти.

7.4.1. Создание генератора для получения квадратов

Генератор представляет собой особую разновидность итератора, создаваемого функцией-генератором. Так как генератор является итератором, он выдает свои элементы один за другим. Особенность генератора заключается в том, что он не хранит свои элементы в памяти, а получает и выдает их по мере надобности. Эта характеристика означает, что он эффективен по затратам памяти. В этом разделе мы займемся изучением генераторов.

Для начала решим задачу с применением нового приема: использования генераторов для вычисления суммы квадратов. Решение приведено в листинге 7.10.

Листинг 7.10. Создание генератора для вычисления суммы квадратов

```
def perfect_squares(limit):
    n = 1
    while n <= limit:
        yield n * n
        n += 1

squares_gen = perfect_squares(upper_limit)

sum_gen = sum(squares_gen)

assert sum_gen == sum_list == 333333833333500000
```

Функция `perfect_squares` является генераторной функцией. Вызывая эту функцию с передачей `upper_limit`, мы создаем генератор с именем `squares_gen`. Этот генератор генерирует квадраты натуральных чисел: 1^2, 2^2, 3^2, 4^2 и так далее

до 1 000 000². Как и ожидалось, сумма квадратов, полученных от генератора, совпадает с результатом, полученным из списка `squares_list`.

Механизм, обеспечивающий работу генератора, кроется в теле функции-генератора. Главное, на что следует обратить внимание, — ключевое слово `yield`, фирменный признак такой функции. Каждый раз, когда управление передается в строку с `yield`, генератор выдает элемент `n * n`. А самое замечательное в генераторах — то, что они помнят, какой элемент должен быть сгенерирован следующим. Схема работы генератора изображена на рис. 7.4.

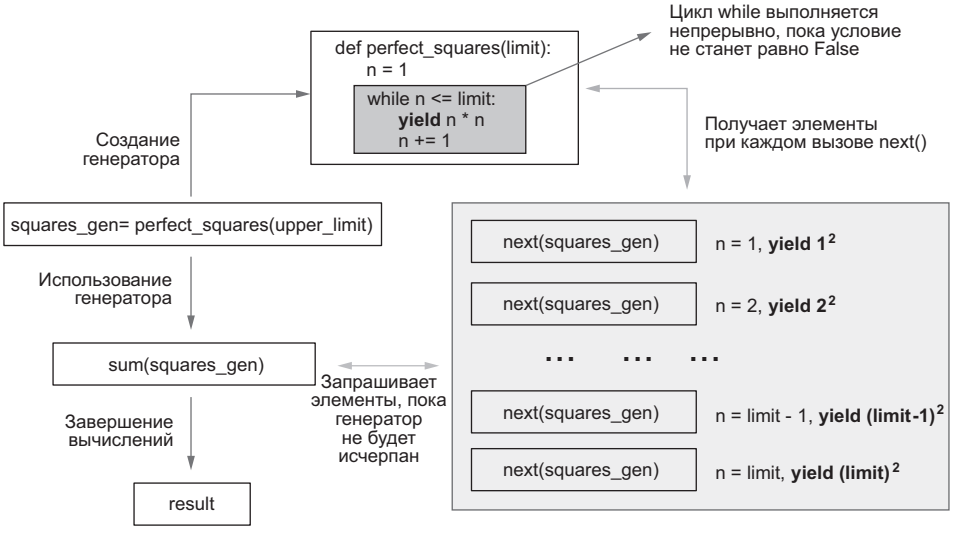


Рис. 7.4. Схема создания и использования генератора. Генератор создается вызовом функции-генератора. При использовании генератор перебирает свои элементы в цикле while. Каждый раз, когда генератор встречает ключевое слово yield, он незаметно для пользователя вызывает next (серый прямоугольник). Когда условие (n <= limit) перестает выполняться, цикл while завершается, ключевое слово yield становится недостижимым и перебор заканчивается

Как видно из рис. 7.4, генератор, по сути, является итератором, так что использование генератора подразумевает вызов функции `next`. Каждый вызов `next(squares_gen)` возобновляет выполнение генератора с точки, на которой он остановился: строки, следующей за выполнением последней инструкции `yield`. Так как инструкция `yield` является частью цикла `while`, цикл продолжает выполняться, и в каждом цикле `yield` встречается только один раз. При завершении цикла все элементы были сгенерированы, генератор исчерпан, а перебор завершен.

НАПОМИНАНИЕ Если вызвать `next` для итераторов вручную, произойдет исключение `StopIteration`.

Важное обстоятельство: `yield` отличается от инструкции `return`, которая завершает текущее выполнение и возвращает управление в точку вызова. А инструкция `yield` приостанавливает текущее выполнение и временно уступает управление стороне вызова. По запросу выполнение продолжается. Этот сценарий напоминает управление машиной на дороге с двухполосным движением: вы можете при необходимости уступить дорогу другим машинам, а после этого возвращаетесь на свою полосу.

7.4.2. Использование генераторов для экономии памяти

В предыдущем разделе вы научились создавать генераторы. Но стоят ли они затраченных усилий? В данном разделе вы получите ответ на этот вопрос.

Самая важная особенность генератора заключается в том, что он выдает элементы по запросу. С этой особенностью тесно связана концепция программирования, которая называется *отложенным, или ленивым, вычислением* (lazy evaluation): операции или переменные вычисляются только тогда, когда это становится необходимым. Генераторы не создают все элементы заранее. Вместо этого генератор создает следующий элемент только тогда, когда получит соответствующий запрос.

Отложенное вычисление

Отложенное вычисление существует в разных формах в разных языках программирования, например в Kotlin и Swift. Объект может содержать атрибут, в котором содержится значительный объем данных, но этот атрибут не критичен для его работы. При создании такого объекта на подготовку атрибута уходит слишком много времени, прежде чем вы сможете использовать объект. Вместо этого для атрибута можно включить «ленивое» вычисление, при котором объект создается без этого атрибута. Он будет подготовлен только при первом обращении.

Так как генераторы выдают свои элементы только по запросу, они эффективно расходуют память. Вы уже видели, что объект `list` с квадратами чисел от 1 до 1 миллиона занимает более 8 Мбайт (точнее, 8 448 712 байт). Теперь пора разобраться с тем, сколько памяти расходует генератор, который генерирует такой же объем данных:

```
squares_gen = perfect_squares(upper_limit)

print(squares_gen.__sizeof__())
# Вывод: 88  ◄——— На вашем компьютере значение может быть другим
```

Размер генератора составляет всего 88 байт — около 0.001 процента от объекта `list`. Это объясняется тем, что ему достаточно знать только текущее состояние: когда понадобится следующий элемент, генератор может начать с текущего состояния и создать этот следующий элемент. А вот объект `list` должен загрузить все элементы заранее, прежде чем он сможет использовать их.

7.4.3. Использование генераторных выражений

Вы уже видели, насколько полезны генераторы. Однако прежде чем использовать генератор, приходится создавать функцию-генератор, что может быть довольно утомительно. В этом разделе будет представлен альтернативный способ создания генераторов: применить выражение в так называемых *генераторных выражениях* (generator expression).

Рассказывая о включениях в разделе 5.2, я упоминал о том, что не существует включений кортежей, иначе можно было бы использовать следующий синтаксис: `(выражение for x in итерируемый_объект)`. В сущности, это выражение близко к синтаксису генераторных выражений. А теперь перепишем функцию `perfect_squares` в виде генераторного выражения:

```
>>> squares_gen_exp = (x * x for x in range(1, upper_limit))
>>> squares_gen_exp
<generator object <genexpr> at 0x7f89a8111f50>
```

Вместо использования ключевого слова `yield` в функциях-генераторах генераторное выражение напрямую указывает, какие данные должны выдаваться. Что касается синтаксиса, следите за использованием круглых скобок: если вы по ошибке используете квадратные скобки, у вас получится список. Чтобы продемонстрировать, что генератор является итератором, можно использовать функцию `next` для получения элементов от генератора по одному:

```
>>> next(squares_gen_exp)
1
>>> next(squares_gen_exp)
4
>>> next(squares_gen_exp)
9
```

Вычислим сумму для генераторного выражения:

```
>>> sum_gen_exp = sum(squares_gen_exp)
>>> sum_gen_exp
333332833333499986
```

Работает! Но подождите: почему сумма отличается на 14 от вычисленной ранее?

ВОПРОС Что мы сделали перед использованием `squares_gen_exp`?

Как уже упоминалось, генератор выдает свои элементы в отложенном режиме, запоминая свое состояние. Первый вызов `next` получает 1, второй — 4, а третий — 9. При вызове `sum(squares_gen_exp)` генератор все еще помнит свое состояние, поэтому начинает выдавать следующий элемент, который равен 16. Как вы, вероятно, уже поняли, разность в суммах происходит из-за невозможности использовать первые три элемента, которые уже были использованы при трех ручных вызовах `next`.

С точки зрения синтаксиса функцию `sum` можно вызвать при помощи генераторного выражения напрямую, что снимает необходимость в создании промежуточной переменной. Если генератор не слишком сложен, такое решение предпочтительно:

```
>>> sum(x*x for x in range(4))
14
```

Обратите внимание: в этом выражении мы опускаем круглые скобки для генераторного выражения, так как при использовании внутри другой пары круглых скобок они не обязательны.

7.4.4. Обсуждение

Во внутренней реализации генераторов используется ключевое слово `yield`. Кроме генераторов, `yield` также используется в другой расширенной функциональности — *сопрограммах*; эта разновидность сопрограмм называется *сопрограммами на базе генераторов*. Впрочем, эти сопрограммы в настоящее время постепенно выводятся из Python и встречаются только в устаревших проектах с использованием его старых версий. Так что если вы плохо разбираетесь в сопрограммах на базе генераторов, не беспокойтесь.

7.4.5. Задача

Джеймс ведет начальный курс программирования на Python для студентов математического факультета. Чтобы использовать в примерах знакомые им концепции, он обратился к числам Фибоначчи — последовательности чисел, в которой каждый элемент равен сумме двух предыдущих: 0, 1, 1, 2, 3, 5, 8, 13 и т. д. Он предложил своим студентам написать функцию-генератор с верхней границей. Функция должна производить генератор, выдающий числа Фибоначчи до тех пор, пока не будет достигнута эта заданная граница.

ПОДСКАЗКА Можно определить первые два числа вручную, а затем строить остальные элементы по формуле $значение_{n+2} = значение_n + значение_{n+1}$.

7.5. КАК СОЗДАТЬ ЧАСТИЧНЫЕ ФУНКЦИИ ДЛЯ УПРОЩЕНИЯ ВЫЗОВА ФУНКЦИЙ

Функции не изолируются от других компонентов вашего приложения. Они взаимодействуют с другими сущностями, получая входные данные и возвращая обработанный вывод. Для повышения гибкости функции мы часто определяем в функции несколько параметров, чтобы она могла обрабатывать разные входные данные и получать необходимые результаты в разных ситуациях.

Предположим, вы используете Python в data science. Имеется следующая функция для выполнения статистического моделирования с заданным набором данных:

```
def run_stats_model(dataset, model, output_path):
    # Обработка набора данных
    # Применение модели
    # Сохранение статистики в выходном каталоге
    calculated_stats = 123  ◄──  Номинальное значение, обеспечивающее работу кода
    return calculated_stats
```

Эта функция настолько важна и универсальна, что вы применяете ее в нескольких проектах. В каждом проекте используется одна и та же модель, а данные выводятся в одну и ту же папку для разных наборов данных. Следующий фрагмент кода показывает, что может происходить в рамках проекта:

```
# Проект A
run_stats_model(dataset_a1, "model_a", "project_a/stats/")
run_stats_model(dataset_a2, "model_a", "project_a/stats/")
run_stats_model(dataset_a3, "model_a", "project_a/stats/")
run_stats_model(dataset_a4, "model_a", "project_a/stats/")
```

Как видите, возникает повторение, потому что одни и те же параметры используются между разными вызовами функции. Возможно, вам сразу придет в голову назначить параметры по умолчанию для функции `run_stats_model`. Но такое решение не оптимально, потому что все равно придется задавать эти параметры для других проектов:

```
# Проект B
run_stats_model(dataset_b1, "model_b", "project_b/stats/")
run_stats_model(dataset_b2, "model_b", "project_b/stats/")
run_stats_model(dataset_b3, "model_b", "project_b/stats/")
run_stats_model(dataset_b4, "model_b", "project_b/stats/")

# Проект C
run_stats_model(dataset_c1, "model_c", "project_c/stats/")
run_stats_model(dataset_c2, "model_c", "project_c/stats/")
run_stats_model(dataset_c3, "model_c", "project_c/stats/")
run_stats_model(dataset_c4, "model_c", "project_c/stats/")
```

В следующем разделе вы узнаете о новой конструкции — так называемых *частичных функциях* (partial functions), а также о том, как использовать частичные функции для упрощения вызовов функций с параметрами, общими для разных проектов.

7.5.1. «Локализация» общих функций для упрощения вызовов

В условиях нашей задачи для функции `run_stats_model` в каждом проекте используется одна и та же модель и выходной путь. Так как функция `run_stats_model` совместно используется в разных проектах, ее применение в рамках каждого проекта можно считать локальным. Таким образом, эту задачу можно переформулировать в виде вопроса о локализации. В этом разделе рассматривается рабочее решение, основанное на уже существующих у вас знаниях.

Так как в каждом проекте используется одна и та же модель и выходной путь, мы можем создать адаптированную версию общей функции для каждого проекта. Например, в начале файла проекта A можно создать функцию следующего вида:

```
def run_stats_model_a(dataset):
    model_stats = run_stats_model(dataset, "model_a", "project_a/stats/")
    return model_stats
```

> **УДОБОЧИТАЕМОСТЬ** Хотя здесь можно было бы применить запись `return run_stats_models(dataset,"model_a","project_a/stats/")`, я предпочитаю использовать промежуточную переменную, чтобы более ясно показать природу значения, возвращаемого при вызове функции. Всегда лучше возвращать четко определенную переменную, вместо того чтобы возвращать напрямую что-то, полученное при вызове другой функции.

Функция `run_stats_model_a` устроена прямолинейно. Она предоставляет вспомогательную обертку для функции `run_stats_models`. С этой локализованной функцией все исходные вызовы `run_stats_models` принимают следующий вид:

```
# Проект A
run_stats_model_a(dataset_a1)
run_stats_model_a(dataset_a2)
run_stats_model_a(dataset_a3)
run_stats_model_a(dataset_a4)
```

7.5.2. Создание частичной функции для локализации функции

В предыдущем разделе обычная функция определяется для локализации общей функции. Такое решение работает. Но его эффективность оставляет желать лучшего. Дело в том, что Python уже реализует эту функциональность за нас.

Более питоническое решение основано на использовании функции `partial` для локализации общей функции:

```
from functools import partial

run_stats_model_a = partial(run_stats_model, model="model_a",
     output_path="project_a/stats/")

run_stats_model_a("dataset_a")
# Вывод: 123
```

Функция `partial` находится в модуле `functools`, в котором собраны расширенные средства для работы с функциями в стандартной библиотеке Python. В функции `partial` указывается общая функция и все дополнительные параметры, которые необходимо задать, — в данном случае специфическая для проекта модель и путь для вывода результата.

> **НАПОМИНАНИЕ** Ранее мы использовали `wraps` для сохранения метаданных функции при декорировании. Функция `wraps` также находится в модуле `functools`.

Созданная функция `run_stats_model_a` называется *частичной* (partial) функцией. Когда мы вызываем ее, задавать общие параметры не нужно — это уже сделано за нас. Используя эту функциональность, вы можете создать отдельные частичные функции для каждого проекта. Частичные функции способны значительно упростить сигнатуру вызова и сделать код более удобочитаемым.

7.5.3. Обсуждение

Раздел 7.5 получился довольно коротким. Я на простом примере продемонстрировал полезный механизм частичных функций. По мере накопления кодовой базы вы увидите, как часто возникает необходимость использования некоторых функций в нескольких местах. В таком случае можно создать частичные функции на базе существующих функций. Эти частичные функции фиксируют общие параметры в конкретной точке, и параметры можно будет опустить при вызове, чтобы код стал более понятным.

7.5.4. Задача

Частичные функции создаются на базе других функций. Как определить, из какой функции была создана частичная функция?

> **ПОДСКАЗКА** Частичная функция содержит дополнительные атрибуты по сравнению с обычными функциями. Для проверки ее атрибутов можно воспользоваться вызовом `dir(partial_function_created)`. Просмотрите список и определите, какой атрибут вам нужен.

ИТОГИ

- Лямбда-функции предназначены для выполнения небольших одноразовых операций. Следовательно, лямбда-функции не должны присваиваться переменным.
- Хотя лямбда-функции удобны, не пытайтесь изобретать велосипед. По возможности используйте встроенные функции для выполнения той же работы: например, используйте встроенную версию `int` вместо `lambda x: int(x)`.
- В языке Python функции — это полноправные сущности, то есть они тоже являются объектами. Любые операции, которые могут выполняться с объектами, также могут применяться к функциям.
- Функции высшего порядка получают функции на входе и/или возвращают функции на выходе. К числу часто используемых функций высшего порядка относятся `sorted`, `map` и `filter`.
- Используя декораторы, можно применять дополнительную функциональность к другим функциям без изменения исходной функциональности декорируемых функций.
- Хотя в этой главе не приводится формальное определение замыканий, это очень важная концепция из области программирования. Замыкания представляют собой внутренние функции, создаваемые и возвращаемые функциями высшего порядка; кроме того, в них связываются переменные, определенные функцией высшего порядка. Декораторы являются практическим применением концепции замыканий.
- Генераторы могут создаваться функциями-генераторами, в которых ключевое слово `yield` выдает элемент и временно отдает управление. При повторном вызове генератор помнит свое состояние и продолжает выполнение со следующего элемента или завершает итерацию.
- По сравнению с другими итераторами генераторы более эффективны по затратам памяти, так как они не загружают все свои элементы заранее в отличие от традиционных итераторов (таких, как списки и кортежи, которые должны загрузить все элементы до начала перебора).
- Частичные функции используются для фиксирования некоторых параметров общих функций. Таким образом создается локализованная версия функции, предназначенная для конкретного проекта. Частичная функция избавляет от необходимости указывать зафиксированные параметры, что делает код более чистым.

Часть 3
Определение классов

Встроенные структуры данных отличаются универсальностью и могут быть использованы независимо от того, какого рода приложения вы создаете. Несмотря на популярность этих типов данных, их универсальность не позволяет определять специализированные данные и операции для таких объектов. Поэтому почти всегда приходится определять собственные классы, адаптированные для конкретной задачи. В таких классах задаются различные атрибуты, в которых могут храниться специализированные данные, а также наборы методов для выполнения специализированных операций. По мере роста сложности приложения в нем определяются все новые классы; вы должны позаботиться о том, чтобы эти классы согласованно и логично взаимодействовали друг с другом. Разумеется, спроектировать классы, поведение которых хорошо подходит для целей приложения, непросто. В этой части книги рассматриваются важнейшие практические приемы для определения пользовательских классов.

8 Определение пользовательских классов

В этой главе

- ✓ Определение метода инициализации
- ✓ Создание методов экземпляров, статических методов и методов класса
- ✓ Применение инкапсуляции в классах
- ✓ Создание правильных строковых представлений
- ✓ Определение надкласса и подклассов

В центре любого приложения находятся его данные. Хотя встроенные типы данных полезны для управления данными, их возможности все же ограниченны, потому что их атрибуты и методы рассчитаны только на базовую функциональность. Это относится и к именованным кортежам (раздел 3.3). Возможно, вы заметили, что у именованных кортежей нет удобных методов для выполнения операций с задачами. Но приложение для управления задачами, то есть наш таск-менеджер (как и любое приложение вообще), предназначено для конкретных практических целей, и нужны модели данных, способные достигать этих целей. Поэтому пользовательские классы становятся незаменимыми компонентами вашего приложения. Определив подходящие атрибуты для класса, вы сможете лучше организовать данные, необходимые для приложения. А определение

подходящих методов позволит приложению более эффективно обрабатывать эти данные.

В этой главе мы сосредоточимся на определении атрибутов и различных категорий методов для вашего класса. Рассматриваемые вопросы в основном будут демонстрироваться на примере класса `Task` из таск-менеджера. Целью определения хорошего пользовательского класса должно быть его удобство для пользователя — не только широта возможностей в отношении атрибутов и методов (то есть то, что должно быть доступно в классе), но и простота их сопровождения в плане четко структурированной реализации (то есть то, как они организованы).

8.1. КАК ОПРЕДЕЛИТЬ МЕТОД ИНИЦИАЛИЗАЦИИ ДЛЯ КЛАССА

Когда мы работаем со встроенными классами (такими, как `list` или `dict`), то для создания их экземпляров вызываются конструкторы этих классов. Процесс создания экземпляра (instance) называется *инстанцированием* (instantiation). Во внутренней реализации создание экземпляра включает вызов метода `__init__`, как показано в листинге 8.1.

Листинг 8.1. Создание класса Task без содержательной инициализации

```
class Task:
    def __init__(self):
        print("Creating an instance of Task class")

task = Task()
# Вывод: Creating an instance of Task class
```

Как видите, мы вызываем конструктор `Task()` для создания экземпляра, инициируя тем самым вызов метода `__init__`. Если вас интересует, почему метод называется `init`, то это сокращение от «инициализация» (initialization), то есть определение исходного состояния объекта. Поэтому это один из важнейших методов, который почти всегда определяется в пользовательских классах. В этом разделе вы узнаете некоторые правила определения метода инициализации `__init__`.

8.1.1. Загадочный self: первый параметр __init__

Хотя метод `__init__` в листинге 8.1 не содержит никакой реализации, он получает один параметр: `self`. Впрочем, если вы когда-либо читали чужой код, вы наверняка видели, что метод `__init__` всегда получает первый параметр с именем `self`. Если вас заинтересовало, что такое `self`, то именно в этом разделе мы раскроем его секреты, ответив на четыре вопроса:

- Что означает `self`?
- Почему не нужно предоставлять аргумент для `self`?
- Является ли `self` ключевым словом?
- Обязательно ли использовать `self` как имя параметра?

self: текущий экземпляр

Первый вопрос: что означает `self`? Когда вы определяете методы в классе, в большинстве случаев эти методы предназначены для выполнения операций с конкретным экземпляром, как, например, метод `__init__`, который задает исходные значения атрибутов нового экземпляра. Следовательно, необходим удобный способ обращения к экземпляру. Если вы владеете другими языками объектно-ориентированного программирования, то уже знаете, что в этих языках встречаются слова `this`, `that`, `self` или `it` для обозначения экземпляра. В Python обозначение `self` используется для ссылок на объект экземпляра в определениях методов. Чтобы доказать, что `self` относится к только что созданному экземпляру объекта, воспользуемся встроенной функцией `id`, однозначно идентифицирующей объект в памяти:

```
class Task:
    def __init__(self):
        print(f"Memory address (self): {id(self)}")

task = Task()
# Вывод: Memory address (self): 140702458470768

task_address = f"Memory address (task): {id(task)}"
print(task_address)
# Вывод: Memory address (task): 140702458470768
```

На вашем компьютере значение будет другим; кроме того, при каждом запуске тоже будет выводиться новое значение

Из вывода следует, что адреса памяти `self` и `task` совпадают. Это означает, что обе ссылки указывают на один и тот же объект — только что созданный объект класса `Task`.

> **НАПОМИНАНИЕ** Функция `id` возвращает адрес объекта в памяти. Каждый объект обладает уникальным адресом памяти; следовательно, если адреса памяти двух объектов совпадают, это один и тот же объект.

Неявное присваивание значения self

При создании экземпляра вызовом конструктора `Task()` никакие аргументы не используются. Однако метод `__init__` должен получать один аргумент — `self`. Как объяснить это очевидное противоречие? Дело в том, что значение аргумента `self` задает сам Python (хотя и неявно). Python создает экземпляр вызовом `__new__` и передает его `__init__` в аргументе `self`. Чтобы понять,

как происходит неявное задание аргумента `self`, проследите за следующим фрагментом кода:

```
class Task:
    def __init__(self):
        print(f"__init__ gets called, creating object at {id(self)}")

    def __new__(cls):
        new_task = object.__new__(cls)
        print(f"__new__ gets called, creating object at {id(new_task)}")
        return new_task

task = Task()
# Выводимые строки:
__new__ gets called, creating object at 140702458469952
__init__ gets called, creating object at 140702458469952
```

В этом коде вызывается конструктор `Task()`. Обратите внимание: при конструировании «за кулисами» последовательно автоматически вызываются два специальных метода: `__new__` и `__init__`. Метод `__new__` создает и **возвращает** новый экземпляр, а метод `__init__` ничего не возвращает. Чем вызваны эти различия в возвращаемом значении? Дело в том, что после вызова `__new__` вам необходимо обратиться к только что созданному экземпляру. Если метод `__new__` не будет возвращать новый экземпляр, вы не сможете к нему обратиться и использовать его. Напротив, метод `__init__` получает `self` в аргументе; он ссылается на новый экземпляр и обрабатывает экземпляр на месте.

Чтобы смоделировать тот факт, что процесс конструирования экземпляра состоит из двух фаз с вызовами `__new__` и `__init__`, мы вызовем эти два метода вручную. Учтите, что эта модель всего лишь демонстрирует лежащий в ее основе механизм и редко встречается в реальном коде:

```
task = Task.__new__(Task)
# Вывод: __new__ gets called, creating object at 140702458476192

Task.__init__(task)
# Вывод: __init__ gets called, creating object at 140702458476192
```

Сначала мы используем метод `__new__` для создания экземпляра `task`. Далее `task` используется в качестве аргумента `self` в методе `__init__`. Как легко определить по адресу памяти, мы оперируем одним и тем же экземпляром. Схема процесса изображена на рис. 8.1.

Из-за эквивалентности вызова конструктора и двухфазного процесса инстанцирования возможно рассматривать прямое использование конструктора как синтаксически удобное средство для представления этого двухфазного процесса. К тому же использование конструктора для создания экземпляра более компактно и лучше читается.

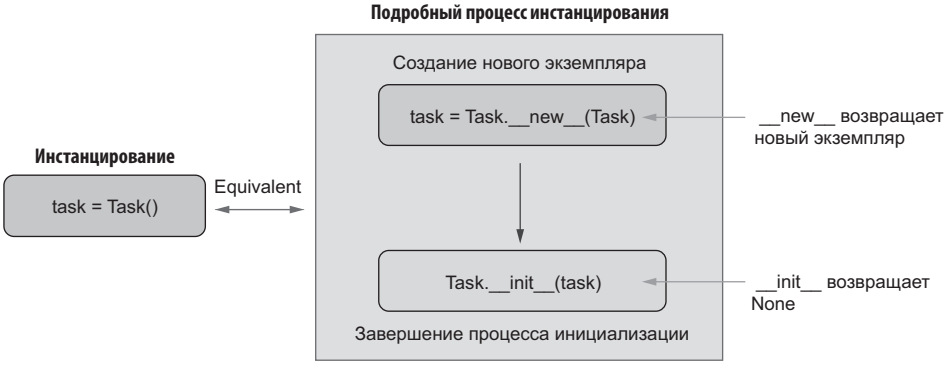


Рис. 8.1. Подробная схема инстанцирования. Когда вы создаете экземпляр вызовом конструктора, сначала вызывается метод __new__. Созданный экземпляр передается методу __init__ для завершения инициализации, где задаются атрибуты экземпляра

self не является ключевым словом

В языке Python `def` означает, что мы создаем функцию, а `for` — что мы выполняем цикл `for`. `def` и `for` являются примерами ключевых слов в Python; это означает, что они зарезервированы в языке для специальных операций. Так как `self` используется для ссылки на экземпляр в Python, что вроде бы является специальной операцией, у некоторых пользователей может сложиться впечатление, что `self` является ключевым словом. Но как вы вскоре увидите, это не так. Один из признаков ключевых слов — они не могут быть использованы в качестве имен переменных. Посмотрите на пример:

```
def = 5
# ERROR: SyntaxError: invalid syntax

class = 7
# ERROR: SyntaxError: invalid syntax

self = 9
# Работает!
```

Присвоить значение `def` или `class` невозможно, но присваивание значения `self` выполняется без проблем; это четко указывает, что `self` принципиально отличается от ключевых слов. На более формальном уровне проверить, является ли слово ключевым, можно с помощью модуля `keyword`, содержащего удобную функцию `iskeyword`:

```
import keyword

words_to_check = ["def", "class", "self", "lambda"]
for word in words_to_check:
    print(f"Is {word:^8} a keyword? {keyword.iskeyword(word)}")
```

```
# Выводимые строки:
Is   def    a keyword? True
Is  class   a keyword? True
Is   self   a keyword? False
Is  lambda  a keyword? True
```

Как показывает этот фрагмент, `def`, `class` и `lambda` распознаются функцией `iskeywords` как ключевые слова. А вот `self` ключевым словом не является.

> **ОБРАТИТЕ ВНИМАНИЕ** Полный список ключевых слов можно получить вызовом функции `kwlist` модуля `keyword`.

Использование self в качестве имени параметра

Мы знаем, что `self` обозначает экземпляр в `__init__` и не является ключевым словом. Мы видели, что в качестве первого аргумента `__init__` обычно указывается `self`. Напрашивается предположение, что он должен называться `self`. Однако использовать `self` в качестве имени параметра не обязательно. Вы можете использовать любое допустимое имя переменной (но не ключевое слово). В следующем фрагменте вместо `self` в `__init__` используется имя `this`:

```
class Task:
    def __init__(this):
        print("An instance is created with this instead of self.")

task = Task()
# Вывод: An instance is created with this instead of self.
```

Как видите, экземпляр класса `Tasks` без каких-либо проблем создается с `this`. С точки зрения синтаксиса использование `self` в `__init__` не обязательно. Тем не менее использование `self` в `__init__` является общепринятым соглашением, которое должно соблюдаться всеми программистами Python.

> **УДОБОЧИТАЕМОСТЬ** Следуйте общепринятым соглашениям, таким как использование `self` в `__init__`. При соблюдении этих соглашений другим пользователям будет проще читать ваш код, потому что они будут точно понимать, что вы имеете в виду.

8.1.2. Правильный выбор аргументов в __init__

В приведенных примерах я не включал в метод `__init__` никаких аргументов, кроме `self`. В этом разделе вы узнаете, что следует учитывать при выборе аргументов для метода `__init__`.

Метод `__init__` должен завершать процесс инициализации нового экземпляра, в частности присваивание значений важнейшим атрибутам в экземпляре. При обсуждении именованных кортежей в разделе 3.3 упоминалось о том, что класс `Task` должен хранить для каждой задачи три атрибута: название, описание

и степень срочности. В следующем фрагменте представлена модель данных, созданная на базе именованных кортежей:

```
from collections import namedtuple

Task = namedtuple("Task", "title desc urgency")

task = Task("Laundry", "Wash clothes", 3)

print(task)
# Вывод: Task(title='Laundry', desc='Wash clothes', urgency=3)
```

Как видите, при использовании модели данных на базе именованных кортежей экземпляр создается с указанием всех трех атрибутов. Поэтому когда мы создаем пользовательский класс, не относящийся к именованным кортежам, нужен механизм, который бы позволил пользователю задать эти атрибуты с передачей необходимых аргументов методу `__init__`:

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
```

При получении аргументов `__init__` проводит дополнительную инициализацию: задает исходные значения атрибутов на основании аргументов. Важно заметить, что аргументы должны быть связаны с атрибутами экземпляра. В теле метода `__init__` значения атрибутов экземпляра задаются на основании аргументов. С обновленной версией метода `__init__` создается экземпляр с передачей аргументов:

```
task = Task("Laundry", "Wash clothes", 3)
```

При создании экземпляра будут заданы все необходимые атрибуты. Для просмотра атрибутов нового экземпляра воспользуемся специальным атрибутом экземпляра `__dict__`. У нового экземпляра `task` эти атрибуты хранятся в виде объекта `dict`:

```
print(task.__dict__)
# Вывод: {'title': 'Laundry', 'desc': 'Wash clothes', 'urgency': 3}
```

В случае с классом `Task` этот конкретный пример относится к таск-менеджеру, но в вашем проекте будут использоваться разные пользовательские классы, обеспечивающие потребности в моделировании данных. Таким образом, вопрос в том, что необходимо учитывать при выборе аргументов метода `__init__`, когда вы строите собственный пользовательский класс. В общем случае руководствуйтесь следующими правилами:

- *Выберите необходимые аргументы*. Когда вы конструируете экземпляр, все его атрибуты должны быть настроены и готовы к использованию. Следо-

вательно, нужно выбрать аргументы, необходимые для задания атрибутов экземпляра.

- *Расставьте приоритеты аргументов*. Для работы пользовательского класса могут потребоваться десять исходных атрибутов, которые должны задаваться для нового экземпляра. При этом некоторые атрибуты всегда важнее других. Более важные атрибуты должны предшествовать менее важным.
- *Сделайте важные аргументы позиционными*. Это требование скорее является соглашением по стилю программирования, чем жестким правилом. Пользователь должен иметь возможность задать важные аргументы как позиционные, потому что вызов конструктора без указания ключевых аргументов обычно более понятен, чем вызов с ключевыми аргументами.
- *Ограничьте количество позиционных аргументов*. Этот пункт непосредственно связан с предыдущим. Хотя для метода `__init__` рекомендуется использовать позиционные аргументы, при слишком большом количестве позиционных аргументов читатель кода не поймет, какой аргумент к чему относится. По этой причине я не рекомендую использовать более четырех позиционных аргументов. При необходимости дополнительные аргументы следует сделать только ключевыми (см. раздел 6.4.1).
- *Определите подходящие значения по умолчанию*. `__init__` по своей сути является функцией. Чтобы упростить вызов этой функции, стоит задать значения по умолчанию для тех аргументов, которые большинство пользователей изменять не станет. Вполне возможно, что из десяти исходных атрибутов семь будут оставаться одинаковыми в большинстве практических ситуаций — для этих семи атрибутов определите значения по умолчанию.

8.1.3. Назначение всех атрибутов в __init__

В разделе 8.1.2 обсуждалось назначение аргументов в методе `__init__`. Все соответствующие атрибуты экземпляра инициализируются в теле метода `__init__`. У экземпляра бывает больше атрибутов, чем создается на основании аргументов `__init__`. Хотя значения атрибутов экземпляра можно задать где угодно в теле класса, лучше всего задавать все атрибуты в теле метода `__init__`. Данная практика рассматривается в этом разделе.

Начнем со следующего листинга, в котором атрибуты экземпляра инициализируются в разных местах. Учтите, что я не рекомендую применять эту схему, потому что она не сообщает, какие атрибуты содержит экземпляр.

Листинг 8.2. Присваивание значений атрибутов за пределами __init__

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
```

```
    def complete(self):
        self.status = "completed"

    def add_tag(self, tag):
        if not self.tags:
            self.tags = []
        self.tags.append(tag)
```

ЗАБЕГАЯ ВПЕРЕД Методы, первым параметром которых является `self`, называются методами экземпляров, они должны вызываться для конкретных экземпляров класса. Эти методы будут рассмотрены в разделе 8.2.

В листинге 8.2 кроме атрибутов `title`, `desc` и `urgency` мы задаем атрибуты `status` и `tags` в методах `complete` и `add_tag` соответственно. Инициализировать атрибуты экземпляров в любых других местах, кроме как внутри метода `__init__`, нежелательно по двум причинам:

- При попытке обратиться к таким атрибутам происходит ошибка `AttributeError`, если только вы не вызвали эти два метода, которые соответствующим образом задают значения атрибутов. Иначе говоря, если вы случайно обратитесь к атрибутам без вызова сопутствующих методов, в приложении произойдет фатальный сбой:

```
task = Task("Laundry", "Wash clothes", 3)
print(task.status)

# ERROR: AttributeError: 'Task' object has no attribute 'status'

task.complete()
print(task.status)
# Вывод: completed
```

- Пользователю будет труднее понять, какие атрибуты содержит экземпляр класса. Классы часто содержат разные виды функциональности, особенно в сложных приложениях. Если вы будете задавать атрибуты в нестандартных методах, пользователю придется изрядно потрудиться, разбираясь в составе атрибутов экземпляра.

По этим двум причинам все атрибуты следует задавать в `__init__`, даже если некоторые атрибуты будут обновляться вызовами других методов. В таких случаях атрибутам должно присваиваться разумное начальное значение. Схема представлена в листинге 8.3.

Листинг 8.3. Задание всех атрибутов в __init__

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
```

```
        self.status = "created"
        self.tags = []

    def complete(self):
        self.status = "completed"

    def add_tag(self, tag):
        self.tags.append(tag)
```

В обновленной схеме после создания экземпляра все его атрибуты будут заданы правильно; чтобы просмотреть их, выведите специальный атрибут `__dict__`:

```
task = Task("Laundry", "Wash clothes", 3)
print(task.__dict__)
# output: {'title': 'Laundry', 'desc': 'Wash clothes',
➥ 'urgency': 3, 'status': 'created', 'tags': []}
```

СОПРОВОЖДАЕМОСТЬ Размещая все атрибуты в `__init__`, вы четко показываете своим коллегам, какие атрибуты может содержать экземпляр класса. При обращении к любому атрибуту он всегда имеет определенное значение, что предотвращает возможные ошибки `AttributeError`.

Теперь к атрибутам `status` и `tags` можно обращаться без предварительного вызова методов `complete` и `add_tag`. Для читателей еще важнее, что они могут просмотреть метод `__init__`, чтобы получить информацию о доступных атрибутах экземпляра, а не просматривать атрибуты, спрятанные в разных методах (листинг 8.2). На рис. 8.2 сравниваются эти две схемы.

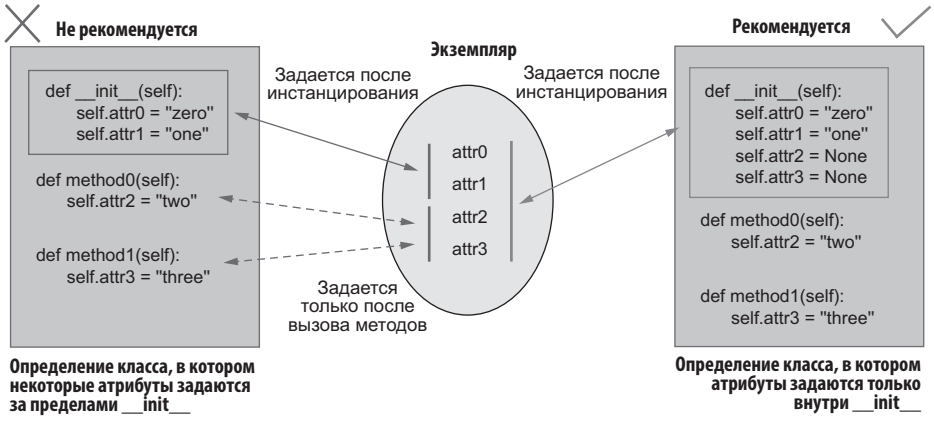


Рис. 8.2. Сравнение двух схем с присваиванием значений атрибутов экземпляра. В левой (не рекомендуемой) схеме атрибуты инициализируются в разных местах. В правой (рекомендуемой) схеме атрибуты инициализируются только в методе __init__, что ясно показывает читателям, какие атрибуты содержит экземпляр

8.1.4. Определение атрибутов класса за пределами метода __init__

Метод инициализации должен обеспечивать инициализацию экземпляра, определяя свои атрибуты на уровне экземпляра. В списке атрибутов могут присутствовать атрибуты, общие для всех экземпляров. Такие атрибуты не следует включать в число атрибутов экземпляров — лучше определить их как атрибуты класса. Эта возможность рассматривается в данном разделе.

> **ОСНОВНЫЕ ПОНЯТИЯ** *Атрибутами класса* называются атрибуты, принадлежащие классу. Все экземпляры класса совместно используют такие атрибуты через класс.

Для простоты предположим, что у каждой задачи имеется атрибут `user`, указывающий пользователя, который создал задачу. Теоретически `user` можно было бы сделать атрибутом экземпляра при помощи следующего метода `__init__`:

```
def __init__(self, title, desc, urgency, user):
    self.title = title
    self.desc = desc
    self.urgency = urgency
    self.user = user
```

Так как `user` является атрибутом экземпляра, понятно, что вашему приложению потребуется больше памяти, так как данные пользователя должны храниться для каждого экземпляра. Однако в приложении после ввода регистрационных данных будет существовать только один пользователь, который создал все задачи. Таким образом, все экземпляры должны использовать общий атрибут `user`. Чтобы сократить затраты памяти на хранение `user` с каждым экземпляром, в данном случае следует создать атрибут класса:

```
class Task:
    user = "the logged in user"

    def __init__(self, title, desc, urgency):
        pass
```

Возможно, в зависимости от модели данных вам потребуется определить дополнительные атрибуты класса в вашем классе. Определение атрибутов класса — важный способ экономии памяти, так как все экземпляры совместно используют одни и те же атрибуты, для чего они хранят ссылки на один используемый объект в памяти. С точки зрения удобочитаемости важно помнить, что атрибуты класса должны размещаться под заголовком определения класса, но над методом `__init__`.

> **УДОБОЧИТАЕМОСТЬ** Все атрибуты класса должны быть четко и однозначно выражены в коде. Разместите их под заголовком определения класса.

8.1.5. Обсуждение

Метод `__init__` почти всегда реализуется в пользовательских классах. Метод `__init__` должен включать все атрибуты экземпляра, чтобы читателю кода не приходилось гадать, какие атрибуты содержат экземпляры. Кроме того, метод `__init__` следует размещать перед любыми другими методами в теле класса. Почему? С точки зрения удобочитаемости читатель кода должен понимать, какие данные может содержать класс; атрибуты экземпляра представляют данные, хранящиеся в классе. Определение метода `__init__` — первое, с чем следует работать в пользовательском классе.

8.1.6. Задача

Леа изучает программирование на Python и работает над учебным проектом — приложением для управления задачами (таск-менеджером). Ей пришло в голову дать пользователю возможность задавать теги при создании экземпляров. Соответственно, ей понадобилось добавить аргумент `tags` в метод `__init__` (см. листинг 8.3). Она предполагает, что в большинстве случаев в аргументе `tags` будет передаваться пустой список. Какое значение по умолчанию ей следует назначить для `tags` в этом случае?

> **ПОДСКАЗКА** По сути `__init__` является функцией. Как упоминалось в разделе 6.1, значение по умолчанию может задаваться для изменяемых аргументов функций.

8.2. КАК ОПРЕДЕЛЯЮТСЯ МЕТОДЫ ЭКЗЕМПЛЯРОВ, СТАТИЧЕСКИЕ МЕТОДЫ И МЕТОДЫ КЛАССОВ

После того как вы задали необходимые атрибуты для экземпляров, пришло время определить функциональность класса. В листинге 8.3 класс содержит две функции: `complete` и `add_tag`. Эти функции называются *методами экземпляров*. Кроме методов экземпляров, определяются также статические методы и методы класса. Эти категории методов предназначены для разных сценариев использования. В данном разделе описаны ситуации, в которых следует определять методы экземпляров, статические методы и методы классов.

8.2.1. Определение методов экземпляров для выполнения операций с отдельными экземплярами

Метод экземпляра вызывается для конкретного экземпляра класса. Соответственно, когда вы хотите изменить данные отдельного экземпляра или выполнить операции, зависящие от данных конкретного экземпляра (атрибутов или других методов экземпляра), следует определять методы экземпляров.

НАПОМИНАНИЕ Синтаксис допускает использовать другое имя параметра для аргумента `self`. Но по общепринятым соглашениям обычно выбирается имя `self`.

Отличительным признаком метода экземпляра становится передача `self` в первом параметре. В разделе 8.1.1 подробно рассматривалось, что `self` обозначает экземпляр в методе `__init__`; это относится ко всем методам экземпляров. В листинге 8.4 мы проверяем, что аргумент `self` в методах экземпляров также обозначает текущий экземпляр, для чего используется простая модификация метода `complete` класса `Task` из листинга 8.3. Обратите внимание: для экономии места я не привожу другие подробности реализации класса `Task`, например `__init__`.

Листинг 8.4. Создание и использование метода экземпляра

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
        self._status = "created"

    def complete(self):
        print(f"Memory Address (self): {id(self)}")
        self.status = "completed"

task = Task("Laundry", "Wash clothes", 3)
task.complete()
# Вывод: Memory Address (self): 140508514865536

task_id = f"Memory Address (task): {id(task)}"
print(task_id)
# Вывод: Memory Address (task): 140508514865536
```

Как видите, `self` в методе `complete` соответствует такой же адрес памяти, как у экземпляра `task`, что указывает, что `self` действительно является экземпляром, для которого вызывается метод. Во внутренней реализации метод экземпляра вызывается классом, который вызывает метод и передает экземпляр в аргументе, как показано на рис. 8.3.

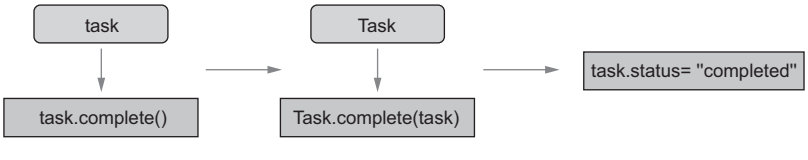


Рис. 8.3. Действия, выполняемые при вызове метода экземпляра. Когда вы используете экземпляр для вызова метода экземпляра, вызов осуществляется классом, а экземпляр передается в аргументе. В итоге функция применяется к экземпляру, для которого был вызван метод

Метод экземпляра всегда вызывается только для конкретного экземпляра. Следовательно, вызовы методов экземпляров всегда записываются по схеме: `экземпляр.метод_экземпляра(арг0, арг1, арг2)`.

В теле метода экземпляра все операции должны быть связаны с экземпляром, для которого был вызван метод. Поэтому если выясняется, что метод не работает с экземпляром или не зависит от данных, относящихся к экземплярам, весьма вероятно, что его изначально не следовало реализовывать в виде метода экземпляра. Вместо этого стоило выбрать реализацию метода как статического.

8.2.2. Определение статических методов для вспомогательной функциональности

Если вы реализуете вспомогательные функции, не относящиеся ни к какому конкретному экземпляру, для них следует определить статический метод. В этом разделе вы узнаете, как определяются статические методы.

В отличие от методов экземпляров, получающих `self` в первом параметре, статические методы не используют `self`, так как эти методы не должны быть связаны ни с каким экземпляром, и указывать конкретный экземпляр не нужно. Чтобы определить статический метод, примените к функции в теле класса декоратор `staticmethod`. Возьмем пример из листинга 8.5.

> **НАПОМИНАНИЕ** Декораторы добавляют новую функциональность к декорируемой функции без изменения ее исходной функциональности.

Листинг 8.5. Создание статического метода

```
from datetime import datetime

class Task:
    @staticmethod
    def get_timestamp():
        now = datetime.now()
        timestamp = now.strftime("%b %d %Y, %H:%M")
        return timestamp
```

В листинге 8.5 `get_timestamp` — статический метод, определенный с декоратором `@staticmethod`. В этом статическом методе создается отформатированная строка с временной меткой, которая может использоваться в любом месте, где пользователю потребуется вывести точное время. Для вызова этого метода применяется следующая схема: `ПользовательскийКласс.статический_метод(арг0, арг1, арг2)`. Опробуем эту схему на статическом методе `get_timestamp`:

```
refresh_time = f"Data Refreshed: {Task.get_timestamp()}"

print(refresh_time)
# Вывод: Data Refreshed: Mar 04 2022, 15:43
```

Как видно из листинга, статический метод для получения текущей временной метки в нужном формате вызывается в форме `Task.get_timestamp()`. Эта операция представляет обобщенную вспомогательную операцию; нетрудно вообразить разные ситуации, в которых возникает необходимость в выводе временной метки. Как правило, статистические методы создаются для предоставления вспомогательной функциональности. Иначе говоря, когда вам нужно определить вспомогательные методы, не зависящие от конкретного экземпляра, используйте декоратор `@staticmethod` для создания статического метода. Если вы читаете код чужого пользовательского класса и видите в нем `@staticmethod`, вы знаете, что перед вами статический метод, так как декоратор `staticmethod` является отличительным признаком любого статического метода.

8.2.3. Определение методов класса для обращения к атрибутам уровня класса

В разделе 8.2.2 вы научились определять статические методы — вспомогательные методы, которым не нужно обращаться к конкретному экземпляру. При этом существуют методы, которым может понадобиться доступ к атрибутам класса. В таком случае необходимо определить метод класса.

Первым отличительным признаком метода класса является передача `cls` в первом параметре. Как и `self` у методов экземпляра, `cls` не является ключевым словом, и этому аргументу присваиваются другие допустимые имена. Тем не менее по общепринятым соглашениям используется имя `cls`, и каждый Python-программист должен соблюдать это соглашение.

> **УДОБОЧИТАЕМОСТЬ** Первому параметру метода класса должно присваиваться имя `cls`. Когда другие программисты видят `cls`, они знают, что это обозначение класса.

Реализация статических методов требует декоратора `staticmethod`. Метод класса также использует декоратор `classmethod` — второй отличительный признак метода класса.

Метод называется *методом класса*, потому что ему необходим доступ к атрибутам класса или его методам. Рассмотрим пример. Допустим, в таск-менеджере мы получаем данные в форме объекта `dict`, содержащего информацию о задаче:

```
task_dict = {"title": "Laundry", "desc": "Wash clothes", "urgency": 3}
```

Конструирование экземпляра класса `Task` на основе объекта `dict` могло бы выполняться так:

```
task = Task(task_dict["title"], task_dict["desc"], task_dict["urgency"])
```

Но поскольку в приложении часто возникает необходимость в получении данных `dict` и создании соответствующего экземпляра `Task`, следует предоставить

более удобный способ решения этой задачи. Хорошим решением оказывается метод класса, как показано в листинге 8.6.

Листинг 8.6. Создание метода класса

```
class Task:
     def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
        self._status = "created"

    @classmethod
    def task_from_dict(cls, task_dict):
        title = task_dict["title"]
        desc = task_dict["desc"]
        urgency = task_dict["urgency"]
        task_obj = cls(title, desc, urgency)
        return task_obj
```

Как видно из листинга 8.6, мы определяем метод класса с именем `task_from_dict` с помощью декоратора `@classmethod`. В теле класса имя `cls` обозначает класс, с которым вы работаете (`Task`), что позволяет напрямую использовать конструктор класса — `cls(title, desc, urgency)` — для создания объекта экземпляра. С таким методом класса удобно создавать экземпляр `Task` на основе объекта `dict`:

```
task = Task.task_from_dict(task_dict)
print(task.__dict__)
# Вывод: {'title': 'Laundry', 'desc': 'Wash clothes',
➥ 'urgency': 3, 'status': 'created', 'tags': []}
```

Вообще говоря, метод класса используется в основном как *фабричный метод* (factory method); так называется разновидность методов, используемых для создания экземпляров конкретной формы данных. В разделе 4.5 упоминалось о том, что класс `DataFrame` из библиотеки pandas представляет собой модель данных типа электронной таблицы. Он содержит пару методов класса (`from_dict` и `from_records`), которые используются для конструирования экземпляров класса `DataFrame`.

8.2.4. Обсуждение

Из трех разновидностей методов методы экземпляров и методы класса наиболее просты. Со статическими методами все немного сложнее. Так как эти методы предназначены для предоставления вспомогательной функциональности, обычно их уместно определять вне классов, ведь они не предназначены для работы с конкретным экземпляром или классом. В общем случае я рекомендую размещать статические методы вне классов, если они обеспечивают более общую функциональность, выходящую за рамки классов. Для примера возьмем библиотеку обработки данных pandas, в которой основными моделями данных

являются классы `Series` и `DataFrame`. Вспомогательная функция `to_datetime` преобразует данные в объект даты. Эта функция решает более общую задачу; по этой причине она не реализуется в виде статического метода внутри класса `Series` или `DataFrame`.

8.2.5. Задача

Продолжая работать над таск-менеджером, Леа осознает, что ей нужно создать экземпляр класса `Task` на основе объекта `tuple`: `("Laundry", "Wash clothes", 3)`. Какую разновидность метода ей следует реализовать, чтобы обеспечить эту возможность в классе?

> **ПОДСКАЗКА** В листинге 8.6 приведена реализация метода, который создает экземпляр на основе объекта `dict`.

8.3. КАК ОРГАНИЗОВАТЬ БОЛЕЕ ТОЧНОЕ УПРАВЛЕНИЕ ДОСТУПОМ В КЛАССЕ

В пользовательском классе могут определяться десятки методов. Некоторые методы являются внутренними — для разработчика класса, тогда как другие предназначены для разработчиков, использующих ваш класс. Рассмотрим следующий сценарий: в классе `Task` другой метод форматирует атрибут `note` для метода `complete`:

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
        self._status = "created"
        self.note = ""

    def complete(self, note = ""):
        self.status = "completed"
        self.note = self.format_note(note)


    def format_note(self, note):
        formatted_note = note.title()
        return formatted_note
```

Когда пользователь вызывает метод `complete`, этот метод присваивает атрибуту `note` отформатированный текст заметки, для чего используется вызов `format_note`. Заметим, что пользователь также может вызвать `format_note` напрямую. Такое поведение нежелательно, так как одним из ключевых принципов ООП является инкапсуляция: доступ предоставляется только к тем атрибутам

и методам, которые должны быть доступными для пользователя, и не более того. К требованиям инкапсуляции относится возможность более точного управления доступом к классу. В этом разделе рассматриваются основные средства управления доступом.

ОСНОВНЫЕ ПОНЯТИЯ Термином «*инкапсуляция*» (encapsulation) называется принцип программирования, широко применяемый в ООП-языках. В соответствии с этим принципом данные и методы группируются в классе, а доступ предоставляется только к той части данных, которая важна для пользователей.

public, protected и private

В типичном ООП-языке для ограничения доступа к некоторому атрибуту или методу используются ключевые слова `protected` или `private`. Противоположностью `protected` или `private` является `public`; оно означает, что атрибуты и методы доступны для всех пользователей как внутри класса, так и за его пределами (являются публичными). Ключевое слово `protected` (защищенный) означает, что атрибуты и методы доступны для класса и его подклассов, но не за пределами класса. Ключевое слово `private` означает, что атрибуты и методы доступны только для самого класса, но не для его подклассов или пользователей за пределами класса (приватные). Из-за ограничения доступа к внутренней реализации `protected` или `private` также называют непубличными (non-public) атрибутами и методами.

8.3.1. Создание защищенных методов с префиксом _

По своей сути Python является ООП-языком. Однако в отличие от других ООП-языков, использующих `private` и/или `protected` для управления доступом, в Python нет формального механизма, который бы ограничивал доступ к атрибуту или методу. Другими словами, все компоненты класса общедоступны, и в Python нет ключевых слов `protected` или `private`. По общепринятым соглашениям механизм управления доступом основан на использовании символов подчеркивания (`_`) в качестве префиксов атрибута или метода. Префикс из одного символа подчеркивания соответствует защищенному (`protected`) уровню, а двойной символ подчеркивания соответствует приватному (`private`) уровню (раздел 8.3.2). В этом разделе рассматривается определение защищенных методов. Заметим, что для создания защищенных и приватных атрибутов используется один и тот же механизм.

При описании именованных кортежей в разделе 3.3 я упоминал о том, что создание модели данных именованных кортежей позволяет задействовать

функциональность автозаполнения в интегрированной среде разработки (IDE) и получать список доступных атрибутов после ввода точки за именем объекта. Однако этот инструмент создаст вам неудобства, если в список будут включаться имена методов, которые не предназначены для вашего использования. Пользователь не должен вызывать метод `format_note` самостоятельно; соответственно, в идеале список автозаполнения не должен включать `format_note` (рис. 8.4).

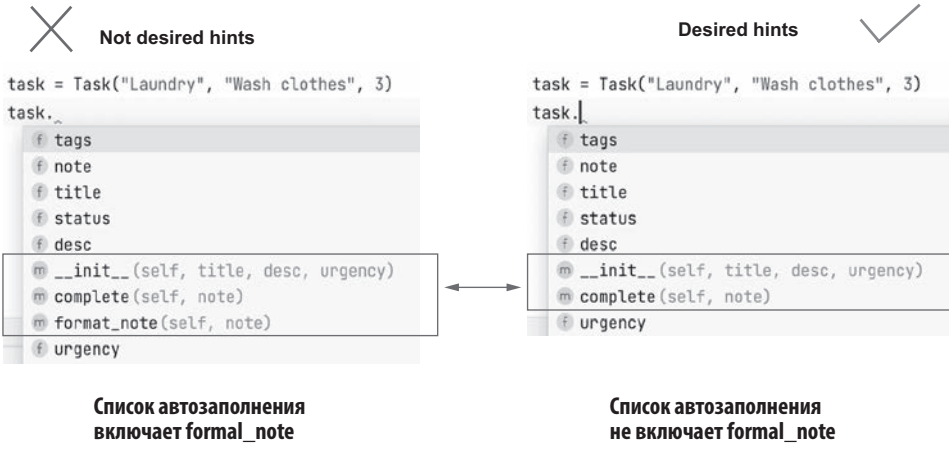


Рис. 8.4. Разные подсказки автозаполнения, выводимые для экземпляра. Список автозаполнения не должен включать методы, которые не предназначены для вызова пользователем, — в данном случае это метод format_note

Очевидно, исключение функций, которые не должны входить в список рекомендаций автозаполнения, повышает эффективность программирования. Но как IDE узнает, какие функции следует скрыть? Все дело в префиксе _ перед именем метода — он показывает, что метод является защищенным. Вместо `format_note` методу следует присвоить имя `_format_note`. Префиксы _ важны в двух отношениях:

- Метод не предназначен для использования за пределами класса, поэтому он не включается в подсказки автозаполнения во внешнем коде (правая половина рис. 8.4).
- Метод остается в подсказках автозаполнения при использовании внутри класса (рис. 8.5).

Эти два аспекта согласуются с принципом инкапсуляции. Вы ограничиваете доступ пользователям к функциям, которые им не нужны, и сохраняете доступ к тем же функциям для пользователей, которые с ними работают.

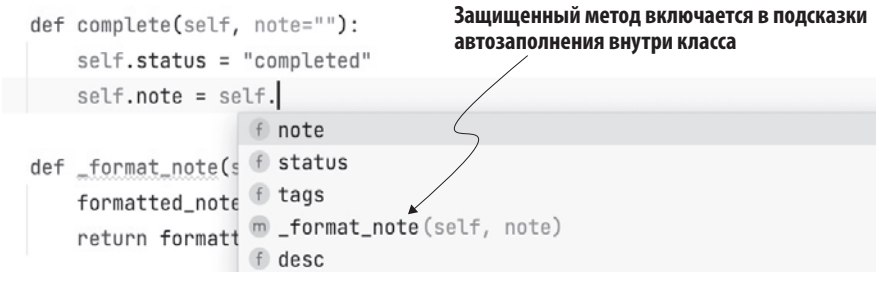


Рис. 8.5. Доступность защищенного метода внутри класса. После ввода точки в списке выводятся доступные атрибуты и методы для объекта экземпляра, и список включает защищенный метод

8.3.2. Создание приватных методов с префиксом __

В разделе 8.3.1 вы научились использовать защищенные методы для ограничения всеобщего доступа к методам, которые пользователь видеть не должен. Кроме защищенных методов, также можно определять приватные методы, которые обеспечивают аналогичный эффект инкапсуляции. В этом разделе вы научитесь определять приватные методы. Более того, вы увидите, почему иногда бывает полезно определять приватные методы вместо защищенных.

Вы уже знаете, что для определения приватного метода используется префикс из двух символов _. Продолжим использовать метод `format_note` в качестве примера. Чтобы сделать метод приватным, переименуйте его в `__format_note`. С переименованием метода доступ к нему ограничивается внутренней реализацией класса (рис. 8.6).

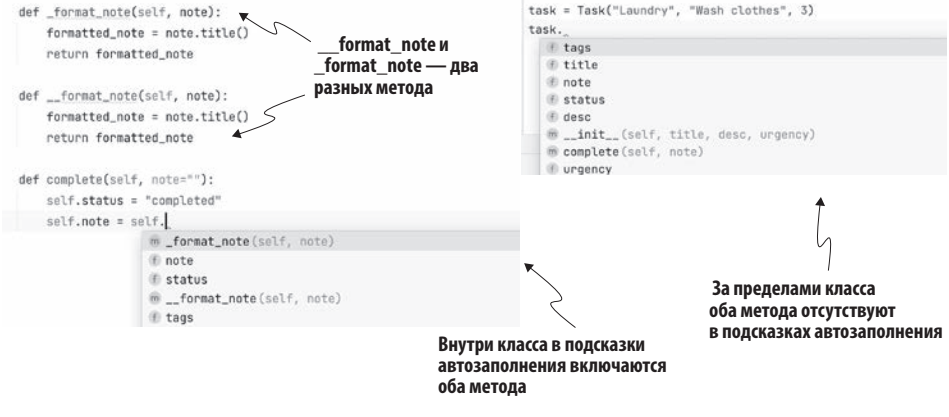


Рис. 8.6. Для приватных методов возможен только внутренний доступ. Имя метода __format_note начинается с двух символов _ — это означает, что он является приватным. Приватный метод доступен только в пределах класса

Защищенные и приватные методы похожи с точки зрения их доступности внутри класса. Но как упоминалось в начале раздела 8.3.1, в Python нет строгих ограничений доступа. Если вы захотите обратиться к защищенным методам, это можно будет сделать, хотя многие IDE выведут предупреждение, как показано на рис. 8.7.

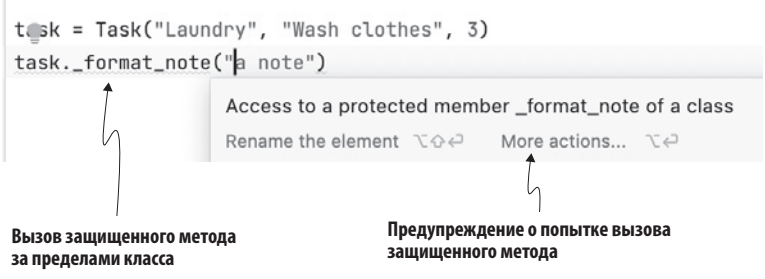


Рис. 8.7. Вызов защищенного метода внутри класса технически возможен. Но в ответ на это нежелательное поведение IDE выдает предупреждение, так как защищенные методы не предназначены для внешнего использования

Что происходит, когда кто-то пытается вызвать приватный метод за пределами класса? Нечто странное на первый взгляд. Как видно из следующего фрагмента кода, такой метод или атрибут не существует:

```
task.__format_note("a note")
# ERROR: AttributeError: 'Task' object has no attribute '__format_note'.
```

«Недоступность» `__format_note` за пределами класса демонстрирует принципиальное различие между приватными и защищенными методами, так как для этого метода действуют более жесткие ограничения, чем для защищенных методов (как `_format_note`). Следовательно, если вы хотите ограничить доступ к методам, используйте префикс из двух символов _ для создания приватных методов вместо одного символа _ для защищенных методов.

> **СОПРОВОЖДАЕМОСТЬ** Используйте приватные методы, если вам нужно установить более жесткие ограничения.

Я уже говорил, что в Python нет настоящих ограниченных методов, именно поэтому «недоступность» из предыдущего абзаца заключена в кавычки. Вопрос в том, как в случае необходимости все же обратиться к приватному методу. Допустим, вам потребовалось модифицировать код в пакете, разработанном другими авторами. Как показано в следующем фрагменте, к приватному методу можно обратиться вызовом `_Task__format_note("a note")`:

```
task._Task__format_note("a note")
# Вывод: 'A Note'
```

Такой механизм, называемый *искажением имен* (name mangling), преобразует приватный метод в метод с другим именем, что позволяет вызвать приватный метод за пределами класса. Точнее говоря, искажение имен происходит по схеме `__приватный_метод -> _ИмяКласса__приватный_метод`. Таким образом, `__format_note` превращается `_Task__format_note`, и этот приватный метод вызывается за пределами класса `Task`.

> **ОСНОВНЫЕ ПОНЯТИЯ** *Искажение имен* представляет собой схему преобразования имени приватного метода посредством добавления префикса `_ИмяКласса`. Эта схема позволяет обратиться к приватному методу за пределами класса.

Кроме разных правил, касающихся внешнего доступа, для защищенных и приватных методов также действуют разные правила обращения из подклассов. Эта тема рассматривается в разделе 8.5.

8.3.3. Создание атрибутов, доступных только для чтения (read-only)

Одна из главных причин создания пользовательских классов заключается в том, что в них можно определить любое количество атрибутов. В пользовательском классе как в единой сущности все взаимосвязанные данные группируются через четко определенные атрибуты и методы. Примечательно, что пользовательский класс является *изменяемым*, что подразумевает возможность изменения его атрибутов. Однако иногда оказывается, что изменение некоторых атрибутов вашего класса нежелательно. В таком случае следует воспользоваться другим механизмом управления доступом: *атрибутами*, *доступными только для чтения*. Пользователь может прочитать значение такого атрибута, но не изменить его. В этом разделе вы научитесь определять атрибуты, доступные только для чтения.

Возьмем атрибут `status` класса `Task`. В текущей версии пользователи способны свободно изменять атрибут `status` экземпляра:

```
print(task.status)
# Вывод: created

task.status = "completed"
print(task.status)
# Вывод: completed
```

Из-за требований инкапсуляции пользователь не должен изменять атрибут `status` по своему усмотрению. Например, чтобы обновить статус задачи и пометить ее как завершенную, он должен вызвать метод `complete`. Вопрос состоит в том, как помешать пользователю вручную задать значение `status`. Задача решается использованием декоратора `property`, как показано в листинге 8.7.

Листинг 8.7. Использование декоратора property

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
        self._status = "created"

    @property
    def status(self):
        return self._status

    def complete(self):
        self._status = "completed"
```

В листинге 8.7 приведен лишь код, относящийся к определению атрибутов, доступных только для чтения. В этом коде следует обратить внимание на три важных аспекта:

- Экземпляр содержит защищенный атрибут `_status`.
- Мы определяем метод экземпляра `status`, к которому применяется декоратор `property`.
- В методе `complete` обновляется атрибут `_status`.

Мы знаем, что при вызове метода для объекта используется оператор вызова — круглые скобки после имени метода. Однако с декоратором `property` метод становится доступным так, как если бы он был атрибутом. Для простоты мы будем называть метод с декоратором `property` *свойством*, причем для обращения к свойству оператор вызова не нужен:

```
task = Task("Laundry", "Wash clothes", 3)

print(task.status)
# Вывод: created
```

Свойство представляет атрибут, доступный только для чтения. Его можно прочитать, как показано в предшествующем фрагменте. Однако присвоить ему новое значение не удастся, а это именно то, что нам нужно: запретить пользователям задавать значение `status` напрямую, как показано в листинге 8.8.

Листинг 8.8. Свойство, доступное только для чтения

```
>>> task.status = "completed"
# ERROR: AttributeError: can't set attribute 'status'
```

СОПРОВОЖДАЕМОСТЬ Определение свойств, доступных только для чтения, не дает пользователям возможности изменять атрибуты и улучшает стабильность данных.

В более общем сценарии при определении доступного только для чтения свойства часто приходится создавать защищенные атрибуты, предназначенные для внутренних операций с соответствующими данными. Например, свойство `status` доступно только для чтения, а свойство `_status` может использоваться для работы с определяющими статус задачи данными внутри класса.

> **ВОПРОС** В какой ситуации возникает необходимость в использовании защищенного атрибута вместо приватного? Подумайте, чем различаются эти уровни в контексте доступа из подкласса.

8.3.4. Проверка целостности данных с использованием set-метода (сеттера)

В разделе 8.3.3 был представлен декоратор `property`, который используется для создания доступного только для чтения свойства `status` в классе `Task`. Однако определяя свойство, доступное только для чтения, вы не сможете присвоить ему значение. Такое поведение не всегда желательно. Иногда нужно иметь специальный механизм для присваивания значения свойству. В частности, такой механизм используется для проверки целостности данных, о чем будет рассказано в этом разделе.

> **ОСНОВНЫЕ ПОНЯТИЯ** В традиционных ООП-языках, таких как Java, со свойствами связываются get-методы и set-методы. *Get-метод, или геттер,* предназначен для получения значения свойства, а *set-метод, или сеттер,* используется для присваивания значения свойству. Декоратор `property` создает get-метод, а в оставшейся части этого раздела мы займемся созданием set-метода.

Предположим, вы разрешили пользователям задавать свойство `status` напрямую. Значение, используемое для присваивания, должно быть действительным. Допустим, задача может находиться в одном из нескольких заранее определенных состояний: созданная, начатая, завершенная или приостановленная. Как проследить за тем, чтобы для присваивания использовались только эти варианты, а попытки присваивания любых других значений отклонялись? Такие проверки целостности данных лучше всего реализуются сеттерами свойств, как показано в листинге 8.9.

Листинг 8.9. Создание сеттера для свойства

```
class Task:
    # Метод __init__ остается без изменений

    @property
    def status(self):
        return self._status
```

```
    @status.setter
    def status(self, value):
        allowed_values = ["created", "started", "completed", "suspended"]
        if value in allowed_values:
            self._status = value
            print(f"task status set to {value}")
        else:
            print(f"invalid status: {value}")
```

В таких случаях рекомендуется выдавать исключение (раздел 12.4)

В листинге 8.9 после создания свойства `status` мы определяем сеттер для этого свойства с помощью декоратора `@status.setter`, который строится по общей схеме `@имя_свойства.setter`. Сеттер представляет собой метод экземпляра; он получает аргумент `value` со значением, которое должно быть присвоено свойству. В теле сеттера мы проверяем, что присваиваемое значение совпадает с одним из четырех допустимых вариантов, и если проверка проходит успешно, задаем свойство `status`:

```
task = Task("Laundry", "Wash clothes", 3)
task.status = "completed"
# Вывод: task status set to completed

task.status = "random"
# Вывод: invalid status: random
```

Как видите, свойству `status` напрямую задается значение `completed`. Важно, что при попытке задать недопустимое значение вы получите оповещение об ошибке. И хотя есть возможность создавать get- и set-методы для преобразования атрибутов в свойства, лучше этого не делать, потому что они усложняют класс. Если вы не реализуете свойства ради доступности только для чтения или проверки данных, лучше обращайтесь к атрибутам и задавайте их значения напрямую вместо выполнения операций через свойства. Схема с прямым доступом отличает Python от других ООП-языков, делая его код более компактным.

8.3.5. Обсуждение

Определение приватных и защищенных методов — важнейший механизм для реализации инкапсуляции в классах, оно помогает минимизировать открытые атрибуты класса. Работающий с классом пользователь будет получать подсказки автозаполнения для общедоступных атрибутов, что сделает его деятельность более эффективной. Не пытайтесь инкапсулировать все подряд, создавая set- и get-методы, как это делается в некоторых других ООП-языках, — такая практика не является питонической. В большинстве случаев следует использовать прямые обращения и присваивание атрибутов вместо свойств, потому что этот способ более прямолинеен и требует меньшего объема кода реализации.

8.3.6. Задача

Допустим, атрибут `urgency` должен содержать целое значение от 1 до 5. Сможете ли вы преобразовать его в свойство при помощи сеттера? Сеттер позволяет проверить присваиваемое значение.

> **ПОДСКАЗКА** Вы можете использовать защищенный атрибут (например, `_urgency`) для внутреннего представления данных `urgency` и создать свойство с именем `urgency`.

8.4. КАК НАСТРОИТЬ СТРОКОВОЕ ПРЕДСТАВЛЕНИЕ КЛАССА

В разделе 8.1 изучался метод инициализации `__init__`. Методы, имена которых заключаются в двойные символы `_`, называются *специальными методами* (special methods). Специальные методы выполняют специальные операции, например, метод `__init__` вызывается при создании экземпляра конструктором. Когда вы реализуете специальный метод в классе, это можно назвать переопределением метода, потому что все классы Python являются подклассами класса `object`, в котором реализуются эти специальные методы.

> **ОСНОВНЫЕ ПОНЯТИЯ** В ООП-языках *переопределением* называется предоставление подклассом новых реализаций для методов, определяемых в родительском классе.

В этом разделе мы рассмотрим два других специальных метода: `__str__` и `__repr__`, предоставляющих специализированные строковые представления для класса.

8.4.1. Переопределение __str__ для вывода содержательной информации об экземпляре

Нередко возникает необходимость в анализе данных экземпляров, с которыми вы работаете. При этом часто используется функция `print`, выводящая строковое представление объекта. Например, следующий фрагмент показывает, как выглядит экземпляр класса `Task`:

```
print(task)
# Вывод: <__main__.Task object at 0x7f9f280d6800>
```

В выводимую информацию включены класс экземпляра и его адрес в памяти, и ничего более. Иначе говоря, мы не видим никакой содержательной информации об экземпляре, в том числе значений его атрибутов. В этом разделе вы

узнаете, как выводить более содержательную информацию об экземпляре при помощи функции `print`.

