МУХАММАД АЗИФ



PYTHON ДЛЯ ГИКОВ

Создавайте эффективные приложения,
используя лучшие практики программирования

bhv®

Packt›

Python for Geeks

Build production-ready applications using advanced Python concepts and industry best practices

Muhammad Asif



Packt>

BIRMINGHAM—MUMBAI

"Python" and the Python Logo are trademarks of the Python Software Foundation.

МУХАММАД АЗИФ

PYTHON ДЛЯ ГИКОВ

Санкт-Петербург
«БХВ-Петербург»
2024

УДК 004.43
ББК 32.973.26-018.1
А35

Азиф М.

А35 Python для гиков: Пер. с англ. — СПб.: БХВ-Петербург, 2024. — 432 с.: ил.

ISBN 978-5-9775-0956-5

Книга подробно рассказывает о разработке, развертывании и поддержке крупномасштабных проектов на Python. Представлены такие концепции, как итераторы, генераторы, обработка ошибок и исключений, обработка файлов и ведение журналов. Приведены способы автоматизации тестирования приложений и разработки через тестирование (TDD). Рассказано о написании приложений с использованием кластера Apache Spark для обработки больших данных, о разработке и развертывании бессерверных программ в облаке на примере Google Cloud Platform (GCP), о создании веб-приложений и REST API, использовании среды Flask. Показаны способы применения языка для создания, обучения и оценки моделей машинного обучения, а также их развертывания в облаке, описаны приемы использования Python для извлечения данных с сетевых устройств и систем управления сетью (NMS).

Для программистов

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Марины Попович*
Редактор	*Александр Морозов*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Анна Брезман*
Оформление обложки	*Зои Канторович*

© Packt Publishing 2021. First published in the English language under the title 'Python for Geeks – (9781801070119)'

Впервые опубликовано на английском языке под названием 'Python for Geeks – (9781801070119)'

Подписано в печать 05.09.23.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 34,83.
Тираж 1500 экз. Заказ № 7579.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

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

ISBN 978-1-80107-011-9 (англ.)
ISBN 978-5-9775-0956-5 (рус.)

© Packt Publishing, 2021
© Перевод на русский язык, оформление.
ООО "БХВ-Петербург", ООО "БХВ", 2024

Содержание

Об авторе **14**

О рецензентах **15**

Предисловие **16**
Для кого предназначена эта книга 16
О чем эта книга 17
Как получить максимальную отдачу от книги 18
Загрузка файлов с примерами кода 18
Условные обозначения 19

Раздел 1. Python помимо основ **21**

Глава 1. Оптимальный жизненный цикл разработки на Python **23**
Культура и сообщество Python 23
Этапы проекта Python 26
Стратегия процесса разработки 27
Итерация по этапам 28
Стремление к MVP в первую очередь 28
Стратегия разработки для специализированных предметных областей 29
Эффективное документирование кода Python 32
Комментарии Python 32
Docstring 32
Документация на уровне функций или классов 34
Разработка эффективной схемы именования 35
Методы 36
Переменные 36
Константы 37
Классы 37
Пакеты 38
Модули 38
Соглашения об импорте 38
Аргументы 38
Полезные инструменты 38

Системы контроля версий ... 39
Что не стоит хранить в репозитории системы контроля версий ... 39
Понимание стратегий развертывания кода ... 40
Пакетная разработка ... 40
Среды разработки Python ... 42
IDLE ... 42
Sublime Text ... 42
Atom ... 42
PyCharm ... 42
Visual Studio Code ... 43
PyDev ... 43
Spyder ... 43
Заключение ... 43
Вопросы ... 44
Дополнительные ресурсы ... 44
Ответы ... 44

Глава 2. Использование модулей для сложных проектов ... 45

Технические требования ... 46
Знакомство с модулями и пакетами ... 46
Импорт модулей ... 46
Оператор import ... 48
Оператор __import__ ... 52
Инструмент importlib.import_module ... 52
Абсолютный и относительный импорт ... 53
Загрузка и инициализация модуля ... 55
Загрузка модуля ... 55
Установка параметров для специальных переменных ... 55
Выполнение кода ... 56
Стандартные модули ... 57
Написание многоразовых модулей ... 58
Независимая функциональность ... 58
Генерализация функционала ... 59
Традиционный стиль программирования ... 60
Четко определенная документация ... 61
Сборка пакетов ... 62
Именование ... 63
Файл инициализации пакета ... 63
Сборка пакета ... 63
Доступ к пакетам из любого расположения ... 66
Общий доступ к пакету ... 70
Создание пакета в соответствии с рекомендациями PyPA ... 70
Установка из локального исходного кода с помощью pip ... 73
Публикация пакета в Test PyPI ... 75
Установка пакета из PyPI ... 76

Заключение ... 77
Вопросы ... 77
Дополнительные ресурсы ... 77
Ответы ... 78

Глава 3. Расширенное объектно-ориентированное программирование на Python ... 79

Технические требования ... 80
Знакомство с классами и объектами ... 80
Различия между атрибутами класса и атрибутами экземпляра ... 80
Конструкторы и деструкторы классов ... 83
Различия между методами класса и методами экземпляра ... 84
Специальные методы ... 85
Принципы ООП ... 86
Инкапсуляция данных ... 87
Объединение данных и действий ... 87
Сокрытие информации ... 89
Защита данных ... 91
Традиционный подход к использованию геттеров и сеттеров ... 91
Использование декоратора property ... 92
Расширение классов с помощью наследования ... 94
Простое наследование ... 94
Множественное наследование ... 96
Полиморфизм ... 97
Перегрузка метода ... 97
Переопределение метода ... 98
Абстракция ... 100
Композиция как альтернативный подход к проектированию ... 102
Утиная типизация в Python ... 104
Когда не стоит использовать ООП в Python ... 105
Заключение ... 106
Вопросы ... 106
Дополнительные ресурсы ... 107
Ответы ... 107

Раздел 2. Расширенные концепции программирования ... 109

Глава 4. Библиотеки Python для продвинутого программирования ... 111

Технические требования ... 111
Введение в контейнеры данных Python ... 112
Строки ... 112
Списки ... 113
Кортежи ... 114
Словари ... 114
Множества ... 115

Итераторы и генераторы для обработки данных ... 116
Итераторы ... 116
Генераторы ... 120
Обработка файлов в Python ... 122
Операции с файлами ... 123
Обработка ошибок и исключений ... 126
Работа с исключениями в Python ... 127
Вызов исключений ... 129
Определение пользовательских исключений ... 130
Модуль logging в Python ... 131
Основные компоненты системы логирования ... 132
Работа с модулем logging ... 134
Что стоит и не стоит записывать в журнал ... 140
Заключение ... 141
Вопросы ... 141
Дополнительные ресурсы ... 141
Ответы ... 141

Глава 5. Тестирование и автоматизация с помощью Python ... 143

Технические требования ... 144
Понимание различных уровней тестирования ... 144
Модульное тестирование ... 145
Интеграционное тестирование ... 145
Системное тестирование ... 145
Приемочное тестирование ... 146
Работа с тестовыми фреймворками Python ... 146
Работа с фреймворком unittest ... 148
Фреймворк тестирования pytest ... 157
Разработка через тестирование ... 165
Красный ... 165
Зеленый ... 166
Рефакторинг ... 166
Автоматизированная непрерывная интеграция ... 167
Заключение ... 168
Вопросы ... 168
Дополнительные ресурсы ... 168
Ответы ... 169

Глава 6. Дополнительные советы и приемы Python ... 170

Технические требования ... 170
Расширенные приемы использования функций в Python ... 171
Функции counter, itertools и zip для итерационных задач ... 171
Использование методов filter, map и reduce для преобразования данных ... 175
Создание лямбда-функций ... 178
Внедрение одной функции в другую ... 179
Изменение поведения функции с помощью декораторов ... 181

Расширенные концепции структур данных ... 187
Внедрение словаря в словарь ... 187
Использование включений ... 190
Введение в Pandas DataFrame ... 192
Операции с объектом DataFrame ... 193
Сложные случаи использования DataFrame ... 198
Заключение ... 203
Вопросы ... 204
Дополнительные ресурсы ... 204
Ответы ... 204

Раздел 3. Масштабирование за пределы одного потока ... 205

Глава 7. Многопроцессорная обработка, многопоточность и асинхронное программирование ... 207
Технические требования ... 208
Многопоточность в Python и ее ограничения ... 208
Слепое пятно Python ... 209
Ключевые компоненты многопоточного программирования на Python ... 210
Практический пример: многопоточное приложение для загрузки файлов с Google Диска ... 218
Многопроцессорная обработка ... 221
Создание нескольких процессов ... 221
Обмен данными между процессами ... 224
Обмен объектами между процессами ... 228
Синхронизация процессов ... 230
Практический пример: многопроцессорное приложение для загрузки файлов с Google Диска ... 231
Асинхронное программирование для адаптивных систем ... 233
Модуль asyncio ... 234
Распределение задач с помощью очередей ... 236
Практический пример: асинхронное приложение для загрузки файлов с Google Диска ... 238
Заключение ... 240
Вопросы ... 240
Дополнительные ресурсы ... 241
Ответы ... 241

Глава 8. Масштабирование Python с помощью кластеров ... 242
Технические требования ... 243
Возможности кластеров для параллельной обработки ... 243
Hadoop MapReduce ... 244
Apache Spark ... 246
Устойчивые распределенные наборы данных (RDD) ... 249
Операции с RDD ... 249
Создание RDD ... 250

PySpark для параллельной обработки данных ... 251
Создание программ SparkSession и SparkContext ... 253
PySpark для операций с RDD ... 254
PySpark DataFrames ... 257
PySpark SQL ... 261
Практические примеры использования Apache Spark и PySpark ... 262
Пример 1: калькулятор числа π в Apache Spark ... 262
Заключение ... 268
Вопросы ... 269
Дополнительные ресурсы ... 269
Ответы ... 270

Глава 9. Программирование на Python для облака ... 271

Технические требования ... 271
Знакомство с облачными возможностями для приложений Python ... 272
Среды разработки Python для облака ... 272
Облачные среды выполнения для Python ... 274
Создание веб-сервисов Python для облачного развертывания ... 276
Использование Google Cloud SDK ... 277
Использование веб-консоли GCP ... 284
Использование Google Cloud Platform для обработки данных ... 287
Введение в основы Apache Beam ... 287
Конвейеры Apache Beam ... 289
Создание конвейеров для Cloud Dataflow ... 294
Заключение ... 298
Вопросы ... 299
Дополнительные ресурсы ... 299
Ответы ... 300

РАЗДЕЛ 4. PYTHON ДЛЯ ВЕБ-РАЗРАБОТКИ, ОБЛАКА И СЕТИ ... 301

Глава 10. Использование Python для разработки веб-приложений и REST API ... 303

Технические требования ... 304
Требования к веб-разработке ... 304
Веб-фреймворки ... 304
Пользовательский интерфейс ... 305
Веб-сервер/сервер приложений ... 306
База данных ... 307
Безопасность ... 307
API ... 307
Документация ... 307
Знакомство с фреймворком Flask ... 308
Создание базового веб-приложения с маршрутизацией ... 308
Обработка запросов с разными типами HTTP-методов ... 310
Отображение статического и динамического контента ... 312

Извлечение параметров из HTTP-запроса ... 313
Взаимодействие с системами управления базами данных ... 315
Обработка ошибок и исключений в веб-приложениях ... 318
Создание REST API ... 321
Использование Flask для REST API ... 322
Разработка REST API для доступа к базе данных ... 324
Пример: создание веб-приложения с помощью REST API ... 326
Заключение ... 331
Вопросы ... 332
Дополнительные ресурсы ... 332
Ответы ... 332

Глава 11. Разработка микросервисов на Python ... 334

Технические требования ... 334
Введение в микросервисы ... 335
Практические рекомендации по созданию микросервисов ... 337
Создание приложений на базе микросервисов ... 338
Варианты разработки микросервисов на Python ... 339
Варианты развертывания микросервисов ... 340
Разработка приложения на основе микросервисов ... 341
Заключение ... 352
Вопросы ... 352
Дополнительные ресурсы ... 352
Ответы ... 353

Глава 12. Создание бессерверных функций на Python ... 354

Технические требования ... 355
Знакомство с бессерверными функциями ... 355
Преимущества бессерверных функций ... 356
Варианты использования ... 356
Варианты развертывания бессерверных функций ... 357
Написание бессерверных функций ... 358
Создание облачной функции на основе HTTP с помощью консоли GCP ... 359
Практический пример: создание приложения для уведомлений о событиях в облачном хранилище ... 363
Заключение ... 367
Вопросы ... 367
Дополнительные ресурсы ... 367
Ответы ... 367

Глава 13. Python и машинное обучение ... 369

Технические требования ... 370
Введение в машинное обучение ... 370
Использование Python для машинного обучения ... 372
Библиотеки машинного обучения в Python ... 372
Рекомендации по обучающим данным ... 374

Создание и оценка модели машинного обучения ... 375
Процесс построения модели машинного обучения ... 375
Создание примера машинного обучения ... 376
Оценка модели с помощью кросс-валидации и тонкой настройки гиперпараметров ... 381
Сохранение ML-модели в файл ... 384
Развертывание и прогнозирование ML-модели в GCP Cloud ... 385
Заключение ... 388
Вопросы ... 388
Дополнительные ресурсы ... 388
Ответы ... 389

Глава 14. Python для автоматизации сети ... 390

Технические требования ... 391
Введение в автоматизацию сети ... 391
Плюсы и минусы автоматизации сети ... 392
Варианты использования ... 393
Взаимодействие с сетевыми устройствами ... 394
Протоколы для взаимодействия с сетевыми устройствами ... 394
Взаимодействие с сетевыми устройствами с помощью библиотек Python на основе SSH ... 397
Взаимодействие с сетевыми устройствами с помощью NETCONF ... 404
Интеграция с системами управления сетью ... 408
Использование конечных точек сервиса определения местоположения ... 409
Получение токена аутентификации ... 410
Получение сетевых устройств и инвентаризация интерфейсов ... 411
Обновление порта на сетевом устройстве ... 412
Интеграция с событийно-ориентированными системами ... 414
Создание подписок для Apache Kafka ... 416
Обработка событий от Apache Kafka ... 417
Продление и удаление подписки ... 418
Заключение ... 418
Вопросы ... 419
Дополнительные ресурсы ... 419
Ответы ... 420

Предметный указатель ... 421

Моей жене, Сайме Аруж, без любящей поддержки которой
было бы невозможно завершить эту книгу.
Моим дочерям, Сане Азиф и Саре Азич и моему сыну Зейну Азиф,
которые вдохновляли меня на протяжении всего этого путешествия.
В память о моих отце и матери за их жертвы и за то,
что показали силу решимости.
А также моим братьям и сестрам за всю их поддержку,
которую они оказывали в отношении моей учебы и работы.

- Мухаммад Азиф

Об авторе

Мухаммад Азиф — главный архитектор решений с большим опытом в области веб-разработки, виртуализации, машинного обучения, а также автоматизации сетей и облачных систем. Благодаря огромному опыту в разных областях он привел многие крупномасштабные проекты к успешному развертыванию. В последние годы Мухаммад перешел на более высокие руководящие должности, но ему по-прежнему нравится самому писать код и использовать современные технологии для решения реальных задач. В 2012 году он получил докторскую степень по компьютерным системам в Карлтонском университете в Оттаве, а сейчас руководит разработкой решений в Nokia.

Я хочу поблагодарить всех, кто поддерживал меня, особенно Имрана Ахмеда, который помог мне написать главу 1.

О рецензентах

Харшит Джейн (**Harshit Jain**) — специалист по data science с пятилетним опытом в сфере программирования, помогает компаниям создавать и применять сложные алгоритмы машинного обучения для повышения эффективности бизнеса. Он имеет широкий спектр знаний — от классического машинного обучения до глубокого обучения и компьютерного зрения. В свободное время он наставляет будущих специалистов data science. Его доклады и статьи, в том числе «*Learning the Basics of Data Science*» и «*How to Check the Impact on Marketing Activities — Market Mix Modeling*», показывают, что наука о данных — это его страсть. Рецензирование этой книги — его первая, но, определенно, не последняя попытка укрепить свои знания о Python для помощи в вашем обучении.

Сураб Бхавсар (**Sourabh Bhavsar**) — старший fullstack-работаботчик, практикует Agile и облачные технологии, имеет опыт в разработке программного обеспечения более 7 лет. Он закончил аспирантуру по искусственному интеллекту и машинному обучению в Техасском университете в Остине, получил степень магистра делового администрирования (MBA) по маркетингу и степень бакалавра инженерных наук в области информационных технологий в университете Пуны, Индия. В настоящий момент он работает ведущим инженером в PayPal, где отвечает за проектирование и разработку решений на основе микросервисов, а также реализацию различных движков рабочих процессов и оркестрации. Сураб верит, что учиться никогда не поздно, и любит изучать новые веб-технологии. В свободное время он играет на барабанах табла и изучает астрологию.

Предисловие

Python — многоцелевой язык, который помогает решать задачи любой сложности в множестве областей. *«Python для гиков»* научит вас, как продвигаться по карьерной лестнице с помощью советов и приемов экспертов.

Сначала вы исследуете различные способы работы с Python как с точки зрения проектирования, так и с точки зрения реализации. Далее вы рассмотрите жизненный цикл масштабного проекта Python. По мере продвижения сфокусируетесь на создании элегантной структуры с помощью ее модульного разделения. Вы узнаете, как масштабировать свой код за пределы одного потока, а также как реализовать многопроцессорную и многопоточную обработки. В дополнение поймете, как можно использовать язык не только для развертывания на одной машине, но и для использования целых кластеров в облачной вычислительной среде. Затем вы познакомитесь с методами обработки информации, рассмотрите многоразовые и масштабируемые конвейеры данных и узнаете, как использовать эти методы для автоматизации сети, бессерверных функций и машинного обучения. В завершение вы научитесь, как составить эффективный стратегический план веб-разработки, используя практики и рекомендации из этой книги.

К концу этой книги вы сможете уверенно программировать на Python для крупных и сложных проектов.

Для кого предназначена эта книга

Эта книга будет полезна Python-разработчикам среднего уровня в любой области, которые стремятся развить навыки проектирования и управления масштабными сложными проектами. Она также пригодится программистам, которые хотят создавать многоразовые модули и библиотеки Python и разрабатывать приложения для облачного развертывания. Предыдущий опыт работы с этим языком поможет вам извлечь максимальную пользу из книги.

О чем эта книга

Глава 1 «Оптимальный жизненный цикл разработки на Python» поможет понять жизненный цикл стандартного проекта Python и его этапы, а также здесь приведены лучшие практики и советы по написанию кода.

Глава 2 «Использование модулей для сложных проектов» посвящена модулям и пакетам Python.

Глава 3 «Расширенное объектно-ориентированное программирование на Python» описывает, как можно реализовать продвинутые концепции объектно-ориентированного программирования с помощью Python.

Глава 4 «Библиотеки Python для продвинутого программирования» познакомит с такими концепциями, как итераторы, генераторы, обработка ошибок и исключений, обработка файлов и ведение журналов в Python.

Глава 5 «Тестирование и автоматизация с помощью Python» помогает понять разные типы автоматизации тестов, например, модульное, интеграционное и системное тестирование, а также как реализовать модульные тесты с помощью популярных фреймворков.

Глава 6 «Дополнительные советы и приемы Python» посвящена расширенным возможностям языка по преобразованию данных, созданию декораторов, а также использованию структур данных, включая *pandas DataFrame*, для аналитических приложений.

Глава 7 «Многопроцессорная обработка, многопоточность и асинхронное программирование» приводит рекомендации по созданию многопоточных и многопроцессорных приложений с помощью встроенных библиотек Python.

Глава 8 «Масштабирование Python с помощью кластеров» рассказывает о работе *Apache Spark* и написании приложений с использованием кластера *Apache Spark* для обработки больших данных.

Глава 9 «Программирование на Python для облака» описывает, как разрабатывать и развертывать приложения на облачной платформе, а также как использовать *Apache Beam* в целом и для *Google Cloud Platform* в частности.

Глава 10 «Использование Python для разработки веб-приложений и REST API» посвящена использованию среды *Flask* для разработки веб-приложений, взаимодействия с базами данных и создания *REST API* для веб-сервисов.

Глава 11 «Разработка микросервисов на Python» рассматривает общие понятия, а также использование фреймворка *Django* для создания примера микросервиса и интеграции его с микросервисом на базе *Flask*.

Глава 12 «Создание бессерверных функций на Python» описывает роль бессерверных функций в облачных вычислениях и способы их создания с помощью Python.

Глава 13 «Python и машинное обучение» поможет понять использование языка для создания, обучения и оценки моделей машинного обучения, а также их развертывания в облаке.

Глава 14 «Python для автоматизации сети» посвящена библиотекам Python для извлечения данных с сетевых устройств и систем управления сетью (**NMS**), а также для отправки конфигурации на устройства и NMS.

Как получить максимальную отдачу от книги

Для получения реальной пользы от этой книги необходимо предварительно изучить Python. Вам понадобится Python версии 3.7 или более поздней. Все примеры кода были протестированы в версиях 3.7 и 3.8 и должны работать с более поздними «*3.x*» версиями.

Аккаунт *Google Cloud Platform* (подойдет пробная бесплатная версия) будет полезен для развертывания некоторых примеров кода в облаке.

Таблица 1.1. Программное обеспечение и оборудование, описанное в книге

ПО и оборудование для книги	**Требования к операционной системе**
Python 3.7 или выше	Windows
Apache Spark 3.1.1	macOS
Cisco IOS XR 7.12 (сетевое устройство)	Linux
Nokia Network Services Platform (NSP) 21.6	
Google Cloud SDK 343.0.0	

Если вы читаете цифровую версию книги, мы советуем набирать код самостоятельно или использовать его из репозитория **GitHub** (ссылка доступна в следующем подразделе). Это поможет избежать ошибок, связанных с копированием и вставкой кода.

Загрузка файлов с примерами кода

Файлы с примерами кода можно скачать по адресу:

https://github.com/PacktPublishing/Python-for-Geeks.

Если выйдет обновление кода, оно будет отражаться в репозитории GitHub.

Мы также предлагаем другие пакеты из нашего обширного каталога книг и видео: *https://github.com/PacktPublishing/*.

Вы можете найти там что-то интересное.

Условные обозначения

В книге используется ряд текстовых условных обозначений и соглашений.

`Код в тексте`: указывает кодовые слова в тексте, имена таблиц баз данных, имена папок и файлов, расширения файлов, пути, фиктивные URL-адреса и дескрипторы Twitter. Пример: «Смонтируйте загруженный файл образа диска `WebStorm-10*.dmg` как еще один диск в системе».

Блок кода выглядит следующим образом:

```
resource = {
        "api_key": "AIzaSyDYKmm85kebxddKrGns4z0",
        "id": "0B8TxHW2Ci6dbckVwTRtTl3RUU",
        "fields": "files(name, id, webContentLink)",
}
```

Для привлечения вашего внимания к определенной части кода мы будем выделять строки **жирным шрифтом**:

```
#casestudy1.py: Pi calculator
from operator import add
from random import random

from pyspark.sql import SparkSession

spark = SparkSession.builder.
master("spark://192.168.64.2:7077") \
 .appName("Pi claculator app") \
 .getOrCreate()

partitions = 2
n = 10000000 * partitions

def func(_):
Preface xxi
    x = random() * 2 - 1
    y = random() * 2 - 1
    return 1 if x ** 2 + y ** 2 <= 1 else 0
count = spark.sparkContext.parallelize(range(1, n + 1),
 partitions).map(func).reduce(add)
print("Pi is roughly %f" % (4.0 * count / n))
```

Входные и выходные данные в командной строке:

```
Pi is roughly 3.141479
```

Жирным шрифтом и *курсивом* выделяются новые термины и важные слова. Например, пункты меню и сообщения в диалоговых окнах. Пример: «Как мы уже говорили, в Cloud Shell есть редактор, который можно запустить, нажав на кнопку **Open editor**».

> ***Советы и важные примечания***
>
> Выделены таким образом.

Раздел 1
Python помимо основ

Мы начинаем наше путешествие с изучения различных способов, как оптимально использовать Python как с точки зрения проектирования, так и с точки зрения реализации. Подробно рассмотрим жизненный цикл крупномасштабного проекта и его этапов. Изучим возможные способы, как создать элегантный дизайн с помощью модульности проекта. Везде, где это необходимо, заглянем «под капот» для лучшего понимания внутренностей Python. И в конце глубоко погрузимся в объектно-ориентированное программирование.

Этот раздел содержит следующие главы:

- «Глава 1: Оптимальный жизненный цикл разработки на Python».
- «Глава 2: Использование модулей для сложных проектов».
- «Глава 3: Расширенное объектно-ориентированное программирование на Python».

1

Оптимальный жизненный цикл разработки на Python

Эта книга предназначена для разработчиков, у которых уже есть опыт работы с Python, поэтому самые азы мы опустим. Начнем с нескольких слов о сообществе разработчиков и их уникальной культуре, которая отражается в коде. Затем поговорим об этапах стандартного проекта и посмотрим, как сформировать стратегию для его разработки.

После познакомимся со способами документирования кода и эффективными схемами именования, которые упрощают его обслуживание. Также рассмотрим системы контроля версий в проектах Python, в том числе **Jupyter Notebook**. В конце обсудим рекомендации по развертыванию кода после разработки и тестирования.

Темы этой главы:

- Культура и сообщество Python.
- Этапы проекта Python.
- Стратегия процесса разработки.
- Эффективное документирование кода Python.
- Разработка эффективной схемы именования.
- Системы контроля версий.
- Понимание стратегий развертывания кода.
- Среды разработки Python.

Культура и сообщество Python

Python — это высокоуровневый интерпретируемый язык, созданный Гвидо ван Россумом в 1991 году. Язык имеет собственное сообщество программистов, которое

уделяет особое внимание написанию кода и придерживается своей уникальной философии разработки. Сегодня Python используется в самых разных отраслях — от образования до медицины, и в каждом проекте ощущается его характерная культура.

Разработчики считают, что код должен быть максимально простым. Если задачу можно решить несколькими способами, следует выбрать вариант, наиболее соответствующий соглашениям и философии Python. Настоящие фанаты даже создают *артефакты*, максимально соответствующие правилам языка, а нарушителей этих правил считают плохими программистами. В этой книге мы постараемся не отступать от философии Python.

Существует даже официальный документ, написанный Тимом Питерсом, — «*Дзен Python*» (*The Zen of Python*), который призывает поддерживать принципы языка. В нем сказано, что Python должен быть максимально аккуратным, понятным и хорошо задокументированным. Прочесть его можно, выполнив в консоли команду `import this` (рис. 1.1):

```
[1] import this
    this

    The Zen of Python, by Tim Peters

    Beautiful is better than ugly.
    Explicit is better than implicit.
    Simple is better than complex.
    Complex is better than complicated.
    Flat is better than nested.
    Sparse is better than dense.
    Readability counts.
    Special cases aren't special enough to break the rules.
    Although practicality beats purity.
    Errors should never pass silently.
    Unless explicitly silenced.
    In the face of ambiguity, refuse the temptation to guess.
    There should be one-- and preferably only one --obvious way to do it.
    Although that way may not be obvious at first unless you're Dutch.
    Now is better than never.
    Although never is often better than *right* now.
    If the implementation is hard to explain, it's a bad idea.
    If the implementation is easy to explain, it may be a good idea.
    Namespaces are one honking great idea -- let's do more of those!
    <module 'this' from '/usr/lib/python3.6/this.py'>
```

Рис. 1.1. Дзен Python

Красивое лучше, чем уродливое.

Явное лучше, чем неявное.

Простое лучше, чем сложное.

Многосоставное лучше, чем запутанное.

Последовательное лучше, чем вложенное.

Умеренное лучше, чем нагроможденное.

Читаемость имеет значение.

Особые случаи не настолько особые, чтобы нарушать правила.

При этом практичность важнее безупречности.

Ошибки никогда не должны замалчиваться.

Если они не замалчиваются явно.

Встретив двусмысленность, не пытайся угадать.

Должен существовать, предпочтительно, только один очевидный способ решить задачу.

Хотя он поначалу может быть и не очевиден.

Сейчас лучше, чем никогда.

*Хотя никогда зачастую лучше, чем *прямо* сейчас.*

Если реализацию сложно объяснить, то идея плохая.

Если реализацию легко объяснить, то идея, возможно, хорошая.

Пространство имен — отличная штука! Будем делать его больше!

Текст выглядит загадочно, будто выбит на стене в гробнице фараона, но это сделано намеренно. При этом каждая строчка имеет глубокое значение. Мы будем ссылаться на «*Дзен Python*» на протяжении всей книги, поэтому обсудим несколько отрывков:

- «*Красивое лучше, чем уродливое*»: важно, чтобы код был хорошо написан, читаем и не требовал разъяснений; он должен не только отлично работать, но и красиво выглядеть; не следует использовать кратчайший путь при написании кода, это может затруднить его чтение.
- *«Простое лучше, чем сложное»*: не нужно усложнять; если есть выбор, следует использовать более простой вариант; лишних сложностей нужно избегать; если для большего удобства необходимо удлинить код, лучше предпочесть этот вариант.
- *«Должен существовать, предпочтительно, только один очевидный способ решить задачу»*: у проблемы должно быть только одно *наилучшее* решение, которое нужно постараться найти; работая над структурой кода, стоит стремиться к такому решению.
- *«Сейчас лучше, чем никогда»*: не стоит ждать идеальных условий; решать проблему следует *сейчас*, опираясь на имеющуюся информацию, предположения, навыки, инструменты и инфраструктуру; после уже можно улучшать решение; не нужно ждать подходящего момента, он может никогда не наступить.
- *«Явное лучше, чем неявное»*: код не должен требовать пояснений; к выбору имен переменных, классов и структуры функций, а также к созданию общей архитек-

туры следует подходить разумно; здесь следует проявить чрезмерную бдительность и сделать код максимально понятным.

- *«Последовательное лучше, чем вложенное»*: вложенная структура занимает меньше места, но затрудняет чтение; по возможности используйте последовательные структуры.

Этапы проекта Python

Для начала определим основные этапы проекта. Каждый из них включает несколько схожих действий, как показано на схеме (рис. 1.2):

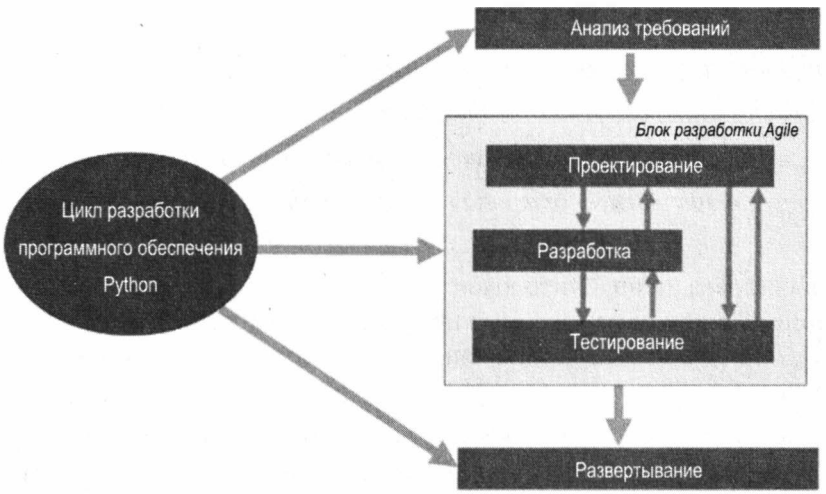


Рис. 1.2. Этапы проекта Python

Стандартный проект Python состоит из следующих этапов:

1. **Анализ требований:** на этом этапе мы общаемся с ключевыми заинтересованными лицами и анализируем их требования. Важно понять, *что* нужно делать, прежде чем решить, *как* это делать. Такими лицами могут быть пользователи или руководители компании. Необходимо наиболее подробно записать все требования, изучить их и обсудить с конечными пользователями перед следующим этапом; анализ требований не входит в цикл **проектирования**, **разработки** и **тестирования**, его нужно завершить до этих этапов. Важны как *функциональные*, так и *нефункциональные* требования. Первые следует разделить на модули, которые должны максимально соответствовать будущим модулям кода.
2. **Проектирование:** это техническое решение в ответ на требования из предыдущего этапа. Здесь идет обдумывание, *как* мы будем создавать наше решение. Это творческий процесс, в ходе которого используются опыт и навыки для создания подходящего набора и структуры модулей и оптимальных связей между

ними. На этом этапе особенно важно не допускать ошибок. Любые неверные решения на этом этапе обходятся дороже, чем на более поздних этапах. На исправление ошибок потребуется в 20 раз больше усилий, чем на устранение ошибок аналогичного масштаба на этапе *написания кода* (разработка). Например, последствия при неверном подборе подходящих данных или неправильно рассчитанном масштабе проекта будут более серьезными, по сравнению с ошибками при реализации функции. Проектирование часто лежит в более абстрактной плоскости идей. Ошибки здесь не так очевидны, их невозможно обнаружить в ходе тестирования, но можно легко перехватывать с помощью хорошо продуманной системы обработки исключений.

3. На этапе *проектирования* мы:
   - создаем структуру кода и определяем его модули;
   - выбираем фундаментальный подход (функциональное программирование, объектно-ориентированное программирование или их гибрид);
   - определяем классы и функции и выбираем имена для этих высокоуровневых компонентов;
4. **Разработка** (Написание кода)**:** здесь мы воплощаем проект на Python, начиная с высокоуровневых абстракций, компонентов и модулей, а затем переходя к деталям.
5. **Тестирование:** на этом этапе выполняется проверка, правильно ли работает код.
6. **Развертывание:** после тщательного тестирования мы предоставляем наше решение конечному пользователю, которому неважно, чем мы занимались на предыдущих этапах. Он просто решает свои задачи, описанные в требованиях. Допустим, мы работаем над проектом по машинному обучению, цель которого — прогнозировать осадки в Оттаве, тогда в ходе развертывания мы пытаемся определить, как предоставить пользователю удобное решение.

Теперь вы понимаете, из каких этапов состоит проект, и мы можем перейти к стратегическим аспектам.

Стратегия процесса разработки

Стратегическое планирование разработки охватывает каждый этап и переходы между ними. Для составления плана мы должны ответить на следующие вопросы:

1. Собираемся ли мы использовать подход с минимальным проектированием и сразу перейти к написанию кода?
2. Выберем ли мы *разработку через тестирование* (**Test-Driven Development, TDD**), где сначала пишутся тесты на основе требований, а затем создается код?
3. Хотим ли мы сначала создать *минимально жизнеспособный продукт* (**Minimum Viable Product, MVP**), а затем постепенно развивать его?

4. Какой стратегии мы будем придерживаться при проверке таких нефункциональных требований, как безопасность и производительность?
5. Будем ли мы разрабатывать под конкретные устройства или же развернем целый кластер или облако?
6. Какими будут объем, скорость и типы *входных* и *выходных* данных? Будем использовать *распределенную файловую систему Hadoop* (**Hadoop distributed file system, HDFS**) или файловую структуру **Simple Storage Service (S3)** от **Amazon**? Базу данных будем использовать **SQL** или **NoSQL**? Данные будут храниться локально или в облаке?
7. Работаем ли мы над такими специализированными вариантами использования, как *машинное обучение* (**Machine Learning**) с особыми требованиями к созданию конвейеров данных и моделей для тестирования, развертывания и обслуживания?

Ответы на эти вопросы помогут определить этапы процесса разработки. В последнее время предпочтительно использовать процессы *итеративного* подхода в том или ином виде. Также на старте популярна концепция MVP. Обо всем этом мы поговорим в следующих подразделах.

Итерация по этапам

Современный подход к разработке предусматривает короткие циклы проектирования, разработки и тестирования. Традиционная *каскадная модель* уже давно мертва. Выбор правильных детализации, акцента и частоты этапов зависит от характера проекта и избранной стратегии разработки. Если мы хотим выбрать стратегию с минимальной разработкой и сразу перейти к написанию кода, этап проектирования будет коротким. Но даже в этом случае необходимо обдумать структуру будущих модулей.

Независимо от выбора стратегии этапы связаны между собой. Мы начинаем с проектирования, реализуем план в коде, затем тестируем его. Отметив все недостатки, мы возвращаемся на этап проектирования.

Стремление к MVP в первую очередь

Рекомендуется отбирать только самые важные требования для создания минимально жизнеспособного продукта и постепенного его улучшения. Циклы проектирования, написания кода и тестирования повторяются снова и снова, пока не будет готов итоговый продукт, который можно развернуть и использовать.

Рассмотрим, как реализовать решение некоторых специализированных предметных областей на Python.

Стратегия разработки для специализированных предметных областей

Python используется в самых разных сценариях. Рассмотрим пять важных областей, где он применяется, и узнаем, как составлять стратегию разработки в зависимости от специфики каждого из них:

- машинное обучение;
- облачные и кластерные вычисления;
- системное программирование;
- сетевое программирование;
- бессерверные вычисления.

Рассмотрим каждый вариант поподробнее.

Машинное обучение

На сегодняшний день Python — самый популярный язык для написания алгоритмов машинного обучения, где требуется хорошо структурированная среда. Язык имеет обширную коллекцию высококачественных библиотек для реализации ML.

В стандартном проекте обычно используется методология **CRISP-DM (Cross-Industry Standard Process for Data Mining)** — стандарт, описывающий общие процессы и подходы к аналитике данных. Схема жизненного цикла CRISP-DM выглядит следующим образом (рис. 1.3):

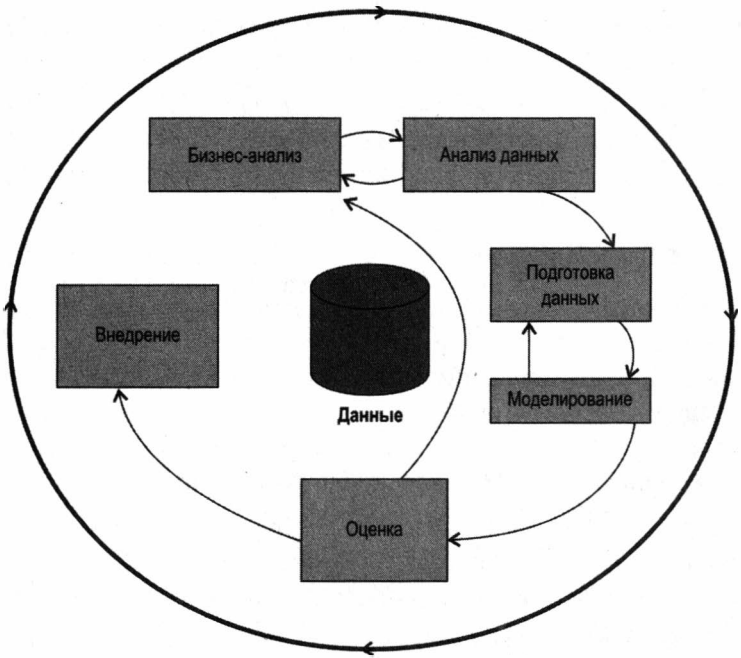


Рис. 1.3. Жизненный цикл CRISP-DM

В проектах ML проектирование и реализация конвейеров обработки данных занимают почти 70% процесса разработки, поэтому нужно стремиться к следующему:

- Их можно легко масштабировать.
- Их можно использовать многократно.
- Они поддерживают потоковую и пакетную обработку в соответствии со стандартами **Apache Beam**.
- Они сочетают в себе функции `fit()` и `transform()` (подробнее об этом в главе 6 *«Расширенные советы и приемы Python»*).

Оценка модели машинного обучения — важная часть этапа тестирования. Важно выбрать подходящие метрики производительности для верной оценки модели в соответствии с задачей, характером данных и типом реализуемого алгоритма. Что нам важнее: правильность, точность, полнота, показатель **F1-score** или комбинация этих метрик? Помимо стандартных тестов обязательно проводится оценка модели.

Облачные и кластерные вычисления

Облака и кластеры усложняют инфраструктуру тем, что требуют подключения специализированных библиотек. Архитектура языка позволяет начать с минимального количества базовых пакетов, а затем импортировать дополнительные, что имеет решающее значение и отлично подходит для облачных и кластерных вычислений. Python — предпочтительный язык для **Amazon Web Services (AWS)**, **Microsoft Azure** и **Google Cloud Platform (GCP)**.

Проекты в этой области имеют отдельные среды разработки, тестирования и производства. Важно синхронизировать их между собой.

При использовании модели *инфраструктура как услуга* (**infrastructure-as-a-service, IaaS**) контейнеры **Docker** могут сильно помочь — код можно запускать где угодно, и он везде будет иметь одинаковые среду выполнения и зависимости.

Системное программирование

В Python есть интерфейсы для служб операционной системы (ОС). Основные библиотеки имеют привязки к **Portable Operating System Interface (POSIX)**, что позволяет разработчикам создавать так называемые *инструменты оболочки* и использовать их для системного администрирования и различных утилит. Такие инструменты, написанные на Python, совместимы на разных платформах. Один и тот же инструмент можно использовать на Linux, Windows и macOS без изменений.

Например, инструмент оболочки, который копирует полный каталог, разработанный и протестированный под Linux, будет без изменений работать в Windows. Поддержка Python для системного программирования включает следующее:

- Определение переменных среды.
- Поддержка файлов, сокетов, каналов, процессов и многопоточности.

- Возможность использовать *регулярные выражения* (**Regular Expression**, **regex**) для сопоставления по шаблону.
- Возможность предоставлять аргументы командной строки.
- Поддержка стандартных потоковых интерфейсов, запуска shell-команд и расширений файлов.
- Возможность создавать ZIP-архивы для файловых утилит.
- Поддержка парсинга файлов XML и JSON.

Когда для системного программирования используется Python, этап развертывания будет минимальным. Код просто упаковывается в исполняемый файл. Кстати, язык не подходит для разработки системных драйверов или библиотек операционной системы.

Сетевое программирование

В эпоху автоматизации систем информационных технологий самым узким местом считаются сети. Проблема кроется в использовании собственных сетевых операционных систем от разных производителей и недостаточная их открытость. Предпосылки цифровой трансформации меняют эту тенденцию. Ведутся масштабные работы по изменению сети с целью сделать ее легко программируемой и *потребляемой как услуга* (**network-as-a-service**, **NaaS**). Главный вопрос: можно ли использовать Python для сетевого программирования? Разумеется, ДА. Фактически это один из самых популярных языков для автоматизации сетей.

Поддержка Python для сетевого программирования включает следующее:

- Программирование сокетов, включая **TCP** и **UDP**.
- Поддержка связи *«клиент-сервер»*.
- Поддержка прослушивания портов и обработки данных.
- Выполнение команд в системе **Secure Shell** (**SSH**).
- Загрузка и выгрузка файлов по протоколам **SCP** и **FTP**.
- Поддержка библиотек для работы с **SNMP**.
- Поддержка протоколов **RESTCONF** и **NETCONF** для извлечения и обновления конфигурации сетевых устройств.

Бессерверные вычисления

Это облачная модель выполнения приложений, в которой *поставщики облачных сервисов* (**Cloud Service Provider**, **CSP**) предоставляют вычислительные ресурсы и серверы приложений. Таким образом, разработчики могут развертывать и выполнять код без проблем, связанных с управлением этими вычислительными ресурсами и серверами. Все основные поставщики (**Microsoft Azure Serverless Functions**, **AWS Lambda** и **Google Cloud Platform**) поддерживают бессерверные вычисления для Python.

Важно понимать, в таких вычислениях все же есть серверы, но управляют ими CSP. Разработчики не занимаются установкой и обслуживанием серверов и напрямую не отвечают за их масштабируемость и производительность.

Для Python существуют следующие популярные бессерверные библиотеки и фреймворки:

- **Serverless**: фреймворк с открытым кодом для бессерверных функций или сервисов AWS Lambda, написанный на Node.js; это первый фреймворк, разработанный для создания приложений на AWS Lambda.
- **Chalice**: бессерверный микрофреймфорк для Python, разработанный AWS; лучший вариант для всех, кто хочет через AWS Lambda быстро развертывать приложения, автоматически меняющие масштаб по необходимости; имеет ключевую особенность — утилиту для локального моделирования приложения до отправки его в облако.
- **Zappa**: это инструмент развертывания, встроенный в Python, который упрощает развертывание **WSGI**-приложений.

Далее рассмотрим эффективные способы разработки кода Python.

Эффективное документирование кода Python

Найти эффективный способ документирования кода всегда важно. Задача состоит в составлении всеобъемлющего, но простого способа разработки кода. Сначала рассмотрим комментарии в Python, а потом и строки документации.

Комментарии Python

В отличие от строк документации, комментарии не видны компилятору. Они нужны для пояснения кода. Комментарий в Python начинается с символа «#», как показано на снимке (рис. 1.4):

```
# Простой пример для демонстрации комментариев.
print ("This is Chapter 1")

This is Chapter 1
```

Рис. 1.4. Пример комментария в Python

Docstring

Основной способ документирования кода — многострочный блок комментариев, именуемый **Docstring**. Особенность языка в том, что строки документации связаны с объектом и доступны для проверки. Рекомендации по docstring описаны в *Предложениях по улучшению Python* **(Python Enhancement Proposal, PEP)** *257*, и их

цель — дать читателю понимание, что они должны быть краткими, но достаточно подробными. Строки docstring обозначаются тремя двойными кавычками (""").

Некоторые рекомендации по созданию docstring:

- Размещать его следует сразу после определения функции или класса.
- Сначала лучше написать однострочное пояснение, а затем более подробное описание.
- Пустые места следует умело использовать для организации комментариев, а пустые строки для организации кода, но не следует ими злоупотреблять.

В следующих разделах подробнее поговорим о docstring.

Стили docstring

В Python есть несколько стилей оформления:

- Google.
- NumPy/SciPy.
- Epytext.
- Restructured.

Типы docstring

При разработке кода мы создаем разные типы документации, в том числе:

- построчные комментарии;
- документация по функциям или классам;
- описание алгоритмов.

Рассмотрим каждый вариант подробнее.

Построчные комментарии

Один из простых вариантов использования docstring — написание комментария в несколько строк (рис. 1.5):

```
"""
Это комментарий
написанный,
более чем в одну строку
"""
print ("Chapter 2 will be about Python Modules")

Chapter 2 will be about Python Modules
```

Рис 1.5. Пример построчного комментария

Документация на уровне функций или классов

Если docstring поместить сразу после определения функции или класса, Python свяжет документирование именно с ними. Комментарий будет храниться в атрибуте `__doc__` этой функции или класса, и его можно просмотреть непосредственно во время выполнения программы с помощью атрибута `__doc__` или функции `help`, как показано в примере (рис. 1.6):

```
[5] def double(n):
        '''Takes in a number n, returns the double of its value'''
        return n**2

[6] print(double.__doc__)

    Takes in a number n, returns the double of its value

[8] help(double)

    Help on function double in module __main__:

    double(n)
        Takes in a number n, returns the double of its value
```

Рис. 1.6. Пример функции `help`

```
class ComplexNumber
    """
    Это класс для математических операций над комплексными числами

    Атрибуты:
        real (тип int): вещественная часть комплексного числа.
        imag (тип int): мнимая часть вещественного числа.
    """

    def __init__(self, real, imag):
        """
        The constructor for ComplexNumber class.

        Параметры:
            real (тип int): вещественная часть комплексного числа.
            imag (тип int): мнимая часть вещественного числа.
        """

    def add(self, num):
        """
        Функция добавления двух комплексных чисел.

        Параметры:
            num (класс ComplexNumber): добавляемое комплексное число.

        Возвращает:
            ComplexNumber: комплексное число , которое содержит сумму.
        """

        re = self.real + num.real
        im = self.imag + num.imag

        return ComplexNumber (re, im)
```

Рис. 1.7. Пример docstring для класса

Для документирования классов рекомендуется придерживаться следующей структуры:

- Краткое описание, желательно в одну строку.
- Первая пустая строка.
- Подробное описание.
- Вторая пустая строка.

Пример показан на рис. 1.7.

Детали алгоритмов

В Python все чаще используется *описательная* или *прогностическая* аналитика и прочая сложная логика. Детали используемого алгоритма должны быть четко указаны со всеми предположениями. Если алгоритм реализован как функция, его логику лучше описать до сигнатуры этой функции.

Разработка эффективной схемы именования

Если разработать и реализовать правильную логику в коде — это наука, то сделать ее красивой и читаемой — это искусство. Разработчики уделяют особое внимание схеме наименования и соблюдению философии Дзен Python. Это один из немногих языков, для которых существуют подробные рекомендации в этой области, написанные самим Гвидо ван Россумом. Они описаны в документе *PEP 8*, где есть целый раздел про соглашения об именовании, за которым следует множество баз кода. Об этом можно узнать подробнее по ссылке *https://www.Python.org/dev/peps/pep-0008/*.