Когда вы используете `print` с экземпляром пользовательского класса, вызывается специальный метод `__str__`, который определяет строковое представление экземпляра. Чтобы обеспечить специализированное строковое представление, отличное от приведенного выше, мы переопределим `__str__` в классе `Task`, как показано в листинге 8.10.

Листинг 8.10. Переопределение __str__ в классе

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency

    def __str__(self):
        return f"{self.title}: {self.desc}, urgency level {self.urgency}"
```

При переопределении `__str__` в классе следует обратить внимание на три обстоятельства:

- Это метод экземпляра, так как он должен предоставить строковое представление конкретного экземпляра.
- Его возвращаемым значением должен быть объект `str`.
- Возвращаемая строка должна предоставлять содержательную информацию для создания экземпляра. В нашем примере в ней необходимо присутствие ключевых атрибутов экземпляра, включая `title`, `desc` и `urgency`.

Мы видим, как выглядит результат вызова `print` после переопределения метода `__str__`:

```
task = Task("Laundry", "Wash clothes", 3)
print(task)
# Вывод: Laundry: Wash clothes, urgency level 3
```

Помимо `print`, для подготовки выводимых на экран строковых данных часто применяются f-строки. Когда вы включаете в строку экземпляр в фигурных скобках, при интерполяции экземпляра во внутренней реализации вызывается метод `__str__`. Посмотрите, как это происходит:

```
planned_task = f"Next Task - {task}"
print(planned_task)
# Вывод: Next Task - Laundry: Wash clothes, urgency level 3
```

Если вы захотите вызвать метод `__str__` для экземпляра явно, рекомендуется использовать форму `str(instance)`, хотя также можно вызвать `Class.__str__(instance)` напрямую:

```
str(task)
# Вывод: Laundry: Wash clothes, urgency level 3
```

8.4.2. Переопределение __repr__ для получения информации экземпляра

Многие разработчики предпочитают работать с Python в интерактивной консоли (особенно во время изучения языка), потому что в консоли предоставляется вывод кода в реальном времени. Если ввести в консоли имя переменной `str`, вы увидите ее строковое значение:

```
>>> planned_task
'Next Task - Laundry: Wash clothes, urgency level 3'
```

Но если вы попытаетесь сделать то же с экземпляром `task`, результат будет иным:

```
>>> task
<__main__.Task object at 0x7f9f280d6f80>
```

Мы уже реализовали метод `__str__`, который не изменяет информацию, выводимую для экземпляра в интерактивной консоли. В этом разделе вы узнаете, как изменить строковое представление, выводимое в консоли.

Когда в интерактивной консоли выводится строковое представление для экземпляра, вызывается специальный метод `__repr__`. Сначала я покажу, как реализовать `__repr__` в классе (листинг 8.11), и объясню некоторые ключевые обстоятельства:

- Это метод экземпляра, так как он предоставляет информацию о строковом представлении на уровне конкретного экземпляра.
- Он возвращает строковое значение.
- Строка должна предоставлять информацию для создания экземпляра, а именно: если другие пользователи введут строку как код, то для нее должен быть создан экземпляр с такими же атрибутами, как у текущего экземпляра.

Листинг 8.11. Переопределение __repr__ в классе

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
    def __str__(self):
        return f"{self.title}: {self.desc}, urgency level {self.urgency}"
    def __repr__(self):
        return f"Task({self.title!r}, {self.desc!r}, {self.urgency})"
```

!r требует использовать метод __repr__ для строковой интерполяции

После реализации `__repr__` вы сможете просмотреть экземпляр класса `Task` в интерактивной консоли Python:

```
>>> task = Task("Laundry", "Wash clothes", 3)
>>> task
Task('Laundry', 'Wash clothes', 3)
```

Чтобы вызвать `__repr__` для экземпляра, используйте `repr(instance)` вместо `Class.__repr__(instance)`:

```
repr(task)
# Вывод: Task('Laundry', 'Wash clothes', 3)
```

8.4.3. Различия между __str__ и __repr__

В разделах 8.4.1 и 8.4.2 вы узнали о методах `__str__` и `__repr__`. Оба метода формируют строковое представление экземпляров пользовательского класса. В этом разделе вы узнаете, чем различаются эти два метода.

Разные цели

Первое (и самое серьезное) различие состоит в том, что методы служат разным целям. Строка, предоставляемая `__repr__`, предназначена для отладки и разработки, поэтому она должна использоваться разработчиками. И разработчику нужна возможность сконструировать экземпляр непосредственно по строке. Как упоминалось в разделе 2.2, встроенная функция `eval` используется для вычисления строкового литерала и конструирования представленного объекта. То же самое можно сделать с помощью `__repr__`:

```
task = Task("Laundry", "Wash clothes", 3)

task_repr = repr(task)

task_repr_eval = eval(task_repr)

print(type(task_repr_eval))
# Вывод: <class '__main__.Task'>

print(task_repr_eval)
# Вывод: Laundry: Wash clothes, urgency level 3
```

С другой стороны, строка, предоставляемая `__str__`, должна содержать описательную информацию, и предназначена она для рядовых пользователей кода. Соответственно, эта строка менее формальна, чем строка, формируемая `__repr__`, которая содержит информацию для создания экземпляра.

Разные применения

Хотя оба метода предоставляют строковое представление класса, метод `__str__` используется как функцией `print`, так и интерполяцией в f-строках. А вот метод

`__repr__` используется тогда, когда вы хотите просмотреть информацию экземпляра в интерактивной консоли.

В листинге 8.11 видно, что к интерполяции `self.title` присоединяется суффикс `!r` — так называемый *флаг преобразования* (conversion flag), который требует, чтобы интерполированная строка объекта вызывала `__repr__` вместо `__str__` для создания строковых представлений. По умолчанию для интерполяции экземпляра пользовательского класса используется строка, созданная вызовом `__str__`. Чтобы переопределить поведение по умолчанию, укажите флаг преобразования после экземпляра: `f"{instance!r}"`. По умолчанию для экземпляров используется флаг преобразования `!s`, который использует строку, созданную вызовом `__str__`. Другими словами, выражения `f"{instance}"` и `f"{instance!s}"` эквивалентны.

Вероятно, вас интересует, почему флаг `!r` необходимо использовать для `title` и `desc`, но не для `urgency`. Дело в том, что `title` и `desc` являются объектами `str`. Их строковые представления из `__str__` не заключаются в кавычки. Если использовать интерполяцию по умолчанию, то полученная строка не будет применяться для конструирования экземпляра, как показывает следующий пример:

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency

    def __str__(self):
        return f"{self.title}: {self.desc}, urgency level {self.urgency}"

    def __repr__(self):
        return f"Task({self.title}, {self.desc}, {self.urgency})"

task = Task("Laundry", "Wash clothes", 3)

print(repr(task))
# Вывод: Task(Laundry, Wash clothes, 3)
```

В измененной версии класса флаг преобразования `!r` для `title` и `desc` опущен. Из вывода видно, что `Laundry` и `Wash clothes` не заключены в кавычки. Как и следовало ожидать, создать экземпляр `Task` на основе этой строки не удастся:

```
eval(repr(task))
# ERROR: SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Строковое представление из `__repr__`, наоборот, содержит кавычки, необходимые для строковых литералов, — например, `"Laundry"` вместо `Laundry`. Первый вариант является действительным объектом `str`, а второй нет (он будет интерпретирован как переменная с именем `Laundry`, но не будет использоваться, потому что переменная с именем `Laundry` не определена).

8.4.4. Обсуждение

Основное предназначение метода `__repr__` — однозначно показывать, что собой представляет объект. Строка, генерируемая методом `repr` (обратите внимание: при вызове `repr` вызывается метод `__repr__` из класса), должна содержать текст, который может использоваться для воссоздания аналогичного объекта. По этой причине все строки, генерируемые `repr`, должны содержать кавычки, чтобы они рассматривались как действительные строковые литералы Python. Не забывайте включать флаг преобразования `!r` при использовании f-строк. Я рекомендую реализовать для пользовательских классов оба метода, `__str__` и `__repr__`. Если вы хотите ограничиться только одним методом, переопределяйте `__repr__`, потому что если `__str__` не реализован, Python использует `__repr__`.

8.4.5. Задача

В классе `Task` метод `__repr__` возвращает результат `f"Task({self.title!r}, {self.desc!r}, {self.urgency})"`. В этой f-строке жестко зафиксировано имя класса `Task`. Общий принцип программирования требует свести к минимуму фиксацию данных в коде. Как получить имя класса на программном уровне?

> **ПОДСКАЗКА** Экземпляр содержит специальный атрибут `__class__`, определяющий его класс, а у класса имеется специальный атрибут `__name__` для получения имени класса.

8.5. ДЛЯ ЧЕГО И КАК СОЗДАЮТСЯ НАДКЛАССЫ И ПОДКЛАССЫ

Одной из важнейших концепций ООП является *наследование*. В общем случае этим термином обозначается процесс создания производного (дочернего) класса, который может повторно использовать реализации родительского класса (или их часть). В производный класс можно включить специализированные реализации, которые решают некоторые задачи более эффективно, чем реализации родительского класса. Производный класс также называется *подклассом* (subclass), а родительский класс также называется *надклассом* (superclass).

> **ОБРАТИТЕ ВНИМАНИЕ** Роли подклассов и надклассов относительны. Подкласс является надклассом для своих подклассов.

Создание подклассов — тема более сложная, чем многие из рассмотренных ранее. Как вы узнаете в этом разделе, управлять надклассом с несколькими подклассами не так просто, как несколькими несвязанными классами. Поэтому прежде, чем браться за реализацию подклассов, нужно убедиться в том, что их применение оправданно. В этом разделе вы узнаете, чем обосновывается использование

подклассов. Кроме того, мы рассмотрим технические подробности реализации подкласса.

8.5.1. Когда уместно использовать подклассы

С расширением проекта вам придется определять больше классов для работы с растущими объемами данных. На текущей стадии классы еще не связаны отношениями наследования. Однако вы замечаете, что некоторые классы похожи по своей функциональности; в проекте присутствует повторение кода. Припомнив принцип DRY («не повторяйтесь»), вы понимаете, что пришло время провести рефакторинг этих классов. Один из важнейших подходов к созданию подклассов основан на сокращении перекрывающихся реализаций между классами. В этом разделе вы узнаете, в каких ситуациях уместно использовать подклассы.

Сверху вниз (от надкласса к подклассам) или снизу вверх (от подклассов к надклассу)?

Возможны два типичных сценария реализации подклассов в проекте. В первом сценарии вы начинаете с одного класса, определяющего модель данных, а потом осознаете, что на базе этого подкласса необходимо создать подклассы для формирования более конкретных моделей данных. Во втором сценарии вы начинаете с нескольких классов, определяющих разные модели данных, а затем понимаете, что в функциональности этих классов много общего. И тогда вы создаете надкласс, от которого могут наследовать эти подклассы.

В проекте могут встретиться оба сценария. В этом разделе мы сосредоточимся на втором сценарии: восходящем проектировании (от подклассов к надклассу). По моему опыту, проект обычно начинается с плоской структуры модели данных — для каждой модели создаются множественные классы. После того как эти классы будут реализованы, вы замечаете между ними сходство и решаете, что нужно создать надкласс.

Допустим, таск-менеджер поддерживает регистрацию пользователей. Все пользователи делятся на две категории: руководители и подчиненные. В начале разработки приложения были созданы два разных класса, `Supervisor` и `Subordinate`, для руководителей и подчиненных соответственно. На рис. 8.8 приведена наглядная сводка атрибутов и методов этих двух классов.

Как видите, классы получились похожими, и у них много общих атрибутов и методов. В таком случае следует рассмотреть возможность создания надкласса, который предоставляет общую функциональность. Чтобы реализовать разную функциональность для каждого типа, можно создать два подкласса посредством наследования от надкласса. На рис. 8.9 представлена схема структуры наследования.

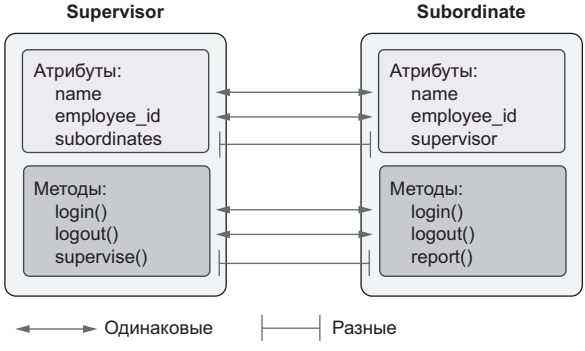


Рис. 8.8. Сходства и различия между классами `Supervisor` и `Subordinate`. В этих двух классах присутствует ряд общих атрибутов и методов; другие атрибуты и методы отличаются

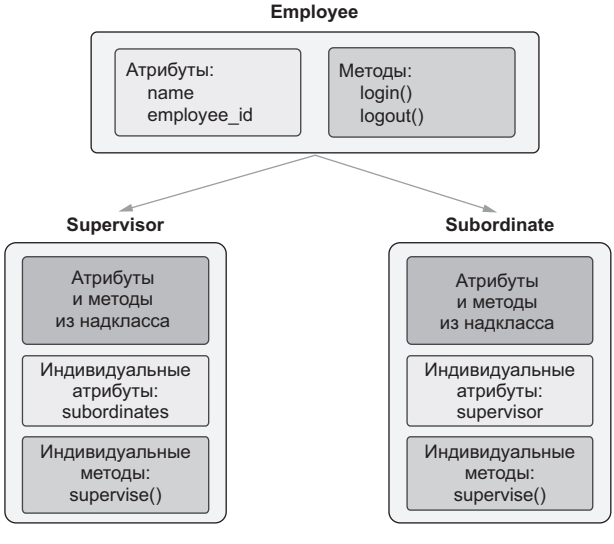


Рис. 8.9. Создание надкласса с общими атрибутами и методами. В подклассах реализуются конкретные атрибуты и методы. По умолчанию подклассы наследуют все неприватные атрибуты и методы от надкласса

Как показано на рис. 8.9, при создании надкласса все общие атрибуты и методы перемещаются из подклассов в надкласс. В подклассах реализуются специализированные атрибуты и методы. Впрочем, такие описания выглядят слишком абстрактно. В следующем разделе будет рассмотрен код реализации.

8.5.2. Автоматическое наследование атрибутов и методов надкласса

Ранее я упоминал о том, что перекрытие функциональности между классами лежит в основе создания надклассов, что способствует сокращению повторений в коде. В этом разделе вы узнаете, почему наследование сокращает объем кода.

Чтобы понять, как надкласс работает в сочетании с подклассами, продолжим рассматривать пример с иерархией `Employee`/`Supervisor`. Для начала прочитайте код из листинга 8.12. Мы не реализуем версию `__init__` в классе `Supervisor`, это будет сделано далее в разделе 8.5.6.

Листинг 8.12. Базовая структура надкласса и подклассов

```
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def login(self):
        print(f"An employee {self.name} just logged in.")

    def logout(self):
        print(f"An employee {self.name} just logged out.")

class Supervisor(Employee):
    pass
```

При определении подклассов имя надкласса указывается в круглых скобках после имени класса. В данном случае надклассом является `Employee` (сотрудник), поэтому он указывается после `Supervisor`. Заметим, что подкласс `Supervisor` автоматически наследует все от своего надкласса `Employee`, включая инициализацию и другие методы. Эта возможность продемонстрирована в следующем фрагменте:

```
supervisor = Supervisor("John", "1001")

print(supervisor.name)
# Вывод: John

supervisor.login()
# Вывод: An employee John just logged in.
```

Как видно из листинга, экземпляр создается вызовом `Supervisor("John", "1001")`. В теле класса `Supervisor` содержится только инструкция `pass`. `Supervisor` поддерживает создание экземпляров, но созданный экземпляр уже содержит атрибуты и методы, потому что класс `Supervisor` наследует от класса `Employee`.

Как правило, когда ваши подклассы содержат те же атрибуты и методы, что и надкласс, вам не нужно предоставлять никакую реализацию в подклассе, так как подкласс автоматически получает все атрибуты и методы из надкласса.

8.5.3. Переопределение методов надкласса для реализации нестандартного поведения

В разделе 8.5.2 вы узнали, что подклассы автоматически наследуют все атрибуты и методы из надкласса. Однако в некоторых случаях необходимо предоставить специализированное поведение для подкласса. В этом разделе вы научитесь переопределять методы надкласса, чтобы определить специализированные реализации в подклассах.

Полное переопределение метода

Метод надкласса можно полностью переопределить в подклассе. В отличие от некоторых ООП-языков, где переопределение обозначается ключевым словом `override`, Python позволяет определить тот же метод с реализацией, отличной от реализации из надкласса. Для примера возьмем метод `login`:

```
class Supervisor(Employee):
    def login(self):
        print(f"A supervisor {self.name} just logged in.")
```

При наличии обновленного метода `login` в подклассе мы видим, что экземпляр класса `Supervisor` вызывает метод `login` подкласса вместо реализации из надкласса:

```
supervisor = Supervisor("John", "1001")
supervisor.login()
# Вывод: A supervisor John just logged in.
```

У метода `logout` нет специализированной реализации. Как и следовало ожидать, при вызове `logout` для экземпляра будет выполнена реализация `logout` класса `Employee`. Как Python определяет, какую реализацию следует использовать в каждом конкретном случае? Ответ напрямую связан с важной концепцией *порядка разрешения методов*, или *MRO* (Method Resolution Order), определяющей порядок выбора конкретной реализации метода в иерархической структуре класса.

> **ОСНОВНЫЕ ПОНЯТИЯ** MRP определяет, как метод или атрибут экземпляра рассматривается в структуре наследования классов.

В Python поддерживается *множественное наследование* (то есть наследование класса от нескольких классов), и в этой ситуации порядок MRO усложняется.

Но пока сосредоточимся на самом распространенном сценарии: подклассе с одним надклассом. На рис. 8.10 показано, как работает порядок MRO. Следует заметить, что при определении класса, который не имеет явно заданного надкласса, Python неявно использует класс `object` в качестве надкласса — так, `Employee` является подклассом `object`.

Когда вы вызываете метод для экземпляра, у этого экземпляра существует четко определенный порядок MRO, который можно просмотреть при помощи метода `mro`:

```
Supervisor.mro()

# Выводимая строка:
[<class '__main__.Supervisor'>, <class '__main__.Employee'>,
➥ <class 'object'>]
```

Как видите, выбор происходит в порядке `Supervisor -> Employee -> object`. Другими словами, если в указанном порядке реализация будет обнаружена в каком-либо из классов, она будет выбрана и использована. Если после рассмотрения всех классов реализация так и не будет найдена, выдается исключение `AttributeError`.

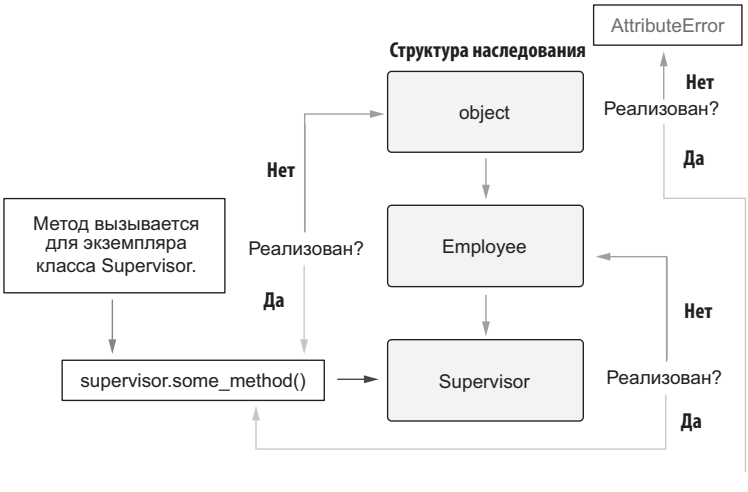


Рис. 8.10. MRO определяет иерархическую структуру классов. При вызове метода для экземпляра Python сначала ищет метод в самом классе. Если метод разрешен, его реализация применяется, а если нет — поиск переходит к надклассу. Если метод не будет найден и там, поиск продолжается в надклассе object. Если вызов метода так и не будет разрешен, выдается исключение AttributeError. Если структура классов содержит большее количество уровней, то при поиске будет проверен каждый уровень

Частичное переопределение метода

Не всегда требуется определять в подклассе новую реализацию, отличную от реализации из надкласса. Иногда удобнее оставить реализацию из надкласса, поверх которой накладываются дополнительные действия. Такая ситуация называется *частичным переопределением* метода.

На этот раз обратимся к методу `logout`. Кроме реализации из надкласса, необходимо добавить поведение, специфическое для руководителя, — для простоты пусть это будет вывод сообщения `Additional logout actions for a supervisor`. Следующий фрагмент кода показывает, как можно было бы реализовать такое поведение:

```
class Supervisor(Employee):
    def logout(self):
        super().logout()
        print("Additional logout actions for a supervisor")
```

В первую очередь следует обратить внимание на использование `super()` для создания объекта-заместителя, представляющего надкласс. С концептуальной точки зрения `super()` рассматривается как временный экземпляр надкласса, позволяющий вызвать метод `logout` надкласса для этого объекта. Какого вывода следует ожидать с этим частичным переопределением метода `logout`? Результат выглядит так:

```
supervisor = Supervisor("John", "1001")

supervisor.logout()
# Выводимые строки:
An employee John just logged out.
Additional logout actions for a supervisor
```

Из вывода видно, что при вызове `logout` для экземпляра `supervisor` вызывается не только метод `logout` класса `Employee` через `super().logout()`, но и дополнительная специализированная реализация из метода `logout` класса `Supervisor`.

8.5.4. Создание ограниченных методов в надклассе

В разделе 8.3 были представлены две категории ограниченных (не общедоступных) атрибутов/методов: защищенные и приватные. Кроме различий в именах (префиксы из одного и двух символов подчеркивания), также упоминалось, что они отличаются друг от друга по своей доступности в подклассе. В этом разделе мы проанализируем это различие и вы узнаете, в каких случаях следует создавать защищенные или приватные методы с точки зрения наследования классов.

Для начала предположим, что надкласс `Employee` содержит следующую реализацию. Кроме метода инициализации, мы определим один защищенный метод `_request_vacation` и один приватный метод `__transfer_group`:

```
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def _request_vacation(self):
        print("Send a vacation request to the employee's supervisor.")

    def __transfer_group(self):
        print("Transfer the employee to a different group.")
```

Все готово для создания подкласса `Supervisor`, наследующего от `Employee`. Чтобы продемонстрировать различия между защищенными и приватными компонентами в отношении доступности в подклассе, попробуем обратиться к следующим ограниченным методам из `Supervisor`:

```
class Supervisor(Employee):
    def do_something(self):
        self._request_vacation()
        self.__transfer_group()
```

В этом подклассе определяется метод экземпляра `do_something`, внутри которого мы вызываем `_request_vacation` и `__transfer_group`. Как вы думаете, что произойдет при вызове `do_something`? Не торопитесь с ответом. Вспомните, что подклассы наследуют защищенные методы. А если вы готовы, то проверьте свой ответ:

```
supervisor = Supervisor("John", "1001")
supervisor.do_something()

# Выводимые строки:
Send a vacation request to the employee's supervisor.
# ERROR: AttributeError: 'Supervisor' object has no attribute
➥ '_Supervisor__transfer_group'
```

Как видите, метод `_request_vacation` вызывается успешно, чего и следовало ожидать. Но метод `__transfer_group` вызвать не удалось, потому что префикс `__` инициирует искажение имен. Вместо того чтобы пытаться вызвать `__transfer_group`, Python пытается вызвать `_Supervisor__transfer_group` — метод, не определенный в `Supervisor`!

С учетом разной доступности в подклассах ограниченные методы следует определять на основании следующего критерия: если ожидается, что подклассам понадобится доступ к ограниченным методам, определяйте защищенные методы, которые могут наследоваться подклассами. Если ожидается, что подклассы не должны иметь доступа к ограниченным методам, определяйте приватные методы.

8.5.5. Обсуждение

Создание иерархической структуры классов — один из важнейших механизмов в мире ООП, и этот навык критичен для построения ясной, простой

в сопровождении кодовой базы. Надкласс отвечает за атрибуты и методы, которые используются всеми его подклассами. Вместо того чтобы работать с методами в нескольких разных местах (как это было бы при определении одинаковых методов в похожих классах), достаточно управлять этими методами только в одной точке: в надклассе.

Следует понимать, что создание иерархической структуры классов не дается даром. Так как подклассы зависят от поведения надклассов, эта взаимосвязь (сильное связывание) может усложнить обновление кодовой базы. Когда вы хотите что-нибудь добавить в подкласс, вам также необходимо обновить его надкласс. Таким образом, в проектах лучше использовать плоские модели данных. Но если вы заметите перекрытие функциональности между классами, без лишних колебаний реализуйте надклассы и подклассы.

8.5.6. Задача

В разделе 8.1 вы узнали, как реализовать метод `__init__` в пользовательском классе. Если подкласс содержит такую же реализацию, как надкласс, переопределять `__init__` вообще не придется. Но если вам нужна специальная инициализация, как в случае `Supervisor`, необходимо переопределить `__init__`. Как переопределить `__init__` в классе `Supervisor`?

> **ПОДСКАЗКА** Переопределение `__init__` не отличается от переопределения других методов. Вызов `super()` используется для создания объекта-заместителя, через который будет вызываться конструктор надкласса.

ИТОГИ

- Ваш класс прежде всего должен содержать метод `__init__` и инициализировать все атрибуты экземпляра, даже если некоторые атрибуты содержат значение `None`.
- Метод инициализации `__init__` является методом экземпляра, который получает `self` в первом параметре. Вы должны знать, что происходит «за кулисами» — как создается экземпляр при вызове конструктора.
- Если все экземпляры совместно используют одни и те же значения атрибутов, их следует определить в виде атрибутов класса для экономии памяти.
- В общем случае в классе могут определяться три разновидности методов: методы экземпляров (первый параметр `self`), статические методы (использующие декоратор `@staticmethod`) и методы класса (использующие декоратор `@classmethod`). Вы должны знать, чем различаются эти методы и в каких случаях использовать ту или иную разновидность.
- Когда вы определяете класс, постарайтесь свести к минимуму состав атрибутов и методов, которые должны быть доступны для пользователя. «Скрывая»

их (например, определением защищенных и приватных методов), вы помогаете пользователю повысить эффективность программирования, потому что ему не нужно отвлекаться на эти ограниченные методы в списке подсказок автозаполнения.

- Декоратор `property` позволяет создать свойство, доступное только для чтения. Такие свойства помогают обеспечить целостность данных за счет блокировки их данных. Если вы намерены разрешить пользователям изменять свойство, создайте для него set-метод (сеттер). Заодно в сеттере можно проверить целостность данных.
- При определении класса желательно переопределить как `__str__`, так и `__repr__`, чтобы вы могли предоставить правильные строковые представления для пользователей и разработчиков.
- Иерархическая структура классов помогает организовать управление данными при сходстве между моделями данных. Общие данные выносятся в надкласс, что упрощает разработку и сопровождение кода.
- Дважды подумайте перед созданием иерархической структуры классов, потому что необходимость работы с надклассами и подклассами усложнит модели данных.

9 Продвинутое использование классов

В этой главе

- ✓ Создание перечислений
- ✓ Устранение шаблонного кода из пользовательских классов
- ✓ Обработка данных в формате JSON
- ✓ Создание отложенных атрибутов
- ✓ Рефакторинг неэффективных классов

Python по своей сути является объектно-ориентированным языком. Отличительный признак объектно-ориентированного языка — возможность использования объектов для хранения данных и предоставления функциональности, для чего вам обычно приходится реализовывать качественно определенные пользовательские классы. В главе 8 рассматривались основные средства определения классов. Но существует много других механизмов, которые помогают определять более надежные и функциональные пользовательские классы, чтобы вы могли строить простую в сопровождении кодовую базу с четко определенными моделями классов.

Пользовательские классы обычно требуют реализации нескольких специальных методов, например `__init__` и `__repr__`. По мере накопления практического опыта вы увидите, что написание таких методов — рутинная и однообразная

работа, так как они в основном содержат шаблонный код. А вы знали, что для устранения шаблонного кода можно воспользоваться декоратором `dataclass`?

В этой главе рассматриваются средства Python более высокого уровня. Некоторые из них (такие, как создание перечислений) предназначены для конкретных ситуаций (например, для представления статуса задач в таск-менеджере). Другие средства имеют более фундаментальную природу (среди них рефакторинг неэффективных классов и создание отложенных атрибутов), и они пригодятся независимо от того, какое приложение вы создаете. Эти средства, не зависящие от специфики проектов, заслуживают особого внимания.

9.1. КАК СОЗДАВАТЬ ПЕРЕЧИСЛЕНИЯ

В наших приложениях некоторые данные естественным образом группируются под одной концептуальной «крышей». Например, четыре стороны света — север, восток, юг и запад — принадлежат к одной категории «направление». Когда вы представляете эти данные в своем приложении, проще всего воспользоваться строками: `"north"`, `"east"`, `"south"` и `"west"`. Но если вы написали функцию, которая ожидает получить направление, пользователям может быть неочевидно, какие данные они должны предоставить, даже если снабдить функцию аннотациями типов, как в следующем примере:

```
def move_to(direction: str, distance: float):
    if direction in {"north", "south", "east", "west"}:
        message = f"Go to the {direction} for {distance} miles"
    else:
        message = "Wrong input for direction"
    print(message)
```

Так как строки не обладают внутренней семантикой, при вызове этой функции пользователь не знает, что именно нужно передать. Он может использовать семантически осмысленную строку, которая окажется несовместимой с функцией:

```
move_to("North", 2)
# Вывод: Wrong input for direction
```

Понятно, что если предоставить более конкретную информацию типа о параметре `direction`, пользователю будет ясно, что именно нужно передать. Если вам потребуется определить тип, который представляет собой набор дискретных значений (например, дни недели или времена года), в такой ситуации идеально подойдет перечисление. В этом разделе рассматривается данная возможность.

9.1.1. Перечисления без обычных классов

Когда речь заходит о реализации перечислений, некоторым разработчикам кажется, что они строятся на базе обычных пользовательских классов. Но как показано в этом разделе, использование обычного класса для перечисления

приводит к ряду проблем. Для начала посмотрим, как выглядит возможная реализация на базе пользовательского класса:

```
class Direction:
    NORTH = 0
    SOUTH = 1
    EAST = 2
    WEST = 3
```

С точки зрения стиля стоит обратить внимание на два обстоятельства:

- Так как четыре направления являются константами, уместно записывать их имена в верхнем регистре.
- В большинстве языков программирования перечисления используют целые числа как значения перечисляемых элементов.

Кроме этих двух замечаний по стилю, реализация с определением атрибутов класса в классе `Direction` выглядит немного неаккуратно. Вы можете использовать эти «перечисления» (они не являются полноценными перечислениями, как вы увидите в разделе 9.1.3), обращаясь к атрибутам класса:

```
print(Direction.NORTH)
# Вывод: 0
print(Direction.SOUTH)
# Вывод: 1
```

Здесь заметны два недостатка. Прежде всего, эти элементы не относятся к типу `Direction`, что не позволяет применять их при использовании `Direction` в функциях (рис. 9.1).

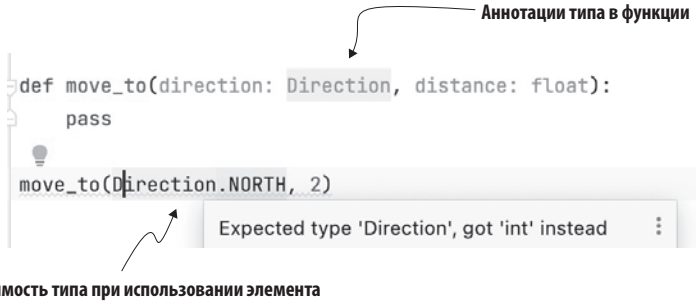


Рис. 9.1. Несовместимость типа при использовании атрибутов класса как перечислений. Пользовательский класс указан в аннотации типа для аргумента, но вы не можете использовать элемент перечисления при вызове функции

Значение элемента `Direction.North` равно `0`, но это целое число, а не экземпляр класса `Direction`. При использовании перечислений ожидается, что каждый элемент будет экземпляром класса перечислений.

Другой недостаток заключается в том, что класс не может использоваться для перебора всех элементов, так как «элементы» являются атрибутами класса; они не образуют объединенную сущность и не представляют концепцию перечисления. А полноценный класс перечислений должен поддерживать перебор всех элементов. Описанные недостатки препятствуют использованию обычного класса для реализации перечислений, поскольку такая реализация не может считаться питонической. Как будет показано в следующем разделе, для устранения этих недостатков используется модуль `enum`.

9.1.2. Создание класса перечислений

Подклассы рассматривались в разделе 8.5. Чтобы создать *класс перечислений* (enumaration class), вы создаете подкласс встроенного класса `Enum` из модуля `enum`. В этом разделе вы узнаете, как реализовать класс перечислений для направлений. Код приведен в листинге 9.1.

Листинг 9.1. Реализация класса перечислений

```
from enum import Enum

class Direction(Enum):
    NORTH = 0
    SOUTH = 1
    EAST = 2
    WEST = 3
```

По сравнению с реализацией в форме пользовательского класса полноценный класс перечислений является подклассом класса `Enum`. При создании подклассов `Enum` класс перечислений преобразует атрибуты класса в дискретные элементы. В теле класса задаются элементы и связанные с ними значения. Стоит заметить, что класс перечислений также можно создать в однострочной форме:

```
class DirectionOneLiner(Enum):
    NORTH = 0; SOUTH = 1; EAST = 2; WEST = 3
```

И хотя элементы в классе перечисления можно объявлять через точку с запятой в одной строке, я рекомендую использовать первый стиль (с определением каждого элемента в отдельной строке), потому что он лучше читается.

> **УДОБОЧИТАЕМОСТЬ** Каждый элемент класса перечислений должен занимать отдельную строку, потому что в этом случае пользователю будет проще увидеть, какие элементы входят в перечисление, и узнать количество этих элементов.

Во многих случаях исходные значения элементов не представляют интереса. В наших примерах это последовательные малые числа, но использовать можно любые целые числа:

```
class DirectionRandomInt(Enum):
    NORTH = 100
    SOUTH = 200
    EAST = 300
    WEST = 400
```

К тому же Python не ограничивает данные, которые будут использоваться для исходных значений элементов. Вместо целых чисел возможны строки, как в следующем примере:

```
class DirectionString(Enum):
    NORTH = "N"
    SOUTH = "S"
    EAST = "E"
    WEST = "W"
```

9.1.3. Использование перечислений

После того как мы определили класс перечислений, пришло время использовать его перечисления. Рассмотрим, как это делать.

Проверка типа элемента перечисления

Первое использование перечислений связано с проверкой типа перечисляемых элементов. Из раздела 9.1.1 мы знаем, что при использовании обычного класса перечисления, использующие атрибуты класса, не относятся к типу класса. В полноценном классе перечислений все происходит совершенно иначе, как показывает следующий пример:

```
north = Direction.NORTH

print("north type:", type(north))
# Вывод: north type: <enum 'Direction'>

print("north check instance of Direction:", isinstance(north, Direction))
# Вывод: north check instance of Direction: True
```

Как видите, «атрибуты» класса перечислений относятся к типу класса — переменная `north` имеет тип класса `Direction`. Иначе говоря, каждый элемент представляет заранее определенный экземпляр класса.

Использование атрибутов элемента перечисления

Так как элементы перечисления фактически являются экземплярами класса перечислений, вполне естественно, что каждый элемент обладает атрибутами экземпляра. Среди них важнейшими являются атрибуты `name` и `value`, содержащие имя элемента и связанное с ним значение соответственно:

```
print("north name:", north.name)
# Вывод: north name: NORTH
print("north value:", north.value)
# Вывод: north value: 0
```

Значение элемента бывает полезно знать во многих ситуациях. Допустим, вы получаете от API ответ, в котором целое число указывает, в каком направлении следует двигаться пользователю. Этот сценарий продемонстрирован в следующем фрагменте:

```
direction_value = 2

direction = Direction(direction_value)

print("Direction to go:", direction)
# Вывод: Direction to go: Direction.EAST
```

Как видите, мы конструируем элемент перечисления, передавая конструктору необходимое значение. Так как `EAST` в классе `Direction` соответствует значение `2`, вызов конструктора с `2` создает направление `EAST`. Если вы попытаетесь создать элемент со значением, которое отсутствует среди определенных значений, происходит исключение:

```
unknown_direction = Direction(8)
# ERROR: ValueError: 8 is not a valid Direction
```

Итерация по всем элементам перечисления

Главной причиной для определения перечислений является группировка взаимосвязанных концепций в форме элементов в классе перечислений. Если пользователь захочет узнать, какие элементы определены, он может перебрать класс перечислений, а для обычных классов эта возможность недоступна. В этом разделе показано, как перебрать элементы класса перечислений.

Класс перечислений `Direction`, являющийся подклассом `Enum`, намеренно создается как итерируемый объект, состоящий из своих элементов. Следовательно, к классу `Direction` применима механика перебора:

```
all_directions = list(Direction)
print(all_directions)
# Вывод: [<Direction.NORTH: 0>, <Direction.SOUTH: 1>,
➥ <Direction.EAST: 2>, <Direction.WEST: 3>]
```

Этот пример показывает, как создать объект `list`, содержащий все возможные направления. Как было сказано в разделе 5.1, список создается вызовом конструктора `list` с итерируемым объектом — классом `Direction`. Так как `Direction` является итерируемым объектом, он также используется в цикле `for`:

```
for direction in Direction:
    pass
```

9.1.4. Определение методов для класса перечислений

В своей основе класс перечислений все еще остается пользовательским классом Python, поэтому вы можете определять методы для реализации в нем более

гибкой функциональности. Вы научились создавать перечисления и узнали, что класс перечислений является итерируемым объектом. Теперь все готово для обновления функции `move_to`, как показано в следующем фрагменте:

```
def move_to(direction: Direction, distance: float):
    if direction in Direction:
        message = f"Go to the {direction} for {distance} miles"
    else:
        message = "Wrong input for direction"
    print(message)
```

Обратите внимание на конструкцию `direction in Direction` для определения того, допустим ли переданный аргумент `direction`. При вызове этой функции вы получаете нужные аннотации типов. Тем не менее результат выглядит не идеально:

```
move_to(Direction.NORTH, 3)
# Вывод: Go to the Direction.NORTH for 3 miles
```

Вместо `north`, как следовало бы предположить, выводится направление `"Direction.NORTH"`. Чтобы решить эту проблему, определим специальный метод экземпляра для получения удобочитаемого вывода элементов, как показано в листинге 9.2.

Листинг 9.2. Добавление специального метода

```
class Direction(Enum):
    NORTH = 0
    SOUTH = 1
    EAST = 2
    WEST = 3

    def __str__(self):
        return self.name.lower()

def move_to(direction: Direction, distance: float):
    if direction in Direction:
        message = f"Go to the {direction} for {distance} miles"
    else:
        message = "Wrong input for direction"
    print(message)

move_to(Direction.NORTH, 3)
# Вывод: Go to the north for 3 miles
```

В листинге 9.2 следует обратить внимание на два важных обстоятельства:

- Мы переопределяем метод `__str__` в классе `Direction`. Как объяснялось в разделе 8.4, метод `__str__` определяет строковое представление экземпляра.
- В f-строке сообщения `direction` заключается в фигурные скобки, что приводит к вызову метода `__str__` во внутренней реализации. Из результатов видно, что мы получаем удобочитаемый вывод для аргумента `direction`.

Фрагмент кода в листинге 9.2 показывает, что в классе перечислений переопределяются специальные методы. Также при необходимости можно определять другие методы. Например, можно определить функцию `move_to` как метод экземпляра класса `Direction`; в разделе 9.1.6 эта задача будет предложена вам для самостоятельного решения.

9.1.5. Обсуждение

Перечисления — самая распространенная структура данных для работы со взаимосвязанными концепциями, принадлежащими одной категории. Чтобы использовать перечисления, определите класс перечислений, создав подклассы класса `Enum` в модуле `enum`. Если потребуется добавить нестандартное поведение в класс перечислений, определите методы, как это делается с обычными классами.

9.1.6. Задача

В ходе работы над картографическим приложением Зоя определяет класс `Direction` так, как показано в предыдущих разделах. В листинге 9.2 функция `move_to` определяется за пределами класса `Direction`, но Зое кажется, что эту функцию лучше определить как метод экземпляра. Помогите ей выполнить это преобразование.

> **ПОДСКАЗКА** Разместите функцию `move_to` в теле класса `Direction`. Не забудьте, что у метода экземпляра первый аргумент `self` обозначает текущий экземпляр.

9.2. КАК ИСПОЛЬЗОВАТЬ КЛАССЫ ДАННЫХ ДЛЯ УСТРАНЕНИЯ ШАБЛОННОГО КОДА

Данные играют важную роль в любом проекте. В каждой программе найдется место для работы с данными. В разделе 3.3 вы узнали о создании облегченной модели данных на базе именованных кортежей. Однако именованные кортежи из-за их неизменяемости лучше использовать для простого хранения данных. Если вам нужна изменяемость данных и более гибкие операции с ними, придется создать пользовательский класс, как было показано в главе 8. В пользовательском классе рекомендуется реализовать такие специальные методы, как `__init__` и `__repr__`:

```
class CustomData:
    def __init__(self, attr0, attr1, attr2):
        self.attr0 = attr0
        self.attr1 = attr1
        self.attr2 = attr2

    def __repr__(self):
        return f"CustomData({self.attr0}, {self.attr1}, {self.attr2})"
```

В методе `__init__` все аргументы присваиваются атрибутам экземпляра, тогда как в методе `__repr__` мы создаем f-строку, которая воспроизводит строковый литерал для создания экземпляра. Оба метода содержат шаблонный код; это означает, что каждый метод строится по заранее определенному шаблону. Если вы определяете много других классов, вам придется делать практически одно и то же в других методах. Почему бы не избавиться от шаблонного кода? В этом разделе вы узнаете, как использовать классы данных для построения классов без шаблонного кода.

ОСНОВНЫЕ ПОНЯТИЯ В программировании *шаблонным кодом* (boilerplate) называется код, который без существенных изменений используется во всех местах, где требуется очень похожий (или идентичный) код. Шаблонный код также считается разновидностью дублирования, хотя и на более высоком уровне.

9.2.1. Создание класса данных с декоратором dataclass

В разделе 7.3 я рассказывал о декораторах, которые предоставляют дополнительную функциональность для декорируемой функции без изменения ее основной функциональности. Впрочем, возможности декораторов вовсе не ограничиваются функциями; при соответствующем определении они также могут применяться к классам. К числу таких специальных декораторов принадлежит декоратор `dataclass`, который решает проблему шаблонного кода посредством декорирования класса.

Декоратор `dataclass` доступен в модуле `dataclasses`. Но прежде чем разбираться в том, как использовать этот декоратор, рассмотрим код из следующего листинга. Он создает класс данных, моделирующий работу со счетами в ресторане.

Листинг 9.3. Создание класса данных

```
from dataclasses import dataclass

@dataclass
class Bill:
    table_number: int
    meal_amount: float
    served_by: str
    tip_amount: float
```

В листинге 9.3 стоит обратить внимание на следующие моменты:

- Декоратор `dataclass` импортируется из модуля `dataclasses`, который является частью стандартной библиотеки Python. Если вы установили Python с официального веб-сайта, модуль `dataclasses` должен уже находиться на вашем компьютере.

- Как и при использовании декоратора с функцией, декоратор размещается над заголовком класса в форме `@dataclass`.
- В теле класса задаются атрибуты с соответствующими типами. Учтите, что для класса данных указание типа обязательно.

В начале этого раздела упоминалось о том, что классы данных используются для устранения шаблонного кода, включая `__init__` и `__repr__`. Иначе говоря, декоратор `dataclass` позаботится о шаблонном коде:

```
bill0 = Bill(5, 60.5, "John", 10)
bill_output = f"Today's bill: {bill0}"
print(bill_output)
# Вывод: Today's bill: Bill(table_number=5, meal_amount=60.5,
➥ served_by='John', tip_amount=10)
```

Как видите, мы создаем объект экземпляра класса `Bill`, хотя метод `__init__` и не определяется явно в классе. Аналогичным образом без реализации метода `__repr__` вы получаете строковое представление экземпляра в правильной форме, которая воспроизводит строку для создания экземпляра.

9.2.2. Определение значений по умолчанию для полей

Установка значений по умолчанию для некоторых атрибутов в методе инициализации способствует чистоте кода и экономит время пользователей. Классы данных поддерживают значения по умолчанию для атрибутов. В этом разделе будут представлены правила определения значений по умолчанию в классах данных.

Прежде чем переходить к техническим аспектам, необходимо прояснить одну важную концепцию. В пользовательском классе под заголовком перечисляются атрибуты класса. В классе данных декоратор `dataclass` преобразует эти атрибуты в атрибуты экземпляра, называемые *полями*. Ранее я упоминал о том, что для этих полей обязательны аннотации типов. Почему? Из-за особенностей работы декоратора `dataclass`, который использует аннотации класса для обнаружения полей:

```
print(Bill.__annotations__)
# output: {'table_number': <class 'int'>, 'meal_amount':
➥ <class 'float'>, 'served_by': <class 'str'>,
➥ 'tip_amount': <class 'float'>}
```

Как видите, для получения всех полей класса используется специальный атрибут `__annotations__`. Если какие-то атрибуты не помечены аннотациями, они не станут частью атрибута `__annotations__`, из-за чего декоратор `dataclass` не сможет их найти. Таким образом, декоратор `dataclass` не сможет правильно сконструировать класс данных. На рис. 9.2 приведена схема создания класса данных.

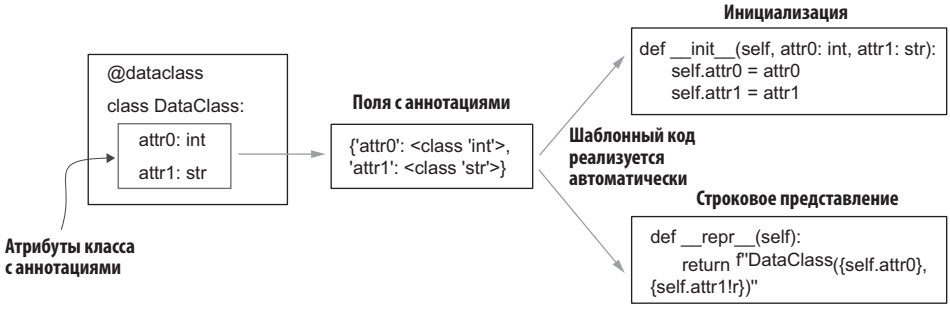


Рис. 9.2. Процесс создания класса данных с декоратором dataclass. Декоратор dataclass использует аннотации типов полей для создания шаблонного кода, включая __init__ и __repr__

На рис. 9.2 класс `dataclass` создает подходящий метод `__init__`, используя снабженные аннотациями поля. Когда вы задаете значения по умолчанию для полей, они также становятся частью метода `__init__`. При этом используется синтаксис, описанный в главе 6: значение по умолчанию указывается после аннотации типа, как в листинге 9.4.

Листинг 9.4. Определение значений по умолчанию для полей

```
@dataclass
class Bill:
    table_number: int
    meal_amount: float
    served_by: str
    tip_amount: float = 0
```

Так как для поля `tip_amount` задается значение по умолчанию, при создании экземпляра класса `Bill` это поле указывать не обязательно, оно будет заполнено значением по умолчанию:

```
bill1 = Bill(5, 60.5, "John")
print(bill1)
# Вывод: Bill(table_number=5, meal_amount=60.5,
➥ served_by='John', tip_amount=0)
```

Когда в разделе 6.1 рассматривались аргументы по умолчанию, я упоминал, что аргумент со значением по умолчанию не может предшествовать аргументам без таких значений. Это правило распространяется и на классы данных. При попытке задать поле со значением по умолчанию, предшествующее другим полям без таких значений, происходит ошибка `TypeError`. Если вы используете интегрированную среду разработки (IDE), например PyCharm, выдается предупреждение (рис. 9.3).

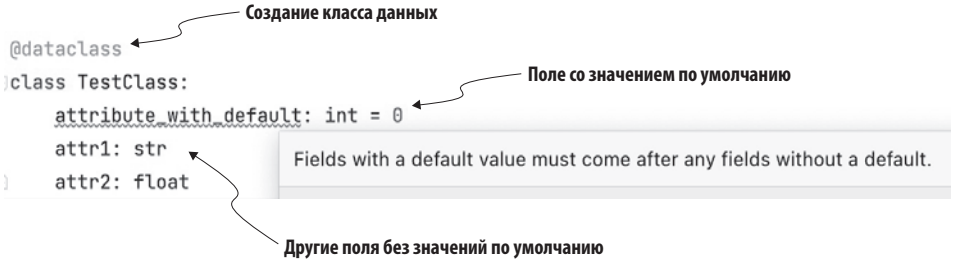


Рис. 9.3. Предупреждение о том, что в определении класса данных поле со значением по умолчанию предшествует полям, не имеющим значений по умолчанию

9.2.3. Определение неизменяемых классов данных

В противоположность неизменяемым именованным кортежам поля классов данных могут изменяться для каждого экземпляра; следовательно, классы данных являются изменяемыми. Однако в некоторых конкретных ситуациях изменяемость оказывается нежелательной. В этом разделе вы узнаете, как сделать классы данных неизменяемыми.

Декоратор `dataclass` не используется без аргументов в форме `@dataclass`, но он может получать дополнительные аргументы, чтобы предоставить нестандартное поведение декорирования. Среди аргументов стоит выделить `init` и `repr`, которым по умолчанию присваивается `True`; тем самым вы требуете, чтобы декоратор `dataclass` реализовал `__init__` и `__repr__`. Из остальных аргументов один — `frozen` — относится к изменяемости. Если вам нужно, чтобы ваш класс данных был неизменяемым, присвойте `frozen` значение `True`. Пример использования будет выглядеть так:

```
@dataclass(frozen=True)
class ImmutableBill:
    meal_amount: float
    served_by: str


immutable_bill = ImmutableBill(50, "John")
immutable_bill.served_by = "David"

# ERROR: dataclasses.FrozenInstanceError: cannot assign
➥ to field 'served_by'
```

Как показано для класса данных `ImmutableBill`, после создания экземпляра обновить его поля не удается. Неизменяемость защищает вас от непреднамеренных изменений данных — как в случае с именованными кортежами, которые проектировались так, чтобы быть неизменяемыми.

СОПРОВОЖДАЕМОСТЬ Если вы не хотите, чтобы ваши классы данных изменяли свои данные, рассмотрите возможность сделать их поля зафиксированными, или «замороженными» (`frozen`), для предотвращения нежелательных изменений.

9.2.4. Создание подкласса для существующего класса данных

По сути, класс данных предоставляет такие же возможности для расширения, как и обычные пользовательские классы. Как упоминалось в разделе 8.5, классы объединяются в иерархию. Класс данных также может использоваться для создания подклассов. Тем не менее из-за некоторых аспектов декоратора `dataclass` создание подклассов класса данных отличается от создания подклассов обычных классов (определяемого без декоратора `dataclass`), о чем и будет рассказано в этом разделе.

Наследование полей надкласса

Вы уже знаете, что в классе данных его атрибуты становятся полями данных. Когда вы создаете подкласс на базе существующего класса данных, все поля надкласса автоматически становятся частью полей подкласса:

```
@dataclass
class BaseBill:
    meal_amount: float

@dataclass
class TippedBill(BaseBill):
    tip_amount: float
```

ВОПРОС Можно ли создать подклассы зафиксированного (`frozen`) класса данных?

Как видно из этого примера, класс `TippedBill` создается как подкласс `BaseBill`. Оба класса должны использовать декоратор `dataclass`, чтобы стать классами данных. Конструктор подкласса `TippedBill` включает как поля надкласса, так и свои собственные поля:

```
tipped_bill = TippedBill(60, 10)
print(tipped_bill)
# Вывод: TippedBill(meal_amount=60, tip_amount=10)
```

При создании экземпляра подкласса помните, что сначала следуют поля надкласса, а за ними — поля подкласса. Порядок важен!

Отказ от значений по умолчанию в надклассе

Вы уже видели, что подкласс класса данных использует все поля из своего надкласса, а также свои собственные поля в порядке *надкласс → подкласс*. Однако

в разделе 9.2.2 вы узнали, что поля со значениями по умолчанию должны предшествовать полям, у которых значений по умолчанию нет. У этого требования есть одно важное следствие: если надкласс содержит поля со значениями по умолчанию, вы должны задать значения по умолчанию для всех полей подкласса. В противном случае код работать не будет, как показывает следующий пример:

```
@dataclass
class BaseBill:
    meal_amount: float = 50

@dataclass
class TippedBill(BaseBill):
    tip_amount: float

# ERROR: TypeError: non-default argument 'tip_amount'
➥ follows default argument
```

Поэтому в большинстве случаев лучше избегать значений по умолчанию в надклассе, чтобы иметь больше гибкости в реализации подклассов. Если вы зададите значения по умолчанию для надкласса, вам также придется задать значения по умолчанию и для подклассов:

```
@dataclass
class BaseBill:
    meal_amount: float = 50

@dataclass
class TippedBill(BaseBill):
    tip_amount: float = 0
```

9.2.5. Обсуждение

При помощи декоратора `dataclass` обычный класс легко преобразуется в класс данных. Такое преобразование помогает избавиться от значительного объема шаблонного кода, который вам пришлось бы писать вручную. Если сравнить классы данных с именованными кортежами, представляющими облегченную модель данных, мы используем классы данных, потому что их модель данных изменяема и они обеспечивают расширяемость за счет определения специализированной функциональности, как и обычные пользовательские классы. При необходимости атрибуты фиксируются, чтобы предотвратить нежелательное изменение данных; впрочем, именованные кортежи также имеют это преимущество.

9.2.6. Задача

Брэдли работает в аналитической группе компании, занимающейся разработкой веб-сайтов. Он использует классы данных в своем проекте. Он знает, что если для изменяемого аргумента функции (раздел 6.1) задать значение по умолчанию,

то обычно в качестве такого значения используется `None`. Но он не знает, какое значение следует использовать для поля изменяемого класса данных (например, `list`). Какое значение по умолчанию вы бы предложили?

> **ПОДСКАЗКА** Модуль `dataclass` содержит функцию с именем `field`, которая предназначена для того, чтобы задавать значения по умолчанию для изменяемых полей.

9.3. КАК ПОДГОТОВИТЬ И ОБРАБОТАТЬ ДАННЫЕ JSON

Если ваше приложение взаимодействует с внешними источниками (например, веб-сайтами), в нем должен быть предусмотрен механизм обмена данными. Допустим, вам приходится загружать данные с другого сервера, обычно через специализированные API. Одним из самых популярных форматов для обмена данными между разными системами является *JSON* (JavaScript Object Notation). Допустим, таск-менеджер получает с сервера следующие данные JSON, напоминающие объект `dict` в Python:

```
{
  "title": "Laundry",
  "desc": "Wash clothes",
  "urgency": 3
}
```

Другой API возвращает следующие данные, напоминающие объект `list` из двух объектов `dict` в Python. Строка была отформатирована с использованием отступов, чтобы она лучше читалась:

```
[
  {
    "title": "Laundry",
    "desc": "Wash clothes",
    "urgency": 3
  },
  {
    "title": "Homework",
    "desc": "Physics + Math",
    "urgency": 5
  }
]
```

Если вы получаете данные в виде строк, для дальнейшей обработки их необходимо преобразовать в соответствующие классы (глава 8). В целом же замечательная удобочитаемость формата и его подобная объектной структура делают JSON универсальным форматом данных для любых приложений. В этом разделе вы узнаете о важнейших средствах обработки данных JSON в Python.

9.3.1. Структура данных JSON

Прежде чем изучать принципы обработки данных JSON, необходимо разобраться со структурой JSON и ее связью с типами данных Python. В этом разделе мы познакомимся с данными JSON. Если вы уже хорошо знакомы с этой темой, переходите к следующему разделу.

Данные JSON структурированы в виде объектов JSON в форме пар «ключ — значение», заключенных в фигурные скобки: `{"title": "Laundry", "desc": "Wash clothes", "urgency": 3}`. Ключи объектов JSON могут быть только строками, и это требование позволяет организовать стандартизированный обмен данными между разными системами. Здесь показаны строки и целые числа, но JSON также поддерживает другие типы данных, включая логические значения, массивы (такие, как `list` в Python) и объекты; эти типы перечислены в табл. 9.1.

Таблица 9.1. Типы данных JSON

Тип данных	Содержание
Строка	Строковые литералы, заключенные в двойные кавычки
Число	Числовые литералы, включая целые и дробные числа
Логическое значение	Логические значения true или false (в нижнем регистре)
Массив	Список поддерживаемых типов данных, заключенный в квадратные скобки
Объект	Пары «ключ — значение», заключенные в фигурные скобки
Null	Специальное значение (null), представляющее пустое значение для любого действительного типа данных

Вы уже знаете, что строки Python заключаются в одинарные или двойные кавычки. А вот строки JSON должны заключаться только в двойные кавычки. Использование одинарных кавычек некорректно и создает недействительные данные JSON, которые не могут быть обработаны обычным парсером JSON.

НА ЗАМЕТКУ Строки JSON могут заключаться только в двойные кавычки.

Важно, что JSON поддерживает вложенные структуры данных. Например, объект JSON может содержать другой объект JSON. Массив может содержать любой список поддерживаемых типов данных, включая объекты. Приведу несколько примеров:

```
embedded object:    {"one": {"one": 1}, "two": {"two": 2}}
array of strings:   ["one", "two", "three"]
```

Гибкость смешения разных типов данных в JSON позволяет конструировать имеющие понятную структуру сложные данные в форме пар «ключ — значение».

9.3.2. Соответствие типов данных JSON и Python

Если вы используете Python для создания приложений, а ваши приложения взаимодействуют с другими системами через JSON, вам необходимо преобразовывать данные между JSON и Python. На высоком уровне преобразование сводится к отображению разных типов данных JSON на соответствующие типы данных Python.

Так как и JSON, и Python достаточно универсальны, не приходится удивляться, что у типов данных JSON имеются соответствующие встроенные структуры данных Python. На рис. 9.4 представлены эти соответствия. Многие преобразования выполняются прямолинейно, однако в Python нет встроенного типа данных, который считался бы аналогом чисел в объектах JSON, где целые числа не отличаются от чисел с плавающей точкой и обозначаются общим термином «числа». А в Python для представления целых чисел и чисел с плавающей точкой используются типы `int` и `float` соответственно.

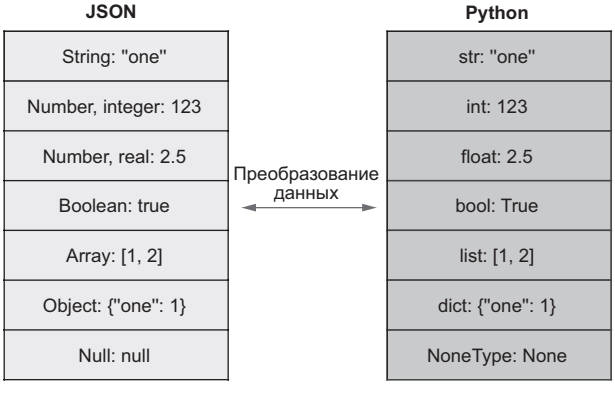


Рис. 9.4. Преобразование данных между JSON и Python с примерами. В JSON и Python эти типы имеют разные имена (например, String и str) из-за различий в терминологии, используемой в этих языках

9.3.3. Десериализация строк JSON

При чтении данных JSON в структуры данных других языков программирования, таких как Python, выполняется *декодирование* (decoding), или *десериализация* (deserialization), данных JSON. Термин «десериализация» является более формальным обозначением процесса чтения с декодированием. В этом разделе вы научитесь читать данные JSON в Python.

Ранее я упоминал о том, что многие веб-сервисы используют объекты JSON в ответах API и что эти ответы передаются в текстовой форме для упрощения

обмена данными между системами. Рассмотрим ответ на запрос, выраженный в виде объекта строки Python:

```
tasks_json = """
[
  {
    "title": "Laundry",
    "desc": "Wash clothes",
    "urgency": 3
  },
  {
    "title": "Homework",
    "desc": "Physics + Math",
    "urgency": 5
  }
]
"""   <-- """ Тройные кавычки используются для многострочного текста
```

Стандартная библиотека Python содержит модуль `json`, специализирующийся на десериализации данных JSON. Для чтения строк JSON используется метод `loads`. Как показано в следующем фрагменте, мы получаем объект `list` из двух отформатированных объектов `dict`, которые представляют два объекта JSON, изначально хранившихся в массиве JSON:

```
import json

tasks_read = json.loads(tasks_json)

print(tasks_read)
# Вывод: [{'title': 'Laundry', 'desc': 'Wash clothes', 'urgency': 3},
➥ {'title': 'Homework', 'desc': 'Physics + Math', 'urgency': 5}]
```

Если данные хранятся в форме словарей, мы не сможем воспользоваться функциональностью, определенной в классе `Task`, что рассматривалось в главе 9. Следовательно, объекты `dict` необходимо преобразовать в экземпляры класса `Task`. Для такого преобразования идеально подходят методы класса, как показано в листинге 9.5.

Листинг 9.5. Преобразование объектов dict в экземпляры пользовательского класса

```
from dataclasses import dataclass

@dataclass
class Task:
    title: str
    desc: str
    urgency: int

@classmethod
```

```
def task_from_dict(cls, task_dict):
    return cls(**task_dict)

tasks = [Task.task_from_dict(x) for x in tasks_read]

print(tasks)
# output: [Task(title='Laundry', desc='Wash clothes', urgency=3),
➥ Task(title='Homework', desc='Physics + Math', urgency=5)]
```

В листинге 9.5 список `list` объектов `dict` успешно преобразуется в список экземпляров `Task`, как и было запланировано. При этом использовались некоторые приемы, которые были описаны ранее. Как упоминалось в главе 1 (раздел 1.4), мы стараемся синтезировать различные практические приемы в процессе изложения материала. Отметим ряд ключевых моментов:

- С классом `Task` используется декоратор `dataclass` (раздел 9.2), чтобы нам не пришлось программировать шаблонные реализации `__init__` и `__repr__`.
- Аргумент `cls` метода класса (раздел 8.2.3) `task_from_dict` обозначает класс `Task`.
- Как упоминалось ранее, `**kwargs` обозначает переменное количество ключевых аргументов (раздел 6.4), упакованных в объект `dict`. И наоборот, чтобы получить доступ к парам «ключ — значение», оператор ** преобразует объект `dict` в ключевые аргументы, которые используются конструктором для создания нового экземпляра класса `Task`.

Вы уже знаете, как преобразовать массив JSON в объект `list` в Python. Метод `loads` отличается гибкостью. Он не ограничивается преобразованием массивов JSON, а также разбирает любые типы данных JSON помимо объектов. Приведу несколько примеров:

```
json.loads("2.2")
# Вывод: 2.2

json.loads('"A string"')
# Вывод: 'A string'

json.loads('false')  <-- Логическое значение
# Вывод: False

json.loads('null') is None  <-- null в JSON преобразуется в None на языке Python
# Вывод: True
```

Эти строки представляют данные JSON, включая числа с плавающей точкой, строки, логические значения и Null, и все они преобразуются вызовом `loads` без какой-либо дополнительной настройки. Все преобразования выполняются автоматически, что подчеркивает мощь Python как языка общего назначения.

9.3.4. Сериализация данных Python в формат JSON

При обработке данных JSON, полученных из внешних источников, вы строите входной коммуникационный маршрут. Также может понадобиться построить выходной маршрут, чтобы ваше приложение передавало информацию внешнему миру.

Как показано на рис. 9.5, противоположностью десериализации данных JSON является создание данных JSON по другим данным — этот процесс называется *сериализацией*. Таким образом, при преобразовании данных Python в данные JSON выполняется сериализация объектов Python в данные JSON.

Рис. 9.5. Преобразование данных между JSON и Python. Преобразование JSON в Python называется десериализацией; преобразование Python в JSON называется сериализацией

Как и метод `loads`, модуль `json` содержит метод `dumps` для выполнения сериализации данных JSON. Для основных встроенных типов данных преобразование выполняется достаточно просто:

```
builtin_data = ['text', False, {”0”: None, 1: [1.0, 2.0]}]

builtin_json = repr(json.dumps(builtin_data))

print(builtin_json)
# Вывод: '["text", false, {"0": null, "1": [1.0, 2.0]}]'
```

Чтобы строка была заключена в кавычки, используйте repr

В этом примере метод `dumps` создает массив JSON, содержащий разные виды данных JSON. Главное, что следует учесть: хотя исходный объект `list` использует структуры данных Python, сгенерированная строка JSON будет содержать соответствующие структуры данных JSON. Обратите внимание на следующие преобразования:

- Строка, заключенная в одинарные кавычки (`'text'`), теперь использует двойные кавычки (`"text"`).
- Объект Python `bool`, содержащий `False`, преобразуется в `false`.
- Объект `None` преобразуется в `null`.
- Так как только строки могут быть ключами JSON, число `1` автоматически преобразуется в свое строковое представление `"1"`.

Что произойдет при попытке сериализовать экземпляр пользовательского класса, например `Task`? Результат выглядит так:

```
json.dumps(tasks[0])
# ERROR: TypeError: Object of type Task is not JSON serializable
```

Как видите, сериализовать экземпляр пользовательского класса невозможно. Главная причина заключается в том, что для пользовательского класса экземпляр может содержать множество атрибутов и других метаданных, так что без необходимых инструкций Python не будет знать, какие данные нужно сериализовать. Таким образом, чтобы сделать пользовательский класс сериализуемым, необходимо предоставить инструкции для сериализации. Вот одно из возможных решений (замечу, что существуют и другие):

```
dumped_task = json.dumps(tasks[0], default=lambda x: x.__dict__)

print(dumped_task)
# Вывод: {"title": "Laundry", "desc": "Wash clothes", "urgency": 3}
```

Самое значительное изменение, внесенное в функцию `dumps`, использует аргумент по умолчанию. Этот аргумент указывает, какой объект должен использоваться кодировщиком (объектом, который выполняет кодирование или сериализацию), если тот не может выполнить сериализацию объекта. В данном случае, так как мы знаем, что кодировщик не может сериализовать экземпляр класса `Task`, мы указываем ему использовать представление в виде словаря. Кодировщик знает, как преобразовать встроенный класс `dict`.

В процессе преобразования часто используются две другие возможности. Прежде всего, чтобы объекты JSON создавались в более удобочитаемом формате, задается аргумент `indent` для создания отступов правильного размера:

```
task_dict = {"title": "Laundry", "desc": "Wash clothes", "urgency": 3}

print(json.dumps(task_dict, indent=2))
# Выводимые строки:
{
  "title": "Laundry",
  "desc": "Wash clothes",
  "urgency": 3
}
```

Каждый уровень снабжается отступом, который обозначает относительную структуру объектов JSON и их пар «ключ — значение».

> **УДОБОЧИТАЕМОСТЬ** Используйте отступы для улучшения удобочитаемости данных JSON. Удобочитаемость особенно важна при создании строк JSON.

Другая полезная возможность — аргумент `sort_keys`. Так как мы присвоили ему `True`, в созданной строке JSON ключи упорядочиваются по алфавиту, что упрощает поиск информации, особенно для множественных элементов. В листинге приведен пример использования этой возможности:

```
user_info = {"name": "John", "age": 35, "city": "San Francisco",
➥"home": "123 Main St.", "zip_code": 12345, "sex": "Male"}

print(json.dumps(user_info, indent=2, sort_keys=True))
# output the following lines:
{
  "age": 35,
  "city": "San Francisco",
  "home": "123 Main St.",
  "name": "John",
  "sex": "Male",
  "zip_code": 12345
}
```

9.3.5. Обсуждение

Вероятно, JSON — самый популярный формат, используемый при обмене данными между разными системами. Вы должны уметь сериализовать и десериализовать данные JSON с использованием объектов Python. Важно заметить, что экземпляры пользовательских классов в Python по умолчанию не сериализуются в JSON, так что вам придется самостоятельно определить поведение кодирования. Кроме методов для работы со строками JSON, модуль `json` также содержит методы `dump` и `load` для прямой обработки файлов JSON. Сигнатуры вызова этих методов почти идентичны сигнатурам `dumps` и `loads` соответственно.

9.3.6. Задача

Лукас строит веб-приложение социальной сети для своей курсовой работы. В этом приложении он использует именованные кортежи в моделях данных. Допустим, проект содержит следующий класс на базе именованных кортежей:

```
from collections import namedtuple

User = namedtuple("User", "first_name last_name age")
user = User("John", "Smith", "39")
```

Что произойдет при попытке сериализовать объект `user`?

> **ПОДСКАЗКА** Объект `tuple` сериализуется в JSON и становится массивом JSON после сериализации.

9.4. КАК СОЗДАТЬ ОТЛОЖЕННЫЕ АТРИБУТЫ ДЛЯ УЛУЧШЕНИЯ БЫСТРОДЕЙСТВИЯ

Отложенное, или ленивое, вычисление (lazy evaluation) — общая парадигма программной реализации, при которой вычисление откладывается непосредственно до того момента, когда его потребуется произвести. Обычно отложенное вычисление желательно применять для затратных операций, требующих значительного времени обработки или памяти. Например, генераторы (раздел 7.4) являются примерами отложенного вычисления, которое откладывает получение и выдачу следующего элемента. Отложенное вычисление также актуально для пользовательских классов — вы определяете отложенные атрибуты для экземпляров с целью экономии времени или памяти. В этом разделе вы узнаете, как определяются отложенные атрибуты.

9.4.1. Сценарий использования

Для начала рассмотрим пример ситуации, в которой уместно использовать отложенные атрибуты. Допустим, таск-менеджер представляет собой клиент социальной сети, в котором пользователь способен подписываться на учетные записи других пользователей. Одной из возможных функций является просмотр всех подписчиков пользователя. Далее в приложении можно просмотреть подробный профиль пользователя, если нажать на его аватарку. Реализация приведена в следующем листинге.

Листинг 9.6. Создание класса User

```
class User:
    def __init__(self, username):
        self.username = username
        self.profile_data = self._get_profile_data()
        print(f"### User {username} created")

    def _get_profile_data(self):
        # Получение данных с сервера и загрузка их в память
        print("*** Run the expensive operation")
        fetched_data = " Extensive data, including thumbnail,
        ➥ followers, etc."
        return fetched_data

def get_followers(username):
    # Получение информации о подписчиках пользователя с сервера
    usernames_fetched = ["John", "Aaron", "Zack"]
    followers = [User(username) for username in usernames_fetched]
    return followers
```

Мы определяем класс `User` для управления данными, относящимися к пользователю, а функция `get_followers` получает подписчиков для пользователя. При выполнении этой функции будет получен следующий вывод:

```
followers = get_followers("Ashley")
# Выводимые строки:
*** Run the expensive operation
### User John created
*** Run the expensive operation
### User Aaron created
*** Run the expensive operation
### User Zack created
```

Как видите, при получении списка подписчиков мы создаем несколько экземпляров для каждого пользователя. Этот процесс требует затратной операции загрузки данных профиля, так как приложение должно соединиться с удаленным сервером для скачивания данных и загрузить их в память. Однако данные профиля не нужны, потому что если пользователь не нажмет на аватарку одного из подписчиков, достаточно вывести только их имена. Данные профиля подписчика становятся актуальными только при нажатии на его аватарку. Предварительная загрузка данных всех пользователей была бы излишней, так что стоит рассмотреть использование метода отложенного вычисления, чтобы избавиться от ненужной работы. В следующих разделах рассмотрены два способа реализации отложенных атрибутов.

9.4.2. Переопределение специального метода __getattr__ для реализации отложенных атрибутов

В пользовательском классе можно переопределить ряд других специальных методов для определения нестандартного поведения помимо `__str__` и `__repr__` (раздел 8.3). Один из таких методов — `__getattr__` — относится к загрузке атрибутов экземпляра. В этом разделе вы узнаете, как реализовать отложенные атрибуты переопределением `__getattr__`.

Для пользовательских классов атрибуты экземпляра хранятся в объекте `dict`, для обращения к которому используется специальный атрибут `__dict__`. Объект `dict` использует имена атрибутов в качестве ключей, а значения атрибутов становятся соответствующими значениями.

Когда вы обращаетесь к атрибуту экземпляра в точечной записи, вы получаете его значение, если атрибут присутствует в объекте `dict`. Если в объекте `dict` нет такого атрибута, вызывается специальный метод `__getattr__`, который пытается предоставить значение для запрашиваемого атрибута. На рис. 9.6 представлена схема разрешения атрибута экземпляра, в которой участвуют обращения к `__dict__` и `__getattr__`.

ПРИМЕЧАНИЕ Порядок разрешения атрибутов сложнее изображенного на рис. 9.6. Например, атрибут экземпляра также может использовать атрибут класса в качестве резервного значения. На рис. 9.6 представлена упрощенная версия, которая применима к типичным сценариям.

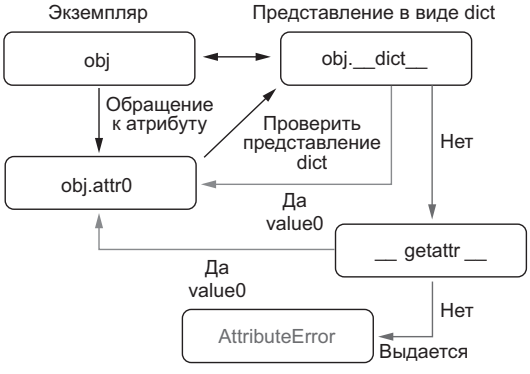


Рис. 9.6. Порядок разрешения атрибута экземпляра. Python сначала проверяет, содержится ли атрибут в объекте dict экземпляра. Если атрибут отсутствует в объекте dict, Python проверяет, возможно ли вернуть значение вызовом специального метода __getattr__

Теперь вы знаете, как `__dict__` и `__getattr__` совместно работают для предоставления запрашиваемых атрибутов экземпляра. Рассмотрим конкретную реализацию с переопределением `__getattr__` для отложенного атрибута, как показано в листинге 9.7.

Листинг 9.7. Переопределение __getattr__ в классе

```
class User:
    def __init__(self, username):
        self.username = username
        print(f"### User {username} created")

    def __getattr__(self, item):
        print(f"__getattr__ called for {item}")
        if item == "profile_data":
            profile_data = self._get_profile_data()
            setattr(self, item, profile_data)
            return profile_data

    def _get_profile_data(self):
        # Получение данных с сервера и загрузка их в память
        print("*** Run the expensive operation")
        fetched_data = "Extensive data, including thumbnail,
        ➥ followers, etc."
        return fetched_data
```

По сравнению с листингом 9.6 в листинге 9.7 присутствуют два важных изменения:

- *Из метода* `__init__` *удаляется присваивание атрибута* `profile_data`. Это удаление необходимо, потому что если атрибут будет инициализирован

(даже значением `None`), атрибут `profile_data` и его значение будут сохранены в атрибуте `__dict__` объекта. Специальный метод `__getattr__` не будет вызван, что противоречит нашему намерению реализовать отложенный атрибут с использованием `__getattr__`.

- *В методе* `__getattr__` *указано, что при обращении к атрибуту* `profile_data` *выполняется затратная операция получения данных профиля пользователя.* Важно заметить, что мы задаем загруженные данные вызовом `setattr`; при повторном обращении к атрибуту `profile_data` он будет доступен немедленно.

С внесенными изменениями ожидается следующий сценарий:

- *Действие 1* — при создании пользователя данных профиля нет, что предотвращает упреждающее выполнение затратной операции.
- *Действие 2* — при обращении к атрибуту может быть инициирована затратная операция, предоставляющая запрошенный атрибут.
- *Действие 3* — при повторном обращении к атрибуту затратную операцию не нужно будет выполнять повторно.

Посмотрим, оправдываются ли эти ожидания:

```
followers = get_followers("Ashley")  <-------  Действие 1
# output the following lines:
### User John created
### User Aaron created
### User Zack created

follower0 = followers[0]
follower0.profile_data  <-------  Действие 2
# output the following lines:
__getattr__ called for profile_data
*** Run the expensive operation
'Extensive data, including thumbnail, followers, etc.'

follower0.profile_data  <-------  Действие 3
'Extensive data, including thumbnail, followers, etc.'
```

Если выполняется действие 1, при получении подписчиков пользователя созданные объекты `User` содержат только имена пользователей, что экономит память. Если выполняется действие 2, то при первом обращении к `profile_data` выполняется затратная операция для получения данных. В случае действия 3 при повторном обращении к `profile_data` данные будут получены без инициирования затратной операции, что экономит время.

9.4.3. Реализация свойства в виде отложенного атрибута

В разделе 8.3 вы научились использовать декоратор `property` для создания доступных только для чтения свойств, чтобы организовать более точное управление

доступом. Так как декоратор `property` позволяет «перехватить» обращения к атрибутам, мы воспользуемся им для реализации функциональности отложенных атрибутов, как обсуждалось в этом разделе. Помните, что формально свойство не является атрибутом, но свойства и атрибуты похожи в том, что касается поддержки точечной записи.

К настоящему моменту вы должны уже разобраться, как работает декоратор `property`. В листинге 9.8 показано, как создать отложенный атрибут с использованием `@property`.

Листинг 9.8. Создание декоратора для отложенного атрибута

```
class User:
    def __init__(self, username):
        self.username = username
        self._profile_data = None
        print(f"### User {username} created")

    @property
    def profile_data(self):
        if self._profile_data is None:
            print("_profile_data is None")
            self._profile_data = self._get_profile_data()
        else:
            print("_profile_data is set")
        return self._profile_data

    def _get_profile_data(self):
        # get the data from the server and load it into memory
        print("*** Run the expensive operation")
        fetched_data = "Extensive data, including thumbnail,
        ➥ followers, etc."
        return fetched_data
```

По сравнению с листингом 9.6 в листинге 9.8 присутствуют два важных изменения:

- *В методе* `__init__` *атрибуту* `_profile_data` *присваивается значение* `None`. Атрибут `_profile_data` является аналогом свойства `profile_data`, находящимся под внутренним управлением; присваивание ему `None` экономит память по сравнению с получением данных во время создания экземпляра.
- *Мы реализуем* `profile_data` *в виде свойства.* В этом методе проверяется, задано ли значение `_profile_data`, и затратная операция выполняется только в том случае, если значение `_profile_data` не задано. Если же оно задано, то возвращается его значение.

Как упоминалось в разделе 9.4.2, должны выполняться те же три действия из класса `User`, реализованные в листинге 9.8:

```
followers = get_followers("Ashley")
# output the following lines:
### User John created
### User Aaron created
### User Zack created

follower0 = followers[0]
follower0.profile_data
# output the following lines:
_profile_data is None
*** Run the expensive operation
'Extensive data, including thumbnail, followers, etc.'

follower0.profile_data
# output the following lines:
_profile_data is set
'Extensive data, including thumbnail, followers, etc.'
```

В полном соответствии с ожидаемыми действиями данные профилей пользователей не загружаются при создании. Вместо этого затратная операция выполняется только тогда, когда данные профиля пользователя будут запрошены явно. В этом и состоит суть отложенного вычисления — оно откладывается до того момента, когда в нем возникает прямая необходимость. Это способствует экономии времени (отказ от выполнения операции, занимающей много времени) и памяти (отказ от хранения большого объема данных).

9.4.4. Обсуждение

Чтобы реализовать атрибуты с отложенным вычислением в пользовательском классе, можно переопределить `__getattr__` или реализовать свойство. Я рекомендую решение со свойством: оно более прямолинейно, а все реализации выражаются явно. А переопределение `__getattr__` требует знания того, как работает порядок разрешения атрибутов экземпляра Python.

9.4.5. Задача

Тим работает над обновлением пакета Python, опубликованного его компанией. API из этого пакета обращается к атрибуту объекта, например инициалам пользователя `user.initials`. С последними обновлениями ему требуется более точно управлять этим атрибутом. Как определить свойство для сопровождения API?

> **ПОДСКАЗКА** И свойства, и атрибуты поддерживают точечную запись. В обновленной кодовой базе ранее определенный атрибут можно преобразовать в свойство.

9.5. КАК ОПРЕДЕЛИТЬ КЛАССЫ С ЧЕТКИМ РАЗДЕЛЕНИЕМ ОТВЕТСТВЕННОСТИ

В процессе разработки проекта вам придется иметь дело с большими объемами данных. Допустим, вы начинаете с одного класса для работы с данными. Со временем объем данных будет расти, и если вы захотите ограничиться одним классом, он станет слишком громоздким и неудобным. Одна из причин заключается в том, что этот класс будет иметь смешанную ответственность; если один класс моделирует разные виды данных, это усложняет сопровождение проекта.

Представьте два сценария, изображенные на рис. 9.7. В первом сценарии один большой прямоугольник (ваш класс) содержит две разновидности объектов (данные). Во втором сценарии используются два меньших прямоугольника (два разных класса), в каждом из которых находится только одна разновидность объектов. Сразу понятно, в каком сценарии удобнее работать с объектами.

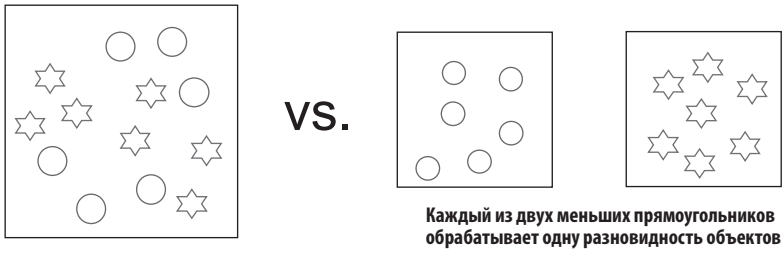


Рис. 9.7. Более эффективная структура, в которой каждая разновидность объектов обрабатывается отдельным типом (прямоугольником), в отличие от одного типа, обрабатывающего разные объекты

В этом разделе я покажу, как определять классы, имеющие четко определенную ответственность, — это может рассматриваться как одна из важнейших форм рефакторинга ваших проектов. Эта тема важна для повышения эффективности сопровождения вашего проекта в долгосрочной перспективе — переместить несколько маленьких блоков проще, чем один большой и неподъемный. Модели данных проще обновлять и сопровождать, если каждый класс будет сконцентрирован на одной цели.

9.5.1. Анализ класса

В идеальном проекте имеется опытный руководитель, который способен спроектировать идеальные структуры данных проекта: иерархию из нескольких небольших классов, каждый из которых обслуживает конкретную модель данных. Но допустим, вам поручили обновление и сопровождение некоторого

унаследованного проекта, и вы выяснили, что важнейшая модель данных представляет собой один огромный класс. Из-за этого обновление проекта становится практически невозможным. В этом разделе вы узнаете, как может выглядеть такой громоздкий класс и как его анализировать.

Допустим, в проекте задействована программа, которая используется образовательным округом для управления данными. Один из ключевых классов — `Student` — используется для хранения всех данных, относящихся к студентам (ученикам). Структура этого класса представлена в листинге 9.9. Обращаю ваше внимание, что для простоты я привожу лишь часть класса `Student`.

Листинг 9.9. Класс со смешанными целями

```
class Student:
    def __init__(self, first_name, last_name, student_id):
        self.first_name = first_name
        self.last_name = last_name
        self.student_id = student_id
        self.account_number = self.get_account_number()
        self.balance = self.get_balance()
        age, gender, race = self.get_demographics()
        self.age = age
        self.gender = gender
        self.race = race

    def get_account_number(self):
        # Запрос к базе данных для получения номера счета
        # по значению student_id
        account_number = 123456
        return account_number

    def get_balance(self):
        # Запрос к базе данных для получения баланса
        # по номеру счета
        balance = 100.00
        return balance

    def get_demographics(self):
        # Запрос к базе данных для получения демографических данных
        # по идентификатору student_id
        birthday = "08/14/2010"
        age = self.calculated_age(birthday)
        gender = "Female"
        race = "Black"
        return age, gender, race

    @staticmethod
    def calculated_age(birthday):
        # Получение текущей даты и вычисление разности с днем рождения
        age = 12
        return age
```

Прежде чем что-нибудь делать с существующим классом, неплохо бы нарисовать его диаграмму и изучить его компоненты. Такие диаграммы строятся

разными способами, главное, рассмотреть структуру на высоком уровне. Для этой цели удобно использовать диаграммы UML (Unified Modeling Language) (рис. 9.8).

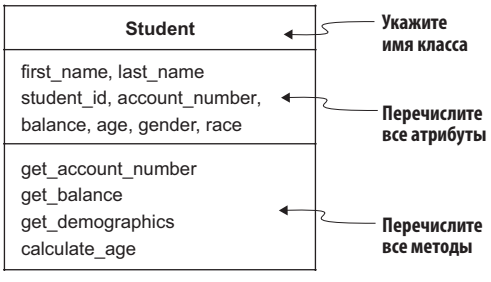


Рис. 9.8. Диаграмма UML (версия 0) класса Student. На диаграмме перечислены все атрибуты и методы класса

ОСНОВНЫЕ ПОНЯТИЯ UML — стандартный формат наглядного представления архитектуры системы. На диаграммах UML представлены компоненты системы и связи между ними.

В версии 0 диаграммы UML приводятся только структурные компоненты класса `Student`. Чтобы данные было удобнее просматривать, вы указываете имена методов без подробностей реализации. После получения структурной информации класса следующим шагом становится анализ его функциональных компонентов (рис. 9.9).

На диаграмме UML (версия 1) методы, совместно реализующие одну и ту же функциональность, группируются вместе. Здесь используются два функциональных компонента: один обрабатывает данные о счетах для оплаты питания, а другой — демографические данные. Кроме того, каждый функциональный компонент содержит взаимосвязанные атрибуты.

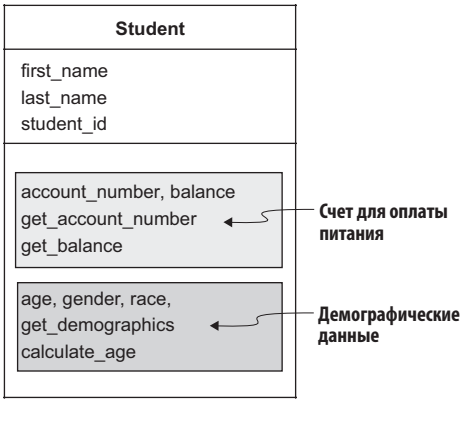


Рис. 9.9. Диаграмма UML (версия 1) класса Student. На диаграмме методы группируются в зависимости от их функциональности

9.5.2. Создание дополнительных классов для разделения ответственности

На рис. 9.9 изображена часть класса `Student`. В реальном проекте класс будет содержать множество других видов функциональности, а такая функциональность, как счета для оплаты питания или демографические данные, может включать другие методы. Так, функция для управления счетами для оплаты питания способна содержать много дополнительных функций, например приостановку действия утерянных электронных карт и консолидацию нескольких счетов. Реализация этих операций усложняет класс `Student`. В таких случаях следует создавать дополнительные классы с отдельными зонами ответственности.

При рассмотрении класса `Student` мы выделили два основных функциональных компонента: счета для оплаты питания и демографические данные. Эти зоны ответственности существуют отдельно от класса `Student`, а следовательно, эти два функциональных компонента могут образовать собственные классы. Но прежде чем писать код, мы продолжим работать над диаграммой UML (рис. 9.10), обновленная версия которой отражает дополнительные структурные компоненты нашего приложения.

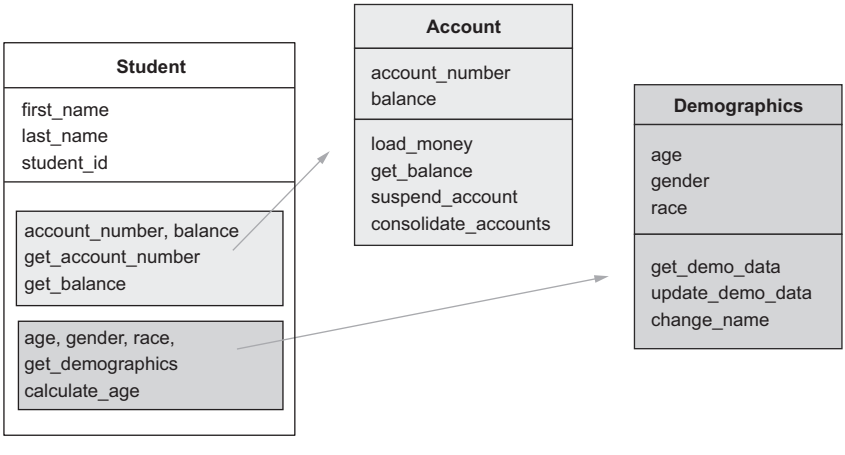


Рис. 9.10. Изоляция функциональности обработки счетов для оплаты питания и демографических данных, выделяемой в отдельные классы (диаграмма UML, версия 2). В каждом случае я перечисляю другие атрибуты и методы, которые могут существовать в каждом классе

На обновленной диаграмме UML представлены два дополнительных класса — `Account` и `Demographics`. Класс `Account` содержит атрибуты и методы для управления счетом для оплаты питания, а класс `Demographics` — все атрибуты и методы для обработки демографических данных студента.

9.5.3. Связывание взаимодействующих классов

Когда мы создавали классы `Account` и `Demographics`, процесс был односторонним, мы извлекали информацию из имеющегося класса `Student`. Эти два класса все еще существуют сами по себе и не взаимодействуют с классом `Student`. В этом разделе мы соединим классы и организуем их точное взаимодействие.

Связывание данных через атрибуты

Вероятно, вы заметили, что классы `Account` и `Demographics` содержат атрибут `student_id`. Так как идентификатор студента уникален, все данные конкретного студента можно связать через `student_id`. Например, для создания двустороннего обмена данными между `Student` и `Account`/`Demographics` экземпляр класса `Student` должен иметь доступ к информации счета для оплаты питания и демографическим данным через `student_id`. Связывание экземпляров продемонстрировано в листинге 9.10.

Листинг 9.10. Изоляция классов для разделения ответственности

```
class Account:
    def __init__(self, student_id):
        # Запрос к базе данных для получения дополнительной информации
        # по значению student_id
        self.account_number = 123456
        self.balance = 100

class Demographics:
    def __init__(self, student_id):
        # Запрос к базе данных для получения дополнительной информации
        # по значению student_id
        self.age = 12
        self.gender = "Female"
        self.race = "Black"

class Student:
    def __init__(self, first_name, last_name, student_id):
        self.first_name = first_name
        self.last_name = last_name
        self.student_id = student_id
        self.account = Account(self.student_id)
        self.demographics = Demographics(self.student_id)
```

В листинге 9.10 определяются классы `Account` и `Demographics`, при этом реализуется только метод инициализации. Метод инициализации класса `Student` обновляется, и в него добавляются два атрибута: `account` и `demographics`, содержащие экземпляры классов `Account` и `Demographics` соответственно. Тем самым мы связываем все три класса. Теперь можно проверить атрибуты экземпляра класса `Student`:

```
student = Student("John", "Smith", "987654")
print(student.account.__dict__)
```

```
# Вывод: {'account_number': 123456, 'balance': 100}

print(student.demographics.__dict__)
# Вывод: {'age': 12, 'gender': 'Female', 'race': 'Black'}
```

Как видите, экземпляр `Student` содержит правильную информацию о счете и демографии, потому что экземпляры `Account` и `Demographics` являются его атрибутами. Также обратите внимание, что мы можем сохранить `student_id` как атрибут экземпляров классов `Account` и `Demographics`. Впрочем, делать это не нужно, потому что экземпляр `Student` уже содержит атрибуты `account` и `demographics`; связь уже была установлена.

Связывание методов

Связывание данных между этими классами весьма прямолинейно. Самое интересное начинается при связывании методов.

Дополнительные классы создаются не только для хранения конкретных атрибутов. Гораздо важнее, что эти классы способны предоставлять специализированную функциональность. Мы планировали переместить все реализации управления счетами в класс `Account`, а все реализации демографических данных — в класс `Demographics`. В листинге 9.11 приведены обновленные версии классов `Account` и `Demographics`.

Листинг 9.11. Обновленные классы Account и Demographics

```
class Account:
    def __init__(self, student_id):
        self.student_id = student_id
        # Запрос к базе данных для получения дополнительной информации
        # по значению student_id
        self.account_number = self.get_account_number_from_db()
        self.balance = self.get_balance_from_db()

    def get_account_number_from_db(self):
        # Запрос к базе данных для получения номера счета
        # по значению student_id
        account_number = 123456
        return account_number

    def get_balance_from_db(self):
        # Запрос к базе данных для получения баланса
        # по номеру счета
        balance = 100.00
        return balance

class Demographics:
    def __init__(self, student_id):
        self.student_id = student_id
        # Запрос к базе данных для получения дополнительной информации
        age, gender, race = self.get_demographics_from_db()
        self.age = age
```

```
        self.gender = gender
        self.race = race

    def get_demographics_from_db(self):
        # Запрос к базе данных для получения демографических данных
        # по идентификатору student_id
        birthday = "08/14/2010"
        age = self.calculated_age(birthday)
        gender = "Female"
        race = "Black"
        return age, gender, race

    @staticmethod
    def calculated_age(birthday):
        # Получение текущей даты и вычисление разности с днем рождения
        age = 12
        return age
```

ВОПРОС Если операции с базой данных обходятся дорого (например, если база данных размещается в облаке), их можно реализовать в форме отложенных атрибутов. А вы помните, как это делается? Обратитесь к разделу 9.4.

Каждый раз, когда в приложении требуется вывести информацию о счете, мы можем воспользоваться классом `Account` напрямую. Чтобы вывести баланс для конкретного студента, выполним следующий код:

```
balance_output = f"Balance: {student.account.balance}"
print(balance_output)
# Вывод: Balance: 100.0
```

Аналогичным образом демографические данные студента выводятся с помощью такого кода:

```
demo = student.demographics
demo_output = f"Age: {demo.age}; Gender: {demo.gender}; Race: {demo.race}"
print(demo_output)
# Вывод: Age: 12; Gender: Female; Race: Black
```

Обратите внимание: некоторые пользователи предпочитают работать с методами из меньшего количества классов, поэтому они создают некоторые методы в классе `Student`:

```
class Student:
    def __init__(self, first_name, last_name, student_id):
        self.first_name = first_name
        self.last_name = last_name
        self.student_id = student_id
        self.account = Account(self.student_id)
        self.demographics = Demographics(self.student_id)

    def get_account_balance(self):
        return self.account.balance
```

```
def get_demographics(self):
    demo = self.demographics
    return demo.age, demo.gender, demo.race
```

Для получения баланса и демографических данных вызываются методы `get_account_balance` и `get_demographics` экземпляра `Student`. Тем не менее я не рекомендую использовать эту схему. Она создает слишком сильную связь между классами `Student` и `Account`/`Demographics` — эта проблема известна под названием *сильного связывания, или сцепления* (tight coupling). Если вы обновляете класс `Account`, то возможно, вам также придется обновить класс `Student`, потому что его функциональность (`get_account_balance`) зависит от `Account`.

СОПРОВОЖДАЕМОСТЬ Не создавайте сильное сцепление между взаимосвязанными классами. Слабое сцепление упрощает сопровождение.

9.5.4. Обсуждение

Прежде чем запускать проект, полезно построить диаграмму UML, чтобы спроектировать структуру необходимых классов для управления данными. Впрочем, не ожидайте, что у вас все получится с первого раза. В ходе работы над проектом некоторые классы усложняются. Выработайте полезную привычку время от времени продумывать модели данных в ходе процесса разработки. Стремитесь к тому, чтобы классы были компактными и слабо сцепленными, — иначе говоря, чтобы взаимосвязанные классы работали вместе, но не имели сильных зависимостей друг от друга, так как архитектура с сильными связями усложняет рефакторинг.

9.5.5. Задача

Во фрагментах этого раздела я намеренно сделал все методы классов общедоступными (публичными). Впрочем, как упоминалось в разделе 8.3, доступ к методам, которые не должны вызываться пользователем, лучше ограничить. Выберите в листинге 9.11 методы, доступ к которым следует ограничить, и реализуйте эти ограничения.

ПОДСКАЗКА Чтобы ограничить доступ к методу, снабдите его имя префиксом с символом подчеркивания.

ИТОГИ

- Если вам понадобится сгруппировать взаимосвязанные концепции, определите класс перечислений, создав подклассы `Enum`.
- Класс перечислений поддерживает перебор содержимого и упрощает проверку принадлежности.

- Используйте декоратор `dataclass` при создании классов, чтобы избежать создания шаблонного кода, например, в реализациях `__init__` и `__repr__`. Применяя этот декоратор, не забывайте использовать аннотации данных для создания подходящих полей.
- JSON — универсальный формат обмена данными между системами. Модуль `json` может использоваться для преобразования JSON в собственные структуры данных Python (десериализация JSON) и обратного преобразования (сериализация JSON).
- Экземпляры пользовательских классов обычно несериализуемы в формат JSON. Чтобы для таких классов поддерживалась сериализация JSON, необходимо предоставить конкретные инструкции по кодированию в JSON.
- Метод `__getattr__` может использоваться для реализации отложенных атрибутов, однако вы должны понимать, что `__getattr__` представляет собой резервный механизм, который используется для атрибутов, не содержащихся в атрибуте `__dict__` объекта.
- Реализация свойств позволяет более точно управлять отдельными атрибутами. В случае отложенных атрибутов присвойте `None` атрибуту, находящемуся под внутренним управлением. При запросе атрибута выполняется операция и присваивается значение соответствующего аргумента.
- Классы должны создаваться только с одной целью. Если ваш класс разрастается и вы начинаете понимать, что цели становятся смешанными, проведите его рефакторинг и создайте несколько разных классов, каждый из которых обеспечивает конкретную потребность.
- Используйте диаграммы UML для анализа структуры классов. Они помогают понять логику работы класса на высоком уровне.

Часть 4

Операции с объектами и файлами

Python изначально проектировался как объектно-ориентированный язык. Его модули, пакеты и встроенные типы данных, а также функции и пользовательские классы и их экземпляры представляют собой объекты. Поэтому общие характеристики объектов становятся важнейшей темой, которая должна быть хорошо знакома каждому питонисту. В этой части мы сосредоточимся на основах использования объектов в Python.

Кроме объектов, в этой части также рассматриваются операции чтения и обработки файлов — самые распространенные механизмы хранения данных. Python как язык общего назначения позволяет делать следующее:

- читать данные, хранящиеся в файлах — в виде обычного текста или данных, разделенных запятыми;
- записывать данные в файлы;
- перемещать, удалять и копировать файлы;
- получать метаданные файлов (например, время последнего изменения).

10 Основы работы с объектами

В этой главе

- ✓ Анализ объектов
- ✓ Жизненный цикл объекта
- ✓ Копирование объекта
- ✓ Разрешение переменной: правило LEGB
- ✓ Возможность вызова объекта

В Python нет ничего, кроме объектов, и Python изначально проектировался как язык объектно-ориентированного программирования. В приложениях мы постоянно работаем с объектами. Следовательно, для разработчика важно знать основы использования объектов, особенно экземпляров пользовательских классов, так как эта модель доминирует в приложениях. Так, следует ожидать, что пользователи будут передавать в функции разные типы данных, и мы можем увеличить гибкость путем соответствующей обработки передаваемых данных. Другой пример: если у вас имеется рабочая копия объекта, которую нужно обновить, оставив исходный объект неизменным на случай решения об отмене изменения, объект необходимо скопировать. В этой главе рассматриваются основные принципы работы с объектами. Безусловно, рассказать обо всем в одной главе не удастся, так как в Python все сущности являются объектами и мы не сможем рассмотреть все аспекты их использования. Также

стоит заметить, что в некоторых разделах рассматриваются конкретные задачи (скажем, раздел 10.4 посвящен изменению переменной в другой области видимости), но они используются для объяснения более общих тем (например, порядка поиска переменных).

10.1. КАК ПРОВЕРИТЬ ТИП ПЕРЕМЕННОЙ ДЛЯ ПОВЫШЕНИЯ ГИБКОСТИ КОДА

В программе вы всегда работаете с разными объектами — функциями, классами, экземплярами и т. д. Для примера возьмем пользовательские функции. Бóльшая часть работы программиста связана с написанием функций — определение входных данных, выполнение операций и генерирование вывода. Функции обычно устанавливают конкретные требования к типу входных данных; соответственно, пользователи должны использовать один конкретный тип данных для вызова функции. Возьмем следующую функцию из таск-менеджера, которая фильтрует список задач (аргумент `tasks`) по степени срочности:

```
def filter_tasks(tasks, by_urgency):
    pass
```

Первое, что приходит в голову, — аргумент `by_urgency` должен быть целым числом, чтобы 4 и 5 были допустимыми аргументами. Таким образом, реализация функции может выглядеть так:

```
def filter_tasks(tasks, by_urgency):
    filtered = [x for x in tasks if x.urgency == by_urgency]
    return filtered
```

> **НАПОМИНАНИЕ** Чтобы использовать эту функцию, необходимо создать класс `Task` (глава 8) и экземпляры, которые будут использоваться в качестве аргумента `tasks`.

В теле функции списковое включение используется для выбора задач, степень срочности которых соответствует значению аргумента `by_urgency`. Но ничто не мешает реализовать возможность фильтрации задач с разными степенями срочности: `filter_tasks([4, 5])`. Для поддержки этой функциональности реализация должна выглядеть так:

```
def filter_tasks(tasks, by_urgency):
    filtered = [x for x in tasks if x.urgency in by_urgency]
    return filtered
```

Вместо сравнения целых значений мы используем конструкцию `элемент in список` для проверки того, что степень срочности задачи входит в диапазон предоставленных значений.

Для поддержки этих двух случаев необходим механизм проверки аргумента `by_urgency` и соответствующей фильтрации задач. Эта форма проверки типа объекта является примером *интроспекции* — анализа объекта для определения его характеристик: типов, атрибутов и методов. В этом разделе рассматриваются ключевые приемы интроспекции объектов и практические ситуации для ее применения, при этом основное внимание уделяется гибкости кода. Взяв функцию `filter_tasks` за основу, мы напишем одну функцию, которая будет получать разные виды входных данных.

ОСНОВНЫЕ ПОНЯТИЯ *Интроспекцией* (introspection) называется анализ типа или свойств объекта (например, его атрибутов) во время выполнения программы.

10.1.1. Проверка типа объекта функцией type

В примере из раздела 10.1 для повышения гибкости и возможности передачи `int` или `list` в аргументе функции `filter_tasks` необходимо проверить тип аргумента. В этом разделе вы увидите, какие встроенные функции используются для проверки типа объекта.

Вероятно, в первую очередь вы вспомните о функции `type`. Вызов `type` для объекта возвращает его тип, вы уже неоднократно встречались с этим вариантом использования. Пара примеров напомнит вам, как это делается:

```
print(type(4))
# Вывод: <class 'int'>

print(type([4, 5]))
# Вывод: <class 'list'>
```

Как и ожидалось, `4` имеет тип `int`, а `[4, 5]` — тип `list`. Вы знаете, как получить информацию о типе объекта, поэтому возникает следующий вопрос: как проверить, совпадает ли тип объекта с нужным типом? Ответ оказывается неожиданно простым: нужно сравнить тип объекта с классом:

```
assert (type(4) is int)
assert (type([4, 5]) is list)
```

ВОПРОС Эквивалентны ли операторы `==` и `is` при сравнении двух объектов?

На основании этих сравнений теперь можно обновить функцию `filter_tasks` для обоих сценариев вызова, как показано в листинге 10.1. Обратите внимание: мы упрощаем условие, допуская для аргумента `by_urgency` только две возможности — `int` и `list`.

Листинг 10.1. Сравнение типа объекта с классом

```
def filter_tasks(tasks, by_urgency):
    if type(by_urgency) is list:
        filtered = [x for x in tasks if x.urgency in by_urgency]
    else:
        filtered = [x for x in tasks if x.urgency == by_urgency]
    return filtered
```

Как показывает этот листинг, когда `by_urgency` имеет тип `list`, мы проверяем присутствие `urgency` в списке, а когда `by_urgency` имеет тип `int`, мы сравниваем степень срочности каждой задачи с числом.

10.1.2. Проверка типа объекта с использованием isinstance

Другая полезная функция интроспекции — `isinstance` — проверяет, является ли объект экземпляром заданного класса. Как видно из этого раздела, `isinstance` делает примерно то же, что и `type`, но этот способ проверки типа объекта считается предпочтительным.

При изучении создания doc-строк для функций (раздел 6.5) мы применяли справку для функции `isinstance`, но ее использование подробно не рассматривалось. Теперь пришло время более детально разобраться в том, что можно сделать при помощи `isinstance`:

```
assert isinstance(4, int)

assert isinstance([4, 5], list)
```

В первом аргументе передается объект, а во втором — конкретный класс. На самом деле вторым аргументом также может быть кортеж классов, что позволяет вам гибко проверить объект, сравнивая его с несколькими классами. Пример использования этой возможности:

```
passed_arg0 = [4, 5]
passed_arg1 = (4, 5)

assert isinstance(passed_arg0, (list, tuple))
assert isinstance(passed_arg1, (list, tuple))
```

Например, если ваша функция получает `list` или `tuple`, две проверки объединяются в одном вызове `isinstance`, как показано в предшествующем фрагменте кода. Заметим, что отношения между этими классами эквивалентны проверке условия «или»:

```
assert isinstance([4, 5], list) or isinstance([4, 5], tuple)
```

При помощи функции `isinstance` обновим функцию `filter_tasks` для обработки `by_urgency` как типа `int` или `list`, как показано в листинге 10.2.

Листинг 10.2. Проверка типа объекта с помощью функции isinstance

```
def filter_tasks(tasks, by_urgency):
    if isinstance(by_urgency, list):
        filtered = [x for x in tasks if x.urgency in by_urgency]
    else:
        filtered = [x for x in tasks if x.urgency == by_urgency]
    return filtered
```

Сравнивая листинги 10.1 и 10.2, можно заметить, что `type` и `isinstance` определяют, является ли объект экземпляром заданного типа. Тем не менее эти две функции не эквивалентны.

Когда вы используете `type` для определения типа объекта, выполняется сравнение «один к одному»: тип объекта сравнивается с заданным типом. Функция же `isinstance` обладает большей гибкостью и использует сравнение «один ко многим»: она проверяет на соответствие не только с классом, но и с его надклассом. Таким образом, `isinstance` учитывает наследование классов, а `type` не учитывает. Звучит непонятно? Простой пример:

```
class User:
    pass

class Supervisor(User):
    pass

supervisor = Supervisor()

comparisons = [
    type(supervisor) is User,
    type(supervisor) is Supervisor,
    isinstance(supervisor, User),
    isinstance(supervisor, Supervisor)
]

print(comparisons)
# Вывод: [False, True, True, True]
```

Из первого и второго сравнений видно, что при использовании `type` полученная информация типа относится к конкретному классу `Supervisor`. С другой стороны, хотя `supervisor` является экземпляром класса `Supervisor`, а не класса `User`, `isinstance` также использует информацию о том, что `Supervisor` является подклассом `User`, и возвращает `True` даже в том случае, если экземпляр проверяется на соответствие надклассу `User`.

Такая гибкость важна, так как даже если ваша функция использует `isinstance` для проверки конкретного типа (например, `User`), при вызове функции с передачей экземпляра `Supervisor` (аргумент с именем `user`) проверка `isinstance(user, User)` все равно будет успешно пройдена.

СОПРОВОЖДАЕМОСТЬ Для повышения надежности используйте `isinstance` при проверке типа объекта, так как эта функция проверяет не только непосредственный класс объекта, но и его подклассы.

10.1.3. Обобщенная проверка типа объекта

В листингах 10.1 и 10.2 предполагалось, что в аргументе `by_urgency` передается либо `int`, либо `list`. Но представьте, что пользователь попытается вызвать функцию `filter_tasks` в виде `filter_tasks(tasks, (4, 5))`. Иначе говоря, вместо `list` пользователь вызовет функцию с объектом `tuple`. Итак, если вы хотите обеспечить большую гибкость вашей функции, проверка типа аргумента по одному конкретному варианту оказывается ограниченной. В этом разделе вы увидите, как получить информацию типа для объекта на более общем уровне.

Вы уже знаете, что при проверке типа объекта функции `isinstance` следует отдавать предпочтение перед `type`. Более того, `isinstance` можно передать сразу несколько возможных классов. В листинге 10.3 представлено рабочее решение для проверки аргумента `by_urgency` функции `filter_tasks` по нескольким возможным классам.

Листинг 10.3. Проверка типа объекта по нескольким классам с использованием isinstance

```
def filter_tasks(tasks, by_urgency):
    if isinstance(by_urgency, (list, tuple)):
        filtered = [x for x in tasks if x.urgency in by_urgency]
    else:
        filtered = [x for x in tasks if x.urgency == by_urgency]
    return filtered
```

Как и следовало ожидать, обновленная функция `filter_tasks` поддерживает как тип `list`, так и тип `tuple` для аргумента `by_urgency`. Однако также нельзя исключать, что пользователь захочет вызвать функцию для объекта множества `set`: `filter_tasks(tasks, {4, 5})`. Текущая реализация с таким вызовом не справится. Теоретически можно добавить `set` в вызов функции `isinstance`. Проблема в том, что для аргумента `by_urgency` используются многие другие типы данных, сходные с `list`, например `Series` из библиотеки pandas. Таким образом, перечислить все эти типы один за другим просто нереально, особенно если учесть возможность определения пользовательских классов. Необходим механизм обобщенной проверки типа объекта.

В стандартной библиотеке модуль `collections.abc` определяет несколько *абстрактных базовых классов* (Abstract Base Classes — отсюда `abc`), которые используются для проверки наличия нужных атрибутов или методов в конкретном классе. В программировании эта концепция известна под названием *интерфейса*.

ОСНОВНЫЕ ПОНЯТИЯ В ООП *интерфейс* представляет определенные атрибуты, функции, методы, классы и другие компоненты сущности (например, класса или пакета), которые используются разработчиками.

Важным для рассмотрения этой темы является абстрактный класс `Collection`, который должен содержать три ключевых специальных метода: `__contains__` (для проверки существования элемента: `item in obj`), `__iter__` (для преобразования в итератор: `iter(obj)`) и `__len__` (для проверки количества элементов: `len(obj)`). `list`, `tuple`, `set` и многие другие типы контейнеров данных, включая `Series`, реализуют эти методы, и все они являются конкретными (то есть неабстрактными) классами `Collection`. Соответственно, функцию `filter_tasks` можно обновить так, чтобы она имела более общую природу в отношении проверки типа аргумента `by_urgency`, как показано в листинге 10.4.

Листинг 10.4. Проверка типа объекта с использованием абстрактного класса

```
from collections.abc import Collection

def filter_tasks(tasks, by_urgency):
    if isinstance(by_urgency, Collection):
        filtered = [x for x in tasks if x.urgency in by_urgency]
    else:
        filtered = [x for x in tasks if x.urgency == by_urgency]
    return filtered
```

Использование абстрактного класса `Collection` позволяет охватить все типы данных, сходные с коллекциями, без явного перечисления всех разнообразных классов, которые могут передаваться пользователем. Это улучшает гибкость кода.

Как видно из разделов выше, мы постепенно улучшаем гибкость нашей функции, проверяя тип аргумента при помощи `type` и `isinstance` для одного типа, `isinstance` — для нескольких определенных типов и `isinstance` — для обобщенного типа. На рис. 10.1 приведена наглядная сводка этих вариантов использования.

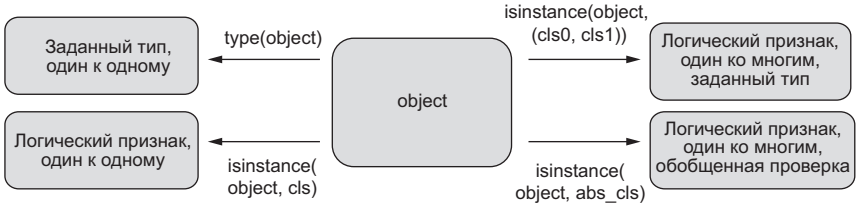


Рис. 10.1. Проверка информации о типе объекта с использованием `type` и `isinstance`. Класс `cls`, как и `cls0` и `cls1`, относится к конкретным классам, тогда как `abs_cls` — абстрактный класс, который может представлять неограниченное количество классов, использующих интерфейс

10.1.4. Обсуждение

Проверка типа объекта — важнейшая задача при интроспекции объектов. Существует слишком много других методов интроспекции, чтобы полностью описать их здесь. Когда разработчик начинает использовать новую, еще не знакомую ему библиотеку, то вместо поиска информации в интернете следует выполнить вызов `dir(obj)`, который вернет информацию обо всех доступных атрибутах и методах объекта.

Модуль `collections.abc` содержит много других абстрактных базовых классов. К числу таких абстрактных классов относится класс последовательности `Sequence`, а `list` является конкретным потомком `Sequence`. Другой абстрактный класс `Iterable` определяет интерфейс `__iter__`.

10.1.5. Задача

В листинге 5.1 мы определили следующую функцию для проверки того, является ли объект итерируемым:

```
def is_iterable(obj):
    try:
        _ = iter(obj)
    except TypeError:
        print(type(obj), "is not an iterable")
    else:
        print(type(obj), "is an iterable")
```

Ранее упоминалось о том, что `Iterable` является абстрактным классом из модуля `collections.abc`. Сможете ли вы переписать функцию `is_iterable` с использованием класса `Iterable`?

> **ПОДСКАЗКА** Если объект является итерируемым, его класс должен реализовать `__iter__` и иметь соответствующий интерфейс для класса `Iterable`.

10.2. КАК ВЫГЛЯДИТ ЖИЗНЕННЫЙ ЦИКЛ ОБЪЕКТА

С ростом масштабов проекта вы определяете собственные пользовательские классы. Когда вы учитесь реализовывать пользовательские классы (главы 8 и 9), вам встречаются разные термины, относящиеся к их созданию. Каждый Python-разработчик должен хорошо знать жизненный цикл объектов; это фундаментальный навык, который позволяет ему правильно управлять экземплярами классов.

В этом разделе рассматриваются ключевые события жизненного цикла экземпляра на нескольких примерах. При этом будут представлены термины, описывающие важнейшие концепции программирования, которые необходимо знать для эффективного общения с другими разработчиками. Некоторые из этих терминов рассматривались в главе 8; я кратко опишу их здесь в контексте жизненного цикла объекта.

10.2.1. Инстанцирование объекта

Жизненный цикл объекта начинается с его создания; эта процедура называется *инстанцированием* (instantiation). Процесс инстанцирования рассматривается в разделе ниже.

Для некоторых встроенных типов данных (как `str` и `list`) экземпляры могут создаваться на основе литералов, например, `"Hello, World!"` для экземпляра `str` и `[1, 2, 3]` для экземпляра `list`. Кроме создания встроенных типов данных на базе литералов, существует более общий способ создания экземпляра с использованием конструктора класса. Возьмем следующий класс `Task` (представлена минимальная реализация, чтобы мы сосредоточились на самых важных аспектах):

```
class Task:
    def __new__(cls, *args):
        new_task = object.__new__(cls)
        print(f"__new__ is called, creating an instance at {id(new_task)}")
        return new_task
    def __init__(self, title):
        self.title = title
        print(f"__init__ is called, initializing an instance
        ➥ at {id(self)}")
```

В классе `Task` кроме метода `__init__` реализуется метод `__new__`. Обычно метод `__new__` не реализуется, так как происходящее в нем не представляет особого интереса. В данном случае в `__new__` и `__init__` были включены вызовы функции `print`, чтобы вы видели, где вызывается каждая функция. Очень важно, что выводимое сообщение включает адрес экземпляра в памяти (полученный при помощи функции `id`), по которому отслеживаются объекты. Посмотрим, что происходит при создании экземпляра:

```
task = Task("Laundry")

# Выводимые строки:
__new__ is called, creating an instance at 140557771534976
__init__ is called, initializing an instance at 140557771534976

print("task memory address:", id(task))
# Вывод: task memory address: 140557771534976
```

На вашем компьютере адрес будет другим

При вызове конструктора `Task` сначала вызывается метод `__new__`, который создает экземпляр без присваивания каких-либо атрибутов; пока что это совершенно новый объект, на что указывает его имя. На этом шаге выделяется блок памяти для хранения объекта, что позволяет получить адрес экземпляра в памяти.

На следующем шаге вызывается метод `__init__`, на этом шаге задаются значения атрибутов только что созданного экземпляра для завершения процесса инициализации. Как показывает совпадение адресов памяти, в `__new__`, `__init__`

и созданной переменной `task` мы имеем дело с тем же объектом. На рис. 10.2 все этапы процесса создания экземпляра собраны воедино.

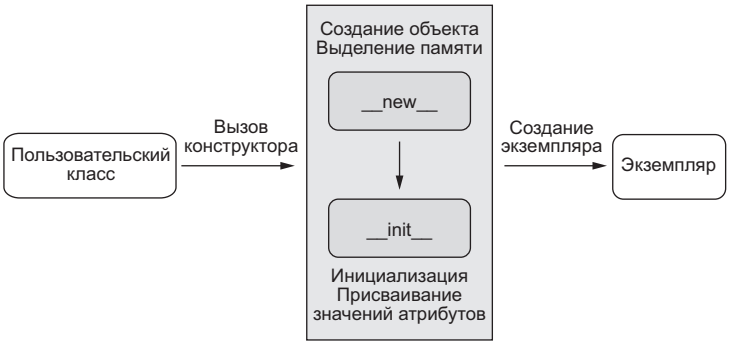


Рис. 10.2. Процесс инстанцирования пользовательского класса. После вызова конструктора пользовательского класса во внутренней реализации последовательно вызываются методы __new__ и __init__; __new__ создает новый объект, а __init__ завершает процесс инициализации. Конечным результатом процесса становится создание экземпляра

10.2.2. Активность в пространствах имен

Экземпляр создается вызовом конструктора класса, после чего его можно использовать. В этом разделе представлена концепция пространств имен. Вы увидите, что созданный экземпляр становится активным в соответствующем пространстве имен, это и позволяет его использовать.

Экземпляр класса `Task` создается командой `task = Task ("Laundry")`, где переменная `task` представляет экземпляр. Допустим, позднее в коде вам потребовалось получить атрибут `title` объекта `task`:

```
title_output = f"Title: {task.title}"
```

Когда вы пишете эту строку кода, неявно предполагается, что переменная `task` ссылается на определенную нами переменную — экземпляр класса `Task`. Но когда Python пытается выполнить эту строку кода, он не знает о наших предположениях; ему необходим механизм обнаружения переменной `task` для создания f-строки. Механизм поиска переменных учитывает *пространства имен*, в которых отслеживаются определяемые в программе переменные.

> **ОСНОВНЫЕ ПОНЯТИЯ** *Пространство имен* (namespace) работает по тому же принципу, что и словарь: в нем отслеживаются переменные, определенные в этом пространстве. Пространство имен помогает получить информацию об используемых переменных.

Допустим, класс `Task` определен и экземпляр `task` создается в том же файле Python, который образует модуль. В этом модуле существует *глобальное пространство имен*, в котором отслеживаются все переменные, и для проверки этих переменных вызывается функция `globals`:

```
print(globals())
# output the following data:
{'__name__': '__main__', '__doc__': None, '__package__': None,
➥ '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
➥ '__spec__': None, '__annotations__': {}, '__builtins__':
➥ <module 'builtins' (built-in)>, 'Task': <class '__main__.Task'>,
➥ 'task': <__main__.Task object at 0x7fd6280af280>}
```

Пространства имен можно рассматривать как словари, в которых активные переменные являются ключами, а соответствующие объекты — значениями. В приведенном примере выделены две переменные: класс `Task` и экземпляр `task`. После того как вы определили класс и создали экземпляр, оба объекта включаются в пространство имен и будут найдены каждый раз, когда мы используем эти переменные. Следующие проверки показывают, что значения `'Task'` и `'task'` действительно являются классом и экземпляром:

```
assert Task is globals()["Task"]
assert task is globals()["task"]
```

После того как экземпляр создан, вы можете использовать его в коде, причем его имя доступно при поиске по глобальному пространству имен, в котором созданный экземпляр был зарегистрирован.

10.2.3. Подсчет ссылок

Если объект активен в пространстве имен, Python в целях управления памятью следит за тем, сколько других объектов хранят ссылки на этот объект. Эти важные события происходят незаметно для пользователя. Аналогичная функциональность существует во многих современных ООП-языках. В этом разделе рассматривается механизм отслеживания ссылок.

Объем памяти, установленной на компьютере, ограничен. Во время работы приложения создают объекты, которые эту память расходуют. Чем больше объектов вы создаете, тем больше памяти использует приложение. Если бесконтрольно продолжать создавать объекты, вся память может быть исчерпана и в ваших приложениях начнутся сбои, возможно, даже случится зависание системы. Следовательно, в приложениях должен существовать механизм удаления неиспользуемых объектов из памяти. Таким механизмом является *подсчет ссылок* (reference counting).

Различия между объектами и переменными

Чтобы понять, как работает подсчет ссылок, сначала необходимо разобраться, чем объекты отличаются от переменных. При выполнении команды `task = Task("Laundry")` происходят два события:

- Создается экземпляр, то есть фактический объект, и связанные с ним данные сохраняются в памяти.
- Ссылка на объект сохраняется в переменной `task` — метке, которая используется для обращения к объекту в памяти.

Важно, что связь между объектом и меткой изменяема. В Python как в языке с динамической типизацией с той же меткой можно связать другой объект. При этом объект, ранее связанный с меткой, остается в памяти, но теперь метка ссылается на новый объект (рис. 10.3).

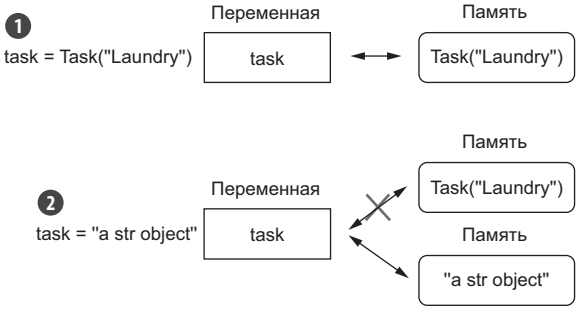


Рис. 10.3. Отношения между объектами и переменными. В команде присваивания экземпляр класса Task создается в памяти и этот объект связывается с переменной task. Позднее переменной task присваивается объект str. Повторное присваивание разрывает предшествующую связь между task и фактическим объектом Task(“Laundry”) и создает новую связь между task и объектом str

Как показано на рис. 10.3, мы создаем переменную с именем `task` и присваиваем ей экземпляр класса `Task`, чтобы переменная `task` ссылалась на экземпляр `Task`. Когда мы связываем ту же переменную `task` с другим объектом `str`, переменная `task` перестает ссылаться на экземпляр `Task`; вместо этого она ссылается на объект `str`.

Увеличение и уменьшение счетчиков ссылок

Теперь вы понимаете, в чем состоит различие между объектами и переменными, и знаете, что переменная представляет ссылку на фактический объект, хранящийся в памяти. Счетчик ссылок для такого объекта изначально равен 1 (для исходного присваивания). В этом разделе вы узнаете, как изменяется счетчик ссылок.

Прежде чем изменять счетчик ссылок, следует выяснить, как получить его текущее значение. В Python для этого используется функция `getrefcount` из модуля `sys`:

```
import sys

task = Task("Laundry")

assert sys.getrefcount(task) == 2
```

В этом примере используются две ссылки на экземпляр `Task`. Но погодите, разве здесь не должна быть только одна ссылка — переменная `task` из присваивания? Дело в том, что использование переменной в вызове функции `getrefcount` создает другую ссылку на объект, в результате чего текущий счетчик ссылок увеличивается до 2. В общем случае использование переменной при вызове функции увеличивает счетчик ссылок используемого объекта.

Теперь вы знаете, как узнать текущее значение счетчика ссылок объекта, и мы немного поэкспериментируем с изменением счетчика. Один из распространенных способов увеличения счетчика — включение переменной в контейнер данных (например, объект `dict` или `list`):

```
work = {"to_do": task}
assert sys.getrefcount(task) == 3

tasks = [task]
assert sys.getrefcount(task) == 4
```

В обоих случаях использование `task` в `dict` и `list` увеличивает счетчик ссылок на 1. Вы уже видели, как увеличивается счетчик ссылок; пора узнать, как он уменьшается. Обычно для этого используется команда `del`:

```
del tasks

assert sys.getrefcount(task) == 3
```

Вызов `del` для `tasks` удаляет ссылку на экземпляр; таким образом, счетчик ссылок уменьшается на 1. Также можно удалить `work`, чтобы уменьшить счетчик ссылок еще на 1, но вызывать `del` раз за разом слишком утомительно. Вместо того чтобы удалять объект `dict`, заменим `task` непосредственно в объекте `work` другим значением; это приведет к удалению ссылки на экземпляр `Task`:

```
work["to_do"] = "nothing"

assert sys.getrefcount(task) == 2
```

Как видите, Python моментально изменяет счетчик ссылок за вас. Но для чего нужны все эти манипуляции с подсчетом ссылок? Продолжим исследовать жизненный цикл объекта.

10.2.4. Уничтожение объекта

В разделе 10.2.3 описан механизм отслеживания счетчиков ссылок в Python. У него есть одна ключевая особенность: когда счетчик ссылок объекта уменьшается до 0, Python уничтожает объект и освобождает занимаемую им память для использования системой. В этом разделе процесс уничтожения рассматривается более подробно.

Как и процесс конструирования, процесс уничтожения обычно обрабатывается Python с применением автоматического подсчета ссылок. Чтобы получить

информацию о процессе уничтожения, следует переопределить `__del__` — специальный метод, относящийся к уничтожению объекта, — как показано в листинге 10.5.

Листинг 10.5. Переопределение __del__ в классе

```
class Task:
    def __init__(self, title):
        print(f"__init__ is called, initializing an instance
        ➥ at {id(self)}")
        self.title = title

    def __del__(self):
        print(f"__del__ is called, destructing an instance at {id(self)}")
```

Небольшой фрагмент с обновленным классом `Task` поможет понять, что происходит при инициализации и как работает глобальное пространство имен:

```
task = Task("Homework")
# Вывод: __init__ is called, initializing an instance at 140557504542416
assert "task" in globals()
```

Чтобы вручную обнулить счетчик ссылок для запуска процесса уничтожения объекта, воспользуйтесь командой `del`:

```
del task
# Вывод: __del__ is called, destructing an instance at 140557504542416
assert "task" not in globals()
```

Как видите, при вызове `del` для `task` вызывается специальный метод `__del__`. По адресу памяти можно определить, что удаляется именно тот экземпляр, который был создан. И действительно, после уничтожения `"task"` также удаляется из пространства имен и переменная `task` становится недоступной. Если вы все же попытаетесь это сделать, произойдет ошибка:

```
title_output = f"Title: {task.title}"
# ERROR: NameError: name 'task' is not defined. Did you mean: 'Task'?
```

10.2.5. Обсуждение

В этом разделе рассматриваются главные события в жизненном цикле объекта на примере экземпляра пользовательского класса. На рис. 10.4 приведена общая схема жизненного цикла объекта, объединяющая все сказанное.

Самое замечательное при работе с объектами в Python заключается в том, что эти события в основном обрабатываются автоматически — Python берет на себя всю черную работу. Если только вы не строите приложение, интенсивно расходующее память, вам не придется беспокоиться об этих событиях. Однако эти концепции играют фундаментальную роль в ООП, и если вы также изучаете другой ООП-язык, эта информация ускорит процесс изучения.

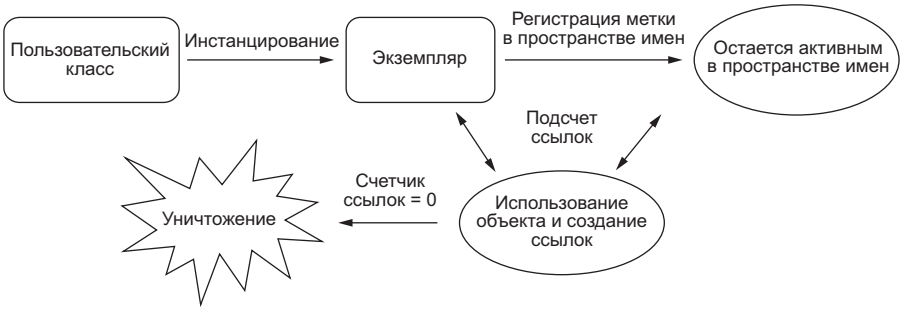


Рис. 10.4. Ключевые события жизненного цикла объекта. Существование объекта начинается с его конструирования, после чего он активизируется в соответствующем пространстве имен. В процессе использования объекта Python автоматически изменяет счетчик ссылок. Когда на объект не остается ни одной ссылки, Python уничтожает объект, а занимаемая им память снова становится доступной

Я еще не упоминал о важнейшем модуле `gc`, имя которого представляет собой сокращение от Garbage Collection (то есть «сборка мусора»). Модуль содержит продвинутые алгоритмы для управления памятью при использовании механизма подсчета ссылок. Например, подсчет ссылок уничтожает объекты при возникновении циклических ссылок. Этот проблемный сценарий возникает тогда, когда два и более объекта содержат ссылки друг на друга, а их счетчики ссылок не достигают 0. Заинтересованные читатели могут исследовать модуль `gc`, чтобы узнать, как решаются проблемы, подобные циклическим ссылкам.

10.2.6. Задача

Джеймс, изучающий программирование на языке Python, интересуется тем, как работает подсчет ссылок для экземпляров пользовательских классов. У него возник вопрос. Допустим, он создает переменную экземпляра, например `task = Task("Homework")`, и знает, что счетчик ссылок для используемого объекта равен 1 для переменной `task`. Приводит ли использование переменной `task` в функции к увеличению ее счетчика ссылок? Напишите код, который показывает, что при этом происходит.

> **ПОДСКАЗКА** Чтобы проверить счетчик ссылок аргумента, включите в функцию вызов `getrefcount`.

10.3. КАК СКОПИРОВАТЬ ОБЪЕКТ

При работе с объектом вы изменяете его атрибуты. Однако иногда требуется сохранить исходные атрибуты на случай, если вы захотите отменить изменение. Эта необходимость встречается во многих приложениях. В частности, в нашем

таск-менеджере пользователь может редактировать существующую задачу. После внесения некоторых изменений пользователь либо сохранит обновление, либо отменит все правки. В такой ситуации мы создаем копию исходной задачи, чтобы иметь новую копию для хранения обновлений и оригинал как резервную копию. В этом разделе вы узнаете, как правильно копировать объекты.

10.3.1. Создание (поверхностной) копии

В языке Python модуль `copy` предоставляет функциональность, связанную с копированием объектов. В этом разделе вы узнаете, как создать копию. Точнее, мы обсудим создание поверхностных (shallow) копий в отличие от глубоких (deep); различия между этими двумя процессами рассматриваются в разделе 10.3.2.

Допустим, вы создали следующий класс `Task` для своего приложения. Для простоты предположим, что в классе реализуются только методы `__init__` и `__repr__`:

```
class Task:
    def __init__(self, title, desc):
        self.title = title
        self.desc = desc

    def __repr__(self):
        return f"Task({self.title!r}, {self.desc!r})"

    def save_data(self):
        # Обновление базы данных
        pass
```

В приложении пользователь просматривает список задач и при желании редактирует конкретную задачу. Например, он может внести изменения в следующий экземпляр `Task`:

```
task = Task("Homework", "Math and physics")
```

Если пользователь доволен правкой, то обновленная задача будет сохранена, а если отменил правку, то остаются данные из исходной задачи. Так как экземпляр класса `Task` представляется в виде словаря, наивный подход к созданию копии основан на использовании объекта `dict` в качестве «неформальной» копии исходного экземпляра:

```
task_dict = task.__dict__

task_dict_copied = task_dict.copy()

print(task_dict_copied)
# Вывод: {'title': 'Homework', 'desc': 'Math and physics'}
```

Как показано в этом примере, мы получаем представление в форме `dict` с использованием `__dict__`. Для этого объекта `dict` можно создать копию с использованием метода экземпляра `copy`. Когда пользователь редактирует задачу, мы используем объект `dict` для отслеживания изменений. Однако такое решение

усложняется тем, что после того, как объект `dict` будет обновлен, его необходимо вернуть экземпляру `Task`, чтобы использовать дополнительную функциональность, реализованную классом `Task`. В противном случае вы практически ничего не сможете сделать с объектом `dict`, так как функциональность, связанная с задачами (например, `save_data`), будет недоступна.

Вместо создания копии представления экземпляра в виде словаря следует скопировать его напрямую средствами, доступными в модуле `copy`. Следующий фрагмент кода демонстрирует более эффективное решение с реальным копированием экземпляра:

```
from copy import copy

task_copied = copy(task)

print(task_copied)
# Вывод: Task('Homework', 'Math and physics')
```

> **ОБРАТИТЕ ВНИМАНИЕ** Обратите внимание: имя функции `copy` совпадает с именем модуля `copy`. Это не единственный случай, в котором имя функции совпадает с именем модуля. Так, в модуле `datetime` присутствует функция `datetime`, поэтому в программах иногда встречается команда `from datetime import datetime`.

Функция `copy` импортируется из модуля `copy`, после чего экземпляр `task` передается функции `copy`. Из приведенных результатов видно, что скопированная переменная `task_copied` содержит те же данные, что и `task`, то есть является копией исходной задачи. Со скопированной задачей после внесения правок пользователем вызывается функция `task_copied.save_data()` для обновления базы данных.

10.3.2. Потенциальные проблемы с поверхностным копированием

В начале раздела 10.3.1 я упоминал о том, что существуют две разновидности копий: поверхностные и глубокие. Функция `copy` создает поверхностную копию. Но что такое поверхностная копия и что такое глубокая копия? В этом разделе будут продемонстрированы различия между этими видами копий, а также описаны потенциальные проблемы, которые могут возникать из-за поверхностного копирования.

Допустим, что в нашем таск-менеджере каждой задаче назначается тег. Для этого класс `Task` может выглядеть так:

```
class Task:
    def __init__(self, title, desc, tags = None):
        self.title = title
```

```
        self.desc = desc
        self.tags = [] if tags is None else tags
    def __repr__(self):
        return f"Task({self.title!r}, {self.desc!r}, {self.tags})"

    def save_data(self):
        pass
```

Тернарное присваивание

ОСНОВНЫЕ ПОНЯТИЯ *Тернарное выражение* (ternary expression) вычисляется на основании результата логического условия и имеет формат `значение_для_true if условие else значение_для_false`. Когда вы используете тернарное выражение для присваивания, этот процесс называется тернарным присваиванием.

После обновления класса мы создадим экземпляр и его копию при помощи функции `copy` (листинг 10.6).

Листинг 10.6. Создание копии существующей задачи

```
task = Task("Homework", "Math and physics", ["school", "urgent"])

task_copied = copy(task)

print(task_copied)
# Вывод: Task('Homework', 'Math and physics', ['school', 'urgent'])
```

В приложении пользователь начинает обновлять задачу, а именно добавляет к задаче новый тег:

```
task_copied.tags.append("red")

print(task_copied)
# Вывод: Task('Homework', 'Math and physics', ['school', 'urgent', 'red'])
```

Как видите, теги скопированной задачи обновляемы. Но затем пользователь решает отменить правку. В этой ситуации мы все еще используем данные исходной задачи. Так как исходная задача не изменялась, ее данные должны оставаться прежними:

```
print(task)
# Вывод: Task('Homework', 'Math and physics', ['school', 'urgent', 'red'])
```

Исходная задача содержит теги `['school', 'urgent']`, но почему она изменилась? Объект `list` соответствует объекту `list` в скопированной задаче. Конечно, это не случайное совпадение. Похоже, `task` и `task_copied` используют для `tags` один и тот же объект `list`. Эту гипотезу нетрудно проверить:

```
assert task.tags is task_copied.tags

assert id(task.tags) == id(task_copied.tags)
```

Проверка равенства при помощи is или ==

Возможно, вы заметили, что при сравнении двух объектов в Python я в одних случаях использую `is`, а в других — `==`. `is` проверяет, являются ли два объекта одним и тем же объектом, поэтому такая проверка иногда называется *проверкой тождественности*. А `==` проверяет, имеют ли два объекта одинаковое значение. Так как они предназначены для разных сравнений (тождественность и равенство значений), они должны использоваться по-разному. В типичной ситуации со сравнением объекта с `None`, например, следует использовать `is`, хотя, возможно, вам встречалось в программах использование `==`. `None` является одиночным объектом (синглетоном); это означает, что в приложении существует только один объект, содержащий `None`. Каждый раз, когда вы используете `None`, это один и тот же объект, к которому вы обращаетесь в памяти. Таким образом, сравнение объекта с `None` должно использовать `is`, так как сравнение должно быть проверкой тождественности. Та же проверка тождественности должна использоваться для сравнения `task.tags` и `task_copied.tags`.

Если же вы хотите сравнить адреса двух объектов `list` в памяти, используйте `==`. Каждый раз, когда вы вызываете функцию `id` для объекта, она создает объект `int` для обозначения адреса объекта в памяти. Таким образом, два вызова `id` создают два разных объекта `int`, и мы проверяем лишь то, что эти два объекта `int` имеют одинаковые значения.

Как следует из примера, обе проверки (тождественности и адресов в памяти) подтверждают нашу гипотезу о том, что объект `list` из атрибута `tags` экземпляра `task_copied` является тем же объектом, что и объект из `task`. Почему это могло произойти? Совместное использование объекта `list` подчеркивает различия между поверхностным и глубоким копированием. При поверхностном копировании копируется только внешний контейнер данных. При этом копии совместно используют такие внутренние изменяемые объекты, как объект `list` для `tags`. В то же время глубокое копирование не ограничивается копированием внешнего контейнера, но и создает рекурсивные копии внутренних объектов. Оба типа копирования оставляют внутренние неизменяемые объекты (такие, как строки и кортежи) без изменений, так как они все равно не могут оперировать этими объектами. На рис. 10.5 показано, чем глубокое копирование отличается от поверхностного.

На рис. 10.5 используется объект `list`, который содержит строку `"hello"` и список `[3, 4, 5]`. При создании поверхностной копии копируется только внешний объект `list`. Внутренний объект `list [3, 4, 5]` и неизменяемый объект `str "hello"` совместно используются поверхностной копией и исходным списком. А при создании глубокой копии внешний контейнер и его изменяемый элемент, внутренний объект `list`, копируются по отдельности.

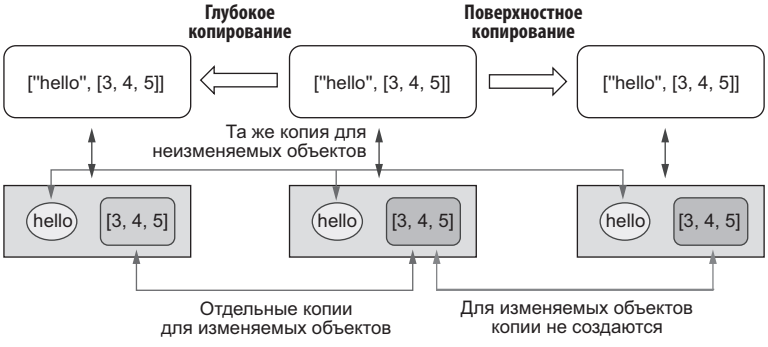


Рис. 10.5. Различия между поверхностными и глубокими копиями. При поверхностном копировании копируется внешний контейнер данных (или любой объект, не являющийся контейнером, например строка) и содержащиеся в нем неизменяемые объекты, но не внутренние изменяемые объекты (например, list). При глубоком копировании для внешнего контейнера и всех его внутренних объектов создаются отдельные копии. Серые прямоугольники представляют объекты в памяти

Из-за различий в том, как два вида копирования поступают с внутренними изменяемыми объектами, при создании только поверхностной копии появляется риск случайной перезаписи данных в исходном объекте. Следовательно, если вам нужны две реальные копии независимых объектов, следует создать глубокую копию, как показано в следующем разделе.

10.3.3. Создание глубокой копии

Теперь вы знаете разницу между поверхностными и глубокими копиями, и можно повторно вернуться к функциональности по редактированию задач в нашем приложении. Для этого две задачи, исходная и скопированная, должны быть полностью изолированными, то есть не имеющими общих внутренних изменяемых объектов — в нашем случае атрибута `tags`, — что позволяет свободно изменять атрибут `tags` без влияния на исходную задачу. После всего, что было сказано о поверхностных и глубоких копиях, логично сделать вывод, что эта функциональность требует создания глубокой копии.

Кроме функции `copy`, модуль `copy` содержит функцию `deepcopy`. Эта функция предназначена специально для создания глубокой копии объекта:

```
from copy import deepcopy

task = Task("Homework", "Math and physics", ["school", "urgent"])

task_deepcopied = deepcopy(task)

print(task_deepcopied)
# Вывод: Task('Homework', 'Math and physics', ['school', 'urgent'])
```

В этом коде функция deepcopy используется для создания копии исходной задачи. На этой стадии различия между поверхностной и глубокой копией еще не проявляются, потому что мы не выполнили никаких операций с внутренним изменяемым объектом. Пора продемонстрировать полезность глубокой копии:

```
task_deepcopied.tags.append("red")

print(task_deepcopied)
# Вывод: Task('Homework', 'Math and physics', ['school', 'urgent', 'red'])

print(task)
# Вывод: Task('Homework', 'Math and physics', ['school', 'urgent'])
```

В этом фрагменте обновляются данные атрибута tags глубокой копии задачи. Примечательно, что изменение существует в task_deepcopied, но не в task — такое поведение ожидаемо, потому что глубокая копия создает обособленную копию каждого внутреннего объекта, включая изменяемый список tags.

10.3.4. Обсуждение

Поверхностные и глубокие копии различаются своим поведением при копировании внутренних изменяемых объектов, обычно в форме контейнеров данных, таких как list, dict и set. Поверхностные копии не создают копию этих внутренних контейнеров данных, что может обеспечить экономию памяти, если вас не интересуют общие внутренние объекты. С другой стороны, когда предполагается создание копии с отдельными данными, например, если вы редактируете задачу и хотите сохранить ее исходные данные, — следует использовать глубокое копирование.

10.3.5. Задача

В рассмотренных примерах использовались функции copy и deepcopy из модуля copy. При их вызове создаются поверхностная и глубокая копия соответственно. Однако в пользовательском классе можно переопределить два специальных метода, __copy__ и __deepcopy__, которые вызываются при использовании функций copy и deepcopy. Допустим, в случае переопределения __copy__ мы изменяем название скопированной задачи: "Homework" ->" Copied: Homework". Копия также должна содержать отдельную копию атрибута tags, то есть обладать поведением глубокой копии. Сможете ли вы реализовать эту функциональность?

> **ПОДСКАЗКА** Копирование экземпляра должно осуществляться на уровне экземпляра, так что метод __copy__ должен быть методом экземпляра. В теле метода нужно возвращать новый экземпляр с обновленным названием задачи и новым объектом list для tags.

10.4. КАК ОБРАТИТЬСЯ К ПЕРЕМЕННОЙ И ИЗМЕНИТЬ ЕЕ В ДРУГОЙ ОБЛАСТИ ВИДИМОСТИ

В разделе 10.2 была представлена концепция пространств имен. Когда вы определяете класс (например, `Task`) в модуле Python (файл `.py`), класс регистрируется в глобальном пространстве имен, которое реализовано в форме словаря: идентификаторы являются ключами, а соответствующие объекты — значениями. Допустим, в таск-менеджере имеется модуль, находящийся в файле с именем `task.py`. Код, содержащийся в этом файле, приведен в листинге 10.7.

Листинг 10.7. Попытка изменения глобальной переменной

```
db_filename = "N/A"

def set_database(db_name):
    db_filename = db_name

set_database("tasks.sqlite")

print(db_filename)
# Вывод: "N/A"
```

В листинге 10.7 определена переменная `db_filename`, в которой хранится путь к файлу с таск-менеджером. Вызывая функцию `set_database`, мы присваиваем `db_name` значение `db_filename`. Однако в выводе `db_filename` видим значение `"N/A"`. Результат неожиданный, так как мы вроде бы изменили переменную. Что же произошло?

В этом разделе я покажу, как обратиться к переменной и изменить ее в этом сценарии. На более общем уровне эта разновидность задач связана с выполнением операций с переменными в других областях видимости, при этом особое внимание будет уделяться двум ключевым словам: `global` и `nonlocal`.

На этих примерах вы научитесь обращаться к переменным, при разрешении которых применяется правило LEGB.

10.4.1. Обращение к произвольной переменной: правило LEGB при поиске имен

Между областями видимости и пространствами имен существует тесная связь. Области видимости образуют границы пространств имен, а пространства имен образуют содержимое областей видимости. На рис. 10.6 изображены отношения между пространствами имен и областями видимости на примере модуля Python.