Общие принципы именования, описанные в PEP 8:

- Имена модулей записываются в `нижнем_регистре`.
- Имена классов и исключений записываются в `ВерблюжьемРегистре`.
- Глобальные и локальные переменные записываются в `нижнем_регистре`.
- Имена функций и методов записываются в `нижнем_регистре`.
- Константы записываются в `ВЕРХНЕМ_РЕГИСТРЕ`.

Некоторые рекомендации по структуре кода из PEP 8:

- Отступы важны. используйте для них четыре пробела вместо табуляции (*TAB*).
- Вложение должно быть не глубже четырех уровней.
- Строка не должна превышать 79 символов. длинные строки можно разделять символом «\».
- Для лучшей читаемости кода функции следует разделять двумя пустыми строками.
- Между логическими частями кода лучше вставлять пустую строку.

Данные рекомендации — это просто предложения, которые можно адаптировать. При этом схема именования все же должна опираться на принципы PEP 8 в качестве основного документа.

Рассмотрим подробнее различные схемы именования в контексте языковых структур Python.

Методы

В именах методов следует использовать символы в нижнем регистре. Имя должно состоять из одного или нескольких слов, разделенных символом подчеркивания. Пример:

```
calculate_sum
```

Для простоты читаемости кода выбирать для имени метода следует глагол, который указывает на действие, выполняемое методом.

Если это метод с доступом, отличным от *public*, его имя начинается с символа подчеркивания. Пример:

```
_my_calculate_sum
```

Dunder (**Double Under**) или *магические методы* (**Magic Methods**) начинаются и заканчиваются символом подчеркивания. Пример:

- `__init__`;
- `__add__`.

Два символа подчеркивания не рекомендуется указывать в начале или в конце имени метода. Такая схема именования предназначена для встроенных методов Python.

Переменные

Имена переменных записываются в нижнем регистре и состоят из одного или нескольких слов, разделенных символом подчеркивания. Это должно быть существительное, соответствующее сущности, которую оно представляет. Пример:

- `x`;
- `my_var`.

Имена переменных с доступом *private* начинаются с символа подчеркивания. Пример:

```
_my_secret_variable
```

Булевы переменные

Булевы переменные, которые начинаются со слова `is` или `has`, проще воспринимаются. Пример:

```
Class Patient:
    Is_admitted = False
    Has_heartbeat = False
```

Коллекции

Коллекции (или наборы) содержат несколько переменных, поэтому лучше давать им имя во множественном числе. Пример:

```
Class Patient:
    Admitted_patients = ['John', 'Peter']
```

Словари

Имя словаря должно быть максимально явным. Если есть словарь людей, привязанных к городам, в этом случае его лучше назвать следующим образом:

```
persons_cities = {'Imran': 'Ottawa', 'Steven': 'Los Angeles'}
```

Константы

В Python нет неизменяемых переменных. Например, в C++ ключевое слово `const` указывает, что переменная представляет собой константу. В Python переменные указываются как константы по соглашению об именовании. При обработке компилятор не выдаст ошибку.

Для констант рекомендуется использовать слова в верхнем регистре, разделенные символом подчеркивания. Пример:

```
CONVERSION_FACTOR
```

Классы

Классы обозначаются в стиле *ВерблюжьегоРегистра* (**CamelCase**). Имя начинается с заглавной буквы. Если название содержит больше одного слова, их следует писать слитно и каждое нужно начинать с заглавной буквы.

В имени следует использовать существительные. Оно должно наилучшим образом представлять сущность, которой класс соответствует. Одна из рекомендаций — добавлять дополнительные слова для уточнения типа или характера сущности. Пример:

- `HadoopEngine`;
- `ParquetType`;
- `TextboxWidget`.

О чем еще нужно помнить:

- Классы исключений для обработки ошибок всегда заканчиваются словом `Error`, например:

  ```
  FileNotFoundError
  ```

- Некоторые встроенные классы Python не следуют этим рекомендациям.

- Для удобства восприятия следует указывать префиксы `Base` и `Abstract` для базовых и абстрактных классов соответственно, например:

```
AbstractCar
BaseClass
```

Пакеты

В имени пакета не рекомендуется указывать символ подчеркивания. Название должно быть коротким и в нижнем регистре. Если используется больше одного слова, они также записываются строчными буквами. Пример:

```
Mypackage
```

Модули

Имя модуля должно быть коротким и понятным. Оно записывается в нижнем регистре, слова разделяются символом подчеркивания. Пример:

```
main_module.py
```

Соглашения об импорте

С годами сообщество Python выработало соглашение для псевдонимов, которые используются для популярных пакетов. Пример:

```
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels as sm
import matplotlib.pyplot as plt
```

Аргументы

Имена аргументов функций рекомендуется называть, как и переменные, поскольку они по своей сути и есть временные переменные.

Полезные инструменты

Есть пара инструментов, которыми можно проверить, насколько код соответствует рекомендациям PEP 8. Рассмотрим их.

Pylint

Устанавливается следующей командой:

```
pip: $ pip install pylint
```

Это анализатор исходного кода, который проверяет соблюдение рекомендаций PEP 8 при наименованиях и выводит отчет. Его можно настроить и под другие соглашения.

PEP 8

Устанавливается следующей командой:

```
pip: $ pip install pep8
```

Утилита проверяет код на соблюдение правил PEP 8.

Далее поговорим о системах контроля версий для Python.

Системы контроля версий

Сначала кратко познакомимся с историей систем контроля. Их эволюция прошла несколько этапов развития:

- **Этап 1**: исходный код изначально запускался локальными системами, хранившимися на жестком диске; такая коллекция кода называлась локальным репозиторием.
- **Этап 2**: в больших командах невозможно было использовать локальную систему контроля версий; разработчики постепенно перешли на централизованные репозитории, хранившиеся на сервере, к которому был доступ у всей команды; это решило трудности совместного использования кода, но создало дополнительную проблему блокировки файлов в многопользовательской среде.
- **Этап 3**: эти проблемы решены в современных репозиториях, наподобие **Git**; члены команды имеют полную копию, и им необходимо подключаться к репозиторию, только когда есть необходимость поделиться кодом.

Что не стоит хранить в репозитории системы контроля версий

Во-первых, ничего, кроме файла исходного кода, регистрировать в системе не нужно. Файлы, сгенерированные компьютером, сохранять не стоит. Предположим, есть исходный файл `main.py`. Если его скомпилировать, сгенерированный код не принадлежит репозиторию. Этот код является производным файлом и не должен регистрироваться системой контроля версий. На это есть три причины:

- Любой член команды может создать производный файл после получения исходного кода.
- Во многих случаях скомпилированный код больше исходного, его добавление сделает репозиторий медленным; если в команде много разработчиков, каждый из них получит копию сгенерированного файла, что очень замедлит работу всей системы.

- Системы контроля версий предназначены для хранения *дельты* (разницы) или *изменений*, внесенных разработчиками в исходные файлы с момента последнего *коммита* (**commit**); файлы, отличные от исходных, обычно имеют двоичный формат, и в системе вряд ли есть инструмент для их сравнения (`diff tool`), так что ей придется сохранять файл целиком; это также замедлит работу всей системы.

Во-вторых, в системе нельзя хранить конфиденциальную информацию, в том числе ключи API и пароли.

В качестве исходного репозитория сообщество Python предпочитают **GitHub.** Там находится большая часть известных пакетов Python. Если команд разработчиков несколько, необходимо разработать и поддерживать правильный протокол и процедуры взаимодействия.

Понимание стратегий развертывания кода

Для крупномасштабных проектов, где имеется вполне определенные **DEV-** или **PROD-**среды, развертывание кода и разработка стратегии становятся важными.

Python является наиболее предпочтительным языком для облачных и кластерных вычислений. При развертывании кода может возникнуть несколько сложностей:

- В точности одинаковые преобразования должны происходить в DEV-, TEST- и PROD-средах.
- Если код постоянно обновляется в DEV-среде, возникает сложность, как синхронизировать эти изменения с PROD-средой.
- Какой тип тестирования планируется проводить в средах DEV и PROD?

Рассмотрим две главные стратегии развертывания кода.

Пакетная разработка

Эта традиционный процесс разработки. Мы пишем код, компилируем его, а затем тестируем. Процесс повторяется, пока все требования не будут выполнены. Затем происходит развертывание кода.

Непрерывная интеграция и непрерывная доставка

Непрерывная интеграция и непрерывная доставка (**Continuous integration/ Continuous delivery**, **CI/CD**) в контексте Python относится к непрерывной интеграции и развертыванию, а не к их пакетному процессу. Это помогает создать среду **DevOps** (**Development-Operations**), устраняя разрыв между разработкой и эксплуатацией программного обеспечения.

CI — это непрерывная интеграция, сборка и тестирование разных модулей кода по мере их обновления. Это означает, что код, разработанный индивидуально каждым членом команды, интегрируется, создается и тестируется, как правило, много раз в день. После этого репозиторий в системе контроля версий обновляется.

Главное преимущество CI — проблемы и ошибки обнаруживаются в самом начале, обычно в тот же день. Их можно поправить сразу, а не через несколько дней, недель или даже месяцев, когда они просочились в другие модули, а те, в свою очередь, создали множество зависимостей.

В отличие от Java и C++, Python является *интерпретируемым* языком, а значит, код выполняется интерпретатором на любой целевой машине. Для сравнения, *компилируемый* код обычно пишется для одного типа целевых машин и может разрабатываться разными членами команды. Когда будет понятно, *какие* действия выполняются при внесении изменения, их можно автоматизировать.

Код Python зависит от внешних пакетов, поэтому отслеживание их имен и версий является частью автоматизации процесса сборки. Хорошей практикой будет указывать эти пакеты в файле `requirements.txt`. Имя может быть любым, но сообщество разработчиков называет его так.

Пакеты устанавливаются следующей командой:

```
$pip install -r requirements.txt
```

Создание файла `requirements.txt` выполняется следующей командой:

```
$pip freeze > requirements.txt
```

Цель непрерывной интеграции — обнаружить ошибки на ранних стадиях, но она может сделать процесс разработки нестабильным. Допустим, один из программистов внес серьезную ошибку, остальной команде приходится ждать, пока ошибка будет устранена. Надежное самотестирование и выбор правильной частоты интеграций помогут решить проблему. Тестировать следует каждое внесенное изменение, причем этот процесс в конечном итоге должен быть автоматизирован. При наличии в коде ошибки сборка завершится сбоем, а член команды получит уведомление. Он сможет сначала предоставить быстрое решение, прежде чем тратить время на устранение и полное тестирование проблемы, и его коллеги не будут заблокированы.

Когда новый код создан и протестирован, его можно обновить с помощью непрерывной доставки — CD. В полном конвейере CI/CD каждое изменение собирается и тестируется, а после отображается в развернутом коде. При правильном подходе конечное решение будет постоянно улучшаться. В некоторых случаях каждый цикл CI/CD может быть постоянным переходом от MVP к полному решению. Таким образом, решение адаптируется к меняющимся условиям с учетом новой информации. Хороший пример — пандемия **COVID-19**, когда новые данные поступают очень быстро. Если продукт зависит от подобных изменений, конвейер CI/CD принесет только пользу, поскольку разработчики смогут постоянно обновлять решение при поступлении новой информации.

Далее обсудим популярные среды разработки для Python.

Среды разработки Python

Для маленьких проектов достаточно использовать любой текстовый редактор. Для средних и крупных — лучшим выбором будет *интегрированная среда разработки* (**Integrated Development Environment**, **IDE**). Она помогает при написании кода, отладке и устранении ошибок, а также совместима с системой контроля версий для упрощения развертывания. На рынке существует множество решений, в основном бесплатных. Обратите внимание, что мы не будем составлять их рейтинг или ранжировать в каком-либо порядке, а просто опишем их преимущества. Таким образом, читатель сможет сделать лучший для себя выбор, основываясь на своем прошлом опыте, проектных требованиях и сложности проекта.

IDLE

Integrated Development and Learning Environment (**IDLE**) — стандартный редактор Python, доступный для Windows, macOS и Linux. Он поставляется бесплатно и хорошо подходит новичкам в целях обучения. Для сложных проектов не рекомендуется.

Sublime Text

Это популярный многоязычный редактор, который подходит для основных платформ (Windows, macOS, и Linux).

Бесплатно предоставляется только ознакомительная версия. Python поддерживается на базовом уровне. Но, благодаря возможности устанавливать различные плагины, можно создать полноценную среду разработки, правда, это требует времени и навыков. С помощью плагинов среду можно интегрировать с системой контроля версий, например, Git или Subversion (**SVN**), но не все функции могут быть доступны.

Atom

Также популярный редактор, похожий на Sublime Text. Предоставляется бесплатно.

PyCharm

Это одна из лучших интегрированных сред разработки для Python. Доступна в Windows, macOS и Linux. Это полноценная IDE с поддержкой автозавершения кода, отладки, рефакторинга, умного поиска, доступа к популярным серверам баз данных, интеграции с системами контроля версий и многих других возможностей. Базовые функции можно дополнять с помощью плагинов. PyCharm поставляется в следующих форматах:

- *Community Edition* — бесплатная версия для разработки на Python.

- *Professional Edition* — платная версия с поддержкой веб-разработки (HTML, JavaScript и SQL).

Visual Studio Code

VS Code — среда с открытым исходным кодом от Microsoft. Лучшая IDE для разработки на Python в Windows. По умолчанию поставляется без поддержки этого языка, но можно установить расширения.

Поставляется бесплатно и доступна также для macOS и Linux. Это легкая среда с множеством удобных функций: автозавершение кода, отладка, рефакторинг, поиск, доступ к серверам баз данных, интеграция с системами контроля версий и т.д.

PyDev

Если вы знакомы со средой *Eclipse*, на PyDev стоит обратить внимание. Он представляет собой плагин в виде редактора для Eclipse, который также можно использовать для Jython и IronPython. Предоставляется бесплатно и доступен для всех основных платформ. В нем есть все преимущества Eclipse, а также удобная интеграция с Django, модульное тестирование и **Google App Engine (GAE)**.

Spyder

Если вы планируете использовать язык для *data science* и машинного обучения, Spyder хорошо подойдёт в качестве IDE. Среда написана на Python и предлагает инструменты для редактирования, отладки, интерактивного выполнения, глубокой проверки и расширенной визуализации. Она также поддерживает интеграцию с Matplotlib, SciPy, NumPy, Pandas, Cython, IPython и SymPy.

Профессиональным разработчикам можно порекомендовать PyCharm или PyDev. Но если вы увлекаетесь data science и машинным обучением, Spyder, безусловно, стоит изучить.

Заключение

В этой главе мы заложили фундамент, на котором будут строиться последующие главы книги. Начали с духа и философии языка, затем обсудили этапы стандартного проекта и способы оптимизировать работу над ними в зависимости от варианта применения. Для такого лаконичного языка, как Python, особенно важна качественная документация, упрощающая понимание кода. Поэтому мы также рассмотрели практики ее ведения. Поговорили о схемах именования, рассмотрели системы контроля версий и узнали о способах развертывания кода. В заключение сравнили несколько сред разработки с целью помочь вам сделать выбор в зависимости от требований.

Эта глава будет полезна всем, кто начинает работу над проектом на Python. Она поможет разработать эффективную стратегию и принять взвешенные решения на этапе проектирования. В следующей главе мы поговорим, как разделить код в проекте на модули.

Вопросы

1. Что такое Дзен Python?
2. Какой тип документирования доступен в Python во время выполнения программы?
3. Как устроен жизненный цикл CRISP-DM?

Дополнительные ресурсы

- *Modern Python Cookbook*, второе издание, автор: Стивен Ф. Лотт (Steven F. Lott).
- *Python Programming Blueprints*, автор: Дэниел Фуртадо (Daniel Furtado).
- *Secret Recipes of the Python Ninja*, автор: Коди Джексон (Cody Jackson).

Ответы

1. Это рекомендации по разработке проектов на Python, написанных Тимом Питерсом.
2. В отличие от обычных комментариев, docstring доступен компилятору во время выполнения.
3. CRISP-DM расшифровывается как **Cross-Industry Standard Process for Data Mining**. Он применяется к жизненному циклу проектов Python в сфере машинного обучения и определяет различные этапы проекта.

2

Использование модулей для сложных проектов

Начинающие Python-программисты часто предпочитают помещать весь код в один файл, поскольку так не возникает проблем с определением функций и классов, они находятся вместе с основной программой. Такой способ привлекает новичков из-за простоты выполнения кода и возможности избежать управления несколькими файлами. Этот вариант плохо масштабируется и не подходит для средних и крупных проектов. С увеличением в программе функций и классов становится сложнее их отслеживать.

Решить проблему можно с помощью разделения на модули. Это ключевой инструмент для снижения сложности проекта. Модульность способствует повышению эффективности программирования, простой отладке, совместной работе и повторному использованию. В этой главе мы расскажем, как создавать и использовать модули и пакеты в Python.

Темы этой главы:

- Знакомство с модулями и пакетами.
- Импорт модулей.
- Загрузка и инициализация модуля.
- Написание многоразовых модулей.
- Сборка пакетов.
- Доступ к пакетам из любого расположения.
- Общий доступ к пакету.

Технические требования

В этой главе вам понадобится:

- Python 3.7 или более поздней версии.
- Аккаунт Test PyPI и токен API в этом аккаунте.

Пример кода для этой главы можно найти по ссылке:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter02.

Знакомство с модулями и пакетами

Модули представляют собой файлы Python с разрешением `.py`. Это способ организовать функции, классы и переменные с помощью одного или нескольких файлов для простоты управления, повторного использования и дополнения по мере усложнения программы.

Пакет — это следующий уровень модульного программирования. Он похож на папку для организации множества модулей или подпакетов, что имеет основополагающее значение для обмена пакетами с целью многократного использования.

Исходными файлами Python, в которых используются стандартные библиотеки, можно легко делиться через электронную почту, GitHub и общие диски. Но делать это можно с единственным предостережением — версии языка должны быть совместимы. Такой подход не работает в проектах с большим количеством файлов и зависимостями от сторонних библиотек, тем более если код написан для определенной версии языка. Для выхода из такой ситуации создание и совместное использование пакетов являются обязательными условиями для эффективного обмена кодом и многократного его использования.

Далее рассмотрим, как импортировать модули и какие способы импорта поддерживаются в Python.

Импорт модулей

Код в одном модуле может получить доступ к коду в другом модуле с помощью процесса, называемого *импортом модулей*.

Для наглядности создадим два модуля и один скрипт, который будет их использовать. Модули будут обновляться и повторно использоваться на протяжении всей главы.

Для этого создадим файл с расширением `.py` и именем модуля. В нашем примере это будет `mycalculator.py` с двумя функциями: `add()` и `subtract()`. Функция `add()` вычисляет сумму двух чисел, переданных ей в качестве аргументов, и возвращает результат. Функция `subtract()` вычисляет разность двух чисел, переданных ей в качестве аргументов, и возвращает результат.

Фрагмент кода `mycalculator.py`:

```
# mycalculator.py с функциями add и subtract
def add(x, y):
    """Эта функция складывает 2 числа"""
    return x + y
def subtract(x, y):
    """Эта функция вычисляет разность 2 чисел"""
    return x - y
```

Имя модуля совпадает с именем файла.

Затем создадим второй модуль, добавив файл с именем `myrandom.py`. В нем будут две функции: `random_1d()` и `random_2d()`. Функция `random_1d()` генерирует случайное число от 1 до 9, а `random_2d()` — от 10 до 99. Обратите внимание, этот модуль использует библиотеку `random`, которая является встроенным модулем Python.

Фрагмент кода из файла `myrandom.py`:

```
# myrandom.py с пользовательской и встроенной функциями random
import random
def random_1d():
    """Эта функция генерирует случайное число от 0 до 9"""
    return random.randint (0,9)
def random_2d():
    """ Эта функция генерирует случайное число от 10 до 99"""
    return random.randint (10,99)
```

Далее создадим главный скрипт `calcmain1.py`, который импортирует модули, описанные выше, и использует их для реализации функций калькулятора. Наиболее распространенный способ импортировать модули — оператор `import`.

Фрагмент кода `calcmain1.py`:

```
# calcmain1.py с функцией main()
import mycalculator
import myrandom
def my_main( ):
    """ Это главная функция, которая создает 2 случайных числа\
    и применяет к ним функции калькулятора """
    x = myrandom.random_2d( )
    y = myrandom.random_1d( )
    sum = mycalculator.add(x, y)
    diff = mycalculator.subtract(x, y)
Importing modules 35
    print("x = {}, y = {}".format(x, y))
    print("sum is {}".format(sum))
    print("diff is {}".format(diff))
```

```
""" Этот фрагмент выполняется только в случае, если специальная переменная\
    '__name__' установлена в качестве главной"""
if __name__ == "__main__":
    my_main()
```

С помощью оператора `import` мы импортируем оба модуля в главный скрипт. В качестве *основной* определена функция `my_main()`, которая будет выполняться при условии, что модуль `calcmain1` выполняется в качестве *главной* программы. Детали выполнения основной функции из тела главного скрипта будут рассмотрены в разделе *«Установка параметров для специальных переменных»*. В функции `my_main()` мы создаем два случайных числа с помощью модуля `myrandom`, а затем вычисляем сумму и разность этих чисел с помощью модуля `mycalculator`. В конце выводим результаты на консоль с помощью оператора `print`.

> ***ВАЖНОЕ ПРИМЕЧАНИЕ***
>
> Модуль подгружается только один раз. Если он импортирован другим модулем или главным скриптом, то будет инициализирован в процессе выполнения кода. Если другой модуль в программе повторно импортирует тот же модуль, он не будет загружен дважды. Это означает, что если внутри модуля есть локальные переменные, они будут действовать как *одиночки* (**Singleton**), то есть инициализироваться однократно.

Импортировать модуль можно и другими способами, например, с помощью `importlib.import_module()` и встроенной функции `__import__()`. Рассмотрим, как работает `import` и альтернативные варианты.

Оператор import

Как упоминалось ранее, `import` — распространенный способ подключить модуль. Пример использования :

```
import math
```

Оператор отвечает за два действия: во-первых, ищет модуль, указанный после ключевого слова `import`, во-вторых, привязывает результаты этого поиска к имени переменной (которое совпадает с именем модуля) в локальной области выполнения. В следующих двух подразделах мы поговорим, как работает этот оператор и как подгружать определенные элементы из модуля или пакета.

Как работает оператор import

Все глобальные переменные и функции добавляются в глобальное пространство имен в начале выполнения. Для примера напишем небольшую программу, которая выделяет содержимое пространства имен `globals`:

```
# globalmain.py с функцией globals()
def print_globals():
    print (globals())
```

```
def hello():
    print ("Hello")
if __name__ == "__main__":
    print_globals()
```

Программа содержит две функции: `print_globals()` и `hello()`. Первая выводит содержимое глобального пространства имен. А вторая выполняться не будет, она приводится здесь с целью показать ссылку на нее в консольном выводе. После выполнения кода вывод консоли будет следующим:

```
{
   "__name__":"__main__",
   "__doc__":"None",
   "__package__":"None",
   "__loader__":"<_frozen_importlib_external.\
    SourceFileLoader object at 0x101670208>",
   "__spec__":"None",
   "__annotations__":{
   },
   "__builtins__":"<module 'builtins' (built-in)>",
Importing modules 37
   "__file__":"/ PythonForGeeks/source_code/chapter2/\
     modules/globalmain.py",
   "__cached__":"None",
   "print_globals":"<function print_globals at \
     0x1016c4378>",
   "hello":"<function hello at 0x1016c4400>"
}
```

Ключевые моменты кода:

- Переменной `__name__` присваивается значение `__main__`; подробнее об этом мы поговорим в подразделе *«Загрузка и инициализация модуля»*.
- Переменной `__file__` задается путь к файлу главного модуля.
- Ссылки на каждую функцию добавляются в конце.

Если добавить `print(globals())` в скрипт `calcmain1.py`, вывод консоли будет выглядеть следующим образом:

```
{
   "__name__":"__main__",
   "__doc__":"None",
   "__package__":"None",
   "__loader__":"<_frozen_importlib_external.\
    SourceFileLoader object at 0x100de1208>",
   "__spec__":"None",
   "__annotations__":{},
   "__builtins__":"<module 'builtins' (built-in)>",
```

```
 "__file__":"/PythonForGeeks/source_code/chapter2/module1/
 main.py",
   "__cached__":"None",
   "mycalculator":"<module 'mycalculator' from \
    '/PythonForGeeks/source_code/chapter2/modules/\
    mycalculator.py'>",
   "myrandom":"<module 'myrandom' from '/PythonForGeeks/source_
 code/chapter2/modules/myrandom.py'>",
   "my_main":"<function my_main at 0x100e351e0>"
}
```

Обратите внимание на две дополнительные переменные (`mycalculator` и `myrandom`), добавленные в глобальное пространство имен. При каждом подключении библиотеки создается переменная с тем же именем, которая содержит ссылку на модуль, прямо как переменная для глобальных функций (в нашем примере это `my_main`).

Можно заметить, что в других подходах к импорту модулей есть возможность явно определять переменные для каждого модуля. Оператор `import` делает все это автоматически за нас.

Импорт отдельных частей модуля

Мы можем подгрузить не весь модуль, а только его часть (например, переменную, функцию или класс). Для этого используется оператор `from`:

```
from math import pi
```

Еще одной рекомендуемой практикой является использование другого имени для подключаемого модуля. Это делается для удобства или в случае, когда используются одинаковые имена для разных ресурсов в двух разных библиотеках. Для демонстрации внесем изменения в файл `calcmain1.py` (обновленной программой будет `calcmain2.py`), используя сокращенные обозначения `calc` и `rand` для модулей `mycalculator` и `myrandom` соответственно. Это изменение значительно упростит использование модулей в главном скрипте:

```
# calcmain2.py с сокращенными обозначениями для модулей
import mycalculator as calc
import myrandom as rand
def my_main():
    """ Это главная функция, которая создает 2 случайных числа\
        и применяет к ним функции калькулятора """
    x = rand.random_2d()
    y = rand.random_1d()
    sum = calc.add(x,y)
    diff = calc.subtract(x,y)
    print("x = {}, y = {}".format(x,y))
    print("sum is {}".format(sum))
    print("diff is {}".format(diff))
```

```
Importing modules 39
""" Этот фрагмент выполняется только в случае, если специальная переменная\
    '__name__' установлена в качестве главной"""
if __name__ == "__main__":
    my_main()
```

На следующем шаге объединим две концепции, о которых говорилось ранее, в новой версии программы (`calcmain3.py`). В этом обновлении мы используем оператор `from` с именами модулей, а затем импортируем отдельные функции из каждого модуля. В случае функций `add()` и `subtract()` применим оператор `as` для установки другого локального определения ресурса модуля.

Фрагмент кода из файла `calcmain3.py`:

```
# calcmain3.py с оператором from и сокращениями
from mycalculator import add as my_add
from mycalculator import subtract as my_subtract
from myrandom import random_2d, random_1d

def my_main():
    """ Это главная функция, которая создает 2 случайных числа\
        и применяет к ним функции калькулятора """
    x = random_2d()
    y = random_1d()
    sum =  my_add(x,y)
    diff = my_subtract(x,y)
    print("x = {}, y = {}".format(x,y))
    print("sum is {}".format(sum))
    print("diff is {}".format(diff))
    print(globals())
""" Этот фрагмент выполняется только в случае, если специальная переменная\
    '__name__' установлена в качестве главной"""
if __name__ == "__main__":
    my_main()
```

Поскольку мы использовали `print(globals())`, вывод программы на консоль покажет, что переменные, соответствующие каждой функции, созданы согласно обозначениям:

```
{
   "__name__":"__main__",
   "__doc__":"None",
   "__package__":"None",
   "__loader__":"<_frozen_importlib_external.\
    SourceFileLoader object at 0x1095f1208>",
   "__spec__":"None",
   "__annotations__":{},
   "__builtins__":"<module 'builtins' (built-in)>", "__
 file__":"/PythonForGeeks/source_code/chapter2/module1/
```

```
 main_2.py",
   "__cached__":"None",
   "my_add":"<function add at 0x109645400>",
   "my_subtract":"<function subtract at 0x109645598>",
   "random_2d":"<function random_2d at 0x10967a840>",
   "random_1d":"<function random_1d at 0x1096456a8>",
   "my_main":"<function my_main at 0x109645378>"
}
```

Обратите внимание, переменные, выделенные жирным шрифтом, соответствуют изменениям оператора `import` в файле `calcmain3.py`.

Оператор __import__

Оператор `__import__` представляет собой низкоуровневую функцию, которая принимает строку в качестве входных данных и запускает операцию импорта. Такие функции являются частью ядра Python и предназначены для разработки библиотек или доступа к ресурсам операционной системы. Для создания приложений они обычно не используются. Это ключевое слово можно использовать для подключения библиотеки `random` в модуль `myrandom.py`:

```
# импорт random
random = __import__('random')
```

Остальной код в `myrandom.py` можно оставить без изменений.

Этот метод приведен только для демонстрации, но вы можете самостоятельно изучить все подробности. Такой способ не рекомендуется использовать для пользовательских приложений, он разработан для *интерпретаторов*.

Если нужен расширенный функционал, который не дает обычная инструкция `import`, существует оператор `importlib.import_module`.

Инструмент importlib.import_module

Библиотека `importlib` дает возможность импортировать любые модули. Она предлагает разнообразие функций (включая `__import__`), связанных с более гибким подключением модулей. Простой пример, как импортировать модуль `random` в `myrandom.py` с помощью `importlib`:

```
import importlib
random = importlib.import_module('random')
```

Остальной код в `myrandom.py` можно оставить без изменений.

Модуль `importlib` хорошо известен благодаря динамическому импорту модулей. Он очень полезен, когда имя модуля заранее не известно и модули нужно подключать

во время выполнения. Это распространенное требование при разработке плагинов и расширений.

Наиболее популярны несколько следующих функций:

- `__import__`: реализация функции `__import__`, о которой мы уже говорили;
- `import_module`: используется чаще всего для динамического подключения модулей и позволяет указывать абсолютный или относительный путь к ним; является *оберткой* для `importlib.__import__`; обратите внимание, что первая функция возвращает пакет или модуль (например, `packageA.module1`), указанный в функции, а вторая — пакет или модуль верхнего уровня (например, `packageA`);
- `importlib.util.find_spec`: заменяет метод `find_loader`, устаревший с версии Python 3.4; используется для проверки, существует ли модуль и является ли он действующим;
- `invalidate_caches`: используется для очистки внутреннего кэша *искателей* (**finder**), хранящихся в `sys.meta_path`; внутренний кэш позволяет быстрее загружать модули без повторного запуска искателей; но если мы импортируем модуль динамически, особенно если он создан после запуска интерпретатора, хорошей практикой будет применить метод `invalidate_caches`; эта функция удалит из кэша все модули или библиотеки, и можно быть уверенным, что запрошенный модуль загружен из системного пути методом `import`;
- `reload`: повторно загружает ранее подключенный модуль; в качестве входного параметра этой функции необходимо предоставить объект модуля; это означает, что `import` должен быть выполнен успешно; функция очень полезна в ситуациях, когда исходный код модуля был изменен и нужно загрузить новую версию без перезапуска программы.

Абсолютный и относительный импорт

Абсолютный и *относительный* импорт особенно важен, когда есть необходимость подключения настраиваемых и проектно-специфичных модулей. Для демонстрации этих двух концептов возьмем проект с различными пакетами, подпакетами и модулями:

```
project
  ├── pkg1
  |   ├── module1.py
  |   └── module2.py (содержит функцию func1())
  └── pkg2
      ├── __init__.py
      ├── module3.py
      └── sub_pkg1
          └── module6.py (содержит функцию func2 ())
```

```
├── pkg3
│   ├── module4.py
│   ├── module5.py
│   └── sub_pkg2
│       └── module7.py
```

Используя эту структуру, обсудим, как использовать абсолютный и относительный импорт.

Абсолютный импорт

Абсолютный путь можно указать, начав с пакета верхнего уровня и двигаясь вниз по структуре. Несколько примеров того, как можно импортировать модули:

```
from pkg1 import module1
from pkg1.module2 import func1

from pkg2 import module3
from pkg2.sub_pkg1.module6 import func2

from pkg3 import module4, module5
from pkg3.sub_pkg2 import module7
```

При абсолютном импорте необходимо указать подробный путь к каждому пакету или файлу, начиная с пакета верхнего уровня. Это аналогично пути к файлу.

Такой способ наиболее предпочтителен, поскольку он легко читается и помогает отслеживать точное местоположение подключаемых ресурсов. На абсолютный импорт меньше всего влияют совместное использование проекта и изменения в текущем расположении операторов `import`. Фактически PEP 8 рекомендует использовать именно этот тип подключения.

Однако иногда абсолютный импорт представляет собой довольно длинные операторы, зависящие от размера структуры папок в проекте.

Относительный импорт

При относительном импорте указывается подключаемый ресурс относительно текущего расположения, а именно, расположения файла с кодом, где находится `import`.

Возвращаясь к примерам, рассмотрим несколько сценариев относительного импорта. Эквивалентные операторы выглядят следующим образом:

- **Сценарий 1**: импорт `func1` в `module1.py`:

  ```
  from .module2 import func1
  ```

- Мы поставили одну точку (.) только потому, что `module2.py` находится в той же папке, что и `module1.py`.

- **Сценарий 2**: импорт `module4` в `module1.py`:

  ```
  from ..pkg3 import module4
  ```

- В этом случае мы поставили две точки (..), поскольку `module4.py` находится в родственной с `module1.py` папке.
- **Сценарий 3**: импорт `func2` в `module1.py`:

  ```
  from ..pkg2.sub_pkg_1.module6 import func2
  ```

  Здесь мы поставили две точки (..), поскольку целевой модуль (`module6.py`) находится внутри папки, которая расположена в родственном каталоге для `module1.py`; мы использовали одну точку для доступа к пакету `sub_pkg1`, а другую — для доступа к `module6`.

Одним из преимуществ относительного импорта является значительное сокращение объемов длинных строк кода. Но эти операторы могут быть запутанными и сложными в обслуживании при совместном использовании проектов между командами и организациями. Относительный импорт нелегко читать, и им сложно управлять.

Загрузка и инициализация модуля

Всякий раз при взаимодействии интерпретатора Python с `import` или эквивалентными инструкциями он выполняет три операции, описанные далее.

Загрузка модуля

На этом шаге интерпретатор ищет указанный модуль в `sys.path` (подробнее — в подразделе «*Доступ к пакетам из любого расположения*») и загружает исходный код. Об этом мы уже говорили в подразделе «*Как работает оператор import*».

Установка параметров для специальных переменных

На этом шаге интерпретатор определяет несколько специальных переменных, например, `__name__`, которая определяет пространство имен для модуля Python. Это одна из самых важных переменных.

В примере с модулями `calcmain1.py`, `mycalculator.py` и `myrandom.py` переменная `__name__` будет задана для каждого модуля следующим образом:

Таблица 2.1. Значение атрибута `__name__` для разных модулей

Имя модуля	Значение `__name__`
main.py	__main__
myrandom.py	myrandom
mycalculator.py	mycalculator

Существуют два способа задания переменной `__name__`, которые описаны далее.

Случай А: модуль является главной программой

Если модуль запускается в качестве основной программы, переменной `__name__` будет установлено значение `__main__` независимо от имени файла или модуля Python. Например, при выполнении `calcmain1.py` интерпретатор назначает жестко заданную строку `__main__` переменной `__name__`. Если в качестве основной программы запускаются `myrandom.py` или `mycalculator.py`, переменная `__name__` автоматически получит значение `__main__`.

Поэтому добавим строку `if __name__ == '__main__'` во все главные скрипты для проверки, является ли программа основной для выполнения.

Случай Б: модуль импортируется другим модулем

В этом случае модуль не является основной программой, а импортируется другим модулем. В примере `myrandom` и `mycalculator` импортируются в `calcmain1.py`. Как только интерпретатор Python находит файлы `myrandom.py` и `mycalculator.py`, он назначает имена `myrandom` и `mycalculator` из оператора `import` переменной `__name__` для каждого модуля. Назначение происходит до выполнения кода внутри этих модулей. Это показано в ***таблице 2.1***.

Есть еще несколько специальных переменных, которые заслуживают внимания:

- `__file__`: содержит путь к импортируемому в данный момент модулю;
- `__doc__`: выводит docstring, добавляемую в класс или метод; как уже упоминалось в предыдущей главе, docstring — это строка комментария, добавляемая сразу после определения класса или метода;
- `__package__`: используется для указания, является ли модуль пакетом или нет; в качестве значения могут быть имя пакета, пустая строка или `none`;
- `__dict__`: возвращает все атрибуты экземпляра класса в виде словаря;
- `dir`: возвращает все связанные методы или атрибуты в виде списка;
- `locals` и `globals`: используются как методы для отображения локальных и глобальных переменных в виде записей словаря.

Выполнение кода

После установки специальных переменных интерпретатор Python выполняет код в файле построчно. Важно понимать, что функции (и код в классах) не выполняются, пока они не вызываются другими строками кода. Приведем краткий анализ с точки зрения выполнения при запуске `calcmain1.py`:

- `mycalculator.py`: после установки специальных переменных в этом модуле нет кода, который должен выполняться при инициализации;
- `myrandom.py`: после установки специальных переменных и оператора `import` в этом модуле отсутствует дальнейшее выполнение кода во время инициализации;

- `calcmain1.py`: после установки специальных переменных и выполнения операторов `import` выполняется следующий оператор: `if name == "__main__"`; это выражение вернет значение `true`, поскольку мы запустили файл `calcmain1.py`; внутри `if` будет вызвана функция `my_main()`, которая затем вызовет методы из модулей `myrandom.py` и `mycalculator.py`.

Строку `if __name__ == "__main__"` можно добавить в любой модуль, независимо от того, является ли он основной программой или нет. Преимущество такого подхода заключается в возможности использовать модуль, в том числе и как основную программу. Существует также другое применение этого способа, которое заключается в добавлении модульных тестов в модуль.

Стандартные модули

Python имеет встроенную библиотеку с более 200 стандартных модулей. Точное количество варьируется от дистрибутива к дистрибутиву. Все их можно импортировать в программу. Список модулей очень обширен, но в качестве примера рассмотрим наиболее часто используемые:

- `math`: предоставляет математические функции для арифметических операций;
- `random`: полезен для генерации псевдослучайных чисел с использованием различных типов распределений;
- `statistics`: предлагает статистические функции, например, среднее значение (`mean`), медиана (`median`) и дисперсия (`variance`);
- `base64`: предоставляет функции для кодирования и декодирования данных;
- `calendar`: здесь расположены функции, связанные с календарем, что очень удобно при использовании календарных расчетов;
- `collections`: содержит специализированные типы данных для контейнеров, отличные от встроенных (`dict`, `list`, или `set`); такие типы данных включают в себя `deque`, `Counter` и `ChainMap`;
- `csv`: служит для чтения из CSV-файлов и записи в них;
- `datetime`: предлагает универсальные функции для работы с датой и временем;
- `decimal`: предлагает функции с десятичными арифметическими операциями;
- `logging`: упрощает регистрацию и вход в приложение;
- `os` и `os.path`: дают доступ к системным функциям;
- `socket`: содержит низкоуровневые функции для сетевых коммуникаций через сокеты;
- `sys`: предоставляет доступ к интерпретатору Python для низкоуровневых переменных и функций;
- `time`: предоставляет функции для расчетов со временем, например, для преобразования одних единиц времени в другие.

Написание многоразовых модулей

Модуль можно объявить многоразовым, когда он обладает следующими характеристиками:

- Независимая функциональность.
- Универсальная функциональность.
- Традиционный стиль программирования.
- Четко определенная документация.

Если модуль или пакет не обладает этими характеристиками, использовать его в других программах будет сложно или даже невозможно. Рассмотрим каждую характеристику отдельно.

Независимая функциональность

Функции в модуле должны работать независимо от других модулей, а также локальных или глобальных переменных. Чем более независимы функции, тем больше возможностей повторного использования модулей. Если есть зависимость от других модулей, она должна быть минимальной.

Например, в модуле `mycalculator.py` обе функции независимы друг от друга и могут использоваться в других программах (рис. 2.1):

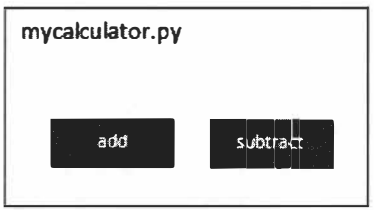


Рис. 2.1. Модуль mycalculator с функциями Add и Subtract

В модуле `myrandom.py` используется системная библиотека `random` для генерации случайных чисел. Этот модуль можно легко использовать снова, поскольку библиотека является встроенной в Python (рис. 2.2):

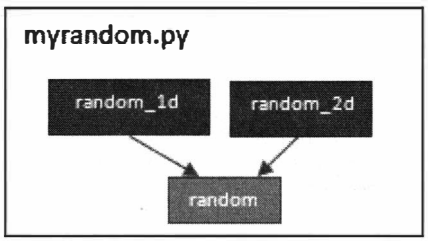


Рис. 2.2. Модуль myrandom с зависимостью от библиотеки random

Модуль со сторонними библиотеками будет работать в нашем окружении, но при работе с другими модулями могут возникнуть проблемы, если в их окружении не подключены те же библиотеки, что и в нашем.

Убедимся в этом на примере. Создадим новый модуль `mypandas.py`, который будет использовать базовый функционал библиотеки `pandas`. Для простоты добавим в него только одну функцию, которая выводит **DataFrame** в соответствии со словарем, предоставленным в качестве входной переменной для этой функции.

Фрагмент кода `mypandas.py`:

```
#mypandas.py
import pandas

def print_dataframe(dict):
    """ Эта функция выводит словарь в виде фрейма данных """
    brics = pandas.DataFrame(dict)
    print(brics)
```

Модуль `mypandas.py` будет использовать библиотеку `pandas` для создания объекта `dataframe` из словаря. Эта зависимость также показана на следующей схеме (рис. 2.3):

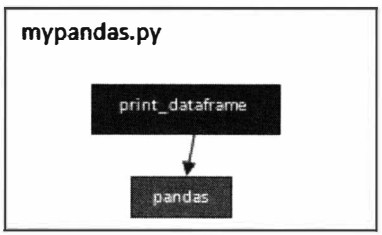


Рис. 2.3. Модуль mypandas с зависимостью от сторонней библиотеки pandas

Обратите внимание, библиотека `pandas` не является встроенной или системной. Если попробовать поделиться нашим модулем с другими без определения явной зависимости от сторонней библиотеки (в нашем случае `pandas`), программа при попытке использовать этот модуль выдаст следующее сообщение об ошибке:

```
ImportError: No module named pandas'
```

Вот почему важно сделать модуль максимально независимым. Если приходится использовать сторонние библиотеки, необходимо определить четкие зависимости и использовать соответствующий подход к упаковке их в пакет. Об этом мы поговорим в подразделе *«Общий доступ к пакету»*.

Генерализация функционала

В идеале многоразовый модуль должен фокусироваться на решении общей проблемы, а не какой-то конкретной. Допустим, мы хотим переводить дюймы в санти-

метры. Для этого можно написать функцию на основе формулы преобразования. А как насчет функции, которая переводит любые значения из имперской системы в метрическую? У нас может быть одна функция для всех преобразований (дюймы в сантиметры, футы в метры, мили в километры) или разные функции для каждого типа преобразования. А что насчет обратных преобразований (сантиметры в футы)? Возможно, сейчас в этом нет необходимости, но это может понадобиться позже нам или кому-то, кто также использует этот модуль. Такая *генерализация* сделает функционал модуля не только исчерпывающим, но и подходящим для повторного использования без расширения его функционала.

Для демонстрации принципам изменим модуль `myrandom` и сделаем его более общим и, следовательно, более пригодным для повторного использования. Сейчас у нас определены разные функции для однозначных и двузначных чисел. Что если мы захотим сгенерировать трехзначное случайное число или число от 20 до 30? Для генерализации этого требования введем в модуль новую функцию `get_random`, которая принимает пользовательский ввод для верхнего и нижнего пределов для случайных чисел. Эта функция будет обобщением для двух других, которые мы определили ранее. Теперь их можно удалить или оставить в модуле для удобства использования. Мы создали новую функцию только для сравнения обобщенной функции (`get_random`) с отдельными функциями (`random_1d` и `random_2d`).

Обновленная версия модуля `myrandom.py` (`myrandomv2.py`) выглядит так:

```
# myrandomv2.py со стандартной и пользовательской функциями random
import random

def random_1d():
   """Эта функция генерирует случайное число между 0 and 9"""
   return random.randint(0,9)

def random_2d():
   """Эта функция генерирует случайное число между 10 and 99"""
   return random.randint(10,99)

def get_random(lower, upper):
   """Эта функция генерирует случайное число между нижним\
    и верхним пределами
    return random.randint(lower,upper)
```

Традиционный стиль программирования

В первую очередь это касается того, как мы пишем имена функций, переменных и модулей. В Python существует система написания кода и соглашения об именовании, которые обсуждались в предыдущей главе. Важно следовать этим соглашениям, особенно при создании многоразовых модулей и пакетов. В противном случае, их будет очень неудобно использовать снова.

Для наглядности рассмотрим следующий фрагмент кода, где для имен функций и параметров используется верблюжий регистр:

```
def addNumbers(numParam1, numParam2)
    #код функции пропущен
def featureCount(moduleName)
    # код функции пропущен
```

Если вы привыкли работать с **Java**, этот стиль написания вам подойдет. Но в Python это считается плохой практикой. Использование стиля, не принятого в Python, затрудняет повторное использование таких модулей.

Ниже приведен еще один фрагмент с уже верным стилем написания имен функций:

```
def add_numbers(num_param1, num_param2)
    # код функции пропущен
Def feature_count(module_name)
    # код функции пропущен
```

Другой пример хорошего многоразового стиля программирования приводится на скриншоте из PyCharm IDE для библиотеки `pandas` (рис. 2.4):

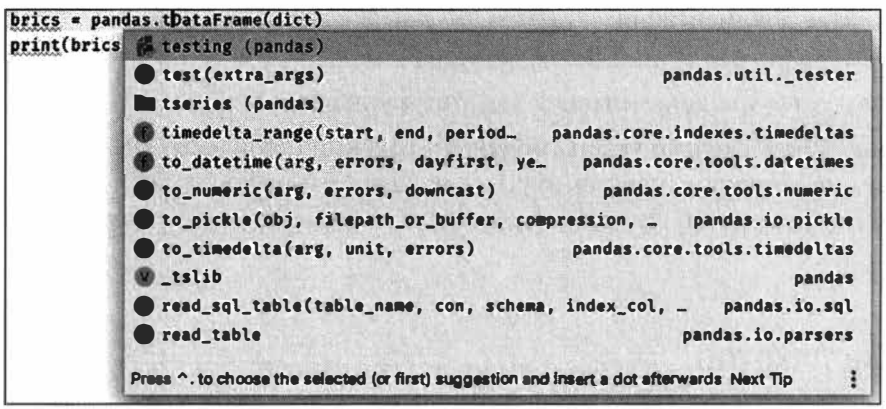


Рис. 2.4. Библиотека pandas в PyCharm IDE

Имена функций и переменных легко понять даже без чтения какой-либо документации. Соблюдение стандартного стиля программирования делает код более удобным для многократного использования.

Четко определенная документация

Четко определенная и понятная документация так же важна, как и написание универсального и независимого модуля с соблюдением всех рекомендаций. Без прозрачной документации модуль не вызовет интерес разработчиков к его повторному использованию. Как программисты, мы уделяем больше внимания написанию кода.

Но написание всего нескольких строк документации может сделать 100 строк нашего кода более удобными в использовании и сопровождении.

Рассмотрим несколько хороших примеров документирования модуля на примере mycalculator.py:

```
""" mycalculator.py
    Этот модуль предоставляет функции для сложения и вычитания 2 чисел"""
def add(x,  y):
    """ Эта функция складывает 2 числа.
    пример: add (3, 4) """
    return x + y
def subtract(x, y):
    """ Эта функция находит разность 2 чисел
    пример: subtract (17, 8) """
    return x - y
```

В Python важно помнить следующее:

- Можно использовать тройные кавычки для многострочных комментариев.
- Строки в тройных кавычках указываются в начале модуля, а затем используются в качестве документации.
- Если функция начинается с комментария в тройных кавычках, этот комментарий используется как документация для этой функции.

Можно написать сколько угодно модулей с сотнями строк кода. Но требуется нечто большее, чем простое программирование, для возможности многократно их использовать. Например, генерализация, стиль написания кода и, самое главное, документация.