Как показано на рис. 10.6, пространство имен отслеживает все объекты, каждому из которых назначается собственный идентификатор в модуле. Таким образом, пространство имен можно рассматривать как контейнер, внутреннее

пространство которого заполнено разными объектами. Область видимости представляет собой внешнюю структуру контейнера, определяющую границы модуля.

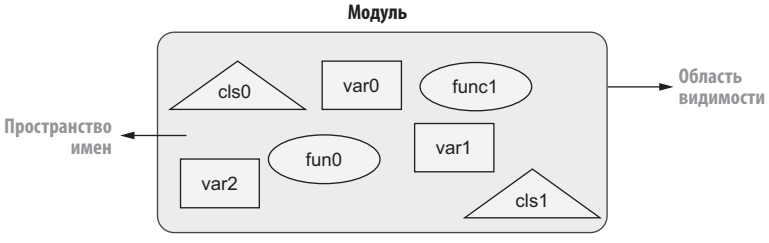


Рис. 10.6. Отношения между глобальным пространством имен и глобальной областью видимости. В модуле глобальное пространство имен отслеживает все переменные, а также все функции и классы, в форме словаря. Глобальное пространство имен находится в глобальной области видимости, определяющей границы модуля

Если интерпретировать код с точки зрения Python, то, встречая переменную, Python пытается разрешить ее, то есть найти объект, на который ссылается переменная. В разделе 10.2.2 упоминалось о том, что Python ищет переменные в пространстве имен, связанном с областью видимости. Порядок поиска проходит через несколько уровней областей видимости, которые обозначаются сокращением LEGB.

ОСНОВНЫЕ ПОНЯТИЯ *Правило LEGB* определяет порядок разрешения переменных в Python: от локальной области видимости (L, local) к охватывающей (E, enclosing), глобальной (G, global) и встроенной (B, built-in).

Правило LEGB означает последовательный переход от локальной через охватывающую и глобальную к встроенной области видимости. Модуль образует глобальную область видимости. Встроенная область видимости содержит пространства имен для всех встроенных функций и классов. В модуле можно определить класс или функцию, которые образуют локальную область видимости.

ОБРАТИТЕ ВНИМАНИЕ На первый взгляд обозначение области видимости модуля как глобальной выглядит немного странно. Но если вспомнить, что функция внутри модуля создает локальную область видимости, вполне логично, что область видимости, превышающая локальную, называется глобальной. Это сравнение поможет вам запомнить разницу.

Но как насчет охватывающей области видимости? Когда я описывал декораторы в разделе 7.3, функции определялись внутри других функций (как вложенные). Для внутренней функции локальная область видимости внешней функции называется *охватывающей областью видимости* (enclosing scope). На рис. 10.7 показано, как происходит разрешение переменных/функций (объединяемых общим термином «имена») посредством поиска в конкретной области видимости.

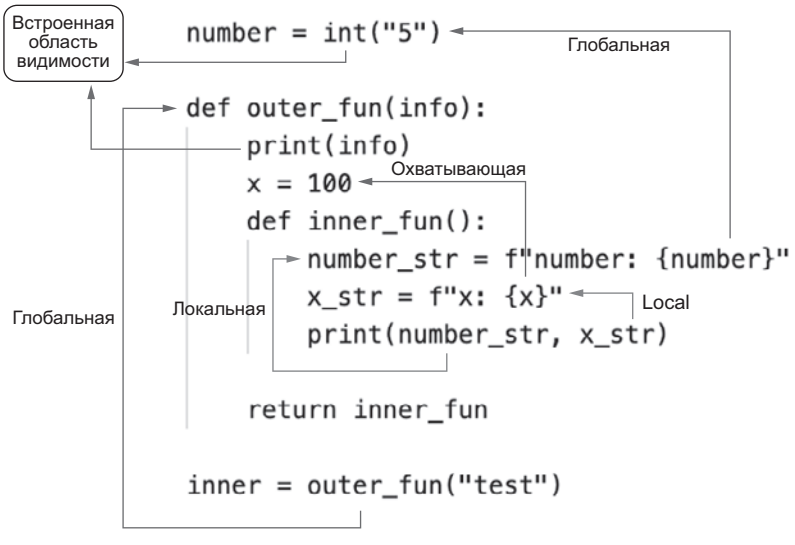


Рис. 10.7. Примеры разрешения переменных. Такие функции, как int и print, называются встроенными функциями, и их разрешение производится поиском во встроенной области видимости. Переменная number и функция outer_fun разрешаются в глобальной области видимости. Переменная x используется в функции inner_fun, которая разрешается в охватывающей области видимости. number_str и x_str разрешаются в локальной области видимости

Правило LEGB описывает последовательный порядок разрешения переменных. Как показано на рис. 10.8, для переменной (а также для имени вообще или для имени как идентификатора, который может относиться к функции, списку и даже классу) Python сначала проводит поиск в локальной области видимости. Если имя удается успешно разрешить, используется соответствующее значение. В противном случае Python продолжает поиск в охватывающей области видимости. Если имя будет разрешено, используется значение — и далее поочередно для глобальной и встроенной области видимости. Если имя так и не удается разрешить после того, как Python проверит все эти области видимости, выдается исключение `NameError`.

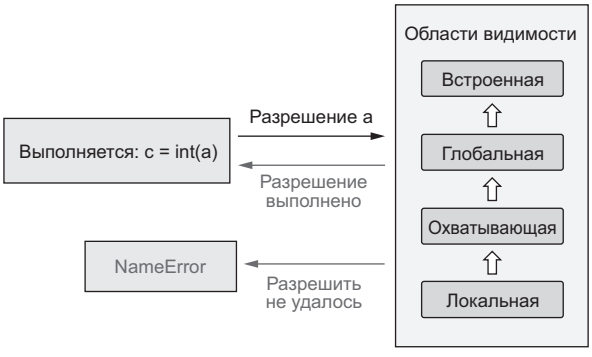


Рис. 10.8. Общий процесс разрешения переменной: правило LEGB. Когда Python встречает переменную, он пытается разрешить эту переменную, для чего проводит поиск в локальной, охватывающей (если применимо), глобальной и встроенной областях видимости соответственно. Если переменная будет разрешена успешно, Python использует значение; в противном случае выдается исключение NameError

10.4.2. Изменение глобальной переменной в локальной области видимости

В начале этого раздела я описал проблему, когда вызов функции `set_database` не изменял переменную `db_filename`. В разделе 10.4.1 вы узнали, что `db_filename` представляет глобальную переменную, а функция `set_database` образует локальную область видимости. Таким образом, проблему можно обобщить как изменение глобальной переменной в локальной области видимости; эта тема рассматривается в данном разделе.

Прежде чем приводить решение, присмотритесь к коду в листинге 10.7. Обратите внимание на вызов функции `print`, который показывает, какие сущности доступны в локальной области видимости функции:

```
db_filename = "N/A"

def set_database(db_name):
    db_filename = db_name
    print(list(locals()))
```

Для первой команды присваивания (`db_filename = "N/A"`) создается переменная с именем `db_filename` в глобальной области видимости. Затем в нескольких следующих строках определяется функция `set_database`. При проверке глобальной области видимости ожидается, что она включает как `db_filename`, так и `set_database`:

```
print(list(globals()))
# output: ['__name__', '__doc__', '__package__', '__loader__', '__spec__',
➥ '__annotations__', '__builtins__', 'db_filename', 'set_database']
```

В теле функции `set_database` особого внимания требует код `db_filename = db_name`, целью которого является обновление глобальной переменной `db_filename`. Однако вывод листинга 10.7 показывает, что этого не происходит.

Прежде чем искать причину, сделаем еще одно наблюдение. Возможно, вы заметили, что я включил еще одну строку кода: `print(list(locals()))`, которая генерирует зарегистрированные объекты в локальной области видимости функции `set_database`. Вызов этой функции позволяет ознакомиться с содержимым локального пространства имен:

```
set_database("tasks.sqlite")
# Вывод: ['db_name', 'db_filename']
```

Локальное пространство имен функции `set_database` содержит две переменные: `db_name` и `db_filename`. Когда Python выполняет строку кода `db_filename = db_name`, как именно работает правило LEGB при разрешении `db_filename` и `db_name` соответственно?

Переменная `db_name` существует только в локальной области видимости, и она разрешается как аргумент, который используется при вызове функции. Что касается `db_filename`, переменная с таким именем присутствует как в локальной, так и в глобальной области видимости, но в соответствии с правилом LEGB используется переменная из локальной области. Так как переменная из локальной области видимости не имеет зарегистрированного значения, Python интерпретирует эту строку кода как команду присваивания для создания новой переменной, а не как обновление существующей глобальной переменной.

Теперь вы знаете, что произошло в программе, и вам будет проще понять решение: использование ключевого слова `global` для обозначения того, что конкретная переменная является глобальной, а не локальной, как показано в листинге 10.8.

Листинг 10.8. Успешное изменение глобальной переменной

```
db_filename = "N/A"

def set_database(db_name):
    global db_filename
    db_filename = db_name
    print(list(locals()))

set_database("tasks.sqlite")
# Вывод: ['db_name']

print(db_filename)
# Вывод: tasks.sqlite
```

В теле функции `set_database` перед присваиванием мы указываем, что переменная `db_filename` является глобальной, чтобы локальная область видимости не регистрировала это имя заново. Затем выполняется присваивание. Python знает, что обновляется переменная `db_filename` из глобальной области видимости.

Затем мы проверяем обновленное значение (`tasks.sqlite`), выводя переменную `db_filename`, которая уже не содержит исходное значение `"N/A"`.

Обратите внимание: ключевое слово `global` используется только при попытке обновления глобальной переменной в локальной области видимости. Если вы используете глобальную переменную без присваивания или обновления, то ключевое слово `global` использовать не нужно, так как имя будет успешно разрешено через обращение к глобальной области видимости.

10.4.3. Изменение охватывающей переменной

В разделе 10.4.2 вы научились использовать ключевое слово `global` для изменения глобальной переменной в локальной области видимости. Другое ключевое слово, `nonlocal`, изменяет охватывающую (enclosing) переменную в локальной области видимости. `nonlocal` используется не так часто, как `global`, так как глобальные области видимости встречаются повсеместно, а охватывающие существуют только в функциях, содержащих вложенные функции. По этой причине в данном разделе я ограничусь кратким описанием изменений охватывающих переменных. Чтобы вам было проще понять происходящее, мы воспользуемся простым примером кода из листинга 10.9.

Листинг 10.9. Изменение нелокальной переменной

```
def change_text(using_nonlocal: bool):
    text = "N/A"
    def inner_fun0():
        text = "No nonlocal"

    def inner_fun1():
        nonlocal text
        text = "Using nonlocal"

    inner_fun1() if using_nonlocal else inner_fun0()
    return text

change_text(using_nonlocal=False)
# Вывод: 'N/A'

change_text(using_nonlocal=True)
# Вывод: 'Using nonlocal'
```

В функции `change_text` определяется локальная переменная `text`. Две внутренние функции образуют собственные локальные пространства имен; для них область видимости функции `change_text` является охватывающей областью видимости. Эти две функции различаются тем, объявляется ли в них `text` как нелокальная переменная при помощи ключевого слова `nonlocal`. Используя ключевое слово `nonlocal`, вы приказываете Python использовать переменную `text` из охватывающей области видимости.

Из результатов видно, что вызов внутренней функции `inner_fun1` успешно изменяет нелокальную переменную `text`. А вот вызов `inner_fun0` не отражается на `text`, потому что Python интерпретирует присваивание `text = "No nonlocal"` как обычную команду присваивания вместо обновления нелокальной переменной.

10.4.4. Обсуждение

В разделе 10.4 рассматривается разрешение переменных в Python, а также функций и классов в соответствии с порядком LEGB (локальная → охватывающая → глобальная → встроенная). Если вы пишете код, в котором задействовано несколько разных пространств имен, помните, что переменные должны разрешаться в конкретных областях видимости. Из-за сложностей, связанных с порядком LEGB, не забывайте использовать ключевое слово `global`, если вам потребуется обновить глобальную переменную в локальной области видимости. Не совершайте глупых ошибок, полагая, что обновление можно произвести вызовом функции, как мы попытались сделать в листинге 10.7.

10.4.5. Задача

У Джона имеется опыт программирования на Swift — языке, используемом для создания приложений для macOS и iOS. В Swift инструкция `if...else...` может создавать область видимости, отдельную от глобальной. Как ему узнать, создает ли инструкция `if...else...` локальную область видимости в Python?

> **ПОДСКАЗКА** Создайте глобальную переменную и попытайтесь изменить ее в инструкции `if...else...` . Если локальная область видимости существует, то вы не сможете изменить значение переменной без ключевого слова `global`.

10.5. ЧТО ТАКОЕ ВЫЗЫВАЕМОСТЬ И ЧТО ОНА ОЗНАЧАЕТ

Python является языком ООП, поэтому все его структурные элементы — пакеты, модули, классы, функции и данные — представляются разными видами объектов. Таким образом, понимание характеристик объектов исключительно важно для написания качественного Python-кода. В разделе 3.1, когда мы рассматривали выбор между списками и кортежами, упоминались свойства хешируемости и изменяемости, относящиеся к способности объекта хешироваться и изменяться соответственно.

Наряду с хешируемостью и изменяемостью одной из ключевых характеристик объекта является *вызываемость* (callability), то есть возможность его вызова. Как и в большинстве современных языков, объекты в Python вызываются при помощи пары круглых скобок (оператор вызова). Таким образом, если объект

может использоваться с оператором вызова, можно сказать, что он является вызываемым; если же использование объекта с оператором вызова невозможно, то объект не является вызываемым. В Python существует встроенная функция `callable`, которая проверяет вызываемость объекта. Как известно, функции можно вызывать, поэтому логично ожидать, что они обладают свойством вызываемости:

```
def doubler(x):
    return 2 * x

assert callable(doubler)
```

Концепция вызываемости кажется элементарной, однако механизм вызываемости лежит в основе ряда ключевых возможностей Python. В этом разделе рассматриваются важные практические следствия вызываемости объекта.

10.5.1. Различия между классами и функциями

Класс можно вызвать: например, вызов `Task("Homework", "Math and physics")` создает экземпляр класса `Task`. Также можно вызвать функцию для выполнения определенной операции, например `print("Hello, World!")`. Следовательно, и классы, и функции являются вызываемыми, и из-за общей характеристики вызываемости становится сложнее отличить класс от функции. Часто приходится слышать, что в Python есть много полезных встроенных функций, таких как `list`, `range` и `sum`, но на самом деле не все они являются функциями. Первое следствие вызываемости связано с особенностями классов и функций.

> **ОСНОВНЫЕ ПОНЯТИЯ** *Вызываемость* (callability) означает, что к объекту может применяться операция вызова. Когда функция ожидает получить вызываемый объект (например, в аргументе `key` функции `sorted`), ей можно передать функцию или класс. Если у вас имеется пользовательский класс, реализующий `__call__`, экземпляр этого класса тоже может использоваться в качестве вызываемого объекта!

Многие из этих «функций» на самом деле функциями не являются. Например, это классы `bool`, `int` и `dict` — в отличие от `callable` и `hash`, которые являются функциями. Основная причина, по которой их путают, — общая характеристика вызываемости, но различия заметны с точки зрения семантики. Вызывая эти классы, вы получаете экземпляр класса: например, `bool` для получения объекта `bool` или `dict` для получения объекта `dict`.

> **ОБРАТИТЕ ВНИМАНИЕ** Имена встроенных классов записываются в нижнем регистре в отличие от схемы «верблюжьего регистра» для пользовательских классов. Встроенные типы записываются в нижнем регистре по историческим причинам — такая запись использовалась в ранних версиях Python.

А вот реальные функции не связываются с классами напрямую. Следовательно, при вызове таких функций вы не получаете одноименный экземпляр. Например, никто не рассчитывает получить объект `sum` при вызове `sum` или объект `hash` при вызове `hash`. В то же время вы получите объект `range` при вызове `range` или объект `slice` при вызове `slice`.

10.5.2. Снова о функции высшего порядка map

Одним из проявлений функционального программирования в Python являются *функции высшего порядка*, то есть функции, которые получают другие функции в аргументах или возвращают их. В разделе 7.2 была представлена функция высшего порядка `map`, но можно ли считать ее настоящей функцией? Интуиция подсказывает положительный ответ. Но интуиция порой подводит. Мы вернемся к функции `map` в этом разделе.

Самый простой способ проанализировать объект — передать его при вызове `print`. Ожидается, что пользовательская или встроенная функция действительно является функцией:

```
def do_something():
    pass

print(do_something)
# Вывод: <function do_something at 0x7fe8180f30a0>

print(sum)
# Вывод: <built-in function sum>
```

Если `map` на самом деле является функцией, должно быть выведено сообщение о том, что это встроенная функция (как в примере с `sum`). Посмотрим, так ли это:

```
print(map)
# output: <class 'map'>
```

Неожиданно, но `map` — не функция. Оказывается, это класс с именем `map`. В соответствии с тем, что `map` является классом, при вызове `map` создается объект `map`, как и в случае со встроенными классами `list` и `dict`:

```
print(map(int, ["1", "2.0", "3"]))
# Вывод: <map object at 0x7fe8180df700>
```

Ошибочное представление о `map` как о функции может происходить от предположения о том, что классы обычно получают не являющиеся функциями объекты при создании экземпляра. Однако следует помнить, что все функции Python — это объекты. Таким образом, класс `map` занимает особое место в том смысле, что при его конструировании в аргументах передаются функции.

10.5.3. Использование вызываемых объектов в аргументе key

Некоторые функции Python содержат параметр `key`, который используется для функций, выполняющих сортировку (`sorted`) или сравнения (например, `max`). В разделе 3.2 метод списка `sort` использует в `key` функцию; значит, можно предположить, что аргументом `key` может быть только функция. Однако в аргументе `key` может передаваться любой вызываемый объект, как будет показано в этом разделе.

Простейшая ситуация для использования класса вместо функции в аргументе `key` функции `sorted` — использование встроенного класса `str`. Допустим, вы хотите отсортировать список карт в покерной колоде. Без аргумента `key` сортировка завершится неудачей из-за невозможности сравнения целых чисел и строк:

```
cards = [10, 1, "J", "A"]

print(sorted(cards))
# ERROR: TypeError: '<' not supported between instances of 'str' and 'int'

print(sorted(cards, key=str))
# Вывод: [1, 10, 'A', 'J']
```

Так как `str` используется в качестве `key`, сортировка возможна, но порядок окажется неправильным: туз (`A`) должен быть больше валета (`J`). Решим проблему созданием класса `PokerOrder`, представленного в листинге 10.10.

Листинг 10.10. Создание пользовательского класса для сортировки карт

```
class PokerOrder(int):
    def __new__(cls, x):
        numbers_mapping = {'J': 11, 'Q': 12, 'K': 13, 'A': 14}
        casted_number = numbers_mapping.get(x, x)
        return super().__new__(PokerOrder, casted_number)
```

> **НАПОМИНАНИЕ** Если вы пытаетесь получить значение из объекта `dict`, метод `get` может содержать резервное значение, которое используется при отсутствии ключа в словаре.

В классе `PokerOrder` метод `__new__` переопределяется, чтобы мы могли изменять поведение по умолчанию при конструировании экземпляра `PokerOrder` — класса, являющегося подклассом `int`. Как упоминалось в разделе 8.1, `super()` создает объект-заместитель, ссылающийся на суперкласс `int`, который должен получать число (`casted_number` в нашей реализации) для конструирования экземпляра. Точнее говоря, если номинал карты лежит в диапазоне от 2 до 10, используется число. Если картой является валет, дама, король или туз (`J`, `Q`, `K` или `A`), ее необходимо преобразовать в соответствующее целое число, чтобы класс мог отобразить нечисловую карту на правильное числовое значение. Пример сортировки:

```
print(sorted(cards, key=PokerOrder))
# Вывод: [1, 10, 'J', 'A']
```

10.5.4. Создание декораторов в форме классов

В разделе 7.3 вы узнали о создании декораторов — функций высшего порядка, которые изменяют декорируемые функции без последствий для выполняемых ими операций. «За кулисами» процесс декорирования передает декорируемую функцию декоратору. Таким образом, процесс декорирования фактически эквивалентен вызову функции высшего порядка. Так как классы тоже вызываются, эта характеристика позволяет создавать декораторы в форме пользовательского класса, как будет показано в этом разделе. Чтобы напомнить вам, о чем идет речь, приведу фрагмент кода, показывающий, как создать декоратор для измерения времени выполнения функции:

```
import time

def logging_time(func):
    def logger(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"Calling {func.__name__}: {time.time() - start:.5f}")
        return result

    return logger
```

В данном случае я использую минимум элементов для декоратора. Если вы не знакомы с декораторами, обращайтесь к рекомендациям по созданию декораторов в разделе 7.3. Чтобы преобразовать эту функцию в класс, вспомните, что конструктор класса рассчитывает получить функцию в аргументе. Решение выглядит так:

```
import time

class TimeLogger:
    def __init__(self, func):
        def logger(*args, **kwargs):
            start = time.time()
            result = func(*args, **kwargs)
            print(f"Calling {func.__name__}: {time.time() - start:.5f}")
            return result
        self._logger = logger

    def __call__(self, *args, **kwargs):
        return self._logger(*args, **kwargs)
```

В этом фрагменте заслуживают внимания два обстоятельства:

- Защищенный атрибут `_logger` используется для хранения созданной внутренней функции во внутренней реализации, так как мы знаем, что процесс декорирования создает замыкание, представляющее собой внутреннюю функцию.
- Мы переопределяем специальный метод `__call__`, который вызывается при попытке вызвать экземпляр класса. Таким образом, при вызове декорируемой

функции вызываться будет замыкание из атрибута `_logger`. Обратите внимание: реализуя `__call__` в пользовательском классе, мы делаем экземпляры класса вызываемыми. Как показано на рис. 10.9, следует знать, что наряду с функциями и классами экземпляры класса, реализующего `__call__`, также являются вызываемыми, как в случае класса `TimeLogger`.

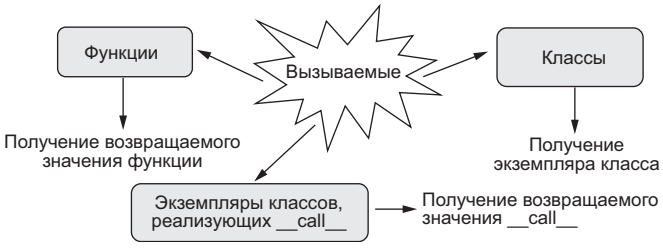


Рис. 10.9. Три разновидности вызываемых объектов и ожидаемые результаты вызова. Функции вызываются для получения возвращаемых значений. Классы вызываются для получения экземпляров. Экземпляры вызываются для получения результата метода __call__

С таким классом используем уже знакомый синтаксис для декорирования функции:

```
@TimeLogger
def calculate_sum(n):
    return sum(range(n))
result = calculate_sum(100_000)
# Вывод: Calling calculate_sum: 0.00181
```

Обратите внимание на то, что декорируемая функция уже не является функцией. Она представляет собой экземпляр класса `TimeLogger`:

```
print(calculate_sum)
# Вывод: <__main__.TimeLogger object at 0x7fe8180de710>
```

По умолчанию вызывать экземпляры невозможно, Например, вы не сможете использовать запись вида `[1, 2, 3]()` или `"Hello, World!"()`. Чтобы экземпляр вел себя как функция, следует переопределить специальный метод `__call__`, который возвращает атрибут `_logger` — функцию, то есть вызываемый объект. Другими словами, чтобы сделать экземпляр вызываемым, мы передаем операцию вызова его атрибуту, содержащему функцию (`_logger`).

10.5.5. Обсуждение

Этот раздел посвящен теме вызываемости объектов Python — их способности вызываться оператором вызова `()`. Как классы, так и функции обладают вызываемостью, что создает множество возможностей для нетривиальных

взаимодействий, например передачи в аргументе `key` и создания декоратора с использованием пользовательского класса. При этом появляется возможность реализации сложных декораторов, получающих параметры (особенно во втором случае). Использование класса упрощает реализацию гибкости, так как позволяет вам добавлять другие атрибуты к экземпляру.

10.5.6. Задача

Руби создает класс `TimeLogger` как декоратор для измерения быстродействия функций в ее проекте. Как говорилось в разделе 7.3, одной из передовых практик реализации декораторов является использование декоратора `wraps` из модуля `functools`. Как Руби использовать `wraps` в классе `TimeLogger`?

> **ПОДСКАЗКА** Декоратор `wraps` применяется к декорируемой функции до определения внутренней функции.

ИТОГИ

- Тип объекта проверяется при помощи встроенной функции `type`. Получение информации о типе во время выполнения программы позволяет писать более гибкий код.
- Функция `isinstance` проверяет, является ли объект экземпляром класса или кортежа классов. `isinstance` обладает большей гибкостью, чем `type`, так как она дает действительный результат с учетом надклассов проверяемого класса.
- Модуль `collections.abc` позволяет проверить обобщенный тип объекта для применения одних и тех же операций к нескольким классам, реализующим тот же интерфейс.
- Экземпляр класса проходит через следующие процессы: создание экземпляра → активность в пространстве имен → отслеживание с использованием счетчика ссылок (происходит одновременно с активностью в пространстве имен) → уничтожение.
- При создании копии объекта по умолчанию копируется только внешний контейнер данных; эта разновидность копирования называется поверхностной.
- Если вам понадобится создать копию, которая содержит копии для внутренних изменяемых объектов, следует создать глубокую копию, позволяющую изменять внутренние изменяемые объекты без последствий для исходных данных.
- Встроенный модуль `copy` спроектирован для копирования объектов стандартным способом. Но если вам понадобится определить нестандартное поведение копирования для своего класса, переопределите методы `__copy__` и `__deepcopy__`.

- Когда Python потребуется провести разрешение переменной или имени вообще, поиск значения для использованного имени осуществляется по схеме LEGB (локальная → охватывающая → глобальная → встроенная).
- Если вам понадобится изменить переменную в локальной области видимости, необходимо использовать ключевое слово `global` или `nonlocal` в зависимости от того, где находилось исходное определение переменной; первое предназначено для глобально определенной переменной, а второе — для переменной в охватывающей области видимости, которая существует только у вложенных функций.
- И классы, и функции изначально используются для вызова. Несмотря на общую характеристику вызываемости, вы должны понимать разницу между классами и функциями для встроенных функций. Наличие этой общей характеристики означает, что классы и функции могут использоваться в некоторых типичных ситуациях, например при передаче аргумента `key` или при создании декоратора.

11 Работа с файлами

В этой главе

- ✓ Чтение и запись файлов
- ✓ Обработка файлов с табличными данными
- ✓ Сохранение данных в файлах
- ✓ Управление файлами на вашем компьютере
- ✓ Обращение к метаданным файлов

Файлы становятся неотъемлемой частью любой компьютерной системы или приложения. Они используются для хранения данных. Если потребуется передать данные другому участнику команды, вы используете для этого файл. Когда вы получаете файл от кого-нибудь, его необходимо открыть, прочитать его содержимое, обработать данные и либо отправить результаты в другой файл, либо присоединить их к тому же файлу. Эти операции связаны с содержимым файлов. В вашем приложении могут использоваться сотни разных объектов Python, а некоторые объекты требуют серьезных вычислений или другой обработки, поэтому такие объекты было бы удобно хранить в файлах. Когда вам потребуется снова использовать тот же объект, вы загрузите его из файла, что значительно сэкономит время обработки.

Файлы присутствуют в любой компьютерной системе, и ваша работа может требовать разнообразных операций с ними: перемещения файлов в другой каталог,

извлечения файлов определенного вида, проверки того, был ли файл изменен на прошлой неделе, и т. д. Умение выполнять такие операции на программном уровне позволит вам делать то, что трудно реализовать вручную, и отслеживать любые изменения, внесенные в файлы. В этой главе рассматриваются темы, связанные с файлами, — не только чтение и запись содержимого файлов, но и такие распространенные операции с ними, как перемещение и копирование.

11.1. КАК ВЫПОЛНЯТЬ ЧТЕНИЕ И ЗАПИСЬ В ФАЙЛЫ ЧЕРЕЗ УПРАВЛЕНИЕ КОНТЕКСТОМ

В наших проектах используются разнообразные типы файлов: табличные, мультимедийные, текстовые и т. д. При работе с такими файлами первым делом необходимо прочитать их, чтобы обработать содержащиеся в них данные. И хотя для выполнения операций с файлами можно использовать специальные программные продукты, в проектах часто требуется программная обработка файлов, особенно при большом их количестве. Чтобы воспользоваться специальными средствами Python, например pandas для обработки табличных данных, также необходимо читать файлы на программном уровне. Понятно, что программная работа с файлами становится неотъемлемой частью общей обработки данных. В этом разделе вы научитесь читать и записывать файлы в Python. Так как текстовая информация — это наиболее распространенный вид данных, в наших примерах будут использоваться текстовые файлы. Однако общие приемы применимы и к другим форматам, например к двоичным файлам, в которых хранятся байтовые данные.

11.1.1. Открытие и закрытие файлов: менеджер контекста

Основные операции с файлами — открытие и закрытие файлов. В этом разделе вы научитесь открывать и закрывать файлы двумя способами: базовым и с использованием менеджера контекста (который будет рассмотрен позднее).

Допустим, вы используете текстовые файлы для хранения данных своего таск-менеджера. Начнем с создания текстового файла с именем tasks.txt, в котором хранятся следующие данные:

```
1001,Homework,5
1002,Laundry,3
1003,Grocery,4
```

Каждая строка данных представляет информацию задачи: идентификатор, название и степень срочности. Для простоты ограничимся тремя строками данных. Файл можно открыть при помощи встроенной функции open:

```
text_file = open("tasks.txt")

print(text_file)
# Вывод: <_io.TextIOWrapper name='tasks.txt' mode='r' encoding='UTF-8'>
```

Проверяя содержимое объекта text_file при помощи функции `print`, мы получаем следующую информацию:

- Объект является экземпляром `_io.TextIOWrapper` — класса, создающего буферизованный текстовый поток с получением высокоуровневого доступа к текстовым данным, хранящимся в файле. Такие объекты называют также *потоками* (stream) или *файловыми бъектами* (file object).
- `name` сообщает имя файла.
- `mode` указывает, как должен читаться файл. `'r'` означает режим чтения; в этом режиме разрешено только чтение данных из файла. Другие операции (например, запись данных в файл) в этом режиме невозможны.
- `encoding` указывает, как закодирован текстовый файл. В большинстве случаев вам не придется задумываться о способе кодирования, потому что чаще всего данные кодируются в UTF-8 (при этом также обеспечивается обратная совместимость с кодировкой ASCII, о которой вы наверняка слышали) — самой распространенной форме кодирования данных в системе Юникод.
- После создания объекта файла можно переходить к чтению данных. Существуют разные способы чтения данных (раздел 11.1.2), но самым простым из них является метод `read`, представленный в листинге 11.1.

Листинг 11.1. Чтение данных в виде строки

```
text_data = text_file.read()

print(type(text_data))  <-- Проверяет при помощи type
# Вывод: <class 'str'>

print(text_data)
# Выводимые строки:
1001,Homework,5
1002,Laundry,3
1003,Grocery,4
```

Метод `read` читает все текстовые данные из файла в строку. Вы можете вывести эту строку, чтобы убедиться в том, что она совпадает с текстом в файле. К ней применима дополнительная обработка, например разбиение каждой записи и извлечение содержащихся в ней данных (раздел 2.3). Когда обработка завершена, файл закрывается методом `close`. После того как файл будет закрыт, его статус проверяется по атрибуту `closed`, который должен содержать `True`:

```
text_file.close()

assert text_file.closed
```

Всегда закрывайте файл после завершения работы с ним. Файлы являются общими ресурсами на вашем компьютере, и если вы забудете закрыть их,

то любые изменения, внесенные в файловый объект, могут быть потеряны в реальном файле. При закрытии файла все обновления в нем сохраняются на диске, и другие обращающиеся к файлу процессы будут работать с новейшими данными.

Чтобы предотвратить потерю данных из-за того, что вы забыли закрыть файл, можно воспользоваться методом управления контекстом — инструкцией `with`, которая считается питоническим способом чтения файлов. Это показано в листинге 11.2.

Листинг 11.2. Использование with для открытия файлов

```
with open("tasks.txt") as file:  <-- Инструкция with
    print(f"file object: {file}")
    data = file.read()
    print(data)

# Выводимые строки:
file object: <_io.TextIOWrapper name='tasks.txt' mode='r' encoding='UTF-8'>
1001,Homework,5
1002,Laundry,3
1003,Grocery,4
```

Синтаксис использования `with` выглядит так: `with open("filepath") as file` — заголовок, который создает файловый объект и присваивает его переменной `file`. Затем создаются отступы, обозначающие тело, в котором определяются операции. Как видно из листинга 11.2, мы получаем такой же результат, как в листинге 11.1. Важнейшее преимущество инструкции `with` заключается в том, что файл не нужно закрывать явно. После завершения инструкции `with` файл будет закрыт автоматически:

```
assert file.closed
```

Инструкция `with` известна как прием управления контекстом. Менеджер контекста связывает соответствующий объект ресурса с заголовком инструкции `with`, а в теле `with` выполняются операции с объектом. При завершении тела и выходе из `with` менеджер контекста автоматически закрывает связь с ресурсом. Для файлов менеджер освобождает объект файла. На рис. 11.1 работа менеджера контекста продемонстрирована на конкретном примере.

11.1.2. Разные способы чтения данных из файла

В листинге 11.1 представлен метод `read`, который получает все текстовые данные. При большом размере текстового файла загрузка всех данных занимает значительное время, и иногда у компилятора может не хватить памяти для хранения такого объема данных. В этих случаях приходится использовать другие способы чтения данных в зависимости от конкретной ситуации; некоторые из них будут описаны в этом разделе.

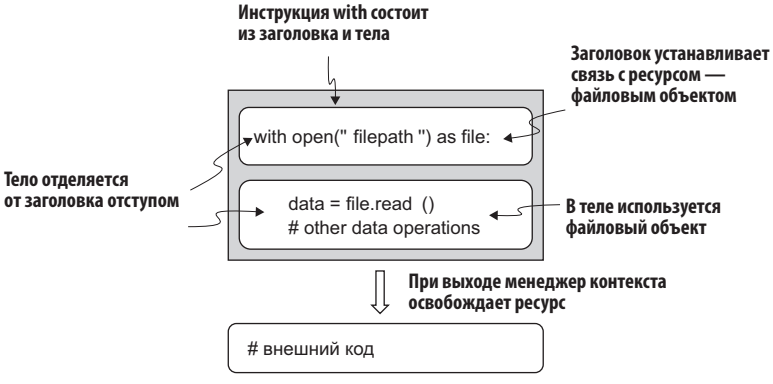


Рис. 11.1. Логика работы менеджера контекста на примере управления файлами. Инструкция with состоит из заголовка и тела. Заголовок устанавливает связь с ресурсом, а в теле этот ресурс используется. При выходе из инструкции with менеджер контекста освобождает ресурс

Чтение строк с использованием генератора

В разделе 7.4 рассматривались генераторы — поставщики данных, экономно расходующие память и выдающие данные по запросу. Файловый объект представляет собой поток данных, и его можно использовать так, словно он является генератором, последовательно выдающим строки данных.

Самый распространенный вариант обработки файла в форме генератора — последовательность чтения строк в цикле `for` с обработкой данных по строкам, как показано в листинге 11.3. Чтобы сделать чтение файла чуть более интересным, я включил код, преобразующий каждую строку в экземпляр класса `Task`.

Листинг 11.3. Чтение файла с использованием генератора

```
from collections import namedtuple

Task = namedtuple("Task", "task_id title urgency")

with open("tasks.txt") as file:
    for line in file:
        stripped_line = line.strip()
        task_id, title, urgency = stripped_line.split(",")
        task = Task(task_id, title, urgency)
        print(f"{stripped_line}: {task}")

# Выводимые строки:
1001,Homework,5: Task(task_id='1001', title='Homework', urgency='5')
1002,Laundry,3: Task(task_id='1002', title='Laundry', urgency='3')
1003,Grocery,4: Task(task_id='1003', title='Grocery', urgency='4')
```

Создает класс именованного кортежа

Удаляет завершающий разрыв строки

Разбивает строку по запятым

Как видите, `file` используется в качестве итератора в цикле `for`, который последовательно выдает каждую строку. Учтите, что каждая строка завершается «невидимым» разрывом строки, который необходимо удалить вызовом `strip`. Для создания экземпляра класса `Task` используются элементы, полученные при разбиении.

ВОПРОС Что произойдет, если не удалить разрыв строки?

Чтение строк с созданием списка

Если файл содержит не слишком много данных, строки можно прочитать в объект `list` при помощи метода `readlines`. Так как объекты `list` являются изменяемыми, проще изменить данные и сохранить их для других целей.

Допустим, вы хотите извлечь все данные из текстового файла tasks.txt в объект `list` и добавить номер в каждую строку. Нужный вывод выглядит так:

```
desired_output = [
    '#1: 1001,Homework,5',
    '#2: 1002,Laundry,3',
    '#3: 1003,Grocery,4'
]
```

Так как ожидаемым результатом является объект `list`, воспользуемся методом `readlines` для создания объекта `list`. Это позволит оперировать данными благодаря их изменяемости, как показано в листинге 11.4.

Листинг 11.4. Чтение строк в список

```
with open("tasks.txt") as file:
    lines = file.readlines()
    updated_lines = [f"#{row}: {line.strip()}" for row, line
    ➥ in enumerate(lines, start=1)]  <-- enumerate создает счетчик

assert desired_output == updated_lines
```

Функция `enumerate` (раздел 5.3.1) используется для создания счетчика итераций для элементов. Используя списковое включение (раздел 5.2.1), мы создаем объект `list` с именем `updated_lines`, который соответствует ожидаемому объекту `list` с именем `desired_output`.

Чтение одной строки

Реже встречаются случаи, когда требуется прочитать отдельную строку. Допустим, вы хотите прочитать только заголовок файла, чтобы определить имена столбцов в CSV-файле (дополнительная информация об обработке CSV-файлов приведена в разделе 11.2). И хотя мы можем прочитать все строки и получить первый элемент, чтение такого объема данных займет много

времени. Вместо этого следует воспользоваться методом `readline` и прочитать текст одной строки. Очевидно, что это потребует намного меньше времени, чем чтение всех строк.

Метод `readline` может вызываться многократно. Файловый объект следит за тем, где останавливается каждый вызов `readline` (как и генератор, этот объект знает позицию элемента в последовательности), и при следующем вызове `readline` продолжает читать данные от позиции последней остановки, как показано в листинге 11.5.

Листинг 11.5. Чтение одной строки

```
with open("tasks.txt") as file:
    print(file.readline())
    print(file.readline())
    print(file.readline(5))
    print(file.readline(8))
    print(file.readline())

# Выводимые строки:
1001,Homework,5  ◄——  Выводит следующую пустую строку из-за разрыва строки

1002,Laundry,3

1003,
Grocery,
4
```

В листинге 11.5 заслуживают внимания три обстоятельства:

- `readline` может получать аргумент `size`, который определяет количество читаемых символов в строке. Например, `file.readline(5)` читает строку `1003,`, а `file.readline(8)` читает `Grocery,`.
- Чтобы получить отдельные строки, вызовите `readline` многократно.
- Строка завершается разрывом строки. Когда вы вызываете `readline`, вызов читает всю строку, включая завершающий разрыв строки; отсюда и пустые строки в выводимом сообщении.

ПРИМЕЧАНИЕ Как и `readline`, вызовы `read` и `readlines` могут получать аргумент `size`, определяющий количество символов, читаемых из файла.

11.1.3. Разные способы записи данных в файл

Обычно данные читаются из файлов для обработки хранимой информации. Когда редактирование будет завершено или будут подготовлены данные из другого источника, необходимо записать данные в файл для долгосрочного хранения. В этом разделе описаны типичные сценарии записи данных в файлы.

Запись строковых данных в новый файл

Распространенная ситуация — у вас есть подготовленные данные и вам надо сохранить их в новом файле. Допустим, имеются следующие данные:

```
data = """1001,Homework,5
1002,Laundry,3
1003,Grocery,4"""
```

Чтобы записать данные в новый файл, создадим объект файла инструкцией `with`. Вместо заблаговременного создания пустого файла мы вызываем функцию `open` с указанием пути к новому файлу; при этом новый файл будет автоматически создан по заданному пути, как показано в листинге 11.6.

Листинг 11.6. Запись в новый файл

```
with open("tasks_new.txt", "w") as file:  <-- Определяет режим записи
    print("File:", file)
    result = file.write(data)
    print("Writing result:", result)

# Выводимые строки:
File: <_io.TextIOWrapper name='tasks_new.txt' mode='w' encoding='UTF-8'>
Writing result: 45
```

При вызове функции `open` помимо пути к файлу мы указываем, что объект файла открывается в режиме `"w"`, то есть в режиме записи вместо режима чтения, используемого по умолчанию. Из выводимого сообщения видно, что объект файла использует режим `'w'`. Чтобы записать строковые данные в новый файл, мы вызываем метод `write`. Вызов этого метода возвращает количество записанных символов — 45 в нашем примере.

Указание режима `"w"` для объекта файла необходимо для операций записи. Если вы откроете файл в режиме чтения, который используется по умолчанию, вы не сможете записать никакие данные, как в следующем примере:

```
with open("tasks_new.txt") as file:  <-- По умолчанию используется режим чтения
    print("File:", file)
    result = file.write(data)
    print("Writing result:", result)

# ERROR: io.UnsupportedOperation: not writable
```

Запись списка строк в новый файл

Ранее вы видели, что данные можно прочитать из файла в объект `list`, элементами которого являются строки. Вполне ожидаемо, что список строк также можно записать в файл. Для выполнения этой операции используется метод

`writelines`. Как и при записи строковых данных, файл должен быть открыт в режиме записи, как показано в листинге 11.7.

Листинг 11.7. Запись списка в файл

```
list_data = [
    '1001,Homework,5',
    '1002,Laundry,3',
    '1003,Grocery,4'
]

with open("tasks_list_write.txt", "w") as file:
    file.writelines(list_data)  <-- writelines возвращает None
```

Открыв файл tasks_list_write.txt, вы увидите, что данные могут выглядеть неправильно:

```
with open("tasks_list_write.txt") as file:
    print(file.read())
# Вывод: 1001,Homework,51002,Laundry,31003,Grocery,4
```

На самом деле такое поведение предсказуемо. `writelines` записывает данные последовательно, между элементами нет разрывов строк, и файл не должен быть многострочным. Следовательно, если вы хотите создать файл, разбитый на строки, вам придется добавить в данные разрывы строк. Я оставляю вам эту задачу для самостоятельного решения (раздел 11.1.5).

К настоящему моменту вы узнали, как читать и записывать данные разными методами, включая `read`, `write`, `readline`, `readlines` и `writelines`. Чтобы вам было проще различать их, на рис. 11.2 приведена сводка этих операций.

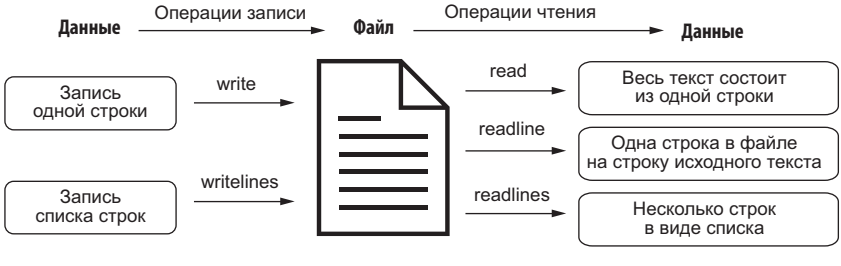


Рис. 11.2. Ключевые функции чтения и записи данных в файлы. Имеющиеся данные можно записать в файл методами write и writelines. При чтении из файла текстовые данные можно получить вызовами read, readline и readlines. Эти функции служат разным целям

Возможно, вы заметили, что операции чтения и записи почти симметричны; единственное отличие — отсутствие метода `writeline` в левой части. Впрочем, он и не нужен. Если вам понадобится записать строку, используйте метод `write`.

Присоединение строковых данных к существующему файлу

Если у вас появились новые данные, бывает нужно присоединить их к существующему файлу. Допустим, вы создали новую задачу, которая содержит следующие данные:

```
new_task = "1004,Museum,3"
```

Эти данные нужно добавить в конец файла tasks.txt. Вместо режима записи следует использовать режим присоединения, как показано в листинге 11.8.

Листинг 11.8. Присоединение данных к существующему файлу

```
with open("tasks.txt", "a") as file:
    file.write(f"\n{new_task}")  ◄—— Добавление разрыва строки
```

Внутри вызова функции `open` укажите режим `"a"`, чтобы открыть файл в режиме присоединения; метод `write` добавляет данные в конец файла. Следует заметить, что перед `new_task` находится префикс `\n` (разрыв строки), чтобы данные добавлялись в новой строке, а не включались в последнюю строку файла.

В основе режима присоединения лежит идея о том, что при чтении и записи данных позиция операции определяется курсором. Я уже упоминал, что объект файла представляет поток данных, а курсор обозначает позицию в этом потоке. Таблица 11.1 содержит больше информации о режимах и позициях их курсоров.

Таблица 11.1. Режимы открытия файлов

Режим[a]	read	write	create	truncate	Позиция курсора
`r`	*				Начало
`w`		*	*	*	Начало
`a`		*	*		Конец
`r+`	*	*			Начало
`w+`	*	*	*	*	Начало
`a+`	*	*	*		Конец
`x`			*		Начало

[a] `read`: чтение данных; `write`: запись новых данных; `create`: создание нового файла; `truncate`: изменение размера файла; позиция курсора: на начало операции.

Если файл открывается в режиме `"a"`, курсор устанавливается в конец файла, поэтому новый текст присоединяется в конец файла. Для часто используемых режимов `"r"` и `"w"` курсор находится в начале, так что соответствующие операции чтения и записи выполняются в начале.

11.1.4. Обсуждение

Использование инструкции `with` при чтении и записи файлов считается питоническим решением. Однако инструкция `with` создавалась скорее для задач управления контекстом, чем для обработки файлов. В более широком смысле менеджер контекста используется при работе с общими ресурсами, например подключением к базе данных. Как показано в разделе 14.3, при работе с базами данных SQLite операции могут выполняться с использованием подключения, которым управляет менеджер контекста, как в следующем примере:

```
import sqlite3
with con = sqlite3.connect("database.sqlite"):
    # Выполняемые операции
    pass
```

11.1.5. Задача

Лео работает инженером-электромехаником, и в повседневной работе ему часто приходится использовать Python для записи данных в файлы. Однажды ему понадобилось записать объект `list` в новый файл, как было сделано в листинге 11.7. Однако он обнаружил, что файл содержит только одну строку вместо многих, при этом каждая строка представляет элемент объекта `list`. Как ему изменить объект `list`, чтобы файл был многострочным?

> **ПОДСКАЗКА** К каждому элементу списка можно присоединить разрыв строки.

11.2. КАК РАБОТАТЬ С ТАБЛИЧНЫМИ ФАЙЛАМИ ДАННЫХ

Для работы с данными многие используют Microsoft Excel, где файлы имеют формат *электронных таблиц*. На более общем уровне электронные таблицы называются *табличными данными*; такие данные состоят из строк и столбцов. Компания может хранить в табличном виде данные продаж, а колледж — результаты экзаменов. В табличном виде сохраняются данные, полученные в ходе исследовательского проекта. Как видите, табличные данные находят повсеместное применение, так что их обработка становится важнейшим навыком обработки данных. Табличные данные можно преобразовать в файлы в формате CSV (Comma-Separated Values, то есть «данные, разделенные запятыми») для упрощения обмена данными между разными системами. Этот раздел посвящен обработке CSV-файлов — типичного формата табличных файлов данных.

11.2.1. Чтение CSV-файлов

Как обычно, обработку данных мы начнем с их чтения. Для клиентских приложений данные должны быть прочитаны перед их выводом. Допустим, наш

таск-менеджер использует CSV-файл для хранения данных, относящихся к задачам, — файл tasks.txt (раздел 11.1). Чтобы вывести данные задач в приложении, необходимо знать, как прочитать CSV-файлы.

Хотя в разделе 11.1 об этом не говорилось, файл tasks.txt представляет собой CSV-файл. А значит, вы уже знаете, как читать данные из CSV-файлов. Однако при этом нам приходилось самостоятельно разбивать строку для получения хранимых данных, а при работе с CSV-файлами операция разбиения становится стандартной. Неудивительно, что стандартная библиотека Python включает встроенное решение для этой задачи: модуль `csv` позволяет прочитать данные напрямую при помощи объекта `csv_reader`, как показано в листинге 11.9.

Листинг 11.9. Чтение CSV-файла с использованием модуля csv

```
import csv  <-- Импортирует модуль

with open("tasks.txt", newline="") as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)

# Выводимые строки:  <-- Если вы присоединяли данные в разделе 11.1, результат может выглядеть иначе
['1001', 'Homework', '5']
['1002', 'Laundry', '3']
['1003', 'Grocery', '4']
```

ОБРАТИТЕ ВНИМАНИЕ В официальной документации Python рекомендуется указывать символ новой строки в виде `''` для обеспечения кросс-платформенной совместимости при обработке его системой. За дополнительной информацией обращайтесь к ресурсу https://docs.python.org/3/library/csv.html.

Как показано в листинге 11.9, объект `csv_reader` создается вызовом функции `reader` с объектом файла. Созданный объект `csv_reader` представляет собой итератор, что позволяет перебрать его содержимое в цикле `for`. Каждый элемент представляет собой объект `list`, состоящий из значений, разделенных запятыми, — такой же результат был получен в листинге 11.5. Однако на этот раз нам не пришлось заново изобретать велосипед — мы просто воспользовались встроенным модулем `csv`!

НАПОМИНАНИЕ Не пытайтесь изобретать велосипед. Всегда используйте доступные решения, особенно если они предоставляются стандартной библиотекой.

Вы уже знаете, что конструктор `list` может получать итерируемый объект для создания объекта `list`. Следовательно, для получения всех строк в виде объекта `list` следует вызвать конструктор `list`:

```
with open("tasks.txt", newline="") as file:
    csv_reader = csv.reader(file)
    tasks_rows = list(csv_reader)
    print(tasks_rows)

# Выводимая строка:
[['1001', 'Homework', '5'], ['1002', 'Laundry', '3'],
➥ ['1003', 'Grocery', '4']]
```

11.2.2. Чтение CSV-файла с заголовком

В файле tasks.txt содержатся только три поля данных: идентификатор, название и степень срочности. Если файл состоит из множества полей, становится трудно определить, в каком поле хранятся те или иные данные. Для предотвращения любых неоднозначностей многие CSV-файлы имеют заголовок, содержащий метки всех полей. В этом разделе вы научитесь читать CSV-файлы с заголовком. Допустим, имена полей были включены в файл tasks.txt, который содержит следующие данные:

```
task_id,title,urgency
1001,Homework,5
1002,Laundry,3
1003,Grocery,4
```

Как видите, первая строка определяет три поля, которым соответствует каждое значение в следующих строках. Если у вас имеется CSV-файл с заголовком, лучше читать строки как объект `dict`, в котором имена полей являются ключами, как показано в листинге 11.10.

Листинг 11.10. Чтение CSV-файла с заголовком при помощи csv_reader

```
with open("tasks.txt", newline="") as file:
    csv_reader = csv.reader(file)
    fields = next(csv_reader)  <-- Получает следующий элемент
    print("Field:", fields)
    for row in csv_reader:
        task_dict = dict(zip(fields, row))  <-- Создает объект dict
        print(task_dict)

# Выводимые строки:
Field: ['task_id', 'title', 'urgency']
{'task_id': '1001', 'title': 'Homework', 'urgency': '5'}
{'task_id': '1002', 'title': 'Laundry', 'urgency': '3'}
{'task_id': '1003', 'title': 'Grocery', 'urgency': '4'}
```

Чтобы напомнить некоторые приемы, которые были описаны ранее, отмечу заслуживающие внимания моменты из листинга 11.10:

- Так как объект `csv_reader` является итератором (раздел 5.1), для него можно вызвать функцию `next` с целью получения данных первой строки.

- После использования первого элемента итератора перебор продолжается со второго элемента. В цикле `for` объект `csv_reader` выдает элементы, начиная со второй строки.
- Конструктор `dict` получает итерируемый объект, у которого каждый элемент содержит два элемента. Функция `zip` используется для создания объекта `zip` посредством соединения `fields` и `row`. Из выходных данных видно, что мы получаем три объекта `dict`, соответствующие данным в CSV-файле.

Вероятно, вы заметили, что отдельное чтение первой строки с конструированием необходимых данных нельзя назвать интуитивно понятным. Однако CSV-файлы с заголовком встречаются настолько часто, что должно существовать более простое решение. В самом деле, модуль `csv` предоставляет дополнительный класс `DictReader`, предназначенный специально для этой цели, как показано в листинге 11.11.

Листинг 11.11. Чтение CSV-файла с заголовком при помощи DictReader

```
with open("tasks.txt", newline="") as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
        print(row)

# Выводимые строки:
{'task_id': '1001', 'title': 'Homework', 'urgency': '5'}
{'task_id': '1002', 'title': 'Laundry', 'urgency': '3'}
{'task_id': '1003', 'title': 'Grocery', 'urgency': '4'}
```

Вместо вызова функции `reader` мы вызываем конструктор `DictReader` для создания объекта `DictReader`, который получает первую строку как набор ключей. Как видите, решение в листинге 11.11 намного чище решения из листинга 11.10. Это еще одна демонстрация того, насколько компактным может получиться код Python при использовании правильных средств. Небольшой совет: если вы столкнетесь с частой проблемой в своей повседневной работе, скорее всего, в Python для нее уже существует решение, и вам нужно только найти его! По моему опыту, можно начать поиск в Google с запроса «`Python` + *текущая задача*». Например, если вам понадобится читать PDF-файлы в Python, введите запрос «Python read pdf files» («Python чтение pdf файлов»). Обычно нескольких начальных страниц результатов поиска будет достаточно для нахождения потенциального решения.

ВОПРОС Учитывая, что `csv_reader` является итератором, как можно получить все данные в виде объекта `list`, содержащего эти объекты `dict`?

11.2.3. Запись данных в CSV-файл

После того как данные будут обработаны, они записываются обратно в CSV-файл. У `reader` и `DictReader` существуют парные средства записи: `writer` и `DictWriter`.

Как нетрудно догадаться, `writer` записывает объект `list`, а `DictWriter` — объект `dict`. В этом разделе показано, как это делается. Так как запись происходит достаточно тривиально, раздел получился коротким.

Допустим, вы хотите добавить в CSV-файл строку `1004,Museum,3`. При использовании `writer` эту строку необходимо преобразовать в объект `list`:

```
new_task = "1004,Museum,3"
with open("tasks.txt", "a", newline="") as file:
    file.write("\n")
    csv_writer = csv.writer(file)
    csv_writer.writerow(new_task.split(","))
```

Создает список, который требуется записать, с использованием split

Как и в случае с записью данных в обычный текстовый файл, если вы знаете, что последняя строка данных не завершается разрывом строки, его следует добавить вручную: `file.write("\n")`.

> **НАПОМИНАНИЕ** Файл должен быть открыт в режиме `"a"` — режиме присоединения. Если использовать режим `"w"`, все существующие данные будут стерты.

Иногда данные обрабатываются в форме объектов `dict`. Допустим, вам нужно сохранить следующие данные в новом CSV-файле:

```
tasks = [
    {'task_id': '1001', 'title': 'Homework', 'urgency': '5'},
    {'task_id': '1002', 'title': 'Laundry', 'urgency': '3'},
    {'task_id': '1003', 'title': 'Grocery', 'urgency': '4'}
]
```

Данные представляют собой объект `list`, состоящий из нескольких объектов `dict`. В данном случае будет использоваться объект `DictWriter`, как показано в листинге 11.12.

Листинг 11.12. Запись данных в CSV-файл при помощи DictWriter

```
fields = ['task_id', 'title', 'urgency']

with open("tasks_dict.txt", "w", newline="") as file:
    csv_writer = csv.DictWriter(file, fieldnames=fields)
    csv_writer.writeheader()
    csv_writer.writerows(tasks)
```

Записывает заголовок

Записывает несколько строк

В листинге 11.12 заслуживают внимания три обстоятельства:

- При создании экземпляра `DictWriter` необходимо указать имена полей в аргументе `fieldnames`.
- Для записи заголовка вызывается метод `writeheader`.

- Так как в объекте `list` хранятся объекты `dict`, весь набор данных можно записать вызовом метода `writerows` вместо метода `writerow`, который записывает всего одну строку.

До настоящего момента рассматривались операции чтения и записи данных с CSV-файлами. Нетрудно догадаться, что чтение и запись данных включают симметричные операции: `reader` с `writer` и `DictReader` с `DictWriter`. На рис. 11.3 приведена наглядная сводка этих операций. Если вы работаете со списками, выбирайте `reader` и `writer`. Если вы работаете со словарями, выбирайте `DictReader` и `DictWriter`. Еще один фактор, который необходимо учитывать, — использование заголовка в CSV-файле; при наличии заголовка операции проще выполнять с `DictReader` и `DictWriter`.

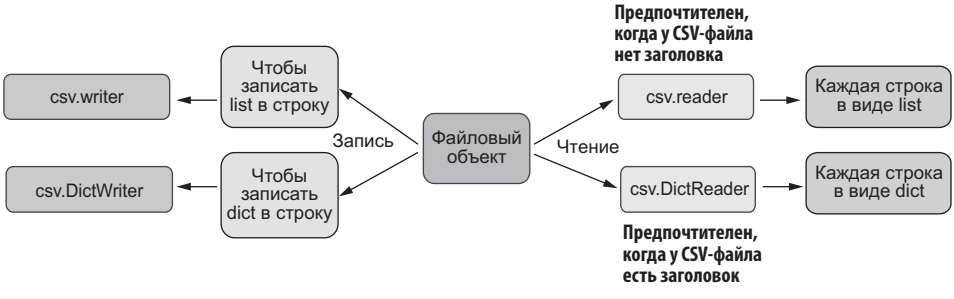


Рис. 11.3. Операции чтения и записи с использованием модуля csv. В операциях чтения используются объекты reader и DictReader; первый читает каждую строку в list, а второй — каждую строку в dict. В операциях записи используются объекты writer и DictWriter; первый записывает в строку содержимое list, а второй — содержимое dict

11.2.4. Обсуждение

Табличные данные преобразуются в формат CSV. Встроенный модуль `csv` предоставляет удобные средства для обработки данных CSV, включая чтение и запись. Вы должны уметь выполнять эти операции в обоих направлениях. Заметим, что если вам понадобится выполнять числовые операции с данными CSV, для расширенных возможностей обработки придется изучать сторонние библиотеки (например, pandas). Эти пакеты способны читать файлы CSV простым вызовом функции. Так, вызов `pandas.read_csv("filepath.csv")` создает на базе CSV-файла `DataFrame` табличную модель данных, которая позднее может использоваться для выполнения различных операций.

11.2.5. Задача

Лео использует CSV-файлы для хранения экспериментальных данных своих проектов. Для одного проекта он вызвал метод `writerows` с `DictWriter`, чтобы

записать объект `list`, состоящий из нескольких объектов `dict`, как в листинге 11.12. Как ему использовать этот метод с обычным объектом CSV `writer`, чтобы записать несколько объектов `list`?

ПОДСКАЗКА Данные необходимо организовать в объект `list`, в котором каждый элемент представляет данные одной строки данных:

```
tasks = [
    ['1001', 'Homework', '5 '],
    ['1002', 'Laundry', '3'],
    ['1003', 'Grocery', '4']
]
```

11.3. КАК СОХРАНЯТЬ ДАННЫЕ В ФАЙЛАХ С ИСПОЛЬЗОВАНИЕМ КОНСЕРВАЦИИ

Во время выполнения программ наш код порождает сотни объектов. Когда аналитик занимается подготовкой данных, он выполняет ряд действий по обработке и создает значительный объем данных. Некоторые данные занимают много памяти — сотни мегабайтов и даже гигабайтов, и повторное выполнение кода, генерирующего данные, может занять много времени. Было бы удобно сохранить данные в форме файлов на компьютере.

В разделе 11.2 рассматривалась запись табличных данных в файлы. Однако данные также могут существовать и в других формах, например `dict`, `list` и `tuple`, а также классов и функций. Следовательно, должен существовать более общий механизм хранения данных. В этом разделе вы узнаете о механизме *консервации* (pickling), который позволяет сохранять различные формы данных Python.

11.3.1. Консервация объектов для сохранения данных

Словом «консервация» называют обработку пищевых продуктов для долгосрочного хранения. В Python этим термином обозначается процесс преобразования объектов в двоичный формат для сохранения данных. При нормальной остановке программы несохраненные данные теряются, что может быть нежелательно. Обработка некоторых данных требует значительного времени, и полученные данные желательно сохранить, чтобы легко загрузить в будущем. В этом разделе вы узнаете, как консервировать данные с использованием встроенного модуля `pickle`.

ОСНОВНЫЕ ПОНЯТИЯ *Консервацией* (pickling) называется процесс создания двоичного представления существующих объектов для сохранения данных.

Консервировать можно практически любой объект в Python. Допустим, вы используете разные формы данных для хранения информации задачи в таск-менеджере. Посмотрим, как консервировать эти объекты:

```
import pickle  ◄——  Импортирует модуль

task_tuple = (1001, "Homework", 5)
task_dict = {'task_id': '1002', 'title': 'Laundry', 'urgency': 3}

with open("task_tuple_saved.pickle", "wb") as file:
    pickle.dump(task_tuple, file)

with open("task_dict_saved.pickle", "wb") as file:
    pickle.dump(task_dict, file)
```

В этом фрагменте кода мы создаем один кортеж и один объект `dict` для консервации. Обратите внимание на два ключевых момента:

- Функция `dump` сохраняет данные в файле. При работе с файлом функция `open` используется для создания файлового объекта, который устанавливает связь с нужным вам файлом.
- При открытии файла следует использовать режим `"wb"`. Этот режим означает, что мы выполняем операции записи, а файл должен храниться в двоичном формате. А вот при работе с текстовыми файлами вам не нужно беспокоиться о выборе режима, так как по умолчанию используется режим `"t"` для текстовых данных.

После выполнения кода в текущем каталоге появляются два новых файла — task_tuple_saved.pickle и task_dict_saved.pickle. Попытавшись открыть их в текстовом редакторе, вы не увидите ничего осмысленного. Аналогичным образом при попытке открыть графическое изображение в текстовом редакторе вы увидите отдельные понятные фрагменты, чередующиеся с бессмысленным текстом, потому что данные хранятся в двоичном виде. Как использовать данные, сохраненные в `pickle`-файлах? Об этом рассказано в следующем разделе.

11.3.2. Восстановление данных посредством расконсервации

Для сохранения объектов в файлах используется процесс, называемый консервацией. Когда позднее вам снова понадобятся данные, вы загрузите их из законсервированного файла — процесс, обратный консервации, называется *расконсервацией* (unpickling). В этом разделе вы научитесь восстанавливать данные посредством расконсервации.

При рассмотрении сериализации данных формата JSON в разделе 9.4 функция `dump` использовалась для создания JSON-файла. Эта функция имеет такую же сигнатуру вызова, как функция `dump` для консервации. При чтении файлов

в формате JSON использовалась функция `load`. Естественно, что при расконсервации также используется функция `load`:

```
with open("task_tuple_saved.pickle", "rb") as file:
    task_tuple_loaded = pickle.load(file)

with open("task_dict_saved.pickle", "rb") as file:
    task_dict_loaded = pickle.load(file)
```

Для расконсервации файл должен быть открыт в режиме чтения. Помните, что `pickle`-файлы содержат двоичные данные, поэтому они должны открываться в режиме `"rb"`. В отличие от функции `dump`, которая возвращает `None`, при вызове функции `load` для объекта файла мы ожидаем получить данные. По этой причине возвращаемое значение присваивается переменной.

Чтобы проверить точность сохранения данных, сравним расконсервированные данные с исходными объектами. Восстановленные объекты должны быть равны исходным. Это важный момент, так как он доказывает, что исходные данные можно воссоздать после консервации:

```
assert task_tuple == task_tuple_loaded
assert task_dict == task_dict_loaded
```

Можно ли консервировать экземпляры пользовательских классов? Да, можно. Рассмотрим следующий пример:

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

task = Task("Laundry", 3)

with open("task_class_saved.pickle", "wb") as file:
    pickle.dump(task, file)

with open("task_class_saved.pickle", "rb") as file:
    task_class_loaded = pickle.load(file)

assert task.__dict__ == task_class_loaded.__dict__
assert task is not task_class_loaded
```

В этом фрагменте кода мы консервируем и расконсервируем экземпляр класса `Task`. Исходный объект и законсервированный/расконсервированный объект содержат одинаковые атрибуты, как видно из сравнения их словарных представлений. Обратите внимание: это не один и тот же объект, что подтверждается проверкой тождественности (`is not`).

Хотя возможна и консервация экземпляров пользовательских классов, она требует особого внимания. Дело в том, что при расконсервации объектов встроенных классов Python уже знает, как распаковывать их, потому что их типы

известны. Но Python может ничего не знать о ваших пользовательских классах, если вы не определяете их при расконсервации. То есть при расконсервации экземпляра — в нашем случае экземпляра класса `Task` — произойдет ошибка, если в пространстве имен отсутствует класс `Task`. Рассмотрим следующий пример:

```
del Task  ◄——  Удаляет Task из глобального пространства имен

with open("task_class_saved.pickle", "rb") as file:
    task_class_loaded = pickle.load(file)

# ERROR: AttributeError: Can't get attribute 'Task' on <module '__main__'
➥ (built-in)>
```

Чтобы смоделировать ситуацию расконсервации экземпляра класса, определение которого отсутствует, мы удаляем класс `Task` из глобального пространства имен командой `del Task`. После этого получить пользовательский экземпляр при расконсервации не удастся, так как Python не находит определение класса `Task` для создания экземпляра.

> **СОПРОВОЖДАЕМОСТЬ** Если вы расконсервируете экземпляры пользовательского класса, убедитесь в том, что класс определен в соответствующем пространстве имен.

Когда мы изучали преобразование данных JSON, вы узнали, что функции `dump` и `load` предназначены для операций с файлами JSON, а `dumps` и `loads` — для операций со строками JSON. В механизме консервации имеются аналогичные функции с такими же именами: `dump` и `load` для консервации файлов, а `dumps` и `loads` для консервации строк в двоичной форме (то есть в виде байтов, листинг 11.13), как показано на рис. 11.4.

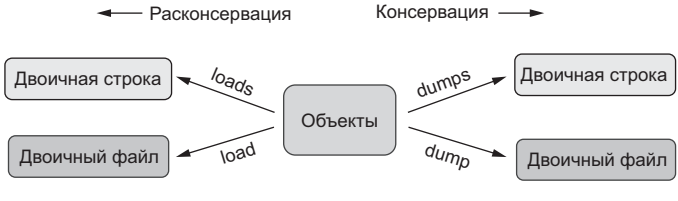


Рис. 11.4. Консервация и расконсервация в форме строк и файлов. При консервации `dumps` вызывается для создания двоичной строки, а `dump` — для создания двоичного файла. При расконсервации `loads` вызывается для создания объекта из двоичной строки, а `load` — для создания объекта из двоичного файла

Предыдущие примеры были посвящены консервации данных в файлах. В следующем разделе приводятся примеры использования двоичных строк при консервации данных.

11.3.3. Достоинства и недостатки консервации

Вы видели, как работает консервация/расконсервация для сохранения данных. Но у нее есть и достоинства, и недостатки, которые важно знать. В этом разделе рассматриваются важнейшие аспекты консервации, которые помогут определить, насколько уместно применять ее при сохранении данных в проекте.

Совместимость с большинством объектов

JSON — распространенный механизм хранения и обмена данными — совместим со встроенными типами данных, но он не работает с пользовательскими классами, если только вы не предоставите конкретные инструкции по сериализации JSON, например, передачей аргумента `default` при вызове `dump` или `dumps` (раздел 9.3). Более того, JSON в исходном виде не может работать с некоторыми объектами (например, функциями). Напротив, консервация изначально совместима с гораздо большим количеством объектов. Чтобы убедиться в гибкости консервации, взгляните на пример консервации простой функции в листинге 11.13.

Листинг 11.13. Консервация функции в байты

```
def doubler(x):
    return x * 2

doubler_pickle = pickle.dumps(doubler)
print(doubler_pickle)

# output: b'\x80\x04\x95\x18\x00\x00\x00\x00\x00\x00\x00\x8c\x08
➥ __main__\x94\x8c\x07doubler\x94\x93\x94.'
```

Как показано в этом коде, функция `doubler` консервируется в данных `bytes`. Результат напоминает по виду строку, но начинается с префикса `b`, который показывает, что это объект `bytes`. Объект `bytes` можно расконсервировать и восстановить функцию, которая должна делать то же самое, что `doubler`:

```
doubler_loaded = pickle.loads(doubler_pickle)

assert doubler_loaded(5) == doubler(5)
```

Как было показано выше, консервация работает с пользовательскими классами без каких-либо конкретных инструкций (листинг 11.14) в отличие от сериализации JSON, которая требует особых инструкций для кодирования экземпляров (раздел 9.3). Однако консервация работает не со всеми объектами в Python. Например, вы не сможете законсервировать модуль:

```
import os

os_dumped = pickle.dumps(os)

# ERROR: TypeError: cannot pickle 'module' object
```

Также невозможно консервировать объекты файлов и подключений к базам данных, так как они используют ресурсы динамически, с чем консервация работать не способна. Помимо этих ограничений, консервацию можно применить к большинству видов объектов, и она становится универсальным механизмом сохранения данных.

Безопасность данных

При работе с любыми данными их безопасность — это важный фактор, который легко упустить из виду. Когда вы получаете файлы, стоит задуматься над тем, насколько они безопасны. Тот же принцип применяется к законсервированным файлам — следует с осторожностью относиться к безопасности таких данных.

Так как консервация позволяет сохранить практически любой объект, у хакера появляется возможность внедрить в него вредоносный код. В разделах 11.3.1 и 11.3.2 вы уже видели, как консервация работает с такими встроенными типами данных, как `tuple` и `dict`. Но если кто-либо создаст пользовательский класс, он сможет определить специализированное поведение, способное взломать работу системы консервации. Рассмотрим следующий пример:

```
import os

class MaliciousTask:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

    def __reduce__(self):
        print("__reduce__ is called")
        return os.system, ('touch hacking.txt',) ◄— Создает кортеж из одного элемента
```

В этом фрагменте злоумышленник определил класс `MaliciousTask`. Этот класс реализует специальный метод `__reduce__`, участвующий в процессе консервации. В случае его выполнения на вашем компьютере будет создан файл hacking.txt. В данном примере файл пуст, но в нем мог бы содержаться вредоносный код, который повредит вашу компьютерную систему!

Если вы не заметите вредоносный код и попытаетесь расконсервировать экземпляр этого класса, возникнет угроза безопасности вашего компьютера из-за лишнего файла, созданного при вызове `__reduce__`. Эффект продемонстрирован в листинге 11.14.

Листинг 11.14. Консервация экземпляра пользовательского класса

```
malicious_task = MaliciousTask("Set fire", 5)

with open("test_malicious.pickle", "wb") as file:
    pickle.dump(malicious_task, file)

# Вывод: __reduce__ is called
```

Обратите внимание: я включил вывод `"__reduce__ is called"`, чтобы показать, что `__reduce__` участвует в консервации. Команда создания потенциально вредоносного файла является частью законсервированного файла. При его расконсервации возникает следующая проблема:

```
with open("test_malicious.pickle", "rb") as file:
    pickle.load(file)
```

Если после расконсервации файла проверить содержимое каталога, вы увидите, что в нем появился файл hacking.txt! Впрочем, настоящий вредоносный код не оставит столь очевидных следов. Отсюда следует, что при консервации и расконсервации объектов необходима осторожность. На практике рекомендуется консервировать только объекты, полученные из доверенных источников, например встроенные, созданные вами или полученные из надежных сторонних пакетов.

Затраты памяти и скорость

Еще одно преимущество консервации — меньшие затраты памяти и ускорение чтения/записи по сравнению с такими текстовыми форматами, как CSV. Ранее я уже несколько раз упоминал о pandas — одном из популярных пакетов Python для data science. Основная модель данных pandas — `DataFrame` — представляет табличную структуру данных. Объекты `DataFrame` можно сохранять в CSV-файлах или файлах `pickle`. В общем случае чтение и запись данных с файлами `pickle` выполняются намного быстрее операций с CSV-файлами, к тому же файлы `pickle` обычно занимают меньше места, чем CSV-файлы с тем же объемом данных.

11.3.4. Обсуждение

Консервация — удобный механизм хранения данных, совместимый с большинством видов объектов Python, включая пользовательские классы. Конечно, у консервации есть свои достоинства и недостатки. При работе над проектами, связанными с данными, консервация обычно является отличным вариантом из-за ускорения чтения/записи по сравнению с CSV-файлами. Напомним, что консервация данных из ненадежных источников может создать угрозу для безопасности.

11.3.5. Задача

Роджер работает аналитиком в сфере кибербезопасности в больнице. Он оценивает риски безопасности, связанные с применением консервации в Python. Роджер пытается законсервировать экземпляр класса `MaliciousTask`, который добавляет файл (hacking.txt) в текущий рабочий каталог, как было показано в листинге 11.14. Как ему изменить класс, чтобы он удалял файл hacking.txt в ходе консервации?

ПОДСКАЗКА Мы использовали команду touch hacking.txt для создания этого файла. Для удаления файла можно воспользоваться командой rm hacking.txt. Не забудьте, где должна размещаться эта команда.

11.4. КАК УПРАВЛЯТЬ ФАЙЛАМИ НА СВОЕМ КОМПЬЮТЕРЕ

Над каким бы проектом вы ни работали, рано или поздно вам неизбежно придется работать с файлами. В конце концов, файлы являются универсальными контейнерами для хранения упорядоченной информации. В предыдущих разделах вы научились читать данные из файлов и записывать их в файлы. Однако я еще не упоминал об управлении файлами в целом (не их содержимым, а самими файлами), а также об управлении каталогами, например о перемещении и копировании файлов.

Рассмотрим следующий сценарий. Допустим, вы проводите научный эксперимент, в котором каждый участник выполняет тест на скорость реакции. Тест состоит из нескольких испытаний, и после его проведения программа генерирует ряд файлов. Так как эксперимент проводится с несколькими участниками, каталог содержит следующие файлы:

```
subject_123.config
subject_123.dat
subject_123.txt
subject_124.config
subject_124.dat
subject_124.txt
subject_125.config
subject_125.dat
subject_125.txt
```

Нас интересуют файлы определенного типа, а именно: когда вы работаете с набором данных, требуется извлечь только файлы данных (`.dat`) и переместить их в новый каталог. Также требуется удалить текстовые файлы (`.txt`), потому что они уже не нужны.

В данном разделе мы рассмотрим эти задачи и стандартные методы работы с файлами. Следует учесть, что Python является языком общего назначения, поэтому в области управления файлами существует множество решений, основанных на разных библиотеках, как, например, `os` и `pathlib`. Я сосредоточусь на общих решениях.

11.4.1. Создание каталогов и файлов

Чтобы задать общую линию изложения для всего раздела, начнем с создания нового каталога и набора фиктивных файлов. Если ранее вы использовали

модуль os, при работе с путями к файлам и каталогам я рекомендую использовать модуль pathlib; это более компактный модуль, специализирующийся на работе с путями. При помощи pathlib можно легко создать новый каталог:

```
from pathlib import Path
data_folder = Path("data")
data_folder.mkdir()  ◄——  Создает каталог
```

Центральной моделью данных в pathlib является Path — класс, предназначенный для операций с путями. Так, чтобы использовать Path для каталога, вызовите метод mkdir, который создает папку данных в текущем каталоге. Чтобы проверить ее существование на программном уровне, вызовите метод exists:

```
assert data_folder.exists()
```

Когда каталог будет создан, можно переходить к созданию фиктивных файлов, которые будут использоваться для выполнения операций в следующем разделе:

```
subject_ids = [123, 124, 125]
extensions = ["config", "dat", "txt"]
for subject_id in subject_ids:
    for extension in extensions:
        filename = f"subject_{subject_id}.{extension}"
        filepath = data_folder / filename  ◄——  Создает путь к файлу
        with open(filepath, "w") as file:
            file.write(f"It's the file {filename}.")
```

К настоящему моменту вы уже должны знать, как создавать файлы при помощи функции open с инструкцией with (раздел 11.1). Следует заметить, что путь к файлу конструируется операцией путь_к_каталогу/имя_файла. Как известно, в Windows и macOS используются разные символы для разделения уровней иерархии каталогов: data\subject_123.dat и data/subject_123.dat. Когда вы создаете путь к файлу конструкцией путь_к_каталогу/имя_файла, эта операция не зависит от операционной системы, то есть один код будет работать на любой из этих платформ. А если, скажем, создать путь в виде data\subject_123.dat, может оказаться, что ваш код не работает в другой системе. Кросс-платформенная совместимость — еще одно преимущество pathlib перед модулем os, зависящим от платформы (в нем в качестве путей используются неформатированные строки).

11.4.2. Получение списка файлов по шаблону

На следующем шаге требуется отобрать в каталоге все файлы .dat, чтобы обработать их для получения результатов тестирования. Чтобы получить все файлы определенного типа, вызывается метод glob для пути к каталогу, в котором задается шаблон имен файлов. Все файлы, соответствующие этому шаблону, будут включены в результаты, как показано в листинге 11.15.

Листинг 11.15. Получение списка файлов заданного типа

```
data_folder = Path("data")

data_files = data_folder.glob("*.dat")
print("Data files:", data_files)  <-- Создает объект-генератор

for data_file in data_files:
    print(f"Processing file: {data_file}")
    # Здесь выполняется необходимая обработка данных

# Выводимые строки:
Data files: <generator object Path.glob at 0x100b5c040>  <-- На вашем компьютере адрес памяти будет другим

Processing file: data/subject_124.dat
Processing file: data/subject_125.dat
Processing file: data/subject_123.dat
```

Указанный шаблон *.dat отбирает файлы с расширением .dat. Заметим, что список файлов, удовлетворяющих этому шаблону, образует генератор, который может использоваться для цикла for. Из выводимого сообщения видно, что программа действительно получила все файлы .dat. Один потенциальный недостаток заключается в том, что список не отсортирован, а это усложняет понимание того, какие файлы были обработаны. В порядке усовершенствования отсортируем генератор для улучшения организации файлов:

```
data_files = data_folder.glob("*.dat")  <-- Создает генератор заново

for data_file in sorted(data_files):  <-- Сортирует генератор для создания списка
    print(f"Processing file: {data_file}")
    # Здесь выполняется необходимая обработка данных

# Выводимые строки:
Processing file: data/subject_123.dat
Processing file: data/subject_124.dat
Processing file: data/subject_125.dat
```

НАПОМИНАНИЕ Когда все элементы, выдаваемые генератором, будут исчерпаны, генератор необходимо создать заново, чтобы он мог снова выдавать свои элементы.

11.4.3. Перемещение файлов в другой каталог

Чтобы организовать данные нашего проекта в научном эксперименте, следует разместить данные участников в отдельных каталогах. Например, для участника с идентификатором 123 все данные должны находиться в каталоге subject_123. В этом разделе вы научитесь перемещать файлы.

Перемещение файлов осуществляется «переименованием» пути к файлу. Если переименовать файл data/subject_123.dat в subjects/subject_123/subject_123.dat, то он

переместится из каталога data в каталог subject_123. Используя эту информацию, рассмотрим решение в листинге 11.16. Обратите внимание на использование метода `mkdir`: он позволяет создать многоуровневый каталог даже в том случае, если некоторые промежуточные уровни не существуют. Аргументу `parents` при вызове `mkdir` в листинге 11.16 присваивается значение `True`; все недостающие промежуточные уровни пути будут созданы в случае необходимости.

Листинг 11.16. Перемещение файлов в заданный каталог

```
subject_ids = [123, 124, 125]
data_folder = Path("data")

for subject_id in subject_ids:
    subject_folder = Path(f"subjects/subject_{subject_id}")
    subject_folder.mkdir(parents=True, exist_ok=True)

    for subject_file in data_folder.glob(f"*{subject_id}*"):
        filename = subject_file.name
        target_path = subject_folder / filename
        _ = subject_file.rename(target_path)
        print(f"Moving {filename} to {target_path}")
```

Создает каталог subject_

Получает имя файла

Конструирует целевой путь

```
# Выводимые строки:
Moving subject_123.config to subjects/subject_123/subject_123.config
Moving subject_123.dat to subjects/subject_123/subject_123.dat
Moving subject_123.txt to subjects/subject_123/subject_123.txt
Moving subject_124.config to subjects/subject_124/subject_124.config
Moving subject_124.dat to subjects/subject_124/subject_124.dat
Moving subject_124.txt to subjects/subject_124/subject_124.txt
Moving subject_125.dat to subjects/subject_125/subject_125.dat
Moving subject_125.config to subjects/subject_125/subject_125.config
Moving subject_125.txt to subjects/subject_125/subject_125.txt
```

После выполнения этого кода в текущем каталоге должна появиться новая папка subjects, которая содержит три папки для трех разных участников. Перемещение файла обычно состоит из четырех шагов (рис. 11.5): определение перемещаемого файла, получение имени файла, конструирование нового имени файла и переименование файла.

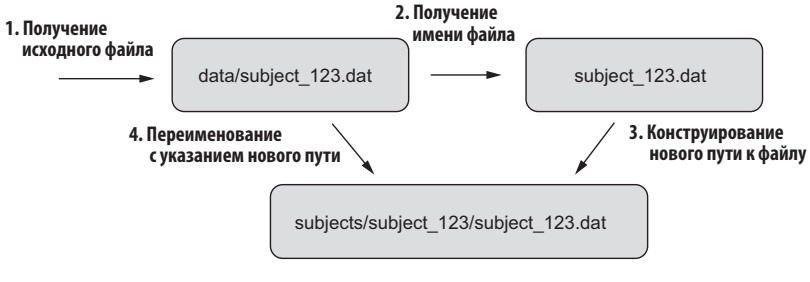


Рис. 11.5. Общий процесс перемещения файла. По сути, происходит переименование файла с изменением пути

11.4.4. Копирование файлов в другой каталог

Копирование файлов позволяет оставить на месте исходный файл и создать его новую копию. Допустим, вместо перемещения файлов из data в папку subjects будет выполнено копирование данных. Советую создать исходные файлы данных заново, если вы захотите повторить приведенные примеры. Модуль `shutil`, который будет использоваться в этом разделе, предоставляет высокоуровневый программный интерфейс (API) для операций с файлами.

Модуль содержит метод `copy` с сигнатурой вызова `copy(src, dst)`, где `src` обозначает исходный файл, а `dst` — итоговый путь. При помощи этого метода можно скопировать файлы в папку каждого участника, как показано в листинге 11.17.

Листинг 11.17. Копирование файлов в другой каталог

```
import shutil

shutil.rmtree("subjects")  <-- Удаляет папку вместе с ее содержимым

subject_ids = [123, 124, 125]
data_folder = Path("data")

for subject_id in subject_ids:
    subject_folder = Path(f"subjects/subject_{subject_id}")
    subject_folder.mkdir(parents=True, exist_ok=True)

    for subject_file in data_folder.glob(f"*{subject_id}*"):
        filename = subject_file.name
        target_path = subject_folder / filename
        _ = shutil.copy(subject_file, target_path)  <-- Если возвращаемое значение функции не используется, укажите символ подчеркивания
        print(f"Copying {filename} to {target_path}")
```

```
# Выводимые строки:
Copying subject_123.config to subjects/subject_123/subject_123.config
Copying subject_123.dat to subjects/subject_123/subject_123.dat
Copying subject_123.txt to subjects/subject_123/subject_123.txt
Copying subject_124.config to subjects/subject_124/subject_124.config
Copying subject_124.dat to subjects/subject_124/subject_124.dat
Copying subject_124.txt to subjects/subject_124/subject_124.txt
Copying subject_125.dat to subjects/subject_125/subject_125.dat
Copying subject_125.config to subjects/subject_125/subject_125.config
Copying subject_125.txt to subjects/subject_125/subject_125.txt
```

Как показано в листинге 11.17, функция `rmtree` используется для удаления папки вместе с содержимым, так как `rmtree` не интересует, пуст каталог или нет. А вот при попытке использовать `Path.rmdir` для непустого каталога возникнут проблемы. В этом нетрудно убедиться:

```
Path("subjects").rmdir()
# ERROR: OSError: [Errno 66] Directory not empty: 'subjects'
```

В листинге 11.16 выполнялось перемещение файлов. Копирование файлов выполняется по той же схеме: определение файлов, получение имен, конструирование новых имен и использование функции `copy` из модуля `shutil`.

11.4.5. Удаление файлов

В начале раздела 11.4 уже упоминалось о том, что одной из целей задачи является удаление файлов .txt в папке данных (а именно файлов, которые могут содержать конфиденциальные данные участника) по соображениям безопасности. В обобщенном виде требуется удалить конкретные виды файлов, и эта задача рассматривается в данном разделе.

Класс `Path` предоставляет метод `unlink` для удаления файлов. Чтобы использовать эту возможность, необходимо получить экземпляры объектов `Path` и вызвать для них `unlink`:

```
data_folder = Path("data")

for file in data_folder.glob("*.txt"):
    before = file.exists()
    file.unlink()
    after = file.exists()
    print(f"Deleting {file}, existing? {before} -> {after}")

# Выводимые строки:
Deleting data/subject_123.txt, existing? True -> False
Deleting data/subject_124.txt, existing? True -> False
Deleting data/subject_125.txt, existing? True -> False
```

Чтобы показать, что удаление работает, мы проверяем существование файлов до и после удаления. Как видите, каждый файл существует до удаления, но пропадает после удаления.

11.4.6. Обсуждение

Многие операции с файлами выполняются вручную, но при этом легко упустить, что делалось с теми или иными файлами. Конечно, описания операций можно записывать, но вести записи обо всем, что делалось, слишком скучно и неудобно. Чтобы операции с файлами были более воспроизводимыми и контролируемыми, следует выполнять их из программного кода.

11.4.7. Задача

Касси использует Python для управления файлами на своем компьютере. Один из уроков, который она усвоила, — при копировании файлов в другую папку не следует перезаписывать другие файлы. Иначе говоря, может оказаться, что в итоговой папке уже существуют файлы с такими же именами, как у копируемых. Более того, может оказаться, что эти файлы уже прошли обработку и содержат

новые данные. Как ей обновить код в листинге 11.17, чтобы копировались только файлы, не существующие в итоговой папке?

> **ПОДСКАЗКА** Чтобы проверить, существует ли файл, можно вызвать `exists` для экземпляра `Path`.

11.5. КАК ПОЛУЧИТЬ МЕТАДАННЫЕ ФАЙЛА

В разделе 11.4 вы научились выполнять операции с файлами на компьютере. Для операций копирования и перемещения мы получали имя файла из атрибута `name` объекта `Path`. Кроме имени, файл содержит метаданные, которые могут понадобиться в конкретных ситуациях. Например, вам нужно будет узнать каталог, в котором находится файл, чтобы построить другой путь для обращения к другому файлу в том же каталоге.

Допустим, вы продолжили работать с экспериментальными данными из раздела 11.4. Требуется обработать все файлы данных (.dat) в папке. Кроме этого, для каждого участника необходимо получить еще дополнительный файл конфигурации (.config). Конечно, можно вызвать `glob` для получения списка файлов .dat. Но как без лишнего труда найти соответствующий файл .config для каждого участника? В этом разделе вы получите ответ на этот вопрос, а также научитесь вызывать другие операции, связанные с обращениями к метаданным файлов.

11.5.1. Получение информации, относящейся к имени файла

Под «информацией, относящейся к имени файла» я подразумеваю каталог, имя и расширение файла. Эти данные являются атрибутами класса `Path`. Примеры кода помогут вам лучше понять их.

Начнем с файла данных subjects/subject_123/subject_123.dat. Как получить файл subjects/subject_123/subject_123.config? Эти два файла имеют одинаковые каталоги и имена, но разные расширения. На основании этих характеристик я предлагаю решение, приведенное в листинге 11.18.

Листинг 11.18. Получение информации об именах файлов

```
from pathlib import Path

subjects_folder = Path("subjects")

for dat_path in subjects_folder.glob("**/*.dat"):  <-- Получает все файлы данных

    subject_dir = dat_path.parent  <-- Получает каталог для файла
```

```
    filename = dat_path.stem  <-- Получает имя файла

    config_path = subject_dir / f"{filename}.config"

    print(f"{subject_dir} & {filename} -> {config_path}")

    dat_exists = dat_path.exists()

    config_exists = config_path.exists()

    with open(dat_path) as dat_file, open(config_path) as config_file:  <-- Получает оба файла
        print(f"Process {filename}: dat? {dat_exists}, config?
        ➥ {config_exists}\n")
        # Обработка данных участника
```

```
# Выводимые строки:
subjects/subject_125 & subject_125 -> subjects/subject_125/subject_125.config
Process subject_125: dat? True, config? True

subjects/subject_124 & subject_124 -> subjects/subject_124/subject_124.config
Process subject_124: dat? True, config? True

subjects/subject_123 & subject_123 -> subjects/subject_123/subject_123.config
Process subject_123: dat? True, config? True
```

В листинге 11.18 из выводимого сообщения видно, что мы обрабатываем данные каждого участника, обращаясь к файлам .dat и .config. Заслуживают внимания некоторые обстоятельства:

- Так как в subjects_folder содержатся другие папки, при обращении к файлам в этих подпапках шаблон включает символы `**/`, означающие, что файлы находятся в подпапках.
- Для каждого экземпляра `Path` можно обратиться к его атрибуту `parent`, который возвращает каталог для заданного пути.
- Для каждого экземпляра `Path` можно обратиться к его атрибуту `stem`, который возвращает только имя файла без расширения.
- В инструкции `with` два файла открываются одновременно. При этом создаются два объекта файлов, с которыми можно работать одновременно.

Вы можете получить все имя файла, включая расширение, из атрибута `name` (листинг 11.17) или только расширение из атрибута `suffix`, как показано ниже (учтите, что расширение включает символ «`.`»):

```
dat_path = Path("subjects/subject/subject_123.dat")

assert dat_path.suffix == ".dat"
```

На рис. 11.6 показано, какие атрибуты соответствуют данным об именах файлов.

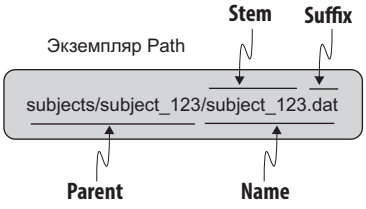


Рис. 11.6. Чтение данных, относящихся к именам файлов, из экземпляра класса Path. Информация читается из атрибутов `parent` (каталог), `name` (имя файла, включая расширение), `stem` (имя файла без расширения) и `suffix` (расширение файла)

11.5.2. Получение информации о размере файла и времени изменения

Если вы запустите файловый менеджер на своем компьютере, то увидите несколько столбцов с дополнительными сведениями, помимо имени, например размер файла и время последнего обновления. Эти метаданные могут пригодиться в разных ситуациях, некоторые из них рассматриваются в данном разделе.

В эксперименте файлы данных участников должны иметь постоянный размер, если все было записано правильно. Это позволяет даже без открытия файла для анализа содержимого проверить размер файла, чтобы убедиться в целостности данных перед дальнейшей обработкой. Функция, приведенная в листинге 11.19, решает эту задачу.

Листинг 11.19. Функция для проверки размера файлов

```
def process_data_using_size_cutoff(min_size, max_size):
    data_folder = Path("data")
    for dat_path in data_folder.glob("*.dat"):
        filename = dat_path.name
        size = dat_path.stat().st_size  <-- Получает размер файла
        if min_size < size < max_size:  <-- Последовательные сравнения
            print(f"{filename}, Good; {size},  within
            ➥ [{min_size}, {max_size}]")
        else:
            print(f"{filename}, Bad; {size}, outside
            ➥ [{min_size}, {max_size}]")
```

В этом фрагменте кода функция `stat()` вызывается для получения связанных со статусом данных о файле, где `st_size` содержит размер в байтах. При помощи этой функции осуществляется несколько вариантов проверки целостности данных:

```
process_data_using_size_cutoff(20, 40)
# Выводимые строки:
subject_124.dat, Good; 30,  within [20, 40]
subject_125.dat, Good; 30,  within [20, 40]
subject_123.dat, Good; 30,  within [20, 40]
```

```
process_data_using_size_cutoff(40, 60)

# Выводимые строки:
subject_124.dat, Bad; 30, outside [40, 60]
subject_125.dat, Bad; 30, outside [40, 60]
subject_123.dat, Bad; 30, outside [40, 60]
```

Как видите, когда мы устанавливаем допустимый диапазон 20–40, все файлы проходят проверку, так как их размер равен 30. Если определить окно размера 40–60, то все файлы будут считаться недопустимыми.

Иногда требуется отобрать файлы на основании времени модификации их содержимого. Чтобы получить метаданные, относящиеся к времени, вызовем метод `stat` для экземпляра `Path`:

```
import time

subject_dat_path = Path("data/subject_123.dat")

modified_time = subject_dat_path.stat().st_mtime  <-- Время изменения содержимого

readable_time = time.ctime(modified_time)  <-- Преобразует в человекочитаемое время

print(f"Modification time: {modified_time} -> {readable_time}")

# Вывод: Modification time: 1652123144.9999998 -> Mon May  9 14:05:44 2022
                                                  На вашем компьютере значение будет другим
```

В этом коде мы обращаемся к атрибуту `st_mtime`, который содержит время изменения файла, вернее, его содержимого (не имени файла или других метаданных). Значение задается в секундах от начала эпохи: 1 января 1970 г., 00:00:00 (UTC). Для преобразования этого значения в понятный человеку формат воспользуйтесь функцией `ctime` из модуля `time`.

11.5.3. Обсуждение

В этом разделе мы рассмотрели такие метаданные, как каталог для хранения файла, имя файла, его расширение, размер и время модификации содержимого. Однако следует заметить, что в метаданных файла хранится много других видов информации, включая разрешения доступа к файлу. Возможно, для ваших проектов будет достаточно метаданных, описанных в этом разделе. А когда потребуется обратиться к метаданным файла, знайте, что для этого вызывается метод `stat` для экземпляра класса `Path`.

11.5.4. Задача

Альберт — студент-старшекурсник. Ему нравится использовать Python для управления компьютером на программном уровне. Как ему написать функцию, которая находит в заданном каталоге файлы, измененные за последние 24 часа?

ПОДСКАЗКА С модулем `time` вызывается функция `time` для получения времени от начала эпохи в секундах. Время изменения содержимого файла можно сравнить с этим значением с 24-часовой поправкой. Помните, что при сравнении должно использоваться количество секунд в 24 часах.

ИТОГИ

- При выполнении операций чтения/записи с файлами используйте инструкцию `with`, которая автоматически закрывает файл при помощи менеджера контекста.
- По умолчанию файлы открываются в режиме `"r"` (чтение). Для выполнения любых операций записи необходимо выбрать режим `"w"` (запись) или режим `"a"`(присоединение). В режиме `"a"` данные добавляются в конец файла.
- Встроенный модуль `csv` специализируется на чтении и записи данных в формате CSV. В книге я не рассказываю об этом, но если вам понадобится выполнять числовые вычисления и обработку данных, советую рассмотреть возможность использования сторонней библиотеки, например pandas.
- Если CSV-файлы содержат заголовки, рекомендуется использовать обрабатывающий их объект `csv.DictReader` вместо другого популярного объекта чтения данных `csv.reader`.
- Объекты `csv.writer` и `csv.DictWriter`, обратные по отношению к `csv.reader` и `csv.DictReader`, предназначены для создания CSV-файлов. Второй лучше подходит для работы с заголовками.
- Консервация — встроенный механизм для хранения объектов Python в двоичных данных. По сравнению с JSON консервация обладает большей гибкостью, потому что она поддерживает больше типов данных, в том числе функции.
- Будьте осторожны с безопасностью данных при использовании консервации. Не консервируйте и не расконсервируйте данные, полученные из потенциально небезопасных источников.
- Вместо использования CSV-файлов в качестве механизма хранения табличных данных используйте консервацию для экономии размера данных и повышения скорости чтения/записи.
- Встроенный модуль `pathlib` предоставляет различные методы и атрибуты в классе `Path`. Вы должны уметь использовать `pathlib` для выполнения операций управления файлами, таких как создание каталогов и перемещение файлов.
- Файл содержит не только собственно данные, но и имя, каталог, время изменения и другие метаданные, которые могут содержать нужную вам информацию. Вы должны знать, как получить эти данные при помощи класса `Path`.

Часть 5
Защита кодовой базы

Мы, программисты, несем ответственность за свой код. Это означает, что мы следим за его качеством и обеспечиваем его работоспособность при минимуме ошибок (а лучше вообще без них). Повышение качества кода обычно ведется по четырем направлениям:

- Регистрация важных событий во время выполнения программы позволяет узнать, что произошло, и быстро предоставить решение при возникновении каких-либо проблем.
- Интеграция обработки возможных исключений в программу предотвращает ее аварийное завершение.
- Отладка программы в фазе разработки — лучшее время для устранения ошибок, потому что вы еще хорошо помните, как работает код.
- Тщательное тестирование программы проверяет работоспособность каждой части до поставки продукта.

В части 5 будут рассмотрены все четыре направления для построения надежных и стабильных программ.

12 Ведение журнала и обработка исключений

В этой главе

- ✓ Ведение журнала (логирование)
- ✓ Форматирование журналов
- ✓ Обработка исключений
- ✓ Выдача исключений

Когда приложение переходит в фазу реальной эксплуатации (в продакшен), мы временно теряем контроль над своим продуктом, нам приходится полагаться только на его собственное поведение. Если вы были крайне внимательны во время разработки, возможно, вам повезло и у вас получился идеальный продукт, не содержащий ошибок. На практике такого почти никогда не бывает. Следовательно, мы должны знать, что в приложении могут возникнуть разнообразные проблемы, например аномальный всплеск трафика (для веб-приложений). Если такое случится, не паникуйте, а начинайте решать возникшие проблемы.

Не всегда есть возможность поговорить с пользователями, сообщившими о проблеме; но даже если это удастся сделать, предоставленная ими информация, скорее всего, окажется слишком ограниченной, чтобы позволить выявить причину проблемы. К счастью, мы изначально ожидали, что с продуктом может что-то пойти не так, а приложение ведет журнал действий пользователя и соответствующих событий. Это позволяет анализировать, с чего указанные

проблемы начались. Записи в журнале играют важнейшую роль в обеспечении бесперебойной работы продукта и непрерывном контроле за его производительностью. Механизм ведения журнала (логирование, или журналирование) чрезвычайно полезен, поэтому он должен интегрироваться в приложение еще на стадии разработки. К тому же следует ожидать возникновения различных исключений из-за непредсказуемости пользовательского ввода. Например, может случиться, что пользователь попытается получить результат деления 1 на 0, что приведет к исключению `ZeroDivisionError`. Исключение необходимо правильно обработать, чтобы приложение могло продолжать выполнять свои функции. В этой главе будут рассмотрены механизмы логирования и обработки исключений.

12.1. КАК СЛЕДИТЬ ЗА РАБОТОЙ ПРОГРАММЫ ПОСРЕДСТВОМ ЛОГИРОВАНИЯ

Самое раздражающее в процессе разработки — попытки отладить проблему, которую вы не можете воспроизвести. Если повезет, у вас будут различные устные описания от конечных пользователей, не обладающих технической квалификацией. Впрочем, эти описания часто бессмысленны, потому что одна и та же проявляющаяся проблема может иметь разные первопричины.

Поэтому прежде, чем передавать приложение конечным пользователям, следует настроить в нем ведение журнала (логирование) для отслеживания эффективности работы приложения. Если пользователь столкнется с какими-либо проблемами в конкретном модуле вашего приложения, вы сможете загрузить из журнала информацию, относящуюся к этой проблеме, а это значительно ускорит ее решение. В этом разделе представлены важнейшие средства ведения журнала в Python.

12.1.1. Создание регистратора для логирования событий приложения

В Python нет ничего, кроме объектов, поэтому неудивительно, что для логирования тоже используется объект. А именно все операции с журналом выполняются через регистратор, то есть объект `Logger`. В этом разделе рассказано о том, как следует создавать объекты `Logger`.

В стандартной библиотеке Python модуль `logging` предоставляет функциональность ведения журнала. В модуле имеется класс `Logger`, и конструктор этого класса получает имя для создания экземпляра:

```
import logging

logger_not_good = logging.Logger("task_app")
```

Этот фрагмент создает регистратор. Но почему я назвал его `logger_not_good` (плохой регистратор)? Прежде чем я перейду к объяснению, посмотрите, как выглядит правильный способ создания регистратора:

```
logger_good = logging.getLogger("task_app")
```

Здесь мы вызываем функцию `getLogger`, передавая имя для регистратора. Причина, по которой рекомендуется использовать `getLogger` вместо вызова конструктора, заключается в том, что для ведения журнала должен использоваться общий экземпляр класса `Logger`. Может оказаться, что в приложении или в модуле регистратор придется использовать в нескольких местах. Если использовать конструктор, вы создадите несколько разных регистраторов, как в следующем примере:

```
logger0 = logging.Logger("task_app")
logger1 = logging.Logger("task_app")
logger2 = logging.Logger("task_app")

assert logger0 is not logger1
assert logger1 is not logger2
assert logger0 is not logger2
```

Эти сравнения можно объединить в одно сравнение при помощи операций AND

Эти регистраторы придется настраивать по отдельности (настройка рассматривается в разделе 12.2), а также следить за тем, чтобы они имели одинаковую конфигурацию для правильной работы. Однако нет никаких причин использовать несколько регистраторов в одном модуле; все операции должны выполняться одним регистратором. Как показывает этот пример, использование `getLogger` гарантирует, что все вызовы будут получать один и тот же регистратор:

```
logger0_good = logging.getLogger("task_app")
logger1_good = logging.getLogger("task_app")
logger2_good = logging.getLogger("task_app")

assert logger0_good is logger1_good is logger2_good
```

Сравнения `is` показывают, что регистратор остается одним и тем же, сколько бы раз вы ни вызывали `getLogger`. Так как во всех случаях используется один регистратор, его можно настроить один раз, и в процессе работы приложения он будет вести себя одинаковым образом на протяжении всего своего жизненного цикла.

Если вы создаете регистратор уровня модуля для каждого модуля приложения, я рекомендую делать это при помощи вызова `logging.getLogger(__name__)`, где `__name__` — специальный атрибут для имени модуля. Например, если модулю присвоено имя `taskier.py`, то атрибут `__name__` модуля будет содержать значение `taskier`.

СОПРОВОЖДАЕМОСТЬ Всегда используйте `getLogger`, чтобы получать один и тот же регистратор для вашего модуля или приложения. На уровне модуля для получения объекта `Logger` лучше использовать вызов `getLogger(__name__)`.

12.1.2. Использование файлов для хранения событий приложения

Во всех предыдущих главах я почти всегда использовал функцию `print` для вывода важных сообщений в процессе выполнения фрагментов кода. Допустим, вы хотите создавать запись в журнале (лог), когда пользователь удаляет задачу в таск-менеджере.

В листинге 12.1 приведена упрощенная версия кода.

Листинг 12.1. Создание лога с использованием print

```
class Task:
    def __init__(self, title):
        self.title = title

    def remove_from_db(self):
        # Операции для удаления задачи из базы данных
        task_removed = True
        return task_removed

task = Task("Laundry")
if task.remove_from_db():
    print(f"removed the task {task.title} from the database")
```

После успешного удаления задачи выводится сообщение. Однако такое решение работает только в активной фазе программирования, потому что сообщение выводится на консоль Python. Когда вы передаете приложение конечным пользователям, последовательный контроль выводимых сообщений становится практически невозможным. Приходится думать о том, чтобы сохранять события приложения в долгосрочном хранилище — файлах. В этом разделе я покажу, как направлять информацию о событиях в файлы.

> **ПРИМЕЧАНИЕ** После того как события будут сохранены в файлах, вы сможете проанализировать их столько раз, сколько понадобится; такое решение будет долгосрочным. Напротив, при использовании функции `print` события направляются на консоль, и при ее закрытии вся выведенная информация будет потеряна.

Допустим, у вас имеется регистратор, который обеспечивает все потребности приложения в логировании. Чтобы сохранить события в файле, необходимо определить конкретную конфигурацию регистратора, для чего следует назначить *обработчики* (handlers). Модуль `logging` включает класс с именем `FileHandler`; при помощи этого класса указывается файл, в котором регистратор будет сохранять события, как показано в листинге 12.2.

Листинг 12.2. Добавление файлового обработчика к регистратору

```
logger = logging.getLogger(__name__)

file_handler = logging.FileHandler("taskier.log")  <-- Задает файловый обработчик

logger.addHandler(file_handler)  <-- Добавляет обработчик к регистратору
```

Как показано в листинге 12.2, мы указываем, что все записи должны поступать в файл taskier.log, и связываем файл с регистратором вызовом метода `addHandler`. После выполнения этого кода в текущем каталоге должен появиться файл taskier.log. Теперь регистратор знает, где должны сохраняться записи, и мы проверим работу логирования, как показано в листинге 12.3.

Листинг 12.3. Запись информации в файл журнала

```
task = Task("Laundry")
if task.remove_from_db():
    logger.warning(f"removed the task {task.title} from the database")
```

В этом фрагменте кода записывается предупреждение, для чего используется вызов `logger.warning`. Открыв файл taskier.log, вы сможете просмотреть запись.

> **ЗАБЕГАЯ ВПЕРЕД** Каждое сообщение, сохраняемое в журнале, представляется экземпляром класса `LogRecord`. Форматирование записей в журнале рассматривается в разделе 12.2.3.

Если вы предпочитаете просмотреть запись на программном уровне, выполните следующий код. Вы уже знаете, как читать текстовые файлы (раздел 11.1), верно? Обратите внимание: я написал функцию для проверки содержимого файла, потому что файл журнала еще будет неоднократно проверяться позднее, и для этой цели удобно определить функцию:

```
def check_log_content(filename):
    with open(filename) as file:
        return file.read()

log_records = check_log_content("taskier.log")
print(log_records)

# Вывод: removed the task Laundry from the database
```

> **НАПОМИНАНИЕ** Используйте инструкцию `with` для открытия файла, чтобы файл закрывался автоматически.

Как видите, мы читаем весь файл, и его содержимое совпадает с ожидаемым: одна запись об удалении задачи из базы данных.

12.1.3. Добавление нескольких обработчиков

В разделе 12.1.2 вы узнали, как добавить файловый обработчик к регистратору, чтобы журнальные записи направлялись в файл. Регистратор может иметь несколько обработчиков, как будет показано в этом разделе.

Кроме файловых обработчиков, модуль `logging` также предоставляет потоковые обработчики, которые выводят записи на интерактивную консоль. В процессе разработки для сохранения журнальных записей используются файлы. Но мы сейчас добавим потоковый обработчик, чтобы просматривать записи в консоли для получения информации в реальном времени, как показано в листинге 12.4. В этом случае не придется открывать или читать журнал для получения записей.

Листинг 12.4. Использование потокового обработчика с регистратором

```
stream_handler = logging.StreamHandler()

logger.addHandler(stream_handler)

logger.warning("Just a random warning event.")
# Вывод: Just a random warning event.
```

Вызов конструктора `StreamHandler` создает потоковый обработчик, который затем добавляется к регистратору. Когда регистратору отправляется запись с предупреждением, сообщение выводится на консоль. Проверим, что регистратор также передает сообщение файловому обработчику, который был добавлен ранее:

```
log_records = check_log_content("taskier.log")

print(log_records)
# Выводимые строки:
removed the task Laundry from the database
Just a random warning event.
```

Как видите, в файл журнала записывается то же событие, что и в потоковый обработчик. Также обратите внимание, что файл журнала содержит запись, введенную ранее.

Регистратору назначается не только один файловый и один потоковый обработчик. Ничто не мешает назначить регистратору несколько файловых обработчиков. Допустим, вы хотите иметь два одинаковых файла журнала для целей резервного копирования; тогда создайте два файловых обработчика, по одному для каждого файла. Более того, для обработчиков можно назначать разные уровни (как объясняется в разделе 12.2.2) и точнее управлять обработчиками с учетом того, какие журнальные записи в них должны сохраняться.

В большинстве ситуаций достаточно использовать только потоковые и файловые обработчики. Однако существуют и другие виды обработчиков, которые могут пригодиться в конкретных ситуациях. Здесь они подробно не рассматриваются,

потому что используются не так часто, но все же полезно знать об их существовании (http://mng.bz/E0pD).

Как показано на рис. 12.1, к регистратору присоединяются разные виды обработчиков. Потоковые и файловые обработчики уже упоминались ранее. Стоит отметить обработчики SMTP, отправляющие журнальные записи по электронной почте; обработчики HTTP, отправляющие журнальные записи на веб-сервер в запросах `GET` или `POST`; а также обработчики `Queue`, которые могут направлять журнальные записи в очередь, например, поддерживаемую другим программным потоком.

Рис. 12.1. К регистратору присоединяются различные обработчики. После создания регистратора можно инстанцировать несколько обработчиков и присоединить их к регистратору. Каждый из них будет предназначен для соответствующей цели

12.1.4. Обсуждение

Файлы используются для регистрации важных событий приложения, чтобы разработчик мог найти необходимую информацию и исправить все возникающие проблемы. В фазе разработки бывает полезно добавить потоковый обработчик к регистратору, чтобы просматривать журнальные записи в консоли в реальном времени.

12.1.5. Задача

Джон недавно начал интегрировать в свой проект логирование. Он знает о возможности вызвать `logging.getLogger(__name__)` для получения регистратора, используемого модулем. Он выполняет код из листинга 12.2, который добавляет файловый обработчик к регистратору. Если он выполнит код несколько раз, регистратор будет содержать несколько файловых обработчиков, хотя все они ссылаются на один и тот же файл. При регистрации любых событий в файле позволяются повторяющиеся записи. Как ему обновить код в листинге 12.2, чтобы файловый обработчик добавлялся только один раз? Если к регистратору добавилось несколько обработчиков, как удалить лишние?

ПОДСКАЗКА 1 Регистратор содержит метод `hasHandlers`, при помощи которого проводится проверка, содержит ли регистратор обработчики. Если ни одного обработчика нет, можно добавить новый.

ПОДСКАЗКА 2 Обработчики регистратора сохраняются в объекте `List`. Содержимое списка можно удалить, чтобы обработчики были отсоединены от регистратора.

12.2. КАК ПРАВИЛЬНО ХРАНИТЬ ЖУРНАЛЬНЫЕ ЗАПИСИ

За долгое время в файле журнала накапливается множество записей, тысячи и даже миллионы в зависимости от размера приложения. Просмотр записей для поиска необходимой информации становится в этом случае серьезной проблемой. Для демонстрационных целей я использовал простые сообщения для журнальных записей в разделе 12.1. Однако в таск-менеджере следует ожидать появления в журнале записей следующего вида:

```
-- app is starting
-- created a new task Laundry
-- removed the task from the database
-- successfully changed the tags for the task
-- updated the task's status to completed
-- FAILED to change the task's status!!![1]
```

Как видите, с минимальным форматированием записей (два начальных дефиса) трудно обнаружить записи, содержащие потенциальную информацию о возникших проблемах. К счастью, записи можно классифицировать и форматировать для включения более подробной информации, что упростит процесс отладки. В этом разделе вы узнаете, как правильно сохранять записи с определением разных уровней ведения журнала, и я покажу, как следует применять форматирование к журнальным записям для улучшения удобочитаемости.

12.2.1. Классификация событий приложения по уровням

Не все проблемы в работе продукта имеют одинаковый уровень приоритета. Одни проблемы должны исправляться немедленно, другие могут подождать. Та же логика применима к системе логирования. Использование разных уровней логирования позволяет подчеркнуть важность или неотложность тех или иных проблем. В листинге 12.3 мы вызываем `logger.warning` для сохранения в журнале записи на уровне предупреждения. Как показано в этом разделе, существуют

[1] --старт приложения; --создана новая задача «Стирка»; --из базы данных удалена задача; --успешно изменен тег задачи; --статус задачи обновлен на «Завершено»; --НЕ УДАЛОСЬ изменить статус задачи!!! — *Примеч. пер.*

уровни более высокие, чем предупреждения, и вы узнаете, как файловые обработчики и средства логирования работают с системой уровней.

Модуль Python `logging` предоставляет доступ к пяти уровням: `DEBUG`, `INFO`, `WARNING`, `ERROR` и `CRITICAL` (отладка, информация, предупреждение, ошибка, критично), а также к базовому уровню (`NOTSET`), который представляется числовым значением 0 и обычно не используется. У каждого уровня имеется числовое значение, и чем оно выше, тем серьезнее проблема. На рис. 12.2 изображены эти уровни с приведением общих рекомендаций относительно того, какие записи должны сохраняться на каждом уровне.

Рис. 12.2. Уровни логирования для различных применений. Определены пять уровней: DEBUG, INFO, WARNING, ERROR и CRITICAL (по возрастанию критичности)

Эти пять уровней определяются целочисленными константами в модуле `logging`, им соответствуют числовые значения от 10 до 50 с шагом 10. Как показано на рис. 12.1, эти уровни предназначены для разных целей, и вам стоит соблюдать рекомендации по их применению. Однако я еще ничего не сказал о том, как использовать эти уровни.

Первый вариант использования — назначение уровня для регистратора. Кроме атрибута с файловыми обработчиками, у регистратора также имеется важный атрибут с именем `level`. Когда вы задаете для регистратора конкретный уровень (допустим, `INFO`), этот регистратор будет сохранять все записи с уровнем `INFO` или более серьезным (то есть `WARNING`, `ERROR` и `CRITICAL`). Посмотрим, как это работает:

```
logger = logging.getLogger(__name__)
logger.setLevel(logging.WARNING)

print(logger.level, logging._levelToName[logger.level])
# Вывод: 30 WARNING
```

Получает имя, соответствующее уровню

В этом фрагменте кода назначается регистратор с уровнем WARNING, и при проверке уровня регистратора из вывода видно, что это действительно WARNING. Когда регистратору назначается уровень WARNING, регистратор должен сохранять только предупреждения, сообщения об ошибках и критические сообщения. Этот эффект продемонстрирован в листинге 12.5.

Листинг 12.5. Сохранение записей разных уровней

```
def logging_messages_all_levels():
    logger.critical("--Critical message")
    logger.error("--Error message")
    logger.warning("--Warning message")
    logger.info("--Info message")
    logger.debug("--Debug message")

logging_messages_all_levels()
log_records = check_log_content("taskier.log")
print(log_records)

# Выводимые строки:
removed the task Laundry from the database
Just a random warning event.
--Critical message
--Error message
--Warning message
```

В листинге 12.5 отправляются пять сообщений, соответствующих пяти разным уровням. Из вывода видно, что сообщения INFO и DEBUG не сохраняются в файле журнала, потому что обработчику назначен уровень WARNING.

Вероятно, вы заметили, что мы используем `logger.critical` для отправки критических сообщений, `logger.error` для отправки сообщений об ошибках, и т. д. Важно знать эти методы, так как журнальные записи могут создаваться на разных уровнях. Выбор уровня напрямую определяет, как регистратор будет сохранять записи. Файловым обработчикам также назначаются уровни, как показано в следующем разделе.

12.2.2. Назначение уровня для обработчика

Еще одно возможное применение уровней — назначение уровня для обработчика. Регистратор может иметь несколько обработчиков с разными назначенными уровнями, так что они могут сохранять записи в соответствии со своим уровнем.

Для примера будут использоваться файловые обработчики. Допустим, таск-менеджер использует два файла журнала: в одном сохраняются записи уровня WARNING и выше, а в другом — только записи CRITICAL. Возможная реализация приведена в листинге 12.6.

Листинг 12.6. Назначение уровней для отдельных файловых обработчиков

```
logger.setLevel(logging.DEBUG)  ◄── Назначает для регистратора уровень DEBUG
handler_warning = logging.FileHandler("taskier_warning.log")
handler_warning.setLevel(logging.WARNING)  ◄── Добавляет обработчик на уровне WARNING
logger.addHandler(handler_warning)
handler_critical = logging.FileHandler("taskier_critical.log")
handler_critical.setLevel(logging.CRITICAL) ◄── Добавляет обработчик на уровне CRITICAL
logger.addHandler(handler_critical)
logging_messages_all_levels()
warning_log_records = check_log_content("taskier_warning.log")
print(warning_log_records)
# output the following lines:
--Critical message
--Error message
--Warning message
critical_log_records = check_log_content("taskier_critical.log")
print(critical_log_records)
# Вывод:
--Critical message
```

Как показано в листинге 12.6, сначала регистратору назначается уровень `DEBUG`, с которым регистратор перехватывает любые сообщения с уровнем `DEBUG` и выше. Чтобы показать, как настраиваются уровни на уровне обработчика, я добавил к регистратору два файловых обработчика: один с уровнем `WARNING`, другой с уровнем `CRITICAL`.

После регистрации нескольких сообщений на всех уровнях мы видим, что каждый файл сохраняет записи предполагаемых уровней. Файл taskier_critical.log содержит только одну запись `CRITICAL`, а файл taskier_warning.log содержит сообщения `WARNING`, `ERROR` и `CRITICAL`.

12.2.3. Форматы для обработчика

В предыдущем разделе вы узнали, как инициализировать регистратор и настроить его, назначая файловый обработчик и выбирая нужный уровень сохраняемых записей. Другой важный параметр конфигурации — форматирование записей в журнале. Без наглядного форматирования будет трудно найти информацию о проблемах. Форматирование записей должно выделять ключевую информацию каждой записи, например время события и уровень сообщения.

Мы могли бы продолжить настройку файлового обработчика для форматирования. Но при этом для получения записей пришлось бы читать файл журнала, что не слишком удобно в учебных целях. По этой причине мы пойдем по другому пути и воспользуемся потоковым обработчиком. Потоковый обработчик выводит журнальные записи в интерактивную консоль, упрощая просмотр результатов, как показано в листинге 12.7.

Листинг 12.7. Форматирование журнальных записей для потокового обработчика

```
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

logger.handlers = []

formatter = logging.Formatter("%(asctime)s [%(levelname)s] –
➥ %(name)s - %(message)s")

stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.DEBUG)
stream_handler.setFormatter(formatter)
logger.addHandler(stream_handler)

def log_some_records():
    logger.info("App is starting")
    logger.error("Failed to save the task to the db")
    logger.info("Created a task by the user")
    logger.critical("Can't update the status of the task")

log_some_records()

# Выводимые строки:
2022-05-18 10:45:00,900 [INFO] - __main__ - App is starting
2022-05-18 10:45:00,907 [ERROR] - __main__ - Failed to save the
➥ task to the db
2022-05-18 10:45:00,912 [INFO] - __main__ - Created a task by the user
2022-05-18 10:45:00,917 [CRITICAL] - __main__ - Can't update the
➥ status of the task
```

Получает регистратор и назначает уровень

Удаляет ранее назначенные обработчики

Создает форматировщик

Назначает форматер для обработчика

Как показано в листинге 12.7, модуль `logging` содержит класс `Formatter`, при помощи которого создается экземпляр для форматирования. Обратите внимание: вместо f-строк (см. раздел 2.1) форматер использует стиль `%`. По существу, форматер должен включать время создания события, уровень регистрации и сообщение. Также будет полезно включить имя модуля — в нашем случае `__main__`, потому что код будет выполняться в интерактивной консоли.

Как видно из выводимых результатов, отформатированное содержимое журнала читается намного лучше. Включение уровня позволяет сосредоточиться на нужных записях (уровня `ERROR` или `CRITICAL`). Кроме того, события снабжаются временны́ми метками, по которым легче связать происходящее с событиями за пределами приложения. Например, если в полночь регистрируется множество ошибок, не связано ли это с тем, что на сервере в это время происходит техническое обслуживание?

УДОБОЧИТАЕМОСТЬ Всегда форматируйте записи в журнале так, чтобы упростить поиск возможных проблем.

12.2.4. Обсуждение

К настоящему моменту вы уже достаточно хорошо понимаете, как работает логирование в Python. На рис. 12.3 изображена общая схема механизма логирования.

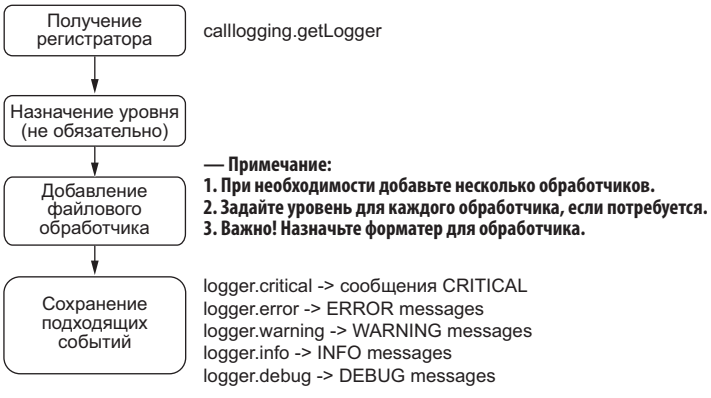


Рис. 12.3. Процесс логирования. Все начинается с получения регистратора вызовом getLogger. Затем (при желании) можно задать уровень сохраняемых сообщений. Чтобы сохранять записи в файле, к регистратору необходимо добавить файловый обработчик. Сохранение сообщения на конкретном уровне осуществляется вызовом соответствующего метода

Следует четко понимать, что представляет собой каждый из пяти уровней, и использовать его так, как и ожидается. Например, если какая-то функциональность необходима для нормальной работы приложения, возникающие с ней проблемы должны помечаться уровнем `CRITICAL`. Обработчик сохраняет сообщения только на уровнях равных или выше заданного, поэтому если вы хотите создать более подробные записи в журнале, важно установить для регистратора уровень `INFO` или `DEBUG`, чтобы сохранять больше записей.

12.2.5. Задача

Джон осваивает средства логирования событий и интегрирует их в свой проект. Он узнал, что уровни сообщений могут назначаться как для регистратора, так и для обработчика. Допустим, для регистратора установлен уровень `WARNING`, а для обработчика — уровень `DEBUG`. Что произойдет при вызове `logger.info("It's an info message.")`? Сохранит ли обработчик эту запись?

> **ПОДСКАЗКА** Сообщение проверяется на уровне регистратора до того, как регистратор передаст его обработчику.

12.3. КАК ОБРАБАТЫВАТЬ ИСКЛЮЧЕНИЯ

Когда в разделе обсуждалось преобразование строк для получения представляемых ими данных, вы узнали, что некоторые строки представляют числа (например, `"1"` и `"2"`) и для получения этих целых значений можно вызвать конструктор `int`. Допустим, в таск-менеджере имеется функция, которая обрабатывает строковые данные: это строки текстового файла, хранящего задачи. Для простоты будем считать, что набор атрибутов задачи ограничивается названием и степенью срочности.

```
from collections import namedtuple
Task = namedtuple("Task", ["title", "urgency"]) ◄── Создает класс именованного кортежа
task_text0 = "Laundry,3"

def process_task_string0(text):
    title, urgency_str = text.split(",") ◄── Распаковывает созданный объект list
    urgency = int(urgency_str)
    task = Task(title, urgency)
    return task

processed_task0 = process_task_string0(task_text0)

assert processed_task0 == Task(title='Laundry', urgency=3)
```

В этом фрагменте определяется функция `process_task_string0` для обработки текстовых данных и создания экземпляра класса `Task`. Вроде бы все нормально. Но что произойдет, если текст будет поврежден (например, `Laundry,3#`)? Давайте попробуем:

```
task_text1 = "Laudry,3#"
processed_task1 = process_task_string0(task_text1)
# ERROR: ValueError: invalid literal for int() with base 10: '3#'
```

`3#` не удается преобразовать в действительное целое число вызовом `int("3#")`, что приводит к исключению `ValueError`.

В большинстве случаев у нас нет оснований полагать, что все пойдет именно так, как мы ожидаем, особенно при использовании блоков кода, для работы которых требуются входные данные определенного вида. Например, конструктор `int` должен получать целое число или строку, представляющую целое число. В таком случае следует организовать обработку потенциальных исключений `ValueError` на фазе разработки, чтобы ошибка не останавливала работу нашего приложения на фазе выполнения. В этом разделе рассматриваются ключевые аспекты обработки исключений в Python.

12.3.1. Обработка исключений в блоке try…except…

Когда в приложении происходят такие исключения, как `ValueError`, оно аварийно завершается (если только вы не обеспечите обработку исключений, как

объясняется в этом разделе). Такое явление — аварийное завершение работы программы — называется *фатальным сбоем*. Фатальные сбои в приложениях объясняются разными причинами, часть из которых не зависит от самой программы (например, на компьютере заканчивается свободная память). Если вы ожидаете, что выполнение блока кода способно привести к конкретным исключениям, следует предусмотреть эту возможность и правильно обработать исключения, предотвратив возможные сбои приложения. В этом разделе описаны основные конструкции обработки исключений.

Исключения (exceptions), или *ошибки* (errors), принадлежат к числу основных концепций многих языков программирования. Как правило, для обработки исключений в Python используются блоки `try…except…`. Во многих других языках используются блоки `try…catch…`. На рис. 12.4 изображена схема работы инструкции `try…except…`.

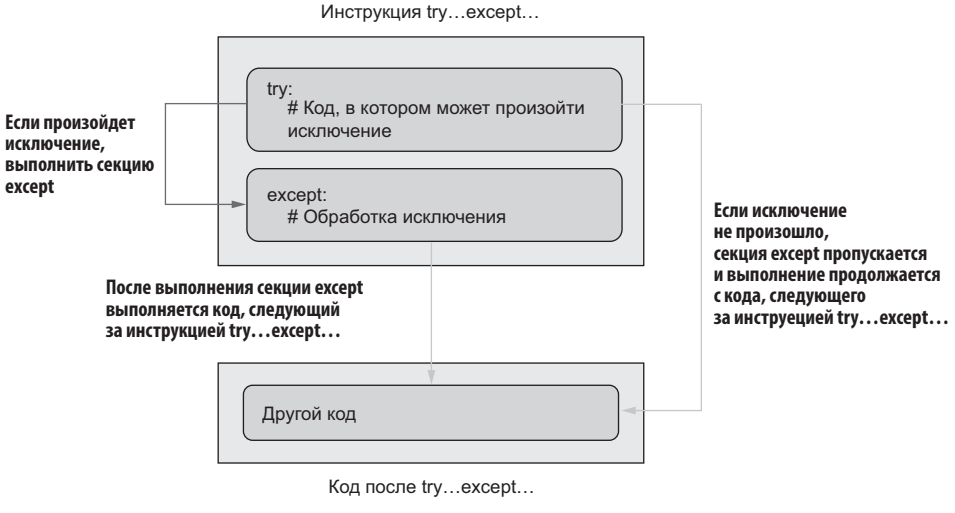


Рис. 12.4. Логика работы инструкции try…except…. Код, в котором могут возникнуть исключения, заключается в секцию try. При возникновении такого исключения выполняется секция except и управление передается коду за пределами инструкции. Если во время выполнения кода в секции try исключения не происходили, Python пропускает секцию except

Как показано на рис. 12.4, Python пытается выполнить код в секции `try`. Если все проходит нормально, то секция `except` пропускается и выполнение продолжается с кода за пределами инструкции `try...except`. При возникновении исключения выполняется секция `except`, а любой код в секции `try` после точки, в которой возникло исключение, пропускается. В следующем листинге приведен пример использования `try...except...`.

Листинг 12.8. Использование try…except… в функции

```
def process_task_string1(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
    except:
        print("Couldn't cast the number")
        return None
    task = Task(title, urgency)
    return task
```

ЗАБЕГАЯ ВПЕРЕД Не используйте `except` без указания исключений (раздел 12.3.2).

В листинге 12.8 функция `process_task_string1` включает инструкцию `try`... `except`.... В секции `try` размещается код, который способен породить исключение — в данном случае преобразование `urgency_str` в целое число. Важное замечание: не загромождайте секцию `try` большим объемом кода, потому что вам будет сложно определить, какой код привел к исключению.

УДОБОЧИТАЕМОСТЬ В секции `try` должен размещаться только код, который может вызвать исключение.

Для простоты и в учебных целях в секцию `except` был включен только вызов функции `print`. Важно понимать, что секция `except` выполняется только при перехвате исключения. За этим эффектом проследим в следующем фрагменте кода:

```
processed_task1 = process_task_string1(task_text1)
# Вывод: Couldn't cast the number

assert processed_task1 is None
```

Из секции `except` возвращается значение `None`; в этом можно убедиться, сравнив `processed_task1` с `None`. Если при выполнении секции `try` не выдаются никакие исключения, то секция `except` пропускается, а выполнение продолжается с точки, следующей после `try...except...`:

```
processed_task0 = process_task_string1(task_text0)

assert processed_task0 == Task(title='Laundry', urgency=3)
```

ВОПРОС Как вы думаете, будет ли сравнение работать с объектами пользовательских классов (вместо класса именованного кортежа — `Task` в этом примере)?

Как видите, когда `task_text0` содержит правильные данные для конструирования экземпляра класса `Task`, все работает так же, как в функции `process_task_string0`, словно инструкция `try...except...` вообще не существует в `process_task_string1`!

12.3.2. Указание исключений в секции except

В листинге 12.8 в заголовке секции `except` нет ничего, кроме ключевого слова. Однако я не рекомендую так делать. Секция `except` позволяет указать, какое исключение в ней обрабатывается. Как показано в разделе ниже, перехватываемое исключение должно быть указано явно.

Указывать исключение необходимо, в противном случае секция `except` будет перехватывать все исключения, даже те, что вы не ожидаете. Допустим, у вас имеется незавершенная задача, данные которой должны обновляться после преобразования степени срочности:

```
def process_task_string2(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
        pending_task.urgency = urgency
    except:
        print("Couldn't cast the number")
        return None
    task = Task(title, urgency)
    return task
```

> **ПРИМЕЧАНИЕ** Как правило, секция `try` должна содержать минимальный объем кода. Я включил лишнюю строку кода, приводящую к исключению, только чтобы продемонстрировать возможность обработки нескольких исключений.

В предыдущей версии `try` присутствует дополнительная строка `pending_task.urgency = urgency`. Возможно, вы поняли, что этот код приводит к исключению `NameError`, потому что переменная с таким именем нигде не определяется и она не доступна ни в каком пространстве имен. Следующий фрагмент демонстрирует этот эффект:

```
pending_task.urgency = 3
# ERROR: NameError: name 'pending_task' is not defined
```

При вызове `process_task_string2` может возникнуть как исключение `ValueError`, так и `NameError`. Минимальная секция `except` обработает оба исключения, не различая их:

```
process_task_string2("Laundry,3")
# Вывод: Couldn't cast the number
```

Можно было бы ожидать, что обработка `task_text0` пройдет без проблем и мы получим преобразованное значение 3. Однако выведенное сообщение создает впечатление, что проблема связана с неудачной попыткой преобразования.

Чтобы избежать неоднозначности, никогда не используйте минимальную секцию `except`, всегда явно указывайте исключение. В данном случае мы уже знаем

о возможности исключения `ValueError`; оно указывается после ключевого слова `except`. Эта секция будет выполнена, если исключение `ValueError` произойдет при выполнении секции `try`, как показано в листинге 12.9.

Листинг 12.9. Секция except с указанием исключения

```
def process_task_string3(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
        pending_task.urgency = urgency
    except ValueError:
        print("Couldn't cast the number")
        return None
    task = Task(title, urgency)
    return task
```

С обновленной функцией код выводит сообщение только в том случае, если будет перехвачено исключение `ValueError`:

```
process_task_string3("Laundry,3#")
# Вывод: Couldn't cast the number
```

Так как конструктор `int` не может преобразовать `"3#"` в целое число, исключение `ValueError` обрабатывается, как и ожидалось. Обратите внимание: при вызове функции со строкой, которая должна произвести правильный экземпляр `Task`, появится сообщение об исключении `NameError`, потому что мы не написали код для его обработки:

```
process_task_string3("Laudry,3")
# ERROR: NameError: name 'pending_task' is not defined
```

12.3.3. Обработка нескольких исключений

Мы знаем, что код выполняется линейно, и после операции преобразования `int(urgency_str)` выполнение продолжается командой `pending_task.urgency = urgency`, которая должна породить исключение `NameError`. В текущей версии это исключение не обрабатывается. Однако инструкция `try…except…` может обрабатывать сразу несколько исключений.

Существуют два способа обработки множественных исключений. Если исключения не связаны между собой, используйте несколько секций `except`; каждая секция обрабатывает отдельную разновидность исключений, как показано в листинге 12.10.

Листинг 12.10. Использование нескольких секций except

```
def process_task_string4(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
        pending_task.urgency = urgency
```

```
    except ValueError:
        print("Couldn't cast the number")
        return None
    except NameError:
        print("You're referencing an undefined name")
        return None
    task = Task(title, urgency)
    return task
```

Как видно из листинга 12.10, в функцию добавляется лишняя секция `except` для обработки потенциального исключения `NameError`.

ПРИМЕЧАНИЕ Наш код включает кажущиеся глупыми ошибки в целях демонстрации. Некоторые ошибки относятся к качеству самого кода, и для их исправления следует вносить изменения в код, а не обрабатывать исключения.

После обновления убеждаемся в том, что исключение действительно обрабатывается:

```
process_task_string4("Laundry,3")
# Вывод: You're referencing an undefined name
```

СОПРОВОЖДАЕМОСТЬ Используйте отдельные секции `except` для исключений, не связанных друг с другом. Если исключения семантически связаны, сгруппируйте их в одну секцию `except`. Впрочем, если хотите, обрабатывайте их по отдельности.

Кроме использования нескольких секций `except`, можно указать несколько исключений в одной секции `except`. Пример приведен в листинге 12.11.

Листинг 12.11. Несколько исключений в одной секции except

```
def process_task_string5(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
        pending_task.urgency = urgency
    except (ValueError, NameError):
        print("Couldn't process the task string")
        return None
    task = Task(title, urgency)
    return task
```

В этом примере оба исключения указываются в объекте `tuple` в одной секции `except`. В таком случае при перехвате любого из исключений будет выполнена одна секция `except`:

```
process_task_string5("Laundry,3")  <-- Ожидается исключение NameError
# Вывод: Couldn't process the task string
process_task_string5("Laundry,3#")  <-- Ожидается исключение ValueError
# output: Couldn't process the task string
```

Мы опробовали две строки: `"Laundry,3"` выдает исключение `NameError`, а `"Laundry,3#"` выдает исключение `ValueError`. Обратите внимание: при перехвате исключения управление передается в секцию `except`. Во втором случае, когда `int(urgency_str)` выдает исключение `ValueError`, мы не ожидаем, что также возникнет исключение `NameError`.

12.3.4. Вывод расширенной информации об исключении

Секция `except` обрабатывает заданное исключение, если оно возникает в программе. В примерах кода, использовавшихся ранее, выводились сообщения с кратким описанием исключений. Однако в этих сообщениях не хватало подробной информации, которая пригодилась бы пользователю.

Чтобы получить больше информации о перехваченном исключении, следует присвоить исключение переменной при помощи синтаксической конструкции `except SpecificException as имя_переменной`. В листинге 12.12 приведена обновленная версия функции, в которой используется данная возможность.

Листинг 12.12. Создание переменной, представляющей исключение

```
def process_task_string6(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
    except ValueError as ex:
        print(f"Couldn't cast the number. Description: {ex}")
        return None
    task = Task(title, urgency)
    return task
```

Как показывает листинг 12.12, перехваченное исключение `ValueError` присваивается переменной `ex`, чтобы эту переменную можно было использовать в коде секции. Для простоты я ограничусь выводом `ValueError`:

```
process_task_string6("Laundry,3#")
# Выводимая строка:
Couldn't cast the number. Description: invalid literal
➥ for int() with base 10: '3#'
```

Из сообщения видно, что попытка преобразования завершается неудачей, потому что `"3#"` не преобразуется в целое число. Следует заметить, что функция `print` для вывода подробного описания исключения показана только для демонстрации в учебных целях. В приложении, ориентированном на конечного пользователя (как в таск-менеджере), можно выдать предупреждение (`WARNING`), чтобы оповестить пользователей об ошибке и дать им возможность исправить ее.

12.3.5. Обсуждение

Обработка исключений играет ключевую роль в том, какое впечатление произведет ваше приложение на пользователя. Последствия исключений трудно

переоценить; если не обработать их, они приведут к фатальному сбою приложения. Следовательно, в фазе разработки следует осторожно относиться к коду, в котором что-то может пойти не так. Не беспокойтесь об инструкциях `try...except...` в коде. На первый взгляд они удлиняют код, но зато приложение становится более надежным, оно может продолжить работу даже при возникновении исключений.

12.3.6. Задача

Боб — опытный программист, применяющий в своем коде лучшие практики. Он понимает, что при написании инструкции `try...except...` следует явно указывать, какие исключения в ней обрабатываются. Существует много видов исключений. Как ему узнать, какие исключения могут возникнуть в конкретной ситуации в фазе разработки? Например, как ему узнать на примере листинга 12.9, что нужно обрабатывать возможное исключение `ValueError`?

> **ПОДСКАЗКА** Кроме поиска информации об исключениях в официальной документации Python, стоит выполнить потенциально проблемный код и посмотреть, какие исключения в нем возникнут. После этого их можно будет обработать соответствующим образом.

12.4. КАК ИСПОЛЬЗОВАТЬ СЕКЦИИ ELSE И FINALLY ПРИ ОБРАБОТКЕ ИСКЛЮЧЕНИЙ

Простейшая форма обработки исключений в Python — инструкция `try...except...`. Инструкция состоит из одной секции `try` и одной или нескольких секций `except`. Следующий фрагмент является частью листинга 12.12:

```
try:
    urgency = int(urgency_str)
except ValueError as ex:
    print(f"Couldn't cast the number. Description: {ex}")
    return None
task = Task(title, urgency)
```

Мы знаем, что код `task = Task(title, urgency)` выполняется после инструкции `try...except...`. Обратите внимание: в секцию `except` включена инструкция `return` (`return None`). Если не включить ее, то произойдет исключение `UnboundLocalError` из-за выполнения команды `task = Task(title, urgency)` без определения `urgency` в секции `except`. Но мы знаем, что команда `task = Task(title, urgency)` должна выполняться только в том случае, если код в секции `try` был выполнен без возникновения исключений. Нельзя ли более явно выразить, что код должен выполняться только при отсутствии исключений? Этот вопрос приводит к следующему разделу: включению секции `else` в инструкцию `try...except...`.

В разделе 12.4.2 рассматривается секция `finally` — еще один необязательный компонент полной инструкции `try...except...`.

12.4.1. Использование else для продолжения логики секции try

В разделе 12.3 я упоминал о том, что длина секции `try` должна быть минимальной и в эту секцию следует включать только код, в котором могут возникать исключения. Когда секция `try` завершает выполнение, Python выполняет код, следующий после инструкции `try...except...`. Однако код, следующий после инструкции, имеет смысл только в том случае, если при его выполнении в секции `try` не происходили исключения. Для реализации этой функциональности следует использовать секцию `else` в дополнение к `try` и `except`.

В инструкции `try...except...` ключевое слово `try` означает, что вы собираетесь выполнить код, в котором могут быть выданы исключения, а ключевое слово `except` — что вы собираетесь обрабатывать перехваченные исключения. Как насчет `else`? Чтобы понять его смысл, необходимо осознать, что вся инструкция `try...except...else...` предназначена для обработки исключений. А если говорить более конкретно, одной из целей является перехват таких исключений. Таким образом, можно сказать, что если исключение перехвачено, мы обрабатываем его; в ином случае выполнение продолжается. Секция `else` описывает часть «в ином случае». Пример приведен в листинге 12.13.

Листинг 12.13. Включение секции else в инструкцию try…except

```
def process_task_string7(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
    except ValueError as ex:
        print(f"Couldn't cast the number. Description: {ex}")
        return None  <-- Эта инструкция return None не обязательна, ее можно опустить
    else:
        task = Task(title, urgency)
        return task
```

Как показано в листинге 12.13, секция `else` следует после секции `except`. В секции `else` мы создаем экземпляр класса `Task` (определяемого в начале раздела 12.3) со значениями `title` и `urgency`. Ожидается, что при отсутствии исключения `ValueError` будет получен экземпляр:

```
processed_task7 = process_task_string7("Laundry,3")
assert processed_task7 == Task("Laundry", 3)
```

Как видно из приведенного фрагмента, мы получаем экземпляр класса `Task`, из чего следует, что код в секции `else` был успешно выполнен. Что произойдет при выдаче исключения `ValueError`? Взгляните на результат:

```
processed_task = process_task_string7("Laundry,3#")
# Вывод:
Couldn't cast the number. Description: invalid literal for
➥ int() with base 10: '3#'

print(processed_task)
# Вывод: None
```

Прежде всего обратите внимание на то, что секция `except` выполняется из-за перехваченного исключения `ValueError`. Кроме того, возвращаемое значение вызова `process_task_string7` равно `None`, из чего следует, что код в секции `else` не будет выполнен, когда секция `except` выполняется и возвращает `None`.

12.4.2. Завершение обработки исключений в секции finally

Как было показано в разделе 12.4.1, выполняется только одна из секций `except` и `else`. Если в секции `try` происходят исключения, выполняется секция `except` (обработка исключений); если секция `try` выполняется без исключений, выполняется секция `else`. Однако иногда есть код, который должен выполняться независимо от статуса исключений. Например, в функции, которая обрабатывает строку с данными задачи, после завершения обработки нужно сообщить пользователю, успешно ли завершилась обработка. Именно для таких задач подходит секция `finally`, как будет показано в этом разделе. На рис. 12.5 наглядно изображена схема работы всех четырех секций при обработке исключений.

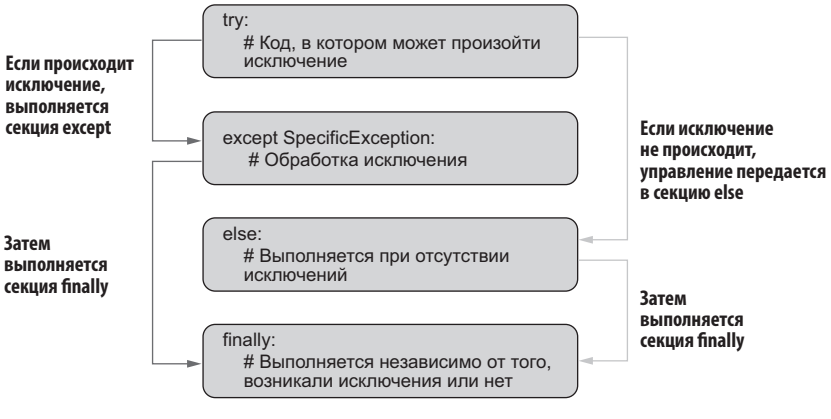


Рис. 12.5. Четыре секции инструкции `try...except...else...finally`.... Секция `try` содержит код, который может выдать исключение. Секция `except` включает код, обрабатывающий возможное исключение. Секция `else` выполняется только в том случае, если выполнение `try` прошло без исключений. Секция `finally` выполняется после секции `except` или после секции `else`

Секция `finally` должна размещаться в конце инструкции `try…except` (рис. 12.5). Если в инструкции присутствует секция `else`, секция `finally` должна следовать за ней; в противном случае она следует за секцией `except`. Код в секции `finally` выполняется независимо от того, произошло ли исключение. В листинге 12.14 работа `finally` продемонстрирована на примере обработки строки с данными задачами.

Листинг 12.14. Секция finally в инструкции try…except…

```
def process_task_string8(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
    except ValueError as ex:
        print(f"Couldn't cast the number. Description: {ex}")
        return None
    else:
        task = Task(title, urgency)
        return task
    finally:
        print(f"Done processing text: {text}")
```

В листинге 12.14 в инструкцию `try…except…` добавляется секция `finally`. Для простоты мы ограничиваемся выводом сообщения о завершении обработки. Секция `finally` должна выполняться независимо от того, выдавалось исключение `ValueError` или нет:

```
task_no_exception = process_task_string8("Laundry,3")
# Вывод:
Done processing text: Laundry,3
task_exception = process_task_string8("Laundry,3#")
# Выводимые строки:
Couldn't cast the number. Description: invalid literal for int()
➥ with base 10: '3#'
Done processing text: Laundry,3#
```

В обоих вызовах функции `process_task_string8` мы видим, что секция `finally` выполняется и выводит сообщение в f-строке. Кто-то спросит: зачем вообще использовать `finally`? Если она выполняется независимо от статуса исключения, почему не разместить ее за пределами `try…except…`? Потому что мы знаем, что код обычно выполняется линейно, и размещение его за пределами инструкции гарантирует, что он будет выполнен после секции `except` или `else`.

Я говорю «обычно», потому что к секции `finally` применяется нетипичное правило. Если выполнение секции `try` достигает инструкции `break`, `continue` или `return`, секция `finally` будет выполнена до выполнения `break`, `continue` или `return`. Это правило необходимо для того, чтобы гарантировать выполнение кода в секции `finally`, потому что в типичной ситуации эти инструкции завершают текущее выполнение и пропускают оставшийся код. Следующий пример демонстрирует этот эффект:

```
def process_task_string9(text):
    title, urgency_str = text.split(",")
    try:
        urgency = int(urgency_str)
        task = Task(title, urgency)
        return task
    except ValueError as ex:
        print(f"Couldn't cast the number. Description: {ex}")
        return None
    finally:
        print(f"Done processing text: {text}")

task = process_task_string9("Laundry,3")
# Вывод: Done processing text: Laundry,3

assert task == Task("Laundry", 3)
```

Как показывает этот пример, в секцию `try` включается инструкция `return`. В отличие от других сценариев, `return` немедленно завершает выполнение функции. Здесь, как мы видим, функция `print` вызывается в секции `finally`, что подтверждает предыдущее утверждение о выполнении секции `finally` вне зависимости от статуса исключения и даже от присутствия инструкции `return` в секции `try` или `except`. Так как секция `finally` выполняется независимо от того, произошло исключение или нет, она часто используется при работе с общими ресурсами, например файлами и сетевыми подключениями. Эти ресурсы должны освобождаться (в секции `finally`) вне зависимости от того, была ли запрашиваемая операция выполнена успешно (секция `try`) или же произошло исключение (секция `except`).

12.4.3. Обсуждение

Из четырех секций конструкции обработки исключений всегда должны использоваться секции `try` и `except`, потому что они определяют основы обработки исключения. Секция `try` «пытается» (try) выполнить код, так как при выполнении могут возникнуть исключения, а секция `except` перехватывает и обрабатывает исключения. Хотя секции `else` и `finally` не обязательны, они также находят применение в разных ситуациях, о которых вам следует знать.

12.4.4. Задача

Вы знаете, что если при наличии секции `finally` секция `try` содержит инструкцию `return`, код из `finally` все равно будет выполнен перед выполнением `return` в `try`. Какое значение будет возвращено при вызове `process_task_challenge` в следующем примере?