Сборка пакетов

Существует ряд методов и инструментов для создания и распространения пакетов. Python не может похвастаться богатой историей стандартизации по их созданию. В первое десятилетие XXI века было много проектов по оптимизации этого процесса, но большого успеха они не принесли. За последние годы удалось добиться определенного успеха благодаря усилиям рабочей группы **Python Packaging Authority (PyPA)**.

В этом подразделе мы рассмотрим методы создания пакетов, доступ к ним в программе, а также публикацию и совместное использование в соответствии с рекомендациями PyPA.

Начнем с именования пакетов, рассмотрим файл инициализации, а затем попробуем собрать пакет.

Именование

Имена пакетов должны следовать тому же правилу, что и модули, — нижний регистр и без подчеркиваний. Пакеты действуют как структурированные модули.

Файл инициализации пакета

Пакет может иметь опциональный исходный файл с именем `__init__.py` (или просто файл `init`). Его наличие (даже пустого) рекомендуется для пометки папок как пакетов. Начиная с версии Python 3.3 использовать файл необязательно (*PEP 420: неявные пакеты пространства имен*). Файл инициализации имеет многоцелевое назначение. До сих пор нет единого мнения, что можно указывать в нем, а что — нет. Вот несколько примеров использования:

- **Пустой `__init__.py`**: разработчикам придется использовать явный импорт и управлять пространствами имен по своему усмотрению; они будут вынуждены импортировать отдельные модули, что может быть неудобно для большого пакета.
- **Полный импорт в `__init__.py`**: в этом случае можно импортировать весь пакет, а потом ссылаться на модули напрямую в коде по имени или сокращенному обозначению пакета; это удобнее, но лишь за счет поддержки всех импортов в файле `__init__`.
- **Ограниченный импорт**: при таком подходе разработчики могут импортировать в `init` только ключевые функции из разных модулей и управлять ими в пространстве имен пакета; это дает дополнительное преимущество, заключающееся в предоставлении оболочки вокруг функциональных возможностей базового модуля; если по какой-то причине придется проводить рефакторинг базовых модулей, есть возможность сохранить пространство имен прежним, особенно для потребителей API; единственным недостатком такого подхода являются дополнительные усилия по управлению и поддержке такими файлами.

Иногда разработчики добавляют в файл инициализации код, который выполняется при импорте модуля из пакета. Примером этого служит создание *сеансов* для удаленных систем, таких, как *базы данных* или удаленный *SSH-сервер*.

Сборка пакета

Рассмотрим, как собрать пакет, на простом примере. Создадим пакет `masifutil`, используя следующие модули и подпакет:

- Модуль `mycalculator.py`: его мы уже создали в подразделе «*Импорт модулей*».
- Модуль `myrandom.py`: его мы также уже создали в подразделе «*Импорт модулей*».
- Подпакет `advcalc`: будет содержать один модуль `advcalculator.py`; для этого подпакета определим файл `init`, но оставим его пустым.

Модуль `advcalculator.py` предоставляет дополнительные функции для вычисления квадратного корня и логарифма по основаниям 10 и 2. Исходный код модуля следующий:

```
# advcalculator.py с функциями sqrt, log и ln
import math
def sqrt(x):
    """Эта функция вычисляет квадратный корень числа"""
    return math.sqrt(x)
def log(x):
    """Эта функция возвращает логарифм по основанию 10"""
    return math.log(x,10)
def ln(x):
    """ Эта функция возвращает логарифм по основанию 2"""
    return math.log(x,2)
```

Файловая структура пакета `masifutil` с файлами инициализации выглядит следующим образом (рис. 2.5):



Рис. 2.5. Структура папок в пакете masifutil с модулями и подпакетами

Затем создадим новый скрипт (`pkgmain1.py`), который будет использовать модули из пакета или из подпапки `masifutil`. В этом скрипте мы импортируем модули из основного пакета и подпакета, используя структуру папок. А затем с помощью функций модуля возьмем два случайных числа, вычислим их сумму и разность, а также квадратный корень и логарифмы первого случайного числа. Исходный код `pkgmain1.py` выглядит так:

```
# pkgmain0.py с прямым импортом
import masifutil.mycalculator as calc
import masifutil.myrandom as rand
import masifutil.advcalc.advcalculator as acalc
def my_main():
    """ Это основная функция, которая создает два случайных\
     числа и затем применяет к ним функции калькулятора """
    x = rand.random_2d()
    y = rand.random_1d()
```

```
    sum = calc.add(x,y)
    diff = calc.subtract(x,y)

    sroot = acalc.sqrt(x)
    log10x = acalc.log(x)
    log2x = acalc.ln(x)

    print("x = {}, y = {}".format(x, y))
    print("sum is {}".format(sum))
    print("diff is {}".format(diff))
    print("square root is {}".format(sroot))
    print("log base of 10 is {}".format(log10x))
    print("log base of 2 is {}".format(log2x))
""" Выполняется только, если переменная '__name__' установлена
как основная """
if __name__ == "__main__":
    my_main()
```

Здесь мы используем имена пакета и модуля для импорта. Это неудобно, особенно если мы импортируем также подпакеты. Можно получить те же результаты, используя следующие инструкции:

```
# mypkgmain1.py с операторами from
from masifutil import mycalculator as calc
from masifutil import myrandom as rand
from masifutil.advcalc import advcalculator as acalc
#остальной код такой же, как в mypkgmain1.py
```

Как уже говорилось, создавать пустой файл `__init__.py` необязательно. Но мы добавили его для наглядности.

Далее рассмотрим, как добавить несколько операторов `import` в файл инициализации. Начнем с импорта модулей. В файле `init` верхнего ранга импортируем все функции, как показано ниже:

```
#файл __init__ для пакета 'masifutil'
from .mycalculator import add, subtract
from .myrandom import random_1d, random_2d
from .advcalc.advcalculator import sqrt, log, ln
```

Обратите внимание на использование точки (.) перед именем модуля. В Python это обязательно для относительного импорта.

Благодаря этим трем строкам в файле `init`, основной скрипт будет выглядеть проще:

```
# pkgmain2.py с основной функцией
import masifutil
```

```
def my_main():
    """ Это основная функция, которая создает два случайных\
     числа и затем применяет к ним функции калькулятора """
    x = masifutil.random_2d()
    y = masifutil.random_1d()

    sum = masifutil.add(x,y)
    diff = masifutil.subtract(x,y)

    sroot = masifutil.sqrt(x)
    log10x = masifutil.log(x)
    log2x = masifutil.ln(x)

    print("x = {}, y = {}".format(x, y))
    print("sum is {}".format(sum))
    print("diff is {}".format(diff))
    print("square root is {}".format(sroot))
    print("log base of 10 is {}".format(log10x))
    print("log base of 2 is {}".format(log2x))
""" Выполняется только, если переменная '__name__' установлена
как основная """
if __name__ == "__main__":
    my_main()
```

Функции из двух главных модулей и модуля в подпакете доступны на уровне главного пакета. Разработчикам не придется разбираться в иерархии и структуре модулей. Все это благодаря инструкциям `import` в файле `init`.

Пакет собирается таким образом, что исходный код пакета находится в той же папке, что и главная программа или скрипт. Это работает только для совместного использования модулей в рамках одного проекта. Но что если необходимо использовать пакет в других проектах или программах?

Доступ к пакетам из любого расположения

Пакет, созданный ранее, доступен только в случае, если программа, вызывающая модули, находится на том же уровне, что и расположение пакета. Такое условие нецелесообразно для повторного использования кода.

В этом подразделе мы обсудим, как сделать пакеты доступными и пригодными для использования в любом месте системы.

Добавление в sys.path

Это полезная опция для динамической настройки `sys.path`. Обратите внимание, `sys.path` — это список каталогов, в которых интерпретатор Python делает поиск ка-

ждый раз, когда выполняется оператор `import` в исходной программе. С помощью этого подхода мы присоединяем (добавляем) пути каталогов или папок с нашими пакетами к `sys.path`.

Для пакета `masifutil` мы создадим новую программу `pkgmain3.py`, которая является копией `pkgmain2.py` (будет обновлена позже), но хранится вне папки с пакетом `masifutil`. Программа `pkgmain3.py` разместится в любой папке, кроме `mypackages`. Структура папок с новым основным скриптом (`pkgmain3.py`) и пакетом `masifutil` (рис. 2.6):



Рис. 2.6. Структура папок пакета masifutil и новый основной скрипт pkgmain3.py

При выполнении `pkgmain3.py` программа выдаст ошибку: `ModuleNotFoundError: No module named 'masifutil'`. Это ожидаемо, поскольку путь к пакету `masifutil` не добавлен в `sys.path`. Для добавления папки пакета в `sys.path` отредактируем основную программу. Назовем ее `pkgmain4.py` и дополним инструкциями по добавлению в `sys.path`:

```
# pkgmain4.py с кодом добавления в sys.path

import sys
sys.path.append('/Users/muasif/Google Drive/PythonForGeeks/
source_code/chapter2/mypackages')

import masifutil

def my_main():
    """ Это основная функция, которая создает два случайных\
     числа и затем применяет к ним функции калькулятора """
    x = masifutil.random_2d()
    y = masifutil.random_1d()

    sum = masifutil.add(x,y)
    diff = masifutil.subtract(x,y)

    sroot = masifutil.sqrt(x)
    log10x = masifutil.log(x)
    log2x = masifutil.ln(x)
```

```
    print("x = {}, y = {}".format(x, y))
    print("sum is {}".format(sum))
    print("diff is {}".format(diff))
    print("square root is {}".format(sroot))
    print("log base of 10 is {}".format(log10x))
    print("log base of 2 is {}".format(log2x))
""" Выполняется только, если переменная '__name__' установлена
как основная """
if __name__ == "__main__":
    my_main()
```

После добавления строк основной скрипт выполнился без ошибок и с ожидаемым консольным выводом. Это связано с тем, что пакет `masifutil` теперь доступен по пути, по которому интерпретатор Python может загрузить его при импорте в основной скрипт.

Альтернативой `sys.path` может служить функция `site.addsitedir` из модуля `site`. Единственное преимущество этого подхода в том, что функция также ищет файлы `.pth` в указанных папках. Это полезно для добавлении дополнительных директорий (например, подпакетов). Фрагмент основного скрипта (`pktpamin5.py`) с функцией `addsitedir`:

```
# pkgmain5.py

import site
site.addsitedir('/Users/muasif/Google Drive/PythonForGeeks/
source_code/chapter2/mypackages')
import masifutil
#остальной код без изменений как в pkymain4.py
```

Обратите внимание, при таком подходе каталоги доступны только при выполнении программы. Для использования `sys.path` на постоянной основе (на уровне сеанса или системы) есть другие более полезные способы.

Переменная среды PYTHONPATH

Это удобный способ добавить папку с пакетом в `sys.path`, который интерпретатор Python будет использовать для поиска пакетов и модулей, если они отсутствуют во встроенной библиотеке. Посмотрим, как можно определить эти переменные в зависимости от операционной системы.

В Windows эту переменную среды можно определить следующими способами:

- **Командная строка**: `PYTHONPATH = "C:\pythonpath1;C:\pythonpath2";` это хороший способ для активного сеанса.
- **Графический пользовательский интерфейс**: выполняем переход по меню **Мой компьютер > Свойства системы > Дополнительные параметры системы > Переменные среды**; это постоянная настройка.

В Linux и macOS это можно сделать с помощью команды `export PYTHONPATH=`/some/path/``. Если используется **Bash** или аналогичный *терминал*, переменная среды будет эффективна только для терминального сеанса. Для перманентной установки переменной среды рекомендуется добавить ее в конец файла профиля: `~/bash_profile`.

Если попробовать выполнить `pkgmain3.py` без установки `PYTHONPATH`, интерпретатор вернет ошибку `ModuleNotFoundError: No module named 'masifutil'`. Это также ожидаемо, поскольку путь к пакету `masifutil` не добавлен в `PYTHONPATH`.

На следующем шаге добавим путь к папке с пакетом `masifutil` в переменную `PYTHONPATH` и перезапустим `pkgmain3.py`. Теперь программа работает без ошибок и с ожидаемым консольным выводом.

Использование .pth-файла в пакете site

Это еще один удобный способ добавления пакетов в `sys.path`. Достигается определением файла `.pth` в пакетах `site`. Файл может содержать все папки, которые надо добавить в `sys.path`.

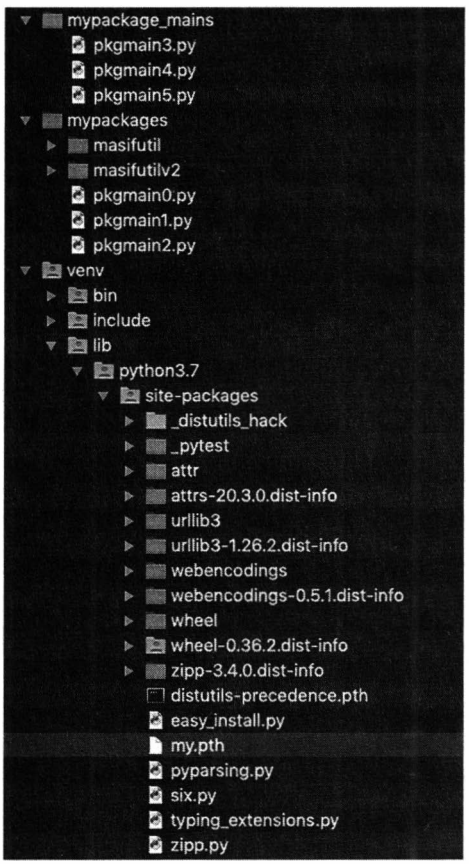


Рис. 2.7. Файловая структура с файлом my.pth

Для демонстрации мы создали файл `my.pth` в каталоге `venv/lib/Python3.7/site-packages`. Как видно на рис. 2.7, мы добавили папку, которая содержит пакет `masifutil`. С этим простым файлом `.pth` основной скрипт `pkymain3.py` отлично работает без ошибок и с ожидаемым консольным выводом.

Рассмотренные методы доступа к пользовательским пакетам эффективны для повторного использования в одной системе с любой программой. Далее мы рассмотрим, как делиться пакетами с другими разработчиками и сообществами.

Общий доступ к пакету

Существует много инструментов для общего доступа к пакетам и проектам Python. Сосредоточимся на тех, которые рекомендованы группой PyPA.

Инструменты, о которых следует упомянуть:

- **distutils**: поставляется с базовой версией Python, но плохо масштабируется для сложных и нестандартных пакетов;
- **setuputils**: сторонний инструмент и расширение для **distutils**; рекомендуется для сборки пакетов;
- **wheel**: формат распространения пакетов Python, который делает установку быстрее и проще по сравнению с двумя предыдущими;
- **pip**: менеджер пакетов и модулей Python, который поставляется, начиная с версии 3.4; легко позволяет устанавливать модули командой:

  ```
  pip install <имя_модуля>;
  ```

- **Python Package Index (PyPI)**: репозиторий программного обеспечения (ПО) для языка Python; используется для поиска и установки ПО, которое разрабатывается и распространяется сообществом Python;
- **Twine**: утилита для публикации пакетов Python в PyPI.

Далее добавим в пакет `masifutil` дополнительные компоненты в соответствии с рекомендациями PyPA. Затем установим обновленный пакет `masifutil` для всей системы с помощью `pip`. В конце опубликуем его в **Test PyPI**, а затем установим его из **Test PyPI**.

Создание пакета в соответствии с рекомендациями PyPA

PyPA рекомендует использовать пример проекта для сборки многоразовых пакетов, который доступен по ссылке *https://github.com/pypa/sampleproject*.

Фрагмент кода из примера проекта на GitHub (рис. 2.8):

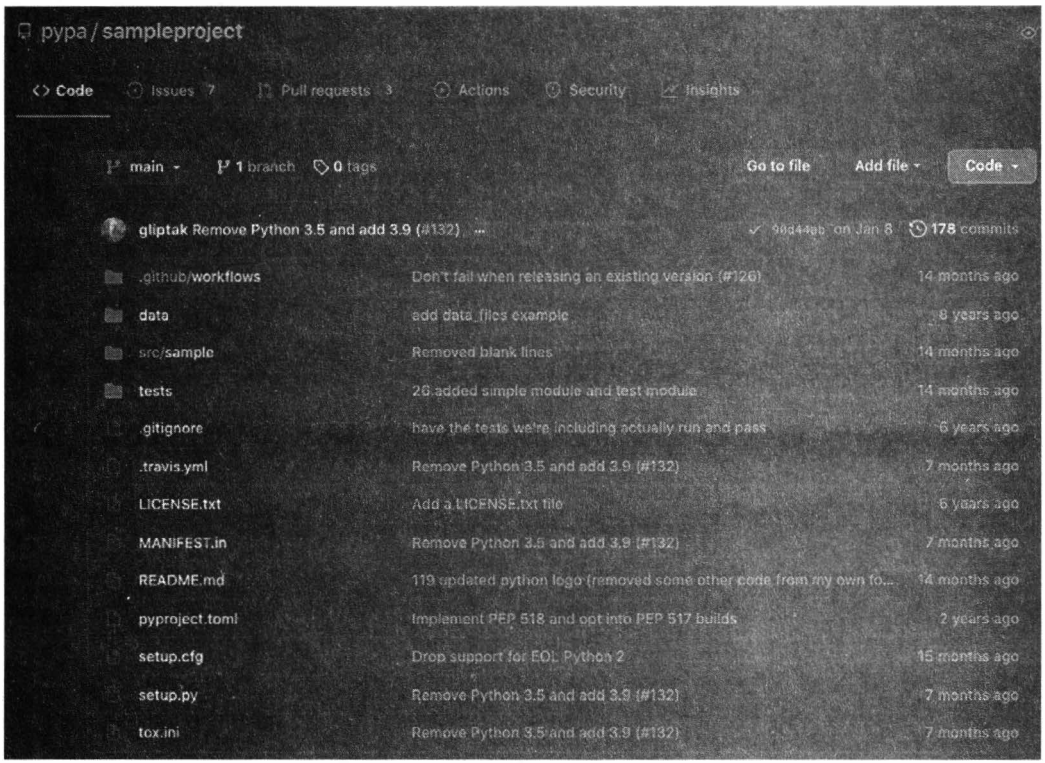


Рис. 2.8. Пример проекта от PyPA на GitHub

Для начала рассмотрим ключевые файлы и папки, прежде чем использовать их для обновления пакета `masifutil`:

- `setup.py`: самый важный файл, который должен быть в корне проекта или пакета; это скрипт для сборки и установки пакета, который содержит глобальную функцию `setup()`;файл также предоставляет *интерфейс командной строки* (**Command Line Interface**, **CLI**) для выполнения различных команд;
- `setup.cfg`: это `ini`-файл, который может использоваться файлом `setup.py` для определения значений по умолчанию;
- ключевые *аргументы*, которые можно передать в функцию `setup()`:
  - `Name` (имя);
  - `Version` (версия);
  - `Description` (описание);
  - `URL` (URL-адрес);
  - `Author` (автор);
  - `License` (лицензия).

- `README.rst/README.md`: этот файл (в формате **reStructured** или **Markdown**) может содержать информацию о пакете или проекте;
- `license.txt`: файл содержит подробные условия распространения и должен быть включен в каждый пакет; файл лицензии особенно важен в странах, где распространять пакеты без соответствующей лицензии незаконно;
- `MANIFEST.in`: с помощью него можно указать список дополнительных файлов, которые должны быть включены в пакет; этот список не включает файлы с исходным кодом (они добавляются автоматически);
- `<имя_пакета>`: это пакет верхнего уровня, который включает в себя все модули и подпакеты; его использование не обязательно, но рекомендовано;
- `data`: здесь можно добавить файлы с данными, если необходимо;
- `tests`: заглушки для будущих модульных тестов.

На этом шаге обновим предыдущий пакет `masifutil` в соответствии с рекомендациями PyPA. Новая структура папок и файлов обновленного пакета `masifutilv2` выглядит так (рис. 2.9):



Рис. 2.9. Обновленная файловая структура masifutilv2

Мы добавили каталоги `data` и `tests`, но они пока пустые. О модульных тестах мы поговорим в одной из следующих глав.

Содержимое большинства дополнительных файлов описывается в примере проекта, поэтому здесь их обсуждение мы пропустим, за исключением файла `setup.py`.

В `setup.py` указаны основные аргументы для проекта. Остальные аргументы доступны в примере файла `setup.py`, который предоставлен группой PyPA вместе с примером проекта. Фрагмент кода из `setup.py` представлен ниже:

```
from setuptools import setup

setup(
    name='masifutilv2',
```

```
    version='0.1.0',
    author='Muhammad Asif',
    author_email='ma@example.com',
    packages=['masifutil', 'masifutil/advcalc'],
    python_requires='>=3.5, <4',
    url='http://pypi.python.org/pypi/PackageName/',
    license='LICENSE.txt',
    description='A sample package for illustration purposes',
    long_description=open('README.md').read(),
    install_requires=[
    ],
)
```

С помощью `setup.py` теперь можно делиться пакетом `masifutilv2` как локально, так и удаленно. Об этом мы поговорим в следующих подразделах.

Установка из локального исходного кода с помощью pip

Мы добавили в пакет новые файлы и теперь можем установить его с помощью `pip`. Самый простой способ установки — выполнить следующую команду с указанием пути к папке `masifutilv2`:

```
> pip install <путь к masifutilv2>
```

Ниже приведен консольный вывод команды при запуске без установки пакета `wheel`:

```
Processing ./masifutilv2
Using legacy 'setup.py install' for masifutilv2, since package
'wheel' is not installed.
Installing collected packages: masifutilv2
    Running setup.py install for masifutilv2 ... done
Successfully installed masifutilv2-0.1.0
```

Утилита установила пакет, но с использованием **EGG-формата**, поскольку пакет `wheel` не был установлен. Как выглядит наше виртуальное окружение после установки показано на рис. 2.10.

После установки пакета в виртуальное окружение мы протестировали его с помощью программы `pkgmain3.py`, которая отработала, как и ожидалось.

> ***Совет***
>
> Удалить пакет можно командой `pip uninstall masifutilv2`.

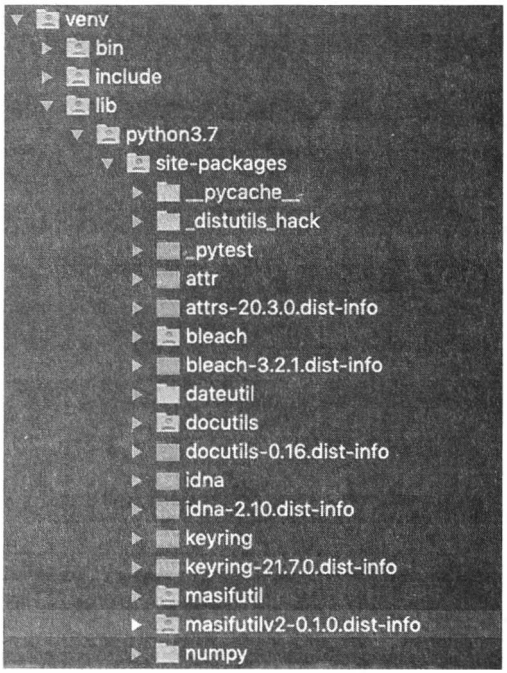


Рис. 2.10. Виртуальное окружение после установки masifutilv2 с помощью pip

Далее установим пакет `wheel`, а затем снова переустановим его:

```
> pip install <путь к masifutilv2>
```

Консольный вывод будет следующим:

```
Processing ./masifutilv2
Building wheels for collected packages: masifutilv2
    Building wheel for masifutilv2 (setup.py) ... done
    Created wheel for masifutilv2: filename=masi
futilv2-0.1.0-py3-none-any.whl size=3497
sha256=038712975b7d7eb1f3fefa799da9e294b34
e79caea24abb444dd81f4cc44b36e
    Stored in folder: /private/var/folders/xp/g88fvmgs0k90w0rc_
qq4xkzxpsx11v/T/pip-ephem-wheel-cache-l2eyp_wq/wheels/
de/14/12/71b4d696301fd1052adf287191fdd054cc17ef6c9b59066277
Successfully built masifutilv2
Installing collected packages: masifutilv2
Successfully installed masifutilv2-0.1.0
```

На этот раз пакет успешно установлен с помощью `wheel`. Также видно, что он отображается в нашем виртуальном окружении, как показано на рис. 2.11.

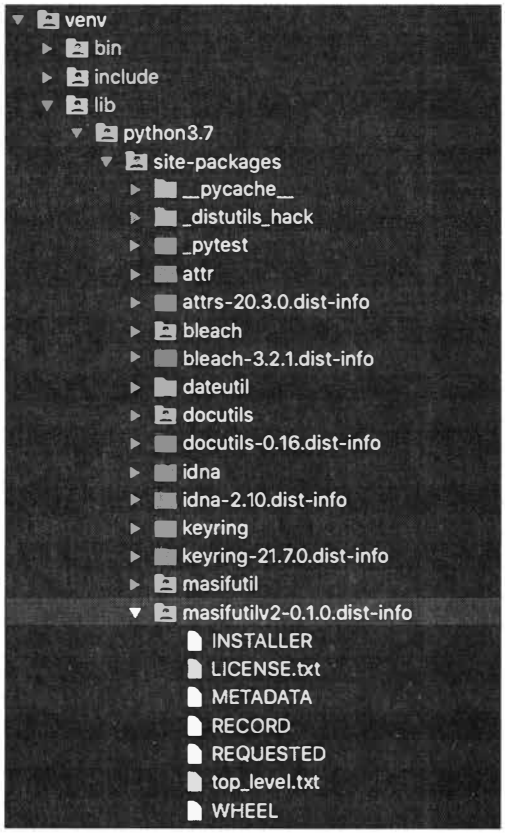


Рис. 2.11. Виртуальное окружение после установки masifutilv2 с пакетом wheel, используя pip

Мы узнали, как установить пакет с помощью утилиты `pip` из локального исходного кода. Далее опубликуем пакет в централизованном репозитории (Test PyPI).

Публикация пакета в Test PyPI

В качестве следующего шага добавим пример пакета в репозиторий PyPI. Перед этим нужно создать аккаунт Test PyPI. Обратите внимание, Test PyPI — это отдельный экземпляр индекса пакета, предназначенный специально для тестирования. Помимо создания аккаунта также необходимо добавить *API-токен*. Детали создания аккаунта и добавления токена мы оставим на веб-сайте Test PyPI (*https://test.pypi.org/*).

Для отправки пакета понадобится утилита **Twine**. Предположим, что она уже установлена с помощью `pip`. Для отправки `masifutilv2` выполним следующие действия:

1. Создадим дистрибутив следующей командой. Утилита `sdist` создаст с помощью архиватора **TAR** ZIP-файл ниже папки `dist`:

```
> python setup.py sdist
```

2. Отправим файл дистрибутива в Test PyPI. При запросе имени пользователя и пароля укажем `__token__` и API-токен соответственно.

```
> twine upload -repository testpypi dist/masifutilv2-0.1.0.tar.gz
```

3. Эта команда отправит ZIP-файл пакета в репозиторий Test PyPI. Вывод консоли будет следующим:

```
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: __token__
Enter your password:
Uploading masifutilv2-0.1.0.tar.gz
100%|████████████████████|
5.15k/5.15k [00:02<00:00, 2.21kB/s]
```

После успешной загрузки можно просмотреть файл по адресу: *https://test.pypi.org/project/ masifutilv2/0.1.0/*.

Установка пакета из PyPI

Установка пакета из Test PyPI аналогична установке из обычного репозитория, за исключением, что нужно указать URL-адрес репозитория с помощью аргументов `index-url`. Команда выглядит следующим образом:

```
> pip install --index-url https://test.pypi.org/simple/ --nodeps masifutilv2
```

Консольный вывод будет следующим:

```
Looking in indexes: https://test.pypi.org/simple/
Collecting masifutilv2
    Downloading https://test-files.pythonhosted.org/
    packages/b7/e9/7afe390b4ec1e5842e8e62a6084505cbc6b9
    f6adf0e37ac695cd23156844/masifutilv2-0.1.0.tar.gz (2.3 kB)
Building wheels for collected packages: masifutilv2
    Building wheel for masifutilv2 (setup.py) ... done
    Created wheel for masifutilv2: filename=masifutilv2-
    0.1.0-py3-none-any.whl size=3497
    sha256=a3db8f04b118e16ae291bad9642483874
    f5c9f447dbee57c0961b5f8fbf99501
    Stored in folder: /Users/muasif/Library/Caches/pip/
    wheels/1c/47/29/95b9edfe28f02a605757c1
    f1735660a6f79807ece430f5b836
Successfully built masifutilv2
Installing collected packages: masifutilv2
Successfully installed masifutilv2-0.1.0
```

Из вывода видно, что `pip` ищет модуль в Test PyPI. Как только он находит пакет с именем `masifutilv2`, загружает его, а затем устанавливает в виртуальном окружении.

Проще говоря, после создания пакета с использованием рекомендуемого формата и стиля публикация и доступ к пакету — лишь вопрос выбора утилит Python и следования стандартным действиям.

Заключение

В этой главе мы рассмотрели концепцию модулей и пакетов в Python. Узнали, как создавать многоразовые модули и как импортировать их другими модулями и программами. Мы также обсудили загрузку и инициализацию модулей, когда они включаются другими программами в процессе импорта. В последней части главы мы поговорили о создании простых и сложных пакетов, предоставили множество примеров кода для доступа к пакетам, а также установили и опубликовали пакет для повторного использования.

Это важные навыки, если вы работаете над проектом в команде организации или создаете библиотеки Python для более крупного сообщества.

В следующей главе мы поговорим о новом уровне модульности с использованием объектно-ориентированного программирования в Python. Охватим такие понятия, как *инкапсуляция*, *полиморфизм* и *абстракция*, которые являются ключевыми инструментами при создании и управлении проектами в реальном мире.

Вопросы

1. В чем разница между модулем и пакетом?
2. Что такое абсолютный и относительный импорт в Python?
3. Что такое PyPA?
4. Что такое Test PyPI и зачем он нужен?
5. Является ли `init`-файл обязательным условием сборки пакета?

Дополнительные ресурсы

- *Modular Programming with Python,* автор: Эрик Вестра (Erik Westra).
- *Expert Python Programming,* авторы: Михал Яворски (Michał Jaworski) и Тарек Зиаде (Tarek Ziadé).
- *Руководство пользователя по пакетам Python* (*https://packaging.python.org/*).
- *PEP 420: неявные пакеты пространства имен* (*https://www.python.org/dev/peps/pep-0420/*).

Ответы

1. Модуль предназначен для организации функций, переменных и классов в отдельные файлы кода Python. Пакет Python похож на папку для организации нескольких модулей или подпакетов.
2. Абсолютный импорт требует указания абсолютного пути к пакету, начиная с верхнего уровня, тогда как относительный импорт основан на относительном пути пакета в соответствии с текущим расположением программы, где используется оператор `import`.
3. Python Packaging Authority (PyPA) — это рабочая группа, которая поддерживает основной набор программных проектов, используемых в создании пакетов в Python.
4. Test PyPI — это репозиторий программного обеспечения в Python. Он предназначен для тестирования.
5. Начиная с Python 3.3 `init`-файл является опциональным.

3

Расширенное объектно-ориентированное программирование на Python

Python можно использовать в качестве *декларативного* модульного языка программирования, такого, как C, а также для *императивного* или полноценного *объектно-ориентированного программирования* (ООП) вместе с другими языками, такими, как Java. Декларативное программирование — это парадигма, которая сосредоточена на том, *что* мы хотим реализовать, императивное же фиксируется на конкретных *шагах* на пути к этой реализации. Python хорошо вписывается в обе парадигмы. Объектно-ориентированное программирование — это форма императивного программирования, в которой свойства и поведение реальных объектов включены в программы. ООП также рассматривает отношения между разными типами объектов из реального мира.

В этой главе мы рассмотрим, как принципы ООП могут быть реализованы с помощью Python. Предположим, что вы уже знакомы с такими основными понятиями, как классы, объекты и экземпляры, а также имеете представление о наследовании между объектами.

Темы этой главы:

- Знакомство с классами и объектами.
- Принципы ООП.
- Полиморфизм.
- Композиция как альтернативный подход к проектированию.
- Утиная типизация в Python.
- Когда не стоит использовать ООП в Python.

Технические требования

В этой главе вам понадобится:

- Python 3.7 или более поздней версии.

Пример кода для этой главы:
https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter03.

Знакомство с классами и объектами

Класс — это схема того, как объект должен быть определен. Сама по себе она не содержит никаких данных, это просто шаблон.

Объект класса — это экземпляр, созданный из класса, поэтому его так же называют *экземпляром класса*. В этой книге мы будем использовать термины *объект* и *экземпляр* как синонимы. Объекты в ООП могут быть представлены как физические предметы — столы, стулья или книги, но в большинстве случаев они представляют собой абстрактные (не физические) сущности, например, счета, имена, адреса и платежи.

Рассмотрим классы и объекты на примерах.

Различия между атрибутами класса и атрибутами экземпляра

Атрибуты определены как часть описания класса, их значения не меняются во всех экземплярах, созданных из этого класса. Обратиться к атрибутам можно по имени класса или имени экземпляра, но первый вариант предпочтительнее (для чтения или обновления). Состояние или данные объекта представлены в *атрибутах экземпляра*.

В Python класс определяется ключевым словом `class`. Как мы уже говорили в первой главе, имя класса указывается в `ВерблюжьемРегистре`. Следующий фрагмент кода создает класс `Car` (автомобиль):

```
#carexample1.py
class Car:
    pass
```

Этот класс не имеет атрибутов и методов. Он пустой, и можно подумать, что без дополнительных компонентов он совершенно бесполезен, но это не так. В Python можно добавлять атрибуты *на лету* без определения в классе. В следующем фрагменте мы добавим атрибуты в экземпляр во время выполнения:

```
#carexample1.py
class Car:
    pass
```

```
if __name__ == "__main__":
    car = Car ()
    car.color = "blue"
    car.miles = 1000
    print (car.color)
    print (car.miles)
```

Здесь мы создали экземпляр (`car`) нашего класса `Car`, а затем добавили два атрибута этого экземпляра: `color` (цвет) и `miles` (мили, пробег). Обратите внимания, в примере мы добавляем атрибуты именно экземпляра.

Затем добавим атрибуты класса и атрибуты экземпляра с помощью *метода конструктора* (`__init__`), который загружается при создании объекта. Фрагмент кода с двумя атрибутами экземпляра (`color` и `miles`) и методом `init` выглядит следующим образом:

#carexample2.py

```
class Car:
    c_mileage_units = "Mi"
    def __init__(self, color, miles):
        self.i_color = color
        self.i_mileage = miles
if __name__ == "__main__":
    car1 = Car ("blue", 1000)
    print (car.i_color)
    print (car.i_mileage)
    print (car.c_mileage_units)
    print (Car.c_mileage_units)
```

В этой программе мы сделали следующее:

1. Создали класс `Car` с атрибутом `c_mileage_units` (единицы пробега) и двумя переменными экземпляра: `i_color` и `i_mileage`.
2. Создали экземпляр `car` класса `Car`.
3. Отправили на вывод атрибуты экземпляра, используя переменную экземпляра `car`.
4. Отправили на вывод атрибуты класса, используя переменную экземпляра `car` и имя класса `Car`. Консольный вывод будет одинаковым в обоих случаях.

> ***Важное примечание***
>
> `self` — это ссылка на создаваемый экземпляр. Часто используется в Python для доступа к атрибутам и методам экземпляра в пределах метода экземпляра, включая метод `init`. Не является ключевым словом, и его использование необязательно. Вместо него может быть использовано любое другое слово, например, `this` или `blah`, кроме случаев, когда он должен быть первым параметром методов экземпляра, но соглашение об использовании `self` как аргумента слишком строгое.

Можно обновлять атрибуты класса, используя переменную экземпляра или имя класса, но результат может быть разным. Когда обновление происходит через имя класса, изменение распространяется на все экземпляры этого класса. Когда обновление осуществляется с помощью переменной экземпляра, изменение касается только этого экземпляра. Убедимся на примере следующего фрагмента кода с классом `Car`:

carexample3.py

```
#определение класса Car такое же как в carexample2.py
if __name__ == "__main__":
    car1 = Car ("blue", 1000)
    car2 = Car("red", 2000)

    print("using car1: " + car1.c_mileage_units)
    print("using car2: " + car2.c_mileage_units)
    print("using Class: " + Car.c_mileage_units)

    car1.c_mileage_units = "km"

    print("using car1: " + car1.c_mileage_units)
    print("using car2: " + car2.c_mileage_units)
    print("using Class: " + Car.c_mileage_units)

    Car.c_mileage_units = "NP"
    print("using car1: " + car1.c_mileage_units)
    print("using car2: " + car2.c_mileage_units)
    print("using Class: " + Car.c_mileage_units)
```

Консольный вывод программы можно проанализировать следующим образом:

1. Первый набор операторов `print` выводит значение атрибута класса по умолчанию — `Mi`.
2. После выполнения выражения `car1.c_mileage_units = "km"` значение атрибута класса останется прежним (`Mi`) для экземпляра `car2` и атрибута на уровне класса.
3. После выполнения выражения `Car.c_mileage_units = "NP"` значение атрибута класса изменится на `NP` и для `car2`, и на уровне класса, но останется прежним (`km`) для `car1`, потому что мы задали его явно.

> ***Важное примечание***
>
> Имена атрибутов начинаются с букв «`c`» и «`i`» с целью показать, что они принадлежат к классу (class) или экземпляру (instance), а не являются обычными локальными или глобальными переменными. Имена атрибутов экземпляра, не относящихся к типу *public*, но относящихся к типам *private* или *protected*, должны начинаться с одинарного или двойного подчеркивания. Мы поговорим об этом позже в этой главе.

Конструкторы и деструкторы классов

Python, как и другие языки ООП, использует *конструкторы* и *деструкторы*, но со своими особенностями именования. Целью конструкторов является инициализация или присвоение значений атрибутам на уровне класса или уровне экземпляра (в основном, второй вариант) при создании экземпляра класса. Метод `__init__` известен как конструктор и всегда выполняется при создании нового экземпляра. В Python поддерживаются три типа конструкторов:

- **Конструктор по умолчанию**: если конструктор (метод `__init__`) не включен в класс или не объявлен, класс использует пустой конструктор по умолчанию; он не делает ничего, кроме, инициализации экземпляра класса;
- **Конструктор без параметров**: этот тип конструктора не принимает никаких аргументов, кроме ссылки на создаваемый экземпляр; в следующем примере продемонстрирован конструктор без параметров для класса `Name`:

```
class Name:
    #конструктор без параметров
    def __init__(self):
        print("A new instance of Name class is \
         created")
```

- Поскольку с этим конструктором не передаются никакие аргументы, его функционал ограничен. Например, в нашем примере мы отправили на консоль сообщение, что для класса `Name` создан новый экземпляр;
- **Конструктор с параметрами**: такой тип конструктора может принимать один или несколько аргументов, а состояние экземпляра можно задать на основе входных аргументов, предоставленных методом конструктора; добавим в класс `Name` конструктор:

```
class Name:
    #конструктор с параметрами
    def __init__(self, first, last):
        self.i_first = first
        self.i_last = last
```

Деструкторы — это противоположность конструкторов. Они выполняются при удалении или уничтожении экземпляра. В Python деструкторы почти не используются, поскольку удалением неиспользуемых экземпляров занимается *сборщик мусора*. Если необходимо добавить логику внутри деструктора, можно использовать метод `__del__`. Он вызывается автоматически при удалении всех ссылок на экземпляр. Синтаксис определения деструктора в Python следующий:

```
def __del__(self):
print("Object is deleted.")
```

Различия между методами класса и методами экземпляра

В Python можно определить три типа методов:

- **Методы экземпляра**: связаны с экземпляром и требуют создания экземпляра перед их выполнением; принимают первый атрибут как ссылку на экземпляр (`self`) и могут считывать и обновлять состояние экземпляра; примером такого метода является `__init__`.
- **Методы класса**: объявляются с помощью *декоратора* `@classmethod`; для выполнения не нужен экземпляр класса; для этого метода ссылка на класс (по соглашению обозначается `cls`) будет автоматически отправлена в качестве первого аргумента.
- **Статические методы**: объявляются с помощью декоратора `@staticmethod`; методы не имеют доступа к объектам `cls` или `self`; они похожи на *функции полезности*, которые принимают определенные аргументы и предоставляют выходные данные на основе их значений; например, если нужно оценить конкретные входные данные или сделать парсинг данных для обработки, то для этих целей можно написать статические методы; они работают как обычные функции, которые определяются в модулях, но доступны в контексте пространства имен класса.

Рассмотрим пример, как эти методы могут быть определены и использованы:

```
#methodsexample1.py
class Car:
    c_mileage_units = "Mi"

    def __init__(self, color, miles):
        self.i_color = color
        self.i_mileage = miles

    def print_color (self):
        print (f"Color of the car is {self.i_color}")

    @classmethod
    def print_units(cls):
        print (f"mileage unit are {cls.c_mileage_unit}")
        print(f"class name is {cls.__name__}")

    @staticmethod
    def print_hello():
        print ("Hello from a static method")

if __name__ == "__main__":
    car = Car ("blue", 1000)
    car.print_color()
    car.print_units()
    car.print_hello()
```

```
    Car.print_color(car);
    Car.print_units();
    Car.print_hello()
```

В этой программе мы сделали следующее:

1. Создали класс `Car` с атрибутом класса (`c_mileage_units`), методом класса (`print_units`), статическим методом (`print_hello`), атрибутами экземпляра (`i_color` и `i_mileage`), методом экземпляра (`print_color`) и методом конструктора (`__init__`).
2. Создали экземпляр `car` класса `Car`, используя его конструктор.
3. Используя переменную экземпляра `car`, вызвали метод экземпляра, метод класса и статический метод.
4. Используя имя класса `Car`, снова вызвали метод экземпляра, метод класса и статический метод. Обратите внимание, метод экземпляра можно вызвать по имени класса, но для этого нужно передать переменную экземпляра в качестве первого аргумента (это также объясняет, зачем нужен аргумент `self` для каждого метода экземпляра).

Консольный вывод этой программы будет следующий:

```
Color of the car is blue
mileage unit are Mi
class name is Car
Hello from a static method
Color of the car is blue
mileage unit are Mi
class name is Car
Hello from a static method
```

Специальные методы

Когда мы определяем класс в Python и пытаемся вывести на консоль один из его экземпляров оператором `print`, мы получаем строку, содержащую имя класса и ссылку на экземпляр объекта, то есть адрес объекта в памяти. Не существует стандартной реализации функционала `to string`, доступной для экземпляра или объекта. Фрагмент кода, демонстрирующий это поведение:

```
#carexampl4.py
class Car:
    def __init__(self, color, miles):
        self.i_color = color
        self.i_mileage = miles
if __name__ == "__main__":
    car = Car ("blue", 1000)
    print (car)
```

Вывод консоли будет аналогичным выводу ниже, что не соответствует ожидаемому выводу от оператора `print`:

```
<__main__.Car object at 0x100caae80>
```

Для какого-либо осмысленного вывода из оператора `print` нужно реализовать специальный метод `__str__`, который вернет строку с информацией об экземпляре и который можно настроить по мере необходимости. Фрагмент кода из файла `carexample4.py` с методом `__str__`:

```
#carexample4.py
class Car:
    c_mileage_units = "Mi"
    def __init__(self, color, miles):
        self.i_color = color
        self.i_mileage = miles

    def __str__(self):
        return f"car with color {self.i_color} and \
         mileage {self.i_mileage}"
if __name__ == "__main__":
    car = Car ("blue", 1000)
    print (car)
```

Теперь консольный вывод оператора `print` будет таким:

```
car with color blue and mileage 1000
```

При правильной реализации `__str__` можно использовать `print` без специальных функций, наподобие `to_string()`. Такой способ управления преобразованием строк в духе Python. Другим популярным методом, используемым по схожим причинам, является `__repr__`, с помощью которого интерпретатор Python проверяет объект. Такой способ больше подходит для отладки.

Эти и некоторые другие методы называются *специальными* или *дандерами* (**Dunder**), так как они всегда начинаются и заканчиваются двойным символом подчеркивания (**Double under**). Обычные методы не должны использовать подобное именование. В некоторой литературе они иногда называются *магическими*. Существует несколько десятков специальных методов, доступных для реализации с классом. Полный их список доступен в официальной документации *Python 3* по адресу *https://docs.python.org/3/reference/datamodel.html#specialnames*.

В этом подразделе мы рассмотрели классы и объекты. В следующем мы поговорим о различных принципах объектно-ориентированного программирования в Python.

Принципы ООП

Объектно-ориентированное программирование — это способ объединения свойств и поведения в единую сущность, которая называется объектом. Для дос-

тижения наибольшей эффективности и модульности в Python доступно несколько принципов:

- Инкапсуляция данных.
- Наследование.
- Полимфоризм.
- Абстракция.

В следующих подразделах мы подробно рассмотрим каждый из этих принципов.

Инкапсуляция данных

Это фундаментальная концепция ООП, иногда также упоминается как *абстракция*. Это понятие включает в себя объединение данных и связанных с ними действий. На самом деле это нечто большее, чем просто объединение или абстракция. Здесь можно перечислить три ключевые задачи инкапсуляции:

- Объединение данных и действий, связанных с ними.
- Сокрытие внутренней структуры и деталей реализации объекта.
- Ограничение доступа к определенным компонентам (атрибутам или методам) объекта.

Инкапсуляция упрощает использование объектов с помощью сокрытия внутренних деталей ее реализации, а также помогает контролировать обновление состояний объекта.

Далее мы подробно рассмотрим эти задачи.

Объединение данных и действий

Для объединения данных и действий в одну инициализацию мы определяем атрибуты и методы в классе. Класс в Python может иметь следующие типы элементов:

- Конструктор и деструктор.
- Методы и атрибуты класса.
- Методы и атрибуты экземпляра.
- Вложенные классы.

В предыдущем разделе уже говорилось обо всех элементах класса, за исключением *вложенных* или *внутренних* классов. Мы рассмотрели примеры реализации конструкторов и деструкторов. Использовали атрибуты экземпляра для инкапсуляции данных в наших экземплярах или объектах. Мы также обсудили методы класса, статические методы и атрибуты класса с примерами кода в предыдущем разделе.

В завершение темы обсудим следующий фрагмент кода с вложенным классом. Возьмем класс `Car` и вложенный внутри него класс `Engine`.

Каждому автомобилю нужен двигатель, поэтому имеет смысл сделать его внутренним:

```
#carwithinnerexamplel.py
class Car:
    """внешний класс"""
    c_mileage_units = "Mi"
    def __init__(self, color, miles, eng_size):
        self.i_color = color
        self.i_mileage = miles
        self.i_engine = self.Engine(eng_size)

    def __str__(self):
        return f"car with color {self.i_color}, mileage \
        {self.i_mileage} and engine of {self.i_engine}"

    class Engine:
        """внутренний класс"""
        def __init__(self, size):
            self.i_size = size

        def __str__(self):
            return self.i_size

if __name__ == "__main__":
    car = Car ("blue", 1000, "2.5L")
    print(car)
    print(car.i_engine.i_size)
```

В примере определен внутренний класс `Engine` внутри обычного класса `Car`. Класс `Engine` имеет только один атрибут `i_size`, метод конструктора (`__init__`) и метод `__str__`. По сравнению с предыдущими примерами в класс `Car` мы внесли следующие изменения:

- Метод `__init__` включает новый атрибут для объема двигателя; также была добавлена строка для создания нового экземпляра `Engine`, связанного с экземпляром `Car`.
- Метод `__str__` класса `Car` включает в себя атрибут внутреннего класса (`i_size`).

Основная программа имеет оператор `print` для экземпляра `Car`, а также имеет строку для вывода значения атрибута `i_size` класса `Engine`. Консольный вывод будет следующим:

```
car with color blue, mileage 1000 and engine of 2.5L
2.5L
```

Вывод показывает, что у нас есть доступ к внутреннему классу изнутри реализации класса и мы можем обратиться к атрибутам внутреннего класса снаружи.

Далее поговорим, как можно скрыть некоторые атрибуты и методы, и они не будут доступны и видимы за пределами класса.

Сокрытие информации

Из предыдущих примеров видно, что доступ ко всем атрибутам на уровне класса и экземпляра не имеет ограничений. Такой подход приводит к плоскому дизайну, а класс становится просто оберткой вокруг переменных и методов. Лучший подход в ООП состоит в сокрытии определенных атрибутов экземпляра и предоставлении доступа извне только к тем из них, которые необходимы. Для понимания, как это реализуется в Python, рассмотрим два термина: *частный* (**private**) и *защищенный* (**protected**).

Частные переменные и методы

Частная переменная или атрибут могут быть определены двойным символом подчеркивания перед именем переменной. В Python нет ключевого слова наподобие `private`, как в других языках программирования. Переменные класса и экземпляра могут быть лишь помечены как частные.

Такой метод может быть определен двойным символом подчеркивания перед именем метода. Он может быть вызван только внутри класса и недоступен за его пределами.

Каждый раз при определении атрибута или метода как частного интерпретатор Python не разрешает доступ к такому атрибуту или методу за пределами определения класса. Ограничение также распространяется на подклассы. Поэтому доступ к частным атрибутам и методам есть только у кода внутри класса.

Защищенные переменные и методы

Защищенные переменные или методы помечаются одинарным символом подчеркивания в начале имени. Они должны быть доступны коду внутри определения классов и подклассов. Например, если надо преобразовать атрибут `i_color` из *public* в *protected*, достаточно просто поменять его имя на `_i_color`. Интерпретатор Python не навязывает такое использование защищенных элементов внутри класса или подкласса. Речь, скорее, о соблюдении соглашений об именовании и использовании защищенных атрибутов и методов в соответствии с их определением.

Используя частные или защищенные переменные и методы, можно скрывать некоторые детали реализации объекта. Это полезно, поскольку позволяет иметь компактный и чистый код внутри большого класса, не открывая все содержимое для внешнего мира. Еще одна причина скрывать атрибуты — контроль обновления и доступа к ним. В заключение рассмотрим новый вариант класса `Car` с частными и защищенными переменными и частным методом:

```
#carexample5.py
class Car:
```

```
    c_mileage_units = "Mi"
    __max_speed = 200

    def __init__(self, color, miles, model):
        self.i_color = color
        self.i_mileage = miles
        self.__no_doors = 4
        self._model = model

    def __str__(self):
        return f"car with color {self.i_color}, mileage
          {self.i_mileage}, model {self._model} and doors
            {self.__doors()}"

def __doors(self):
        return self.__no_doors
if __name__ == "__main__":
    car = Car ("blue", 1000, "Camry")
    print (car)
```

Мы добавили в класс `Car` следующие элементы:

- частная переменная `__max_speed` (максимальная скорость) со значением по умолчанию;
- частная переменная `__no_doors` (нет дверей) со значением по умолчанию внутри метода конструктора `__init__`;
- защищенная переменная `_model` (модель), добавленная только для демонстрации;
- частный метод `__doors()` в экземпляре для хранения информации о количестве дверей;
- метод `__str__` дополнен частным методом `__doors()` для получении информации о количестве дверей.

Консольный вывод будет предсказуемым. Но если попробовать обратиться к частным методам или переменным из основной программы, они будут недоступны, а интерпретатор выдаст ошибку. Такое поведение ожидаемо, поскольку цель частных переменных и методов — быть доступными только внутри класса.

> ***Важное примечание***
>
> Python не делает переменные и методы закрытыми, а только создает видимость этого. Он фактически искажает имена переменных с именем класса так, что они не видны в классе, который их содержит.

В примере с классом `Car` можно обратиться к частным переменным и методам. Python предоставляет доступ к ним за пределами определения класса с другим именем атрибута, которое состоит из символа подчеркивания вначале, имени класса и

имени частного атрибута. Таким же образом можно получить доступ к частным методам.

Строки кода ниже допустимы, но не поощряются и противоречат определениям *private* и *protected*:

```
print (Car._Car__max_speed)
print (car._Car__doors())
print (car._model)
```

Как видно, `_Car` добавлена перед фактическим именем частной переменной. Это сделано для минимизации конфликтов с переменными и во внутренних классах.

Защита данных

В предыдущих примерах понятно, что можно без ограничений получать доступ к атрибутам экземпляра. И также нет ограничений на использование реализованных методов. Поэтому мы маскируем их под частные или защищенные для сокрытия от влияния извне.

Но в реальных задачах нужно предоставить доступ к переменным так, что этот процесс можно было легко контролировать и обслуживать. Во многих ООП-языках это достигается с помощью *модификаторов доступа — геттеров* и *сеттеров*:

- **Геттеры** (от англ. get — получать) используются для доступа к частным атрибутам класса или его экземпляра.
- **Сеттеры** (от англ. set — задавать) используются для задания частных атрибутов класса или его экземпляра.

С помощью этих модификаторов можно реализовать дополнительную логику для доступа и установки атрибутов, которую к тому же удобно хранить в одном месте. Есть два способа реализовать модификаторы: *традиционный* и с помощью *декораторов*.

Традиционный подход к использованию геттеров и сеттеров

В традиционном подходе мы пишем методы экземпляров с префиксами `get` и `set`, за которыми следует символ подчеркивания и имя переменной. Изменим наш класс `Car` для использования модификаторов доступа к атрибутам экземпляра следующим образом:

```
#carexample6.py
class Car:
    __mileage_units = "Mi"
    def __init__(self, col, mil):
        self.__color = col
        self.__mileage = mil
```

```
    def __str__(self):
        return f"car with color {self.get_color()} and \
         mileage {self.get_mileage()}"

    def get_color(self):
        return self.__color

    def get_mileage(self):
        return self.__mileage

    def set_mileage (self, new_mil):
            self.__mileage = new_mil

if __name__ == "__main__":
    car = Car ("blue", 1000)

    print (car)
    print (car.get_color())
    print(car.get_mileage())
    car.set_mileage(2000)
    print (car.get_color())
    print(car.get_mileage())
```

Мы внесли в класс `Car` следующие изменения:

- Атрибуты экземпляра `color` и `mileage` добавлены как частные переменные.
- Добавлены геттеры для атрибутов экземпляра `color` и `mileage`.
- Добавлен сеттер только для атрибута `mileage`, поскольку `color` задается один раз при создании объекта.
- В основной программе получили данные для нового экземпляра класса, используя геттеры; затем обновили пробег, используя сеттер; и снова получили данные для атрибутов `color` и `mileage`.

На консоли видно вполне ожидаемые выходные данные. Мы не стали определять сеттеры для всех атрибутов, а сделали их только там, где это требуется и имеет смысл. В ООП использование геттеров и сеттеров является лучшей практикой, но в Python они не очень популярны. Культура разработчиков по-прежнему заключается в доступе к атрибутам напрямую.

Использование декоратора property

Использование декораторов для определения геттеров и сеттеров — это современный подход, который позволяет реализовать код в духе Python.

Существует декоратор `@property`, с которым код выглядит проще и аккуратнее. Дополним им класс `Car`:

```
#carexample7.py
class Car:
    __mileage_units = "Mi"
    def __init__(self, col, mil):
        self.__color = col
        self.__mileage = mil

    def __str__(self):
        return f"car with color {self.color} and mileage \
         {self.mileage}"

    @property
    def color(self):
        return self.__color

    @property
    def mileage(self):
        return self.__mileage

    @mileage.setter
    def mileage (self, new_mil):
            self.__mileage = new_mil

if __name__ == "__main__":
    car = Car ("blue", 1000)

    print (car)
    print (car.color)
    print(car.mileage)
    car.mileage = 2000
    print (car.color)
    print(car.mileage)
```

Мы внесли следующие изменения:

- Сделали атрибуты экземпляра частными переменными.
- Добавили геттер-методы для `color` и `mileage`, используя декоратор `@property` и имя атрибута в качестве имени метода.
- Добавили сеттер-методы для `mileage` с помощью декоратора `@mileage.setter`, присвоив методу то же имя, что и имя атрибута.

В основном скрипте мы обращаемся к атрибутам `color` и `mileage` по имени экземпляра, за которым следует точка и имя атрибута (в стиле Python). Это делает синтаксис кода лаконичным и читабельным. Использование декораторов также упрощает имена методов.

Итак, мы обсудили все аспекты инкапсуляции в Python и использование классов для объединения данных и действий. Также мы рассмотрели, как скрывать ненужную информацию от внешнего мира и защищать данные в классе с помощью геттеров, сеттеров и декоратора property. Далее поговорим, как реализуется наследование.

Расширение классов с помощью наследования

Концепция похожа на наследование в реальном мире, где дети получают некоторые характеристики своих родителей в дополнение к собственным характеристикам.

Точно также класс может быть дополнен элементами (атрибуты и методы) другого класса. Класс, от которого мы наследуем, называется *родительским*, *базовым* или *суперклассом*. Класс, который мы наследуем, называется *производным*, *дочерним* или *подклассом*. Ниже приводится простая схема отношений между родительским и дочерним классом (рис. 3.1):



Рис. 3.1. Отношения между родительским и дочерним классом

В Python дочерний класс обычно наследует все элементы родительского, но этим поведением можно управлять, используя соглашение об именовании (например, использовать двойное подчеркивание) и модификаторы доступа.

Наследование бывает двух типов: *простое* и *множественное*. Рассмотрим каждый вариант поподробнее.

Простое наследование

В данном случае класс является производным от одного родителя. Это распространенная форма наследования в ООП, которая больше похожа на генеалогическое дерево людей. Синтаксис родительского и дочернего класса при простом наследовании:

```
class BaseClass:
    <атрибуты и методы базового класса >
class ChildClass (BaseClass):
    <атрибуты и методы дочернего класса >
```

Изменим наш пример класса `Car`, сделав его производным от родительского класса `Vehicle` (транспортное средство). Также добавим еще один дочерний класс `Truck` (грузовик) для демонстрации подробностей концепции наследования:

```
#inheritance1.py
class Vehicle:
```

```
    def __init__(self, color):
        self.i_color = color

    def print_vehicle_info(self):
        print(f"This is vehicle and I know my color is \
         {self.i_color}")

class Car (Vehicle):
    def __init__(self, color, seats):
        self.i_color = color
        self.i_seats = seats

    def print_me(self):
        print( f"Car with color {self.i_color} and no of \
         seats {self.i_seats}")

class Truck (Vehicle):
    def __init__(self, color, capacity):
        self.i_color = color
        self.i_capacity = capacity

    def print_me(self):
        print( f"Truck with color {self.i_color} and \
         loading capacity {self.i_capacity} tons")

if __name__ == "__main__":
    car = Car ("blue", 5)
    car.print_vehicle_info()
    car.print_me()
    truck = Truck("white", 1000)
    truck.print_vehicle_info()
    truck.print_me()
```

В этом примере мы создали родительский класс `Vehicle` с одним атрибутом `i_color` и одним методом `print_vehicle_info`. Оба элемента имеют возможность наследования. Затем мы создали два дочерних класса: `Car` и `Truck`. Каждый из них имеет по одному дополнительному атрибуту (`i_seats` и `i_capacity`, соответственно) и один дополнительный метод (`print_me`). В методах `print_me` в каждом дочернем классе мы получаем доступ к атрибуту экземпляра базового класса и к атрибутам экземпляра подкласса.

Такой дизайн является преднамеренным для уточнения идеи наследования элементов от суперкласса и добавления отдельных элементов индивидуально в подкласс. Использование двух дочерних классов показывает роль наследования в отношении возможности повторного использования.

В основной программе мы создали экземпляры `Car` и `Truck` и попытались обратиться к родительскому методу, а также к методу экземпляра. Ожидаемый консольный вывод показан ниже:

```
This is vehicle and I know my color is blue
Car with color blue and no of seats 5
This is vehicle and I know my color is white
Truck with color white and loading capacity 1000 tons
```

Множественное наследование

При множественном наследовании дочерний класс может быть производным от нескольких родителей. Эта концепция применяется в сложных объектно-ориентированных проектах, где объекты связаны с множеством других объектов. Но нужно быть осторожным при наследовании от нескольких классов, особенно, если они наследуются от общего суперкласса. Может возникнуть проблема *ромбовидного наследования* (**diamond inheritance**). Это ситуация, когда мы создаем класс **X**, наследуя его от двух классов **Y** и **Z**, которые, в свою очередь, наследуются от общего класса **A**. Тогда для класса **X** возникает неопределенность относительно кода в классе **A**, который он наследует через классы **Y** и **Z**. Множественное наследование не поощряется из-за возможных проблем, которые оно вызывает.

Для демонстрации изменим классы `Vehicle` и `Car`, а также добавим родительский класс `Engine`. Код с множественным наследованием будет следующий:

```
#inheritance2.py
class Vehicle:
    def __init__(self, color):
        self.i_color = color

    def print_vehicle_info(self):
        print( f"This is vehicle and I know my color is \
         {self.i_color}")

class Engine:
    def __init__(self, size):
         self.i_size = size

    def print_engine_info(self):
        print(f"This is Engine and I know my size is \
         {self.i_size}")

class Car (Vehicle, Engine):
    def __init__(self, color, size, seat):
        self.i_color = color
        self.i_size = size
        self.i_seat = seat
```

```
    def print_car_info(self):
        print(f"This car of color {self.i_color} with \
         seats {self.i_seat} with engine of size \
         {self.i_size}")

if __name__ == "__main__":
    car = Car ("blue", "2.5L", 5 )
    car.print_vehicle_info()
    car.print_engine_info()
    car.print_car_info()
```

В этом примере мы создали два родительских класса: `Vehicle` и `Engine`. Класс `Vehicle` такой же, как в предыдущем примере. Класс `Engine` имеет один атрибут `i_size` и один метод `print_engine_info`. Класс `Car` наследуется от обоих классов `Vehicle` и `Engine`. Он имеет один дополнительный атрибут (`i_seats`) и один дополнительный метод (`print_car_info`). В методе экземпляра у нас есть доступ к атрибутам экземпляра обоих родительских классов.

В основной программе мы создали экземпляр класса `Car`. С помощью него мы можем получить доступ к методам экземпляра родительских и дочерних классов. На консоли мы увидим ожидаемые выходные данные:

```
This is vehicle and I know my color is blue
Car with color blue and no of seats 5
This is vehicle and I know my color is white
Truck with color white and loading capacity 1000 tons
```

В этом разделе мы рассмотрели два типа наследования — простое и множественное. Теперь поговорим о полиморфизме в Python.

Полиморфизм

В буквальном смысле *полиморфизм* означает *«множество форм»*. В ООП полиморфизм — это способность экземпляра вести себя по-разному и использовать один и тот же метод с одними и теми же именем и аргументами для выбора поведения в соответствии с классом, к которому он принадлежит.

Эту концепцию можно реализовать двумя способами: *перегрузкой метода* и *переопределением метода*. Рассмотрим каждый вариант поподробнее.

Перегрузка метода

Это способ достичь полиморфизма за счет наличия нескольких методов с одинаковыми именами, но разным типом или количеством аргументов. В Python нет простого способа реализовать такой подход, поскольку два метода не могут иметь одно имя. Все является объектом, включая классы и методы. При написании методов для

класса с точки зрения пространства имен они являются его атрибутами, следовательно, одинаковых имен быть не может. Если создать два метода с одним именем, синтаксической ошибки не будет, второй просто заменит первый.

Внутри класса метод можно перегрузить, задав значение по умолчанию аргументам. Это не лучший способ реализации, но он работает. Ниже приведен пример перегрузки метода внутри класса в Python:

```
#methodoverloading1.py
class Car:
    def __init__(self, color, seats):
        self.i_color = color
        self.i_seat = seats

    def print_me(self, i='basic'):
        if(i =='basic'):
            print(f"This car is of color {self.i_color}")
        else:
            print(f"This car is of color {self.i_color} \
             with seats {self.i_seat}")

if __name__ == "__main__":
    car = Car("blue", 5 )
    car.print_me()
    car.print_me('blah')
    car.print_me('detail')
```

В этом примере добавлен метод `print_me` с аргументом, имеющим значение по умолчанию, которое будет использоваться, если в метод не передаются никакие параметры. Если `print_me` не получит параметры, на консоли мы увидим только цвет экземпляра `Car`. Когда методу передается аргумент (независимо от его значения), поведение меняется, и теперь метод предоставляет цвет и количество мест для экземпляра `Car`. Вот что будет на консоли:

```
This car is of color blue
This car is of color blue with seats 5
This car is of color blue with seats 5
```

> ***Важное примечание***
>
> В Python можно использовать сторонние библиотеки (например, `overload`) для более аккуратной реализации перегрузки методов.

Переопределение метода

Наличие метода с одинаковым именем в родительском и дочернем классах называется *переопределением метода*. Предполагается, что реализация методов на разных уровнях классов отличается. При вызове переопределяющего метода для экземпля-

ра дочернего класса интерпретатор ищет этот метод в области определения на соответствующем уровне. Метод выполняется на уровне дочернего класса. Если интерпретатор не находит метод на уровне дочернего экземпляра, он ищет его на уровне родительского. Если метод переопределен в дочернем классе с помощью экземпляра его же уровня, но мы хотим выполнить его в родительском, можно использовать метод `super()` для доступа к методу на уровне родительского класса. В Python это более популярный тип полиморфизма, который используется наряду с наследованием и позволяет эффективно его использовать.

Для демонстрации переопределения метода изменим код в `inhertance1.py`, переименовав имя метода `print_vehicle_info` в `print_me`. Как мы знаем, методы `print_me` уже присутствуют в двух дочерних классах с разными реализациями. Ниже приведен обновленный код с выделенными изменениями:

```
#methodoverriding1.py
class Vehicle:
    def __init__(self, color):
        self.i_color = color

    def print_me(self):
        print(f"This is vehicle and I know my color is \
         {self.i_color}")

class Car (Vehicle):
    def __init__(self, color, seats):
        self.i_color = color
        self.i_seats = seats

    def print_me(self):
        print( f"Car with color {self.i_color} and no of \
         seats {self.i_seats}")

class Truck (Vehicle):
    def __init__(self, color, capacity):
        self.i_color = color
        self.i_capacity = capacity

    def print_me(self):
        print( f"Truck with color {self.i_color} and \
         loading capacity {self.i_capacity} tons")

if __name__ == "__main__":
    vehicle = Vehicle("red")
    vehicle.print_me()
    car = Car ("blue", 5)
    car.print_me()
    truck = Truck("white", 1000)
    truck.print_me()
```

В этом примере мы переопределяем метод `print_me` в дочерних классах. Когда мы создаем три разных экземпляра классов `Vehicle`, `Car` и `Truck` и выполняем один и тот же метод, мы получаем разное поведение. Вот что будет в консольном выводе:

```
This is vehicle and I know my color is red
Car with color blue and no of seats 5
Truck with color white and loading capacity 1000 tons
```

Переопределение метода имеет много практических применений. Например, можно наследовать встроенный класс `list` и переопределить его методы для расширения функционала. *Выборочная сортировка* является примером переопределения метода для объекта `list`. В следующих главах мы рассмотрим несколько примеров переопределения метода.

Абстракция

Это еще одна мощная возможность ООП, которая позволяет скрыть детали реализации и показать только необходимые или высокоуровневые функции объекта. Примером из реальной жизни служит автомобиль, который мы имеем, с его основным функционалом для нас как водителя, но без реальных знаний того, как все устроено.

Эта концепция связана с инкапсуляцией и наследованием вместе, именно поэтому мы оставили ее напоследок. Еще одна причина выделить абстракцию в отдельную тему — подчеркнуть использование абстрактных классов в Python.

Абстрактные классы в Python

Абстрактный класс выступает шаблоном для других классов. С его помощью можно создать набор *абстрактных методов* (пустых), которые будут реализованы дочерним классом. Простыми словами, абстрактным называется класс, который содержит один или несколько абстрактных методов. У таких методов есть объявление, но нет реализации.

В таком классе могут быть методы, которые уже реализованы и могут использоваться дочерним классом (как есть) с помощью наследования. Данная концепция помогает реализовать общие *программные интерфейсы приложения* (**Application Programming Interface**, **API**), а также определять в одном месте общую базу кода, которую можно повторно использовать в дочерних классах.

> ***Примечание***
>
> Экземпляры абстрактных классов создавать нельзя.

Абстрактный класс можно реализовать с помощью встроенного модуля *ABC* (***Abstract Base Classes***) из пакета `abc`. Этот пакет также включает модуль abstractmethod, который использует декораторы для объявления абстрактных методов.

Ниже показан простой пример с использование модуля ABC и декоратора abstractmethod:

```
#abstraction1.py
from abc import ABC, abstractmethod

class Vehicle(ABC):
    def hello(self):
        print(f"Hello from abstract class")
    @abstractmethod
    def print_me(self):
        pass

class Car (Vehicle):
    def __init__(self, color, seats):
        self.i_color = color
        self.i_seats = seats

    """Необходимо реализовать этот метод"""
    def print_me(self):
        print( f"Car with color {self.i_color} and no of \
         seats {self.i_seats}")
if __name__ == "__main__":
   # vehicle = Vehicle()    #это невозможно
   # vehicle.hello()
    car = Car ("blue", 5)
    car.print_me()
    car.hello()
```

В этом примере мы сделали следующее:

- Сделали класс Vehicle абстрактным, унаследовав его от класса ABC, а также объявили один из методов (print_me) абстрактным с помощью декоратора @abstractmethod.
- Обновили знакомый класс Car, реализовав в нем метод print_me, и сохранив весь остальной код как в предыдущем примере.
- В основной части программы попытались создать экземпляр класса Vehicle (код прокомментирован на иллюстрации); создали экземпляр класса Car и выполнили методы print_me и hello.

При попытке создать экземпляр класса Vehicle вылезет ошибка, что невозможно создать экземпляр абстрактного класса Vehicle с абстрактным методом print_me:

```
Can't instantiate abstract class Vehicle with abstract methods print_me
```

Также при попытке *не* реализовывать метод print_me в дочернем классе Car мы получим ошибку. Для экземпляра класса Car мы имеем ожидаемый консольный вывод из методов print_me и hello.

Композиция как альтернативный подход к проектированию

Это еще одна популярная концепция, которая также имеет отношение к инкапсуляции. Простыми словами, композиция означает включение одного или нескольких объектов в другой объект для образования объекта из реального мира. Класс, который включает другие объекты класса, называется *составным* (или композитным). А классы, объекты которых входят в составной класс, называются *компонентами*. На скриншоте приведен пример составного класса с тремя компонентами — A, B и C (рис. 3.2):

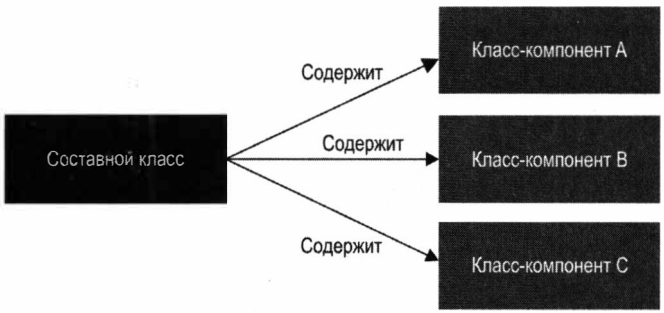


Рис. 3.2. Отношения между составным классом и классами-компонентами

Композиция считается альтернативным подходом к наследованию. Обе концепции устанавливают отношения между объектами. В случае наследования — объекты тесно связаны, так как изменения в родительских классах могут нарушить код в дочерних. При композиции — объекты связаны слабо, что облегчает внесение изменений в один класс без нарушений кода в другом. Благодаря своей гибкости композиционный подход довольно популярен, но это не означает, что он подходит для решения всех задач. Возникает вопрос, какой подход лучше использовать для конкретной проблемы? Для этого есть правило. Когда есть *отношения* между объектами, правильным выбором будет наследование. Например, **автомобиль** *является* **транспортным средством**, а **кот** *является* **животным**. В этом случае дочерний класс является расширением родительского с дополнительным функционалом и возможностью повторного использования функций родительского класса. Если отношение между объектами такое, что один объект *содержит* другой, лучше использовать композицию. Например, **автомобиль** *содержит* **аккумулятор**.

В предыдущем примере с множественным наследием мы реализовали класс `Car` как потомок класса `Engine`, что является не лучшим вариантом использования наследования. Поэтому попробуем использовать композицию и реализовать класс `Car` с объектом `Engine` внутри. У нас может быть еще один класс `Seat`, который может входить в класс `Car`.

Рассмотрим эту концепцию в следующем примере, где мы создаем класс `Car`, который содержит `Engine` и `Seat` внутри себя:

```
#composition1.py
class Seat:
    def __init__(self, type):
        self.i_type = type

    def __str__(self):
        return f"Seat type: {self.i_type}"

class Engine:
    def __init__(self, size):
        self.i_size = size

    def __str__(self):
        return f"Engine: {self.i_size}"

class Car:
    def __init__(self, color, eng_size, seat_type):
        self.i_color = color
        self.engine = Engine(eng_size)
        self.seat = Seat(seat_type)

    def print_me(self):
        print(f"This car of color {self.i_color} with \
         {self.engine} and {self.seat}")

if __name__ == "__main__":
    car = Car ("blue", "2.5L", "leather" )
    car.print_me()
    print(car.engine)
    print(car.seat)
    print(car.i_color)
    print(car.engine.i_size)
    print(car.seat.i_type)
```

Проанализируем этот фрагмент кода:

1. Мы определили классы `Engine` и `Seat` и указали в каждом по одному атрибуту: `i_size` и `i_type` соответственно.
2. Затем определили класс `Car`, добавив атрибут `i_color`, экземпляр `Engine` и экземпляр `Seat`. Экземпляры `Engine` и `Seat` были созданы одновременно с экземпляром `Car`.

3. В основной программе мы создали экземпляр `Car` и реализовали следующие действия:
   - `car.print_me`: обращается к методу `print_me` экземпляра `Car`;
   - `print(car.engine)`: выполняет метод `__str__` класса `Engine`;
   - `print(car.seat)`: выполняет метод `__str__` класса `Seat`;
   - `print(car.i_color)`: обращается к атрибуту `i_color` экземпляра `Car`;
   - `print(car.engine.i_size)`: обращается к атрибуту `i_size` экземпляра `Engine` внутри экземпляра `Car`;
   - `print(car.seat.i_type)`: обращается к атрибуту `i_type` экземпляра `Seat` внутри экземпляра `Car`.

Консольный вывод получится следующим:

```
This car of color blue with Engine: 2.5L and Seat type: leather
Engine: 2.5L
Seat type: leather
blue
2.5L
leather
```

Далее обсудим *утиную типизацию*, которая является альтернативой полиморфизму.

Утиная типизация в Python

Утиная типизация, в основном, используется в языках, которые поддерживают *динамическую типизацию*, например, Python и JavaScript. Название заимствовано из следующей цитаты:

> *«Если это выглядит как утка, плавает как утка*
> *и крякает как утка, то, вероятно, это и есть утка».*

Смысл цитаты в том, что объект можно идентифицировать по его поведению. Если птица ведет себя как утка, то она, скорее всего, утка. Это и есть основной принцип утиной типизации.

В данном случае тип класса объекта менее важен, чем метод (поведение), который им определен. Тип объекта не проверяется, но выполняется метод, который ожидается.

Для демонстрации рассмотрим простой пример с тремя классами `Car`, `Cycle` и `Horse` (машина, велосипед и лошадь, соответственно). И попытаемся реализовать для каждого из них метод `start`. В классе `Horse` вместо `start` мы назовем метод `push`. Ниже приведен фрагмент кода с тремя классами и основной программой в конце:

```
#ducttype1.py
class Car:
```

```
    def start(self):
        print ("start engine by ignition /battery")

class Cycle:
    def start(self):
        print ("start by pushing paddles")

class Horse:
    def push(self):
        print ("start by pulling/releasing the reins")

if __name__ == "__main__":
    for obj in Car(), Cycle(), Horse():
        obj.start()
```

В основной программе мы пытаемся динамически перебирать экземпляры классов и вызывать метод `start`. Как и ожидалось, строка `obj.start()` не сработала для объекта `Horse`, поскольку в классе нет такого метода. В этом примере видно, что можно помещать разные типы классов или экземпляров в один оператор и выполнять методы для них.

Если изменить метод с именем `push` на `start` в классе `Horse`, основная программа будет выполняться без ошибок. Утиная типизация применяется во многих сценариях для упрощения решений. Например, при использовании итераторов и метода `len` ко многим объектам. Мы подробно остановимся на итераторах в следующей главе.

До этого момента мы рассматривали понятия и принципы ООП, а также их преимущества. Далее речь пойдет о ситуациях, когда их использование не очень выгодно.

Когда не стоит использовать ООП в Python

Python является языком, который обладает гибкостью и может быть использован как для объектно-ориентированного программирования (Java), так и для декларативного программирования (C). ООП всегда привлекает разработчиков, поскольку обладает такими мощными инструментами, как инкапсуляция, абстракция, наследование и полиморфизм. Но они подходят не для всех задач. Лучше всего использовать их для больших и сложных приложений, особенно если у них есть *пользовательский интерфейс*.

Если программа больше похожа на скрипт для выполнения определенных задач и нет необходимости хранить состояния объектов, использование ООП излишне. Например, приложения для *Data Science* и *интенсивной обработки данных* не требуют использования принципов ООП. Для них гораздо важнее определить, как выполнять задачи в определенном порядке для достижения целей. Примером из реальной жизни служит написание клиентских программ для выполнения задач по

обработке больших объемов данных в кластере узлов вроде **Apache Spark** для параллельной обработки. Мы рассмотрим эти типы приложений в следующих главах. Несколько дополнительных сценариев, где использование ООП не обязательно:

- Чтение файла, применение логики и запись в новый файл — тип программы, которую проще реализовать с помощью функций в модуле, чем с помощью ООП.
- Настройка устройств с использованием Python очень популярна и может быть решена с помощью обычных функций.
- Анализ и преобразование данных из одного формата в другой также можно выполнить с помощью декларативного программирования вместо ООП.
- Перенос старой кодовой базы на новую с помощью ООП — также не самая лучшая идея; если старый код написан без использования шаблонов ООП, на выходе получатся функции, не связанные с ООП, заключенные в классы и объекты, которые трудно обслуживать и расширять.

Важно сначала проанализировать постановку задачи и требования, прежде чем решить, использовать ООП или нет. Это также зависит от того, какие сторонние библиотеки будут использованы в программе. Если необходимо расширять классы из сторонней библиотеки, без ООП не обойтись.

Заключение

В этой главе мы изучили концепцию классов, а также обсудили, как создавать их и использовать для создания объектов и экземпляров. Затем мы рассмотрели четыре столпа ООП: инкапсуляцию, наследование, полиморфизм и абстракцию. Мы также рассмотрели простые и понятные примеры кода для лучшего понимания этих концепций, которые являются фундаментальными для ООП в Python.

Узнали про утиную типизацию, которая демонстрирует отсутствие зависимости от классов, а затем поговорили о задачах, где не следует использовать ООП.

В этой главе вы не только освежили свои знания об основных понятиях, но и узнали, как реализовать их в синтаксисе Python. В следующей главе мы рассмотрим несколько библиотек для продвинутого программирования.

Вопросы

1. Что такое класс и объект?
2. Что такое dunder?
3. Поддерживает ли Python наследование от нескольких классов?
4. Можно ли создать экземпляр абстрактного класса?
5. Правда ли, что в утиной типизации важен тип класса?

Дополнительные ресурсы

- *Modular Programming with Python*, автор: Эрик Вестра (Erik Westra).
- *Python 3 Object-Oriented Programming*, автор: Дасти Филлипс (Dusty Phillips).
- *Learning Object-Oriented Programming*, автор: Гастон С. Хиллар (Gaston C. Hillar).
- *Python for Everyone — Third edition*, авторы: Кэй Хорстманн (Cay Horstmann) и Ранс Некэз (Rance Necaise).

Ответы

1. Класс — это схема для указания интерпретатору, как что-то должно быть определено. Объект — это экземпляр, созданный из класса на основе того, что в нем определено.
2. Dunder — это специальный метод, который всегда начинается и заканчивается двойным подчеркиванием. Для каждого класса доступно несколько десятков специальных методов.
3. Да, Python поддерживает наследование от нескольких классов.
4. Нет, мы не можем создать экземпляр абстрактного класса.
5. Неправда, в утиной типизации методы важнее классов.

Раздел 2

Расширенные концепции программирования

В этом разделе мы продолжим наше путешествие и изучим передовые концепции Python. Коснемся таких понятий, как итераторы, генераторы, обработка ошибок и исключений. Это поможет вам выйти на новый уровень программирования. Помимо написания кода мы также изучим создание и автоматизацию модульных и интеграционных тестов с помощью таких фреймворков, как **unittest** и **pytest**. В последней главе раздела рассмотрим расширенные функции для преобразования данных и создания декораторов в Python, а также способы использования структур данных, включая **pandas DataFrames**, для аналитических приложений.

Этот раздел содержит следующие главы:

- «Глава 4: Библиотеки Python для решения сложных задач».
- «Глава 5: Тестирование и автоматизация с помощью Python».
- «Глава 6: Расширенные советы и приемы в Python».

4

Библиотеки Python для продвинутого программирования

В предыдущих главах мы рассмотрели разные подходы к созданию модульных и многоразовых программ. В этой главе мы рассмотрим несколько понятий, таких, как итераторы, генераторы, ведение журналов (логирование) и обработка ошибок. Это позволит писать эффективный код, пригодный для повторного использования. Предполагается, что вы уже знакомы с синтаксисом Python и умеете писать управляющие структуры.

Мы изучим работу циклов, обработку файлов, как лучше всего их открывать и получать доступ к ним. А также, как обрабатывать ошибки, которые могут быть как ожидаемыми, так и неожиданными. Мы также поговорим о логировании и о способах, как настроить системы ведения журналов и регистрации событий. Эта глава научит вас использовать библиотеки для создания сложных проектов.

Темы этой главы:

- Введение в контейнеры данных Python.
- Итераторы и генераторы для обработки данных.
- Обработка файлов в Python.
- Обработка ошибок и исключений.
- Модуль logging в Python.

Технические требования

В этой главе вам понадобится:

- Python 3.7 или более поздней версии.

Примеры кода для этой главы:
https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter04.

Начнем с изучения контейнеров данных, которые пригодятся в последующих темах этой главы.

Введение в контейнеры данных Python

Python поддерживает несколько типов данных, как числовых, так и коллекционных (наборы). Определение числовых типов основано на присвоении переменной значения, Это значение и будет определять числовой тип данных, например, с плавающей точкой или целое. Специальный конструктор (например, `int()` или `float()`) также может использоваться для создания переменной определенного типа. Контейнерные типы данных тоже можно определить, либо присвоив значение в соответствующем формате, либо с помощью специального конструктора для каждого типа в коллекции. В этом разделе мы рассмотрим пять различных контейнерных типов данных: *строки*, *списки*, *кортежи*, *словари* и *множества*.

Строки

Строка (**String**), по сути, не является набором (далее слова *коллекция* и *набор* можно считать синонимами), но мы рассмотрим ее, поскольку она широко используется в Python и реализуется с помощью неизменной последовательности кодовых точек Юникода. Так как строка использует последовательность (коллекцию), мы изучим ее в этом разделе.

Строки в Python являются неизменными. Благодаря этому они обеспечивают безопасное выполнение *конкурентных* программ, где множество функций могут обращаться к одному и тому же строковому объекту и давать одинаковый результат. С изменяемыми объектами такой подход невозможен. Поскольку строки неизменяемы, их часто используют как ключи для словарей или как элементы множеств. Недостаток заключается в том, что приходится создавать новый экземпляр, даже если нужно внести небольшое изменение.

> ***Примечание***
>
> Изменяемые объекты можно изменить после их создания, неизменяемые — нельзя.

Строковые литералы могут быть заключены одинарными кавычками (`'blah'`), двойными кавычками (`"blah blah"`) или тремя двойными или одинарными кавычками (`"""none"""` или `'''none'''`). Также стоит отметить, строковые объекты обрабатываются в Python 3 и в Python 2 по-разному. В Python 3 они могут содержать только текстовые последовательности в виде точек данных Юникода, тогда как в Python 2 они могут содержать и текст, и байтовые данные. В Python 3 байтовые данные хранятся в типе `bytes`.

Разделение текста и байтов в Python 3 делает его чистым и эффективным, но в ущерб переносимости данных. Текст Юникода в строках нельзя сохранить на диск или отправить в удаленное расположение в сети без преобразования в двоичный

формат. Для этого необходимо закодировать строковые данные в последовательность байтов одним из следующих способов:

- **Метод `str.encode (encoding, errors)`**: используется со строковыми объектами и принимает два аргумента; пользователь может указать тип кодека (по умолчанию UTF-8) и способ обработки ошибок.
- **Преобразование в байтовый тип данных**: строковый объект можно преобразовать в тип `Bytes`, передав экземпляр строки конструктору вместе со схемой кодирования и схемой обработки ошибок.

Подробную информацию о методах и атрибутах, доступных для строкового объекта, можно найти в официальной документации к Python соответствующей версии.

Списки

Список (**List**) — один из базовых типов коллекций в Python, который используется для хранения нескольких объектов с помощью одной переменной. Списки являются динамичными и изменяемыми, то есть объекты в нем могут быть изменены, а сам список увеличен или уменьшен.

Объекты списка реализованы не с использованием связных списков, а с помощью массива переменной длины, который содержит ссылки на хранимые им объекты. Указатель на массив и его длина хранятся в заголовке списка, который обновляется по мере добавления и удаления объектов в нем. Поведение такого массива похоже на список, но на самом деле это не так, поэтому некоторые операции со списками в Python не оптимизированы. Например, вставка и удаление объектов имеет сложность: **n**.

Для исправления ситуации язык предоставляет тип данных `deque` (двусторонняя очередь) во встроенном модуле `collections`. Такой тип предоставляет функционал *стеков* и *очередей*, что является удобной альтернативой в случаях, когда для решения задачи требуется поведение связанного списка.

Списки можно создавать пустыми или с начальным значением, используя квадратные скобки. Рассмотрим пример, который демонстрирует создание пустого или непустого списка, используя только квадратные скобки или конструктор объекта списка:

```
e1 = []                 #пустой список
e2 = list()             #пустой список с использованием конструктора
g1 = ['a', 'b']         #список с 2 элементами
g2 = list(['a', 'b'])   #список с 2 элементами с использованием конструктора
g3 = list(g1)           #список, созданный из списка
```

Подробности об операциях над объектами списка (`add`, `insert`, `append` и `delete`) можно посмотреть в официальной документации Python. Далее рассмотрим кортежи.

Кортежи

Кортеж (**Tuple**) — это неизменяемый список, соответственно, его нельзя изменить после создания. Обычно используются для малого количества элементов, а также, когда важны их положение и последовательность в коллекции. Для сохранения последовательности элементов кортежи созданы неизменяемыми, и это их отличает от списков. Операции с ними, как правило, выполняются быстрее, чем со списками. Когда значения в коллекции должны быть постоянными и в определенном порядке, предпочтительнее использовать кортежи из-за их превосходной производительности.

Из-за неизменности кортежи обычно инициализируются значениями. Простой кортеж можно создать с помощью круглых скобок. Ниже показаны несколько способов создания экземпляров кортежей:

```
w = ()                  #пустой кортеж
x = (2, 3)              #кортеж с 2 элементами
y = ("Hello World")     #не кортеж, для разделения элементов нужна запятая
z = ("Hello World",)    #запятая делает это кортежем
```

В этом фрагменте создан пустой кортеж (`w`), кортеж с числами (`x`) и кортеж с текстом «Hello World» (`z`). Переменная `y` не является кортежем, поскольку для кортежа из 1 элемента требуется запятая в конце для указания на перечисление объектов.

Теперь познакомимся со словарями.

Словари

Словарь (**Dictionary**) — один из наиболее популярных и универсальных типов данных в Python. Он представляет собой набор для хранения данных в формате *«ключ:значение»*. Это изменяемый и неупорядоченный тип данных. В других языках они называются *ассоциативными массивами* и *хеш-таблицами*.

Словарь может быть задан списком, в котором данные идут парами *(ключ:значение)* и заключены в фигурные скобки. Ключ отделяется от значения двоеточием, а пары между собой разделяются запятой. Ниже представлен код с определением словаря:

```
mydict = {
    "brand": "BMW",
    "model": "330i",
    "color": "Blue"
}
```

Ключи в словаре не могут повторяться. Они должны быть представлены неизменным объектом, например, строкой, кортежем или числом. Значения в словаре могут иметь любой тип данных, включая списки, множества, пользовательские объекты и даже другие словари.

При работе со словарями важны три объекта или списка:

- **Ключи**: используются для перебора элементов словаря; список ключей можно получить с помощью метода `keys()`;

```
dict_object.keys()
```

- **Значения**: это объекты, хранящиеся в паре с ключами; список значений можно получить с помощью метода `values()`:

```
dict_object.values()
```

- **Элементы**: это пары *«ключ-значение»*, хранящиеся в словаре; список элементов можно получить с помощью метода `items()`:

```
dict_object.items()
```

Далее рассмотрим множества, которые являются ключевой структурой данных в Python.

Множества

Множество (**Set**) — набор уникальных объектов. Это изменяемая и неупорядоченная коллекция. Объекты множества не повторяются. Python использует структуру хеш-таблицы при реализации уникальности для множества, как и для ключей в словаре. В Python множества ведут себя почти как в математике. Этот тип данных применяется в ситуациях, когда важен не порядок объектов, а их уникальность. С его помощью можно проверить, входит ли определенный объект в коллекцию.

> ***Подсказка***
>
> Если возникает потребность в неизменяемом множестве, Python имеет подобный вариант реализации, называемый `frozenset`.

Создать новое множество можно с помощью фигурных скобок или конструктора множества `set()`. В следующем фрагменте приведены несколько примеров создания множества:

```
s1 = set()              # пустое множество
s2 = {}                 # пустое множество с фигурными скобками
s3 = set(['a', 'b'])    # множество, созданное из списка, используя конструктор
s3 = {1,2}              # множество, созданное с помощью фигурных скобок
s4 = {1, 2, 1}          # множество будет создано только с элементами 1 и 2,
                        # повторяющийся элемент будет проигнорирован
```

Обращение к заданным объектам невозможно с использованием индексации. Для доступа необходимо извлечь один объект из множества как список или перебрать множество для извлечения объектов по-одному. Как и математические множества, в Python они поддерживают такие операции, как *объединение*, *пересечение* и *разность*.

В этом разделе мы рассмотрели ключевые понятия строк и коллекций в Python 3, которые важны для понимания следующей темы — итераторы и генераторы.

Итераторы и генераторы для обработки данных

Итерация (**Iteration**) — один из ключевых инструментов обработки и преобразования данных, особенно при работе с большими наборами, а также когда разместить весь набор в памяти невозможно или нецелесообразно. Итераторы позволяют вводить данные в память по одному элементу за раз.

Итераторы можно создавать, определяя их в отдельном классе и реализуя специальные методы вроде `__iter__` и `__next__`. Существует также новый способ с помощью оператора `yield`, известного также под названием *генератор*. Далее мы подробнее поговорим об этом.

Итераторы

Итераторы — это объекты, используемые для перебора других объектов. Перебираемый объект называется *итерируемым* (**iterable**). Теоретически, итератор и перебираемый объект — это два разных объекта, но можно реализовать итератор внутри класса объектов типа `iterable`. Так делать не рекомендуется, хоть и технически возможно. Позже мы увидим на примере, почему это считается плохим подходом. В следующем фрагменте приводится несколько примеров использования цикла `for` для итерации в Python:

#iterator1.py

```
#пример 1: итерация по списку
for x in [1,2,3]:
    print(x)

#пример 2: итерация по строке
for x in "Python for Geeks":
    print(x, end="")
print('')

#пример 3: итерация по словарю
week_days = {1:'Mon', 2:'Tue',
             3:'Wed', 4:'Thu',
             5:'Fri', 6:'Sat', 7:'Sun'}
for k in week_days:
   print(k, week_days[k])

#пример 4: итерация по файлу
for row in open('abc.txt'):
    print(row, end="")
```

В примере были использованы разные циклы `for` для прохода по списку, строке, словарю и файлу. Все эти типы данных являются итерируемыми. Далее рассмотрим, какие характеристики делают объект итерируемым и что такое *протокол итерации*.

Важное примечание

Каждый набор в Python является итерируемым по умолчанию.

В Python итератор должен реализовывать два специальных метода: `__iter__` и `__next__`. Доступность объекта для перебора обусловлена, как минимум, наличием метода `__iter__`. Иными словами, когда объект реализует метод `__iter__`, его можно назвать итерируемым. Рассмотрим эти методы:

- `__iter__`: возвращает объект итератора; вызывается в начале цикла для получения объекта итератора;
- `__next__`: вызывается при каждой итерации цикла и возвращает следующий элемент в итерируемом объекте.

Для создания пользовательского объекта, который можно перебирать, реализуем пример с классом `Week`, который хранит номера и названия всех дней недели в словаре. По умолчанию он не будет итерируемым. Сделать его доступным для перебора можно, добавив метод `__iter__`. Для упрощения примера добавим в тот же класс и метод `__next__`. Ниже приведен фрагмент кода с классом `Week` и основной программой, которая выполняет перебор элементов с целью получить названия дней недели:

```
#iterator2.py
class Week:
    def __init__(self):
        self.days = {1:'Monday', 2: "Tuesday",
                     3:"Wednesday", 4: "Thursday",
                     5:"Friday", 6:"Saturday", 7:"Sunday"}
        self._index = 1

    def __iter__(self):
        self._index = 1
        return self

    def __next__(self):

        if self._index < 1 | self._index > 7 :
            raise StopIteration
        else:
            ret_value =  self.days[self._index]
            self._index +=1
        return ret_value

if(__name__ == "__main__"):
    wk = Week()
    for day in wk:
        print(day)
```

Этот пример призван продемонстрировать, как методы __iter__ и __next__ можно реализовать в одном классе объекта. Этот способ часто встречается в Интернете, но он считается плохим подходом к реализации итераторов. Когда мы используем его в цикле for, мы возвращаем главный объект в качестве итератора, поскольку реализовали __iter__ и __next__ в одном классе. Результаты могут быть непредсказуемыми. Для демонстрации выполним следующий код для того же класса Week:

#iterator3.py

```
class Week:
#определение класса как в предыдущем примере

if(__name__ == "__main__"):
    wk = Week()
    iter1 = iter(wk)
    iter2 = iter(wk)

    print(iter1.__next__())
    print(iter2.__next__())
    print(next(iter1))
    print(next(iter2))
```

Здесь перебирается один объект двумя разными итераторами. Результат новой программы не соответствуют ожидаемому выводу. Это происходит из-за общего атрибута _index, который используется двумя итераторами. Консольный вывод будет следующим:

```
Monday
Tuesday
Wednesday
Thursday
```

Обратите внимание, мы намеренно не использовали цикл for. Мы создали два итератора для одного объекта класса Week с помощью функции iter, которая является стандартной в Python и вызывает метод __iter__. Для получения следующего элемента в наборе мы напрямую использовали метод __next__, а также функцию next, которая является стандартной, как и iter. Использование итерируемого объекта в качестве итератора не считается *потокобезопасным*.

Лучшим подходом всегда будет использование отдельного класса итератора и создание нового его экземпляра с помощью метода __iter__. Каждый экземпляр должен управлять собственным внутренним состоянием. Далее приведена исправленная версия того же примера с классом Week и отдельным классом итератора:

#iterator4.py

```
class Week:
    def __init__(self):
```

```
        self.days = {1: 'Monday', 2: "Tuesday",
                     3: "Wednesday", 4: "Thursday",
                     5: "Friday", 6: "Saturday", 7: "Sunday"}

    def __iter__(self):
        return WeekIterator(self.days)

class WeekIterator:
    def __init__(self, dayss):
        self.days_ref = dayss
        self._index = 1

    def __next__(self):
        if self._index < 1 | self._index > 8:
            raise StopIteration
        else:
            ret_value =  self.days_ref[self._index]
            self._index +=1
        return ret_value

if(__name__ == "__main__"):
    wk = Week()
    iter1 = iter(wk)
    iter2 = iter(wk)
    print(iter1.__next__())
    print(iter2.__next__())
    print(next(iter1))
    print(next(iter2))
```

В примере есть отдельный класс с методом `__next__`, который имеет собственный атрибут `_index` для управления состоянием итератора. У экземпляра итератора будет ссылка на объект-контейнер (словарь). Результат консольного вывода будет ожидаемым: каждый итератор выполняется отдельно для одного и того же экземпляра класса `Week`. Вывод консоли будет такой:

```
Monday
Monday
Tuesday
Tuesday
```

Простыми словами, для создания итератора нужно реализовать методы `__iter__` и `__next__`, управлять внутренним состоянием и вызывать исключение `StopIteration`, когда нет доступных значений. Далее мы изучим генераторы, которые упрощают способ возврата итераторов.

Генераторы

Генератор (**Generator**) — простой способ вернуть экземпляр итератора, который можно использовать для перебора. Достигается это путем реализации только функции генератора. Она аналогична обычной функции, но с оператором `yield` вместо `return`. Оператор `return` допускается в функции генератора, но не будет использоваться для возврата следующего элемента в итерируемом объекте.