```
def process_task_challenge(text):
    title, urgency_str = text.split(",")
    try:
```

```
        urgency = int(urgency_str)
        task = Task(title, urgency)
        return task
    except ValueError as ex:
        print(f"Couldn't cast the number. Description: {ex}")
        return None
    finally:
        print(f"Done processing text: {text}")
        return "finally"

processed = process_task_challenge("Laundry,3")
print(processed)
```

ПОДСКАЗКА Так как код в секции `finally` выполняется до инструкции `return` в секции `try`, функция немедленно завершается, как если бы сама секция `finally` включала `return`. Инструкция `return` в секции `try` пропускается.

12.5. КАК ВЫДАВАТЬ СОДЕРЖАТЕЛЬНЫЕ ИСКЛЮЧЕНИЯ С ПОЛЬЗОВАТЕЛЬСКИМИ КЛАССАМИ ИСКЛЮЧЕНИЙ

Когда мы учимся программировать на Python, мы допускаем всевозможные ошибки. Некоторые из них, например пропущенное двоеточие в инструкции `if…else…`, относятся к категории синтаксических ошибок.

Программист, который знает основные элементы синтаксиса, сталкивается с другими ошибками, которые в основном связаны с правильностью использования конкретных средств с точки зрения логики или семантики. К числу таких ошибок принадлежит исключение `ValueError`, часто встречавшееся в разделах 12.3 и 12.4. Или другой пример: при попытке деления числа на ноль происходит исключение `ZeroDivisionError`:

```
int("3#")
# ERROR: ValueError: invalid literal for int() with base 10: '3#'

1 / 0
# ERROR: ZeroDivisionError: division by zero
```

В обоих случаях в сообщении об ошибке указывается не только конкретное имя исключения, но и приводится описание ошибки, которое помогает определить, что было сделано неправильно. Когда вы создаете библиотеку или пакет, предназначенные для использования другими разработчиками, важно выводить соответствующую информацию для пользователей, чтобы они знали, как отлаживать код или обрабатывать исключение. В этом разделе вы научитесь выдавать содержательные исключения с пользовательскими классами исключений.

12.5.1. Выдача исключений с пользовательским сообщением

До сих пор мы видели исключения, которые Python выдает при обработке нашего кода. Однако я еще не объяснил, как выдавать исключения по своей инициативе. В этом разделе вы узнаете, как выдавать пользовательские (кастомные) исключения и как предоставлять собственные сообщения для таких исключений.

> **ОСНОВНЫЕ ПОНЯТИЯ** Когда вы «порождаете» исключение для обозначения неких проблем, это называется *выдачей* исключения (raise an exception).

Для выдачи исключений в Python используется ключевое слово `raise`. Выполнив следующий код в консоли, вы также увидите данные трассировки:

```
>>> raise ValueError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError

>>> raise ZeroDivisionError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError
```

Исключения выдаются в формате `raise КлассИсключения`. Упомянутые выше `ValueError` и `ZeroDivisionError` являются классами исключений. Строго говоря, при выдаче исключения выдается экземпляр класса исключения; таким образом, формат этого примера может рассматриваться как синтаксический сахар для `raise КлассИсключения()`, где `КлассИсключения()` создает экземпляр класса.

> **ОСНОВНЫЕ ПОНЯТИЯ** В программировании *синтаксическим сахаром* (syntactic sugar) называются простые средства, которые выполняют те же операции, что и более сложные аналоги.

При обработке исключения мы тоже имеем дело с экземпляром класса исключения. Следующий пример поясняет сказанное:

```
try:
    1 / 0
except ZeroDivisionError as ex:
    print(f"Type: {type(ex)}")
print(f"Is an instance of ZeroDivisionError?
➥ {isinstance(ex, ZeroDivisionError)}")

# Выводимые строки:
Type: <class 'ZeroDivisionError'>
Is an instance of ZeroDivisionError? True
```

Как показано в примере, мы знаем, что операция `1/0` приводит к выдаче исключения `ZeroDivisionError` и обрабатывается в секции `except`. Из выводимого сообщения видно, что выданное исключение действительно является экземпляром класса `ZeroDivisionError`.

Команда `raise ValueError` особой пользы не принесет. Вспомните, что при вызове `int("3#")` в сообщении об ошибке была явно указана причина исключения: `ValueError: invalid literal for int() with base 10: ' 3#'`. Чтобы передать пользовательское сообщение с исключением, используйте формат `raise КлассИсключения("сообщение")`. Несколько примеров:

```
raise ValueError("Пожалуйста, используйте корректный параметр.")
# ERROR: ValueError: Пожалуйста, используйте корректный параметр.
code_used = "3#"

raise ValueError(f"Вы использовали неверный параметр: {code_used!r}")
# ERROR: ValueError: Вы использовали неверный параметр: '3#'
```

Использует преобразование repr !r для создания строки, заключенной в кавычки

Когда вы передаете такое сообщение с содержательной информацией конструктору класса исключения, текст сообщения выводится вместе с выдаваемым исключением. Следите за тем, чтобы сообщение было компактным; не перегружайте пользователя пространными описаниями, которые только приведут его в замешательство.

УДОБОЧИТАЕМОСТЬ Будьте лаконичны при определении пользовательских сообщений для классов исключений.

12.5.2. Встроенные классы исключений предпочтительны

Когда в предшествующих главах рассматривались модели данных, вы узнали о встроенных типах данных, например `str` (глава 2) или `list` и `tuple` (глава 3), прежде чем перейти к пользовательским классам (главы 8 и 9). Такой порядок был выбран из-за того, что встроенные типы данных являются базовой формой представления данных и они понятны всем программистам Python. Эта философия распространяется и на исключения. Если требуется выдать исключение, лучше использовать встроенные типы исключений.

Мы знаем, что для получения исключений необходимо создать экземпляр класса исключений. Таким образом, для применения встроенных классов исключений надо знать часто используемые классы. Не огорчайтесь, если вы пока их не знаете. Каждый, кто учится программировать, совершает многочисленные ошибки, приводящие к исключениям. Постепенно вы разберетесь, какие исключения связаны с соответствующими ошибками в вашем коде. На рис. 12.6 приведена сводка часто используемых исключений.

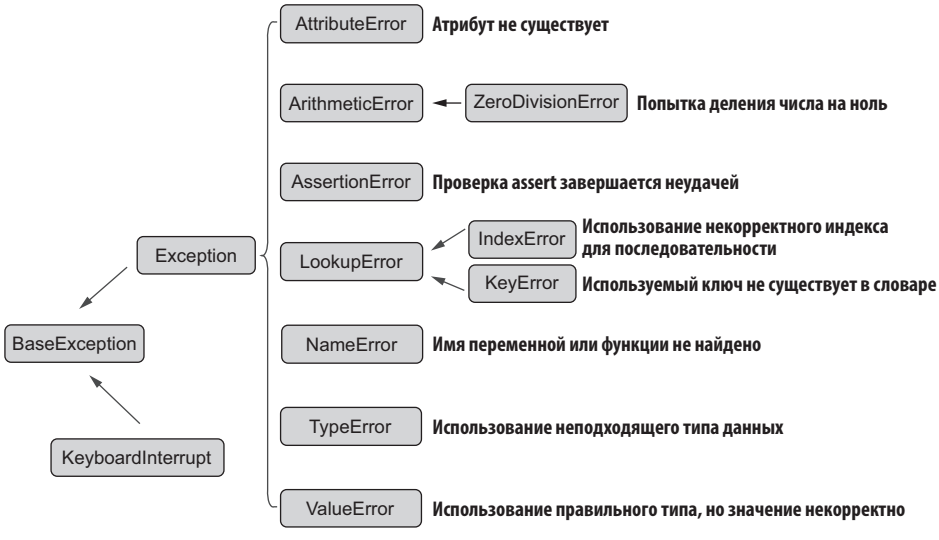


Рис. 12.6. Часто используемые встроенные классы исключений. Класс BaseException является надклассом для всех остальных классов исключений. Многие классы исключений, которыми мы пользуемся, — подклассы Exception

Класс `BaseException` является базовым для всех встроенных исключений, включая такие системные исключения, как `KeyboardInterrupt` и `SystemExit`. При определении собственных пользовательских классов исключений в общем случае не рекомендуется наследовать от этого класса; лучше использовать класс `Exception` (раздел 12.5.3) для предотвращения перехвата системных исключений. Популярные классы исключений, встречавшиеся вам ранее, такие как `ValueError` и `NameError`, являются прямыми или косвенными подклассами класса `Exception`.

Хотя определить пользовательский класс исключения несложно, при рассмотрении возможности выдачи исключения следует начать со встроенных классов, потому что они лучше известны рядовым разработчикам. Рассмотрим простой пример:

```
class Task:
    def __init__(self, title):
        self.title = title
```

В этом фрагменте кода мы определяем класс `Task` с атрибутом `title`, который представляет собой строку. Чтобы гарантировать выполнение этого требования, можно включить в код проверку типа и выдавать исключение, если переданный аргумент не является объектом `str`, как показано в листинге 12.15.

Листинг 12.15. Создание класса, который выдает исключение в конструкторе

```
class Task:
    def __init__(self, title):
        if isinstance(title, str):
            self.title = title
        else:
            raise TypeError("Please instantiate the Task
            ➥ using string as its title")

task = Task(100)
# ERROR: TypeError: Please instantiate the Task using string as its title
```

Встроенное исключение `TypeError` в листинге 12.15 поможет пользователям понять, что они применили неправильный тип при передаче аргумента.

УДОБОЧИТАЕМОСТЬ Старайтесь применять встроенные классы исключений, так как они лучше знакомы пользователям.

12.5.3. Определение пользовательских классов исключений

При создании собственных пакетов Python принято определять пользовательские классы исключений, если встроенные классы не подходят для ваших целей. В этом разделе приводятся рекомендации по определению пользовательских классов исключений.

Как упоминалось в разделе 12.5.2, пользовательские классы исключений должны наследовать от класса `Exception`. При создании пользовательских пакетов лучше всего сначала определить базовый класс исключений для вашего пакета, а затем дополнительные классы исключений посредством наследования от этого базового класса. Определение базового исключения для вашего пакета позволяет пользователю обрабатывать все исключения пакета, если возникнет такая необходимость.

Создайте базовый класс исключения для вашего приложения, если вам нужно определять собственные классы исключений, которые должны наследовать от базового класса. Допустим, в таск-менеджере мы преобразуем приложение в пакет, который будет использоваться другими разработчиками для построения собственных приложений. Они могут пользоваться классом `Task` как моделью данных для построения других приложений, например другой библиотеки клиентских приложений. В этом пакете, назовем его `taskier`, определим базовый класс исключения с именем `TaskierError`:

```
class TaskierError(Exception):
    pass
```

В этом классе исключения, принадлежащем конкретному пакету, никакие подробности реализации не нужны. Можно просто использовать инструкцию `pass` для выполнения требований синтаксиса (тело класса не может быть пустым).

Для пакета `taskier` определяются более конкретные классы исключений. Так, мы можем позволить пользователям отправить CSV-файл со своего компьютера для получения данных из нескольких задач. В листинге 12.16 определяется исключение, которое требует, чтобы файл имел расширение `.csv`.

Листинг 12.16. Определение класса для пользовательского исключения

```
class FileExtensionError(TaskierError):
    def __init__(self, file_path):
        super().__init__()
        self.file_path = file_path

    def __str__(self):
        return f"The file ({self.file_path}) doesn't appear to be a
        ➥ CSV file."

# В другой части пакета
from pathlib import Path

def upload_file(file_path):
    path = Path(file_path)
    if path.suffix.lower() != ".csv":
        raise FileExtensionError(file_path)
    else:
        print(f"Processing the file at {file_path}")
```

В листинге 12.16 стоит обратить внимание на два важных обстоятельства:

- Пользовательский класс исключения может получать дополнительные аргументы для создания экземпляра. Здесь мы включаем аргумент `file_path` (помните, что определять сообщение для создания исключения не обязательно), потому что хотим показать читателям кода, что файл с заданным путем имеет неподходящий формат.
- Мы переопределяем метод `__str__`. Как упоминалось в разделе 8.4, этот метод вызывается при выводе экземпляра.

Этот класс исключения используется в другой части пакета. Как показано в приведенном коде, функция `upload_file` проверяет расширение файла (раздел 11.5) и выдает исключение при неподходящем расширении.

Другой разработчик, использующий наш пакет, может построить виджет для отправки файла. В его приложении будет присутствовать следующая функциональность:

```
def custom_upload_file(file_path):
    try:
        upload_file(file_path)
```

```
    except FileExtensionError as ex:
        print(ex)
    else:
        print("Custom upload file is done.")

custom_upload_file("tasks.csv")   #A Функция вызывается для CSV-файла
# Выводимые строки:
Processing the file at tasks.csv
Custom upload file is done.

custom_upload_file("tasks.docx")   #B Функция вызывается для docx-файла
# Вывод: The file at tasks.docx doesn't appear to be a CSV file.
```

В этом примере функция вызывается для двух разных типов файлов: CSV-файла и файла документа Microsoft Word. Как видите, при передаче неподходящего файла секция `except` перехватывает `FileExtensionError` и выводит сообщение, реализованное в классе `__str__`.

При необходимости следует определить в пакете дополнительные классы пользовательских исключений. Например, класс исключения с именем `FileFormatError` будет использоваться в том случае, если в файле отсутствуют необходимые данные. Другой пример: мы можем определить класс исключения с именем `InputArgumentError`, который будет использоваться при передаче недопустимых аргументов критически важным функциям. Оба класса должны наследовать от `TaskierError`. На рис. 12.7 изображена иерархия классов исключений в пользовательском пакете.

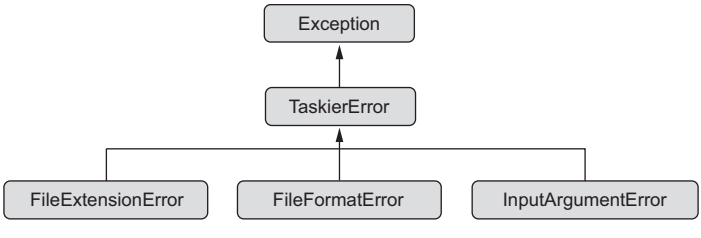


Рис. 12.7. Иерархия нестандартных классов исключений в пользовательском пакете. Класс исключения, специфичный для конкретного пакета, создается наследованием от класса `Exception`. На основе этого базового класса определяются различные классы специализированных исключений

12.5.4. Обсуждение

Несмотря на возможность определения пользовательских классов для выдачи содержательных исключений, по возможности старайтесь ограничиваться встроенными классами. Однако если вы создаете собственный пакет или библиотеку, лучше определить собственные пользовательские классы исключений, которые будут выдавать более конкретные сообщения об ошибках, чтобы пользователям

пакета (разработчикам) было проще отлаживать ошибки. Начать следует с определения базового класса исключения для конкретного пакета. Пользовательские классы исключений ведут себя как обычные классы, и при необходимости в них переопределяются такие специальные методы, как `__str__`.

12.5.5. Задача

В листинге 12.15 класс `Task` способен выдать исключение `TypeError` в своем конструкторе. Сможете ли вы написать код для обработки этого исключения при помощи инструкции `try…except…else…finally…`?

> **ПОДСКАЗКА** Вызовите конструктор в секции `try` и обработайте возможное исключение `TypeError`.

ИТОГИ

- Чтобы получить регистратор для вашего модуля, рекомендуется вызвать `getLogger`. Это гарантирует, что вы получите тот же регистратор, а не создадите несколько новых.
- Для долгосрочного хранения информации к регистратору обычно присоединяется файловый обработчик, чтобы журнальные записи сохранялись в файлах.
- В фазе разработки полезно выводить журнальные записи на консоль. Для этого к регистратору также добавляется потоковый обработчик.
- Чтобы было удобнее отслеживать важность записей, разделите их на уровни: `DEBUG`, `INFO`, `WARNING`, `ERROR` и `CRITICAL`.
- Регистратору и обработчикам можно назначить определенный уровень, чтобы они отслеживали записи только нужного уровня.
- Для получения удобочитаемой информации всегда следует форматировать журнальные записи и включать в них такие ключевые сведения, как временная метка, уровень критичности, название модуля и текст сообщения.
- Инструкция `try…except…` — это базовый метод обработки исключений в Python. Секция `try` должна включать только код, в котором могут возникать исключения. Следует явно указывать исключения, которые обрабатываются в секции `except`.
- Хотя в одной секции `except` можно сгруппировать несколько исключений, включив их в объект `tuple`, я рекомендую использовать несколько секций `except` вместо одной, если только исключения не являются тесно связанными.
- Секция `else` выполняется в том случае, если в секции `try` не возникают исключения. Секция `finally` используется для завершения обработки исклю-

чения; она выполняется вне зависимости от того, произошло ли исключение в секции `try`.

- Для выдачи исключений также можно применять встроенные классы исключений, для большей информативности снабдив их пользовательскими сообщениями.
- При определении пользовательских классов исключений помните, что наследовать необходимо от класса `Exception`, но не от `BaseException`.
- Если ваш пакет включает пользовательские классы исключений, рекомендуется определить для пакета базовый класс исключения. На основе этого класса определяются дополнительные подклассы пользовательских исключений.

13 Отладка и тестирование

В этой главе

- ✓ Чтение трассировки
- ✓ Отладка приложения в интерактивном режиме
- ✓ Тестирование функций
- ✓ Тестирование классов

Выполнение программного проекта с нуля до стадии полной работоспособности напоминает строительство дома. После того как вы заложили фундамент, установили каркас и стены, настелили крышу, поставили двери и окна, кажется, что дом в основном готов. Но когда вы переходите к внутренней отделке — полам, освещению, мебели, становится ясно, что работа еще далека от завершения.

Вы усердно проработали над своим приложением три месяца, и у вас появилось чувство, что проект на 90 % готов. Но прежде чем отправлять проект в эксплуатацию, необходимо убедиться в его надежности, проведя тщательную отладку и тестирование. Меня не удивит, если последние 10 % займут у вас еще три месяца — столько же времени, сколько понадобилось для первых 90 %. Фаза отладки и тестирования сравнима с внутренней отделкой дома — она настолько важна, что ваше приложение не сможет существовать без нее. И вам бы не хотелось получать жалобы от клиентов после выпуска. А значит, отладкой и тестированием

стоит заняться тогда, когда приложение все еще находится в ваших руках. В этой главе описаны важные операции, применяемые в завершающей фазе работы над приложением.

13.1. КАК ВЫЯВИТЬ ПРОБЛЕМЫ С ПОМОЩЬЮ ТРАССИРОВКИ

Когда в вашем коде происходят фатальные сбои из-за исключений, Python не только сообщает об исключении, но и предоставляет информацию о том, где оно произошло. Допустим, при определении класса `Task` при вызове метода было неверно указано его имя. Когда вы создаете экземпляр класса `Task` и вызываете метод экземпляра `update_urgency`, выдается исключение `AttributeError`. Попробуйте выполнить код из листинга 13.1 в консоли.

Листинг 13.1. Вывод трассировки при выполнении кода

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

   def _update_db(self):
        # update the record in the database
        print("update the database")

    def update_urgency(self, urgency):
        self.urgency = urgency
        self.update_db()  <-- Номер строки 10 без учета пустых строк

task = Task("Laundry", 3)
task.update_urgency(4)  <-- Строка кода, в которой произошло исключение
# output the following error:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in update_urgency
AttributeError: 'Task' object has no attribute 'update_db'
```

> **ПРИМЕЧАНИЕ** Когда вы направляете свой код на консоль, пустые строки удаляются, так что номер строки в трассировке отличается от номера строки в файле. Так как мы часто выполняем код и в консоли, и в файле, в этом разделе я приведу трассировки в обоих режимах.

Почти во всех примерах кода с исключениями я приводил только последнюю строку исключения. Здесь я привожу полное выводимое сообщение. Кроме строки исключения, в вывод включена такая информация, как имя метода и номер строки; эти данные помогают обнаружить источник ошибки. Информация о местонахождении ошибки называется *трассировкой*. Применение трассировки

для поиска причины ошибки — первый шаг отладки кода. В этом разделе вы научитесь читать трассировки и использовать их для выявления ошибок в коде.

13.1.1. Как генерируется трассировка

Трассировка представляет собой подробное описание того, как произошло исключение. В главе 12 вы узнали, как читать последнюю строку трассировки, в которой приводится тип и описание исключения. Сейчас мы немного отступим назад и разберемся, как генерируется трассировка; это необходимо для правильного чтения трассировок и сбора информации об исключениях.

Во время выполнения приложения постоянно происходят разные события: приложение создает экземпляры, обращается к их атрибутам и вызывает их методы. Когда что-то работает не так, как ожидалось, может возникнуть исключение, приводящее к прерыванию работы приложения. Хотя выполнение конкретной строки кода (например, `task.update_urgency(4)` в листинге 13.1) кому-то покажется непосредственной причиной завершения приложения, весьма вероятно, что истинная причина кроется вовсе не в этой строке; исключение может возникнуть из-за операции, выполненной в другой точке. Таким образом, чтобы узнать, как выдается исключение, необходимо понимать общий процесс выполнения кода.

В качестве примера возьмем код в листинге 13.1. На рис. 13.1 изображена простая диаграмма основных этапов его выполнения.

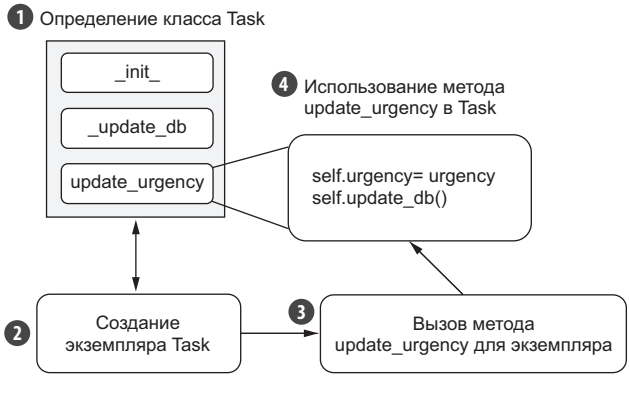


Рис. 13.1. Процесс выполнения кода из листинга 13.1. В первой фазе определяется класс `Task`. Во второй фазе создается экземпляр класса. В третьей фазе вызывается метод `update_urgency`. В четвертой фазе используется определение метода в классе

Код в листинге 13.1 состоит из четырех основных фаз:

- Определение класса `Task`.
- Создание экземпляра `Task`.

- Вызов метода `update_urgency`.
- Использование определения метода `update_urgency` в классе.

Как показано в листинге 13.1, вызов `task.update_urgency(4)` приводит к исключению, и это происходит не из-за того, что с вызовом метода что-то не так. Скорее что-то не так с определением класса во внутренней реализации. Как можно заметить из листинга 13.1, `update_urgency` ошибочно вызывает `update_db` вместо `_update_db`, как должно бы быть.

Эти четыре фазы представляют последовательность отдельных блоков выполнения программы, которые включают тысячи меньших операций. Если рассматривать происходящее в обобщенном виде, можно построить дерево операций (рис. 13.2). Каждый прямоугольник представляет отдельную операцию. Такие операции образуют *стек вызовов*, определяющий текущее состояние выполнения приложения.

ОСНОВНЫЕ ПОНЯТИЯ *Стек вызовов* (call stack) отслеживает последовательность выполнения программы от текущего вызова до других операций, необходимых для завершения выполнения. Эти последовательные операции образуют стек вызова.

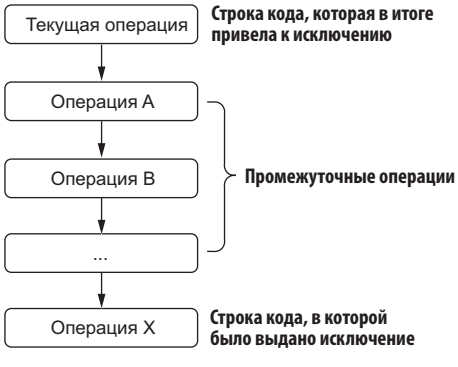


Рис. 13.2. Схематическое представление трассировки. Трассировка начинается со строки кода, которая в конечном итоге привела к исключению, и следует по всем задействованным операциям вплоть до строки кода, в которой непосредственно возникло исключение

Трассировка строится на базе стека вызовов. Она начинается со строки кода, которая в конечном итоге привела к исключению, и содержит операцию, которая была вызвана в этой строке. Если операция не выдала исключение, то трассировка переходит к следующей операции, пока не будет найден код, в котором было выдано исключение. Схематическое представление трассировки изображено на рис. 13.2.

13.1.2. Анализ трассировки при выполнении кода в консоли

В разделе 13.1.1 вы узнали, как генерируется трассировка. Теперь выясним, из каких элементов образуется трассировка, генерируемая при выполнении кода в консоли.

Продолжим анализ трассировки из листинга 13.1. На рис. 13.3 представлены важнейшие элементы трассировки, сгенерированной при выполнении кода в консоли.

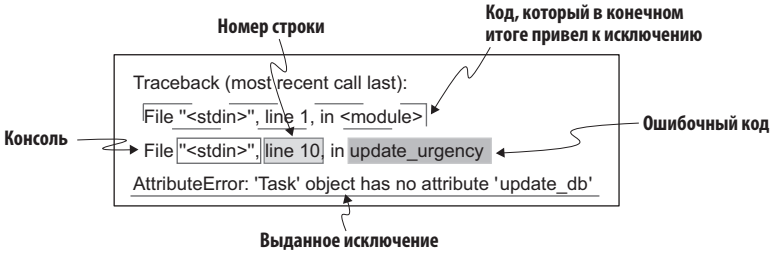


Рис. 13.3. Ключевые элементы трассировки, сгенерированной в консоли. Каждая строка представляет отдельную операцию, как показано на рис. 13.2. В каждую строку включаются такие ключевые элементы, как исходный файл операции, номер строки и ошибочный код. В последней строке указано исключение

Каждая строка в трассировке представляет операцию или вызов. В первой строке приведена строка кода, которая в конечном итоге привела к исключению: `task.update_urgency(4)`. Чтобы проанализировать ключевые элементы, присмотримся ко второй строке. Так как код из листинга 13.1 выполняется в консоли, источником операции является `<stdin>`, то есть стандартный ввод (консоль). Элемент `Line 10` (как показано в листинге 13.1, при нумерации строк не учитываются пустые строки, когда код отправляется на консоль) определяет строку, в которой было выдано исключение во время выполнения метода `update_urgency`. В данном случае это строка `self.update_db()`, которая не работает, потому что в классе отсутствует метод экземпляра `update_db`. Из-за этого, как показано в последней строке, выдается исключение `AttributeError`.

13.1.3. Анализ трассировки при выполнении скриптов

Раздел 13.1.2 был посвящен анализу трассировки, созданной при выполнении кода в консоли. На более общем уровне наш код часто выполняется в виде скриптов с использованием средств командной строки. В этом разделе будут рассмотрены более интересные аспекты трассировки.

Сохраните код в листинге 13.1 в файле скрипта с именем task_test.py. Обратите внимание на изменение в конце этого фрагмента:

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

    def _update_db(self):
        # Обновление записи в базе данных
        print("update the database")


    def update_urgency(self, urgency):
        self.urgency = urgency
        self.update_db()

if __name__ == "__main__":
    task = Task("Laundry", 3)
    task.update_urgency(4)
```

Как видите, вместо того чтобы создавать экземпляр и вызывать метод напрямую, как в листинге 13.1, мы теперь включаем соответствующий код в инструкцию с условием. Она выполняется только в том случае, если специальный атрибут `__name__` равен `"__main__"`. Эта инструкция позволяет выполнять файл в виде скрипта и в виде модуля, и ее включение считается хорошей практикой.

Если файл выполняется в виде скрипта, специальный атрибут `__name__` содержит значение `"__main__"`, так что инструкция дает результат `True` и выполняет включенные операции. Если же файл импортируется в виде модуля, то именем модуля становится имя файла, отличное от `"__main__"`, что предотвращает неожиданное выполнение включенного кода. В следующих разделах мы будем включать инструкцию `if` в свои файлы скриптов.

> **СОПРОВОЖДАЕМОСТЬ** Если ваши файлы Python предназначены для выполнения как в виде скрипта, так и в виде модуля, в большинстве случаев следует включать в инструкцию `if` операции, которые должны выполняться только в виде скрипта (`if __name__ == "__main__": #операции`). Если этого не сделать, то при импортировании файла в виде модуля эти операции тоже будут выполнены.

Вы можете выполнить следующую команду (листинг 13.2) в оболочке командной строки, например в приложении Terminal, если вы используете компьютер Mac, или cmd на компьютере Windows. Помните, что если в инструкции не указывается полный путь к файлу скрипта, то вам придется перейти в соответствующий каталог.

Листинг 13.2. Выполнение скрипта Python, генерирующего трассировку

```
$ python3 task_test.py  <-- $ обозначает приглашение командной строки
Traceback (most recent call last):
  File "/full_path/task_test.py", line 17, in <module>
task.update_urgency(4)
  File "/full_path/task_test.py", line 12, in update_urgency
self.update_db()
AttributeError: 'Task' object has no attribute 'update_db'. Did you mean:
➥ '_update_db'?
```

Я использую команду python3, так как macOS по умолчанию использует версию Python 2

По сравнению с трассировкой, генерируемой при выполнении кода в консоли, трассировка, генерируемая при выполнении скрипта, содержит дополнительную информацию. Как показывает листинг 13.2, в трассировке также приводится конкретная операция. Так, в методе `update_urgency` исключение `AttributeError` выдается в коде `self.update_db()`. Различия между трассировкой при выполнении кода в консоли и в скрипте возникают из-да того, что в этих двух режимах выполнения Python по-разному строит стек вызовов. Когда код выполняется в консоли, в стеке вызовов отслеживаются только строки, тогда как при выполнении скрипта отслеживаются конкретные операции.

13.1.4. Последний вызов в трассировке

Вы видели пару трассировок, генерируемых выполнением кода в консоли Python или выполнением скрипта из командной строки. Возможно, вы поняли, где следует искать проблему в трассировке. В этом разделе данная тема будет представлена более подробно.

В трассировке стек вызовов выводится в линейном виде сверху вниз. Иначе говоря, внизу приводится последний вызов, непосредственно связанный с выданным исключением. Следовательно, для решения проблемы следует сосредоточиться на последнем вызове.

В использованных примерах исключение `AttributeError` сообщает о сути проблемы: в объекте `'Task'` отсутствует атрибут `'update_db'`. В трассировке, сгенерированной при выполнении файла как скрипта (листинг 13.2), в сообщении об ошибке даже высказывается предположение: возможно, имеется в виду `'_update_db'`? (`Did you mean: 'update_db'?`). Впрочем, эта дополнительная информация может оказаться недоступной в более ранних версиях Python.

> **ОБРАТИТЕ ВНИМАНИЕ** Сообщение об исключении `Did you mean` («возможно, имеется в виду...») недавно появилось в Python. Увидите вы его или нет, зависит от версии Python и используемого редактора.

Это предположение — именно то, что нам нужно. Перейдите к определению метода `update_urgency`, указанного в последнем вызове в трассировке (номер

строки поможет быстро найти нужный фрагмент кода), и замените `update_db` на `_update_db`. Имена различаются наличием префикса `_`. После внесения изменения попробуйте снова запустить скрипт:

```
$ python3 task_test.py
# Вывод: update the database
```

Как и ожидалось, исключение `AttributeError` не выдается. После внесения изменений скрипт работает правильно.

13.1.5. Обсуждение

В этом разделе я на простом примере продемонстрировал структуру трассировки и показал, как читать ее для исправления тривиальной проблемы в коде. В общем случае последний вызов относится к проблеме, которую вы пытаетесь исправить. Но если в вашем проекте используются множественные зависимости, весьма вероятно, что полученные трассировки окажутся намного более сложными. И вполне может оказаться, что последний вызов в трассировке вообще не относится к вашему коду! В таком случае необходимо читать трассировку снизу вверх к более ранним вызовам, где вы найдете написанный вами код. Этот вызов с большой вероятностью отвечает за ту проблему, которую вы пытаетесь исправить.

13.1.6. Задача

Джо — джуниор-разработчик. Ему поручена отладка проблем в программной системе повышения производительности, которая разрабатывается его компанией. В процессе обучения он экспериментирует с трассировками. В листинге 13.1 трассировка включает два вызова. Как ему обновить класс `Task` добавлением и использованием нескольких дополнительных методов для получения трассировки, содержащей более двух вызовов?

> **ПОДСКАЗКА** Вы можете добавить метод или два метода, один из которых содержит ошибочный код, порождающий исключение. Используйте эти методы в других методах, чтобы создать несколько последовательных вызовов.

13.2. КАК ПРОВЕСТИ ОТЛАДКУ ПРОГРАММЫ В ИНТЕРАКТИВНОМ РЕЖИМЕ

Ошибки всегда лучше выявлять в фазе разработки, чтобы не приходилось реагировать на жалобы клиентов после выпуска продукта. Возможно, вам хотелось бы отлаживать программу после того, как все части будут (почти) завершены. Но я рекомендую заниматься отладкой шаг за шагом по ходу разработки, чтобы свести к минимуму вероятность ошибок. И хотя вы можете проверить трассировку исключения, чтобы исправить ошибку, этого не всегда достаточно для

тщательной проверки каждой задействованной операции, потому что исключение приводит к немедленному фатальному сбою приложения.

Другое важное средство отладки — интерактивный отладчик, позволяющий анализировать состояние приложения во время его работы в реальном времени. В этом разделе вы узнаете о важнейших возможностях встроенного отладчика. На рис. 13.4 представлены общие аспекты интерактивной отладки программы, которые будут рассмотрены далее.

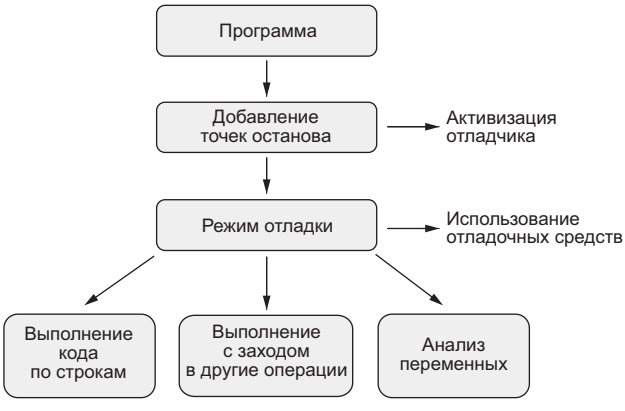


Рис. 13.4. Общие аспекты отладки программ в Python. В тех точках программы, где должна проводиться отладка, устанавливаются точки останова. Когда выполнение достигает точки останова, активизируется интерактивный отладчик. После этого вы можете активизировать различные отладочные операции, например выполнять код строку за строкой

Как упоминалось в главах 6 и 7, функции являются неотъемлемой частью приложений. Они также составляют основную часть тела пользовательского класса (глава 8). Написание функций, свободных от ошибок, является важнейшей целью каждого программиста, поэтому в данном разделе процесс интерактивной отладки будет продемонстрирован на примере функций.

13.2.1. Активизация отладчика в точке останова

В большинстве случаев поиск точки возникновения ошибки не занимает много времени, потому что когда в нашем приложении происходит сбой из-за исключения, сгенерированная трассировка сообщает о месте ошибки. Зная, где произошла ошибка, можно вмешаться в работу программы и установить точку останова для активизации отладчика.

ОСНОВНЫЕ ПОНЯТИЯ *Точкой останова* (breakpoint) называется точка, в которой по вашему требованию приложение прервет выполнение для проведения отладки.

Модуль pdb, являющийся частью стандартной библиотеки Python, предоставляет основную функциональность отладки через интерактивный отладчик. Чтобы активизировать отладчик, вызовите его функцию set_trace:

```
def create_task():
    import pdb; pdb.set_trace()  ◄——  Добавление точки останова

create_task()
# Выводимые строки:
--Return--
> <stdin>(2)create_task()->None
(Pdb)
```

В функции create_task импортируется модуль pdb и вызывается метод set_trace для создания точки останова. Обратите внимание: команду import можно было разместить за пределами функции, размещение ее перед set_trace — всего лишь распространенное соглашение. Когда вы вызываете эту функцию, активизируется отладчик; стандартное приглашение консоли Python >>> заменяется на (Pdb), это указывает на то, что Python перешел в режим отладки.

И хотя отладчик активизируется вызовом import pdb; pdb.set_trace(), я описываю здесь эту возможность только для того, чтобы вы понимали, что означает эта строка кода. Возможно, она встречалась вам в некоторых старых проектах. Правильнее использовать новую возможность, появившуюся в Python 3.7, и напрямую вызвать встроенную функцию breakpoint (активный отладчик завершается нажатием клавиши q):

```
def create_task():
    breakpoint()
create_task()
```

Из вывода видно, что при использовании функции breakpoint достигается тот же эффект, что и при активизации отладчика; это вспомогательная функция, которая вызывает set_trace во внутренней реализации. Режим отладки является интерактивным, и разработчику доступно множество инструментов для отладки ваших функций. Они рассматриваются в следующем разделе.

13.2.2. Построчное выполнение кода

При выполнении операции (например, вызова функции) выполняется сразу все ее тело со всеми внутренними операциями. Если функция выполняется успешно, вы получаете возвращаемое значение (или неявно None). Если происходит сбой, возникает исключение или вы получаете не то значение, которое ожидали. В любом случае операция выполняется слишком быстро, чтобы вам удалось точно определить, что происходит в функции. При выполнении кода строка за строкой легче понять каждый шаг операции, что повышает вероятность

исправления возможных ошибок. В этом разделе я покажу, как выполнять код строка за строкой. Не менее важна представленная в нем информация о некоторых ключевых функциях отладчика.

Допустим, в таск-менеджере мы получаем текстовые данные, содержащие информацию о задаче, и эти данные требуется преобразовать в экземпляр класса `Task`. В учебных целях добавьте точку останова в одну из этих функций и сохраните код в файле сценария с именем task_debug.py, как показано в листинге 13.3. Хотя отладка работает и при выполнении кода в консоли, настоящий проект больше похож на выполнение скрипта, поэтому в нашем примере будет использоваться отладка со скриптом.

Листинг 13.3. Создание функции с точкой останова (task_debug.py)

```
from collections import namedtuple
Task = namedtuple("Task", "title urgency")  <-- Создание именованного класса кортежа

def obtain_text_data(want_bad):
    text = "Laundry,3#" if want_bad else "Laundry,3"
    return text

def create_task(inject_bug: bool):
    breakpoint()  <-- Добавление точки останова
    task_text = obtain_text_data(inject_bug)  <-- Строка с номером 10
    title, urgency_text = task_text.split(",")
    urgency = int(urgency_text)
    task = Task(title, urgency)
    return task
if __name__ == "__main__":
    create_task(inject_bug=False)
```

Функция `create_task` создает задачу, обрабатывая полученные при вызове `obtain_text_data` текстовые данные. Чтобы смоделировать ситуации с неудачным вызовом функции, мы определяем логический аргумент для внедрения ошибки при необходимости. После такой подготовки перейдем к отладке скрипта без внедрения ошибки (`inject_bug=False`). Запустите оболочку командной строки и перейдите в текущий каталог, после чего выполните следующую команду, чтобы запустить скрипт:

```
$ python3 task_debug.py
> /full_path/task_debug.py(10)create_task()
-> task_text = obtain_text_data(inject_bug)
(Pdb)
```

Приглашение `(Pdb)` указывает на то, что Python работает в режиме отладки. Число `(10)` сообщает номер строки, и выполнение останавливается на функции `create_task`. Также указывается строка, которая будет выполнена следующей, а именно вызов функции `obtain_text_data`.

Чтобы выполнить эту строку, нажмите n (сокращение от «next» — «следующий»). Вы увидите, что интерпретатор выполняет текущую строку и в приглашении выводится следующая строка:

```
> /full_path/task_debug.py(11)create_task()
-> title, urgency_text = task_text.split(",")
(Pdb)
```

Чтобы выполнить следующую строку, нажмите «Return» (на Mac) или «Enter» (на компьютере с Windows); при этом будет повторена предыдущая команда: n. Выполнение переходит к следующей строке:

```
> /full_path/task_debug.py(12)create_task()
-> urgency = int(urgency_text)
(Pdb)
```

Как и следовало ожидать, при нажатии «Enter» или «Return» весь сценарий будет выполнен без каких-либо проблем. Но ведь это не интересно, верно? Рассмотрим другие варианты для отладки.

Иногда требуется просмотреть весь код функции, чтобы увидеть функцию в более общем контексте. Для этого нажмите клавишу l (L в нижнем регистре, сокращение от «list» — список):

```
(Pdb) l
  7
  8     def create_task(inject_bug: bool):
  9         breakpoint()
 10         task_text = obtain_text_data(inject_bug)
 11         title, urgency_text = task_text.split(",")
 12  ->     urgency = int(urgency_text)
 13         task = Task(title, urgency)
 14         return task
 15
 16     if __name__ == "__main__":
 17         create_task(inject_bug=False)
(Pdb)
```

Это весьма полезная информация — вы видите весь код, окружающий текущую строку, в котором все строки четко пронумерованы; текущая строка обозначается стрелкой.

13.2.3. Выполнение с заходом в функции

В разделе 13.2.2 в первой строке кода вызывалась другая функция: `task_text = obtain_text_data(inject_bug)`. Следует заметить, что вы получаете возвращаемое значение мгновенно. В данном случае все проходит нормально. Но если в вызванной функции возникнут проблемы, вам захочется просмотреть

ее код, чтобы понять, как она работает. Вы можете завершить текущий сеанс отладки нажатием клавиши q, а затем снова запустить сценарий из командной оболочки:

```
$ python3 task_debug.py
> /full_path/task_debug.py(10)create_task()
-> task_text = obtain_text_data(inject_bug)
(Pdb)
```

Вместо клавиши n, выполняющей следующую строку, нажмите s (сокращение от «step» — «шаг»); тем самым вы требуете исполнить следующий шаг программы. В данном случае следующим шагом становится вызов функции obtain_text_data:

```
(Pdb) s
--Call--
> /full_path/task_debug.py(4)obtain_text_data()
-> def obtain_text_data(want_bad):
```

Как видите, отладчик заходит в вызванную функцию, вместо того чтобы сразу получать ее возвращаемое значение. Продолжая нажимать s или Return, просмотрим полный код функции:

```
Pdb) s
> /full_path/task_debug.py(5)obtain_text_data()
-> text = "Laundry,3#" if want_bad else "Laundry,3"
(Pdb) s
> /full_path/task_debug.py(6)obtain_text_data()
-> return text
(Pdb) s
--Return--
> /full_path/task_debug.py(6)obtain_text_data()->'Laundry,3'
-> return text
```

В последней операции выводится возвращаемое значение, полученное при вызове функции: 'Laundry,3'. Продолжая нажимать s, мы вернемся к исходной функции create_task:

```
(Pdb) s
> /full_path/task_debug.py(11)create_task()
-> title, urgency_text = task_text.split(",")
```

Возможно, вы заметили сходство между командами n (next) и s (step), так как обе в большинстве случаев выполняют следующую строку. Различие в том, что step, как вы уже видели, позволяет вызвать другую функцию. На рис. 13.5 сравниваются команды n и s.

Хотя на рис. 13.5 команда step пытается выполнить следующую строку, она останавливается в ближайшей возможной точке. В данном случае такой точкой становится вызов функции obtain_text_data.

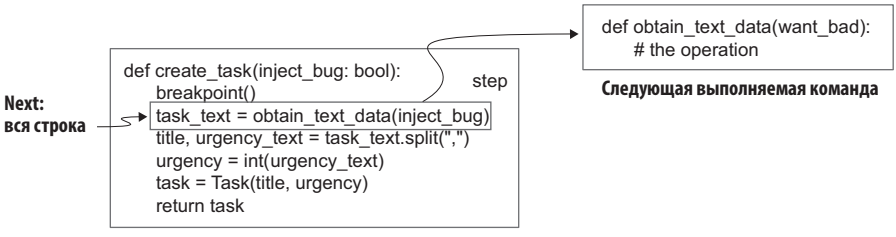


Рис. 13.5. Сравнение команд next и step в режиме отладки. Команда next выполняет следующую строку; команда step пытается выполнить следующую строку, но останавливается в следующей возможной точке. В данном примере команда step передает управление в другую функцию

13.2.4. Анализ важных переменных

Мы видим, какой код выполняется, но до сих пор активно не вмешивались в работу программы. Иногда вызов функции не работает, потому что функция получает неправильные аргументы. Даже если аргументы относятся к правильному типу, нельзя исключить, что значения окажутся несовместимыми. В этом разделе вы узнаете, как проверить значения переменных внутри функции. Замените последнюю строку скрипта (`task_debug.py`) на `create_task(inject_bug=True)` и запустите скрипт из командной строки:

```
> /full_path/task_debug.py(10)create_task()
-> task_text = obtain_text_data(inject_bug)
(Pdb) n
> /full_path/task_debug.py(11)create_task()
-> title, urgency_text = task_text.split(",")
(Pdb) n
> /full_path/task_debug.py(12)create_task()
-> urgency = int(urgency_text)
```

Допустим, вы знаете, что в следующей строке будет выдано исключение `AttributeError`. Можно проверить соответствующие переменные, чтобы узнать потенциальную причину исключения:

```
(Pdb) p urgency_text
'3#'
```

Как видно из приведенного фрагмента, команда `p` используется для получения значения переменной. Если вы хотите вывести сразу несколько переменных, перечислите их, разделяя запятыми:

```
(Pdb) p urgency_text, task_text
('3#', 'Laundry,3#')
```

Перечислять все переменные, которые нужно проверить, будет довольно утомительно. Лучше воспользоваться функциональностью, которая позволяет вызвать

функцию прямо в отладчике. В данном случае следует вызвать функцию `locals`, выводящую содержимое локального пространства имен (раздел 10.4):

```
(Pdb) locals()
{'inject_bug': True, 'task_text': 'Laundry,3#', 'title':
➥ 'Laundry', 'urgency_text': '3#'}
```

Проверка всех переменных в локальной области видимости функции дает полную картину состояния функции.

13.2.5. Обсуждение

Трассировка (раздел 13.1) описывает состояние приложения на момент его остановки, и все, что могло привести к исключению, уже произошло. Статическая информация не дает возможности проверить каждую операцию в процессе выполнения, все случается слишком быстро. Напротив, отладчик, описанный выше, работает по требованию пользователя. Вы сами решаете, когда приложение может перейти к следующей строке. Тем самым вы получаете время для тщательного анализа каждой строки программы и поиска возможной причины ошибки. Что еще важнее, отладчик работает в интерактивном режиме, и вы можете освоить и другие команды помимо `n`, `l`, `s` и `p`. За дополнительной информацией об интерактивном отладчике Python обращайтесь на официальный веб-сайт Python по адресу https:// docs.python.org/3/library/pdb.html.

13.2.6. Задача

Дилан — старательный ученик, который хочет во всех подробностях изучить каждую возможность языка Python. Осваивая процесс отладки, он хочет знать, что происходит во время вызова функции в контексте локального пространства имен. В описанном в разделе 13.2.4 примере вместо вызова `locals` для получения переменных в локальной области видимости после выполнения нескольких строк он решает вызвать `locals` вслед за запуском отладчика. Как будут изменяться списки переменных во время вызова функции?

> **ПОДСКАЗКА** Пространства имен изменяются динамически. После того как в процессе выполнения будет создана новая переменная, она регистрируется в пространстве имен.

13.3. КАК ТЕСТИРОВАТЬ ФУНКЦИИ АВТОМАТИЧЕСКИ

После завершения работы над функциональностью программы и устранения очевидных ошибок с использованием трассировки или интерактивной отладки вам начинает казаться, что приложение почти готово для выпуска. Остается сделать еще одно: тщательно протестировать программу. Тестирование — широкое понятие, оно может проводиться самыми разными способами. Когда вы устраняете ошибки

в своем приложении — это тестирование. Когда вы вызываете функции, чтобы убедиться в том, что они работают в вашем приложении так, как ожидалось, — это тоже тестирование. Впрочем, все это примеры ручного тестирования.

Ручное тестирование приемлемо при работе над небольшими проектами, но оно создает существенную нагрузку в крупномасштабных проектах. Каждый раз, когда вы вносите изменения в код, вы должны проверять всю задействованную функциональность и убеждаться в том, что изменения не нарушат ее работу. Естественно предположить, что ручное тестирование займет много времени и задержит работу над проектом. К счастью, для приложения можно разработать систему автоматического тестирования. Если говорить конкретнее, написать код, который тестирует кодовую базу вашего приложения. После внесения изменений в кодовую базу вы просто запускаете тестовый код, что сэкономит немало времени. В этом разделе описаны некоторые важные приемы реализации автоматического тестирования, при этом особое внимание уделяется функциям.

СОПРОВОЖДАЕМОСТЬ Тестирование — важный механизм, обеспечивающий сопровождаемость вашего кода. В разделах 13.3 и 13.4 приводится только вводная информация. Если ваша работа в основном связана с тестированием, стоит обратиться к учебным материалам, например «The Art of Unit Testing: With Examples in C#» Роя Ошероува (Roy Osherove, Manning, 2019)[1].

13.3.1. Принципы тестирования функций

Вы уже знаете, что функции исключительно важны для приложения. Если вы убедитесь в том, что каждая функция работает так, как ожидалось, ваше приложение устоит перед возможными испытаниями. В этом разделе описаны ключевые аспекты тестирования функций.

Начнем с простой функции, которая станет основой для тестирования более сложных. Допустим, наш таск-менеджер содержит функцию для создания задачи в виде экземпляра класса `Task` из строки. Функция сохраняется в файле task_func.py, чтобы ее можно было использовать в тестах, как показано в листинге 13.4.

Листинг 13.4. Определение тестируемой функции (task_func.py)

```
class Task:  <-------  Создает пользовательский класс
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

def create_task(text):
    title, urgency_text = text.split(",")
    urgency = int(urgency_text)
    task = Task(title, urgency)
    return task
```

[1] Ошероув, Рой. «Искусство автономного тестирования с примерами на C#».

Для конкретной функциональности в проекте обычно ожидается (несмотря на все возможные различия в подробностях реализации), что при определенном вводе функция будет возвращать определенный результат. Например, как бы вы ни изменяли подробности реализации `create_task`, ожидается, что следующее тестовое условие будет истинным:

```
assert create_task("Laundry,3").__dict__ == Task("Laundry", 3).__dict__
```

Здесь инструкция `assert` используется для проверки правильности работы функции. В данном случае ожидается, что словарные представления этих двух экземпляров будут совпадать. Учтите, что экземпляры пользовательского класса изначально не равны, но их словарные представления можно сравнивать на равенство. Вообще говоря, уверенность в том, что для конкретного ввода всегда будет выдаваться конкретный вывод, лежит в основе тестирования функций. На рис. 13.6 показано, как происходит тестирование.

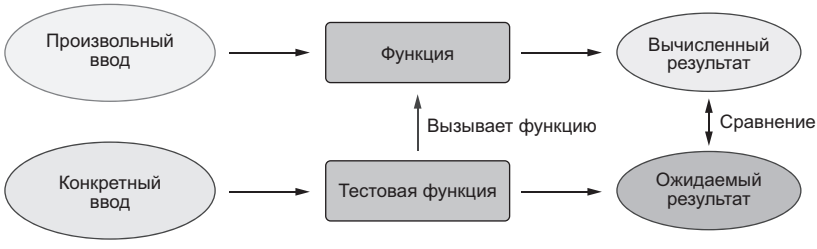


Рис. 13.6. Общий процесс тестирования функций. В тестовой функции конкретная совокупность входных данных используется для вызова функции, а полученный результат сравнивается с ожидаемым

13.3.2. Создание подкласса TestCase для тестирования функций

Теперь, когда вы знакомы с основами тестирования функций, перейдем к реализации автоматического тестирования с использованием модуля `unittest` (часть стандартной библиотеки Python). Этот модуль предоставляет важные средства для автоматического тестирования, а именно: класс `TestCase` этого модуля позволяет протестировать нашу функцию, как показано в листинге 13.5.

Листинг 13.5. Тестирование функции с использованием TestCase (test_task_func.py)

```
from task_func import Task, create_task ◄── Импортирует класс и функцию из файла сценария
import unittest  ◄── Импортирует модуль
class TestTaskCreation(unittest.TestCase):  ◄── Наследует от класса TestCase
    def test_create_task(self):
        task_text = "Laundry,3"
```

```
        created_task = create_task(task_text)  <-- Вызывает тестируемую функцию
        self.assertEqual(created_task.__dict__,
        ➥ Task("Laundry", 3).__dict__)
if __name__ == "__main__":
    unittest.main()
```

ПРИМЕЧАНИЕ Если у вас возникнут проблемы с импортированием класса и функции, откройте папку этой главы в интегрированной среде разработки Python (IDE).

В листинге 13.5 класс `TestTaskCreation` создается наследованием от класса `TestCase`. Тестовым классам принято назначать имена, начинающиеся с префикса `Test`. В теле класса определяется метод экземпляра, предназначенный для тестирования функции `create_task`. Важно снабдить имя этого метода префиксом `test_ prefix`, чтобы указать, что Python должен вызывать этот метод при выполнении теста. На рис. 13.7 изображена структура тестового класса в зависимости от тестируемых функций.

УДОБОЧИТАЕМОСТЬ Присвойте этому тестовому классу имя, начинающееся с префикса `Test`, и включите в него описание конкретной функциональности, которая тестируется классом. Методы класса должны начинаться с префикса `test_`, чтобы Python выполнил эти методы в процессе тестирования.

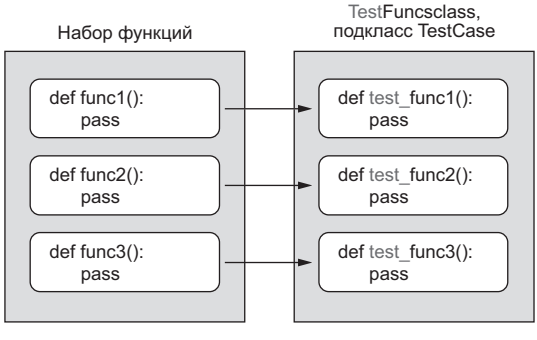


Рис. 13.7. Создание тестового класса для тестирования набора функций. Тестовая функция должна использовать префикс `test_`, за которым следует имя тестируемой функции. Имя класса должно начинаться с префикса `Test`, а сам класс — быть подклассом `TestCase`

Метод `test_create_task` вызывает тестируемую функцию (`create_task`) с конкретным вводом и сравнивает возвращаемое значение с ожидаемым результатом. Сравнение осуществляется вызовом `assertEqual`, который проверяет, содержат ли два экземпляра класса `Task` равные значения. Если условие истинно, можно быть уверенным в том, что функция работает так, как ожидалось. В последней

строке вызывается метод `unittest.main()`, который выполняет все заданные тесты в классе `TestTaskCreation`. После такой подготовки все готово для тестирования функции из командной оболочки:

```
$ python3 test_task_func.py
# output the following lines:
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Пока мы создали только один тестовый вариант — `test_create_task`. Однако вы также можете определить несколько тестовых вариантов.

> **ОСНОВНЫЕ ПОНЯТИЯ** *Тестовым вариантом*, или *тест-кейсом* (test case), называется отдельная единица тестирования, которая проверяет получение конкретного ответа для конкретных входных данных.

Допустим, у вас имеется другая функция, которая создает экземпляр класса `Task` из объекта `dict`. Эта функция добавляется в файл task_func.py:

```
def create_task_from_dict(task_data):
    title = task_data["title"]
    urgency = task_data["urgency"]
    task = Task(title, urgency)
    return task
```

Функция должна быть простой: она получает необходимые значения из объекта `dict` и создает экземпляр. Обновите тестовый класс, чтобы он тестировал и эту функцию, как показано в листинге 13.6.

Листинг 13.6. Тестирование нескольких функций (test_task_func.py)

```
from task_func import Task, create_task, create_task_from_dict
import unittest
class TestTaskCreation(unittest.TestCase):
    def test_create_task(self):
        task_text = "Laundry,3"
        created_task = create_task(task_text)
        self.assertEqual(created_task.__dict__,
        ➥ Task("Laundry", 3).__dict__)

    def test_create_task_from_dict(self):
        task_data = {"title": "Laundry", "urgency": 3}
        created_task = create_task_from_dict(task_data)
        self.assertEqual(created_task.__dict__,
        ➥ Task("Laundry", 3).__dict__)

if __name__ == "__main__":
unittest.main()
```

Как и в случае с `test_create_task`, мы определяем метод, имя которого начинается с `test_`. В этом методе мы проверяем, что функция работает в конкретном случае, который мы выбрали. Снова запустите тест:

```
$ python3 test_task_func.py
# output the following lines:
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Как видите, в тестовом классе были определены два метода, поэтому Python выполнил за нас два теста, и оба прошли успешно. Кстати, обратите внимание на две точки в первой строке: количество точек соответствует количеству выполненных тестов.

13.3.3. Подготовка теста

Ранее было показано, как тестовый класс тестирует сразу две функции. У этих двух функций есть нечто общее: обе создают экземпляр класса `Task`. Когда мы тестируем их, мы также создаем экземпляр класса `Task` для выполнения сравнения. Вспомните (раздел 2.1.4), что повторение — возможный признак необходимости рефакторинга. В этом разделе мы настроим тест, который способен выделить общие аспекты тестовых функций.

> **СОПРОВОЖДАЕМОСТЬ** Всегда обращайте внимание на возможности рефакторинга, например повторение кода. Рефакторинг улучшает сопровождаемость кодовой базы.

Класс `TestClass` содержит метод `setUp`, который переопределяется. Этот метод вызывается перед выполнением каждого теста, и вы можете воспользоваться им для выполнения общих для тестовых методов операций. Заметьте, что эти операции зависят от данных, которые были настроены для тестирования. Пример приведен в листинге 13.7.

Листинг 13.7. Переопределение метода setUp (test_task_func.py)

```
from task_func import Task, create_task, create_task_from_dict
import unittest

class TestTaskCreation(unittest.TestCase):
    def setUp(self):
        task_to_compare = Task("Laundry", 3)
self.task_dict = task_to_compare.__dict__

    def test_create_task(self):
        task_text = "Laundry,3"
        created_task = create_task(task_text)
self.assertEqual(created_task.__dict__, self.task_dict)
```

```
    def test_create_task_from_dict(self):
        task_data = {"title": "Laundry", "urgency": 3}
        created_task = create_task_from_dict(task_data)
self.assertEqual(created_task.__dict__, self.task_dict)

if __name__ == "__main__":
unittest.main()
```

В выделенных фрагментах листинга 13.7 класс обновляется посредством добавления атрибута. А именно определяется атрибут `task_dict`, содержащий объект `dict`, который будет использоваться тестовыми методами для проверки равенства. В тестовых методах к атрибуту экземпляра `task_dict` можно обращаться напрямую, не нужно создавать дубликаты экземпляра для сравнения. Если запустить тестовый сценарий повторно, вы получите тот же результат.

> **ОБРАТИТЕ ВНИМАНИЕ** Как вы, возможно, заметили, в именах методов модуля `unittest` используется схема «верблюжьего регистра» (например, `setUp` и `assertEqual`) вместо «змеиного регистра» (например, `set_up` и `assert_equal`). Такой выбор объясняется тем, что эти имена были унаследованы от средств на базе Java, в которых использовался «верблюжий регистр».

13.3.4. Обсуждение

Из всех методов нашего тестового класса используется только `assertEqual` для проверки равенства между ожидаемым и сгенерированным результатом. Но существуют и другие удобные методы, проверяющие, что сгенерированный результат соответствует ожидаемому. Так, метод `assertIn(a, b)` проверяет, что `a` содержится в `b`, а метод `assertTrue(a)` проверяет, что значение `a` равно `True`. Эти методы просты в использовании, и вам стоит ознакомиться с ними. Информацию о них вы найдете в официальной документации модуля `unittest` (https://docs.python.org/3/library/unittest.html).

13.3.5. Задача

Аарон работает над приложением для получения прогноза погоды. Сейчас он разбирается с запуском модульных тестов. Он следовал рекомендациям раздела 13.3, определил две функции и протестировал их при помощи класса `TestTaskCreation`. Но теперь ему потребовалось написать еще одну функцию и соответствующий тестовый метод. Допустим, функция создает экземпляр класса `Task` из объекта `tuple ("Laundry", 3)`. Сможете ли вы предложить решение?

> **ПОДСКАЗКА** Вероятно, функции стоит присвоить имя `create_task_from_tuple`, а затем использовать в ней распаковку кортежа (раздел 4.4) для получения названия и степени срочности, которые будут использованы для создания экземпляра.

13.4. КАК ПРОВЕСТИ АВТОМАТИЧЕСКОЕ ТЕСТИРОВАНИЕ КЛАССА

Хотя функции являются неотъемлемой частью приложений, пользовательские классы играют в приложении ключевую роль, так как они определяют модели данных, которые объединяют необходимые данные и функциональность в единое целое. Обычно вам не приходится беспокоиться о тестировании атрибутов пользовательского класса, так как эти атрибуты должны определяться напрямую. Следовательно, тестирование класса в основном направлено на тестирование его методов, что и рассматривается в этом разделе.

13.4.1. Создание подкласса TestCase для тестирования класса

По сути, методы являются функциями, а отдельное название — *метод* — связано только с тем, что они определяются в классах. Таким образом, тестирование методов класса сводится к тестированию функций — теме, подробно рассмотренной в разделе 13.3. Как будет показано в разделе ниже, для тестирования класса также создается подкласс `TestCase`. В примерах используются методы класса, но по тем же принципам проводится тестирование методов экземпляров и статических методов.

В разделе 13.3 мы работали над двумя функциями: `create_task` и `create_task_from_dict`. Как вы уже догадались, они преобразуются в методы.

Так как эти два метода используют конструктор для создания экземпляра класса `Task`, они идеально подходят для реализации в виде методов класса, как показано в листинге 13.8.

Листинг 13.8. Создание класса для тестирования (task_class.py)

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

    @classmethod
    def task_from_text(cls, text_data):
        title, urgency_text = text_data.split(",")
        urgency = int(urgency_text)
        task = cls(title, urgency)
        return task

    @classmethod
    def task_from_dict(cls, task_data):
        title = task_data["title"]
        urgency = task_data["urgency"]
        task = cls(title, urgency)
        return task
```

В листинге 13.8 класс `Task` содержит методы класса `task_from_text` и `task_from_dict`, полученные преобразованием функций `create_task` и `create_task_from_dict` соответственно.

> **НАПОМИНАНИЕ** Метод класса получает первый аргумент `cls`, который содержит ссылку на класс (раздел 8.2).

Чтобы протестировать этот класс, мы создадим класс `TestTask` как подкласс класса `TestCase`, в котором определяются два метода, соответствующие двум методам класса. Сохраните код из листинга 13.9 в файле с именем test_task_class.py.

Листинг 13.9. Создание класса для тестирования класса (test_task_class.py)

```
from task_class import Task
import unittest

class TestTask(unittest.TestCase):
    def setUp(self):  <-- Подготовка теста
        task_to_compare = Task("Laundry", 3)
        self.task_dict = task_to_compare.__dict__

    def test_create_task_from_text(self):
        task_text = "Laundry,3"
        created_task = Task.task_from_text(task_text)
        self.assertEqual(created_task.__dict__, self.task_dict)

    def test_create_task_from_dict(self):
        task_data = {"title": "Laundry", "urgency": 3}
        created_task = Task.task_from_dict(task_data)
        self.assertEqual(created_task.__dict__, self.task_dict)

if __name__ == "__main__":
    unittest.main()
```

По аналогии с классом `TestCreationTask` тестовые методы определяются с именами, начинающимися с `test_` в классе `TestTask`, чтобы при выполнении скрипта все эти тестовые методы были выполнены автоматически. Следующий фрагмент кода демонстрирует этот эффект:

```
$ python3 test_task_class.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s
OK
```

Как и предполагалось, оба теста были выполнены и в обоих случаях никаких проблем не возникло.