Функция является генератором, если она имеет хотя бы один оператор `yield`. Основное отличие его в том, что он приостанавливает выполнение функции и сохраняет ее внутреннее состояние. При следующем вызове выполнение начинается со строки, на которой закончил интерпретатор. Такой подход делает функциональность итераторов простой и эффективной.

Методы `__iter__` и `__next__` реализуются автоматически, как и исключение `StopIteration`, которое тоже вызывается автоматически. Локальные атрибуты и их значения сохраняются между последовательными вызовами без написания дополнительной логики. Интерпретатор Python предоставляет все эти возможности всякий раз, когда определяет функцию-генератор (функцию с оператором `yield` внутри).

Для понимания работы генератора начнем с простого примера, который используется при создании последовательности из первых трех букв алфавита:

```
#generators1.py
def my_gen():
    yield 'A'
    yield 'B'
    yield 'C'

if(__name__ == "__main__"):
    iter1 = my_gen()
    print(iter1.__next__())
    print(next(iter1))
    print(iter1.__next__())
```

Здесь реализована простая функция-генератор с тремя операторами `yield` и без оператора `return`. В основной части программы мы сделали следующее:

1. Вызвали функцию-генератор, которая возвращает экземпляр итератора. На этом этапе ни одна строка внутри `my_gen()` не выполняется.
2. Используя экземпляр итератора, вызвали метод `__next__`, который начинает выполнение функции `my_gen()`, приостанавливается после выполнения первой инструкции `yield` и возвращает букву `A`.
3. Затем вызываем функцию `next()` в экземпляре итератора. Результат будет таким же, как при использовании метода `__next__`. Но в этом случае функция `my_gen()` начинает выполнение со строки, идущей следом за той, на которой она остановилась в последний раз из-за оператора `yield`. На следующей строке находится другой оператор `yield`, поэтому после вывода буквы `B` происходит еще одна пауза.

4. Следующий метод `__next__` приведет к выполнению очередного оператора `yield`, который вернет букву `C`.

Далее мы вернемся к нашему классу `Week` и используем генератор вместо класса итератора. Пример кода:

```
#generator2.py
class Week:
    def __init__(self):
        self.days = {1:'Monday', 2: "Tuesday",
                     3:"Wednesday", 4: "Thursday",
                     5:"Friday", 6:"Saturday", 7:"Sunday"}

    def week_gen(self):
        for x in self.days:
            yield self.days[x]

if(__name__ == "__main__"):
    wk = Week()
    iter1 = wk.week_gen()
    iter2 = iter(wk.week_gen())
    print(iter1.__next__())
    print(iter2.__next__())
    print(next(iter1))
    print(next(iter2))
```

В сравнении с `iterator4.py` реализация класса `Week` с генератором выглядит гораздо аккуратнее и дает те же результаты. В этом заключается сила и популярность генераторов в Python. Прежде чем закончить тему, важно выделить несколько дополнительных ключевых особенностей:

- **Генератор-выражения**: используются для создания простых генераторов (известных также как *анонимные функции*) «на лету» без написания специальных методов; синтаксис похож на *списковое включение* (**List Comprehension**), но вместо квадратных скобок используются круглые; следующий фрагмент кода показывает, как можно использовать генератор-выражение для создания и использования генератора; сравним его со списком включения:

  ```
  #generator3.py
  L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
  f1 = [x+1 for x in L]
  g1 = (x+1 for x in L)

  print(g1.__next__())
  print(g1.__next__())
  ```

- **Бесконечные потоки**: с помощью генераторов можно реализовать бесконечные потоки данных; информацию проблематично записывать в память, но ге-

нераторы легко решают задачу, поскольку возвращают только один элемент данных за раз.

- **Конвейерная обработка генераторами**: при решении сложных задач несколько генераторов могут быть использованы в качестве *конвейера*; концепцию конвейерной обработки множеством генераторов можно объяснить на примере, где нужно вычислить сумму квадратов простых чисел; можно решить это с помощью традиционных циклов `for`, но мы попытаемся использовать два генератора: `prime_gen` для генерации простых чисел и `x2_gen` для возведения в квадрат простых чисел, полученных от `prime_gen`; выходные данные из двух генераторов передаются в функцию `sum` для получения их суммы; далее приведен фрагмент кода для решения этой задачи:

```
#generator4.py
def prime_gen(num):
    for cand in range(2, num+1):
        for i in range (2, cand):
            if (cand % i) == 0:
                break
            else:
                yield cand

def x2_gen(list2):
    for num in list2:
        yield num*num

print(sum(x2_gen(prime_gen(5))))
```

Генераторы работают по запросу, поэтому не только эффективно потребляют память, но и позволяют создавать значения по необходимости. Это помогает избежать ненужной генерации данных, которые могут вообще не использоваться. Генераторы хорошо подходят для обработки больших объемов данных, для передачи данных из одной функции в другие, а также для имитации *конкурентности*.

В следующем подразделе поговорим об обработке файлов в Python.

Обработка файлов в Python

Чтение и запись файлов — одни из основных инструментов в любом языке программирования. Python также поддерживает эти операции. Мы рассмотрим основные действия с файлами, доступные в стандартной библиотеке языка, а именно, как открывать и закрывать файлы, читать и записывать данные, управлять файлами с помощью *менеджеров контекста* и открывать несколько файлов с помощью одного дескриптора. Рассмотрим подробнее возможные действия с файлами.

Операции с файлами

Действия над файлами обычно начинаются с их открытия и последующего чтения или обновления содержимого.

Открытие и закрытие файла

Для применения любой операции к файлу сначала нужно получить указатель или ссылку на него. Это можно сделать, открыв файл встроенной функцией `open`, которая возвращает указатель на объект `file` (в некоторой литературе называется *дескриптором файла*). В функцию `open` требуется передать один обязательный параметр — имя файла с абсолютным или относительным путем. Опционально можно также указать режим доступа: чтение, запись, добавление и пр. Полный список режимов доступа:

- `r`: файл открывается только для чтения; применяется по умолчанию, если не указано иное:

```
f = open ('abc.txt')
```

- `a`: файл открывается с добавлением новой строки в конец файла:

```
f = open ('abc.txt', 'a')
```

- `w`: файл открывается для записи; если файл не существует, он будет создан; если существует, он будет переопределен, а его текущее содержимое уничтожено:

```
f = open ('abc.txt', 'w')
```

- `x`: файл открывается для *монопольной записи*; если он уже существует, возникнет ошибка:

```
f = open ('abc.txt', 'x')
```

- `t`: файл открывается в текстовом режиме; применяется по умолчанию;
- `b`: файл открывается в двоичном режиме;
- `+`: файл открывается для чтения и записи:

```
f = open ('abc.txt', 'r+')
```

Режимы можно комбинировать для получения нужного результата. Помимо имени и режима доступа можно также передать тип кодировки, что особенно полезно для текстовых файлов. Ниже приведен пример открытия файла в кодировке `UTF-8`:

```
f = open("abc.txt", mode = 'r', encoding = 'utf-8')
```

После окончания работы необходимо закрыть файл и освободить ресурсы для других процессов. Сделать это можно методом `close` для экземпляра файла или дескриптора. Фрагмент кода с методом `close`:

```
file = open("abc.txt", 'r+w')
#операции с файлом
file.close()
```

После закрытия файла ресурсы, связанные с его экземпляром и блокировками (если они есть), будут освобождены операционной системой. Это является лучшей практикой в любом языке программирования.

Чтение и запись

Файл можно прочесть, открыв его в режиме доступа `r`, а затем используя один из методов чтения:

- `read(n)`: считывает из файла `n` символов;
- `readline()`: возвращает из файла одну строку;
- `readlines()`: возвращает из файла все строки в виде списка.

Точно так же можно добавить или записать данные в файл, когда он открыт в соответствующем режиме:

- `write (x)`: записывает в файл строку или последовательность байтов и возвращает число символов, добавленных в файл;
- `writelines (lines)`: записывает в файл список строк.

В следующем примере мы создадим новый файл, добавим в него несколько текстовых строк, а затем считаем данные с помощью операций чтения:

```
#writereadfile.py: запись в файл и чтение из него
f1 = open("myfile.txt",'w')
f1.write("This is a sample file\n")
lines =["This is a test data\n", "in two lines\n"]
f1.writelines(lines)
f1.close()

f2 = open("myfile.txt",'r')
print(f2.read(4))
print(f2.readline())
print(f2.readline())

f2.seek(0)
for line in f2.readlines():
    print(line)
f2.close()
```

В примере мы сначала записали в файл три строки. Используя операции чтения, мы считали сначала четыре символа, а затем две строки методом `readline`. В конце мы переместили указатель обратно в начало файла методом `seek` и считали все строки методом `readlines`.

Далее рассмотрим, как менеджер контекста упрощает работу с файлами.

Менеджер контекста

Разумное и справедливое потребление ресурсов имеет решающее значение для любого языка программирования. Файловый обработчик или соединение с базой данных — это несколько примеров, когда общепринятой практикой является своевременное освобождение ресурсов. Если они не будут освобождены, это приведет к *утечке памяти*, которая может повлиять на производительность системы или привести к сбою.

Для решения проблемы и своевременного освобождения ресурсов в Python используются *контекстные менеджеры*. Они резервируют и освобождают ресурсы точно в соответствии с необходимостью. Когда менеджер используется с ключевым словом `with`, ожидается, что оператор после `with` должен вернуть объект, который реализует *протокол управления контекстом*. Этот протокол требует от возвращаемого объекта реализации двух методов:

- `.__enter__()`: вызывается с ключевым словом `with` и используется для резервирования ресурсов, необходимых для оператора после ключевого слова `with`;
- `.__exit__()`: вызывается после выполнения блока `with` и используется для освобождения ресурсов, зарезервированных методом `.__enter__()`.

Когда файл открывается, используя блок `with`, нет необходимости закрывать его обратно. Оператор `open` вернет объект дескриптора файла, в котором уже реализован протокол управления контекстом, а файл будет закрыт автоматически, после того как выполнение блока `with` завершится. Измененная версия примера с записью и чтением файла выглядит следующим образом:

#contextmgr1.py

```
with open("myfile.txt",'w') as f1:
    f1.write("This is a sample file\n")
    lines = ["This is a test data\n", "in two lines\n"]
    f1.writelines(lines)

with open("myfile.txt",'r') as f2:
    for line in f2.readlines():
        print(line)
```

Код с менеджером контекста прост и легко читаем. Его использование является рекомендуемым подходом для работы с файлами.

Работа с множеством файлов

Python поддерживает работу с несколькими файлами одновременно. Можно открывать их в разных режимах и работать с ними. Количество файлов не ограничено. Рассмотрим пример, где откроем два файла в режиме чтения и будем обращаться к ним в любом порядке:

1.txt

```
This is a sample file 1
```

```
This is a test data 1
2.txt
This is a sample file 2
This is a test data 2
#multifilesread1.py
with open("1.txt") as file1, open("2.txt") as file2:
    print(file2.readline())
    print(file1.readline())
```

Мы также можем читать из одного файла и записывать в другой. Пример кода выглядит следующим образом:

```
#multifilesread2.py
with open("1.txt",'r') as file1, open("3.txt",'w') as file2:
   for line in file1.readlines():
     file2.write(line)
```

Python также имеет более элегантное решение для работы с множеством файлов — модуль `fileinput`. Он может принимать список из нескольких файлов и обрабатывать их все как единый ввод. Пример кода с двумя входными файлами (`1.txt` и `2.txt`) и модулем `fileinput` показан ниже:

```
#multifilesread1.py
import fileinput
with fileinput.input(files = ("1.txt",'2.txt')) as f:
    for line in f:
        print(f.filename())
        print(line)
```

При таком подходе мы получаем один дескриптор, который работает с несколькими файлами последовательно. Далее обсудим обработку ошибок и исключений в Python.

Обработка ошибок и исключений

В Python существует множество типов ошибок. Наиболее распространенные из них связаны с синтаксисом в коде и называются *синтаксическими*. Также нередко они возникают прямо во время выполнения программы и называются, соответственно, *ошибками выполнения*. Те из них, которые можно обработать внутри программы, называются *исключениями*. В этом разделе мы покажем, как обрабатывать исключения и ошибки выполнения. Прежде чем перейти дальше, рассмотрим наиболее распространенные:

- `IndexError`: программа обращается к элементу по недопустимому индексу (адресу в памяти).

- `ModuleNotFoundError`: указанный модуль не найден по системному пути.
- `ZeroDivisionError`: программа пытается разделить число на ноль.
- `KeyError`: программа пытается получить значение из словаря по недопустимому ключу.
- `StopIteration`: возникает, когда метод `__next__` не находит следующий элемент итерации в контейнере.
- `TypeError`: программа пытается выполнить операцию с объектом неподходящего типа.

Полный список ошибок доступен в официальной документации Python. Далее рассмотрим непосредственно обработку ошибок и исключений соответствующими инструментами.

Работа с исключениями в Python

При возникновении ошибок во время выполнения программа может нанести ущерб системным ресурсам, например, повредив файлы и таблицы базы данных. Вот почему обработка исключений является одним из ключевых компонентов написания надежных программ. Идея состоит в том, что нам необходимо предвидеть потенциальные ошибки и продумать реакцию программы на них.

Как и многие другие языки, Python использует ключевые слова `try` и `except`. За ними следуют два отдельных блока кода, которые нужно исполнить. Блок `try` содержит обычный набор операторов, для которых мы предполагаем, что может возникнуть ошибка. Блок `except` будет выполнен, в случае если блок `try` содержит ошибку. Синтаксис кода с блоками `try` и `except`:

```
try:
    #набор операторов
except:
    #операторы, которые будут выполнены, если в блоке try есть ошибка
```

Если мы ожидаем один или несколько определенных типов ошибок, можно определить блок `except`, используя имя ошибки, и добавить столько блоков, сколько нужно. Такие именованные блоки `except` выполняются, когда в блоке `try` возникает исключение с соответствующим именем. К блоку `except` можно добавить оператор `as` для хранения объекта исключения в качестве переменной. Ниже приведен пример кода с множеством возможных ошибок в блоке `try` и множеством блоков `except`:

```
#exception1.py
try:
    print (x)
    x = 5
    y = 0
    z = x /y
    print('x'+ y)
```

```
except NameError as e:
    print(e)
except ZeroDivisionError:
    print("Division by 0 is not allowed")
except Exception as e:
    print("An error occurred")
    print(e)
```

Рассмотрим блоки except из примера подробнее:

- **Блок NameError**: будет выполняться, когда оператор в блоке try попытается обратиться к неопределенной переменной; в нашем случае — при попытке интерпретатора исполнить строку print(x); кроме того, мы присвоили объект исключения переменной с именем e и использовали ее в операторе print.
- **Блок ZeroDivisionError**: будет выполняться, когда мы попытаемся вычислить выражение z = x/y при y = 0; для выполнения этого блока сначала нужно исправить ошибку в блоке NameError.
- **Блок except по умолчанию**: это универсальный блок except, который будет выполняться, если не найдено совпадений с предыдущими двумя блоками except; строка print('x'+ y) также вызовет ошибку типа TypeError и будет обработана здесь; поскольку в этом блоке мы не получаем никакого конкретного типа исключения, можно использовать ключевое слово Exception для сохранения объекта исключения в переменной.

Обратите внимание, как только при любом операторе в блоке try возникает ошибка, остальные операторы игнорируются, а управление переходит к одному из блоков except. В нашем примере нужно сначала исправить ошибку NameError, тогда можно будет увидеть следующее исключение, и так далее. Мы добавили в пример три разных типа ошибки для демонстрации определения нескольких блоков except для одного блока try. Порядок блоков except важен. Сначала стоит определить конкретные именованные блоки, а блок по умолчанию лучше ставить в конце.

На схеме (рис. 4.1) показаны все возможные блоки обработки исключения.

Как видно из схемы, помимо try и except Python также поддерживает блоки else и finally для более эффективной обработки ошибок. Блок else выполняется, если в блоке try нет ошибок. Код здесь будет выполняться в обычном режиме, и в случае возникновения ошибки исключение не будет выдано. При необходимости в блок else можно добавить вложенные блоки try и except. Обратите внимание, блок else является необязательным.

Блок finally выполняется независимо от наличия или отсутствия ошибок в блоке try. Он обычно используется для освобождения ресурсов через закрытие соединений и открытых файлов. Хотя этот блок необязательный, настоятельно рекомендуется его реализовать.



Рис. 4.1. Блоки обработки исключений в Python

Далее рассмотрим использование этих блоков на примере. В блоке `try` откроем новый файл для записи. Если при открытии файла возникнет ошибка, будет выдано исключение, а сведения об ошибке будут выведены на консоль с помощью оператора `print` в блоке `except`. Если ошибок не возникнет, выполнится код в блоке `else`, который записывает в файл некоторый текст. В обоих случаях (возникнет ошибка или нет) файл будет закрыт в блоке `finally`. Пример кода:

```
#exception2.py
try:
     f = open("abc.txt", "w")
except Exception as e:
    print("Error:" + e)
else:
    f.write("Hello World")
    f.write("End")
finally:
    f.close()
```

Мы подробно рассмотрели обработку исключений в Python. Далее обсудим, как принудительно вызывать исключения в коде.

Вызов исключений

Исключения вызываются интерпретатором Python, когда во время выполнения возникает ошибка. Мы можем вызывать ошибки или исключения сами, когда понимаем, что продолжение выполнения приведет к неверному выводу или сбою работы кода. Вызов ошибки или исключения обеспечит корректный выход из программы.

Объект исключения можно передать вызывающей стороне, используя ключевое слово `raise`. Существуют разные типы исключений:

- Встроенное исключение.
- Пользовательское исключение.
- Универсальный объект `Exception`.

В следующем примере вызовем простую функцию для вычисления квадратного корня и реализуем ее для создания исключения, когда входной параметр не является допустимым положительным числом:

#exception3.py

```
import math
def sqrt(num):

    if not isinstance(num, (int, float)) :
        raise TypeError("only numbers are allowed")
    if num < 0:
        raise Exception ("Negative number not supported")
    return math.sqrt(num)

if __name__ == "__main__":
    try:
        print(sqrt(9))
        print(sqrt('a'))
        print (sqrt(-9))
    except Exception as e:
        print(e)
```

В коде мы вызываем встроенное исключение путем создания нового экземпляра класса `TypeError`, поскольку переданный в функцию `sqrt()` параметр не является числом. Мы также вызываем общее исключение, когда передаем в функцию число меньше 0. В обоих случаях мы передаем в конструктор пользовательский текст. Далее мы рассмотрим, как определить собственное пользовательское исключение и передавать его вызывающему коду.

Определение пользовательских исключений

В Python можно определить собственное исключение с помощью создания нового класса, производного от встроенного класса `Exception` или его подкласса. Для демонстрации изменим предыдущий пример и определим два пользовательских класса исключений, заменив ими встроенные типы ошибок `TypeError` и `Exception`. Новые пользовательские классы исключений будут производными от `TypeError` и `Exception`. Пример кода с пользовательскими исключениями:

#exception4.py

```
import math
```

```
class NumTypeError(TypeError):
    pass

class NegativeNumError(Exception):
    def __init__(self):
        super().__init__("Negative number not supported")

def sqrt(num):

    if not isinstance(num, (int, float)) :
        raise NumTypeError("only numbers are allowed")
    if num < 0:
        raise NegativeNumError

    return math.sqrt(num)

if __name__ == "__main__":
    try:
        print(sqrt(9))
        print(sqrt('a'))
        print (sqrt(-9))
    except NumTypeError as e:
        print(e)
    except NegativeNumError as e:
        print(e)
```

В этом примере класс `NumTypeError` является производным от класса `TypeError`, и мы ничего в него не добавляем. Класс `NegativeNumError` наследуется от класса `Exception`, мы переопределяем его конструктор, добавляя пользовательское сообщение для этого исключения как часть конструктора. Когда мы вызываем эти пользовательские исключения в функции `sqrt()`, мы не передаем никакого текста с классом исключений `NegativeNumError`. Когда мы выполняем основную программу, получаем сообщение с оператором `print(e)`, поскольку он задан как часть определения класса.

В этом подразделе мы рассмотрели, как обрабатывать встроенные типы ошибок с помощью блоков `try` и `except`, как определять пользовательские исключения и вызывать их декларативно. Далее обсудим ведение логов в Python.

Модуль logging в Python

Логирование (**Logging**) является фундаментальным требованием для любых масштабных приложений и представляет собой ведение журнала событий внутри программы. Это помогает не только при отладке, но также дает представление о внутренних процессах и проблемах приложения.

Преимущества логирования:

- Отладка кода, если необходимо проверить, когда и почему в приложении произошел сбой.
- Диагностика необычного поведения приложения.
- Предоставление данных для проверки соблюдения законов и нормативно-правовых норм.
- Оценка поведения пользователей и выявление попыток несанкционированного доступа.

Прежде чем перейти к практическим примерам, рассмотрим основные компоненты системы логирования в Python.

Основные компоненты системы логирования

Для ведения журнала приложения необходимы следующие компоненты:

- *Логгер* (**Logger**, он же *регистратор событий*).
- *Уровни логирования* (**Logging levels**, они же приоритеты событий).
- *Форматтер* (**Logging formatter**, он же форматировщик).
- *Обработчик* (**Logging handler**, он же обработчик сообщений в системе логирования).

Общая схема системы логирования в Python (рис. 4.2):

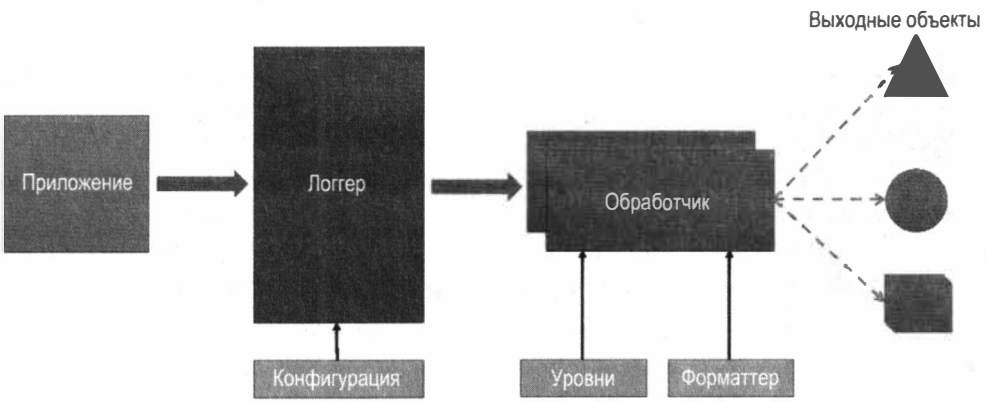


Рис. 4.2. Компоненты системы логирования в Python

Каждый из компонентов подробно обсудим ниже.

Логгер

Логгер — это интерфейс для системы ведения логов, с которым взаимодействует разработчик. Класс `Logger` в Python предоставляет несколько методов регистрации

сообщений с разными приоритетами (уровнями). Чуть позже рассмотрим методы класса `Logger` с примерами.

Приложение взаимодействует с экземпляром `Logger`, который устанавливается с использованием конфигурации и уровней логирования. При получении событий экземпляр `Logger` выбирает один или несколько подходящих обработчиков журнала и делегирует им эти события. Каждый обработчик, как правило, предназначен для конкретного *выходного объекта*. Он отправляет сообщения после применения фильтра и форматирования к предполагаемым выходным объектам.

Уровни логирования

События и сообщения в системе могут иметь разный приоритет. Например, ошибки являются более важными, чем предупреждения. Уровни логирования — это способ задать приоритеты для разных событий. В Python определено шесть уровней, каждый из которых выражен целым числом, указывающим на степень серьезности: `NOTSET`, `DEBUG`, `INFO`, `WARNING`, `ERROR` и `CRITICAL`. Их описание приводится на рис. 4.3:



Рис. 4.3. Уровни логирования в Python

Форматтер

Форматтер (или форматировщик) помогает улучшить форматирование сообщений, что важно для согласованности и удобочитаемости, как для человека, так и для компьютера. Он также добавляет в сообщения дополнительный контекст, например, время, имя модуля, номер строки, потоки и процессы, что очень помогает при отладке. Пример выражения форматировщика:

```
"%(asctime)s - %(name)s - %(levelname)s - %(funcName)
s:%(lineno)d - %(message)s"
```

Когда используется такое выражение, сообщение `hello Geeks` уровня `INFO` на консоли будет выглядеть примерно следующим образом:

```
2021-06-10 19:20:10,864 - a.b.c - INFO - <module name>:10 -
hello Geeks
```

Обработчик

Роль обработчика заключается в записи данных журнала в указанное место назначения, это могут быть консоль, файл или даже электронная почта. В Python доступно множество встроенных обработчиков, несколько популярных из них представлены ниже:

- `StreamHandler`: для вывода на консоль;
- `FileHandler`: для записи в файл;
- `SMTPHandler`: для отправки по электронной почте;
- `SocketHandler`: для отправки в сетевой сокет;
- `SyslogHandler`: для отправки на локальный или удаленный сервер системных логов Unix;
- `HTTPHandler`: для отправки на веб-сервер с использованием методов `GET` или `POST`.

Обработчик использует форматтер для добавления к логам дополнительной контекстной информации. Он также использует уровни логирования для их фильтрации.

Работа с модулем logging

На этом этапе мы рассмотрим, как использовать модуль `logging` на примерах. Начнем с базовых параметров и постепенно доведем их до продвинутого уровня.

Логгер по умолчанию

Если не создавать экземпляр класса `logger`, будет доступен логгер Python по умолчанию, также известный как *корневой логгер*. Его можно реализовать, импортируя модуль `logging` и используя его методы для отправки событий. В следующем примере показано, как использовать корневой логгер для регистрации событий журнала:

```
#logging1.py
import logging

logging.debug("This is a debug message")
logging.warning("This is a warning message")
logging.info("This is an info message")
```

Методы `debug`, `warning` и `info` используются для отправки событий логгеру в соответствии с уровнем серьезности. Для данного логгера приоритет события по умолчанию имеет значение `WARNING`, а вывод по умолчанию установлен `stderr`, а, значит, все

сообщения будут выводиться только на консоль или терминал. Этот параметр блокирует вывод сообщений уровня DEBUG и INFO на консоль. Вывод выглядит следующим образом:

```
WARNING:root:This is a warning message
```

Приоритет корневого логгера может быть изменен добавлением следующей строки после оператора import:

```
logging.basicConfig(level=logging.DEBUG)
```

После изменения уровня на DEBUG консольный вывод показывает все сообщения:

```
DEBUG:root:This is a debug message
WARNING:root:This is a warning message
INFO:root:This is an info message
```

Впрочем, использовать логгер по умолчанию не рекомендуется, кроме как для самых базовых задач. Для лучшей практики следует создавать новый именованный логгер, который мы рассмотрим далее.

Именованный логгер

Мы можем создать отдельный логгер со своими именем, приоритетом, обработчиками и форматтерами. В следующем примере создадим такой логгер и укажем для него пользовательское имя и приоритет:

```
#logging2.py
import logging
logger1 = logging.getLogger("my_logger")
logging.basicConfig()
logger1.setLevel(logging.INFO)
logger1.warning("This is a warning message")
logger1.info("This is a info message")
logger1.debug("This is a debug message")
logging.info("This is an info message")
```

Когда мы создаем экземпляр логгера, используя метод getLogger со строковым именем или с помощью имени модуля (используя глобальную переменную __name__), для одного имени может существовать только один экземпляр. Это значит, что, если мы попытаемся использовать метод getLogger с таким же именем в любой части приложения, интерпретатор Python проверит, создан ли уже экземпляр для этого имени. Если создан, интерпретатор вернет тот же экземпляр.

Создав экземпляр, мы должны сделать вызов к корневому логгеру (basicConfig()) для предоставления обработчика и форматтера нашему логгеру. Без настройки обработчика, в крайнем случае, будет использоваться внутренний обработчик, который просто выведет сообщения без какого-либо форматирования. А уровень сообщения будет WARNING, независимо от приоритета, который установлен в пользовательском логгере.

На консоли можно увидеть ожидаемые выходные данные:

```
WARNING:my_logger:This is a warning message
INFO:my_logger:This is a info message
```

Также важно отметить следующее:

- Мы задали для логгера приоритет INFO; мы смогли зарегистрировать сообщения warning и info, но не debug.
- Когда мы использовали корневой логгер (с помощью экземпляра logging), мы не смогли отправить сообщение info; это вызвано тем, что корневой логгер все еще использовал уровень по умолчанию, то есть WARNING.

Логгер со встроенным обработчиком и пользовательским форматтером

Можно создать объект логгера со встроенным обработчиком, но пользовательским форматировщиком. В этом случае объект обработчика может использовать объект пользовательского форматтера, также объект обработчика может быть добавлен к объекту логгера в качестве его обработчика, прежде чем мы начнем использовать логгер для каких-либо событий журнала. Следующий пример демонстрирует создание обработчика и форматтера, а также добавление первого в логгер:

```
#logging3.py
import logging
logger = logging.getLogger('my_logger')
my_handler = logging.StreamHandler()
my_formatter = logging.Formatter('%(asctime)s - '\
                '%(name)s - %(levelname)s - %(message)s')
my_handler.setFormatter(my_formatter)
logger.addHandler(my_handler)
logger.setLevel(logging.INFO)
logger.warning("This is a warning message")
logger.info("This is an info message")
logger.debug("This is a debug message")
```

Можно создать логгер с теми же настройками, используя метод basicConfig с соответствующими аргументами. Ниже приведена исправленная версия файла logging3.py с настройками basicConfig:

```
#logging3A.py
import logging
logger = logging.getLogger('my_logger')
logging.basicConfig(handlers=[logging.StreamHandler()],
                    format="%(asctime)s - %(name)s - "
                           "%(levelname)s - %(message)s",
                    level=logging.INFO)
```

```
logger.warning("This is a warning message")
logger.info("This is an info message")
logger.debug("This is a debug message")
```

До сих пор мы использовали встроенные классы и объекты для настройки логгеров. Далее рассмотрим логгер с пользовательскими обработчиками и форматтерами.

Логгер с файловым обработчиком

Обработчик отправляет лог-сообщения в место назначения. По умолчанию каждый регистратор (логгер) настроен на отправку сообщений на консоль или терминал. Это можно изменить, настроив его с новым обработчиком, который отправляет сообщения в другое место назначения. Файловый обработчик можно создать, используя один из двух подходов, рассмотренных ранее. В этом разделе использован третий подход для автоматического создания файлового обработчика, используя метод `basicConfig` с помощью указания имени файла в качестве атрибута. Пример кода показан ниже:

```
#logging4.py
import logging

logging.basicConfig(filename='logs/logging4.log'
                    ,level=logging.DEBUG)
logger = logging.getLogger('my_logger')
logger.setLevel(logging.INFO)
logger.warning("This is a warning message")
logger.info("This is a info message")
logger.debug("This is a debug message")
```

Теперь сообщения будут генерироваться в файл, указанный с помощью метода `basicConfig`, и в соответствии с приоритетом события, для которого установлено значение `INFO`.

Логгер с множеством обработчиков

Создать логгер с несколькими обработчиками довольно просто. Это может быть достигнуто либо с помощью метода `basicConfig`, либо путем присоединения обработчиков у логгеру вручную. Для демонстрации изменим пример `logging3.py` следующим образом:

1. Создадим два обработчика (один для вывода на консоль и один для вывода в файл), которые являются экземплярами классов `streamHandler` и `fileHandler`.
2. Создадим два отдельных форматтера по одному для каждого обработчика. Мы не будем включать информацию о времени для форматтера обработчика консоли.
3. Зададим для двух обработчиков разные приоритеты сообщений. Важно понимать, что приоритет на уровне обработчика не может переопределить обработчик корневого уровня.

Полный пример кода:

```
#logging5.py
import logging
logger = logging.getLogger('my_logger')
logger.setLevel(logging.DEBUG)
console_handler = logging.StreamHandler()
file_handler = logging.FileHandler("logs/logging5.log")
#установка приоритетов на уровне обработчика
console_handler.setLevel(logging.DEBUG)
file_handler.setLevel(logging.INFO)

#создание отдельных форматтеров для двух обработчиков
console_formatter = logging.Formatter(
                  '%(name)s - %(levelname)s - %(message)s')
file_formatter = logging.Formatter('%(asctime)s - '
                  '%(name)s - %(levelname)s - %(message)s')

#добавление форматтеров в обработчик
console_handler.setFormatter(console_formatter)
file_handler.setFormatter(file_formatter)
#добавление обработчиков в логгер
logger.addHandler(console_handler)
logger.addHandler(file_handler)

logger.error("This is an error message")
logger.warning("This is a warning message")
logger.info("This is an info message")
logger.debug("This is a debug message")
```

Несмотря на то что мы установили для обработчиков разные приоритеты (`INFO` и `DEBUG`), они будут эффективны только в случае, если уровни логгера имеют более низкий приоритет (по умолчанию `WARNING`). Поэтому нужно в начале программы для логгера установить уровень `DEBUG`. Тогда приоритету на уровне обработчика может быть присвоено значение `DEBUG` или более высокое значение. Это важно учитывать при разработке стратегии логирования для приложения.

В примерах этого раздела мы настраивали логгер программно (непосредственно в коде программы). Далее попробуем настроить его с помощью файла конфигурации.

Настройка логгера с множеством обработчиков с помощью файла конфигурации

Программная настройка логгера привлекательна, но непрактична для *среды развертывания* (также известна как *производственная среда* — **production environment**; не путайте со *средой разработки* — **development environment**). В средах

развертывания есть необходимость настраивать конфигурацию логгера иначе, чем в средах разработки, и иногда приходится повышать уровень логирования для устранения неполадок, с которыми мы сталкиваемся только в реальных средах. Поэтому есть возможность предоставить конфигурацию логгера через файл, который легко изменить в соответствии с требуемой средой. Конфигурация может быть либо в виде файлов JSON (**JavaScript Object Notation**) или YAML (**Yet Another Markup Language**), либо в виде списка пар *«ключ:значение»* в файле `.conf`. Для примера покажем конфигурацию логгера с помощью файла YAML, который не отличается от программной реализации в предыдущем примере. Полный файл YAML и код Python показаны далее.

Файл конфигурации YAML:

```
version: 1
formatters:
  console_formatter:
      format: '%(name)s - %(levelname)s - %(message)s'
  file_formatter:
      format: '%(asctime)s - %(name)s - %(levelname)s -
        %(message)s'
handlers:
  console_handler:
    class: logging.StreamHandler
    level: DEBUG
    formatter: console_formatter
    stream: ext://sys.stdout

  file_handler:
    class: logging.FileHandler
    level: INFO
    formatter: file_formatter
    filename: logs/logging6.log
loggers:
  my_logger:
    level: DEBUG
    handlers: [console_handler, file_handler]
    propagate: no
root:
  level: ERROR
  handlers: [console_handler]
```

Код программы Python, которая использует файл YAML для настройки логгера:

```
#logging6.py
import logging
import logging.config
import yaml
```

```
with open('logging6.conf.yaml', 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)

logger = logging.getLogger('my_logger')

logger.error("This is an error message")
logger.warning("This is a warning message")
logger.info("This is a info message")
logger.debug("This is a debug message")
```

Для загрузки конфигурации из файла мы использовали метод `dictConfig` вместо `basicConfig`. Результат получился аналогичным, как при настройке логгера программно. Для полнофункционального регистратора доступны и другие дополнительные параметры конфигурации.

Далее обсудим, какие события следует регистрировать, а какие — нет.

Что стоит и не стоит записывать в журнал

Однозначного ответа на этот вопрос нет, но в качестве рекомендации для логирования важна следующая информация:

- Приложение должно регистрировать все ошибки и исключения; наиболее подходящий для этого способ — ведение журнала событий в исходном модуле.
- Исключения, которые обрабатываются альтернативным потоком кода, могут регистрироваться как предупреждения.
- В целях отладки полезной информацией являются входные и выходные данные функций.
- Также полезно заносить в журнал *точки принятия решений*, они могут быть полезны для устранения неполадок.
- Активности и действия пользователей, особенно связанные с доступом к определенным ресурсам и функциям в приложении, важно регистрировать в целях безопасности и аудита.

При регистрации сообщений также важна контекстная информация — время, имя логгера, имя модуля, имя функции, номер строки, уровень логирования и т. д. Эта информация имеет важное значение для анализа причин неполадок.

Не следует регистрировать в журнал конфиденциальную информацию, например, идентификаторы пользователей, адреса электронной почты, пароли, а также любые личные и секретные данные. Кроме того, необходимо избегать регистрации служебной и персональной информации вроде медицинских записей, государственных документов и сведений об организациях.

Заключение

В этой главе мы рассмотрели различные темы, связанные с использованием расширенных модулей и библиотек Python. Мы узнали, как создавать и использовать итераторы для перебираемых объектов. Затем рассмотрели генераторы, которые можно использовать и создавать более эффективно, чем итераторы. Кроме того, узнали, как открывать файлы, выполнять чтение и запись данных, а также научились использовать контекстный менеджер. В дополнение мы поговорили об основе любого приличного приложения Python, а именно, об обработке ошибок и исключений, вызове исключений в коде и определении пользовательских исключений. В конце мы рассмотрели, как настроить структуру системы логирования, используя различные параметры обработчиков и форматтеров.

В следующей главе мы сосредоточимся на создании и автоматизации модульных и интеграционных тестов.

Вопросы

1. В чем разница между списком и кортежем?
2. Какой оператор Python всегда используется при работе с менеджером контекста?
3. Зачем нужен оператор `else` в блоке `try-except`?
4. Почему лучше использовать генераторы вместо итераторов?
5. В чем польза использования нескольких обработчиков для логирования?

Дополнительные ресурсы

- «*Python. К вершинам мастерства*», автор: Лучано Рамальо («Fluent Python», Luciano Ramalho).
- «*Advanced Guide to Python 3 Programming*», автор: Джон Хант (John Hunt).
- «*Стандартная библиотека Python 3. Справочник с примерами*», автор: Даг Хеллман («The Python 3 Standard Library by Example», Doug Hellmann).
- *Документация по Python 3.7.10* (*https://docs.python.org/3.7/*).
- *Официальная документация Python*, (*https://docs.python.org/3/library/logging.config.html*).

Ответы

1. Список является изменяемым объектом, тогда как кортеж — неизменным. Обновить список можно после его создания, что недоступно для кортежей.

2. С менеджером контекста используется оператор `with`.
3. Блок `else` выполняется только в случае, если код в блоке `try` завершается без ошибок. В блоке `else` можно написать дальнейшие действия, которые необходимо выполнить после завершения работы основного функционала в блоке `try` без каких-либо проблем.
4. Генераторы эффективны в использовании памяти и более просты в создании, чем итераторы. Генератор автоматически предоставляет экземпляр `iterator` и реализацию функции `next` «из коробки».
5. Один обработчик фокусируется на одном месте вывода сообщений, поэтому распространено использование нескольких обработчиков. Если нужно отправить события журнала на несколько выводов и с разными уровнями приоритета, нам потребуется несколько обработчиков. Также, если необходимо записывать сообщения в несколько файлов с разными уровнями логирования, можно создать разные обработчики для работы с разными файлами.

5

Тестирование и автоматизация с помощью Python

Тестирование программного обеспечения (ПО) — это проверка приложения или программы на соответствие требованиям пользователя или желаемым спецификациям, а также оценка масштабируемости и оптимизации кода. Тестирование реальными пользователями требует много времени и не является эффективным использованием ресурсов. Более того, тесты выполняются не один-два раза, а являются непрерывным процессом. *Автоматизация тестирования* — набор программ, которые проверяют поведение приложения с помощью различных сценариев в качестве входных данных для этих программ. Профессиональные среды разработки ПО требуют выполнения автоматизированных тестов после каждого *коммита* (фиксации изменений исходного кода) в центральном репозитории.

В этой главе мы изучим различные подходы к автоматизации тестирования, а также поговорим о фреймворках и библиотеках для этих целей. Затем остановимся на модульном тестировании и способах его реализации в Python. Далее обсудим преимущества *разработки через тестирования* (**Test-driven Development**, **TDD**) и правильный способ ее исполнения. В конце мы рассмотрим автоматизированную *непрерывную интеграцию* (**Continuous Integration**, **CI**) и сложности, связанные с ее надежной и эффективной реализацией. Эта глава поможет понять концепции автоматизированного тестирования в Python на разных уровнях.

Темы этой главы:

- Понимание различных уровней тестирования.
- Работа с тестовыми фреймворками Python.
- Разработка через тестирование.
- Автоматизированная непрерывная интеграция.

К концу главы вы будете понимать не только различные типы автоматизации тестирования, но также научитесь писать модульные тесты, используя один из двух популярных фреймворков.

Технические требования

Для этой главы понадобится:

- Python 3.7 или более поздней версии.
- Аккаунт Test PyPI и токен API для этого аккаунта.

Примеры кода для этой главы можно найти по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter05.

Понимание различных уровней тестирования

Тестирование выполняется на различных уровнях в зависимости от типа приложения, уровня его сложности и роли команды, работающей над приложением. К разным уровням тестирования относятся:

- модульное тестирование;
- интеграционное тестирование;
- системное тестирование;
- приемочное тестирование.

Эти уровни применяются в следующем порядке (рис. 5.1):



Рис. 5.1. Уровни тестирования при разработке ПО

Рассмотрим эти уровни подробнее.

Модульное тестирование

Это тип тестирования, ориентированный на наименьший возможный блок кода (модуль). Им может быть функция в модуле, метод в классе или модуль в приложении. Такой тест изолированно выполняет один блок кода и проверяет, что тот работает должным образом. Это помогает обнаруживать ошибки на ранней стадии разработки и исправлять их. В Python модульные тесты обычно нацелены на отдельные классы или модули без использования зависимостей.

Тесты пишутся непосредственно разработчиками и могут выполняться в любое время. Это своего рода *тестирование методом белого ящика* (**White-box Testing**). Для этого в Python есть несколько библиотек и инструментов: `pyunit` (`unittest`), `pytest`, `doctest`, `nose` и некоторые другие.

Интеграционное тестирование

Это совместное тестирование отдельных модулей программы в группе на взаимодействие между собой и корректный обмен данными.

Интеграционное тестирование обычно проводится тестировщиками, а не разработчиками и выполняется после модульного тестирования. Основное внимание уделяется выявлению проблем интеграции, когда различные модули и функции используются вместе. Иногда для тестирования требуются внешние ресурсы или данные, которые недоступны в среде разработки. Тогда на помощь приходит *фиктивное тестирование* (**Mock Testing**), которое обеспечивает замену внешним и внутренним зависимостям. *Фиктивные объекты* (или *mock*-объекты) имитируют поведение реальных зависимостей. Примерами служат отправка электронного письма или оплата кредитной картой.

Это своего рода *тестирование методом черного ящика* (**Black-box Testing**). Используемые библиотеки и инструменты почти такие же, как в модульном тестировании, с той лишь разницей, что границы тестов расширяются и включают несколько модулей в один тест.

Системное тестирование

Границы системного тестирования расширяются до уровня системы. Это может быть полноценный модуль или приложение. Здесь проверяется функциональность с точки зрения *сквозного тестирования* (**End-to-End**, **E2E**), то есть от начала и до конца.

Системные тесты также разрабатываются тестировщиками, но уже после завершения процесса интеграционного тестирования. Можно сказать, что интеграционное тестирование является необходимым условием системного тестирования, иначе придется делать много лишних действий. На данном этапе можно выявить потенциальные проблемы, но не наверняка определить их местоположение. Точная при-

чина обычно определяется интеграционным тестированием или даже добавлением дополнительных модульных тестов.

Системное тестирование также относится к типу тестирования методом черного ящика и может использовать те же библиотеки, что и интеграционное.

Приемочное тестирование

Приемочное тестирование выполняется реальным пользователем перед приемкой программы для повседневного использования. Оно также известно, как *UAT-тестирование* (**User Acceptance Testing**). Обычно для таких тестов не применяют автоматизацию, но ее стоит использовать в ситуациях, когда пользователям приходится взаимодействовать с продуктом через API. Этот тип тестирования легко спутать с системным, но он отличается тем, что обеспечивает удобство использования приложения с точки зрения реальных пользователей. Существуют также два дополнительных типа приемочного тестирования: *заводские приемочные испытания* (**Factory Acceptance Testing, FAT**) и *эксплуатационное приемочное тестирование* (**Operational Acceptance Testing, OAT**). Первый более популярен с точки зрения аппаратного обеспечения, а второй выполняется рабочими командами, отвечающими за использование продукта в производственных средах.

Вы также могли слышать про *альфа-* и *бета-тестирование*, которые выполняются на уровне пользователей и не автоматизируются. Альфа-тестирование выполняется разработчиками и другим персоналом для имитации реального поведения пользователей. Бета-тестирование выполняется клиентами или настоящими пользователями для получения обратной связи о продукте, прежде чем он будет выпущен в общий доступ.

Кроме того, существует *регрессионное тестирование*. Оно выполняется всякий раз, когда вносятся изменения в исходный код или любые внешние и внутренние зависимости. Эта практика гарантирует, что продукт работает так же, как и до внесения изменений. Поскольку регрессионное тестирование повторяется многократно, его автоматизация необходима.

Далее рассмотрим, как создавать *тест-кейсы* (**test case** — тестовый случай, тестовый сценарий, тест) с использованием тестовых фреймворков.

Работа с тестовыми фреймворками Python

Для Python есть стандартные и сторонние библиотеки для автоматизации тестирования. Самые популярные из них:

- `pytest`;
- `unittest`;
- `doctest`;
- `nose`.

Эти фреймворки подходят как для модульного, так и для интеграционного и системного тестирования. Мы рассмотрим два из них: `unittest`, который является частью стандартной библиотеку Python, и `pytest`, который доступен как сторонняя библиотека. Основное внимание в главе будет уделено тестовым примерам (по большей части модульным) с использованием этих двух библиотек. Но помните, что интеграционное и системное тестирование также могут быть построены с использованием этих библиотек и шаблонов проектирования.

Прежде чем начать писать тесты, сначала важно понять, что это такое. В контексте этой главы и книги в целом мы будем определять тест-кейс, как способ проверки результатов определенного поведения кода в соответствии с ожидаемыми результатами. Разработку тестов можно разделить на четыре этапа:

1. **Подготовка**: на этом этапе мы подготавливаем среду для тестов. Здесь не выполняется никаких действий или проверок. В сообществе этот этап известен как подготовка *тестовых фикстур* (**Test Fixture**) — приспособлений для проверки.
2. **Действие**: здесь мы запускаем тестируемую систему. Этап действия приводит к изменениям системы. А измененное состояние — это именно то, что мы хотим оценить в рамках тестирования. Обратите внимание, здесь мы по-прежнему ничего не проверяем.
3. **Проверка**: здесь мы оцениваем результаты предыдущего этапа и сравниваем их с ожидаемыми результатами. На основе этого тест-кейс помечается как успешный или неудачный. В большинстве инструментов проверка достигается с помощью встроенных функций или операторов `assert`.
4. **Очистка**: на этом этапе среда очищается. Нужно убедиться, что внесенные изменения на этапе *действия* не влияют на другие тесты.

Ключевые этапы — *действие* и *проверка*. *Подготовка* и *очистка* являются необязательными, но настоятельно рекомендуются. Они обеспечивают подготовку фикстур — оборудования, устройств или ПО, которые предоставляют среду для последовательного тестирования продукта. Термин *«тестовая фикстура»* используется в том же контексте для модульного и интеграционного тестирования.

Фреймворки или библиотеки предоставляют вспомогательные методы или операторы для удобной реализации этих этапов. Далее мы будем оценивать фреймворки `unittest` и `pytest` по следующим аспектам:

- Как создавать базовые тест-кейсы для этапов действия и проверки.
- Как создавать тест-кейсы с помощью фикстур.
- Как создавать тест-кейсы для проверки исключений и ошибок.
- Как запускать тест-кейсы в большом количестве.
- Как включать и исключать тест-кейсы во время выполнения.

Эти темы затрагивают не только создание разных тестовых сценариев, но и способы их выполнения. Начнем знакомство с `unittest`.