13.4.2. Реакция на сбои в тестах

Цель тестирования — убедиться в том, что тестируемые модули работают так, как ожидается. Разумеется, успешное прохождение всех тестов никогда не гарантировано. Если некоторые тесты не проходят, необходимо понять, как

реагировать на эти сбои. Допустим, в класс `Task` из листинга 13.8 была добавлена следующая функция:

```
def formatted_display(self):
    displayed_text = f"{self.title} ({self.urgency})"
    return displayed_text
```

Этот метод экземпляра создает отформатированное описание задачи. Чтобы протестировать этот метод экземпляра, в класс `TestTask` добавляется следующий тестовый метод (листинг 13.9):

```
def test_formatted_display(self):
    task = Task("Laundry", 3)
    displayed_text = task.formatted_display()
    self.assertEqual(displayed_text, "Laundry(3)")
```

Вы уже заметили, что для моделирования тестового сбоя я намеренно опустил пробел между названием и степенью срочности задачи в вызове `assertEqual`. При выполнении теста должна произойти ошибка:

```
$ python3 test_task_class.py
..F
======================================================================
FAIL: test_formatted_display (__main__.TestTask)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/full_path/test_task_class.py", line 22, in test_formatted_display
    self.assertEqual(displayed_text, "Laundry(3)")
AssertionError: 'Laundry (3)' != 'Laundry(3)'
- Laundry (3)
?        -
+ Laundry(3)

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

Вместо трех точек, представляющих три успешных теста, выводится последовательность `..F`. Символ `F` обозначает сбой теста, а подробное описание сообщает причину: `AssertionError` между этими двумя строками. Сообщение предоставляет достаточно информации для решения проблемы. Чтобы строки были равными, в строку `'Laundry(3)'` необходимо добавить пробел.

13.4.3. Обсуждение

Тестирование должно быть неотъемлемым этапом разработки, обеспечивающим качество продукта. Во время разработки следует сосредоточиться на устранении ошибок в минимально возможных фрагментах кода. Другими словами, следует проводить ручное тестирование в том или ином объеме при завершении любой функциональности, даже самой незначительной. Не следует думать: «Сейчас я займусь разработкой, а ручное тестирование подождет». Намного проще решать потенциальные проблемы непосредственно во время работы над ними.

Автоматическое тестирование — мощная возможность, но, вероятно, вам придется освежить в памяти сведения о том, что же происходит в программе, прежде чем вы справитесь с обнаруженными проблемами.

13.4.4. Задача

Сбой теста не всегда происходит из-за ошибки `AssertionError` в тестовом классе. Также может оказаться, что проблема связана с самим кодом. Обновите метод `formatted_display`, чтобы он выдавал исключение, и посмотрите, что происходит во время тестирования.

> **ПОДСКАЗКА** Исключения проще всего выдавать вручную, например командой `raise TypeError`.

ИТОГИ

- Трассировка содержит подробную информацию, которая показывает, как произошло исключение. Подробная информация представляет серию операций или вызовов.
- Пытаясь решить проблему с помощью трассировки, сосредоточьтесь на последнем вызове, где произошло исключение.
- Чтобы внимательно проанализировать, как выполняется тот или иной код, можно установить точку останова, активизирующую отладчик. Модуль `pdb` предназначен для интерактивной отладки.
- В интерактивном отладчике выполните программу по строкам (команда `n`). Режим пошагового выполнения поможет определить, из-за какой строки возникла проблема.
- Если вы хотите выполнить управление с заходом в другую операцию (например, вызов функции), используйте команду `s` вместо `n`. Эта команда выполняет всю строку.
- Модуль `unittest` предоставляет функциональность автоматического тестирования. Он содержит класс `TestClass`, на основе которого можно определять собственные тестовые классы, создавая подклассы.
- Соблюдайте правила выбора имен при создании тестовых методов. Имя метода должно начинаться с `test_`, а имя класса должно начинаться с `Test`.
- Тестирование функции основано на вашей уверенности в ожидаемом поведении функции. Для заранее определенного ввода функция должна генерировать определенный результат без какой-либо неоднозначности.
- В большинстве случаев для проверки результата теста используется `assertEqual`. В классе `TestCase` также можно использовать другие методы.
- Тестирование класса, по сути, сводится к тестированию его методов, а для тестирования методов используются те же средства, что и для тестирования функций.

Часть 6

Построение веб-приложения

Уровень мастерства шахматиста оценивается по результатам реальной партии с другим игроком, а не по количеству известных ему дебютов. Чтобы сыграть реальную партию, игрок должен знать не только дебюты, но и игру в миттельшпиле и эндшпиле.

Для программиста завершение проекта можно сравнить с шахматной партией: он должен обладать разносторонними знаниями, позволяющими правильно выбрать модель данных, написать функции и определить четко структурированные классы (и это еще не все). В этой части мы завершим наше приложение — таск-менеджер, о котором было рассказано в первых пяти частях. Мы не только освоим новые возможности, но и применим их в контексте реального проекта. Завершение проекта всегда дает законный повод для гордости. Согласны?

14 Завершение реального проекта

В этой главе

- ✓ Настройка виртуальных сред
- ✓ Построение моделей данных
- ✓ Работа с локальной базой данных
- ✓ Построение веб-приложения

В главах 2–12 рассматривались отдельные функциональные аспекты Python, при этом в них приводилось множество перекрестных ссылок на сопутствующие темы. Например, при знакомстве со встроенными типами данных (главы 2–5) были созданы функции для выполнения некоторой повторяющейся работы. При обсуждении функций (главы 6 и 7) и классов (главы 8 и 9) мы использовали встроенные типы данных. Примеры, приводившиеся в контексте таск-менеджера, показывают, что при решении реальных задач эти средства зависят друг от друга. Решение изолированных задач полезно для изучения соответствующих подходов и методик. И все же конечной целью изучения этих отдельных средств становится их совместное использование для завершения реального проекта от начала и до конца.

В этой главе мы разработаем проект таск-менеджера (раздел 1.4.3) с самого начала, с созданием виртуальной среды (раздел 14.1), определением подходящих моделей данных (раздел 14.2), использованием внутренней базы данных

(раздел 14.3), реализацией клиентского приложения (раздел 14.4) и публикацией пакета для распространения. Следует заметить, что хотя мы изучаем ряд новых средств (например, работу с локальной базой данных), наше внимание будет сосредоточено на синтезе средств, рассмотренных в главах 2–12.

14.1. КАК ИСПОЛЬЗОВАТЬ ДЛЯ ПРОЕКТА ВИРТУАЛЬНУЮ СРЕДУ

Как упоминалось в главе 1 (раздел 1.2), существует много разных пакетов Python с открытым исходным кодом, которые можно использовать в нашем проекте. Установить сторонние пакеты можно с помощью менеджера пакетов `pip`. Это инструмент командной строки, который позволяет устанавливать и удалять пакеты Python всего одной командой.

По умолчанию эти пакеты устанавливаются на уровне системы; это означает, что все ваши проекты должны их совместно использовать. Однако разные проекты могут требовать разных версий пакетов. Если же версии общесистемных пакетов отличаются от версий, необходимых конкретно вашему проекту, разрешить эти конфликты не так просто. В этом разделе вы узнаете, как решать подобные проблемы при помощи виртуальной среды.

14.1.1. Причины использования виртуальных сред

Виртуальные среды предназначены для решения проблем с разными проектами, которым требуются пакеты в различных версиях. Что же такое виртуальная среда и что она может сделать?

Начнем с проблемы конфликта пакетов. Если у вас имеется только один проект, проблем с использованием пакетов не будет. Часто вам приходится работать над несколькими проектами одновременно — в таких ситуациях могут возникнуть проблемы с управлением пакетами. В одном проекте используется пакет A, версия 1.0; в другом проекте требуется пакет A, версия 1.5, и вы обновляете пакет до версии 1.5. Похоже, вы создали проблему. Когда вы возвращаетесь к первому проекту, ваш код может перестать работать, потому что какие-то возможности пакета A были удалены в версии 1.5.

Конечно, можно вернуться к версии 1.0 для работы над первым проектом, но когда вы захотите поработать над вторым проектом, обновление придется выполнять заново. Вряд ли кому-то понравятся эти постоянные откаты и обновления.

В такой ситуации лучше всего воспользоваться виртуальной средой. *Виртуальные среды* представляют собой изолированные рабочие каталоги, в которых устанавливаются пакеты, необходимые для проекта. Так как каждый проект использует собственную виртуальную среду, вы можете установить разные пакеты (или разные версии пакетов) в соответствующих рабочих каталогах. Более

того, в современных средствах управления виртуальными средами (такими, как conda) для каждой виртуальной среды устанавливается отдельная версия Python со своим набором пакетов. Это обеспечивает бóльшую гибкость управления средами для отдельных проектов, как показано на рис. 14.1.

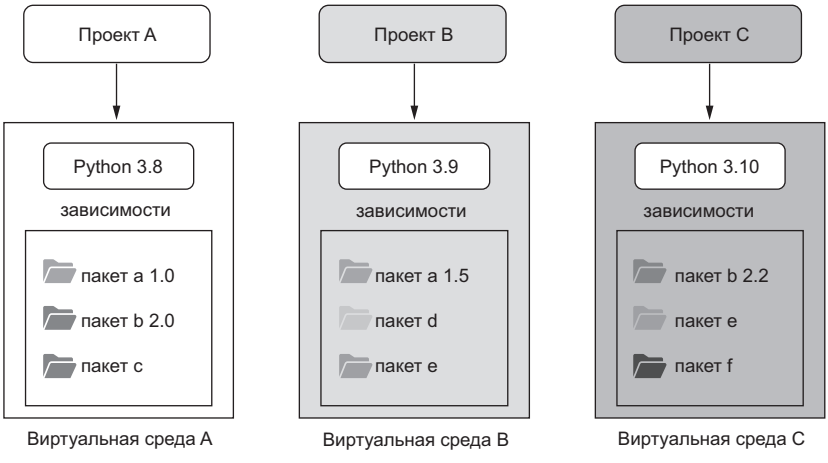


Рис. 14.1. Создание нескольких виртуальных сред для разных проектов. В каждой виртуальной среде используется отдельная версия Python и отдельный набор сторонних зависимостей

ОСНОВНЫЕ ПОНЯТИЯ *Виртуальная среда* (virtual environment) выглядит как дерево каталогов, содержащее Python и сторонние зависимости, изолированные от копий, установленных на компьютере

На рис. 14.1 изображены три проекта с виртуальными средами. В виртуальной среде используется нужная версия Python и сторонние зависимости с нужными версиями. При использовании виртуальной среды вам не приходится беспокоиться о том, что разным проектам потребуются разные версии пакета, потому что каждый проект использует собственные зависимости.

СОПРОВОЖДАЕМОСТЬ Создайте отдельную виртуальную среду для каждого проекта, чтобы предотвратить возможные конфликты зависимостей в ваших проектах.

14.1.2. Создание виртуальной среды для каждого проекта

Корневая причина проблемы с совместным использованием пакетов, описанной в разделе 14.1.1, кроется в том, что пакеты на компьютере были установлены на

системном уровне. А если бы их можно было устанавливать по отдельности для каждого проекта? Именно это и позволяет сделать виртуальная среда.

Модуль `venv`, входящий в стандартную библиотеку Python, предоставляет основную функциональность управления виртуальными средами. Для управления виртуальными средами используются также некоторые сторонние инструменты, например `conda` и `virtualenv`. Несмотря на небольшие различия в функциональности, основные принципы остаются теми же, что во встроенном модуле `venv`. По этой причине в своих примерах я буду использовать `venv`.

Для создания виртуальной среды необходимо открыть командную оболочку, например Terminal для Mac или cmd для Windows. В нашем проекте таск-менеджера создается каталог taskier_app, который неоднократно упоминается в этой главе. Перейдите в каталог taskier_app (используйте команду `cd` для смены каталога) и выполните следующую команду:

```
$ python3 -m venv taskier-env
```

Если вы работаете в Windows, возможно, вам придется использовать `python` вместо `python3` (я ввожу эту команду, потому что работаю на Mac). Команда создает виртуальную среду с именем `taskier-env`, так как эта среда будет использоваться для построения приложения `taskier`. Если вы работаете с несколькими средами, назначайте им имена, связанные с проектами, чтобы знать, к какому проекту относится та или иная среда. Таким образом, каждый проект будет использовать для управления зависимостями свою виртуальную среду с подходящим именем и между проектами не будет конфликтов зависимостей.

> **СОПРОВОЖДАЕМОСТЬ** Назначайте имена виртуальных сред по именам проектов, к которым они относятся.

Обратите внимание на то, что в каталоге присутствует папка с именем taskier-env. В этой папке находятся все папки и файлы, необходимые для виртуальной среды. Если вам это интересно, папка bin (в macOS; в Windows это будет папка с именем Scripts или что-нибудь подобное) содержит необходимые инструменты для среды, включая ссылку на интерпретатор Python, менеджер пакетов `pip` (раздел 14.1.3) и скрипты активизации (раздел 14.1.3).

14.1.3. Установка пакетов в виртуальной среде

Как говорилось ранее, виртуальные среды представляют собой изолированные рабочие каталоги для ваших проектов. В них можно безопасно установить любые необходимые пакеты, и на другие проекты они не повлияют. В этом разделе я покажу, как установить пакеты в виртуальной среде.

Начните с создания виртуальной среды для проекта `taskier-env`. Чтобы использовать среду, выполните следующую команду:

```
# Для Mac:
$ source taskier-env/bin/activate
# Для Windows:
> taskier-env\Scripts\activate.bat
```

ПРИМЕЧАНИЕ Если команда не работает в командной оболочке, обращайтесь к странице на официальном веб-сайте Python за дальнейшими инструкциями: https://docs.python.org/3/library/venv.html.

Команда активизирует виртуальную среду, позволяя установить в ней пакеты. Вы увидите, что в командной строке имя виртуальной среды используется в качестве префикса (`taskier-env`), а это означает, что среда активизирована и готова к установке пакетов.

Из всех программ установки пакетов Python наиболее популярной является `pip`. Для нашего таск-менеджера установите также библиотеку `streamlit`, и она предоставит инструменты для построения клиентской части проекта в виде веб-приложения. Я выбрал эту библиотеку, потому что с ее помощью легко построить веб-приложение, которое позволяет сосредоточиться на содержимом, а не на макетировании веб-элементов. Вот команда для установки `streamlit` (версии 1.10.0 на момент написания книги):

```
$ pip install streamlit==1.10.0
```

Для воспроизводимости результатов я рекомендую вам установить ту же версию. Впрочем, вполне возможно, что веб-приложение будет работать и с новейшей версией `streamlit`.

14.1.4. Использование виртуальных сред в Visual Studio Code

В этом проекте мы воспользуемся Visual Studio Code (VSC) как инструментом кодинга, потому что это интегрированная среда разработки (IDE) с открытым исходным кодом и мощными возможностями расширения. В этом разделе я покажу, как использовать виртуальные среды в VSC.

Откройте каталог проекта (taskier_app) в VSC; нажмите Cmd+Shift+P (Mac) или Ctrl+Shift+P (Windows) для вывода меню и выберите команду Python: Select Interpreter, которая выводит список доступных виртуальных сред. В списке должна присутствовать виртуальная среда taskier-env. Выберите вариант 'taskier-env': venv (рис. 14.2).

ПРИМЕЧАНИЕ Каталог проекта (`taskier_app`) необходимо открыть командой File ▸ Open Folder в VSC. В противном случае среда не появится в списке.

Чтобы убедиться в том, что вы действительно используете эту среду, создайте файл (допустим, test_env.py) в родительском каталоге (taskier_app). При открытии

этого файла в нижней части окна VSC должна появиться строка состояния, как показано на рис. 14.3.

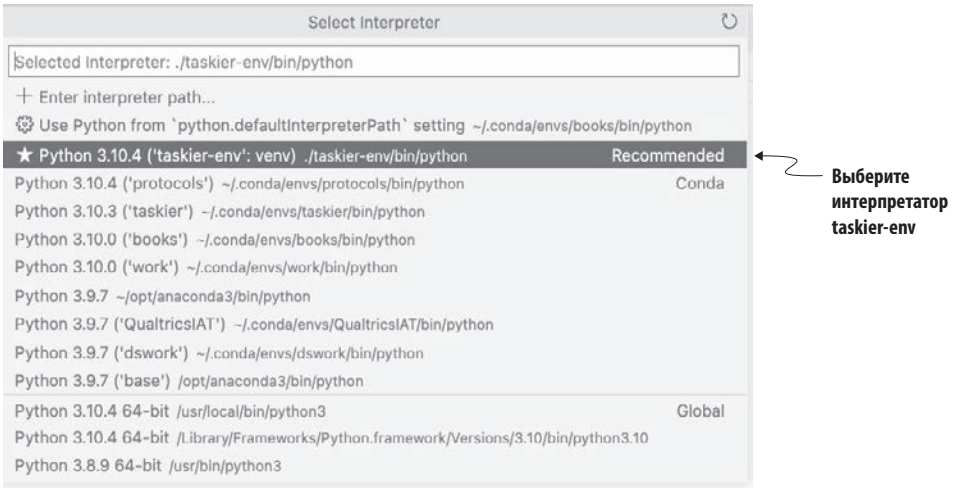


Рис. 14.2. Выбор интерпретатора для виртуальной среды. Учтите, что на вашем компьютере список вариантов может быть другим. На рисунке представлен полный список виртуальных сред, доступных на моем компьютере

Несмотря на то что ваш проект будет реализован с Python 3.10.4, он должен быть совместим с другими версиями (Python 3.8 и выше), так как рассматриваемые средства относятся к базовым функциональным возможностям Python.

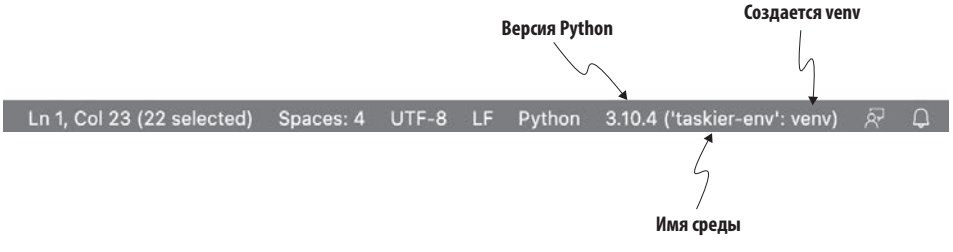


Рис. 14.3. Строка состояния с ключевой информацией о выполнении Python в VSC, включая версию Python, виртуальную среду и инструмент создания среды (venv)

14.1.5. Обсуждение

Модуль venv предоставляет основные средства для создания виртуальной среды, и им удобно пользоваться, потому что он входит в стандартную библиотеку Python. Однако у этого модуля есть недостаток: по умолчанию он использует

общесистемную копию Python. Если вы хотите использовать в своем проекте конкретную версию Python, нужны другие средства управления виртуальными средами, например conda. С conda вы будете пользоваться всеми преимуществами установки пакетов в конкретной среде, которые предоставляет venv. Но кроме этого, вы сможете создать изолированную установку Python в виртуальной среде, что обеспечит большую гибкость настройки конфигурации проекта с учетом версии Python и сторонних пакетов.

Использование conda для управления виртуальной средой

Чтобы иметь отдельный интерпретатор Python для вашего проекта, воспользуйтесь conda для управления виртуальными средами. За инструкциями по установке обращайтесь на официальный веб-сайт: https://conda.io. После того как установка conda будет завершена, вы сможете использовать ее для создания виртуальных сред в той командной оболочке, которую вы предпочитаете.

В нашем проекте будет использоваться Python 3.10.4 и зависимость streamlit 1.10.0. Для создания нужной виртуальной среды примените следующую команду:

```
conda create -n taskier-env python=3.10.4 streamlit=1.10.0
```

Учтите, что если выполнить этот код после создания виртуальной среды при помощи venv, в системе появятся две виртуальные среды с одинаковыми именами и разными путями. После выполнения команды вы можете активизировать среду командой conda activate taskier-env, а затем работать в этой виртуальной среде. Чтобы настроить среду в VSC, откройте список интерпретаторов Python и выберите один из них для среды taskier-env.

14.1.6. Задача

Джерри работает аналитиком данных в компании по недвижимости. Он знает, что для его проекта желательно создать отдельную виртуальную среду. Как ему создать виртуальную среду с именем python-env? В среде необходимо установить библиотеку pandas. После установки он также хочет настроить VSC для этой среды.

ПОДСКАЗКА Выполните инструкции, приведенные в этом разделе.

14.2. КАК ПОСТРОИТЬ МОДЕЛИ ДАННЫХ ДЛЯ ПРОЕКТА

Основой любого приложения являются данные, которые могут принимать различные формы, например текстовую или графическую. Вне зависимости от формы данных, при построении приложений мы обычно определяем

пользовательские классы для представления данных в виде атрибутов. Данные подготавливаются и обрабатываются функциями или методами внутри пользовательского класса. Эти данные и сопутствующие операции объединяются под общим названием *«модели данных»* для приложения. В этом разделе будут описаны модели данных, которые используются в таск-менеджере.

14.2.1. Выявление бизнес-целей

Модели данных должны служить бизнес-целям нашего проекта. Чтобы правильно построить модели данных приложения, необходимо сначала определиться с его функциональностью. Приложение было задумано как демонстрационный проект, поэтому я включил в него минимальную функциональность, которая служит основой для показа основных средств Python. Я не хотел излишне усложнять приложение, потому что это затруднило бы изучение этих основных средств.

В нашем приложении пользователь может создавать новые задачи, просматривать списки задач, редактировать задачи и удалять их. Также было бы удобно, если бы пользователи могли сортировать и фильтровать задачи по определенным критериям. Функциональные аспекты приложения представлены на рис. 14.4.

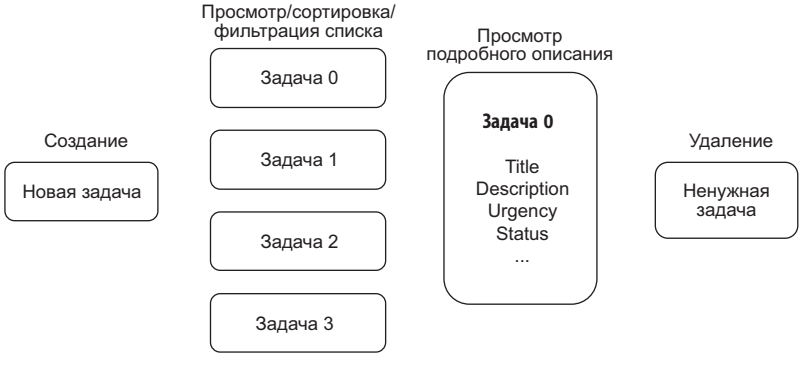


Рис. 14.4. Ключевые функции таск-менеджера. В приложении пользователи могут создавать новые задачи; просматривать, сортировать и фильтровать задачи, просматривать их подробные описания и удалять задачи

Как показано на рис. 14.4, каждая задача имеет ряд атрибутов: `title` (название), `desc` (описание), `urgency` (степень срочности) и `status` (статус). При построении реального приложения со значительно более широкой функциональностью необходимо спроектировать пользовательский интерфейс приложения (клиентскую часть) таким образом, чтобы вы могли определить, реализованы ли все нужные возможности, и выяснить, как они между собой взаимодействуют. В нашем таск-менеджере будет использоваться простой интерфейс, а основное внимание уделяется программному коду, а не проектированию интерфейса приложения.

ЗАБЕГАЯ ВПЕРЕД Мы построим веб-приложение, которое станет клиентской частью (фронтендом) нашего приложения. В ходе работы мы воспользуемся фреймворком `streamlit`, который помогает разместить элементы веб-приложения (такие, как текстовые поля и надписи).

14.2.2. Создание вспомогательных классов и функций

Прежде чем анализировать код класса `Task` (раздел 14.2.3), я хочу представить необходимые вспомогательные классы и функции. Мы создадим файл с именем taskier.py для хранения класса `Task`. В заголовке файла импортируются необходимые зависимости (заметьте, что использование этих модулей рассматривается при обсуждении соответствующего кода):

```
import csv
import re
import sqlite3
from enum import IntEnum, Enum
from pathlib import Path
from random import choice
from string import ascii_lowercase
```

Задача имеет три возможных статуса: созданная, текущая и завершенная. Для представления этих статусов будет использоваться перечисление:

```
class TaskStatus(IntEnum):
    CREATED = 0
    ONGOING = 1
    COMPLETED = 2
    @classmethod
    def formatted_options(cls):
        return [x.name.title() for x in cls]
```

В разделе 9.1 вы узнали о перечислениях, создаваемых с помощью подклассов `Enum`. Здесь создаются подклассы класса `IntEnum`, который похож на `Enum`, но обладает дополнительным преимуществом: статусы можно сортировать, так как их низкоуровневые значения представляют собой целые числа. В этом классе перечислений определяется метод класса (раздел 8.2), который создает список строк, используемый в нашем веб-приложении (раздел 14.4).

В разделе 11.2 рассматривалась обработка табличных данных с помощью модуля `csv`. Чтобы продемонстрировать некоторые полезные приемы, я возьму в качестве источника данных CSV-файл (хотя в реальных приложениях вместо CSV-файлов обычно используются базы данных). В разделе 14.3 будет показано, как работать с одной из распространенных СУБД (систем управления базами данных) SQLite. Чтобы включить в веб-приложение оба варианта, воспользуйтесь классом перечислений:

```
class TaskierDBOption(Enum):
    DB_CSV = "tasks.csv"
```

```
    DB_SQLITE = "tasks.sqlite"

app_db = TaskierDBOption.DB_CSV.value
```

Для хранения информации о базе данных создается глобальная переменная `app_db`. Далее по умолчанию назначается вариант с CSV-файлом. В веб-приложении в учебных целях мы позволим пользователю выбрать вариант с базой данных, а функция из листинга 14.1 обновит информацию при выборе базы данных.

Листинг 14.1. Настройка варианта с базой данных для приложения

```
def set_db_option(option):
    global app_db
    app_db = option
    db_path = Path(option)
    if not db_path.exists():  <-- Проверяет существование пути
        Task.load_seed_data()
    elif app_db == TaskierDBOption.DB_SQLITE.value:
        Task.con = sqlite3.connect(app_db)
```

Так как мы изменяем переменную в глобальной области видимости, перед изменением необходимо использовать ключевое слово `global` (раздел 10.4). Если файл не существует по заданному пути, мы создадим файл данных и в демонстрационных целях загрузим исходные данные при помощи метода `load_seed_data` класса `Task` (раздел 14.2.3). Хотя SQLite более подробно рассматривается в разделе 14.3, листинг 14.1 включает строку кода (`Task.con = sqlite3.connect(app_db)`), которая создает подключение к базе данных, если выбран вариант с БД SQLite.

Для обработки исключений мы создадим собственный класс исключения, чтобы в программе можно было выдавать пользовательские исключения. Как упоминалось в разделе 12.5, класс исключений является подклассом `Exception`:

```
class TaskierError(Exception):
    pass
```

Так как при использовании этого класса можно задать пользовательское сообщение об ошибке, предоставлять реализации этих методов не обязательно; команда `pass` нужна только для выполнения требований к синтаксису. Если потребуется предоставить более конкретные исключения, можно создать подклассы `TaskierError`.

14.2.3. Создание класса Task

Основная функциональность приложения определена. Все готово к реализации класса `Task`, который обеспечивает выполнение бизнес-целей. В этом разделе мы построим класс `Task`. Чтобы упростить изучение, я проанализирую код, останавливаясь на отдельных методах.

Создание и сохранение задач

В нашем приложении каждая задача моделируется экземпляром класса `Task`. В этом разделе приводится код создания и сохранения экземпляров в файле.

Метод инициализации позволяет определить пользовательские атрибуты для экземпляров (раздел 8.1). Метод `__init__` переопределяется для вмешательства в процесс создания экземпляра. В определении используются аннотации типов (раздел 6.3) для каждого аргумента. Также для метода предоставляются docстроки в стиле Google (раздел 6.5.1):

```
class Task:
    def __init__(self, task_id: str, title: str, desc: str, urgency:
    ➥ int, status=TaskStatus.CREATED, completion_note=""):
        """инициализирует экземпляр класса Task

        Аргументы:
            task_id (str): случайно сгенерированная строка-идентификатор
            title (str): название
            desc (str): описание
            urgency (int): степень срочности, 1 - 5
            status (_type_, optional): статус. По умолчанию равен
            ➥ TaskStatus.CREATED.
            completion_note (str, optional): заметка о завершении
            ➥ задачи. По умолчанию "".
        """
        self.task_id = task_id
        self.title = title
        self.desc = desc
        self.urgency = urgency
        self.status = TaskStatus(status)
        self.completion_note = completion_note
```

В форму вводится название, описание и степень срочности, после чего полученная информация используется для создания экземпляра класса `Task`. Как видно из следующего фрагмента, `task_from_form_entry` реализуется в виде метода класса, потому что нам не нужно обращаться к данным экземпляра или изменять их. Вместо этого метод вызывает конструктор класса:

```
@classmethod
def task_from_form_entry(cls, title: str, desc: str, urgency: int):
    """Создание задачи по данным формы

    Аргументы:
        title (str): название задачи
        desc (str): описание задачи
        urgency (int): степень срочности задачи (1 - 5)

    Возвращает:
        Task: экземпляр класса Task
    """
    task_id = cls.random_string()
    task = cls(task_id, title, desc, urgency)
    return task
```

ПРИМЕЧАНИЕ Я мог бы указать, что возвращаемым типом метода класса является `Self`, то есть ссылка на класс, но такая возможность появилась только в Python 3.11. Для совместимости с более ранними версиями Python я опустил аннотации типов для возвращаемого типа.

В этом методе класса вызывается метод `random_string` для получения случайной строки, используемой в качестве идентификатора новой задачи. Так как генерирование случайной строки может быть вспомогательной функцией для других целей, мы реализуем его в виде статического метода, ведь в нем не используются атрибуты, относящиеся к экземпляру:

```
@staticmethod
def random_string(length=8):
    """Создает случайную ASCII-строку заданной длины

    Аргументы:
        length (int, optional): требуемая длина случайной
        ➥ строки. По умолчанию 8.

    Возвращает:
        str: случайная строка
    """
    return "".join(choice(ascii_lowercase) for _ in range(length))
```

В этом методе в качестве источника используется набор символов ASCII нижнего регистра (импортированный из модуля `string`). Из него случайным образом выбираются восемь символов при помощи функции `choice` из модуля `random`, и эти символы объединяются методом `join` (раздел 2.3). Созданный экземпляр необходимо сохранить в базе данных, для чего воспользуемся методом `save_to_db` из листинга 14.2.

Листинг 14.2. Сохранение записи в базе данных

```
def save_to_db(self):
    """Сохранение записи в базе данных
    """
    if app_db == TaskierDBOption.DB_CSV.value:
        with open(app_db, "a", newline="") as file:
            csv_writer = csv.writer(file)
            db_record = self._formatted_db_record()
            csv_writer.writerow(db_record)
    else:
        # Операции для базы данных SQLite3
        pass

def _formatted_db_record(self):
    db_record = (self.task_id, self.title, self.desc, self.urgency,
    ➥ self.status.value, self.completion_note)
    return db_record
```

CSV-файл открывается в режиме присоединения с использованием инструкции `with` (раздел 11.1). Используя объект `writer` для CSV, мы записываем строку данных в CSV-файл. Обратите внимание на вызов защищенного метода `_formatted_db_record` для получения записи, которая будет записана в файл. Префикс `_` означает, что доступ к методу ограничен (раздел 8.3.1).

Чтение задач из источника данных

Когда в базе данных появляются записи нескольких задач, приходит время читать и выводить задачи. Для загрузки задач из базы данных создается метод `load_tasks`, приведенный в листинге 14.3.

Листинг 14.3. Загрузка задач из базы данных

```
@classmethod
def load_tasks(cls, statuses: list[TaskStatus]=None, urgencies:
➥ list[int]=None, content: str=""):
    """Загрузка задач, соответствующих заданному критерию

    Аргументы:
        statuses (list[TaskStatus], optional): фильтрует задачи
        ➥ с заданными статусами.
           По умолчанию None, то есть требований к статусу нет.
        urgencies (list[int], optional): фильтрует задачи
        ➥ с заданными степенями срочности.
            По умолчанию None, то есть требований к срочности нет.
        content (str, optional): фильтрует задачи
        ➥ с заданной информацией (title, desc или note).
            По умолчанию "".

    Возвращает:
        list[Task]: список задач, соответствующих заданному критерию
    """
    tasks = list()
    if app_db == TaskierDBOption.DB_CSV.value:
        with open(app_db, newline="") as file:
            reader = csv.reader(file)
            for row in reader:
                task_id, title, desc, urgency_str, status_str, note = row
                urgency = int(urgency_str)
                status = TaskStatus(int(status_str))
                if statuses and (status not in statuses):
                    continue
                if urgencies and (urgency not in urgencies):
                    continue
                if content and all([note.find(content) < 0,
                ➥ desc.find(content) < 0, title.find(content) < 0]):
                    continue
                task = cls(task_id, title, desc, urgency, status, note)
                tasks.append(task)
    else:
        # SQLite в качестве источника данных
        pass
    return tasks
```

Использует find для поиска подстроки

ПРИМЕЧАНИЕ Использование аннотации типа `list[TaskStatus]` поддерживается в Python 3.9 и более поздних версиях. Если вы столкнетесь с исключением, связанным с ее использованием, вероятно, у вас более старая версия Python.

В листинге 14.3 стоит обратить внимание на следующие обстоятельства:

- Созданный объект `reader` для чтения из CSV-файла используется как генератор (раздел 11.2), при этом каждый элемент представляет строку данных.
- Распаковка кортежа (раздел 4.4) используется для последовательного получения шести элементов данных. Каждый из этих элементов представляет собой строку.
- Для получения нужных атрибутов `urgency` и `status` используются конструкторы `int` и `TaskStatus` соответственно. Для получения данных также можно было воспользоваться инструкцией `try...with...`, но в данном случае мы уверены в целостности данных, так что преобразование должно сработать. При обработке внешних данных следует использовать средства обработки исключений.
- Для поиска подстроки используется функция `find`, так как она не выдает исключение в отличие от метода `index` (раздел 4.3.2).
- Встроенная функция `all` возвращает `True`, если все элементы списка интерпретируются как `True`. Вся строка означает, что если при вызове функции задается аргумент `content` и для него не находится совпадения в `note`, `desc` или `title`, мы пропускаем текущую запись при помощи команды `continue`.
- Так как наше приложение позволяет пользователям выбирать задачи, удовлетворяющие заданным критериям, — включая статусы, уровни срочности и содержимое (название, описание и заметка о завершении), — необходимо определить метод `load_tasks`, который способен загружать не только все задачи, но и их подмножества. Если аргумент `statuses` отличен от `None`, а статус текущей строки не содержится в `statuses`, текущая строка пропускается командой `continue`. Та же логика применяется к аргументам `urgencies` и `content`.

Обновление задачи в источнике данных

Если пользователь вносит изменения в задачу, необходимо обновить запись в базе данных. Для этой цели используется метод `update_in_db`, приведенный в листинге 14.4.

В этом методе я хотел продемонстрировать полезность регулярных выражений. В сущности, мы читаем все текстовые данные из CSV-файла. Схема выглядит так: мы ищем строку, которая начинается с идентификатора задачи и завершается разрывом строки. Она заменяется обновленной записью, полученной в результате

вызова метода `_formatted_db_`. Так как текстовые данные записываются в файл, данные отформатированной строки необходимо преобразовать в строку при помощи функции `map` (раздел 7.2.2).

Листинг 14.4. Обновление записи в базе данных

```
def update_in_db(self):
    """Обновление записи в базе данных
    """
    if app_db == TaskierDBOption.DB_CSV.value:
        updated_record = f"{','.join(map(str,
        ➥ self._formatted_db_record()))}\n"
        with open(app_db, "r+") as file:
            saved_records = file.read()
            pattern = re.compile(rf"{self.task_id}.+?\n")
            if re.search(pattern, saved_records):
                updated_records = re.sub(pattern,
                ➥ updated_record, saved_records)
                file.seek(0)
                file.truncate()
                file.write(updated_records)
            else:
                raise TaskierError("The task appears to be
                ➥ removed already!")
    else:
        # SQLite в качестве источника данных
        pass
```

Компилирует шаблон регулярного выражения

С точки зрения быстродействия обновленную запись можно заменить напрямую без проверки ее существования. Но ввиду особенности архитектуры приложения (раздел 14.4.3) иногда оказывается, что пользователь пытается обновить уже удаленную задачу. В такой ситуации программа должна выдавать исключение, если запись не существует.

Хотя у нас не было возможности обсудить методы `seek` и `truncate` в разделе 11.1, в них легко разобраться. В сущности, мы вызываем `seek(0)` для перемещения курсора файлового потока в начало, а затем вызываем `truncate` для удаления всех текстовых данных. После того как файл станет пустым, можно записать `updated_records` в файл.

Удаление задачи из источника данных

У пользователя должна быть возможность при желании удалить задачу. Для этого определите метод `delete_from_db`, как показано в листинге 14.5.

Листинг 14.5. Удаление записи из базы данных

```
def delete_from_db(self):
    """Удаление записи из базы данных
    """
    if app_db == TaskierDBOption.DB_CSV.value:
```

```
        with open(app_db, "r+") as file:
            lines = file.readlines()
            for line in lines:
                if line.startswith(self.task_id):
                    lines.remove(line)
                    break
            file.seek(0)
            file.truncate()
            file.writelines(lines)
    else:
        # SQLite в качестве источника данных
        pass
```

В этом методе вызывается метод `readlines` (раздел 11.1) для получения текстовых данных в виде объекта `list`. Мы используем этот метод, потому что объекты `list` являются изменяемыми (раздел 3.1), что позволяет нам удалять задачи. Для каждой строки мы проверяем, начинается ли она с заданного идентификатора задачи, и когда он будет найден, происходит немедленный выход из цикла `for` с помощью инструкции `break`. После того как объект `list` будет обновлен, его следует записать обратно в файл вызовом метода `writelines`.

Также определяется метод `load_seed_data` для загрузки исходных задач, чтобы приложение могло вывести некоторые данные. В этом методе мы создаем три задачи и сохраняем их в базе данных вызовом метода `save_to_db`:

```
@classmethod
def load_seed_data(cls):
    """Загрузка начальных данных для веб-приложения
    """
    task0 = cls.task_from_form_entry("Laundry", "Wash clothes", 3)
    task1 = cls.task_from_form_entry("Homework", "Math and physics", 5)
    task2 = cls.task_from_form_entry("Museum", "Egypt things", 4)
    for task in [task0, task1, task2]:
        task.save_to_db()
```

Наконец, для класса определяются методы строкового представления `__str__` и `__repr__` (раздел 8.4):

```
def __str__(self) -> str:
    stars = "\u2605" * self.urgency
    return f"{self.title} ({self.desc}) {stars}"

def __repr__(self) -> str:
    return f"{self.__class__.__name__}({self.task_id!r},
    ➥ {self.title!r}, {self.desc!r}, {self.urgency},
    ➥ {self.status}, {self.completion_note!r})"
```

ПРИМЕЧАНИЕ Если вас заинтересовало, в чем разница между этими двумя методами, вот ответ: `_str_` используется для информационных целей, а `_repr_` — для целей разработки.

14.2.4. Обсуждение

Наши модели данных должны служить бизнес-целям. Важно определить функциональность приложения до того, как переходить к реализации моделей данных. Хотя я привожу в тексте итоговую версию кода, мне понадобилось значительное время и несколько итераций кода, чтобы прийти к этой версии. Работа над любым проектом требует терпения.

ЗАБЕГАЯ ВПЕРЕД Класс `Task` предназначен для веб-приложения, которое будет построено в разделе 14.4.

14.2.5. Задача

Во время чтения этой книги Кэти пишет весь код для изучения рассматриваемых тем. Работая над классом `Task`, она предполагает, что пользователь может попытаться удалить задачу, уже удаленную из базы данных. Как ей обновить метод `delete_from_db`, чтобы выдавалось исключение, если удаляемая запись уже не существует?

ПОДСКАЗКА Прежде чем выполнять нужную операцию, следует проверить, присутствует ли запись в источнике данных.

14.3. КАК ИСПОЛЬЗОВАТЬ В ПРИЛОЖЕНИИ БАЗУ ДАННЫХ SQLITE

База данных (БД) отвечает за управление данными в вашем приложении. В зависимости от специфики приложения (например, объема данных и требований к обработке) существуют различные варианты баз данных[1]: Microsoft SQL, Oracle, MySQL, PostgreSQL и многие другие. Эти БД обычно ориентированы на приложения корпоративного уровня, а развертывание их инфраструктуры и поддержание быстродействия требует времени и ресурсов. В отличие от упомянутых решений корпоративного уровня, SQLite представляет собой облегченную базу данных, которая практически не требует специальной подготовки на вашем компьютере, так как в качестве механизма хранения данных используется его диск. В этом разделе я покажу, как использовать SQLite в качестве базы данных вашего приложения.

14.3.1. Создание базы данных

База данных SQL создается практически мгновенно и требует лишь нескольких вызовов функций. Мы воспользуемся встроенным модулем `sqlite3`, входящим

[1] Более точно — систем управления базами данных (СУБД). — *Примеч. ред.*

в стандартную библиотеку Python. Этот модуль предоставляет все программные интерфейсы (API), необходимые для создания и выполнения операций с базой данных SQLite. Начнем с создания базы данных.

Так как БД совместно используется всеми экземплярами класса `Task`, мы определим подключение к ней как атрибут класса. Через это подключение будут выполняться все операции, связанные с базой данных, включая запросы на выборку и обновление данных. Мы не работаем с БД напрямую на физическом уровне, чтобы другие процессы могли использовать ее по мере необходимости. Таким образом, мы устанавливаем подключение и работаем с ним (по аналогии с работой с файлом через объект файла вместо низкоуровневых операций):

```
class Task:
    con: sqlite3.Connection
```

Для создания базы данных определяется метод `create_sqlite_database`:

```
@classmethod
def create_sqlite_database(cls):
    """Создание базы данных SQLite
    """
    with sqlite3.connect(TaskierDBOption.DB_SQLITE.value) as con:
        cls.con = con  <-- Сохраняется в переменной класса
        cursor = con.cursor()
        cursor.execute("CREATE TABLE task (task_id text, title text,
        ➥ desc text, urgency integer, status integer, completion_note text);")
```

В этом методе выполняются две операции:

- *Вызов функции* `connect` *создает подключение к базе данных по заданному пути*. Если базы данных с заданным путем не существует, функция создает ее. Мы используем инструкцию `with`, которая создает менеджер контекста для автоматического завершения работы с базой.
- *В базу данных добавляется новая таблица* `task`. Этот код выполняется только в том случае, если базы данных не существует. Для создания используется команда `CREATE TABLE имя_таблицы (поле0_имя поле0_тип, поле1_имя поле1_тип, ...)`. Также стоит заметить, что для выполнения команды создается курсор — это стандартная операция для SQLite и баз данных SQL вообще.

Метод `create_sqlite_database` должен вызываться в том случае, если пользователь выбирает вариант с базой данных, поэтому необходимо обновить функцию `set_db_option` из листинга 14.1 и привести ее к следующему виду:

```
def set_db_option(option):
    global app_db
    app_db = option
    db_path = Path(option)
```

```
if not db_path.exists():
    if app_db == TaskierDBOption.DB_SQLITE.value:
        Task.create_sqlite_database()
    Task.load_seed_data()
elif app_db == TaskierDBOption.DB_SQLITE.value:
    Task.con = sqlite3.connect(app_db)
```

Добавленный код выделен жирным шрифтом; это простой вызов метода `create_sqlite_database` в том случае, если базы данных не существует. По поводу части `elif`: если БД SQLite существует и выбирается вариант с SQLite, создается подключение к базе данных. Прежде чем переходить к коду выполнения операций данных с БД SQLite, взгляните на рис. 14.5, где представлены самые распространенные операции с базами данных.

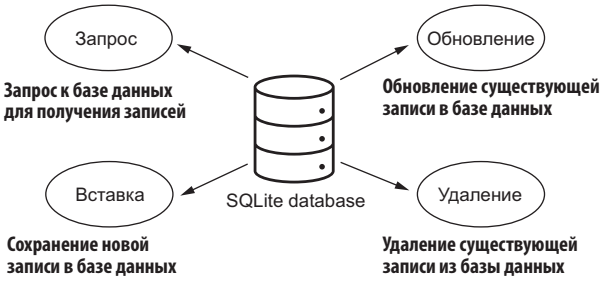


Рис. 14.5. Основные операции с базой данных SQLite: запрос, вставка, обновление и удаление. Запросы читают записи из базы данных, вставка сохраняет новую запись в базе данных, обновление изменяет существующую запись, а удаление уничтожает существующую

Как показано на рис. 14.5, при использовании базы данных SQLite (и вообще любой БД) выполняются четыре основные операции: запрос (чтение записей из базы данных), вставка (сохранение новой записи), обновление (обновление существующей записи) и удаление (удаление записи из базы данных). В следующих разделах эти четыре операции рассматриваются по отдельности.

14.3.2. Чтение записей из базы данных

Чтобы вывести данные в приложении, необходимо прочитать записи из базы данных. Вы уже видели, как использовать модуль `csv` для чтения данных из CSV-файла (раздел 14.2.3). В этом разделе я покажу, как получить данные из базы данных SQLite.

Для получения данных по задачам определяется метод `load_tasks` (листинг 14.3). Теперь мы обновим этот метод, чтобы он работал с БД SQLite (листинг 14.6). Я привожу только код, относящийся к чтению данных из БД SQLite, и опускаю код для использования CSV-файла.

Листинг 14.6. Загрузка данных из БД SQLite

```
@classmethod
def load_tasks(cls, statuses: list[TaskStatus]=None,
➥ urgencies: list[int]=None, content: str=""):
    """Doc-строка остается неизменной
    """
    tasks = list()
    if app_db == TaskierDBOption.DB_CSV.value:
        # Код, относящийся к csv, из листинга 14.3
        pass
    else:
        with cls.con as con:
            if statuses is None:
                statuses = tuple(map(int, TaskStatus))
            else:
                statuses = tuple(statuses) * 2
            if urgencies is None:
                urgencies = tuple(range(1, 6))
            else:
                urgencies = tuple(urgencies) * 2
            sql_stmt = f"SELECT * FROM task WHERE status in {statuses}
            ➥ and urgency in {urgencies}"
            if content:
                sql_stmt += f" and ((completion_note LIKE '%{content}%')
                ➥ or (desc LIKE '%{content}%') or (title LIKE
                ➥ '%{content}%'))"
            cursor = con.cursor()
            cursor.execute(sql_stmt)
            tasks_tuple = cursor.fetchall()
            tasks = [Task(*x) for x in tasks_tuple]
    return tasks
```

Обратите внимание на следующие моменты в этом коде:

- Так как я хочу создать одну инструкцию SQL для двух сценариев — всех задач (без критериев фильтрации) и подмножества задач (с критериями фильтрации), — я перечисляю все статусы, когда аргумент `statuses` равен `None`, командой `statuses = tuple(map(int, TaskStatus))`.
- Эта же логика применима к аргументу `urgencies`. Если пользователь хочет получить все задачи, мы требуем, чтобы значение поля `urgency` записей находилось в диапазоне 1–5, то есть допустимом диапазоне степеней срочности.
- В этом коде есть один неочевидный момент: если значения `statuses` и `urgencies` отличны от `None`, используются выражения `tuple(statuses) * 2` и `tuple(urgencies) * 2`. Это сделано для того, чтобы выполнить требования к синтаксису инструкции SQL, когда пользователь выбирает только один элемент для `status` или `urgency`. Точнее, если пользователи задают только степень срочности (например, 2), на основе этого ввода будет создан объект `tuple` с одним элементом `(2,)`. Напрямую использовать этот объект `tuple` в `sql_stmt` невозможно, поэтому мы дублируем элементы в объекты `tuple`,

превращая `(2,)` в `(2,2)`; таким образом мы получаем действительную инструкцию SQL.

- Операция `LIKE` в синтаксисе SQL предназначена для получения записей, соответствующих заданной подстроке. Значение `sql_stmt` обновляется только при заданном аргументе `content`. Часть `content` интерпретируется как `True`, если строка не пуста.
- Функция `fetchall` получает все записи в виде объекта `list` по результатам выполненной инструкции SQL. Каждая запись возвращается в виде объекта `tuple` в форме (`task_id`, `title`, `desc`, `urgency`, `status`, `completion_note`). Используя списковое включение, мы преобразуем объекты `tuple` в экземпляры `Task`.
- В процессе преобразования объекта `tuple` в экземпляр используется операция *, которая распаковывает объект `tuple` и передает элементы конструктору.

14.3.3. Сохранение записей в базе данных

Когда вы создаете новую запись, ее необходимо сохранить в базе данных. Записи можно сохранять поочередно или же все сразу. В этом разделе я продемонстрирую оба способа.

В листинге 14.2 метод `save_to_db` определяется для CSV-файла как источника данных. Мы обновим этот метод, чтобы сделать его совместимым с базой данных SQLite (листинг 14.7).

Листинг 14.7. Сохранение записи в БД SQLite

```
def save_to_db(self):
    """Сохранение записи в базе данных
    """
    if app_db == TaskierDBOption.DB_CSV.value:
        # operations when the database is the CSV file
        pass
    else:
        with self.con as con:
            cursor = con.cursor()
            sql_stmt = f"INSERT INTO task VALUES (?, ?, ?, ?, ?, ?);"
            cursor.execute(sql_stmt, self._formatted_db_record())
```

Синтаксис сохранения записей в базе данных SQLite — `INSERT INTO table VALUES (?, ?, ...)`. Вопросительный знак — это заполнитель, а количество заполнителей (шесть в данном случае) должно соответствовать количеству элементов записи, полученной вызовом `_formatted_db_record`. Учтите, что инструкцию можно выполнить и без заполнителей, как было сделано в листинге 14.6. Если вы используете заполнители, их значения передаются во втором аргументе вызова функции `execute`.

Также обратите внимание на вызов `self.con` для подключения к базе данных. Хотя `con` определяется как атрибут класса, при обращении к атрибуту `con` экземпляра он используется в качестве резервного значения.

Что делать, если вы хотите сохранить одной инструкцией SQL сразу несколько записей? Такая возможность существует. Вместо вызова `execute` вызывается функция `executemany`. При вызове функции во втором аргументе передается список записей. Хотя мы не будем реализовывать эту возможность в классе `Task` (метода экземпляра `save_to_db` достаточно для учебных целей), в листинге 14.8 показано, как сохранить несколько записей в базе данных SQLite.

Листинг 14.8. Сохранение нескольких записей в БД SQLite

```
task0 = Task.task_from_form_entry("Laundry", "Wash clothes", 3)
task1 = Task.task_from_form_entry("Homework", "Math and physics", 5)
task2 = Task.task_from_form_entry("Museum", "Egypt things", 4)

with Task.con as con:
    cursor = con.cursor()
    tasks = [task0, task1, task2]
    formatted_records = [task._formatted_db_record() for task in tasks]
    sql_stmt = f"INSERT INTO task VALUES (?, ?, ?, ?, ?, ?);"
    cursor.executemany(sql_stmt, formatted_records)
```

14.3.4. Обновление записи в базе данных

Наш таск-менеджер поддерживает возможность редактирования задач. После редактирования задачи необходимо обновить запись в базе данных. В этом разделе вы узнаете, как обновить запись в базе данных SQLite.

За обновление записей отвечает метод `update_in_db`. В следующем фрагменте приведена обновленная версия метода, в которой добавлен новый код в часть, связанную с БД SQLite:

```
def update_in_db(self):
    """Обновление записи в базе данных
    """
    if app_db == TaskierDBOption.DB_CSV.value:
        # Операции для CSV-файла
        pass
    else:
        with self.con as con:
            cursor = con.cursor()
            count_sql = f"SELECT COUNT(*) FROM task WHERE
            ➥ task_id = {self.task_id!r}"
            row_count = cursor.execute(count_sql).fetchone()[0]
            if row_count > 0:
                sql_stmt = f"UPDATE task SET task_id = ?, title = ?,
                ➥ desc = ?, urgency = ?, status = ?, completion_note = ?
                ➥ WHERE task_id = {self.task_id!r}"
                cursor.execute(sql_stmt, self._formatted_db_record())
            else:
                raise TaskierError("The task appears to be
                ➥ removed already!")
```

Подсчитывает существующие записи

Обратите внимание: мы сначала проверяем количество записей с соответствующим идентификатором задачи (оно должно быть равно 1, то есть это положительное число). Если запись была удалена, выдается исключение с информацией об этом факте, как это было сделано в разделе 14.4 при реализации того же метода с CSV-файлом в качестве источника данных.

Синтаксис обновления записей в базе данных SQLite — `UPDATE таблица SET поле0_имя = ?, поле1_имя = ?, ... WHERE условие`. В этом синтаксисе обязательно должна присутствовать секция `WHERE`, которая фильтрует записи; если вы забудете о ней, произойдет непреднамеренное обновление всех записей. Как и в предыдущих случаях, при вызове функции `execute` используются заполнители. В секции идентификатор `task_id` указывается с использованием преобразования `!r`, в результате идентификатор заключается в одинарные кавычки (`'example_id'`) вместо `example_id`.

14.3.5. Удаление записи из базы данных

В нашем таск-менеджере пользователь может удалить существующую задачу. При этом соответствующую запись необходимо удалить из базы данных. В разделе ниже вы узнаете, как это делается.

За удаление записей отвечает метод `delete_from_db`. В следующем фрагменте в метод добавляется новый код для части, относящейся к базе данных SQLite:

```
def delete_from_db(self):
    """Удаление записи из базы данных
    """
    if app_db == TaskierDBOption.DB_CSV.value:
        # Операции для CSV-файла
        pass
    else:
        with self.con as con:
            cursor = con.cursor()
            cursor.execute(f"DELETE FROM task WHERE task_id =
            ➥ {self.task_id!r}")
```

Синтаксис удаления записей в базах данных SQLite — `DELETE FROM таблица WHERE условие`. Единственное, что заслуживает особого упоминания, — применение `!r` к идентификатору задачи для создания строки, заключенной в одинарные кавычки.

14.3.6. Обсуждение

Так как SQLite представляет собой облегченную базу данных с минимальной конфигурацией, ее удобно использовать при построении прототипа приложения. Когда приложение переходит в фазу реальной эксплуатации, SQLite заменяется более мощной базой данных, например Oracle или MySQL. В примере

я сосредоточился на тексте и целых числах как типах данных, достаточных для наших бизнес-целей. Но надо иметь в виду, что у SQLite имеются свои ограничения. Прежде всего, в SQLite поддерживаются не все типы данных — в частности, это касается дат или логических значений. В качестве обходного решения можно использовать строки в формате ММДДГГ-ЧЧММСС, количество секунд от эталонной даты для дат и целые числа 0 и 1 для `false` и `true`.

14.3.7. Задача

Ранее было показано, что в качестве источника данных можно использовать CSV-файлы и базу данных SQLite. Сможете ли вы написать декоратор для регистрации времени, необходимого для вызова метода? С таким декоратором вы сможете сравнить, какой вариант для операций с данными быстрее — CSV-файл или БД SQLite.

ПОДСКАЗКА Создание декораторов рассматривается в разделе 7.3.

14.4. КАК ПОСТРОИТЬ ВЕБ-ПРИЛОЖЕНИЕ ДЛЯ ВЗАИМОДЕЙСТВИЯ С КЛИЕНТОМ (ФРОНТЕНД)

Веб-приложения — популярное решение для многих программных проектов. Самым важным их преимуществом является кросс-платформенная совместимость. Они могут выполняться в любом веб-браузере, а следовательно, доступны на любом компьютере, смартфоне и даже современном телевизоре с веб-браузером. Кроме того, веб-приложения не требуют установки или конфигурации на стороне клиента, потому что они работают в браузере, а вся функциональность такого приложения загружается в виде веб-элементов.

Очевидно, веб-приложения становятся самым привлекательным решением для многих бизнес-операций. В этом разделе я покажу, как построить веб-приложение с использованием `streamlit`, стороннего фреймворка Python для веб-разработки. Отмечу, что этот фреймворк предоставляет широкий диапазон функциональности, но я не стану приводить подробное руководство по его использованию. Вместо этого мы сосредоточимся на реализации функциональности таск-менеджера в форме веб-приложения.

14.4.1. Основные возможности streamlit

Чтобы использовать `streamlit` для создания веб-приложений, необходимо хорошо знать возможности этого фреймворка. В этом разделе я приведу основные сведения о нем.

После установки `streamlit` в виртуальной среде (`taskier-env`; раздел 14.1) появляется не только возможность использования фреймворка в файлах Python,

но и функциональность командной строки. Иначе говоря, средства командной строки могут использоваться как интерфейс для выполнения операций управления веб-приложениями, построенными на базе `streamlit`. Самая важная команда — `streamlit run taskier_app.py`. Как следует из имени, команда запускает веб-приложение из исходного файла taskier_app.py в основном браузере устройства.

Важнейшая возможность `streamlit` — преобразование файла скрипта Python в веб-приложение. Это главная причина, по которой `streamlit` считается популярным веб-фреймворком для разработчиков Python. Если вы знаете Python, то сможете использовать `streamlit` для построения веб-приложений.

Другая важная особенность `streamlit` — автоматическое размещение веб-элементов (кнопок, текстовых полей и т. д.). Фреймворк предоставляет ряд встроенных стандартных веб-элементов (виджетов). На рис. 14.6 изображены доступные виджеты, реализованные во фреймворке. Учтите, что в новой версии фреймворка `streamlit` эти виджеты могут измениться.

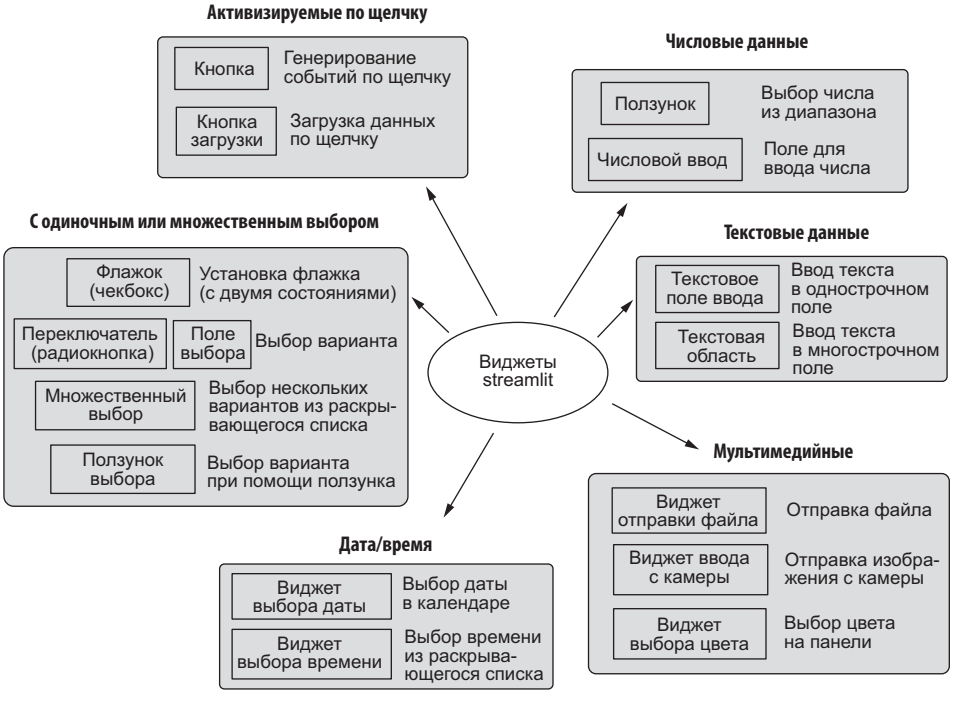


Рис. 14.6. Виджеты, доступные во фреймворке streamlit, делятся на шесть категорий: кликабельные, то есть активизируемые по щелчку (нажатию кнопки); с одиночным или множественным выбором; числовые данные; текстовые данные; мультимедийные; виджеты даты/времени

Я не стану рассказывать, как применять эти виджеты, потому что они достаточно просты; кроме того, инструкции можно найти по адресу https://streamlit.io/. Некоторые снимки экрана приведены в разделе 14.4.2. Вы увидите, что при использовании виджетов в скрипте указывается тип виджета с необходимыми настройками (например, выводимым на кнопках текстом), а вся черная работа по размещению элементов поручается фреймворку.

Еще одна заслуживающая внимания возможность фреймворка `streamlit` — линейная перезагрузка всего скрипта (сверху вниз) при каких-либо изменениях во вводе, например выборе пользователем одного из вариантов в виджете-переключателе. Эта особенность лежит в основе модели исполнения фреймворка. Она может привести в замешательство некоторых начинающих пользователей, потому что их опыт работы с веб-приложениями говорит о том, что страница не перезагружается автоматически, когда они щелкают кнопкой мыши по одному из вариантов переключателя. Хотя в некоторых ситуациях это может стать недостатком, для решения этой проблемы существует обходной путь: состояние сеанса (раздел 14.4.3).

14.4.2. Интерфейс приложения

Прежде чем приводить код для создания веб-приложения, необходимо разобраться, как выглядит само приложение. В этом разделе описан его интерфейс.

На первой странице выводится список задач (рис. 14.7). В левой части расположена боковая панель с командами меню (например, вывод задач или выбор

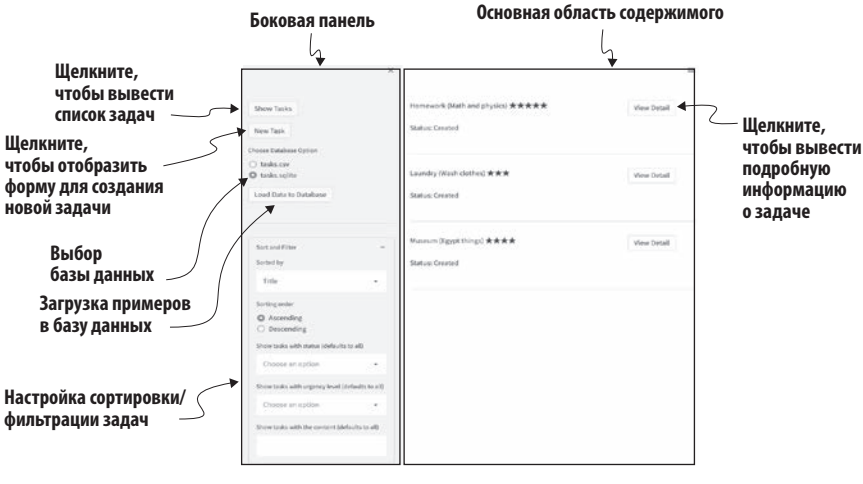


Рис. 14.7. Интерфейс для вывода списка задач. Основной интерфейс включает боковую панель с меню. В основной области содержимого выводится информация о задачах

базы данных как источника данных). Справа находится основная область содержимого. В данном случае содержимым является список задач. Пользователь выбирает способ сортировки и фильтрации списка задач на боковой панели. Для ясности меню сортировки/фильтрации показывается только при выводе списка задач.

Подробную информацию о каждой задаче из списка можно вывести, щелкнув по кнопке View Detail (рис. 14.8). Слева размещаются виджеты, с помощью которых пользователь удаляет задачу. Справа в области основного содержимого выводится подробная информация о задаче, которая включает кнопку Update Task для сохранения обновленной задачи в базе данных.

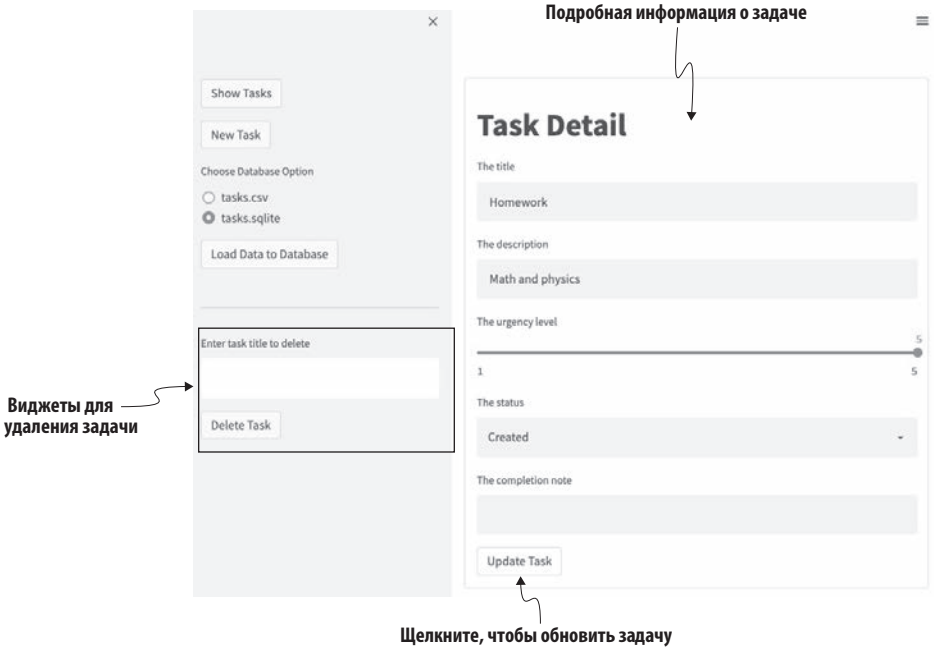


Рис. 14.8. Интерфейс для вывода подробной информации о задаче. На боковой панели размещаются виджеты, с помощью которых пользователь может удалить задачу. В основной области содержимого выводится подробная информация о задаче

Если пользователь щелкает на кнопке New Task на боковой панели, он перенаправляется на форму создания новой задачи (рис. 14.9). В основной области содержимого отображается форма, в которой пользователь вводит данные, необходимые для новой задачи. Кнопка Save Task сохраняет запись в базе данных.

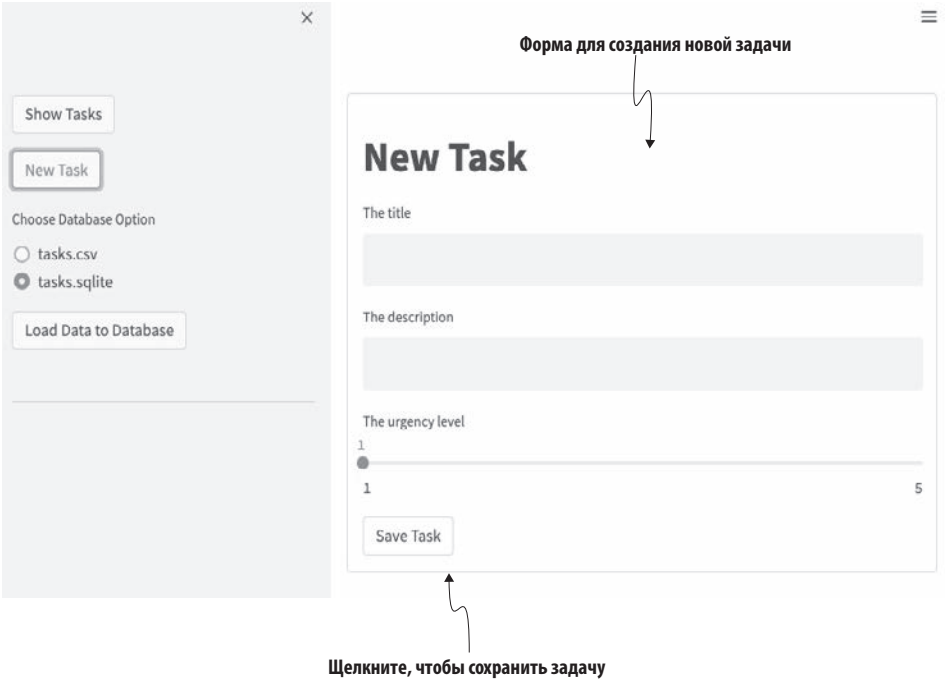


Рис. 14.9. Интерфейс создания новой задачи. После ввода данных пользователь щелкает на кнопке Save Task, чтобы сохранить задачу в базе данных

14.4.3. Отслеживание действий пользователя в течение сеанса

Чтобы упростить дальнейшее обсуждение `streamlit`, я введу неформальное определение *сеанса* как обращения пользователя к веб-приложению в браузере, обычно на вкладке современного браузера. Пока вкладка остается активной без обновления, мы используем состояние сеанса для отслеживания пользовательской активности, сохраняемой в форме пар «ключ — значение». В этом разделе показано, какие данные имеет смысл отслеживать в приложении.

Мы создадим файл taskier_app.py для сценария нашего веб-приложения, и весь код, рассматриваемый в разделе ниже, будет направляться в этот файл (если в тексте не указано иное). В начало файла мы импортируем зависимости. Об этих зависимостях мы поговорим, когда они станут актуальными в контексте приведенного кода, а пока сосредоточимся на `streamlit`. Для `streamlit` обычно используется псевдоним `st`, чтобы было удобнее обращаться

к фреймворку. Например, для получения данных сеанса используется вызов `st.session_state`:

```
import copy
import streamlit as st
from taskier import Task, TaskierDBOption, set_db_option,
➥ TaskStatus, TaskierError
from taskier_app_helper import TaskierMenuOption, TaskierFilterKey

session = st.session_state
sidebar = st.sidebar
status_options = TaskStatus.formatted_options()
menu_key = "selected_menu_option"
working_task_key = "working_task"
sorting_params_key = "sorting_params"
sorting_orders = ["Ascending", "Descending"]
sorting_keys = {"Title": "title", "Description": "desc", "Urgency":
➥ "urgency", "Status": "status", "Note": "completion_note"}
```

Кроме зависимостей, этот код включает переменные, к которым мы часто обращаемся в приложении. Все эти переменные относятся либо к настройке боковой панели, либо к состоянию сеанса.

Первый элемент, который нужно отслеживать в сеансе, — выбранный пункт меню. В приложении должны отображаться три страницы (раздел 14.4.2): список задач, подробная информация о задаче и форма для создания новой задачи. Так как в состоянии сеанса данные хранятся в форме пар «ключ — значение», в этом элементе (ему будет назначен ключ `selected_menu_option`) будет храниться одна из трех команд меню, реализованная в виде элемента перечисления из файла taskier_app_helper.py:

```
from enum import Enum

class TaskierMenuOption(Enum):
    SHOW_TASKS = "Show Tasks"
    NEW_TASK = "New Task"
    SHOW_TASK_DETAIL = "Show Task Detail"

class TaskierFilterKey(Enum):
    SORTING_KEY = "sorting_key"
    SORTING_ORDER = "sorting_order"
    SELECTED_STATUSES = "selected_statuses"
    SELECTED_URGENCIES = "selected_urgencies"
    SELECTED_CONTENT = "selected_content"
```

Вероятно, вы заметили, что класс `TaskierFilterKey` определяется во вспомогательном файле. Этот класс связан со вторым элементом, который будет отслеживаться в состоянии сеанса: выбранном пользователем способе сортировки и фильтрации списка задач. Например, пользователь может просматривать только задачи со степенью срочности 3. Параметры сортировки и фильтрации сохраняются в объекте `dict` с ключом `sorting_params` в состоянии сеанса.

ПРИМЕЧАНИЕ Мы могли бы взять два объекта `dict` для отслеживания параметров сортировки и фильтрации по отдельности. Но многие веб-приложения, в том числе и наше, имеют один пользовательский интерфейс для фильтрации и сортировки. Для нас будет проще использовать один объект `dict` для отслеживания параметров, сгенерированных из одного пользовательского интерфейса. Если в тексте не указано иное, я буду считать параметры сортировки и фильтрации синонимами.

Когда пользователь хочет просмотреть подробную информацию о задаче, нам необходимо отследить, к какой именно задаче он обратился. В состоянии сеанса ключ `working_task` используется для хранения этой задачи, представленной экземпляром класса `Task`. Так как во многих сеансовых функциях требуется обновлять несколько пар «ключ — значение», для этой задачи удобно определить функцию в файле taskier_app.py:

```
def update_session_tracking(key, value):
    session[key] = value
```

Функция `update_session_tracking` используется для обновления значений, связанных с соответствующим ключом. Вспомните, что `streamlit` выполняет весь сценарий сверху вниз при любых изменениях в пользовательском вводе. Таким образом, мы присвоим ключам их исходные значения только в том случае, если эти ключи еще не существуют в сеансе. Если значения с заданными ключами уже существуют, то значения, которые будут использоваться для отслеживания пользовательской активности, не должны переопределяться. Следующий фрагмент кода показывает, как задать исходное состояние сеанса:

```
def init_session():
    if menu_key not in session:
        update_session_tracking(menu_key,
        ➥ TaskierMenuOption.SHOW_TASKS.value)
        update_session_tracking(working_task_key, None)
        update_session_tracking(sorting_params_key, {x.value: None for x
        ➥ in TaskierFilterKey})
```

Так как `streamlit` используется для выполнения файла в форме сценария, рекомендуется применять конструкцию `if __name__ == "__main__"` в конце файла на случай, если вы захотите использовать файл в форме модуля, как показано в листинге 14.9.

Листинг 14.9. Вызов функций для создания веб-приложения

```
if __name__ == "__main__":
    init_session()  ◄——  Запускает сеанс
    setup_sidebar()
    if session[menu_key] == TaskierMenuOption.SHOW_TASKS.value:
        show_tasks()
    elif session[menu_key] == TaskierMenuOption.NEW_TASK.value:
```

```
        show_new_task_entry()
    elif session[menu_key] == TaskierMenuOption.SHOW_TASK_DETAIL.value:
        show_task_detail()
    else:
        st.write("No matching menu")
```

Как видно из листинга 14.9, мы вызываем функцию `init_session`, которая подготавливает состояние сеанса к отслеживанию активности пользователя. Затем вызывается функция `setup_sidebar`, рассмотренная в разделе 14.4.4.

14.4.4. Настройка боковой панели

Обычно боковая панель используется для вывода меню или дополнительных параметров конфигурации. В этом разделе вы узнаете, как настроить боковую панель для нашего приложения. Боковая панель настраивается вызовом функции `setup_sidebar`, как показано в листинге 14.10.

Листинг 14.10. Настройка боковой панели

```
def setup_sidebar():
    sidebar.button("Show Tasks", on_click=update_session_tracking,
    ➥ args=(menu_key, TaskierMenuOption.SHOW_TASKS.value))  <-- Добавляет кнопку

    sidebar.button("New Task", on_click=update_session_tracking,
    ➥ args=(menu_key, TaskierMenuOption.NEW_TASK.value))

    selected_db = sidebar.radio("Choose Database Option", [x.value for x
    ➥ in TaskierDBOption])  <-- Добавляет переключатель
    set_db_option(selected_db)

    sidebar.button("Load Data to Database", on_click=Task.load_seed_data)

    sidebar.markdown("___")  <-- Добавляет разделитель

    if session[menu_key] == TaskierMenuOption.SHOW_TASKS.value:
        setup_filters()
    elif session[menu_key] == TaskierMenuOption.SHOW_TASK_DETAIL.value:
        setup_deletion()
```

ОСНОВНЫЕ ПОНЯТИЯ Markdown — упрощенный язык разметки для создания отформатированного текста. В наших примерах тройное подчеркивание `___` преобразуется в виджет, обеспечивающий визуальное разделение секций меню.

В листинге 14.10 в веб-приложение впервые добавляются виджеты. В общем случае для добавления виджетов используется следующий синтаксис: `st.widget_name(метка_виджета, значение_или_опции, key=идентификатор_виджета, on_click=on_click_если_актуально, args=аргументы_если_есть)`. Для боковой панели можно использовать `sidebar.имя_виджета`. На рис. 14.10 изображена структура кода на примере виджетов кнопки и переключателя.

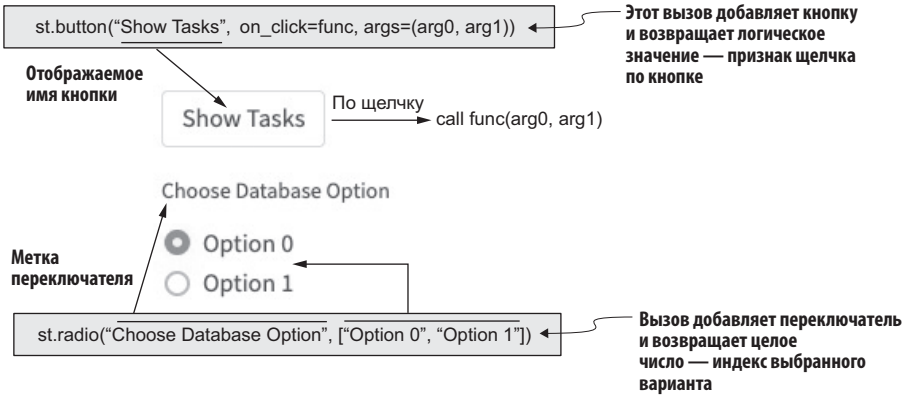


Рис. 14.10. Код добавления кнопки и переключения в streamlit. Вызов st.button добавляет кнопку на веб-страницу и возвращает логическое значение, определяющее состояние щелчка по кнопке. Вызов st.radio добавляет переключатель на веб-страницу и возвращает целое число, определяющее индекс выбранного варианта. Каждая функция включает дополнительные аргументы для настройки виджетов

При добавлении виджета (например, переключателя, как на рис. 14.10) используется возвращаемое значение вызова функции. Например, `st.radio` добавляет переключатель, и когда пользователь выбирает один из вариантов, из вызова функции можно узнать индекс выбранного варианта в группе. В нашем случае по индексу мы узнаем, какой из вариантов источника данных был выбран, — для этого следует вызвать функцию `set_db_option` (листинг 14.1). Если выбрана база данных, то она готовится к использованию — создается база данных SQLite, в которую добавляется таблица для задач. Чтобы с приложением было удобнее работать в процессе обучения, я сделал кнопку Load Data to Database для добавления записей в базу данных.

Когда пользователь выбирает режим просмотра задач, мы выводим виджеты для сортировки и фильтрации вызовом функции `setup_filters`. Возможно, у вас возник вопрос, не стоит ли сделать эту функцию приватной (ведь мы пишем сценарий для разработчиков, а не для других пользователей)? Функции можно присвоить имя без префикса _, который затрудняет чтение кода:

```
def setup_filters():
    filter_params = session[sorting_params_key]
    with sidebar.expander("Sort and Filter", expanded=True):
        filter_params[TaskierFilterKey.SORTING_KEY.value] =
        ➥ st.selectbox("Sorted by", sorting_keys)
        filter_params[TaskierFilterKey.SORTING_ORDER.value] =
        ➥ st.radio("Sorting order", sorting_orders)
        filter_params[TaskierFilterKey.SELECTED_STATUSES.value] =
        ➥ st.multiselect("Show tasks with status (defaults to all)",
```