Работа с фреймворком unittest

Перед обсуждением практических примеров с `unittest` рассмотрим некоторые термины и методы, связанные с модульным тестированием и с самой библиотекой в частности. Терминология ниже используется почти во всех библиотеках:

- **Тест-кейс**: тест, тестовый сценарий или тестовый случай — набор инструкций кода, основанных на сравнении текущего состояния единицы кода с состоянием после ее выполнения.
- **Тестовый набор**: это набор тестовых сценариев, которые могут иметь общие предварительные условия, этапы инициализации и очистки; это способствует повторному использованию кода автоматизации и сокращает время выполнения.
- **Исполнитель тестов**: приложение Python, которое выполняет тесты (модульные), выполняет все проверки, определенные в коде, и возвращает результат как успешный или неудачный.
- `setUp`: специальный метод, который выполняется перед каждым тестом.
- `setupClass`: специальный метод, который выполняется однократно перед началом выполнения тестов из набора.
- `teardown`: специальный метод, который выполняется после завершения каждого теста независимо от его результата.
- `teardownClass`: специальный метод, который выполняется один раз после завершения всех тестов в наборе.

Для написания тестовых сценариев необходимо реализовать их как методы экземпляра класса, который должен наследоваться от базового класса `TestCase`. Класс `TestCase` имеет несколько методов, которые упрощают написание и выполнение тестов. Эти методы сгруппированы в три категории:

- **Методы, связанные с выполнением**: `setUp`, `tearDown`, `setupClass`, `teardownClass`, `run`, `skipTest`, `skipTestIf`, `subTest` и `debug`; они задействуются исполнителем для выполнения кода *до* или *после* тест-кейсов, запуска теста, пропуска теста или выполнения блока кода как подтеста; в классе реализации теста эти методы можно переопределять; подробнее можно узнать в официальной документации по адресу *https://docs.python.org/3/library/unittest.html*.
- **Методы проверки**: используются для реализации тестов, которые проверяют условия успешного или неудачного исхода и возвращают результат автоматически; имена таких методов обычно начинаются с префикса `assert`; список таких методов очень длинный, на рис. 5.2 приведены самые часто используемые.
- **Методы и атрибуты, связанные с дополнительной информацией**: предоставляют дополнительную информацию о тестовых сценариях, которые должны быть или уже выполнены; ниже приведены несколько ключевых методов и атрибутов:
  - Атрибут `failureException`: предоставляет исключение, вызванное тестовым методом, которое можно использовать как суперкласс для определения пользовательского исключения с дополнительной информацией.

- Атрибут `longMessage`: определяет, что делать с пользовательским сообщением, которое передается как аргумент с методом `assert`; если значение аргумента `True`, сообщение добавляется к стандартному сообщению об ошибке; если значение аргумента `false`, пользовательское сообщение заменяет стандартное.
- Метод `countTestCases()`: возвращает количество тестов, прикрепленных к тестовому объекту.
- Метод `shortDescription()`: возвращает описание тестового сценария, если таковое добавлено, используя docstring.

Имя метода	Проверка условий
assertEqual (x, y)	Делает проверку на равенство **x** и **y**
assertTrue (x)	Делает проверку на равенство **x** булевому значению **true**
assertFalse (x)	Делает проверку на равенство **x** булевому значению **false**
assertNotEqual (x)	Делает проверку на неравенство **x** и **y**
assert (a, c)	Делает проверку на наличие элемента **a** в коллекции **c**
assertNotIn (a, c)	Делает проверку на отсутствие элемента **a** в коллекции **c**
assertIs (x, y)	Делает проверку на соответствие объекта **x** объекту **y**
assertIsNot (x, y)	Делает проверку на несоответствие объекта **x** объекту **y**
assertIsNone (x)	Делает проверку на равенство объекта значению **None**

Рис. 5.2. Примеры методов `assert` класса `TestCase`

В этом разделе мы рассмотрели основные методы класса `TestCase`. Далее остановимся на создании модульных тестов с помощью `unittest`.

Создание тестов с помощью базового класса TestCase

Библиотека `unittest` — это стандартный фреймворк тестирования Python, разработчики которого были вдохновлены **JUnit** (популярный фреймворк в Java). Модульные тесты пишутся в отдельных файлах, и их следует подключать к основному проекту. В подразделе *«Сборка пакетов»* в главе 2 (*«Использование модулей для слож-*

ных проектов») уже упоминалось, что сообщество PyPA рекомендует иметь отдельную папку для тестов при создании пакетов в проекте. В примерах кода для этого раздела мы будем следовать следующей структуре :

```
Project-name
|-- src
|   -- __init__.py
|   -- myadd/myadd.py
|-- tests
|   -- __init__.py
|   -- tests_myadd/test_myadd1.py
|   -- tests_myadd/test_myadd2.py
|-- README.md
```

В первом примере создадим набор тестов для функции `add` в модуле `myadd.py`:

```
#myadd.py со сложением двух чисел
def add(x, y):
    """Эта функция складывает 2 числа"""
    return x + y
```

Важно понимать, для одного фрагмента кода (в нашем случае для функции `add`) может быть несколько тестовых сценариев. Для этой функции мы реализовали четыре теста, меняя значения входных параметров, как показано в примере:

```
#test_myadd1.py набор тестов для функции myadd
import unittest
from myunittest.src.myadd.myadd import add

class MyAddTestSuite(unittest.TestCase):

def test_add1(self):
    """тест для проверки сложения двух положительных чисел"""
    self.assertEqual(15, add(10 , 5), "should be 15")

def test_add2(self):
    """тест для проверки сложения положительного и отрицательного чисел"""
    self.assertEqual(5, add(10 , -5), "should be 5")

def test_add3(self):
    """тест для проверки сложения отрицательного и положительного чисел"""
    self.assertEqual(-5, add(-10 , 5), "should be -5")

def test_add4(self):
    """тест для проверки сложения двух отрицательных чисел"""
    self.assertEqual(-15, add(-10 , -5), "should be -15")

if __name__ == '__main__':
    unittest.main()
```

Ключевые моменты кода:

1. Для реализации модульных тестов через `unittest` нужно импортировать стандартную библиотеку с тем же именем.
2. Модули, которые нужно протестировать в наборе, также необходимо импортировать. В нашем случае мы импортировали функцию `add` из модуля `myadd.py` с помощью *относительного импорта* (подробности в подразделе *«Импорт модулей»* в главе 2 *«Использование модулей для сложных проектов»*).
3. Мы реализуем класс тестовых наборов, который наследуется от базового класса `unittest.Testcase`. Сценарии реализуются в подклассе `MyAddTestSuite`. Конструктор класса `unittest.Testcase` может принимать имя метода в качестве входных данных, которые можно использовать для выполнения тест-кейсов. По умолчанию уже реализован метод `runTest`, с помощью которого исполнитель запускает тесты. в большинстве случаев не нужно предоставлять собственный метод или повторно реализовывать метод `runTest`.
4. Для реализации тестового сценария нужно написать метод, который начинается с префикса `test` и символа подчеркивания. Это помогает исполнителю находить нужные сценарии для выполнения. Руководствуясь соглашением об именовании, мы добавили в набор четыре метода.
5. В каждом тестируемом методе мы использовали специальный метод `assertEqual`, доступный из базового класса, который представляет этап проверки и позволяет определить, результат теста будет успешным или неудачным. Первый параметр этого метода — ожидаемый результат теста. Второй параметр — значение, получаемое после выполнения тестируемого кода. Третий параметр (необязательный) — сообщение, которое будет предоставлено в отчете в случае неудачи теста.
6. В конце набора мы добавили метод `unittest.main` для запуска исполнителя тестов. Тот, в свою очередь, необходим для запуска метода `runTest`, который упрощает выполнение без использования консольных команд. Этот `main`-метод (с классом `TestProgram` внутри) сначала обнаружит все тесты, которые нужно выполнить, а затем выполнит их.

> ***Важное примечание***
>
> Модульные тесты можно выполнять с помощью команды `Python -m unittest <тестовый набор или модуль>`, но примеры кода в этой главе предполагают, что мы выполняем сценарии, используя интегрированную среду разработки PyCharm.

Далее рассмотрим следующий уровень сценариев с использованием тестовых фикстур.

Создание тестов с помощью тестовых фикстур

Мы рассмотрели методы `setUp` и `tearDown`, которые запускаются автоматически исполнителями до и после выполнения сценариев. Эти методы (наряду с `setUpClass` и `tearDownClass`) предоставляют тестовые фикстуры и полезны для эффективной реализации модульных тестов.

Для начала пересмотрим реализацию функции `add` и сделаем этот фрагмент кода частью класса `MyAdd`. Мы также реализуем исключение `TypeError`, если входные аргументы будут недопустимыми. Ниже приведен полный фрагмент кода с новым методом `add`:

```
#myadd2.py – класс с методом сложения двух чисел
class MyAdd:
    def add(self, x, y):
        """Эта функция складывает 2 числа"""
        if (not isinstance(x, (int, float))) | \
                (not isinstance(y, (int, float))) :
            raise TypeError("only numbers are allowed")
        return x + y
```

Ранее мы создали тест-кейсы, используя только этапы действия и проверки. Теперь изменим предыдущий пример, добавив методы `setUp` и `tearDown`. Ниже приведен набор тестов для класса `myAdd`:

```
#test_myadd2.py набор тестов для метода класса myadd2
import unittest
from myunittest.src.myadd.myadd2 import MyAdd

class MyAddTestSuite(unittest.TestCase):
    def setUp(self):
        self.myadd = MyAdd()

    def tearDown(self):
        del (self.myadd)

    def test_add1(self):
        """тест для проверки сложения двух положительных чисел"""
        self.assertEqual(15, self.myadd.add(10 , 5), "should be 15")

    def test_add2(self):
        """тест для проверки сложения положительного и отрицательного чисел"""
        self.assertEqual(5, self.myadd.add(10 , -5), "should be 5")

#test_add3 и test_add4 пропущены, они совпадают с test_add1 и test_add2
```

В этом наборе тестов мы добавили или изменили следующие элементы:

1. Мы добавили метод `setUp`, в котором создали новый экземпляр класса `MyAdd` и сохранили указатель на него в качестве атрибута экземпляра. Это означает, что мы будем создавать новый экземпляр класса `MyAdd` до выполнения тестового случая. Возможно, это не идеальное решение для этого сценария, и было бы лучше использовать метод `setUpClass` и создать один экземпляр класса `MyAdd` для всего набора тестов, но мы создали эту реализацию в целях демонстрации.

2. Мы также добавили метод tearDown. Для демонстрации, как его реализовать, мы просто вызвали деструктор (используя функцию del) экземпляра MyAdd, который мы создали в методе setUp. В отличие от метода setUp, метод tearDown выполняется после каждого тестового случая. Если бы мы использовали метод setUpClass, для него существует tearDownClass, эквивалент tearDown.

Далее мы рассмотрим примеры кода, в которых создадим тестовые случаи для обработки исключения TypeError.

Создание тестов с обработкой ошибок

В предыдущих примерах мы просто сравнивали результаты тестов с ожидаемыми выводами. Мы не рассматривали поведение программы в сценариях с обработкой исключений, когда входные аргументы для функции add могут быть недопустимого типа. Модульные тесты должны учитывать такие аспекты.

Поэтому создадим такие тест-кейсы для обработки ожидаемых ошибок или исключений. В примере будем использовать ту же функцию add, которая выдает исключение TypeError, если аргумент не является числом. Тестовые случаи будут созданы передачей нечисловых аргументов в функцию add. Ниже приведен фрагмент кода:

```
#test_myadd3.py набор тестов для метода класса myadd2 для проверки ошибок
import unittest
from myunittest.src.myadd.myadd2 import MyAdd

class MyAddTestSuite(unittest.TestCase):
    def setUp(self):
        self.myadd = MyAdd()

    def test_typeerror1(self):
        """тест для проверки, можем ли мы обработать нечисловые вводные \
         данные"""
        self.assertRaises(TypeError, self.myadd.add, \
         'a' , -5)

    def test_typeerror2(self):
        """тест для проверки, можем ли мы обработать нечисловые вводные \
         данные"""
        self.assertRaises(TypeError, self.myadd.add, \
         'a' , 'b')
```

Мы добавили два дополнительных тестовых сценария в модуль test_add3.py. Они используют метод assertRaises для проверки, выдается исключение определенного типа или нет. В качестве аргументов для сценариев мы использовали одну букву (a) и две буквы (a и b). В обоих случаях ожидается предполагаемое исключение TypeError. Обратите внимание на аргументы метода assertRaises. В качестве второго аргумента этот метод ожидает только имя метода или функции, параметры которых должны передаваться отдельно в качестве аргументов функции assertRaises.

До сих пор мы выполняли несколько тестов в одном наборе. Теперь обсудим, как можно запускать несколько тестовых наборов одновременно как программно, так и с помощью командной строки.

Выполнение нескольких наборов тестов

По мере написания сценариев для каждой единицы кода число модульных тестов быстро растет. Использование тестовых наборов помогает привнести в разработку тестов модульность. Это также упрощает поддержку и расширение продукта в связи с расширением функциональности приложения. Инструменты непрерывной интеграции, такие, как `Jenkins`, прямо «из коробки» предоставляют возможность запускать множество тестовых наборов в основном скрипте. Фреймворки `unittest`, `nose` или `pytest` также обладают подобным функционалом.

Для примера создадим простое приложение калькулятора (класс `MyCalc`) с методами `add`, `subtract`, `multiply` и `divide`. Позже добавим по одному тестовому набору для каждого метода в этом классе. Таким образом, получится четыре набора тестов для приложения. При создании наборов и сценариев в них важно соблюдать структуру каталогов. В нашем приложении она будет следующей (рис. 5.3):

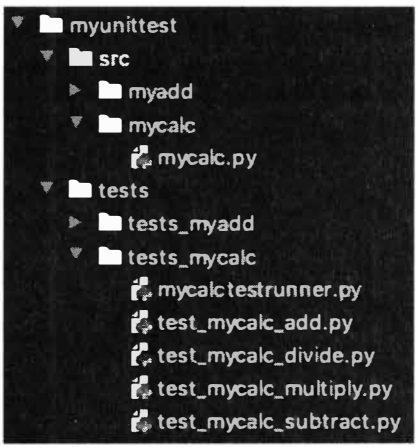


Рис. 5.3. Структура каталогов для приложения mycalc и наборов тестов

Код Python содержится в модуле `mycalc.py`, а файлы с наборами тестов (`test_mycalc*.py`) представлены ниже. Обратите внимание, что в следующих примерах наглядно приводится только один тестовый сценарий из каждого набора. В реальности же их будет несколько. Начнем с функций калькулятора в файле `mycalc.py`:

#mycalc.py с функциями add, subtract, multiply и divide

```
class MyCalc:
    def add(self, x, y):
        """Эта функция выполняет сложение двух чисел"""
        return x + y
```

```
    def subtract(self, x, y):
        """ Эта функция выполняет вычитание двух чисел """
        return x - y

    def multiply(self, x, y):
        """ Эта функция выполняет умножение двух чисел """
        return x * y

    def divide(self, x, y):
        """ Эта функция выполняет деление двух чисел """
        return x / y
```

Далее идет набор тестов для функции `add` в файле `test_mycalc_add.py`:

```
#test_mycalc_add.py набор тестов для метода add
import unittest
from myunittest.src.mycalc.mycalc import MyCalc

class MyCalcAddTestSuite(unittest.TestCase):
    def setUp(self):
        self.calc = MyCalc()

    def test_add(self):
        """тест для проверки двух положительных чисел"""
        self.assertEqual(15, self.calc.add(10, 5), "should be 15")
```

Потом идет набор тестов для функции `substract` в файле `test_mycalc_substract.py`:

```
#test_mycalc_subtract.py набор тестов для метода subtract
import unittest
from myunittest.src.mycalc.mycalc import MyCalc

class MyCalcSubtractTestSuite(unittest.TestCase):
    def setUp(self):
        self.calc = MyCalc()

    def test_subtract(self):
        """тест для проверки двух положительных чисел"""
        self.assertEqual(5, self.calc.subtract(10,5), "should be 5")
```

Затем идет набор тестов для функции `multiply` в файле `test_mycalc_multiply.py`:

```
#test_mycalc_multiply.py набор тестов для метода multiply
import unittest
from myunittest.src.mycalc.mycalc import MyCalc
```

```
class MyCalcMultiplyTestSuite(unittest.TestCase):
    def setUp(self):
        self.calc = MyCalc()

    def test_multiply(self):
        """тест для проверки двух положительных чисел"""
        self.assertEqual(50, self.calc.multiply(10, 5), "should be 50")
```

И, наконец, набор тестов для функции `divide` в файле `test_mycalc_divide. py`:

```
#test_mycalc_divide.py набор тестов для метода divide
import unittest
from myunittest.src.mycalc.mycalc import MyCalc

class MyCalcDivideTestSuite(unittest.TestCase):
    def setUp(self):
        self.calc = MyCalc()

    def test_divide(self):
        """тест для проверки двух положительных чисел"""
        self.assertEqual(2, self.calc.divide(10 , 5), "should be 2")
```

Теперь у нас есть код приложения из всех четырех наборов тестов. Следующий шаг: как запустить все тесты за один раз. Один из простых способов — с помощью *интерфейса командной строки* (**Command-Line Interface**, **CLI**) с ключевым словом `discover`. В нашем примере мы выполним команду ниже для нахождения и выполнения всех сценариев во всех наборах тестов в каталоге `tests_mycalc`:

```
python -m unittest discover myunittest/tests/tests_mycalc
```

Эта команда будет выполняться рекурсивно и сможет находить тесты в подкаталогах. Другие необязательные параметры, приведенные ниже, позволяют выбирать тестовые наборы для выполнения:

- `-v`: делает вывод подробным;
- `-s`: задает начальный каталог для поиска тестов;
- `-p`: шаблон для поиска тестовых файлов; по умолчанию используется `test*.py`, но данный параметр позволяет это значение изменить;
- `-t`: указывает на каталог проекта верхнего уровня; если ничего не указано, начальный каталог считается каталогом верхнего уровня.

Возможность запускать несколько наборов из командной строки очень проста и эффективна, но иногда нужно контролировать процесс запуска определенных тестов из разных наборов, которые к тому же могут быть в разных директориях. Это удобно делать именно с помощью Python. Ниже приведен код, как можно загру-

жать наборы тестов от имени класса, находить тест-кейсы в каждом из наборов и запускать их с помощью исполнителя unittest:

```
import unittest
from test_mycalc_add import MyCalcAddTestSuite
from test_mycalc_subtract import MyCalcSubtractTestSuite
from test_mycalc_multiply import MyCalcMultiplyTestSuite
from test_mycalc_divide import MyCalcDivideTestSuite

def run_mytests():
    test_classes = [MyCalcAddTestSuite, MyCalcSubtractTestSuite,\
      MyCalcMultiplyTestSuite,MyCalcDivideTestSuite ]

    loader = unittest.TestLoader()

    test_suites = []
    for t_class in test_classes:
        suite = loader.loadTestsFromTestCase(t_class)
        test_suites.append(suite)

    final_suite = unittest.TestSuite(test_suites)

    runner = unittest.TextTestRunner()
    results = runner.run(final_suite)

if __name__ == '__main__':
    run_mytests()
```

Мы познакомились с библиотекой unittest. Теперь перейдем к созданию тестовых сценариев с помощью библиотеки pytest.

Фреймворк тестирования pytest

Тестовые сценарии, написанные с помощью unittest, легче читаются и управляются, особенно если вы уже работали с JUnit или аналогичными платформами. Но для крупномасштабных приложений Python библиотека pytest предпочтительнее из-за простоты реализации и способности адаптироваться под сложные требования тестирования. В случае с pytest нет необходимости расширять класс модульного теста из любого базового класса. Можно писать тест-кейсы без реализации каких-либо классов.

Данный фреймворк имеет открытый исходный код. Он может автоматически находить тесты для выполнения, подобно unittest, если имя файла имеет префикс test. И такое поведение можно настраивать. Фреймворк pytest предлагает функционал, подобный unittest, поэтому сосредоточимся на функциях, которые отличаются или дополняют платформу.

Создание тестов без базового класса

Для демонстрации пересмотрим модуль myadd2.py, реализуя функцию add без класса. Эта новая функция сложит два числа и выдаст исключение, если в качестве аргумента получит нечисловое значение. Ниже приведен пример использования pytest:

```
# myadd3.py – класс с методом сложения двух чисел

def add(self, x, y):
    """Эта функция складывает 2 числа"""
    if (not isinstance(x, (int, float))) | not isinstance(y, (int, float))):
        raise TypeError("only numbers are allowed")
    return x + y
```

А также модуль тестовых сценариев:

```
#test_myadd3.py тестовый набор для функции myadd

import pytest
from mypytest.src.myadd3 import add

def test_add1():
    """тест для проверки двух положительных чисел"""
    assert add(10, 5) == 15"

def test_add2():
    """" тест для проверки двух положительных чисел """
    assert add(10, -5) == 5, "should be 5"
```

Мы привели только два теста для test_myadd3.py, поскольку другие сценарии будут аналогичными. Их можно найти в каталоге исходного кода к этой главе на GitHub. Несколько ключевых отличий в реализации тестового сценария:

- Нет необходимости создавать тесты в классе; можно реализовать их как методы класса без наследования от базового класса; это ключевое отличие от библиотеки unittest.
- Операторы assert доступны в качестве ключевого слова для проверки любых условий с целью определения результата теста (успех или неудача); отделение ключевого слова assert от условных операторов делает проверки в тестовых сценариях гибкими и настраиваемыми.

Также следует отметить, фреймворк pytest делает консольный вывод и отчеты эффективнее. Пример консольного вывода после выполнения тестов в модуле test_myadd3.py:

```
test_myadd3.py::test_add1 PASSED                    [25%]
test_myadd3.py::test_add2 PASSED                    [50%]
```

```
test_myadd3.py::test_add3 PASSED                    [75%]
test_myadd3.py::test_add4 PASSED                    [100%]
================== 4 passed in 0.03s ==================
```

Далее рассмотрим, как с помощью pytest проверять ожидаемые ошибки.

Создание тестов с обработкой ошибок

Создание тест-кейсов для проверки ожидаемого исключения в pytest и unittest отличаются. Фреймворк pytest использует менеджер контекста для проверки исключений. В модуле test_myadd3.py мы уже добавили два тестовых примера для проверки исключений. Фрагмент кода приведен ниже:

```
def test_typeerror1():
    """тест для проверки, можем ли мы обработать нечисловое значение"""
    with pytest.raises(TypeError):
        add('a', 5)
def test_typeerror2():
    """тест для проверки, можем ли мы обработать нечисловое значение"""
    with pytest.raises(TypeError, match="only numbers are allowed"):
        add('a', 'b')
```

Для проверки исключения мы используем функцию raises с целью указать, какое исключение ожидается при запуске определенного фрагмента кода (в нашем первом тестовом примере add('a', 5)). Во втором сценарии мы использовали аргумент match для проверки сообщения, которое устанавливается при возникновении исключения.

Далее обсудим использование маркеров с фреймворком pytest.

Создание тестов с маркерами pytest

Фреймворк pytest оснащен *маркерами*, которые позволяют прикреплять метаданные или определять категории для наших тестов. Метаданные могут использоваться для многих целей, например, для включения или исключения определенных тестовых сценариев. Маркеры реализуются с помощью декоратора @pytest.mark.

В pytest есть несколько встроенных маркеров, самые популярные из них представлены ниже:

- skip: при использовании этого маркера исполнитель тестов пропустит сценарий при любых условиях;
- skipif: тест может быть пропущен в зависимости от условного выражения, которое передается этому маркеру как аргумент;
- xfail: ожидаемое неудачное завершение тестового случая будет проигнорировано; используется с определенным условием;

- `parametrize`: используется для выполнения нескольких вызовов тестового сценария с разными значениями аргументов.

Для демонстрации работы первых трех маркеров добавим их в модуль `test_add3.py` и перепишем пример. Новый тестовый модуль (`test_add4.py`) будет таким:

```
@pytest.mark.skip
def test_add1():
    """тест для проверки двух положительных чисел"""
    assert add(10, 5) == 15

@pytest.mark.skipif(sys.version_info > (3,6), \
reason=" skipped for release > than Python 3.6")
def test_add2():
    """тест для проверки двух положительных чисел"""
    assert add(10, -5) == 5, "should be 5"

@pytest.mark.xfail(sys.platform == "win32", \
reason="ignore exception for windows")
def test_add3():
    """тест для проверки двух положительных чисел"""
    assert add(-10, 5) == -5
    raise Exception()
```

Первый тест будет проигнорирован, так как мы использовали для него маркер `skip` без указания дополнительных условий. Для второго теста мы использовали маркер `skipif` с условием, что версия Python должна быть выше 3.6. Для последнего теста мы намеренно вызвали исключение и использовали маркер `xfail`. Этот тип исключения будет проигнорирован, если операционной системой является Windows. Данный маркер полезен для игнорирования ошибок, если они ожидаются для определенного условия, например, для операционной системы. Консольный вывод будет следующий:

```
test_myadd4.py::test_add1 SKIPPED (unconditional skip)              [33%]
Skipped: unconditional skip
test_myadd4.py::test_add2 SKIPPED ( skipped for release > than
Pytho...)                                                           [66%]
Skipped:  skipped for release > than Python 3.6
test_myadd4.py::test_add3 XFAIL (ignore exception for
mac)                                                                [100%]
@pytest.mark.xfail(sys.platform == "win32",
                                    reason="ignore exception for mac")
========== 2 skipped, 1 xfailed in 0.06s ==========
```

Теперь рассмотрим использование маркера `parametrize` из библиотеки `pytest`.

Создание тестов с параметризацией

В предыдущих примерах мы создавали тестовые сценарии без передачи им каких-либо параметров, но часто возникает необходимость запустить один и тот же тест, изменив входные данные. В классическом подходе мы выполняем несколько сценариев, которые отличаются только входными данными. Именно так мы делали в `test_myadd3.py`. Рекомендуемым подходом в таких ситуациях является *тестирование на основе данных* (**Data-driven Testing, DDT**), при котором тестовые данные предоставляются через словари или таблицы (**table** или **spreadsheet**). Такой подход также носит название *табличное тестирование* или *параметризированное тестирование*. Данные, предоставленные через таблицу или словарь, используются для выполнения тестов с помощью стандартной реализации исходного кода теста. Такой подход удобен в сценариях, когда нужно протестировать функционал, используя перестановку входных параметров. Вместо создания тестов для каждой перестановки мы можем предоставить их в формате таблицы или словаря и использовать в качестве входных данных для нашего единственного тестового примера. Фреймворки, вроде `pytest`, будут выполнять тест столько раз, сколько перестановок содержится в таблице или словаре. Примером может служить поведение функции авторизации, в которую подставляется множество допустимых и недопустимых учетных данных пользователей.

В `pytest` тестирование на основе данных можно реализовать, используя параметризацию с помощью маркера. Маркером `parametrize` можно определить, какой входной аргумент надо передать, а также набор тестовых данных, который нужно использовать. Фреймворк `pytest` автоматически выполнит тест множество раз в соответствии с количеством записей в тестовых данных, предоставленных маркером `parametrize`.

Для демонстрации изменим файл `myadd4.py`, в котором будет только один тестовый пример, но данные для входных параметров будут разными:

```
# test_myadd5.py набор тестов с маркером parameterize
import sys

import pytest
from mypytest.src.myadd3 import add

@pytest.mark.parametrize("x,y,ans", [(10,5,15),(10,-5,5),
                                     (-10,5,-5),(-10,-5,-15)],
                                ids=["pos-pos","pos-neg",
                                     "neg-pos", "neg-neg"])
def test_add(x, y, ans):
    """тест для проверки двух положительных чисел"""
    assert add(x, y) == ans
```

Для маркера `parametrize` мы указали три параметра, которые можно описать следующим образом:

- **Аргументы для тест-кейса**: мы предоставили список аргументов, которые должны быть переданы в тестовую функцию в порядке, определенном в ней; тестовые данные, которые необходимо указать в следующем аргументе, также должны следовать в этом порядке.
- **Данные**: тестовые данные представлены в виде списка различных наборов входных аргументов; количество записей в данных будет определять, сколько раз выполнится тест.
- `ids`: это необязательный параметр, прикрепляющий дружественный тег к тестовым наборам, которые мы указали в предыдущем аргументе; эти *теги-идентификаторы* (**ID**) будут использоваться в выходном отчете для идентификации различных исполнений тестового сценария.

Консольный вывод будет следующим:

```
test_myadd5.py::test_add[pos-pos] PASSED                [ 25%]
test_myadd5.py::test_add[pos-neg] PASSED                [ 50%]
test_myadd5.py::test_add[neg-pos] PASSED                [ 75%]
test_myadd5.py::test_add[neg-neg] PASSED                [100%]
========== 4 passed in 0.04s ==========
```

Вывод показывает, сколько раз выполняется тест и с какими данными. Тестовые сценарии, написанные с помощью маркеров `pytest`, лаконичны и просты в реализации. Мы экономим много времени и можем за короткое время сделать больше тестов, лишь меняя входные данные.

Далее обсудим еще одно важное свойство — фикстуры.

Создание тестов с фикстурами pytest

Фикстуры можно реализовать с помощью декоратора `@pytest.fixture`. В `pytest` они реализованы эффективнее, чем в других фреймворках, и тому есть несколько причин:

- **Высокая масштабируемость**: мы можем определить общую настройку или фикстуры, которые можно повторно использовать через функции, классы, модули и пакеты.
- **Модульная реализация**: в тесте могут быть использованы одна или несколько фикстур; они, в свою очередь, могут использовать другие фикстуры (аналогично вызову одной функции из тела другой).
- **Гибкость**: каждый сценарий в наборе может использовать один и тот же или другой набор фикстур.
- Мы можем создавать фикстуры с заданной областью действия; по умолчанию — это `function`, что означает, фикстура будет выполняться перед каждой тестовой функцией (сценарием); можно определить следующие области:
  - `Function`: фикстура уничтожается после выполнения теста.

- `Module`: фикстура уничтожается после выполнения последнего теста в модуле.
- `Class`: фикстура уничтожается после выполнения последнего теста в классе.
- `Package`: фикстура уничтожается после выполнения последнего теста в пакете.
- `Session`: фикстура уничтожается после выполнения последнего теста в сеансе.

Фреймворк имеет несколько полезных встроенных фикстур, которые можно использовать «из коробки». Например, `capfd` для захвата вывода в дескрипторы файлов, `capsys` для вывода данных в `stdout` и `stderr`, `request` для предоставления информации о запрашиваемой тестовой функции и `testdir` для предоставления временного каталога для выполнения тестов.

Фикстуры в `pytest` также можно использовать для очистки в конце тестового сценария. Об этом мы поговорим чуть позже.

В следующем примере мы создадим тесты для класса `MyCalc`, используя пользовательские фикстуры. Фрагмент кода для `MyCalc` мы уже публиковали выше в подразделе *«Выполнение нескольких наборов тестов»*. Реализация фикстур и тестов приведена ниже:

```
#test_mycalc1.py тест функций калькулятора с помощью фикстур
import sys

import pytest
from mypytest.src.myadd3 import add
from mypytest.src.mycalc import MyCalc

@pytest.fixture(scope="module")
def my_calc():
    return MyCalc()

@pytest.fixture
def test_data ():
    return {'x':10, 'y':5}

def test_add(my_calc, test_data):
    """тест для сложения двух чисел"""
    assert my_calc.add(test_data.get('x'),test_data.get('y')) == 15

def test_subtract(my_calc, test_data):
    """тест для разности двух чисел"""
    assert my_calc.subtract(test_data.get('x'), test_data.get('y'))== 5
```

На что следует обратить внимание:

1. Мы создали две фикстуры: `my_calc` и `test_data`. Для фикстуры `my_calc` задана область `module`, поскольку мы хотим реализовать ее выполнение один раз для пре-

доставления экземпляра класса MyCalc. Фикстура test_data использует область по умолчанию (function), то есть будет выполняться перед каждым методом.

2. Для тестов (test_add и test_subtract) мы использовали фикстуры в качестве входных аргументов. Имя аргумента должно совпадать с именем функции фикстуры. Фреймворк pytest автоматически ищет фикстуру с именем, указанным в аргументе.

В нашем примере мы используем фикстуру в целях подготовки тестовых функций. Что касается очистки после выполнения тестов, существуют два подхода для реализации данного функционала, и мы обсудим их далее.

Использование оператора yield вместо return

При таком подходе мы пишем некоторый код для этапа подготовки, используем оператор yield вместо return, а затем пишем код для этапа очистки после инструкции yield. Если у нас есть набор тестов или модуль с множеством фикстур, исполнитель будет выполнять каждую фикстуру (в установленном порядке), пока не встретит оператор yield. Как только выполнение тест-кейса завершено, исполнитель запускает выполнение всех полученных фикстур и выполняет код после инструкции yield. Данный подход является рекомендуемым, поскольку мы получаем код, который легко поддерживать.

Добавление финализатора с помощью фикстуры request

В этом подходе мы должны рассмотреть три шага для создания метода очистки:

- Мы должны использовать в фикстурах объект request; он может быть предоставлен с помощью встроенной фикстуры с тем же именем.
- Мы определим метод teardown отдельно или как часть реализации фикстуры.
- Мы предоставим teardown как вызываемый метод для объекта request, используя метода addfinalizer.

Для демонстрации обоих подходов изменим предыдущую реализацию фикстур. В обновленном коде реализуем фикстуру my_calc, использующую yield-подход, и фикстуру data_set, использующую addfinalizer. Исправленный пример кода:

```
#test_mycalc2.py тест функций калькулятора с помощью фикстур
<операторы import >
@pytest.fixture(scope="module")
def my_calc():
    my_calc = MyCalc()
    yield my_calc
    del my_calc
```

```
@pytest.fixture
def data_set(request):
    dict = {'x':10, 'y':5}
    def delete_dict(obj):
        del obj
    request.addfinalizer(lambda: delete_dict(dict))
    return dict
<остальные тестовые случаи>
```

Обратите внимание, для этих примеров нет реальной необходимости в реализации функционала, мы добавили их в целях демонстрации.

> ***Подсказка***
>
> Использование `nose` и `doctest` для автоматизации тестов аналогично использованию фреймворков `unittest` и `pytest`.

Далее мы поговорим о разработке через тестирование.

Разработка через тестирование

Разработка через тестирование (**Test-driven development**, **TDD**) — хорошо известная практика в разработке ПО, при которой мы сначала пишем тесты, а затем код для необходимой функции в приложении. Подход имеет три простых правила:

- Не стоит писать какой-либо функциональный код, пока модульные тесты для него завершаются неудачей.
- Не стоит в том же тесте писать кода больше, чем необходимо для неудачного результата.
- Не стоит писать функционального кода больше, чем необходимо для прохождения неудачного теста.

Эти правила заставляют нас следовать известному трёхэтапному подходу к разработке ПО, который носит название *«Красный, Зеленый, Рефакторинг»* (**Red**, **Green**, **Refactor**). Эти этапы повторяются непрерывно. Они изображены на **рис. 5.4** и описаны далее.

Красный

На этом этапе мы пишем тест без какого-либо кода, который нужно проверить. Очевидно, что в этом случае тест завершится неудачей. Мы не будем пытаться писать полный тестовый сценарий, а только достаточное количество кода для провала теста.

Зеленый

На этом этапе мы пишем код, пока не сможем пройти тест. Опять же, количество кода не должно превышать необходимый минимум для прохождения теста. После мы запускаем все тесты с целью убедиться, что ранее написанные кейсы тоже завершаются успехом.

Рефакторинг

На этом этапе мы повышаем качество кода, а именно, облегчаем его чтение и оптимизируем использование, например, весь *жесткий код* (**Hardcode**) должен быть удален. Также рекомендуется запускать тесты после каждого цикла рефакторинга, результатом которого является чистый код. Можно повторять этот цикл, добавляя все больше тестовых сценариев и нового кода, пока не будет разработан весь функционал.

Важно понимать, что TDD не является ни подходом к тестированию, ни подходом к проектированию. Это подход к разработке ПО в соответствии со спецификациями, которые определены написанием тестов в первую очередь.

На следующей схеме представлены три этапа TDD (рис. 5.4):

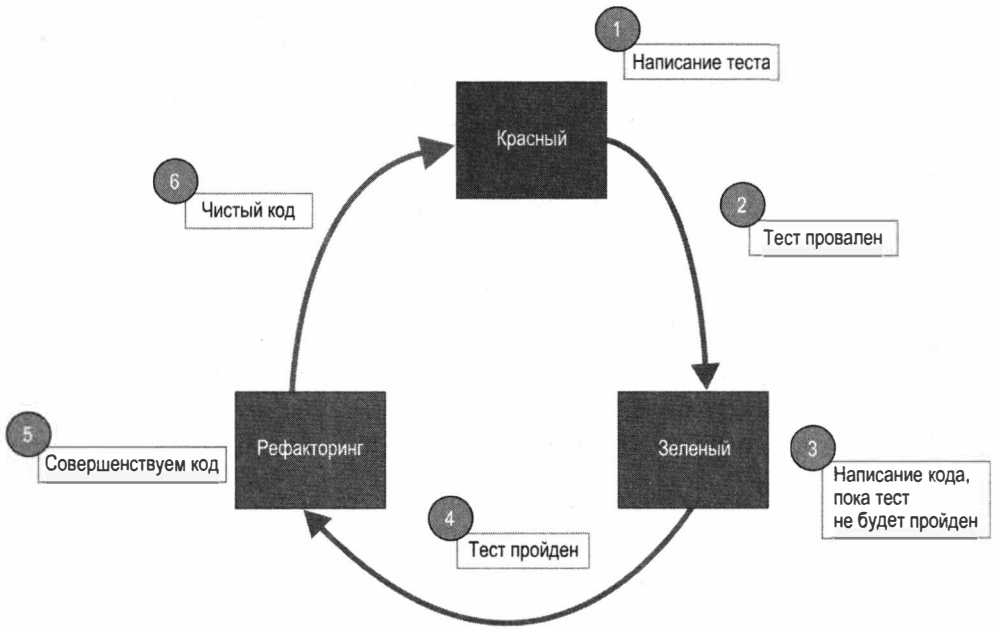


Рис. 5.4. TDD: красный, зеленый, рефакторинг

Далее мы расскажем о роли автоматизации тестов в процессе непрерывной интеграции.

Автоматизированная непрерывная интеграция

Непрерывная интеграция (**continuous integration, CI**) — это процесс, который сочетает преимущества автоматизированного тестирования и систем контроля версий для обеспечения полностью автоматизированной среды интеграции. Этот подход подразумевает внедрение кода в общий репозиторий. При каждом добавлении кода в репозиторий ожидается, что будут запущены следующие два процесса:

- Автоматизированный процесс сборки для подтверждения, что новый код ничего не нарушает с точки зрения компиляции или синтаксиса.
- Автоматизированное выполнение тестов для проверки, что действующий и новый функционал соответствуют определенным тестам.

Различные шаги и этапы CI показаны на схеме ниже. Несмотря на наличие на схеме фазы сборки, для проектов Python она необязательна, поскольку можно выполнять интеграционные тесты без скомпилированного кода (рис. 5.5):



Рис. 5.5. Фазы тестирования в рамках непрерывной интеграции

Для создания CI-системы требуется стабильный распределенный контроль версий и инструмент, с помощью которого можно реализовать рабочий поток для тестирования целого приложения через серию наборов тестов. Существует несколько коммерческих и свободных решений для непрерывной интеграции и *непрерывной доставки* (**CD**). Эти инструменты предназначены для простой интеграции с системой контроля версий и фреймворком автоматизации тестирования. Популярные инструменты для CI: *Jenkins*, *Bamboo*, *Buildbot*, *GitLab CI*, *CircleCI* и *Buddy*. Подробная информация о них приведена ниже в подразделе *«Дополнительные ресурсы»*.

Очевидные преимущества автоматизированного CI заключаются в быстром обнаружении ошибок и в их удобном и максимально быстром исправлении. Важно понимать, что CI предназначена не для исправления ошибок, но определенно помогает их легко выявлять и оперативно исправлять.

Заключение

В этой главе мы рассмотрели различные уровни тестирования приложений и два фреймворка (`unittest` и `pytest`) для автоматизации тестов на Python. Узнали, как создавать сценарии базового и расширенного уровня с помощью них. Затем мы познакомились с подходом TDD (разработка через тестирование) и его очевидными преимуществами. В конце мы затронули основы непрерывной интеграции (CI), которая является ключевым шагом в разработке ПО с использованием моделей **Agile** и **Development-Operations** (**DevOps**).

Эта глава будет полезна всем, кто хочет научиться писать модульные тесты для приложений Python. Приведенные примеры кода станут хорошей отправной точкой для написания тестовых сценариев с помощью любого фреймворка.

В следующей главе поговорим о различных расширенных приемах и советах по разработке приложений на Python.

Вопросы

1. Модульное тестирование выполняется методом белого или черного ящика?
2. Когда следует использовать фиктивные объекты (mock-объекты)?
3. Какие методы используются для реализации тестовых фикстур с фреймворком `unittest`?
4. Чем разработка через тестирование (TDD) отличается от непрерывной интеграции (CI)?
5. Когда следует использовать тестирование на основе данных (DDT)?

Дополнительные ресурсы

- *«Learning Python Testing»*, автор: Дэниел Арбакл (Daniel Arbuckle).
- «Python. Разработка на основе тестирования» (Test-Driven Development with Python), автор: Гарри Персиваль (Harry J.W. Percival).
- «Python. Лучшие практики и инструменты» (Expert Python Programming) авторы: Михал Яворски (Michał Jaworski) и Тарек Зиаде (Tarek Ziadé).
- Сведения о фреймворке unittest в документации Python: *https://docs.python.org/3/library/unittest.html*.

Ответы

1. Метод белого ящика.
2. Фиктивные объекты имитируют поведение внешних или внутренних зависимостей. С их помощью можно сосредоточиться на создании тестов для проверки функционального поведения.

   `setUp`, `tearDown`, `setUpClass`, `tearDownClass`.

3. TDD — это подход к разработке ПО, при котором сначала пишутся тесты. CI — это процесс, при котором все тесты выполняются при каждой сборке нового релиза. Прямой связи между TDD и CI нет.
4. DDT используется, когда необходимо провести функциональное тестирование с несколькими перестановками входных параметров. Например, если нам нужно протестировать конечную точку API с разными комбинациями входных аргументов.

6

Дополнительные советы и приемы Python

В этой главе мы рассмотрим дополнительные рекомендации по решению сложных задач в Python. Мы познакомимся с вложенными функциями и лямбда-функциями, а также поговорим о создании декораторов. Изучим преобразования данных с помощью методов `filter`, `map` и `reduce`. Узнаем, как работать со сложными структурами данных вроде вложенных словарей и включений для разных типов коллекций. В конце познакомимся с функционалом библиотеки **pandas** для объектов **DataFrame**. Все эти приемы не только помогут в решении сложных задач с наименьшим количеством кода, но и научат писать программы быстрее и эффективнее.

Темы этой главы:

- Расширенные приемы использования функций.
- Расширенные концепции структур данных.
- Введение в pandas DataFrame.

К концу главы вы научитесь использовать функции Python для выполнения таких сложных задач, как преобразование данных и создание декораторов. Также вы узнаете, как использовать структуры данных, включая pandas DataFrame, в аналитических приложениях.

Технические требования

Для этой главы понадобится:

- Python 3.7 или более поздней версии.
- Аккаунт Test PyPI и токен API в этом аккаунте.

Примеры кода для этой главы можно найти по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter06.

Начнем с приемов по использованию функций в Python.

Расширенные приемы использования функций в Python

Реализация функций в Python и других языках программирования является ключом к модульности и многократному использованию кода. В современных языках роль функций расширена и включает в себя создание простого, лаконичного кода без использования сложных циклов и условных операторов.

Начнем с функций `counter`, `zip` и `itertools`.

Функции counter, itertools и zip для итерационных задач

Для задач по обработке данных широко используются итераторы, которые мы подробно рассматривали в главе 4 (*«Библиотеки Python для продвинутого программирования»*). В этом разделе мы подробнее остановимся на функциях для удобной работы с итераторами и итерируемыми объектами. К ним относятся `counter`, `zip` и `itertools`, которые мы рассмотрим поподробнее.

Counter

Counter (счетчик) — это тип контейнера, который отслеживает количество каждого элемента, представленного в нем. Он полезен для подсчета частоты встречаемых данных для последующего анализа. Посмотрим на простом примере, как работает эта функция:

#counter.py

```
from collections import Counter

#применение счетчика к объекту строки
print(Counter("people"))

#применение счетчика к объекту списка
my_counter = Counter([1,2,1,2,3,4,1,3])
print(my_counter.most_common(1))
print(list(my_counter.elements()))

# применение счетчика к объекту словаря
print(Counter({'A': 2, 'B': 2, 'C': 2, 'C': 3}))
```

Мы создали несколько экземпляров `Counter` с объектами строки, списка и словаря. Класс `Counter` имеет методы `most_common` и `elements`. Мы использовали метод `most_common` со значением `1`, что позволяет получить элемент, который чаще всего

встречается в контейнере `my-counter`. Метод `elements` использован для возврата исходного списка из экземпляра `Counter`. Вывод консоли будет следующим:

```
Counter({'p': 2, 'e': 2, 'o': 1, 'l': 1})
[(1, 3)]
[1, 1, 1, 2, 2, 3, 3, 4]
Counter({'C': 4, 'A': 2, 'B': 2})
```

Важно отметить, что с объектом словаря мы намеренно использовали повторяющийся ключ, но в экземпляре `Counter` мы получаем только одну пару *«ключ-значение»*, которая встречается последней. Кроме того, элементы в экземпляре `Counter` упорядочены на основе значений каждого элемента. Также обратите внимание, класс `Counter` преобразует словарь в хеш-таблицу.

zip

Функция *zip* используется для создания объединенного итератора из двух и более отдельных итераторов. Это удобно, если надо выполнить несколько итераций параллельно. Например, можно использовать функцию zip для математических алгоритмов, связанных с интерполяцией или распознаванием образов. А также полезно при обработке цифровых сигналов, когда надо объединить несколько сигналов (источников данных) в один. Пример использования функции:

```
#zip.py

num_list = [1, 2, 3, 4, 5]
lett_list = ['alpha', 'bravo', 'charlie']

zipped_iter = zip(num_list,lett_list)
print(next(zipped_iter))
print(next(zipped_iter))
print(list(zipped_iter))
```

Мы объединили два списка с помощью функции `zip`. Обратите внимание, один список больше другого по количеству элементов. Вывод консоли будет следующим:

```
(1, 'alpha')
(2, 'bravo')
[(3, 'charlie'), (4, 'delta')]
```

Как и ожидалось, первые два кортежа мы получаем с помощью функции `next`, которые представляют собой комбинацию соответствующих элементов из каждого списка. Мы использовали конструктор `list` для перебора оставшихся кортежей из итератора `zip`. В итоге мы получили оставшиеся кортежи в формате списка.

itertools

При работе с большим объемом данных без итераторов не обойтись. Модуль `itertools` может предложить дополнительные функции, которые будут полезны. Рассмотрим несколько из них:

- `count`: используется при создании итератора для подсчета чисел; можно указать начальное число (по умолчанию 0) и по желанию задать шаг приращения; фрагмент кода ниже возвращает итератор, который предоставляет числа 10, 12 и 14:

```
#itertools_count.py
import itertools
iter = itertools.count(10, 2)
print(next(iter))
print(next(iter))
```

- `cycle`: позволяет бесконечно перебирать итератор; фрагмент ниже демонстрирует использование этой функции для списка букв алфавита:

```
letters = {'A','B','C'}
for letter in itertools.cycle(letters):
    print(letter)
```

- `repeat`: предоставляет итератор, который снова и снова возвращает объект, если для него не задан аргумент `times`; фрагмент кода ниже будет повторять строковый объект пять раз:

```
for x in itertools.repeat('Python', times=5):
    print(x)
```

- `accumulate`: возвращает итератор, который предоставляет общую сумму или другие накопленные результаты на основе функции-агрегатора, которая была передана в `accumulate` в качестве аргумента; легче понять ее работу на примере:

```
#itertools_accumulate.py
import itertools, operator

list1 = [1, 3, 5]
res = itertools.accumulate(list1)
print("default:")
for x in res:
    print(x)
res = itertools.accumulate(list1, operator.mul)
print("Multiply:" )
for x in res:
    print(x)
```

- Сначала мы использовали функцию `accumulate`, не предоставив агрегатор для получения накопленных результатов; по умолчанию `accumulate` выполнит сложение двух чисел (`1` и `3`) из исходного списка; этот процесс повторится для всех чисел,

а результат сохранится внутри итерируемого объекта (в нашем случае это `res`); во второй части кода мы предоставили функцию `mul` (умножение) из модуля `operator`, и на этот раз будет произведено умножение вместо сложения.

- `chain`: комбинирует два и более итерируемых объекта и возвращает объединенный объект; ниже приведен пример с двумя итерируемыми объектами (списками) и функцией `chain`:

```
list1 = ['A','B','C']
list2 = ['W','X','Y','Z']

chained_iter = itertools.chain(list1, list2)
for x in chained_iter:
    print(x)
```

- Обратите внимание, функция объединяет объекты последовательно, то есть сначала будут доступны элементы из `list1`, а затем из `list2`.
- `compress`: используется для фильтрации элементов из одного набора на основе другого; в примере продемонстрирована выборка букв алфавита из списка на основе итерируемого объекта `selector`:

```
letters = ['A','B','C']
selector = [True, 0, 1]
for x in itertools.compress(letters, selector):
 print (x)
```

- Для объекта `selector` можно использовать значения `True`/`False` или `1`/`0`; на выходе будут буквы `A` и `C`.
- `groupby`: определяет ключи для каждого элемента в объекте и группирует их на основе этих ключей; для работы функции требуется другая функция (`key_func`), которая идентифицирует ключ в каждом элементе итерируемого объекта; следующий пример демонстрирует использование обеих функций:

```
#itertools_groupby.py
import itertools

mylist = [("A", 100), ("A", 200), ("B", 30), "B", 10)]
def get_key(group):
    return group[0]

for key, grp in itertools.groupby(mylist, get_key):
    print(key + "-->", list(grp))
```

- `tee`: используется для дублирования итераторов из единого объекта; ниже приведен пример дублирования двух итераторов из единого списка:

```
letters = ['A','B','C']
iter1, iter2 = itertools.tee(letters)
```

```
for x in iter1:
    print(x)
for x in iter2:
    print(x)
```

Далее мы рассмотрим еще одну категорию функций, которые широко используются для преобразования данных.

Использование методов filter, map и reduce для преобразования данных

Методы `map`, `filter` и `reduce` — это три функции Python, которые помогают писать простой и лаконичный код. Они применяются ко всему набору сразу, без перебора или использования итерационных операторов. Методы `map` и `filter` доступны как встроенные и используются для преобразования и фильтрации данных. Метод `reduce` необходимо импортировать из модуля `functtools`, и он полезен при анализе для получения осмысленных результатов из большого набора данных.

Далее мы рассмотрим каждую функцию с примерами.

map

Функция `map` определяется с помощью следующего синтаксиса:

```
map(func, iter, ...)
```

Аргумент `func` — это имя функции, которая будет применяться к каждому элементу объекта `iter`. Три точки указывают, что можно передать несколько итерируемых объектов. Однако важно понимать, что количество аргументов функции (`func`) должно совпадать с числом итерируемых объектов. Результатом функции `map` будет объект `map` (или `map`-объект), который является генератором. Возвращаемое значение можно преобразовать в список, передав объект `map` конструктору `list`.

> ***Важное примечание***
>
> В Python 2 функция `map` возвращает список. В Python 3 это поведение было изменено.

Перед обсуждением работы `map` реализуем простую функцию преобразования, которая конвертирует список чисел в список квадратов их значений. Пример кода:

`#map1.py` `вычисление квадрата каждого элемента в списке`

```
mylist = [1, 2, 3, 4, 5]
new_list = []

for item in mylist:
     square = item*item
    new_list.append(square)

print(new_list)
```

Здесь используется цикл `for` для перебора списка, вычисления квадрата каждого элемента, а затем добавления в новый список. Такой подход распространен, но не соответствует философии Python. Вывод консоли будет следующим:

```
[1, 4, 9, 16, 25]
```

С помощью `map` этот код можно упростить:

```
# map2.py вычисление квадрата каждого элемента в списке

def square(num):
    return num * num

mylist = [1, 2, 3, 4, 5]
new_list = list(map(square, mylist))
print(new_list)
```

Используя `map`, мы предоставили имя функции (`square`) и ссылку на список (`mylist`). Возвращаемый объект будет преобразован в список с помощью конструктора `list`. Консольный вывод будет аналогичен предыдущему примеру.

В следующем примере мы предоставим два списка в качестве входных данных для функции `map`:

```
#map3.py вычисление квадрата каждого элемента в списке

def product(num1, num2):
    return num1 * num2

mylist1 = [1, 2, 3, 4, 5]
mylist2 = [6, 7, 8, 9]
new_list = list(map(product, mylist1, mylist2))
print(new_list)
```

На этот раз функция `map` будет использовать функцию `product`, которая берет соответствующие элементы из двух списков и перемножает их, прежде чем вернуть обратно в функцию `map`.

На консоли будет следующее:

```
[6, 14, 24, 36]
```

Анализ вывода говорит, что только первые четыре элемента из каждого списка используются функцией `map`. Она автоматически останавливает выполнение, когда в любом из итерируемых объектов заканчиваются элементы. Это означает, что, даже если мы предоставим объекты разных размеров, функция `map` не вызовет никаких исключений, а просто будет работать с тем количеством элементов, которое можно сопоставить между наборами. Поэтому в нашем примере в итоговом списке `new_list` всего четыре элемента.

filter

Функция `filter` также работает с наборами, но только с одним за раз. Она обеспечивает возможность фильтрации итерируемого объекта по критериям, которые предоставляются через определение функции. Синтаксис функции следующий:

```
filter (func, iter)
```

Функция `func` предоставляет критерии фильтрации и должна возвращать значения `True` или `False`. Поскольку `filter` работает только с одним объектом, для `func` допускается только один аргумент. В следующем примере `filter` используется для выбора элементов, значения которых являются четными числами. Для задания критериев выбора реализована функция `is_even`, которая оценивает, является число четным или нет. Пример кода:

```
#filter1.py получение четных чисел из списка

def is_even(num):
    return (num % 2 == 0)

mylist = [1, 2, 3, 4, 5,6,7,8,9]
new_list = list(filter(is_even, mylist))
print(new_list)
```

На консоли будет следующий вывод:

```
[2, 4, 6, 8]
```

reduce

Функция `reduce` используется для применения *кумулятивной обработки* (**Cumulative Processing**) к каждому элементу последовательности, которая передается ей в качестве аргумента. Функция кумулятивной обработки не предназначена для преобразования или фильтрации. Как следует из названия, она нужна для получения в конце единого результата на основе всех элементов последовательности. Синтаксис функции следующий:

```
reduce (func, iter[,initial])
```

Функция `func` используется для применения кумулятивной обработки к каждому элементу итерируемого объекта. Кроме того, `initial` — это опциональное значение, которое можно передать в `func` в качестве начального значения для обработки. Важно понимать, что в `func` всегда нужно будет передавать два аргумента в случае использования `reduce`: первым аргументом будет начальное значение (если оно указано) или первый элемент последовательности, а вторым аргументом будет следующий элемент последовательности.

В примере ниже использован простой список из пяти чисел. Мы реализуем собственный метод для сложения двух чисел, а затем используем `reduce` для суммирования всех элементов списка. Пример кода:

```
#reduce1.py вычисление суммы чисел из списка
from functools import reduce

def seq_sum(num1, num2):
    return num1+num2

mylist = [1, 2, 3, 4, 5]
result = reduce(seq_sum, mylist)
print(result)
```

В выводе будет `15`, что является суммой всех элементов списка `mylist`. Если мы предоставим функции `reduce` начальное значение, оно будет включено в результат. Например, выводом той же программы с оператором ниже будет `25`:

```
result = reduce(seq_sum, mylist, 10)
```

Как уже упоминалось, `reduce` выдает только одно значение, которое соответствует функции `func`. В нашем примере это будет целое число.

Мы рассмотрели методы `map`, `filter` и `reduce`, доступные в Python. Они широко используются в data science для преобразования и обработки данных. Одна из проблем использования `map` и `filter` заключается в том, что они возвращают объекты типа `map` и `filter`, которые нужно явно преобразовывать в `list` для дальнейшей обработки. Включения и генераторы не имеют таких ограничений, при этом предоставляют аналогичную функциональность. Использовать их относительно проще. Поэтому они пользуются большей популярностью, чем `map`, `filter` и `reduce`. Включения и генераторы мы обсудим в подразделе *«Расширенные концепции структур данных»*, а сейчас поговорим о лямбда-функциях.

Создание лямбда-функций

Лямбда-функции (**Lambda function**) — это анонимные функции, основанные на однострочном выражении. В случае с регулярными функциями мы определяем их ключевым словом `def`, тогда как анонимные функции определяются словом `lambda`. Они ограничены одной строкой, поэтому не могут содержать несколько операторов и не могут использовать оператор `return`. Значение возвращается автоматически после вычисления однострочного выражения.

Лямбда-функции можно использовать везде, где используются обычные функции. Самый простой и удобный способ — `map`, `reduce` и `filter`. Такие функции помогают сделать код аккуратным.

Для демонстрации повторно используем предыдущие примеры с `map` и `filter`, но изменим код, заменив `func` на лямбда-функцию:

```
#lambda1.py вычисление квадрата каждого элемента в списке

mylist = [1, 2, 3, 4, 5]
new_list = list(map(lambda x: x*x, mylist))
print(new_list)

#lambda2.py получение четных чисел из списка

mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
new_list = list(filter(lambda x: x % 2 == 0, mylist))
print(new_list)

#lambda3.py вычисление произведения элементов в двух списках

mylist1 = [1, 2, 3, 4, 5]
mylist2 = [6, 7, 8, 9]
new_list = list(map(lambda x,y: x*y, mylist1, mylist2))
print(new_list)
```

Хотя код стал проще, использовать лямбда-функции следует с осторожностью, поскольку их нельзя использовать повторно и сложно поддерживать. Нужно об этом помнить, прежде чем вводить их в эксплуатацию. Добавлять любые изменения и новый функционал будет непросто. Главное правило состоит в том, что использовать их стоит только для простых выражений, когда написание отдельной функции было бы накладно.

Внедрение одной функции в другую

Когда мы добавляем одну функцию внутри другой, она носит название *внутренней* или *вложенной*. Преимущества таких функций заключается в прямом доступе к переменным, которые определены или доступны в области видимости внешней функции. Создать внутреннюю функцию можно аналогично обычной функции, используя ключевое слово `def`, но с соответствующим отступом. Внутренние функции не могут выполняться или вызываться внешней программой. Однако если внешняя функция возвращает ссылку на внутреннюю, та может быть использована вызывающей стороной. Рассмотрим примеры возвращения ссылок позже.

А сейчас обсудим несколько преимуществ и способов применения данных функций.

Инкапсуляция

Распространенный вариант использования вложенных функций — возможность скрыть их функционал от внешнего мира. Они доступны только в пределах области действия внешней функции и скрыты от области действия глобальной функции.

Следующий пример демонстрирует одну внешнюю функцию, которая скрывает внутреннюю:

#inner1.py

```
def outer_hello():
    print ("Hello from outer function")
    def inner_hello():
        print("Hello from inner function")
    inner_hello()

outer_hello()
```

В теле основной программы можно вызывать только внешнюю функцию. Внутреннюю функцию можно вызывать только из тела внешней.

Вспомогательные функции

В некоторых случаях можно обнаружить, что часть кода внутри функции доступна для повторного использования, поэтому есть возможность вынести его в отдельную функцию. Но если код допустимо использовать только внутри функции, тогда речь идет о создании именно внутренней функции. Она также называется *вспомогательной*. Пример такой функции представлен ниже:

```
def outer_fn(x, y):
    def get_prefix(s):
        return s[:2]
    x2 = get_prefix(x)
    y2 = get_prefix(y)
    #дальнейшая обработка x2 and y2
```

Мы определили внутреннюю функцию `get_prefix` (вспомогательная функция) внутри внешней функции фильтрации первых двух букв значения аргумента. Поскольку фильтрация будет повторяться для всех аргументов, мы добавили вспомогательную функцию, которую можно будет использовать повторно внутри внешней функции.

Замыкания и фабричные функции

Это вариант использования, где внутренние функции проявляют себя лучше всего. *Замыкание* (**Closure**) — это вложенная функция вместе с ее окружающей средой. Оно создается динамически и может возвращаться другой функцией. Главное преимущество состоит в том, что возвращаемая функция имеет полный доступ к переменным и пространствам имен, в которых она была создана, даже если замыкающая (в нашем случае внешняя) функция завершила выполнение.

Данную концепцию можно продемонстрировать на примере. В коде ниже мы реализовали *фабрику замыканий* (**Closure factory**) для создания функции, которая вычисляет степень основания, а замыкание будет сохранять основание:

```
#inner2.py
def power_calc_factory(base):
    def power_calc(exponent):
        return base**exponent
    return power_calc

power_calc_2 = power_gen_factory(2)
power_calc_3 = power_gen_factory(3)
print(power_calc_2(2))
print(power_calc_2(3))
print(power_calc_3(2))
print(power_calc_3(4))
```

В примере внешняя функция (`power_calc_factory`) выступает как фабрика замыканий, поскольку создает новое замыкание при каждом вызове, а затем возвращает его вызывающему объекту. Кроме того, `power_calc` — это внутренняя функция, которая принимает одну переменную (`base`) из пространства имен замыкания, а затем вторую переменную (`exponent`), которая передается в качестве аргумента. Обратите внимание, наиболее важным оператором здесь является `return power_calc`. Он возвращает внутреннюю функцию как объект с его замыканием.

Когда мы вызываем `power_calc_factory` в первый раз вместе с аргументом `base`, создается замыкание с его пространством имен, включая переданный аргумент, и затем возвращается вызывающему объекту. Когда мы снова вызываем ту же функцию, мы получаем новое замыкание с объектом внутренней функции. В этом примере мы создали два замыкания: одно со значением `base`, равным 2, и второе со значением `base`, равным 3. Когда мы вызываем внутреннюю функцию, передавая разные значения для переменной `exponent`, код в ней (в нашем случае `power_calc`) также будет иметь доступ к значению `base`, которое уже было передано внешней функции.

Традиционно вложенные функции используются для сокрытия или инкапсуляции функционала внутри функции. Но наиболее мощное их применение проявляется, когда они используются вместе с внешними функциями, выступающими в качестве фабрики для создания динамических функций. Внутренние функции также используются для реализации декораторов. Более подробно мы обсудим это в следующем подразделе.

Изменение поведения функции с помощью декораторов

Концепция декораторов в Python основана на шаблоне структурного проектирования **Decorator**. Он позволяет добавлять к объектам новое поведение, ничего не ме-

няя в самой их реализации. Это поведение добавляется в специальные объекты-оболочки.

В Python декораторы — это специальные функции высшего порядка, которые позволяют разработчикам добавлять новый функционал в существующую функцию (или метод), ничего не изменяя внутри нее. Обычно декораторы добавляются перед определением функции. Они используются для реализации многих возможностей приложения, но особенно популярны при проверке данных, логировании, кэшировании, отладке, шифровании и управлении транзакциями.

Для создания декоратора мы должны определить вызываемый объект (функцию, метод или класс), который принимает функцию в качестве аргумента. Вызываемый объект вернет другой объект-функцию с поведением, которое определил декоратор. Функция, отмеченная декоратором (будем называть ее *декорированная функция*), передается в качестве аргумента функции, которая реализует декоратор (будем называть ее *функция-декоратор*). Функция-декоратор выполняет переданную ей функцию в дополнение к поведению, добавленному как часть функции-декоратора.

Ниже приведен простой пример, в котором мы определяем декоратор для добавления метки времени до и после выполнения функции:

```
#decorator1.py
from datetime import datetime

def add_timestamps(myfunc):
    def _add_timestamps():
        print(datetime.now())
        myfunc()
        print(datetime.now())
    return _add_timestamps

@add_timestamps
def hello_world():
    print("hello world")

hello_world()
```

Мы определили функцию-декоратор `add_timestamps`, которая принимает любую функцию в качестве аргумента. Во внутренней функции (`_add_timestamps`) мы фиксируем время до и после выполнения функции и передаем его в качестве аргумента. Функция-декоратор возвращает объект внутренней функции с замыканием. Как мы уже обсуждали, декораторы не делают ничего, кроме разумного использования внутренних функций. Использование символа `@` для декорирования функции эквивалентно следующим строкам кода:

```
hello = add_timestamps(hello_world)
hello()
```

В данном случае мы явно вызываем функцию-декоратор, передавая имя функции как параметр. Другими словами, декорированная функция равна внутренней функции, которая определяется внутри функции-декоратора. Именно так Python интерпретирует и вызывает функцию-декоратор, когда видит символ `@` перед определением функции.

Однако возникает проблема, когда необходимо получить больше информации о вызове функций, например, для отладки. Когда мы используем встроенный метод `help` с функцией `hello_world`, мы получаем справку только для внутренней функции. То же самое произойдёт при использовании docstring, которая сработает для внутренней функции, но не для декорированной. Кроме того, сериализация кода будет трудной задачей для декорированных функций. В Python есть простое решение: использовать декоратор `wraps` из библиотеки `functools`. Изменим пример кода, добавив декоратор `wraps`:

```
#decorator2.py
from datetime import datetime
from functools import wraps

def add_timestamps(myfunc):
    @wraps(myfunc)
    def _add_timestamps():
        print(datetime.now())
        myfunc()
        print(datetime.now())
    return _add_timestamps

@add_timestamps
def hello_world():
    print("hello world")

hello_world()
help(hello_world)
print(hello_world)
```

Использование `wraps` предоставит дополнительные сведения о выполнении вложенных функций, которые мы можем увидеть в консольном выводе при выполнении данного фрагмента кода.

До этого момента мы рассматривали простые примеры декораторов для понимания концепции. Дальше узнаем, как передавать аргументы с функцией в декоратор, как возвращать значение из декоратора и как объединять несколько декораторов в цепочку. Начнем с передачи атрибутов и возвращения значений с помощью декораторов.

Использование декорированной функции с возвращаемым значением и аргументом

Когда декорированная функция принимает аргументы, добавление декораторов требует дополнительных трюков. Например, можно использовать `*args` и `**kwargs` во внутренней функции-обертке. Это заставит ее принимать произвольное количество позиционных и ключевых аргументов. Ниже приведен простой пример декорированной функции с аргументами и возвращаемым значением:

#decorator3.py

```
from functools import wraps

def power(func):
    @wraps(func)
    def inner_calc(*args, **kwargs):
        print("Decorating power func")
        n = func(*args, **kwargs)
        return n
    return inner_calc

@power
def power_base2(n):
    return 2**n

print(power_base2(3))
```

Внутренняя функция `inner_calc` принимает общие параметры `*args` и `**kwargs`. Для возврата значения из `inner_calc` мы можем удержать его из функции (в нашем случае это `func` или `power_base2(n)`), которая выполняется внутри нашей внутренней функции. А затем вернуть итоговое значение из `inner_calc`.

Создание декоратора с собственными аргументами

В предыдущих примерах мы использовали так называемые *стандартные декораторы*. Это функции, которые принимают имя декорированной функции в качестве аргумента и возвращают внутреннюю функцию, работающую как функция-декоратор. Если декоратор имеет собственные аргументы, все работает немного иначе. А именно, он создается поверх стандартного. Проще говоря, декоратор с аргументами — это еще одна функция, которая на самом деле возвращает стандартный декоратор (а не внутреннюю функцию внутри декоратора). Эту концепцию лучше понять на измененном примере `decorator3.py`. В новом варианте мы вычисляем степень значения `base`, которое передается декоратору в качестве аргумента. Полный код с вложенными функциями-декораторами представлен ниже:

#decorator4.py

```
from functools import wraps
```

```
def power_calc(base):
    def inner_decorator(func):
        @wraps(func)
        def inner_calc(*args, **kwargs):
            exponent = func(*args, **kwargs)
            return base**exponent
        return inner_calc
    return inner_decorator

@power_calc(base=3)
def power_n(n):
    return n

print(power_n(2))
print(power_n(4))
```

Этот код работает следующим образом:

- Функция-декоратор `power_calc` принимает один аргумент `base` и возвращает функцию `inner_decorator`, которая является стандартной реализацией декоратора.
- Функция `inner_decorator` принимает функцию в качестве аргумента и возвращает функцию `inner_calc` для фактического вычисления.
- Функция `inner_calc` вызывает декорированную функцию для получения атрибута `exponent`, а затем использует атрибут `base`, который передается внешней функции-декоратору в качестве аргумента; как и ожидалось, замыкание вокруг внутренней функции делает значение атрибута `base` доступным для функции `inner_calc`.

Далее рассмотрим, как использовать более одного декоратора с функцией или методом.

Использование нескольких декораторов

Существует возможность использовать несколько декораторов с функцией. Это достигается объединением их в цепочку. Декораторы могут быть одинаковыми или разными. Реализовать это можно, размещая их один за другим перед определением функции. Когда с функцией используется несколько декораторов, выполнить ее можно только один раз. Для демонстрации рассмотрим пример, в котором мы заносим лог в журнал целевой системы, используя метку времени. Она добавляется через отдельный декоратор, а целевая система выбирается на основе другого декоратора. В следующем примере показаны определения трех декораторов (`add_time_stamp`, `file` и `console`):

```
#decorator5.py (part 1)
from datetime import datetime
from functools import wraps
```

```
def add_timestamp(func):
    @wraps(func)
    def inner_func(*args, **kwargs):
        res = "{}:{}\n".format(datetime.now(),func(*args,\
          **kwargs))
        return res
    return inner_func

def file(func):
    @wraps(func)
    def inner_func(*args, **kwargs):
        res = func(*args, **kwargs)
        with open("log.txt", 'a') as file:
            file.write(res)
            return res
    return inner_func

def console(func):
    @wraps(func)
    def inner_func(*args, **kwargs):
        res = func(*args, **kwargs)
        print(res)
        return res
    return inner_func
```

В этом примере мы реализовали три функции-декоратора:

- `file`: добавляет в предопределенный текстовый файл сообщение, предоставленное декорированной функцией;
- `console`: выводит сообщение от декорированной функции на консоль;
- `add_timestamp`: добавляет временну́ю метку перед сообщением; выполнение должно происходить перед декораторами `file` и `console`, а, значит, стоять в цепочке он должен последним.

В следующем фрагменте кода можно использовать эти декораторы для различных функций внутри основной программы:

```
#decorator5.py (part 2)
@file
@add_timestamp
def log(msg):
    return msg

@file
@console
@add_timestamp
def log1(msg):
    return msg
```

```
@console
@add_timestamp
def log2(msg):
    return msg

log("This is a test message for file only")
log1("This is a test message for both file and console")
log2("This message is for console only")
```

Мы использовали три функции-декоратора, определенные ранее, в разных комбинациях для демонстрации вариантов поведения одной функции. В первой комбинации мы выводим сообщение в файл только после добавления метки времени. Во второй — выводим сообщение и в файл, и на консоль. Наконец, в последней комбинации мы выводим сообщение только на консоль. Это показывает гибкость, которую предоставляют декораторы без необходимости изменения функций. Стоит отметить, что они полезны для упрощения кода и добавления поведения в лаконичной форме, но влекут за собой дополнительные затраты. Использовать их необходимо только в ситуациях, когда преимущества перевешивают издержки.

На этом можно закончить обсуждение концепций, связанных с расширенными функциями, и перейти к структурам данных.

Расширенные концепции структур данных

Python предлагает обширную поддержку для структур данных, в том числе ключевые инструменты для хранения, обработки и извлечения данных, а также доступа к ним. В главе 4 (*«Библиотеки Python для продвинутого программирования»*) мы уже рассматривали структуры данных, доступные в Python. Здесь мы затронем дополнительные концепции, например, *вложенные словари* и использование *включений* со структурами данных. Начнем со словарей.

Внедрение словаря в словарь

Python позволяет вкладывать один словарь в другой. Вложенный словарь имеет множество применений, например, в обработке и преобразовании данных из одного формата в другой.

На рис. 6.1 показан вложенный словарь. Корневой словарь имеет два словаря для ключа `1` и ключа `3`. Словарь для ключа `1` содержит в себе дополнительные словари. Словарь для ключа `3` — обычный словарь с парами *«ключ-значение»*.

Подобный корневой словарь можно записать следующим образом:

```
root_dict = {'1': {'A': {dictA}, 'B':{dictB}},
          '2': [list2],
             '3': {'X': val1,'Y':val2,'Z': val3}
             }
```

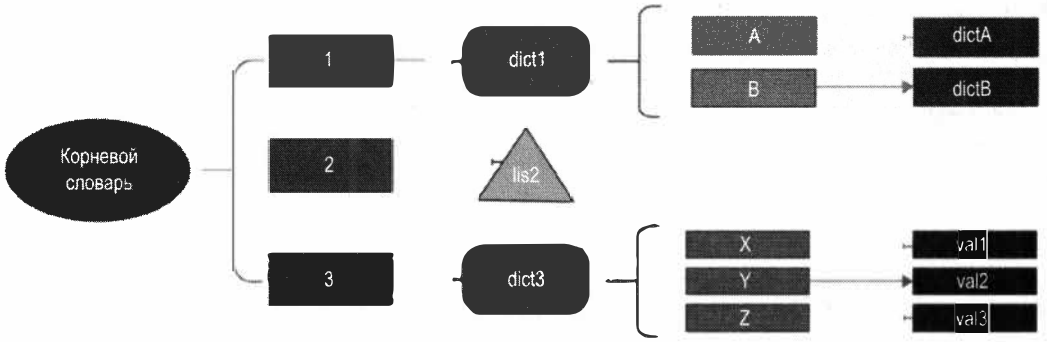


Рис. 6.1. Пример словаря в словаре

Мы создали корневой словарь с набором объектов словаря и списка.

Создание или определение вложенного словаря

Вложенный словарь можно определить или создать, поместив разделенные запятыми словари в фигурные скобки. Для демонстрации создадим словарь со студентами. Каждая запись о студенте будет иметь имя (`name`) и возраст (`age`) в качестве элементов, которые сопоставляются с номером студента:

`#dictionary1.py`

```
dict1 = {100:{'name':'John', 'age':24},
         101:{'name':'Mike', 'age':22},
         102:{'name':'Jim', 'age':21} }

print(dict1)
print(dict1.get(100))
```

Далее рассмотрим, как динамически создавать словари, а также добавлять или обновлять вложенные элементы.

Добавление во вложенный словарь

Динамически создать вложенный словарь или добавить в него элементы можно несколькими способами. В следующем примере рассмотрим три разных подхода. Они аналогичны способам в модуле `dictionary1.py`:

- В первом случае создаем внутренний словарь (`student101`) через прямое присвоение элементов *«ключ-значение»*, а затем назначаем его ключу в корневом словаре; такой подход предпочтителен, где это возможно, потому что код легче читать и обслуживать.
- Во втором случае создаем пустой внутренний словарь (`student102`) и задаем значения ключам с помощью операторов присваивания; такой подход рекомендован, когда значения доступны через другие структуры данных.

- В третьем случае напрямую инициируем пустой каталог для третьего ключа корневого словаря; после этого присваиваем значения, используя двойную индексацию (то есть два ключа): первый ключ для корневого словаря, а второй — для внутреннего; такой подход делает код кратким, но не является предпочтительным, если читабельность кода важна для дальнейшего обслуживания.

Полный пример кода для всех трех подходов выглядит следующим образом:

```
#dictionary2.py
#определение внутреннего словаря 1
student100 = {'name': 'John', 'age': 24}

#определение внутреннего словаря 2
student101 = {}
student101['name'] = 'Mike'
student101['age'] = '22'

#назначение внутренних словарей 1 и 2 корневому словарю
dict1 = {}
dict1[100] = student100
dict1[101] = student101

#создание внутреннего словаря напрямую внутри корневого словаря
dict1[102] = {}
dict1[102]['name'] = 'Jim'
dict1[102]['age'] = '21'

print(dict1)
print(dict1.get(102))
```

Далее обсудим, как получить доступ к различным элементам вложенного словаря.

Доступ к элементам вложенного словаря

Для добавления значений и словарей внутри словаря можно использовать двойную индексацию. В качестве альтернативы можно использовать метод `get` объекта словаря. Тот же подход применим для доступа к различным элементам внутреннего словаря. Ниже приведен пример использования методов `get` и двойной индексации:

```
#dictionary3.py
dict1 = {100:{'name':'John', 'age':24},
         101:{'name':'Mike', 'age':22},
         102:{'name':'Jim', 'age':21} }
print(dict1.get(100))
print(dict1.get(100).get('name'))
print(dict1[101])
print(dict1[101]['age'])
```

Рассмотрим теперь, как удалить внутренний словарь или элементы из него.

Удаление из вложенного словаря

Для удаления словаря или элемента можно использовать общую функцию `del` или метод объекта словаря — `pop`. В следующем примере показаны оба этих варианта:

#dictionary4.py

```
dict1 = {100:{'name':'John', 'age':24},
         101:{'name':'Mike', 'age':22},
         102:{'name':'Jim', 'age':21} }

del (dict1[101]['age'])
print(dict1)
dict1[102].pop('age')
print(dict1)
```

Далее обсудим, как включения помогают обрабатывать данные из разных типов структур данных.

Использование включений

Включение (**Comprehension**) — это быстрый способ создания новых последовательностей (списков, наборов, словарей) из уже существующих. Python поддерживает четыре типа включений:

- Списковое включение.
- Словарное включение.
- Включение множеств.
- Генераторное включение.

Кратко рассмотрим каждый тип с примерами.

Списковое включение

Списковое включение (**List comprehension**) подразумевает создание динамического списка с помощью цикла и условного оператора, если это необходимо.

Примеры помогут лучше понять концепцию. В первом фрагменте кода (`list1.py`) мы создадим новый список из исходного, добавив `1` к каждому элементу исходного списка:

#list1.py

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
list2 = [x+1 for x in list1]
print(list2)
```

В этом случае новый список будет создан с помощью выражения `x+1`, где `x` — элемент исходного списка. Это эквивалентно следующему традиционному коду:

```
list2 = []
for x in list1:
    list2.append(x+1)
```

Используя списковое включение, можно заменить три строки кода одной.

Во втором примере (`list2.py`) мы создадим новый список из исходного списка чисел от 1 до 10, но будем включать только четные. Для этого можно просто добавить условие к предыдущему фрагменту кода:

#list2.py

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list2 = [x for x in list1 if x % 2 == 0]
print(list2)
```

Как видите, условие добавляется в конец выражения включения. Теперь попробуем создать словарь с помощью включения.

Словарное включение

Словарное включение (**Dictionary comprehension**) похоже на списковое и позволяет создавать словарь из другого словаря таким образом, что элементы исходного словаря выбираются или преобразовываются условно. Ниже приведен пример создания словаря из элементов существующего словаря. Элементы выбираются по условию (меньше или равны 200), а затем делятся на 2. Обратите внимание, значения также преобразуются обратно в целые числа как часть выражения включения:

#dictcomp1.py

```
dict1 = {'a': 100, 'b': 200, 'c': 300}
dict2 = {x:int(y/2) for (x, y) in dict1.items() if y <=200}
print(dict2)
```

Данный код эквивалентен следующему фрагменту из традиционного программирования:

```
Dict2 = {}
for x,y in dict1.items():
    if y <= 200:
        dict2[x] = int(y/2)
```

Как видите, включение значительно сокращает код.

Включение множеств

Множества также можно создавать, используя *включение множеств* (**Set comprehension**). Синтаксис похож на создание спискового включения, за исключением фигурных скобок вместо квадратных. Следующий фрагмент демонстрирует это:

#setcomp1.py

```
list1 = [1, 2, 6, 4, 5, 6, 7, 8, 9, 10, 8]
set1 = {x for x in list1 if x % 2 == 0}
print(set1)
```

Данный код эквивалентен следующему фрагменту из традиционного программирования:

```
Set1 = set()
for x in list1:
    if x % 2 == 0:
        set1.add(x)
```

Как и ожидалось, повторяющиеся записи в наборе не будут учитываться.

Мы рассмотрели доступные типы включений для разных структур данных, далее обсудим параметры фильтрации для них.

Введение в Pandas DataFrame

Pandas — библиотека Python с открытым исходным кодом, которая предоставляет инструменты для высокопроизводительной обработки данных и упрощает анализ информации. Обычно используется для изменения формы, сортировки, создания срезов, агрегирования и объединения данных.

Библиотека построена поверх библиотеки **NumPy** — еще одной библиотеки Python для работы с массивами. NumPy работает значительно быстрее, чем традиционные списки Python, поскольку данные хранятся в одном непрерывном месте памяти.

Pandas работает с тремя ключевыми структурами данных:

- `Series`: одномерный массив, который содержит массив данных и массив меток данных; массив меток называется `index` и может быть указан автоматически с помощью целых чисел от 0 до *n-1*, если он явно не задан пользователем.
- `DataFrame`: представление табличных данных, вроде электронной таблицы со списком столбцов; объект DataFrame помогает хранить и обрабатывать табличные данные в строках и столбцах; следует отметить, что DataFrame имеет индекс, как для столбцов, так и для строк.
- `Panel`: трехмерный контейнер данных.

DataFrame — это ключевая структура, которая используется для анализа данных. В оставшейся части этого подраздела мы будем широко использовать объект DataFrame в наших примерах. Прежде чем обсуждать его расширенные возможности, кратко рассмотрим основные операции, доступные для объектов DataFrame.

Операции с объектом DataFrame

Существует несколько способов создания объекта DataFrame, например, из словаря, CSV-файла, листа Excel или массива NumPy. Один из самых простых — использовать данные из словаря в качестве ввода. Рассмотрим на примере, как можно создать объект DataFrame на основе еженедельных данных о погоде, хранящихся в словаре:

```
#pandas1.py
import pandas as pd

weekly_data = {'day':['Monday','Tuesday', 'Wednesday', \
'Thursday','Friday', 'Saturday', 'Sunday'],
               'temperature':[40, 33, 42, 31, 41, 40, 30],
               'condition':['Sunny','Cloudy','Sunny','Rain'
               ,'Sunny','Cloudy','Rain']
        }

df = pd.DataFrame(weekly_data)
print(df)
```

На консоли отобразится содержимое DataFrame (рис. 6.2):

	day	temp	condition
0	Monday	40	Sunny
1	Tuesday	33	Cloudy
2	Wednesday	42	Sunny
3	Thursday	31	Rain
4	Friday	41	Sunny
5	Saturday	40	Cloudy
6	Sunday	30	Rain

Рис. 6.2. Содержимое DataFrame

Библиотека Pandas богата методами и атрибутами, однако здесь мы рассмотрим только самые часто используемые:

- `index`: предоставляет список индексов (или меток) объекта DataFrame;
- `columns`: предоставляет список столбцов в объекте DataFrame;

- `size`: возвращает размер объекта DataFrame в виде количества строк, умноженного на количество столбцов;
- `shape`: предоставляет кортеж, представляющий измерение объекта DataFrame;
- `axes`: возвращает список, который представляет оси объекта DataFrame; простыми словами, включает в себя строки и столбцы;
- `describe`: генерирует статистические данные; например, количество, среднее значение, стандартное отклонение, минимальное и максимальное значения;
- `head`: возвращает *n* (по умолчанию 5) строк из объекта DataFrame, аналогично команде `head` для файлов;
- `tail`: возвращает последние *n* (по умолчанию 5) строк из объекта DataFrame;
- `drop_duplicates`: удаляет дубликаты строк на основе всех столбцов в DataFrame;
- `dropna`: передавая соответствующие аргументы этому методу, можно удалить отсутствующие значения (например, строки и столбцы); можно также указать, строки и столбцы будут удаляться на основе одного пропущенного значения или при отсутствии всех значений в строке или столбце;
- `sort_values`: можно использовать для сортировки строк на основе одного или нескольких столбцов.

Дальше рассмотрим базовые операции с объектами DataFrame.

Настройка пользовательского индекса

Метки столбцов (индексы) обычно добавляются в соответствии с данными, предоставленными словарем или любым другим потоком данных. Можно изменить индекс DataFrame следующими способами:

1. Задать один из столбцов в качестве индекса (например, `day` из предыдущего примера), используя следующий простой оператор:

```
df_new = df.set_index('day')
```

2. DataFrame начнет использовать столбец `day` в качестве индекса следующим образом (рис. 6.3):

	temp	condition
day		
Monday	40	Sunny
Tuesday	33	Cloudy
Wednesday	42	Sunny
Thursday	31	Rain
Friday	41	Sunny
Saturday	40	Cloudy
Sunday	30	Rain

Рис. 6.3. Содержимое DataFrame после применения столбца `day` в качестве индекса

3. Задать индекс вручную, передав его через список, как в следующем примере:

```
#pandas2.py
weekly_data = <данные как в предыдущем примере>
df = pd.DataFrame(weekly_data)
df.index = ['MON','TUE','WED','THU','FRI','SAT','SUN']
print(df)
```

4. В этом случае DataFrame начнет использовать индекс, предоставленный через список. Содержимое DataFrame изменится следующим образом (рис. 6.4):

	day	temp	condition
MON	Monday	40	Sunny
TUE	Tuesday	33	Cloudy
WED	Wednesday	42	Sunny
THU	Thursday	31	Rain
FRI	Friday	41	Sunny
SAT	Saturday	40	Cloudy
SUN	Sunday	30	Rain

Рис. 6.4. Содержимое DataFrame после создания столбца индекса вручную

Далее узнаем, как перемещаться по DataFrame.

Навигация внутри DataFrame

Существует несколько десятков способов получить строку данных или определенное расположение из объекта. Распространенными являются методы `loc` и `iloc`. Рассмотрим несколько способов навигации по DataFrame, используя набор данных из предыдущих примеров:

```
#pandas3.py
import pandas as pd
weekly_data = <как в примере pandas1.py>
df = pd.DataFrame(weekly_data)
df.index = ['MON', 'TUE','WED','THU','FRI','SAT','SUN']
```

Существует несколько способов выбрать строку или расположение в объекте:

- Мы можем выбрать одну или несколько строк, используя метки индекса с методом `loc`; метка предоставляется как отдельный элемент или список; ниже приведен фрагмент с двумя примерами, как можно выбрать одну или несколько строк:

```
print(df.loc['TUE'])
print(df.loc[['TUE','WED']])
```

- Мы можем выбрать значение из расположения в объекте DataFrame по метке строки и метке столбца:

```
print(df.loc['FRI','temp'])
```

- Мы можем выбрать строку по значению индекса без указания меток:

```
#предоставляем строку с индексом 2
print(df.iloc[2])
```

- Мы можем выбрать значение из расположения, используя индексы строки и столбца, путем обработки объекта DataFrame как двумерного массива; в следующем фрагменте кода мы получаем значение из расположения с индексом строки, равным 2, и индексом столбца, равным 2:

```
print(df.iloc[2,2])
```

Далее обсудим способы, как можно добавить строку или столбец в объект DataFrame.

Добавление строки или столбца в DataFrame

Самый простой способ добавить строку в DataFrame — присвоить список значений расположению индекса или его метке. Например, мы можем добавить новую строку с меткой `TST` для предыдущего примера (`pandas3.py`) с помощью следующей инструкции:

```
df.loc['TST'] = ['Test day 1', 50, 'NA']
```

Важно отметить, что если такая метка строки уже существует в объекте, этот же код может обновить новые значения в строке.

Если мы используем не метку индекса, а индекс по умолчанию, можно использовать номер индекса для обновления существующей строки или добавления новой с помощью следующего кода:

```
df.loc[8] = ['Test day 2', 40, 'NA']
```

Для наглядности покажем полный пример кода:

```
#pandas4.py
import pandas as pd
weekly_data = <как в примере pandas1.py>
df = pd.DataFrame(weekly_data)
df.index = ['MON', 'TUE','WED','THU','FRI','SAT','SUN']

df.loc['TST1'] = ['Test day 1', 50, 'NA']
df.loc[7] = ['Test day 2', 40, 'NA']
print(df)
```

В библиотеке Pandas доступны разные способы добавления нового столбца в DataFrame. Рассмотрим три из них:

- **Добавление списка значений после метки столбца**: при таком подходе новый столбец добавляется после существующих столбцов; если указать существующую метку столбца, это позволит заменить или обновить значения в нем.

- **Метод insert**: принимает метку и список значений в качестве аргументов; это особенно удобно, когда мы хотим вставить столбец в любое место; обратите внимание, метод не позволяет вставить столбец, если он уже существует с такой же меткой; это означает, что метод нельзя использовать для обновления.
- **Метод assign**: позволяет добавлять несколько столбцов одновременно; если указать существующую метку, значения в столбце будут заменены или обновлены.

В следующем примере кода мы используем все три подхода для вставки нового столбца в объект DataFrame:

$pandas5.py

```
import pandas as pd

weekly_data = <как в примере pandas1.py >
df = pd.DataFrame(weekly_data)

#Добавляем столбец и обновляем его
df['Humidity1'] = [60, 70, 65,62,56,25,'']
df['Humidity1'] = [60, 70, 65,62,56,251,'']

#Вставляем столбец с индексом 2, используя insert
df.insert(2, "Humidity2",[60, 70, 65,62,56,25,''])

#Добавляем 2 столбца методом assign
df1 = df.assign(Humidity3 = [60, 70, 65,62,56,25,''],
  Humidity4 = [60, 70, 65,62,56,25,''])
print(df1)
```

Далее рассмотрим, как удалять строки и столбцы из объекта DataFrame.

Удаление индекса, строки или столбца из DataFrame

Удалить индекс достаточно просто. Это можно сделать методом `reset_index`. Однако он добавляет индексы по умолчанию и сохраняет столбец с пользовательскими индексами в качестве столбца данных. Для полного удаления столбца вместе с методом необходимо использовать аргумент `drop`. Продемонстрируем на примере:

$pandas6.py

```
import pandas as pd

weekly_data = <как в примере pandas1.py >
df = pd.DataFrame(weekly_data)

df.index = ['MON', 'TUE','WED','THU','FRI','SAT','SAT']
print(df)
print(df.reset_index(drop=True))
```

Для удаления повторяющейся строки из DataFrame можно использовать метод drop_duplicate. Для удаления определенной строки или столбца можно использовать метод drop. В следующем примере удалим все строки с метками SAT и SUN, а также столбцы с меткой condition:

#pandas7.py

```
import pandas as pd

weekly_data = <как в примере pandas1.py>
df = pd.DataFrame(weekly_data)
df.index = ['MON', 'TUE','WED','THU','FRI','SAT','SUN']
print(df)

df1= df.drop(index=['SUN','SAT'])
df2= df1.drop(columns=['condition'])

print(df2)
```

Теперь рассмотрим, как переименовать индекс или столбец.

Переименование индексов и столбцов в DataFrame

Для переименования меток индекса или столбца используется метод rename. Пример кода выглядит следующим образом:

#pandas8.py

```
import pandas as pd

weekly_data = <как в примере pandas1.py>
df = pd.DataFrame(weekly_data)
df.index = ['MON', 'TUE','WED','THU','FRI','SAT','SUN']

df1=df.rename(index={'SUN': 'SU', 'SAT': 'SA'})
df2=df1.rename(columns={'condition':'cond'})

print(df2)
```

Обратите внимание, текущая и новая метки для индекса и столбца предоставляются в виде словаря. Далее изучим некоторые дополнительные приемы использования объектов DataFrame.

Сложные случаи использования DataFrame

На данном этапе мы рассмотрели основные операции, которые можно выполнять с объектом DataFrame. Теперь перейдем к более сложным операциям для вычисления и преобразования данных.

Замена данных

Часто возникает необходимость заменить числовые или строковые данные другим набором значений. Библиотека Pandas имеет много инструментов для этого. Наиболее популярным является метод `at`. Он обеспечивает простой способ доступа или обновления данных в любой ячейке DataFrame. Для массовой замены значений доступен метод `replace`, который можно использовать разными способами. Например, для замены одного числа на другое или одной строки на другую, а также для замены целого фрагмента, заданного регулярным выражением. Кроме того, можно использовать метод для замены любых записей, предоставленных через список или словарь. В следующем примере (`pandastrick1.py`) рассмотрим эти варианты, используя объект DataFrame из предыдущих примеров:

```
#pandastrick1.py
import pandas as pd
weekly_data = <как в примере pandas1.py>
df = pd.DataFrame(weekly_data)
```

Последовательно применим несколько операций замены в этом примере:

- Заменим все вхождения числового значения `40` на `39`, используя следующий оператор:

```
df.replace(40,39, inplace=True)
```

- Заменим все вхождения строки `Sunny` на `Sun` с помощью следующей строки:

```
df.replace("Sunny","Sun",inplace=True)
```

- Заменим все вхождения строки на основе регулярного выражения (а именно, `Cloudy` на `Cloud`) с помощью следующего фрагмента:

```
df.replace(to_replace="^Cl.*",value="Cloud",
inplace=True,regex=True)
#или также можно применить к определенному столбцу
df["condition"].replace(to_replace="^Cl.*",value="Cloud",
inplace=True,regex=True)
```

- Обратите внимание, что использование меток `to_replace` и `value` опционально.
- Заменим все вхождения нескольких строк, представленных списком, другим списком строк:

```
df.replace(["Monday","Tuesday"],["Mon","Tue"],
inplace=True)
```

- Здесь мы заменили `Monday` и `Tuesday` на `Mon` и `Tue`.
- Заменим все вхождения нескольких строк, используя пары *«ключ-значение»* в словаре:

```
df.replace({"Wednesday":"Wed","Thursday":"Thu"},
inplace=True)
```

- В этом случае ключи словаря (`Wednesday` и `Thursday`) будут заменены соответствующими значениями (`Wed` и `Thu`).
- Заменим все вхождения строки для определенного столбца, используя несколько словарей; это можно сделать, используя имя столбца в качестве ключа и следующий оператор:

```
df.replace({"day":"Friday"}, {"day":"Fri"}, inplace=True)
```

- В этом сценарии первый словарь указывает на имя столбца и заменяемое значение; второй словарь указывает на то же имя столбца, но с новым значением, которое заменит исходное; в нашем случае мы заменим все вхождения `Friday` в столбце `day` на `Fri`.
- Заменим все вхождения нескольких строк с помощью вложенного словаря:

```
df.replace({"day":{"Saturday":"Sat", "Sunday":"Sun"},
            "condition":{"Rainy":"Rain"}}, inplace=True)
```

- Внешний словарь (с ключами `day` и `condition`) используется для определения столбцов для этой операции; внутренний словарь хранит заменяемые данные вместе с замещающими значениями; с помощью этого подхода мы заменили `Saturday` и `Sunday` на `Sat` и `Sun` внутри столбца `day`, а также `Rainy` на `Rain` внутри столбца `condition`.

Полный код с этими операциями доступен в файле `pandastrick1.py` в репозитории к этой главе. Обратите внимание, что можно запустить операцию замены во всем объекте DataFrame или только в определенном столбце или строке.

> ***Важное примечание***
>
> Аргумент `inplace=True` задействован во всех вызовах метода `replace`. Он используется для настройки вывода метода `replace` внутри того же объекта DataFrame. Параметр по умолчанию вернет новый объект, не меняя исходный. Для удобства этот аргумент доступен со многими методами DataFrame.

Применение функций к столбцу или строке объекта DataFrame

Иногда нужно очистить, скорректировать или преобразовать данные перед их анализом. Простой способ применить функцию к DataFrame — использовать методы `apply`, `applymap` или `map`. Метод `apply` применим к столбцам или строкам, а метод `applymap` работает поэлементно для всего DataFrame. Метод `map` также работает поэлементно, но для одного ряда. Рассмотрим пару примеров кода для демонстрации работы методов.

Часто импортированные в DataFrame данные нуждаются в очистке. Например, они могут иметь пробелы в конце или в начале, символы новой строки и другие нежелательные элементы. Их можно легко удалить, используя метод `map` и лямбда-функцию в рядах столбцов. Лямбда-функция применяется к каждому элементу

столбца. В нашем примере сначала удалим конечный пробел, точку и запятую. Затем удалим начальный пробел, подчеркивание и тире в столбце `condition`.

После очистки данных в столбце `condition` мы создадим столбец `temp_F` из значений столбца `temp`, а затем преобразуем эти значения из градусов Цельсия в градусы Фаренгейта. Обратите внимание, мы будем использовать другую лямбда-функцию для этого преобразования и метод `apply`. Когда мы получим результат от метода `apply`, сохраним его внутри новой метки столбца (`temp_F`) для создания нового столбца. Пример кода следующий:

```
#pandastrick2.py
import pandas as pd

weekly_data = {'day':['Monday','Tuesday', 'Wednesday',
                'Thursday','Friday', 'Saturday', 'Sunday'],
                'temp':[40, 33, 42, 31, 41, 40, 30],
                'condition':['Sunny,','_Cloudy ',
                'Sunny','Rainy','--Sunny.','Cloudy.','Rainy']
        }
df = pd.DataFrame(weekly_data)
print(df)
df["condition"] = df["condition"].map(
                lambda x: x.lstrip('_- ').rstrip(',. '))
df["temp_F"] = df["temp"].apply(lambda x: 9/5*x+32 )
print(df)
```

Обратите внимание, в этом коде мы предоставили те же входные данные, что и в прошлых примерах, за исключением начальных и конечных символов в столбце `condition`.