```
    ➥options=status_options)
    filter_params[TaskierFilterKey.SELECTED_URGENCIES.value] =
    ➥ st.multiselect("Show tasks with urgency level (defaults to all)",
    ➥ options=range(1, 6))
    filter_params[TaskierFilterKey.SELECTED_CONTENT.value] =
    ➥ st.text_input("Show tasks with the content (defaults to all)")
```

Так как параметры сортировки и фильтрации принадлежат к одной концептуальной категории, я использовал виджет `expander` с именем `Sort and Filter`. Внутри него определяются пять виджетов: `selectbox` для выбора одного из атрибутов задачи (название, описание, степень срочности, статус или заметка о завершении) для сортировки; переключатель (`radio`) для определения порядка сортировки (по возрастанию или по убыванию); `multiselect` для определения выбранных статусов; еще один виджет `multiselect` для определения выбранных степеней срочности; `text_input` для фильтрации задач с заданным содержимым. На рис. 14.11 показано, как выбрать подмножество задач при помощи этих параметров.

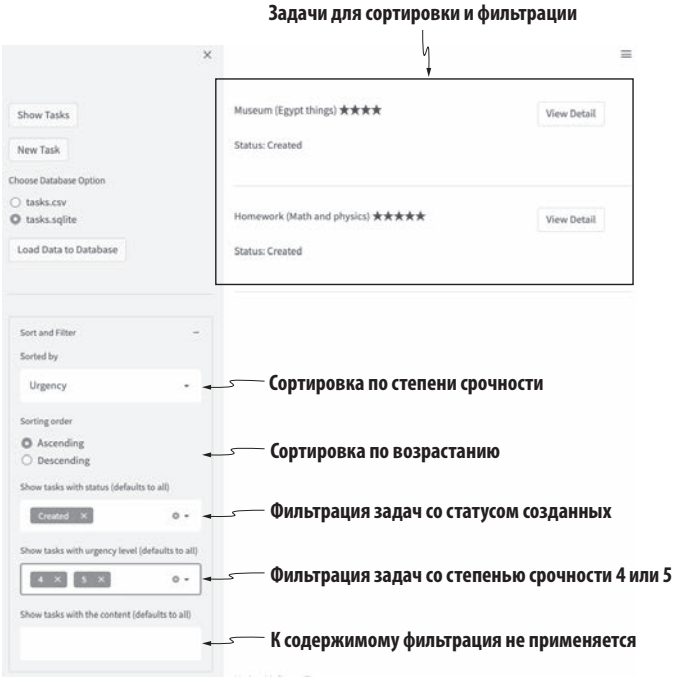


Рис. 14.11. Выбор подмножества задач с использованием виджета Sort and Filter. После того как пользователь задаст параметры сортировки и фильтрации, приложение загружает набор задач, соответствующих этим критериям, и выводит его в основной области содержимого

Когда пользователь просматривает подробную информацию о задаче в основной области содержимого, на боковой панели отображаются средства для вызова функции `setup_deletion`:

```
def setup_deletion():
    task = session[working_task_key]
    text_title = sidebar.text_input("Enter task title to delete",
    ➥ key="existing_delete")
    submitted = sidebar.button("Delete Task")
    if submitted:
        if text_title == task.title:
            task.delete_from_db()
            sidebar.success("Your task has been deleted.")
        else:
            sidebar.error("You must enter the exact text for the
            ➥ title to delete.")
```

В этой функции задача извлекается обращением по ключу сеанса `working_task`. Чтобы предотвратить случайное удаление задачи пользователем, мы требуем, чтобы пользователь ввел название задачи перед ее удалением из базы данных. Новая функциональность вызывает функции `success` и `error`, удобные для оперативного получения положительной и отрицательной обратной связи о действиях, выполняемых пользователем (рис. 14.12).

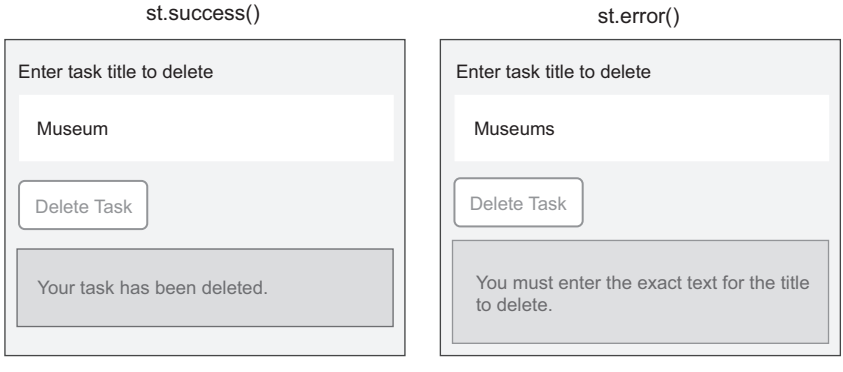


Рис. 14.12. Обратная связь в веб-приложении. Для предоставления положительной обратной связи вызывается st.success, для отрицательной — st.error

14.4.5. Вывод задач

В таск-менеджере полезно было бы вывести список задач, доступных для работы пользователя. А значит, страница для вывода задач достаточно важна. В этом разделе показано, как реализовать эту функциональность с использованием `streamlit`. В листинге 14.9 функция `show_tasks` вызывается для настройки веб-элементов для вывода задач. В листинге 14.11 показано, как реализована функция `show_tasks`.

Листинг 14.11. Вывод списка задач в веб-приложении

```
def show_tasks():
    filter_params = session[sorting_params_key]
    if filter_params[TaskierFilterKey.SORTING_KEY.value] is not None:
        reading_params = get_reading_params(filter_params)
        tasks = Task.load_tasks(**reading_params)
        sorting_key = sorting_keys[filter_params[
        ➥ TaskierFilterKey.SORTING_KEY.value]]
        should_reverse = filter_params[
        ➥ TaskierFilterKey.SORTING_ORDER.value] == sorting_orders[1]
        tasks.sort(key=lambda x: getattr(x, sorting_key),
        ➥ reverse=should_reverse)
    else:
        tasks = Task.load_tasks()  <-- Загружает данные

    for task in tasks:
        col1, col2 = st.columns([3, 1])  <-- Создает два столбца для более наглядного вывода
        col1.write(str(task))
        col2.button("View Detail", key=task.task_id,
        ➥ on_click=wants_task_detail, args=(task,))
        st.write(f"Status: {task.status.name.title()}")
        st.markdown("___")  <-- Выводит данные
```

Код состоит из двух частей. Первая часть загружает данные, с параметрами сортировки и фильтрации и без них, а вторая часть выводит данные с использованием виджетов.

Первая часть листинга 14.11 состоит из двух фаз.

1. *Получение параметров фильтрации из пользовательского ввода с помощью вызова функции* `get_reading_params`. Функция будет рассмотрена ниже в этом разделе.
2. *Сортировка задач на основании предоставленных параметров.* Так как для хранения задач используется список, то есть изменяемый объект (раздел 3.1), задачи сортируются методом `sort` (раздел 3.2). Поскольку ключ сортировки может измениться, будет слишком утомительно создавать разные лямбда-функции для аргумента `key`: например, `lambda x: x.title` для сортировки по названию или `lambda x: x.urgency` для сортировки по степени срочности. По этой причине мы используем обобщенный подход для динамического получения соответствующего атрибута: `lambda x: getattr(x, sorting_key)`.

Во второй части листинга 14.11 соответствующие виджеты используются для отображения задач. Здесь используется новый виджет с именем `columns` — невидимый виджет для упорядочения вывода. Вызов `st.columns([3, 1])` создает два столбца с соотношением ширин 3:1, а возвращаемым значением этого вызова является кортеж, представляющий эти два столбца.

Используя распаковку кортежа, мы получаем ссылки на столбцы с именами `col1` и `col2` и можем добавлять виджеты в столбцы. Один из этих виджетов — кнопка `View Detail`. По щелчку на кнопке выводится подробная информация о задаче в основной области содержимого, как говорилось в разделе 14.4.6. Функция `get_reading_params` работает следующим образом:

```
def get_reading_params(filter_params):
    reading_params = dict.fromkeys(["statuses", "urgencies", "content"])
    if selected_statuses := filter_params[
    ➥ TaskierFilterKey.SELECTED_STATUSES.value]:
        reading_params["statuses"] = [status_options.index(x) for x
        ➥ in selected_statuses]
    if selected_urgencies := filter_params[
    ➥ TaskierFilterKey.SELECTED_URGENCIES.value]:
        reading_params["urgencies"] = selected_urgencies
    if selected_content := filter_params[
    ➥ TaskierFilterKey.SELECTED_CONTENT.value]:
        reading_params["content"] = selected_content
    return reading_params
```

Как показано на рис. 14.11, пользователи настраивают три параметра фильтрации: статус, степень срочности и содержимое. В приведенном фрагменте все достаточно просто, не считая одного нового приема, который ранее нам не встречался: *выражение присваивания* (assignment expression). В этом приеме используется оператор `:=`, появившийся в Python 3.8. Например, код `selected_statuses := filter_params[TaskierFilterKey.SELECTED_STATUSES.value]` означает, что мы пытаемся получить значение, связанное с ключом `selected_statuses` из словаря `filter_params`, и присвоить его переменной с именем `selected_statuses`. Если значение отлично от `None`, то выполняется код в инструкции `if`. Как правило, присваивание является инструкцией, поэтому оно не может использоваться в инструкции `if`, где требуется, чтобы в секции содержалось выражение. Как видите, выражение присваивания выполняет две операции: оно присваивает значение и вычисляет его.

> **НАПОМИНАНИЕ** Результатом вычисления выражения является объект, а инструкция выполняет действие, не возвращая значение. Различия между инструкциями и выражениями подробно рассматриваются в разделе 2.1.3.

14.4.6. Вывод подробной информации о задаче

Список задач предоставляет общую информацию о задачах. Мы можем вывести более подробные сведения о каждой задаче. В этом разделе показано, как это делается.

Для кнопки `View Detail` задается аргумент `on_click` с использованием функции `wants_task_detail` и аргумент `args`, который задается с использованием

(task,). Если пользователь щелкает на этой кнопке, вызывается функция wants_task_detail(task):

```
def wants_task_detail(task: Task):
    update_session_tracking(working_task_key, task)
    update_session_tracking(menu_key,
    ➥ TaskierMenuOption.SHOW_TASK_DETAIL.value)
```

При вызове функции выполняются две операции:

- Задача, связанная с кнопкой View Detail, назначается текущей рабочей задачей.
- Выбранное меню изменяется для вывода подробной информации о задаче. После изменения меню при перезагрузке веб-приложения выводится страница с подробной информацией о задаче; для этого используется вызов функции show_task_detail, приведенной в листинге 14.12.

Листинг 14.12. Вывод подробной информации о задаче

```
def show_task_detail():
    task = session[working_task_key]
    form = st.form("existing_task_form", clear_on_submit=False)

    form.title("Task Detail")

    task.title = form.text_input("The title", value=task.title,
    ➥ key="existing_task_title")

    task.desc = form.text_input("The description", value=task.desc,
    ➥ key="existing_task_desc")

    task.urgency = form.slider("The urgency level", min_value=1,
    ➥ max_value=5, value=task.urgency)

    status = form.selectbox("The status", index=task.status,
    ➥ options=status_options, key="existing_task_status")
    task.status = TaskStatus(status_options.index(status))

    task.completion_note = form.text_input("The completion note",
    ➥ value=task.completion_note, key="existing_task_note")
    submitted = form.form_submit_button("Update Task")
    if submitted:
        try:
            task.update_in_db()
        except TaskierError:
            form.error("Couldn't update the task as it's maybe
            ➥ deleted already.")
        else:
            session[working_task_key] = task
            form.success("Your Task Was Updated!")
```

В листинге 14.12 следует обратить внимание на следующее:

- Виджет `form` используется для группировки отдельных виджетов, например `slider` и `text_input`. Виджет `form` запоминает пользовательский ввод для содержащихся в нем виджетов, так что при перезагрузке веб-страницы в форме отображается введенная информация.
- Разобравшись с обновлением, мы вызываем функцию `form_submit_button`, которая добавляет в форму кнопку Submit и использует возвращаемое значение, равное `True` при щелчке по кнопке.
- При отправке данных формы для обновления записи в базе данных используется инструкция `try...except...else...` (разделы 12.3 и 12.4). Здесь применяется обработка исключений, потому что пользователь мог удалить задачу с боковой панели или использовать другую вкладку для удаления задачи.

Проектировать интерфейс реального веб-приложения подобным образом не следует. Если пользователь удалил элемент, следует направить его на страницу, на которой удаленный элемент не отображается. Я привожу этот пример исключительно в целях демонстрации, чтобы показать, как использовать обработку исключений в проекте.

14.4.7. Создание новой задачи

В таск-менеджере у пользователя есть возможность создания новой задачи. Данный раздел показывает, как реализовать эту возможность в веб-приложении. Для этого мы определяем функцию `show_new_task_entry`, представленную в листинге 14.13.

Листинг 14.13. Создание новой задачи в веб-приложении

```
def show_new_task_entry():
    with st.form("new_task_form", clear_on_submit=True):
        st.title("New Task")

        title = st.text_input("The title", key="new_task_title")

        desc = st.text_input("The description", key="new_task_desc")

        urgency = st.slider("The urgency level", min_value=1, max_value=5)

        submitted = st.form_submit_button("Save Task")

   if submitted:
       task = Task.task_from_form_entry(title, desc, urgency)
       task.save_to_db()
       st.success("Your Task Was Saved!")
```

Как и на странице с подробной информацией о задаче, мы используем виджет `form` для ввода новой задачи. Отличие от листинга 14.12 заключается в том, что для формы используется инструкция `with`, создающая менеджер контекста (раздел 11.1). В инструкции `with` при вызове `st.text_input` для создания текстового поля `streamlit` знает, что поле должно быть размещено внутри формы благодаря менеджеру контекста. С другой стороны, когда в листинге 14.12 менеджер контекста не использовался, мы явно вызывали `form.text_input` для добавления текстового поля в форму. Возможны оба решения — с менеджером контекста и без него.

14.4.8. Организация проекта

Вы увидели, как реализовать разные функциональные аспекты приложения по отдельности. С точки зрения сопровождаемости важно организовать проект так, чтобы участники команды могли легко прочитать код и найти нужную функциональность. В этом разделе я опишу некоторые принципы организации проекта на примере фреймворка `streamlit`. Так как окончательным результатом является веб-приложение, наше внимание будет сосредоточено на файле скрипта taskier_app.py, ответственного за создание этого приложения.

В общем случае скрипт состоит из трех компонентов: зависимостей, глобальных переменных и функций для настройки интерфейса. Для нашего веб-приложения сценарий использует в качестве основной модели данных класс `Task`. И хотя класс `Task` используется только из файла скрипта, включать его в этот файл нежелательно по двум причинам:

- Это усложнит чтение файла скрипта, а читателю будет труднее понять, как устроено веб-приложение, потому что класс `Task` занимает значительное место в коде и не участвует в формировании интерфейса приложения.
- Это затруднит использование этого класса для других целей, например при построении приложений для рабочего стола. Следовательно, для реализации модели данных стоит взять отдельный файл.

Используя модель данных в своем приложении, мы импортируем ее в виде зависимости. В файле скрипта зависимости размещаются в начале файла, как показано в следующем фрагменте. Зависимости не только обслуживают код в сценарии, но и предоставляют важную информацию, которая будет полезна читателям кода (в том числе другим участникам команды), в частности, о библиотеках и пакетах, используемых сценарием:

```
import streamlit as st
from taskier import Task, TaskierDBOption, set_db_option,
➥ TaskStatus, TaskierError
from taskier_app_helper import TaskierMenuOption, TaskierFilterKey
```

Как вы, возможно, заметили, классы `TaskierMenuOption` и `TaskierFilterKey` сохраняются в другом файле (taskier_app_helper.py), чтобы файл taskier_app.py содержал только код построения интерфейса веб-приложения.

Разобравшись с организацией зависимостей, перейдем к анализу организации компонентов файла скрипта. На рис. 14.13 эта организация представлена в графическом виде.

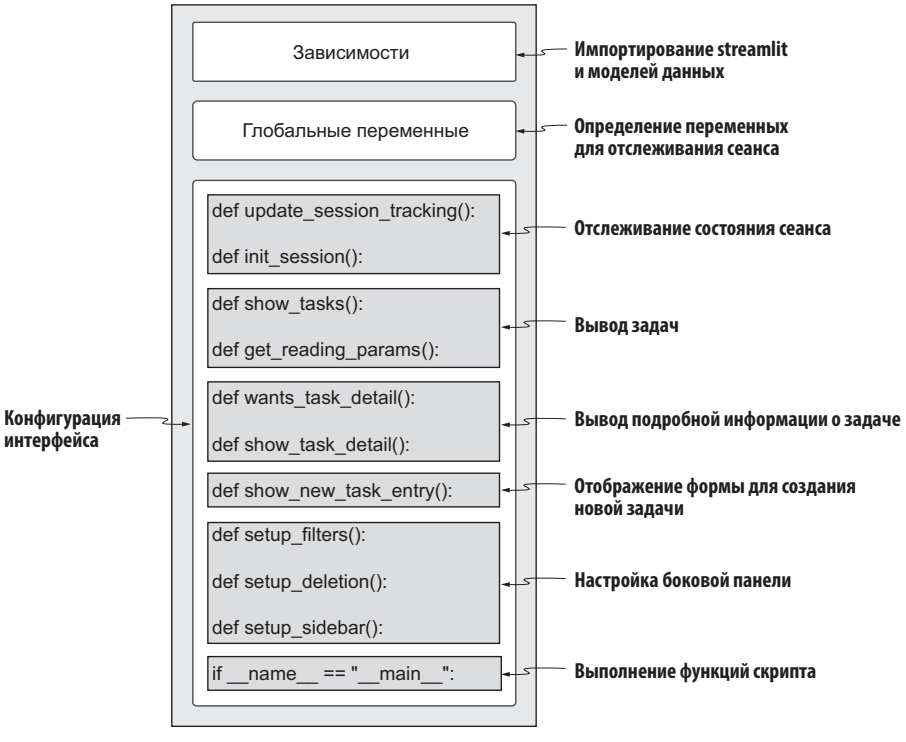


Рис. 14.13. Организация файла taskier_app.py. Файл состоит из трех компонентов: зависимостей, глобальных переменных и конфигурации интерфейса

В коде, определяющем конфигурацию интерфейса, я упорядочил функции в зависимости от их предназначения. Взаимосвязанные функции объединяются в группы. Код отслеживания сеанса расположен наверху, потому что это главное в отслеживании активности пользователя. В середине находятся функции конфигурации основной области содержимого. В конце следуют функции, которые определяют боковую панель.

14.4.9. Запуск приложения

Мы завершили код и аккуратно упорядочили его. Пришло время запустить приложение и опробовать его на практике. Учтите, что в процессе разработки приложение следует запускать в браузере для наблюдения за его быстродействием

в реальном времени. Чтобы запустить приложение, введите следующую команду в командной строке:

```
$ streamlit run taskier_app.py
```

Проследите за тем, чтобы команда выполнялась после перехода к каталогу, в котором был сохранен файл taskier_app.py; в противном случае вам придется задать полный путь к файлу скрипта. В браузере по умолчанию должна появиться новая вкладка, в которой выполняется наше приложение.

14.4.10. Обсуждение

Освоение такого фреймворка, как `streamlit`, потребует некоторого времени. В этом разделе не приводятся технические подробности использования фреймворка. Вместо этого мы построили веб-приложение, включая его интерфейс и модели данных, чтобы вы увидели, как описанные в книге средства используются в реальных проектах. В конце раздела была описана организация проекта. Хотя приложение создавалось в учебных целях, все же важно упорядочивать его код, чтобы он хорошо читался и не создавал проблем с сопровождением.

> **ПРИМЕЧАНИЕ** Компания, стоящая за разработкой фреймворка `streamlit`, разрешает бесплатно публиковать ваши веб-приложения, если код приложения размещается в общем доступе на GitHub. Информацию о публикации приложений можно найти по адресу https://share.streamlit.io/.

14.4.11. Задача

Одной из глобальных переменных, определяемых в приложении, является переменная `sorting_keys`, которая содержит объект `dict`: `{"Title": "title", "Description": "desc", "Urgency": "urgency", "Status": "status", "Note": "completion_note"}`. Этот объект используется для создания виджета `selectbox`: `st.selectbox("Sorted by", sorting_keys)`. В этом вызове объект `dict` используется как набор вариантов для виджета. Но почему мы можем использовать объект `dict` вместо объекта `list`, например `list(sorting_keys.keys())`?

> **ПОДСКАЗКА** Виджету `selectbox` для определения набора вариантов можно передать любой итерируемый объект. Объект `dict` является итерируемым объектом, по умолчанию его ключи рассматриваются как элементы итератора.

ИТОГИ

- Для каждого проекта желательно создать виртуальную среду, которая формирует изолированное окружение для управления зависимостями проекта и позволяет избегать требований к зависимостям между проектами.

- Модуль `venv` предоставляет встроенное решение для управления виртуальными средами.
- Некоторые сторонние средства, такие как `conda`, позволяют определить отдельный интерпретатор Python для каждой виртуальной среды, которая может обеспечить бóльшую гибкость, если в ваших проектах используются разные версии Python.
- Модели данных должны служить бизнес-целям вашего проекта. Следовательно, перед написанием кода для реализации моделей данных следует определиться с бизнес-целями.
- Файлы с кодом должны хорошо читаться, и при создании классов следует писать doc-строки для каждого определяемого метода.
- SQLite — облегченная база данных, не требующая предварительной настройки. Базы данных SQLite создаются во всех основных операционных системах, в том числе для мобильных устройств (например, для смартфонов).
- По сравнению с CSV-файлами база данных SQLite является более формальным вариантом хранения данных. Я использовал CSV-файл в качестве источника данных в учебных целях, но в реальном проекте всегда следует отдавать предпочтение базам данных.
- Веб-приложения отлично подходят для представления ваших проектов, так как они не зависят от платформы. Python поддерживает несколько веб-фреймворков, включая фреймворк `streamlit`, который подходит для простого построения веб-приложений.
- Созданный для этой книги проект был совсем небольшим, но даже в таких проектах вы всегда должны следить за структурой файлов и их внутреннего кода. Организация проекта играет важную роль для улучшения удобочитаемости и простоты сопровождения кода.

Решения задач

Глава 1

Во вводной главе задач не было. Вам повезло!

Глава 2

Раздел 2.1

Начнем со следующего объекта `dict`:

```
product = {"name": "Vacuum", "price": 130.675}
```

Решение, которое выдает нужный результат:

```
product_tag = f"{product['name']}: {{{product['price']:.2f}}}"

assert product_tag == "Vacuum: {130.68}"
```

Обычно для интерполяции переменных используются фигурные скобки. Чтобы эти символы обозначали просто фигурные скобки, а не служебные конструкции для интерполяции, необходимо использовать запись `{{`. Таким образом, `{{{var_name}` интерпретируется как левая фигурная скобка, за которой следует интерполируемая строка из `var_name`.

Раздел 2.2

Запрашивая входные данные от пользователя при помощи функции `input`, мы получаем строки. Если вы рассчитываете получить числовое значение, то строку необходимо преобразовать в соответствующее числовое значение. Например, рассмотрите такой код:

```
x = input("What's today's temperature in your area?")
x_num = float(x)  ◄── Преобразует строку в число с плавающей точкой
if x_num < 10:
    x_output = f"You entered {x_num:.1f} degrees. It's cold!"  ◄── .1f — спецификатор формата с плавающей точкой
elif 10 <= x_num < 25:
    x_output = f"You entered {x_num:.1f} degrees. It's cool!"
else:
    x_output = f"You entered {x_num:.1f} degrees. It's hot!"

print(x_output)
```

Если посмотреть на то, как `x_output` создается несколько раз, можно заметить повторения: разница только в наречии, описывающем погоду (cold, cool, hot — холодно, прохладно, жарко). Следовательно, решение можно улучшить:

```
x = input("What's today's temperature in your area?")
x_num = float(x)
if x_num < 10:
    x_whether = "cold"
elif 10 <= x_num < 25:
    x_whether = "cool"
else:
    x_whether = "hot"

x_output = f"You entered {x_num:.1f} degrees. It's {x_whether}!"
print(x_output)
```

Раздел 2.3

Аргумент `maxsplit` задает максимальное количество разбиений при использовании `split` или `rsplit`. Без этого аргумента оба метода используют все вхождения разделителя. Если же передать в аргументе значение, превышающее количество вхождений, оба метода должны вернуть одинаковые результаты:

```
fruits = "apple,orange,pineapple,cherry,watermelon"
assert fruits.split(",") == fruits.split(",", 10) ==
➥ fruits.rsplit(",") == fruits.rsplit(",", 10) ==
➥ ['apple', 'orange', 'pineapple', 'cherry', 'watermelon']
```

Но если использовать число, меньшее количества максимально доступных разбиений, `split` и `rsplit` вернут разные результаты:

```
assert fruits.split(",", 3) == ['apple', 'orange', 'pineapple',
➥ 'cherry,watermelon']
assert fruits.rsplit(",", 3) == ['apple,orange', 'pineapple',
➥ 'cherry', 'watermelon']
```

Раздел 2.4

Допустим, вы хотите разбить следующую строку:

```
data_to_split = "abc_,abc__,abc,,__abc_,_abc"
```

Как видите, разделители состоят из переменного количества `_` и `,`. Для разбиения таких строковых данных можно воспользоваться шаблоном `[,_]+`, который

обозначает совпадение из нескольких вхождений _ или , в строке. Применяя этот шаблон, можно создать требуемое разбиение:

```
import re
pattern = r"[,_]+"  ◄—— Использует неформатированную строку для создания шаблона
splitted = re.split(pattern, data_to_split)
print(splitted)
# Вывод: ['abc', 'abc', 'abc', 'abc', 'abc']
```

Раздел 2.5

При обработке многострочного текста можно использовать последовательность \n для обозначения конца строки. Таким образом, чтобы извлечь нужные записи без разбиения строк, можно попробовать применить следующий шаблон, указывая, что запись завершается символом новой строки:

```
text_data = """101, Homework; Complete physics and math
some random nonsense
102, Laundry; Wash all the clothes today
54, random; record
103, Museum; All about Egypt
1234, random; record
Another random record"""

import re
pattern = r"(\d{3}), (\w+); (.+)\n"
splitted = re.findall(pattern, text_data)
print(splitted)

# Вывод: [('101', 'Homework', 'Complete physics and math'), ('102',
➥ 'Laundry', 'Wash all the clothes today'), ('103', 'Museum',
➥ 'All about Egypt'), ('234', 'random', 'record')]
```

Вроде бы все работает, но с одним исключением: в результат попала неправильная запись `('234', 'random', 'record')`. Это неудивительно, ведь шаблон не устанавливает никаких ограничений на то, что предшествует идентификатору из трех цифр. Следующий шаблон более точно описывает требуемое:

```
pattern = r"(?<!\d)(\d{3}), (\w+); (.+)\n"
splitted = re.findall(pattern, text_data)
print(splitted)
# output: [('101', 'Homework', 'Complete physics and math'), ('102',
➥ 'Laundry', 'Wash all the clothes today'), ('103', 'Museum',
➥ 'All about Egypt')]
```

Часть `(?<!\d)` называется *негативной ретроспективной проверкой* (negative look-behind assertion). Это означает, что указанный фрагмент совпадает с текстом, содержащим число из трех цифр, только если ему не предшествует другое число. Учтите, что в этом примере продемонстрировано нетривиальное применение регулярных выражений. За дополнительной информацией обращайтесь на официальный сайт Python по адресу https://docs.python.org/3/library/re.html.

Глава 3

Раздел 3.1

Если вам понадобится хранить серию местоположений (например, историю поездок), в качестве модели данных нужно использовать объект `list`, потому что пользователи могут изменить список посещенных мест (скажем, добавить новые).

Местоположение описывается парой координат, которые не должны изменяться. Следовательно, для хранения данных координат следует использовать кортеж:

```
(широта, долгота)
```

Раздел 3.2

Ниже приведен список, который требуется отсортировать по длине описаний:

```
tasks = [
    {'title': 'Laundry', 'desc': 'Wash clothes', 'urgency': 3},
    {'title': 'Homework', 'desc': 'Physics + Math', 'urgency': 5},
    {'title': 'Museum', 'desc': 'Egyptian things', 'urgency': 2}
]
```

Мы знаем, что в аргументе `key` должна передаваться функция, а длина описания должна вычисляться так:

```
def using_by_desc_len(task):
    return len(task["desc"])
tasks.sort(key=using_by_desc_len, reverse=True)
print(tasks)
# output: [{'title': 'Museum', 'desc': 'Egyptian things', 'urgency':
➥ 2}, {'title': 'Homework', 'desc': 'Physics + Math', 'urgency':
➥ 5}, {'title': 'Laundry', 'desc': 'Wash clothes', 'urgency': 3}]
```

Мы определяем функцию `using_by_desc_len`, которая возвращает длину описания задачи. Напомним, что функция, передаваемая в аргументе `key`, должна получать ровно один аргумент. Аргументу `reverse` необходимо присвоить значение `True`, так как упражнение требует, чтобы задачи с более длинным описанием имели более высокий ранг. Если вы уже умеете работать с лямбда-функциями (раздел 7.1), используйте для сортировки следующий код:

```
tasks.sort(key=lambda x:len(x["desc"]), reverse=True)
```

Раздел 3.3

Так как именованный кортеж представляет собой объект `tuple`, его невозможно изменить (кортежи неизменяемы). Если же вы попытаетесь это сделать, возникнет исключение `AttributeError`:

```
from collections import namedtuple

Task = namedtuple("Task", "title desc urgency")
task = Task(title='Laundry', desc='Wash clothes', urgency=3)
```

```
task.urgency = 4
# ERROR: AttributeError: can't set attribute
```

Однако именованные кортежи предоставляют обходное решение в виде метода `_replace`:

```
task._replace(urgency=4)
# Вывод: Task(title='Laundry', desc='Wash clothes', urgency=4)
```

Этот метод создает новый объект `tuple`, который содержит измененное значение (вместо создания измененной копии исходного объекта на месте).

Раздел 3.4

Допустим, имеется объект `dict` с именем `numbers`:

```
numbers = {"one": 1, "two": 2, "three": 3}
numbers_key = numbers.keys()
id_key = id(numbers_key)
print(id_key)

# Вывод: 140660045849520  ◄—— На вашем компьютере значение будет другим
```

В этом фрагменте мы также получаем набор ключей с помощью метода `keys`, который возвращает объект словарного представления. Встроенная функция `id` получает адрес памяти объекта представления. Объект `dict` можно изменить, добавив в него новую пару «ключ — значение»:

```
numbers["four"] = 4
```

После этого изменения мы видим, что ключи автоматически обновляются в объекте `numbers_key` и что адрес памяти остается неизменным, потому что обновление оперирует тем же объектом:

```
print(numbers_key)
# Вывод: dict_keys(['one', 'two', 'three', 'four'])
print(id_key)
# Вывод: 140660045849520
```

Раздел 3.5

Ключи объекта `dict` должны быть хешируемыми, потому что хеш-коды будут использоваться нижележащей хеш-таблицей для хранения данных. Если у вас имеются ключи с одинаковыми хеш-кодами, действует правило «последнего встреченного»: связанное с ключом значение, которое задается позднее, становится значением для данного ключа. В нашем случае целое число `1` и число с плавающей точкой `1.0` имеют одинаковые хеш-коды:

```
assert hash(1) == hash(1.0) == 1
```

Таким образом, по правилу «последнего встреченного» следует ожидать, что значением для данного ключа станет значение, связанное с `1.0`.

```
numbers = {1: "one", 1.0: "one point one"}
print(numbers)
# Вывод: {1: 'one point one'}
```

Раздел 3.6

Как указано в подсказке, такие вычисления называются *ускоренными*. Когда Python пытается вычислить `expr_a or expr_b`, то если первое выражение равно `True`, используется первый объект; в противном случае — второе выражение. Следующие примеры поясняют это правило:

```
assert ({1, 2, 3} or {4, 5, 6}) == {1, 2, 3}
assert (False or []) == []
assert ("Hello" or "World") == "Hello"
```

Если же Python пытается вычислить `expr_a and expr_b` и первое выражение равно `False`, то используется первый объект; в противном случае используется второе выражение:

```
assert ({1, 2, 3} and {4, 5, 6}) == {4, 5, 6}
assert (False and []) == False
assert ("Hello" and "World") == "World"
```

Это правило может быть сложнее запомнить, чем для операций `or`. Небольшая подсказка: при ускоренном вычислении для операций `and` они интерпретируются как `True` только в том случае, если оба операнда равны `True`. А значит, если Python обнаруживает, что первое выражение равно `False`, вычисление на этом завершается; результат в любом случае будет равен `False`. По этой причине Python использует первое выражение.

Глава 4

Раздел 4.1

Когда вы создаете подпоследовательность на основе среза, она должна полностью совпадать с исходной последовательностью. Несколько примеров:

```
num_list = [1, 2, 3, 4]
num_tuple = (1, 2, 3, 4)
num_str = "1234"

print(num_list[:2])
# Вывод: [1, 2]

print(num_tuple[:2])
```

```
# Вывод: (1, 2)

print(num_str[:2])
# Вывод: 12
```

Ту же нарезку можно выполнить с объектом `range`. Нетрудно предположить, что подпоследовательность также является объектом `range`:

```
num_range = range(1, 5)
print(num_range[:2])
# Вывод: range(1, 3)
```

Раздел 4.2

Требуется получить данные продаж за ноябрь. Данные за весь год выглядят так:

```
revenue_by_month = [95, 100, 80, 93, 92, 110, 102, 88, 96, 98, 115, 120]
```

Как упоминалось в разделе, нужную точку данных можно получить при помощи выражения `revenue_by_month[-2]` с использованием отрицательного индекса. Если вы хотите использовать положительный индекс, его можно получить вычислением длины:

```
assert revenue_by_month[-2] ==
➥ revenue_by_month[len(revenue_by_month) - 2]
```

Раздел 4.3

При выполнении следующего фрагмента произойдет исключение `ValueError`:

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

tasks = [
    Task("Laundry", 3),
    Task("Museum", 4),
    Task("Homework", 5),
    Task("Ticket", 2)
]

task_to_search = Task("Homework", 5)
tasks.index(task_to_search)
# ERROR: ValueError: <__main__.Task object at 0x7fee281be3e0>
➥ is not in list
```

Дело в том, что хотя `task_to_search` вроде бы содержит те же атрибуты, что и третий элемент списка `tasks`, экземпляры пользовательского класса в исходном виде несовместимы. Встроенные данные, например строки, совместимы, и для поиска элемента можно использовать метод `index`. Чтобы сравнения работали, необходимо переопределить специальный метод `__eq__`:

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

    def __eq__(self, __o: object):
        return self.__dict__ == __o.__dict__

tasks = [
    Task("Laundry", 3),
    Task("Museum", 4),
    Task("Homework", 5),
    Task("Ticket", 2)
]

task_to_search = Task("Homework", 5)
print(tasks.index(task_to_search))
# Вывод: 2
```

Определение пользовательских классов рассматривается в главе 8.

Раздел 4.4

Когда вы распаковываете объект `list` с внутренними структурами, последние можно распаковать как серию автономных элементов. Следующий фрагмент показывает, как это делается:

```
data_to_unpack = [1, (2, 3), 4]

a, (b, c), d = data_to_unpack
print(a, b, c, d)
# Вывод: 1, 2, 3, 4
```

Раздел 4.5

Если умножить вложенный объект `list` на 3 напрямую, элементы будут повторены 3 раза:

```
numbers = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
print(numbers * 3)

# Вывод: [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [1, 2, 3],
➥ [4, 5, 6], [7, 8, 9], [10, 11, 12], [1, 2, 3], [4, 5, 6], [7, 8, 9],
➥ [10, 11, 12]]
```

Однако в данной задаче требуется другое — умножить каждый элемент на 3. С такими данными необходимо использовать циклы `for`:

```
numbers_multiplied = []
for number_list in numbers:
    embedded_list = []
    for number in number_list:
        number_multiplied = number * 3
        embedded_list.append(number_multiplied)
```

```
    numbers_multiplied.append(embedded_list)

print(numbers_multiplied)
# Вывод: [[3, 6, 9], [12, 15, 18], [21, 24, 27], [30, 33, 36]]
```

Вложенные циклы `for` плохо читаются. Лучше воспользоваться списковыми включениями, описанными в разделе 5.2:

```
numbers_multiplied2 = [x*3 for number_list in numbers for x in number_list]
assert numbers_multiplied == numbers_multiplied2
```

Если в вашем приложении часто выполняются числовые вычисления, стоит обратить внимание на такие структуры данных, как `array` из библиотеки NumPy. Более элегантное решение с использованием библиотеки NumPy выглядит так:

```
import numpy as np

numbers_array = np.array(numbers)

print(numbers_array * 3)

# Выводимые строки:
[[ 3  6  9]
 [12 15 18]
 [21 24 27]
 [30 33 36]]
```

Как показано в этом фрагменте кода, умножение с использованием `array` из библиотеки NumPy подобно другим алгебраическим операциям, которые вы обычно выполняете с числами. Согласитесь, что такой подход намного удобнее.

Глава 5

Раздел 5.1

Чтобы соединить три и более итерируемых объекта, следует последовательно перебрать их. Каждый элемент итератора `zip` состоит из одного элемента каждого итерируемого объекта, образующих кортеж, как в следующем примере:

```
numbers_int = [1, 2, 3]
numbers_word = ("one", "two", "three")
letters = "abc"
for item in zip(numbers_int, numbers_word, letters):
    print(item)

# Выводимые строки:
(1, 'one', 'a')
(2, 'two', 'b')
(3, 'three', 'c')
```

Количество элементов, генерируемых `zip`, зависит от итерируемого объекта с наименьшем количеством элементов. Следующий пример подтверждает это:

```
numbers_fewer = [1, 2]
numbers_more = [3, 4, 5, 6]
for item in zip(numbers_fewer, numbers_more):
    print(item)

# Выводимые строки:
(1, 3)
(2, 4)
```

Один итерируемый объект (`numbers_fewer`) состоит из двух элементов, а другой (`numbers_more`) — из четырех. При их слиянии генерируются две пары, что соответствует количеству элементов в `numbers_fewer`.

Раздел 5.2

Попробуйте выполнить код с использованием (`выражение for элемент in итерируемый_объект`). Рассмотрим следующий пример:

```
numbers = [1, 2, 3]

numbers_gen = (x*x for x in numbers)

print(type(numbers_gen))
# Вывод: <class 'generator'>
```

Как показывает этот фрагмент, выражение (`x*x for x in numbers`) создает генератор — разновидность итерируемого объекта, экономичного по затратам памяти (раздел 7.4). Очевидно, это не объект `tuple`, а в Python не существует такой конструкции, как кортежное включение.

Раздел 5.3

Допустим, имеется следующий объект `dict`:

```
numbers = {"one": 1, "two": 2}
```

Вы можете перебрать ключи этого объекта `dict`:

```
for key in numbers.keys():
    print(key)

# Выводимые строки:
one
two
```

Также можно перебрать значения объекта `dict`:

```
for value in numbers.values():
    print(value)
# Выводимые строки:
1
2
```

Возможен и перебор пар «ключ — значение»:

```
for key, value in numbers.items():
    print(f"{key}: {value}")

# Выводимые строки:
one: 1
two: 2
```

В приведенном выше коде `items` создает объекты `tuple` из ключа и значения, а кортеж можно распаковать. Обратите внимание, что существует синтаксический сахар. При переборе ключей можно использовать сам объект `dict`:

```
for key in numbers:
    print(key)

# Выводимые строки:
one
two
```

Раздел 5.4

Список задач, в котором проводится поиск:

```
from collections import namedtuple

Task = namedtuple("Task", "title, description, urgency")

tasks = [
    Task("Toaster", "Clean the toaster", 2),
    Task("Camera", "Export photos", 4),
    Task("Homework", "Physics and math", 5),
    Task("Floor", "Mop the floor", 3),
    Task("Internet", "Upgrade plan", 5),
    Task("Laundry", "Wash clothes", 3),
    Task("Museum", "Egypt exhibit", 4),
    Task("Utility", "Pay bills", 5)
]
```

Для поиска самой неотложной задачи с помощью инструкции `break` можно использовать следующий код (листинг 5.7):

```
first_urgent_task1 = None

for task in tasks:
    if task.urgency == 5:
        first_urgent_task1 = task
        break

print(first_urgent_task1)
# Вывод: Task(title='Homework', description='Physics and math', urgency=5)
```

В упражнении был вопрос о том, что произойдет, если не задать исходное значение для `first_urgent_task1`. Так как может оказаться, что ни одна неотложная задача не обнаружена, значение `first_urgent_task1` не будет задано и переменную нельзя будет использовать. Следующая модификация помогает увидеть потенциальную проблему:

```
for task in tasks:
    if task.urgency > 5:
        first_urgent_task2 = task
        break

print(first_urgent_task2)
# ERROR: NameError: name 'first_urgent_task2' is not defined.
```

В этом фрагменте задача считается неотложной, если ее степень срочности больше 5. При таком условии ни одна задача не удовлетворяет критерию, так что значение `first_urgent_task2` никогда не будет задано. При попытке вывести значение происходит ошибка `NameError` (раздел 10.4).

Глава 6

Раздел 6.1

Временную метку можно встроить как аргумент по умолчанию. Эта временная метка отражает время своего определения вместо времени вызова:

```
from datetime import datetime
from time import sleep

def set_start_time(time=datetime.today()):
    print(f"Time: {time}")

for _ in range(3):
    set_start_time()
    sleep(1.0)

# Выводимые строки:
Time: 2022-04-25 20:22:06.337848
Time: 2022-04-25 20:22:06.337848
Time: 2022-04-25 20:22:06.337848
```

Как видите, мы вызываем функцию многократно, ожидая, что при этом будут получены разные временные метки. Однако выясняется, что все временные метки одинаковы и в них отражено время создания функции.

Раздел 6.2

Возвращаемые значения имеют постоянную структуру `latitude` и `longitude` (широта и долгота), и для хранения этих двух значений можно создать именованный

кортеж. Ниже приводится возможная версия, полученная в результате рефакторинга:

```
from collections import namedtuple

Coordinate = namedtuple("Coordinate", ["latitude", "longitude"])

def locate_me():
    # Поиск текущего местоположения пользователя
    return coordinate0

def locate_home():
    # Поиск места жительства пользователя
    return coordinate1

def locate_work():
    # Поиск места работы пользователя
    return coordinate2
```

Вместо двух значений каждая из этих функций теперь может вернуть только объект `tuple`.

Раздел 6.3

Чтобы функция получала в аргументе список `list` с элементами `int` или `str`, можно воспользоваться аннотациями типов:

```
def run_computation(numbers: list[int | str]):
    pass
```

В примере используется аннотация `: list[int | str]`, которая означает, что объект `list` может содержать целые числа или строки.

Раздел 6.4

Вызов `example(a=1, b=2)` допустим, так как мы используем два ключевых аргумента. Вызов `example(1, 2)` недопустим, так как в нем используются позиционные аргументы, а функция должна получать ключевые. Вызов `example(2a=1, 2b=2)` недопустим, так как идентификаторы недействительны (они не могут начинаться с цифры). Вызов `example()` допустим, так как в нем нет ни одного ключевого аргумента. `**kwargs` обозначает произвольное количество ключевых аргументов, включая ноль.

Раздел 6.5

Можно определить следующую doc-строку в стиле Google:

```
def quotient(dividend, divisor, taking_int=False):
    """
    Вычисляет частное от деления двух чисел.
    Аргументы:
      dividend: int | float, делимое
```

```
      divisor: int | float, делитель
      taking_int: bool, признак получения целой части
      ➥ частного; default: False, вычисляется точный результат деления
    Возвращаемое значение:
      float | int, частное от деления

    Выдаваемые исключения:
      ZeroDivisionError, если divisor равен 0
    """

    if divisor == 0:
        raise ZeroDivisionError("division by zero")
    result = dividend / divisor
    if taking_int:
        result = int(result)
    return result
```

Глава 7

Раздел 7.1

Всем лямбда-функциям назначается имя `<lambda>`. Это чисто условное имя, поэтому лямбда-функции также называются анонимными. Обычной функции, напротив, назначается имя, которое совпадает с идентификатором, определенным в заголовке функции:

```
add_five = lambda x: x + 5

print(add_five.__name__)
# Вывод: <lambda>

def add_ten(x):
    return x + 10

print(add_ten.__name__)
# Вывод: add_ten
```

Раздел 7.2

Как указано в подсказке, пользователь может ввести аргумент, который не соответствует ни одному из указанных условий. Следует подготовиться к подобным некорректным вызовам. При помощи `get` можно использовать резервное действие `fallback_action`, если заданное действие не входит в объект `dict` с именем `actions`.

Раздел 7.3

Как указано в подсказках, необходимо добавить еще один уровень функции, который обеспечивает передачу аргумента. Возможное решение:

```
import functools
import time
def logging_time_app(app_name):
```

```
    def decorator(func):
        @functools.wraps(func)
        def logger(*args, **kwargs):
            """Log the time"""
            print(f"{app_name} --- {func.__name__} starts")
            start_t = time.time()
            value_returned = func(*args, **kwargs)
            end_t = time.time()
            print(f"{app_name} *** {func.__name__} ends; used time:
            ➥ {end_t - start_t:.2f} s")
            return value_returned

        return logger

    return decorator

@logging_time_app("Task Tracker")
def example_app():
    pass

example_app()
# Выводимые строки:
Task Tracker --- example_app starts
Task Tracker *** example_app ends; used time: 0.00 s
```

Внешняя функция `logging_time_app` представляет собой декоратор, который получает имя приложения в аргументе. Внутри этой функции определяется типичный декоратор, как это делается обычно, и этот декоратор получает реальную функцию, которая должна быть декорирована.

Раздел 7.4

На основании подсказки можно написать следующую функцию-генератор, которая выдает числа из последовательности Фибоначчи:

```
def fibonacci(n):
    a, b = 0, 1
while a < n:
        yield a
        a, b = b, a + b
```

Так как каждый элемент последовательности Фибоначчи вычисляется суммированием двух предыдущих элементов, последовательность инициализируется первыми двумя числами, а все последующие вычисляются по формуле. Функцию можно опробовать на примере создания объекта `list`:

```
below_fiften = fibonacci(15)

numbers = list(below_fiften)

print(numbers)
# Вывод: [0, 1, 1, 2, 3, 5, 8, 13]
```

Список представляет последовательность Фибоначчи до числа 13.

Раздел 7.5

Допустим, имеется функция `run_stats_model` и частичная функция `run_stats_model_a`:

```
from functools import partial

def run_stats_model(dataset, model, output_path):
    calculated_stats = 123
    return calculated_stats

run_stats_model_a = partial(run_stats_model, model="model_a",
➥ output_path="project_a/stats/")
```

Частичная функция создается из `run_stats_model`. Воспользовавшись подсказкой, мы можем просмотреть атрибуты этой частичной функции:

```
print(dir(run_stats_model_a))
# Вывод: ['__call__', '__class__', '__class_getitem__', '__delattr__',
➥'__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
➥'__getattribute__', '__gt__', '__hash__', '__init__',
➥'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
➥'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
➥'__setstate__', '__sizeof__', '__str__', '__subclasshook__',
➥'__vectorcalloffset__', 'args', 'func', 'keywords']
```

Как видно из вывода, у функции есть атрибут с именем `func`, по которому можно определить, какая функция является исходной:

```
print(run_stats_model_a.func)
# Вывод: <function run_stats_model at 0x7fedf82c30a0>
```

И действительно, это функция `run_stats_model`. Также можно попробовать определить, какую информацию содержат атрибуты `args` и `keywords`.

Глава 8

Раздел 8.1

В разделе 6.1 я говорил, что вам следует использовать `None` в качестве значения по умолчанию для изменяемых аргументов. То же следует сделать с методом `__init__`:

```
class Task:
    def __init__(self, title, desc, urgency, tags=None):
        self.title = title
        self.desc = desc
        self.urgency = urgency
        if tags is None:
            self.tags = []
        else:
            self.tags = tags
```

Также можно использовать тернарное выражение `var = значение_true if условие else значение_false`. Таким образом, приведенный выше код можно обновить следующим образом:

```
class Task:
    def __init__(self, title, desc, urgency, tags=None):
        self.title = title
        self.desc = desc
        self.urgency = urgency
        self.tags = [] if tags is None else tags
```

Раздел 8.2

При создании экземпляра на основе объекта `tuple` необходимо иметь доступ к конструктору класса. Следовательно, необходимо определить метод класса для обращения к данным класса:

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency

    @classmethod
    def task_from_tuple(cls, data):
        title, desc, urgency = data
        return cls(title, desc, urgency)
```

Раздел 8.3

Решение из листинга 8.9 также можно применить к `urgency`:

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self._urgency = urgency

    @property
    def urgency(self):
        return self._urgency

    @urgency.setter
    def urgency(self, value):
        if value in range(1, 6):
            self._urgency = value
        else:
            raise ValueError("Can't set a value outside of 1 - 5")
```

За подробными объяснениями обращайтесь к листингу 8.9.

Раздел 8.4

Вместо того чтобы жестко фиксировать имя класса, можно воспользоваться специальными атрибутами для получения этой информации на программном уровне:

```
class Task:
    def __init__(self, title, desc, urgency):
        self.title = title
        self.desc = desc
        self.urgency = urgency
    def __repr__(self):
        return f"{self.__class__.__name__}({self.title!r}, {self.desc!r},
        ➥ {self.urgency})"
```

Специальный атрибут `__class__` содержит класс экземпляра. С помощью его специального атрибута `__name__` получаем имя класса.

Раздел 8.5

Следующий код показывает, как переопределить метод инициализации в подклассе:

```
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id
class Supervisor:
    def __init__(self, name, employee_id, subordinates):
        super().__init__(name, employee_id)
        self.subordinates = subordinates
```

В методе `__init__` класса `Supervisor` вызов `super()` используется для создания объекта-заместителя его надкласса `Employee`, что позволяет использовать его метод `__init__` с передачей `name` и `employee_id`.

Глава 9

Раздел 9.1

Так как вызов `move_to` относится к конкретному экземпляру, его можно преобразовать в метод экземпляра класса `Direction`:

```
from enum import Enum

class Direction(Enum):
    NORTH = 0
    SOUTH = 1
    EAST = 2
    WEST = 3
```

```
    def __str__(self):
        return self.name.lower()

    def move_to(self, distance: float):
        if self in self.__class__:
            message = f"Go to the {self} for {distance} miles"
        else:
            message = "Wrong input for direction"
        print(message)
```

Как показано в этом фрагменте, первый аргумент метода `move_to` переименовывается в `self` и содержит ссылку на экземпляр. Внутри тела метода для получения ссылки на класс `Direction` можно воспользоваться `self.__class__`.

Раздел 9.2

Если при создании класса данных полю задается значение по умолчанию, можно воспользоваться функцией `field` модуля `dataclasses`, которая задает значение по умолчанию для изменяемых полей. Следующий фрагмент показывает, как реализуется эта возможность:

```
from dataclasses import dataclass, field

@dataclass
class Bill:
    table_number: int
    meal_amount: float
    served_by: str
    tip_amount: float
    dishes: field(default_factory=list)
```

В этом коде поле `dishes` является изменяемым, и мы можем задать аргумент `default_factory` в виде `list`, чтобы он создал пустой объект `list`.

Раздел 9.3

Как говорится в подсказке, объекты `tuple` являются сериализуемыми и их можно напрямую преобразовать в строки JSON:

```
import json
from collections import namedtuple

User = namedtuple("User", "first_name last_name age")
user = User("John", "Smith", "39")

print(json.dumps(user))
# Вывод: ["John", "Smith", "39"]
```

Раздел 9.4

Допустим, вы строите клиентское приложение с использованием следующей модели данных `Client`:

```
class ClientV0:
    def __init__(self, first_name, last_name, middle_initial='-'):
        self.first_name = first_name
        self.last_name = last_name
        self.middle_initial = middle_initial
        self.initials = first_name[0] + middle_initial + last_name[0]
```

Все должно быть просто. Когда вы получаете инициалы клиента для экземпляра, используется значение, которое было задано изначально. Но в приложении имеется функция, с помощью которой пользователь может изменить имя, так что инициалы тоже нуждаются в обновлении. Чтобы инициалы пересчитывались на ходу, можно преобразовать атрибут `initials` в функцию следующим образом:

```
class ClientV1:
    def __init__(self, first_name, last_name, middle_initial='-'):
        self.first_name = first_name
        self.last_name = last_name
        self.middle_initial = middle_initial
    def initials(self):
        return self.first_name[0] + self.middle_initial + self.last_name[0]
```

Такое решение работает, но оно может нарушить работоспособность вашего кода. Ранее для обращения к инициалам клиента использовался синтаксис `client.initials`; теперь приходится использовать `client.initials()`. Чтобы избежать использования оператора вызова, можно применить декоратор `property`:

```
class ClientV2:
    def __init__(self, first_name, last_name, middle_initial='-'):
        self.first_name = first_name
        self.last_name = last_name
        self.middle_initial = middle_initial

@property
    def initials(self):
        return self.first_name[0] + self.middle_initial + self.last_name[0]
```

В этом случае можно сохранить целостность программного интерфейса (API), используя `client.initials`, но вам придется обеспечить необходимое преобразование с вызовом функции для этого свойства. Таким образом, использование декоратора позволит избежать изменений, нарушающих API. Вы можете сохранить целостность API даже при том, что реализация стала свойством вместо атрибута.

Раздел 9.5

Так как все эти методы могут быть ограниченными, я преобразую их в защищенные методы при помощи префикса `_`:

```
class Account:
    def __init__(self, student_id):
        self.student_id = student_id
```

```
        # Запрос к базе данных для получения дополнительной информации
        # по значению student_id
        self.account_number = self._get_account_number_from_db()
        self.balance = self._get_balance_from_db()

    def _get_account_number_from_db(self):
        # Запрос к базе данных для получения номера счета
        # по значению student_id
        account_number = 123456
        return account_number

    def _get_balance_from_db(self):
        # Запрос к базе данных для получения баланса
        # по номеру счета
        balance = 100.00
        return balance

class Demographics:
    def __init__(self, student_id):
        self.student_id = student_id
        # Запрос к базе данных для получения дополнительной информации
        age, gender, race = self._get_demographics_from_db()
        self.age = age
        self.gender = gender
        self.race = race

    def _get_demographics_from_db(self):
        # Запрос к базе данных для получения демографических данных
        # по идентификатору student_id
        birthday = "08/14/2010"
        age = self._calculated_age(birthday)
        gender = "Female"
        race = "Black"
        return age, gender, race

    @staticmethod
    def _calculated_age(birthday):
        # Получение текущей даты и вычисление разности с днем рождения
        age = 12
        return age
```

Глава 10

Раздел 10.1

Как упоминалось в подсказке, модуль `collections.abc` содержит класс `Iterable`, а итерируемые объекты должны реализовать обязательный метод `__iter__`. Следовательно, чтобы узнать, является ли объект итерируемым, к этому классу можно применить функцию `isinstance`:

```
from collections.abc import Iterable

def is_iterable(obj):
    if isinstance(obj, Iterable):
```

```
        outcome = "is an iterable"
    else:
        outcome = "is not an iterable"
    print(type(obj), outcome)
```

При помощи обновленной функции можно проверить некоторые распространенные типы данных:

```
is_iterable([1, 2, 3])
# Вывод: <class 'list'> is an iterable

is_iterable((404, "Data"))
# Вывод: <class 'tuple'> is an iterable

is_iterable("abc")
# Вывод: <class 'str'> is an iterable

is_iterable(456)
# Вывод: <class 'int'> is not an iterable
```

Раздел 10.2

Чтобы проверить, как использование переменной изменяет счетчик ссылок, можно написать тривиальную функцию:

```
import sys
class Task:
    def __init__(self, title):
        self.title = title

task = Task("Homework")

def get_detail(obj):
    print(sys.getrefcount(obj))
```

Если вызвать `get_detail` с переменной `task`, счетчик ссылок принимает значение

```
get_detail(task)
# Вывод: 4
```

Почему 4? Первое приращение относится к самой переменной `task`. При вызове `get_detail` передается ссылка на `task`, в результате счетчик увеличивается до 2. Функция `get_detail` получает `task`, счетчик увеличивается до 3 . В теле функции вызов `sys.getrefcount` добавляет еще 1, в результате счетчик увеличивается до 4.

Раздел 10.3

В соответствии с указанными требованиями можно обновить класс `Task` до следующей версии:

```
class Task:
    def __init__(self, title, desc, tags = None):
        self.title = title
        self.desc = desc
```

```
        self.tags = [] if tags is None else tags

    def __copy__(self):
        new_title = f"Copied: {self.title}"
        new_desc = self.desc
        new_tags = self.tags.copy()
        new_task = self.__class__(new_title, new_desc, new_tags)
        return new_task
```

В методе `__copy__` мы создаем новое значение `title` и новый список `tags` для копируемого объекта. Следующий код помогает проверить, работает ли метод так, как предполагалось:

```
from copy import copy

task = Task("Homework", "Math and physics", ["school", "urgent"])

new_task = copy(task)
print(new_task.__dict__)
# Вывод: {'title': 'Copied: Homework', 'desc': 'Math and physics',
➥ 'tags': ['school', 'urgent']}
```

Чтобы лишний раз убедиться в том, что атрибуты `tags` этих двух объектов действительно различаются, можно попробовать изменить один список:

```
task.tags.append("red")
print(task.tags)
# Вывод: ['school', 'urgent', 'red']

print(new_task.tags)
# Вывод: ['school', 'urgent']
```

Все работает так, как предполагалось: `task.tags` и `new_task.tags` — два разных объекта `list`.

Раздел 10.4

В Python инструкция `if...else...` не формирует собственную область видимости в отличие от классов и функций. А поскольку собственной области видимости у нее нет, глобальную переменную можно изменить без ключевого слова `global`, как показано в следующем примере:

```
import random

weather = "sunny"

if random.randint(1, 100) % 2:
weather = "cloudy"
else:
    weather = "rainy"

print(weather)
# Вывод: cloudy ◄—— Из-за фактора случайности ваш результат может быть другим
```

Как показано в этом фрагменте, переменная `weather` изменяется без ключевого слова `global`; это означает, что инструкция `if...else...` не образует область видимости, что позволяет обращаться к `weather` напрямую.

Раздел 10.5

Вам уже известно, что при определении декоратора как класса для хранения метаданных декорируемой функции эту функцию необходимо упаковать. Но в отличие от функции-декоратора, с которой можно использовать декоратор `wraps`, декоратор в форме класса использует метод `update_wrapper`, помогающий хранить метаданные:

```
import time
import functools

class TimeLogger:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        def logger(*args, **kwargs):
            start = time.time()
            result = func(*args, **kwargs)
            print(f"Calling {func.__name__}: {time.time() - start:.5f}")
            return result
        self._logger = logger

    def __call__(self, *args, **kwargs):
        return self._logger(*args, **kwargs)

@TimeLogger
def calculate_sum(n):
    return sum(range(n))

print(calculate_sum.__name__)
# Вывод: calculate_sum
```

Использование `update_wrapper` почти не отличается от использования декоратора `wraps`. Обертка обновляется в методе `__init__` класса `TimeLogger`. Следует заметить, что декоратор `wraps` — это синтаксический сахар, так как во внутренней реализации он вызывает `update_wrapper`.

Глава 11

Раздел 11.1

К каждому элементу необходимо добавить разрыв строки. Используя списковое включение, можно создать новый объект `list` на основе `list_data`:

```
list_data = [
'1001,Homework,5',
'1002,Laundry,3',
```

```
'1003,Grocery,4'
]

updated_list_data = [f"{x}\n" for x in list_data]
```

С обновленным списком можно использовать функцию `writelines` для получения нужного файла. Чтобы убедиться в том, что запись была выполнена успешно, достаточно прочитать данные:

```
with open("tasks_list_write.txt", "w") as file:
    file.writelines(updated_list_data)

with open("tasks_list_write.txt") as file:
    print(file.read())

# Выводимые строки:
1001,Homework,5
1002,Laundry,3
1003,Grocery,4
```

Раздел 11.2

Функция `writerows` работает с объектом `list`, поэтому данные каждой строки (объект `list`) можно встроить во внешний объект `list`, как рекомендуется в подсказке:

```
tasks = [
    ['1001', 'Homework', '5'],
    ['1002', 'Laundry', '3'],
    ['1003', 'Grocery', '4']
]
```

Затем для записи этого списка выполняется следующий код:

```
import csv
with open("tasks_writer.txt", "w", newline="") as file:
    csv_writer = csv.writer(file)
    csv_writer.writerows(tasks)
```

Открыв файл tasks_writer.txt, вы убедитесь в том, что данные были записаны правильно.

Раздел 11.3

Метод `__reduce__` переопределяется в классе `MaliciousTask` следующим образом:

```
import os
class MaliciousTask:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency
    def __reduce__(self):
        print("__reduce__ is called")
        return os.system, ('rm hacking.txt',)
```

В данном случае мы используем `('rm hacking.txt',)` вместо `('touch hacking.txt')`. Команда `rm` удаляет заданный файл. После обновления класса можно выполнить код из листинга 11.14, чтобы увидеть этот эффект.

Раздел 11.4

Чтобы проверить существование файла, можно вызвать для экземпляра класса `Path` метод `exists`. Таким образом, листинг 11.17 принимает следующий вид:

```
from pathlib import Path
import shutil

shutil.rmtree("subjects")  ◄——  Удаляет существующую папку

subject_ids = [123, 124, 125]
data_folder = Path("data")

for subject_id in subject_ids:
    subject_folder = Path(f"subjects/subject_{subject_id}")
    subject_folder.mkdir(parents=True, exist_ok=True)

    for subject_file in data_folder.glob(f"*{subject_id}*"):
        filename = subject_file.name
        target_path = subject_folder / filename
        if not target_path.exists():
            _ = shutil.copy(subject_file, target_path)
            print(f"Copying {filename} to {target_path}")
        else:
            print(f"{filename} already exists at {target_path}")
```

Как показано в этом коде, файлы копируются только в том случае, если файл в целевом пути не существует. Тем самым предотвращается случайная перезапись уже обработанных файлов.

Раздел 11.5

Вы уже знаете, что время изменения файла можно узнать из поля `st_mtime` информации о статусе файла. Следующая функция возвращает файлы, время изменения которых приходится на последние 24 часа:

```
from pathlib import Path
import time

def select_recent_files_24h(directory):
    dir_path = Path(directory)
    current_time = time.time()
    time_cutoff = current_time - 24 * 3600
    good_files = []
    for file_path in dir_path.glob("*"):
        file_time = file_path.stat().st_mtime
        if time_cutoff <= file_time <= current_time:
            good_files.append(file_path)

    return good_files
```

Шаблон `"*"` позволяет перебрать все файлы в каталоге. Мы указываем, что время изменения файла должно лежать в интервале, соответствующем последним 24 часам. Если файл удовлетворяет этому требованию, он добавляется в список `good_files` как часть вывода функции.

Глава 12

Раздел 12.1

Чтобы проверить, содержит ли регистратор какие-либо обработчики перед добавлением нового обработчика, можно вызвать метод `hasHandlers` регистратора:

```
import logging

logger = logging.getLogger(__name__)

if not logger.hasHandlers():
    file_handler = logging.FileHandler("taskier.log")
    logger.addHandler(file_handler)
```

Чтобы очистить все обработчики, можно выполнить операцию с атрибутом `handlers` регистратора, который представляет собой объект `list`:

```
print(logger.handlers)
# Вывод: [<FileHandler /directory/taskier.log (NOTSET)>]

logger.handlers.clear()  <-- Удаляет все обработчики

print(logger.handlers)
# Вывод: []
```

Раздел 12.2

Чтобы продемонстрировать, что происходит в программе, я использую потоковый обработчик, чтобы сообщения могли выводиться на консоль:

```
import logging

logger = logging.getLogger(__name__)
logger.handlers = []
logger.setLevel(logging.WARNING)

stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.DEBUG)
logger.addHandler(stream_handler)

logger.info("It's an info message.")
# Вывод: None (hide automatically in the console)
logger.warning("It's a warning message.")
# Вывод: It's a warning message.
```

Если выполнить этот код в консоли, вы увидите, что выводится только предупреждение: сообщение уровня `INFO` ниже уровня регистратора, поэтому оно не будет отправлено обработчику. А вот сообщение на уровне `WARNING` соответствует требуемому уровню регистратора и передается обработчику.

Раздел 12.3

Как указано в подсказке, можно выполнить потенциально проблемный код в консоли и посмотреть, что при этом произойдет. Пример:

```
>>> urgency = int("3#")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3#'
```

В программе происходит исключение `ValueError`. Сделайте шаг назад и добавьте это исключение в инструкцию `try...except...`:

```
try:
    urgency = int(urgency_str)
except ValueError:
    # Операция при возникновении ValueError
```

Раздел 12.4

Если выполнить код из упражнения, на консоль будет выведен следующий результат:

```
Done processing text: Laundry,3
finally
```

Значение `task` не возвращается, так как инструкция `return` в секции `finally` выполняется до инструкции `return` в секции `try`.

Раздел 12.5

Чтобы код можно было опробовать несколько раз, я определю функцию, которая может создавать задачу на базе разных видов входных данных:

```
def create_task(task_title):
    try:
        print(f"Trying to process {task_title}")
        task = Task(task_title)
    except TypeError as e:
        print(f"Couldn't create the task, error: {e}")
    else:
        print(f"Created task: {task}")
    finally:
        print(f"Done processing {task_title}")
```

При обработке исключений эта функция использует все четыре секции. Попробуем вызвать ее:

```
>>> create_task(100)
Trying to process 100
Couldn't create the task, error: Please instantiate the Task using
➥ string as its title
Done processing 100
>>> create_task("Laundry")
Trying to process Laundry
Created task: <__main__.Task object at 0x1043e7b80>
Done processing Laundry
```

Как видите, при использовании других объектов, кроме `str`, выполняются секции `try`, `except` и `finally`. При использовании объекта `str` выполняются секции `try`, `else` и `finally`.

Глава 13

Раздел 13.1

Трассировку можно усложнить разными способами. Одно из возможных решений выглядит так:

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency

    def _report(self):
        print("report")
        report = "Urgency: " + self.urgency

    def _send_report(self):
        print("send report")
        self._report()

    def _update_db(self):
        # update the record in the database
        print("update the database")
        self._send_report()

    def update_urgency(self, urgency):
        self.urgency = urgency
        self._update_db()

task = Task("Laundry", 3)
task.update_urgency(4)

# Выводимые строки:
update the database
send report
report
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 17, in update_urgency
  File "<stdin>", line 14, in _update_db
  File "<stdin>", line 10, in _send_report
  File "<stdin>", line 7, in _report
TypeError: can only concatenate str (not "int") to str
```

В этом классе разные методы вызывают друг друга, поэтому трассировка включает несколько вызовов.

Раздел 13.2

В пространствах имен переменные отслеживаются динамически. Вызов встроенной функции `locals` выдает имена, доступные в локальном пространстве имен на настоящий момент. Следующий фрагмент кода демонстрирует эффект изменений:

```
$ python3 task_debug.py
> /fullpath/task_debug.py(10)create_task()
-> task_text = obtain_text_data(inject_bug)
(Pdb) locals()
{'inject_bug': False}
(Pdb) n
> /fullpath/task_debug.py(11)create_task()
-> title, urgency_text = task_text.split(",")
(Pdb) locals()
{'inject_bug': False, 'task_text': 'Laundry,3'}
(Pdb)
```

Раздел 13.3

Для создания экземпляра класса `Task` на базе объекта `tuple` может использоваться следующая функция:

```
def create_task_from_tuple(task_tuple):
    title, urgency = task_tuple
    task = Task(title, urgency)
    return task
```

В тестовом классе можно определить следующую функцию для функции `create_task_from_tuple`:

```
import unittest
class TestTaskCreation(unittest.TestCase):
    def setUp(self):
        task_to_compare = Task("Laundry", 3)
        self.task_dict = task_to_compare.__dict__

    def test_create_task_from_tuple(self):
        task_tuple = ("Laundry", 3)
        created_task = create_task_from_tuple(task_tuple)
        self.assertEqual(created_task.__dict__, self.task_dict)
```

Раздел 13.4

Метод можно обновить, чтобы в нем явно выдавалось исключение. Необходимо изменить класс `Task` в файле test_class.py следующим образом:

```
class Task:
    def __init__(self, title, urgency):
        self.title = title
        self.urgency = urgency
    def formatted_display(self):
        displayed_text = f"{self.title} ({self.urgency})"
        raise TypeError("This is a TypeError")
        # Следующая команда return будет пропущена
        ➥ из-за выдачи исключения
        return displayed_text
```

Если снова запустить файл test_task_class.py, вы получите в командной строке вывод (см. ниже), который показывает, что ошибка произошла из-за исключения `TypeError` в нашем коде. Обратите внимание: результаты обозначаются последовательностью `..E` вместо `..F`, потому что сбой происходит из-за ошибки, а не из-за того, что тест не прошел:

```
$ python3 test_task_class.py
..E
======================================================================
ERROR: test_formatted_display (__main__.TestTask)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/fullpath/test_task_class.py", line 21, in test_formatted_display
    displayed_text = task.formatted_display()
  File "/fullpath/task_class.py", line 22, in formatted_display
    raise TypeError("This is a TypeError")
TypeError: This is a TypeError

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (errors=1)
```

Глава 14

Раздел 14.1

Используйте приложение Terminal, если вы работаете на Mac, или командную строку, если вы работаете в системе Windows. Перейдите в нужный каталог, а затем выполните следующую команду для создания виртуальной среды:

```
$ python3 -m venv python-env
```

После создания виртуальной среды необходимо активизировать ее командой:

```
# для Mac:
$ source taskier-env/bin/activate
```

```
# для Windows:
> taskier-env\Scripts\activate.bat
```

Чтобы установить библиотеку pandas, выполните команду:

```
$ pip install pandas
```

Чтобы использовать виртуальную среду в Visual Studio Code, обращайтесь к подробным инструкциям из раздела 14.1.4.

Раздел 14.2

Можно воспользоваться флагом для обозначения того, была ли найдена запись:

```
def delete_from_db(self):
    """Удаление записи из базы данных
    """
    if app_db == TaskierDBOption.DB_CSV.value:
        with open(app_db, "r+") as file:
            lines = file.readlines()
            found_record = False
            for line in lines:
                if line.startswith(self.task_id):
                    found_record = True
                    lines.remove(line)
                    break
            if not found_record:
                raise Exception("Record not found error.")
            else:
                file.seek(0)
                file.truncate()
                file.writelines(lines)
```

Как показано в этом фрагменте, флаг инициализируется значением `False`. Если запись будет найдена, флагу присваивается `True`. Если по результатам поиска флаг содержит `False`, то программа выдает исключение.

Раздел 14.3

В главе 7 показано, как создать декоратор для измерения времени. Ниже приведена возможная реализация из листинга 7.9:

```
import functools
import time

def logging_time_wraps(func):
    @functools.wraps(func)
    def logger(*args, **kwargs):
        """Log the time"""
        print(f"--- {func.__name__} starts")
        start_t = time.time()
        value_returned = func(*args, **kwargs)
        end_t = time.time()
```

```
        print(f"*** {func.__name__} ends; used time: {end_t -
        ➥ start_t:.10f} s")
        return value_returned

    return logger
```

Этот декоратор может использоваться для декорирования методов в классе. Чтобы продемонстрировать эту возможность, я декорирую метод `load_tasks`:

```
@classmethod
@logging_time_wraps
def load_tasks(cls, statuses: list[TaskStatus]=None,
➥ urgencies: list[int]=None, content: str=""):
```

Хотя я не буду проводить формальное сравнение, похоже, база данных SQLite 3 превосходит CSV-файлы по скорости чтения данных. Пожалуйста, обратите внимание на то, что мы имеем дело с небольшим объемом данных, так что различия между этими двумя источниками кажутся тривиальными:

```
# Использование CSV-файла в качестве источника данных
*** load_tasks ends; used time: 0.0008411407 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0005502701 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0004429817 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0002791882 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0003058910 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0005359650 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0002870560 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0004091263 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0004007816 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0002658367 s

# Использование SQLite в качестве источника данных
--- load_tasks starts
*** load_tasks ends; used time: 0.0003259182 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0002837181 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0004198551 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0002789497 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0003492832 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0003030300 s
```

```
--- load_tasks starts
*** load_tasks ends; used time: 0.0004410744 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0003309250 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0003337860 s
--- load_tasks starts
*** load_tasks ends; used time: 0.0002810955 s
```

Раздел 14.4

Для представления вариантов в виджете `selectbox` в `streamlit` может использоваться любой итерируемый объект. Когда в качестве итерируемого объекта используется объект `dict`, `dict` и `dict.keys()` приводят к одному и тому же результату, как в следующем примере:

```
numbers = {0: "zero", 1: "one", 2: "two"}
assert list(numbers) == list(numbers.keys())
```

Юн Цуй

Рецепты Python. Коллекция лучших техник программирования

Перевел с английского Е. Матвеев

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*Е. Ваулина*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 15.03.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 1000. Заказ 0000.