Запрос строк в объекте DataFrame

Запросить строки на основе значений в определенном столбце можно одним из распространенных подходов — применив фильтр с использованием логического «И/ИЛИ». Однако это быстро становится запутанным для простых задач вроде поиска строки со значением в определенном диапазоне. Библиотека pandas предлагает более аккуратный способ: метод `between`, который похож на ключевое слово `between` в SQL.

В примере ниже используется тот же объект `weekly_data`, что и раньше. Сначала мы покажем использование традиционного фильтра, а затем продемонстрируем использование метода `between` для запроса строк со значениями температуры от 30 до 40 включительно:

```
#pandastrick3.py
import pandas as pd

weekly_data = <как в примере pandas1.py>
df = pd.DataFrame(weekly_data)
```

```
print(df[(df.temp >= 30) & (df.temp<=40)])
print(df[df.temp.between(30,40)])
```

В обоих случаях мы получим одинаковый консольный вывод. Однако использовать метод `between` удобнее, чем фильтры с условиями.

Запрос строк на основе текстовых данных также хорошо поддерживается библиотекой Pandas. Этого можно добиться методом доступа `str` для столбцов строкового типа в объекте DataFrame. А именно, если мы хотим найти строки в объекте `weekly_data` на основе состояния дня (например, `Rainy` или `Sunny`), мы можем написать традиционный фильтр или применить `str` к столбцу вместе с методом `contains`. Следующий пример демонстрирует оба варианта получения строк со значениями `Rainy` или `Sunny` в столбце `condition`:

```
#pandastrick4.py
import pandas as pd

weekly_data = <как в примере pandas1.py>
df = pd.DataFrame(weekly_data)

print(df[(df.condition=='Rainy') | (df.condition=='Sunny')])
print(df[df['condition'].str.contains('Rainy|Sunny')])
```

Оба подхода дадут одинаковый результат.

Получение статистики по данным объекта DataFrame

Получить статистические данные, например, центральную тенденцию, стандартное отклонение и форму можно, используя метод `describe`. Вывод метода для числовых столбцов включает следующее:

- `count` (количество);
- `mean` (среднее значение);
- `standard deviation` (стандартное отклонение);
- `min` (минимальное значение);
- `max` (максимальное значение);
- 25^{th} `percentile` (25-й перцентиль), 50^{th} `percentile` (50-й перцентиль), 75^{th} `percentile` (75-й перцентиль).

Разделение перцентилей по умолчанию можно изменить с помощью аргумента `percentiles`.

Если метод `describe` используется для нечисловых данных вроде строк, мы можем получить от них `count`, `unique`, `top` и `freq`. Где `top` — наиболее распространенное значение, а `freq` — частота наиболее распространенного значения. По умолчанию только числовые столбцы оцениваются методом `describe`, если мы не предоставим аргумент `include` с соответствующим значением.

В примере ниже мы вычислим следующую статистику для объекта `weekly_date`:

- Использование метода `describe` с аргументом `include` или без него.
- Использование аргумента `percentiles` с методом `describe`.
- Использование метода `groupby` для группировки данных по столбцам, а затем применение метода `describe` в завершение.

Полный пример получится следующим:

```
#pandastrick5.py
import pandas as pd
import numpy as np
pd.set_option('display.max_columns', None)

weekly_data = <как в примере pandas1.py>
df = pd.DataFrame(weekly_data)

print(df.describe())
print(df.describe(include="all"))
print(df.describe(percentiles=np.arange(0, 1, 0.1)))
print(df.groupby('condition').describe(percentiles=np.arange(0, 1, 0.1)))
```

Обратите внимание, что мы изменили параметры `max_columns` в самом начале для отображения всех столбцов, которые ожидаются в выводе. В противном случае некоторые столбцы будут усечены для консольного вывода метода `groupby`.

На этом можно завершить изучение расширенных приемов работы с объектом DataFrame. С их помощью любой может начать использовать библиотеку Pandas для анализа данных. Для изучения дополнительных приемов можно обратиться к официальной документации по библиотеке.

Заключение

В этой главе мы рассмотрели некоторые продвинутые приемы и хитрости для написания эффективных и лаконичных программ на Python. Мы начали с расширенных функций, таких, как `map`, `reduce` и `filter`. Затем изучили несколько концепций: внутренние функции, лямбда-функции и декораторы. Также обсудили использование структур данных, включая вложенные словари и включения. В конце мы рассмотрели основные операции с объектом DataFrame и коснулись некоторых реальных случаев их использования.

Эта глава, в основном, сосредоточена на практических знаниях и опыте использования передовых концепций Python. Это важно всем, кто хочет разрабатывать приложения на Python, особенно связанные с анализом данных. Примеры кода, представленные в главе, наглядно демонстрируют работу расширенных приемов, доступных для функций, структур данных и библиотеки Pandas.

Следующая глава посвящена многопроцессорной обработке и многопоточности в Python.

Вопросы

1. Какие из функций `map`, `filter` и `reduce` являются встроенными в Python?
2. Какие декораторы являются стандартными?
3. Что предпочтительнее выбрать для большого набора данных — списковое включение или генераторное включение?
4. Что такое DataFrame в контексте библиотеки pandas?
5. Зачем нужен аргумент `inplace` в методах библиотеки pandas?

Дополнительные ресурсы

- «*Mastering Python Design Patterns*», автор: Сакис Касампалис (Sakis Kasampalis).
- «*Python и анализ данных*» (*Python for Data Analysis*), автор: Уэс Маккини (Wes McKinney).
- «*Hands-On Data Analysis with Pandas*», второе издание, автор: Стефани Молин (Stefanie Molin).
- Официальная документация Pandas: *https://pandas. pydata.org/docs/*.

Ответы

1. Встроенными являются функции `map` и `filter`.
2. Декораторы называются стандартными, когда не имеют аргументов.
3. В этом случае предпочтительнее генераторное включение. Оно эффективно использует память, поскольку значения генерируются по-одному.
4. DataFrame — это представление табличных данных, которое широко используется для анализа данных с помощью библиотеки pandas.
5. Когда для аргумента `inplace` в методах pandas установлено значение `True`, результат операции сохраняется в том же объекте DataFrame, к которому применяется операция.

Раздел 3

Масштабирование за пределы одного потока

Этот раздел книги посвящен программированию масштабируемых приложений. Обычно интерпретатор Python использует один *поток*, работающий в одном *процессе*. В этой части книги мы обсудим, как масштабировать приложение за пределы одного потока. Для этого мы затронем такие понятия, как многопоточность, многопроцессорная обработка и асинхронное программирование на одной машине. Затем исследуем, как можно использовать несколько машин и выполнять приложения в кластерах с помощью Apache Spark. В конце рассмотрим среды облачных вычислений, которые позволят сосредоточиться на приложении и оставить управление инфраструктурой поставщикам облачных услуг.

В этот раздел входят следующие главы:

- «Глава 7: Многопроцессорная обработка, многопоточность и асинхронное программирование».
- «Глава 8: Масштабирование приложений Python с помощью кластеров».
- «Глава 9: Программирование на Python для облака».

7

Многопроцессорная обработка, многопоточность и асинхронное программирование

Мы можем написать эффективный код и оптимизировать его для быстрого выполнения, но ресурсы, доступные процессам, всегда ограничены. Для ускорения работы некоторые задачи можно запускать параллельно на одном или нескольких компьютерах. Эта глава посвящена параллельной обработке для приложений Python, выполняющихся на одной машине. О выполнении на нескольких машинах мы поговорим в следующей главе. А сейчас сосредоточимся на встроенной поддержке для реализации параллельных вычислений. Начнем с многопоточности, а затем перейдем к многопроцессорности. После узнаем, как проектировать быстрые системы с помощью асинхронного программирования. Для каждого из подходов мы рассмотрим полноценный практический пример реализации конкурентного приложения для загрузки файлов с Google Диска.

Темы этой главы:

- Многопоточность в Python и ее ограничения.
- Многопроцессорная обработка.
- Асинхронное программирование для адаптивных систем.

Вы узнаете о разных вариантах создания многопоточных и многопроцессорных приложений с использованием встроенных библиотек Python. Эти знания помогут вам создавать не только более эффективные приложения, но и масштабные проекты для большого количества пользователей.

Технические требования

Для этой главы понадобится:

- Python 3.7 или более поздней версии.
- Аккаунт Google Диска.
- Ключ API для аккаунта Google Диска.

Примеры кода к этой главе можно найти по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter07.

Многопоточность в Python и ее ограничения

Поток (**Thread**) — это базовая единица выполнения в операционной системе, состоящая из счетчика команд, стека и набора регистров. Процессы приложения могут быть построены с использованием нескольких потоков, которые могут работать одновременно и совместно использовать одну память.

При многопоточности все потоки процесса используют общий код и другие ресурсы вроде данных и системных файлов. Для каждого потока вся связанная с ним информация хранится как структура данных внутри ядра операционной системы и называется *Блок Управления Потоком* (**Thread Control Block, TCB**). TCB состоит из нескольких основных компонентов:

- *Счетчик команд* (**Program Counter, PC**): отслеживает ход выполнения программы.
- *Системные регистры* (**System Registers, REG**): хранят переменные данные.
- *Стек* (**Stack**): массив регистров, который управляет историей выполнения.

На рис. 7.1 приведена схема с тремя потоками. Каждый из них имеет собственный счетчик команд, стек и системный регистр, но использует общий код и прочие ресурсы с другими потоками.

Блок управления также содержит идентификатор потока, состояние потока (выполняется, ожидает или остановлен) и указатель на процесс, которому он принадлежит. *Многопоточность* — это концепция операционной системы, которая предлагается системным ядром. ОС обеспечивает параллельное выполнение нескольких потоков в контексте одного процесса, позволяя потокам совместно использовать память. Это означает, что полный контроль над управлением потоков имеет ОС, а не приложение. Этот момент важно подчеркнуть для последующего обсуждения различных вариантов параллелизма.

При выполнении потоков на компьютере с одним *центральным процессором* (**ЦП**) операционная система переключает его с одного потока на другой, поэтому кажется, что потоки выполняются параллельно. Дает ли многопоточность с одним ЦП преимущества, зависит от характера приложения. Если оно использует только ло-

кальную память, преимуществ, скорее всего, нет. Более того, производительность может упасть из-за издержек, связанных с переключением потоков. Если приложение зависит от других ресурсов, выполнение может ускориться благодаря более эффективному использованию ЦП. Пока один поток ждет ресурс, другой может использовать процессор.

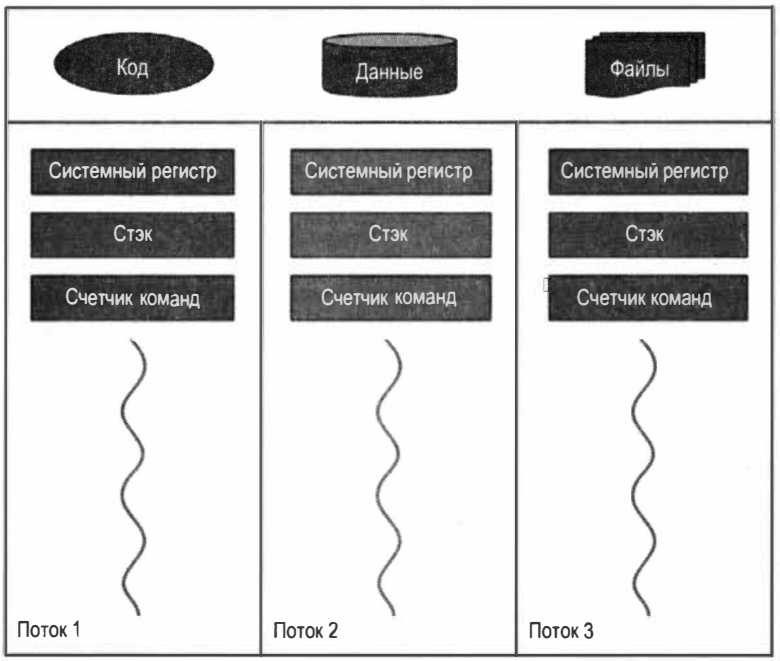


Рис. 7.1. Несколько потоков в процессе

При наличии нескольких процессоров или ядер ЦП потоки можно выполнять параллельно. Рассмотрим ограничения многопоточного программирования.

Слепое пятно Python

С точки зрения программирования многопоточность — это подход, при котором мы выполняем разные части приложения одновременно. Python использует множество потоков ядра для выполнения пользовательских потоков. Реализация Python (CPython) позволяет потокам обращаться к объектам через *глобальную блокировку интерпретатора* (**Global Interpreter Lock, GIL**). В двух словах, GIL — это мьютекс (**mutex**, механизм взаимного исключения), который позволяет использовать интерпретатор только одному потоку за раз и блокирует остальные потоки. Это необходимо для защиты счетчика ссылок от сборки мусора. Без такой защиты счетчик может быть поврежден, если он обновляется несколькими потоками одновременно. Причиной такого ограничения является защита внутренних структур данных интерпретатора и стороннего кода C, который не является потокобезопасным.

Важное примечание

Это ограничение глобальной блокировки интерпретатора отсутствует в других реализациях Python — Jython и IronPython.

Создается впечатление, что написание многопоточных программ на Python не имеет преимуществ, но это не так. Мы по-прежнему можем писать код, который будет выполняться одновременно или параллельно, и мы можем увидеть это на примере. Многопоточность полезна в следующих случаях:

1. **Задачи ввода-вывода**: при наличии нескольких операций ввода-вывода можно повысить производительность, используя несколько потоков. Пока один поток ждет ответа от ресурса, он освобождает GIL и позволяет работать другим потокам. Исходный поток возобновится после ответа от ресурса ввода-вывода.
2. **Адаптивные GUI-приложения**: в них требуется отображать ход выполнения задач, запущенных в фоновом режиме (например, загрузка файла), а также давать пользователю взаимодействовать с другими компонентами интерфейса. Это возможно благодаря отдельным потокам для действий, инициированных пользователем в графическом интерфейсе.
3. **Многопользовательские приложения**: здесь также необходима многопоточность, например, для веб-сервера или файлового сервера. В таких приложениях при поступлении нового запроса в основном потоке создается новый поток, который будет обслуживать этот запрос, пока основной поток ожидает новых запросов.

Прежде чем рассмотреть концепцию на примере, познакомимся с ключевыми компонентами многопоточного программирования на Python.

Ключевые компоненты многопоточного программирования на Python

Многопоточность в Python позволяет выполнять разные компоненты программы одновременно. Для создания нескольких потоков приложения используется модуль `threading`, основные компоненты которого рассмотрим далее.

Модуль threading

Модуль `threading` является стандартным и предоставляет простые методы для создания нескольких потоков программы. Внутри он использует низкоуровневый модуль `_thread`, который часто использовался для реализации многопоточности в более ранних версиях Python.

Для нового потока создадим объект класса `Thread`, который может принимать имя функции в качестве атрибута `target` и аргументы в качестве атрибута `args` для передачи в функцию. Потоку можно при создании задать имя с помощью аргумента `name` в конструкторе.

После создания объекта нужно запустить поток методом `start`. Реализовать, что основная программа или поток будут ждать завершения созданных потоковых объектов, можно с помощью метода `join`. Этот метод гарантирует, что главный поток (вызывающий поток) ожидает, пока поток, для которого вызван метод `join`, не завершит выполнение.

Для демонстрации этих процессов создадим простую программу с тремя потоками. Полный пример кода приведен ниже:

```
#threadl.py создание простых потоков с помощью функции

from threading import current_thread, Thread as Thread
from time import sleep

def print_hello():
    sleep(2)
    print("{}: Hello".format(current_thread().name))

def print_message(msg):
    sleep(1)
    print("{}: {}".format(current_thread().name, msg))

#создаем потоки
t1 = Thread(target=print_hello, name="Th 1")
t2 = Thread(target=print_hello, name="Th 2")
t3 = Thread(target=print_message, args=["Good morning"],name="Th 3")

#запускаем потоки
t1.start()
t2.start()
t3.start()

#ждем, пока все завершатся
t1.join()
t2.join()
t3.join()
```

В этой программе мы сделали следующее:

- Создали две простые функции, `print_hello` и `print_message`, которые будут использоваться потоками; задействовали функцию `sleep` из модуля `time` в обеих функциях с целью убедиться, что они завершают выполнение в разное время.
- Создали три объекта `Thread`, два из которых будут выполнять одну функцию (`print_hello`) для демонстрации совместного использования кода потоками, а третий будет использовать вторую функцию (`print_message`), которая также принимает один аргумент.

- Запустили все три потока один за другим, используя метод `start`.
- Дождались завершения каждого потока с помощью метода `join`.

Объекты `Thread` можно хранить списком для упрощения операций `start` и `join`, используя цикл `for`. Вывод консоли будет следующим:

```
Th 3: Good morning
Th 2: Hello
Th 1: Hello
```

Потоки 1 и 2 имеют большее время ожидания, чем поток 3, поэтому он всегда будет завершаться первым. А потоки 1 и 2 могут завершаться в любом порядке, в зависимости от того кто первый получит процессор.

> ***ВАЖНОЕ ПРИМЕЧАНИЕ***
>
> По умолчанию метод `join` блокирует вызывающий поток на неопределенное время. Но можно указать время ожидания (в секундах) в качестве аргумента для него. Это приведет к блокировке лишь на время ожидания.

Рассмотрим еще несколько понятий, прежде чем перейти к более сложному примеру.

Потоки-демоны

В приложении главная программа неявным образом ждет завершения всех остальных потоков. Но иногда необходимо запустить некоторые из них в фоновом режиме, не блокируя основную программу. Такие потоки называются *потоками-демонами* (**Daemon Thread**). Они остаются активными, пока выполняется главная программа (с обычными потоками). А когда основные потоки завершатся, нормальной практикой считается завершить демоны тоже. Потоки-демоны часто используют в ситуациях, когда не будет ошибкой, если поток внезапно завершится посреди выполнения без потери или повреждения данных.

Поток может быть объявлен демоном, используя один из следующих подходов:

- Передать атрибут `daemon` со значением `True` конструктору (`daemon = True`).
- Задать для атрибута `daemon` значение `True` в экземпляре потока (`thread.daemon = True`).

Если поток установлен как демон, можно запустить его и забыть. Он завершится автоматически после окончания программы, которая его вызвала.

В следующем фрагменте кода приводится использование демона и обычных потоков:

#thread2.py создание обычных потоков и демонов

```
from threading import current_thread, Thread as Thread
from time import sleep

def daeom_func():
    #print(threading.current_thread().isDaemon())
```

```
    sleep(3)
    print("{}: Hello from daemon".format(current_thread().name))

def nondaeom_func():
    #print(threading.current_thread().isDaemon())
    sleep(1)
    print("{}: Hello from non-daemon".format(current_thread().name))

#создаем потоки
t1 = Thread(target=daeom_func, name="Daemon Thread", daemon=True)
t2 = Thread(target=nondaeom_func, name="Non-Daemon Thread")

#запускаем потоки
t1.start()
t2.start()

print("Exiting the main program")
```

В примере мы создали один поток-демон и один обычный поток. Поток-демон (daeom_func) выполняет функцию со временем ожидания 3 секунды, а обычный поток выполняет функцию (nondaeom_func) со временем ожидания 1 секунда. Время ожидания устанавливается таким образом, что обычный поток завершит свое выполнение первым. Консольный вывод будет следующим:

```
Exiting the main program
Non-Daemon Thread: Hello from non-daemon
```

Поскольку мы не использовали метод join ни в одном потоке, сначала завершится основной поток, а затем немного позже завершится обычный поток, выводя print-сообщение. От потока-демона никаких сообщений не будет, так как он завершится сразу после обычного потока. Если изменить время ожидания функции nondaeom_func на 5 секунд, получим следующие выходные данные:

```
Exiting the main program
Daemon Thread: Hello from daemon
Non-Daemon Thread: Hello from non-daemon
```

Задерживая выполнение обычного потока, мы гарантируем, что поток-демон завершит свое выполнение и не будет внезапно прерван.

> ***Важное примечание***
>
> Если применить метод join к демону, главному потоку придется ждать его завершения.

Теперь рассмотрим, как синхронизировать потоки в Python.

Синхронизация потоков

Синхронизация потоков — это механизм, который не позволяет двум или более потокам выполнять один блок кода одновременно. Блок кода, который обращается к общим данным или ресурсам, называется *критической секцией* (**Critical Section**). Наглядная схема показана ниже (рис. 7.2):

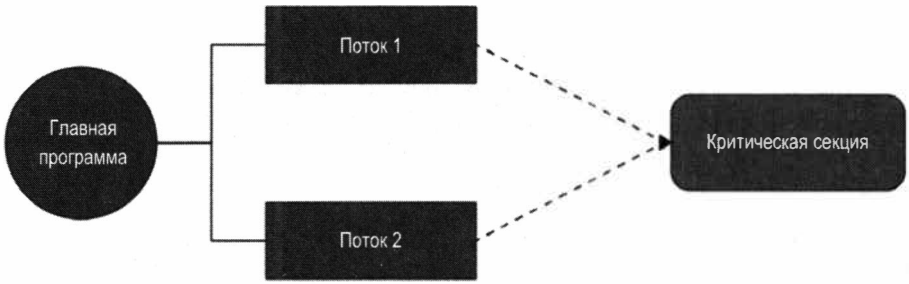


Рис. 7.2. Два потока обращаются к критической секции программы

Когда несколько потоков, обращающихся к критической секции, одновременно пытаются получить доступ к данным или изменить их, результаты могут быть непредсказуемыми. Такая ситуация называется *состоянием гонки* (**Race condition**).

Для демонстрации реализуем простую программу с двумя потоками, где каждый из них выполняет операцию инкремента общей переменной миллион раз. Большое число выбрано специально для наблюдения за состоянием гонки. Его также можно наблюдать, используя меньшее значение переменной на медленном ЦП. Оба потока будут использовать одну функцию (`inc`) в качестве целевой. Код для доступа к общей переменной и увеличения ее на 1 находится в критической секции, и оба потока обращаются к ней без какой-либо защиты:

#thread3a.py без синхронизации потоков

```
from threading import Thread as Thread

def inc():
    global x
    for _ in range(1000000):
        x+=1

#глобальная переменная
x = 0

#создаем потоки
t1 = Thread(target=inc, name="Th 1")
t2 = Thread(target=inc, name="Th 2")
```

```
#запускаем потоки
t1.start()
t2.start()

#ожидаем завершения потоков
t1.join()
t2.join()
print("final value of x :", x)
```

Ожидаемый вывод должен иметь значение 2 000 000, но этого не происходит. При каждом выполнении программы мы получаем разное значение `x`, которое намного меньше 2 000 000. Причина кроется в состоянии гонки между потоками. Рассмотрим сценарий, при котором потоки `Th 1` и `Th 2` выполняют критическую секцию (`x+=1`) одновременно. Оба потока запрашивают текущее значение `x`. Предположим, оно равно 100. Потоки одновременно считают значение как 100 и увеличат его до 101. В память будет записано новое значение 101. Происходит однократное увеличение, хотя оба потока должны увеличивать значение переменной независимо. Конечным значением `x` должно быть 102.

Достичь этого можно с помощью синхронизации потоков, которая использует класс `Lock` из модуля `threading`. *Замóк* (**Lock**) для блокировки потоков реализуется с помощью объекта семафора, предоставленного операционной системой. *Семафор* (**Semaphore**) — это объект синхронизации на уровне ОС для управления доступом к ресурсам и данным для нескольких процессоров и потоков. Класс `Lock` предоставляет два метода, `acquire` и `release`:

1. Метод `acquire` используется для получения замка, который имеет два состояния — **заблокирован** (по умолчанию) или **разблокирован.** Когда замок заблокирован, выполнение запрашивающего потока приостанавливается, пока замок не будет освобожден текущим потоком. Как только замок освобождается, он передается запрашивающему потоку. При запросе незаблокированного замка выполнение потока не блокируется. Если замок доступен (имеет состояние `unlocked`), запрашивающий поток получает его (состояние становится `locked`). В противном случае запрашивающий поток получает `False` в качестве ответа на запрос замка.
2. Метод `release` используется для освобождения замка, то есть состояние принудительно переходит в `unlocked`. Если какой-то поток заблокирован и ждет получения замка, данный метод позволит начать выполнение.

Изменим пример `thread3a.py`, используя замок для оператора инкремента общей переменной `x`. Создадим замок на уровне главного потока, а затем передадим его функции `inc` для реализации блокировки вокруг общей переменной:

`#thread3b.py` с использованием синхронизации потоков

```
from threading import Lock, Thread as Thread

def inc_with_lock (lock):
    global x
```

```
    for _ in range(1000000):
        lock.acquire()
        x+=1
        lock.release()

x = 0
mylock = Lock()
#создаем потоки
t1 = Thread(target= inc_with_lock, args=(mylock,), name="Th 1")
t2 = Thread(target= inc_with_lock, args=(mylock,), name="Th 2")

#запускаем потоки
t1.start()
t2.start()

#ожидаем завершения потоков
t1.join()
t2.join()
print("final value of x :", x)
```

При использовании объекта `Lock` значение `x` всегда будет равно 2000000, поскольку теперь только один поток за раз может увеличивать значение общей переменной. Преимущество синхронизации потоков заключается в более производительном использовании системных ресурсов и предсказуемых результатах.

Однако, использовать замки следует с осторожностью, поскольку неправильная их реализация ведет к ситуации взаимной блокировки. Предположим, поток получает замок для ресурса A и ждет получения замка для ресурса B. Но другой поток уже имеет замок для ресурса B и ждет замок для ресурса A. Оба потока будут ждать освобождения замков, но этого никогда не произойдет. Во избежание взаимоблокировок библиотеки многопоточной и многопроцессорной обработки поставляются с такими механизмами, как *времени ожидания*, в течение которого ресурс может удерживать блокировку, или использовать менеджер контекста для получения замков.

Использование синхронизированной очереди

Модуль `Queue` в Python реализует очереди с множественными *производителями* и множественными *потребителями*. Очереди очень полезны в многопоточных приложениях, когда необходимо безопасно обмениваться информацией между потоками. Прелесть синхронизированной очереди в том, что она поставляется со всеми нужными механизмами блокировки, и нет необходимости использовать дополнительную семантику.

Модуль `Queue` имеет три типа очередей:

- **FIFO (First in First out)**: задача, добавленная первой, извлекается первой;
- **LIFO (Last in First out)**: задача, добавленная последней, извлекается первой;

- **Приоритетная очередь (Priority queue)**: записи сортируются, и первой извлекается запись с наименьшим приоритетом.

Эти очереди используют замки для защиты доступа к записям от конкурирующих потоков. Рассмотрим на примере, в котором создадим очередь FIFO с фиктивными задачами. Для обработки задач из очереди реализуем пользовательский потоковый класс, который наследуется от класса `Thread`. Это еще один способ реализации потока.

Для создания пользовательского класса нужно переопределить методы `init` и `run`. В методе `init` требуется вызвать метод `init` суперкласса (класса `Thread`). Метод `run` является исполнительной частью потокового класса. Результат получится следующим:

#thread5.py с очередью и пользовательским классом Thread

```
from queue import Queue
from threading import Thread as Thread
from time import sleep

class MyWorker (Thread):
    def __init__(self, name, q):
        threading.Thread.__init__(self)
        self.name = name
        self.queue = q
    def run(self):
        while True:
            item = self.queue.get()
            sleep(1)
            try:
                print ("{}: {}".format(self.name, item))
            finally:
                self.queue.task_done()

#заполняем очередь
myqueue = Queue()
for i in range (10):
    myqueue.put("Task {}".format(i+1))

#создаем потоки
for i in range (5):
    worker = MyWorker("Th {}".format(i+1), myqueue)
    worker.daemon = True
    worker.start()

myqueue.join()
```

В этом примере мы создали пять рабочих потоков с помощью пользовательского класса `MyThread`. Потоки обращаются к очереди для получения элемента задачи. По-

сле этого ждут 1 секунду и выводят имя потока и имя задачи. При каждом вызове `get` для элемента очереди последующий вызов `task_done()` указывает на завершение обработки задачи.

Важно отметить, что мы применили метод `join` к объекту `myqueue`, а не к потокам. Таким образом, он блокирует главный поток, пока все элементы в очереди не будут обработаны и завершены (для них вызывается `task_done`). Это рекомендуемый способ блокировать главный поток, когда объект очереди используется при хранении задач для потоков.

Далее реализуем приложение для загрузки файлов с Google Диска, используя класс `Thread`, класс `Queue` и пару сторонних библиотек.

Практический пример: многопоточное приложение для загрузки файлов с Google Диска

Ранее обсуждалось, что многопоточные приложения Python отлично подходят для задач ввода-вывода. Поэтому для реализации мы выбрали именно такое приложение, которое загружает файлы из общего каталога на Google Диске. Нам понадобится следующее:

- **Google Диск**: аккаунт на Google Диске (подойдет базовый бесплатный) с одним общим каталогом.
- **Ключ API**: необходим для доступа к Google API; ключ должен быть включен, тогда Google API можно будет использовать для Google Диска; включить его можно, следуя инструкциям на сайте для разработчиков Google *(https://developers.google.com/drive/api/v3/enable-drive-api)*.
- **getfilelistpy**: сторонняя библиотека, которая получает список файлов из общего каталога Google Диска; для установки библиотеки можно использовать инструмент `pip`.
- **gdown**: сторонняя библиотека, которая загружает файлы с Google Диска; для установки тоже можно использовать `pip`; существуют и другие библиотеки с тем же функционалом, мы выбрали `gdown` из-за ее простоты.

Для работы модуля `getfilelistpy` необходимо создать структуру данных ресурса. Она будет включать в себя идентификатор папки `id` (в нашем случае это идентификатор папки на Google Диске), ключ безопасности API (`api_key`) для доступа к каталогу Google Диска и список атрибутов файла (`fields`), которые будут извлечены при получении списка файлов. Создадим структуру данных ресурса следующим образом:

```
resource = {
    "api_key": "AIzaSyDYKmm85kebxddKrGns4z0",
    "id": "0B8TxHW2Ci6dbckVwTRtTl3RUU",
    "fields": "files(name, id, webContentLink)",
}
```

```
'''Ключ и идентификатор API, используемые в примерах, не являются оригинальными,
поэтому их следует заменить в соответствии с вашей учетной записью и идентификатором
общего каталога'''
```

Атрибуты файлов представлены только идентификатором, именем и веб-ссылкой (`file id`, `name` и `web link`, соответственно). Далее необходимо добавить каждый файл в очередь как задачу для потоков. Очередь будет использоваться несколькими рабочими потоками для параллельной загрузки файлов.

Для гибкости приложения с точки зрения количества рабочих потоков, которые можно использовать, создадим пул рабочих потоков. Размер пула контролируется глобальной переменной, заданной в начале программы. Мы создаем рабочие потоки в соответствии с размером пула. Каждый поток в пуле имеет доступ к очереди со списком файлов. Как и в предыдущем примере, каждый рабочий поток берет один элемент файла из очереди за раз, загружает его и помечает задачу как выполненную с помощью метода `task_done`. Пример кода для определения структуры данных ресурса и определения класса для рабочего потока будет следующим:

#threads_casestudy.py

```
from queue import Queue
from threading import Thread
import time

from getfilelistpy import getfilelist
import gdown
THREAD_POOL_SIZE = 1
resource = {
    "api_key": "AIzaSyDYKmm85kea2bxddKrGns4z0",
    "id": "0B8TxHW2Ci6dbckVweTRtTl3RUU ",
    "fields": "files(name,id,webContentLink)",
}

class DownlaodWorker(Thread):

    def __init__(self, name, queue):
        Thread.__init__(self)
        self.name = name
        self.queue = queue

    def run(self):
        while True:
            #получаем id и имя файла из очереди
            item1 = self.queue.get()
            try:
                gdown.download( item1['webContentLink'],
                    './files/{}'.format(item1['name']), quiet=False)
            finally:
                self.queue.task_done()
```

Мы получаем метаданные файлов из каталога Google Диска, используя структуру данных ресурса, следующим образом:

```
def get_files(resource):
        #глобальный список файлов files_list
        res = getfilelist.GetFileList(resource)
        files_list = res['fileList'][0]
        return files_list
```

В функции `main` мы создаем объект `Queue` для вставки метаданных файлов в очередь. Объект `Queue` передается пулу рабочих потоков. Они, в свою очередь, загружают файлы, как уже обсуждалось ранее. Мы используем класс `time` для измерения времени, необходимого для загрузки всех файлов из каталога. Код функции `main` следующий:

```
def main():
    start_time = time.monotonic()

    files = get_files(resource)

    #добавляем инфрмацию о файлах в очередь
    queue = Queue()
    for item in files['files']:
        queue.put(item)

    for i in range (THREAD_POOL_SIZE):
        worker = DownlaodWorker("Thread {}".format(i+1), queue)
        worker.daemon = True
        worker.start()

    queue.join()
    end_time = time.monotonic()
    print('Time taken to download: {} seconds'.format( end_time - start_time))

main()
```

В каталоге Google Диска имеется 10 файлов, размером от 500 КБ до 3 МБ. Мы запускали приложение с 1, 5 и 10 рабочими потоками. Общее время загрузки 10 файлов с 1 потоком составило примерно 20 секунд. Это эквивалентно коду без использования потоков (он доступен в репозитории для этой книги в качестве примера). Время работы кода без потоков для загрузки 10 файлов составило 19 секунд.

При загрузке файлов с 5 потоками на MacBook (Intel Core i5 с 16 ГБ ОЗУ) время значительно сократилось (до 6 секунд). На других машинах время может отличаться, но все равно оно будет уменьшаться при увеличении количества рабочих потоков. При использовании 10 потоков время сократилось до 4 секунд. Это показывает, что время выполнения задач ввода-вывода улучшается за счет многопоточности, независимо от ограничений глобальной блокировки интерпретатора.

Мы рассмотрели реализацию потоков в Python, а также использование механизмов блокировки с помощью классов `Lock` и `Queue`. Далее поговорим о многопроцессорном программировании на Python.

Многопроцессорная обработка

Мы увидели все сложности и ограничения многопоточного программирования, а также оценили, стоит ли его использование затраченных усилий. Для задач ввода-вывода оно может быть полезно, но в других случаях лучше выбрать альтернативный подход — многопроцессорную обработку, поскольку глобальная блокировка интерпретатора не ограничивает отдельные процессы и выполнение может происходить параллельно. Это особенно эффективно, когда приложения выполняются на многоядерных процессорах и интенсивно их используют. Во встроенных библиотеках Python многопроцессорная обработка — единственный вариант использования многоядерных процессоров.

Графические процессоры (**Graphics Processing Units, GPU**) имеют большее количество ядер, чем обычные процессоры, и лучше подходят для задач параллельной обработки. Единственная сложность заключается в необходимости переноса данных из основной памяти в память графического процессора. Эти дополнительные действия окупятся при обработке большого объема данных, но если объем небольшой, преимуществ будет мало или не будет совсем. Сегодня использование GPU для обработки больших данных, особенно для моделей машинного обучения, становится популярным. NVIDIA представила графический процессор **CUDA** для параллельной обработки, который хорошо поддерживается сторонними библиотеками Python.

На уровне ОС каждый процесс имеет структуру данных, которая называется *блок управления процессом* (**Process Control Block, PCB**). Как и *блок управления задачей* (**Task Control Block, TCB**), PCB использует *идентификатор процесса* (**Process ID, PID**), счетчик команд, регистры ЦП, информацию о планировании ресурсов ЦП и многие другие атрибуты, а также хранит состояние процесса (выполняется или ожидает).

Когда задействуются несколько процессов ЦП, изначально совместного использования памяти нет, а, значит, шанс повреждения данных ниже. Если двум процессам нужен совместным доступ к данным, им необходим какой-либо механизм взаимодействия между собой. Python поддерживает такое взаимодействие через *примитивы* (**Primitive**). Далее рассмотрим основы создания процессов в Python, а также обсудим, как достичь взаимодействия между ними.

Создание нескольких процессов

Для многопроцессорного программирования Python предоставляет пакет `multiprocessing` (аналогичный пакету `multithreading`). Он включает два подхода к

реализации многопроцессорной обработки, которые используют объект `Process` и объект `Pool`.

Рассмотрим каждый из них.

Использование объекта Process

Процессы могут быть реализованы с помощью создания объекта `Process`, а затем использования его метода `start`, аналогичного методу `start` для запуска объекта `Thread`. Фактически, объект `Process` предлагает тот же API, что и объект `Thread`. Простой пример создания нескольких дочерних процессов показан ниже:

```
#process1.py создание простых процессов с помощью функции
import os
from multiprocessing import Process, current_process as cp
from time import sleep

def print_hello():
    sleep(2)
    print("{}-{}: Hello".format(os.getpid(), cp().name))

def print_message(msg):
    sleep(1)
    print("{}-{}: {}".format(os.getpid(), cp().name, msg))

def main():
    processes = []

    #создание процессов
    processes.append(Process(target=print_hello, name="Process 1"))
    processes.append(Process(target=print_hello, name="Process 2"))
    processes.append(Process(target=print_message,
     args=["Good morning"], name="Process 3"))

    #запуск процессов
    for p in processes:
        p.start()

    #ждем завершения всех процессов
    for p in processes:
        p.join()

    print("Exiting the main process")

if __name__ == '__main__':
    main()
```

Как уже упоминалось, методы объекта Process очень похожи на аналогичные методы объекта Thread. Поэтому объяснение данного кода аналогично объяснениям кода в примерах многопоточности.

Использование объекта Pool

Объект Pool предлагает удобный способ (с помощью метода map) создания процессов, назначения им функций и распределения входных параметров по процессам. Для примера мы взяли размер пула, равный 3, но предоставили входные параметры для 5 процессов. Это нужно с целью убедиться, что одновременно активны не более трех дочерних процессов, независимо от количества параметров, переданных методом map нашего объекта Pool. Дополнительные параметры будут переданы тем же дочерним процессам, когда они закончат свое текущее выполнение. Пример кода представлен ниже:

#process2.py создание процессов с помощью pool

```
import os
from multiprocessing import Process, Pool, current_process as cp
from time import sleep

def print_message(msg):
    sleep(1)
    print("{}-{}: {}".format(os.getpid(), cp().name, msg))

def main():
    #создаем процесс из пула
    with Pool(3) as proc:
        proc.map(print_message, ["Orange","Apple","Banana","Grapes","Pears"])

 print("Exiting the main process")

if __name__ == '__main__':
    main()
```

Распределение входных параметров функции, которая привязана к набору процессов пула, осуществляется методом map. Он ожидает завершения выполнения всех функций, поэтому нет необходимости использовать join, если процессы создаются с помощью объекта Pool.

Некоторые различия между объектами Process и Pool представлены в таблице ниже:

Таблица 7.1. Сравнение объектов Pool и Process

Использование объекта Pool	Использование объекта Process
В памяти остаются только активные процессы	В памяти остаются все созданные процессы

Таблица 7.1 (окончание)

Использование объекта Pool	Использование объекта Process
Лучше работает с большими наборами данных и повторяющимися задачами	Лучше работает с малыми наборами данных
Процессы при вводе-выводе блокируются, пока не будет предоставлен ресурс ввода-вывода	Процессы при вводе-выводе не блокируются

Далее обсудим, как обмениваться данными между процессами.

Обмен данными между процессами

Пакет `multiprocessing` предлагает два способа обмена данными между процессами: общая память и серверный процесс. Рассмотрим каждый из них.

Использование общих объектов ctype (общая память)

В этом случае создается блок общей памяти, и процессы имеют к нему доступ. Блок создается при инициализации одного из типов данных `ctype`, доступных в пакете `multiprocessing`. Ими являются `Array` (массив `ctype`) и `Value` (универсальный объект `ctype`). Оба типа выделяются из общей памяти. Создать массив `ctype` можно следующим оператором:

```
mylist = multiprocessing.Array('i', 5)
```

Он создаст массив с типом данных `integer` и размером `5`. Литерал `i` — это код типа, в данном случае обозначает `integer` (целое число). Для типа данных `float` используется код `d`. Также можно инициализировать массив, передав последовательность в качестве второго аргумента вместо размера:

```
mylist = multiprocessing.Array('i', [1,2,3,4,5])
```

Создать объект `Value` можно следующим способом:

```
obj = multiprocessing.Value('i')
```

Будет создан объект `integer`, поскольку указан код `i`. Значение объекта можно задать с помощью атрибута `value`.

Оба объекта `ctype` имеют опциональный аргумент `Lock` со значением `True` по умолчанию. При значении `True` этот аргумент используется для создания нового рекурсивного объекта блокировки, который предоставляет синхронизированный доступ к значениям объектов. При значении `False` защита отключена, и процесс будет небезопасным. Если процесс обращается к общей памяти только для чтения, параметру `Lock` можно установить `False`. В дальнейших примерах кода мы оставим для аргумента `Lock` значение по умолчанию `True`.

Для демонстрации использования объектов `ctype` из общей памяти создадим список с 3 числовыми значениями, массив `ctype` размера `3` для хранения увеличенных значений исходного массива и объект `ctype` для хранения суммы увеличенного масси-

ва. Эти объекты будут созданы родительским процессом в общей памяти, а затем обновлены дочерним процессом. Взаимодействие родительского и дочернего процессов с общей памятью показано на схеме (рис. 7.3):

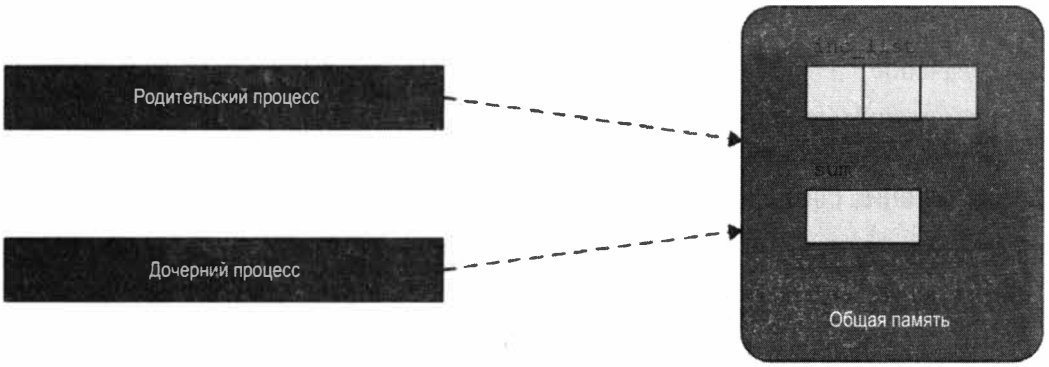


Рис. 7.3. Использование общей памяти родительским и дочерним процессами

Далее показан полный пример кода использования общей памяти:

#process3.py использование общей памяти объектами ctype

```
import multiprocessing
from multiprocessing import Process, Pool, current_process as cp

def inc_sum_list(list, inc_list, sum):
    sum.value = 0
    for index, num in enumerate(list):
        inc_list[index] = num + 1
        sum.value = sum.value + inc_list[index]

def main():

    mylist = [2, 5, 7]

    inc_list = multiprocessing.Array('i', 3)

    sum = multiprocessing.Value('i')

    p = Process(target=inc_sum_list, args=(mylist, inc_list, sum))

    p.start()
    p.join()
    print("incremented list: ", list(inc_list))
    print("sum of inc list: ", sum.value)

    print("Exiting the main process")

if __name__ == '__main__':
    main()
```

К общим типам данных (в нашем случае `inc_list` и `sum`) обращаются как родительский, так и дочерний процессы. Важно отметить, что использовать общую память не рекомендуется, поскольку для этого требуются механизмы синхронизации и блокировки (аналогично механизмам в случае многопоточности), когда к одним и тем же объектам общей памяти обращаются несколько процессов, а аргумент `Lock` имеет значение `False`.

Далее рассмотрим использование серверного процесса.

Использование серверного процесса

В этом случае серверный процесс запускается сразу после старта программы Python. Он используется для создания и управления новыми дочерними процессами, запрошенными родительским процессом. Серверный процесс может содержать объекты Python, к которым другие процессы могут обращаться через прокси.

Для реализации серверного процесса и совместного использования объектов пакет `multiprocessing` предоставляет объект `Manager`. Он поддерживает разные типы данных, включая:

- списки;
- словари;
- `Lock`;
- `Rlock`;
- очереди;
- `Value`;
- массивы.

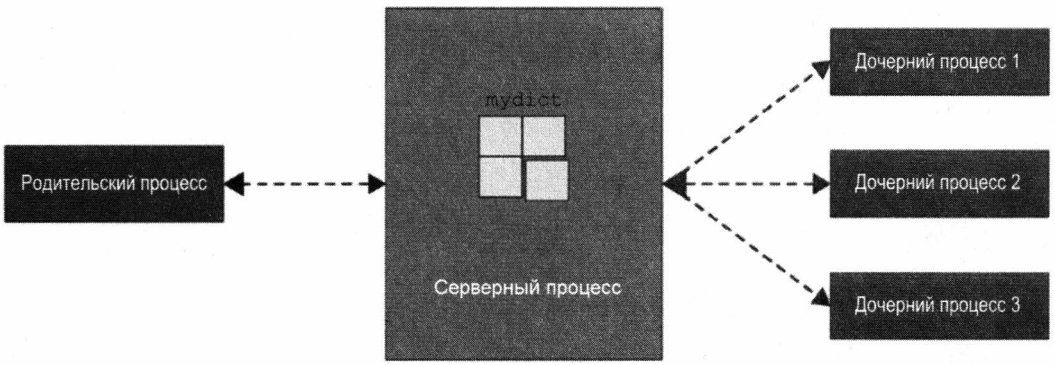


Рис. 7.4. Использование серверного процесса для обмена данными между процессами

Пример кода, который мы выбрали для демонстрации серверного процесса, создает объект `dictionary` с помощью объекта `Manager`, а затем передает словарь дочерним процессам для вставки дополнительных данных и вывода содержимого. Для нашего примера мы создадим три дочерних процесса: два для вставки данных в объект словаря и один для получения содержимого словаря для вывода на консоль. На рис. 7.4

приведена схема взаимодействия между родительским, серверным и тремя дочерними процессами. Родительский создает серверный, как только выполняется запрос нового процесса с помощью контекста `Manager`. Дочерние процессы создаются и управляются серверным процессом. Общие данные доступны в серверном процессе, а также во всех остальных, включая родительский.

Полный пример кода показан ниже:

#process4.py использование общей памяти с помощью серверного процесса

```
import multiprocessing
from multiprocessing import Process, Manager

def insert_data (dict1, code, subject):
    dict1[code] =  subject

def output(dict1):
    print("Dictionary data: ", dict1)

def main():
    with multiprocessing.Manager() as mgr:
        #создаем словарь в серверном процессе
        mydict = mgr.dict({100: "Maths", 200: "Science"})

        p1 = Process(target=insert_data, args=(mydict, 300, "English"))
        p2 = Process(target=insert_data, args=(mydict, 400, "French"))
        p3 = Process(target=output, args=(mydict,))

        p1.start()
        p2.start()

        p1.join()
        p2.join()

        p3.start()
        p3.join()

        print("Exiting the main process")

if __name__ == '__main__':
    main()
```

Серверный процесс обеспечивает больше гибкости, чем использование общей памяти, поскольку он поддерживает огромное разнообразие типов объектов. Однако этот способ имеет низкую производительность по сравнению с общей памятью.

Обмен объектами между процессами

В предыдущем подразделе мы узнали, как процессы могут обмениваться данными через внешний блок памяти или новый процесс. Здесь мы исследуем обмен данными между процессами с использованием объектов Python. Модуль `multiprocessing` предоставляет для этого два способа: объект `Queue` и объект `Pipe`.

Использование объекта Queue

Объект `Queue` доступен в пакете `multiprocessing` и аналогичен объекту синхронизированной очереди (`queue.Queue`), который мы использовали для многопоточности. Объект `Queue` безопасен для процессов и не требует дополнительной защиты. Ниже показан пример работы данного объекта:

#process5.py использование очереди для обмена данными

```
import multiprocessing
from multiprocessing import Process, Queue

def copy_data (list, myqueue):
    for num in list:
        myqueue.put(num)

def output(myqueue):
    while not myqueue.empty():
        print(myqueue.get())

def main():
    mylist = [2, 5, 7]
    myqueue = Queue()

    p1 = Process(target=copy_data, args=(mylist, myqueue))
    p2 = Process(target=output, args=(myqueue,))

    p1.start()
    p1.join()
    p2.start()
    p2.join()

    print("Queue is empty: ",myqueue.empty())
    print("Exiting the main process")

if __name__ == '__main__':
    main()
```

Мы создали стандартный объект `list` и многопроцессорный объект `Queue`. Она оба передаются новому процессу, который связан с функцией `copy_data`. Она копирует

данные из list в Queue. Инициируется новый процесс для вывода содержимого объекта Queue. Обратите внимание, данные в Queue задаются предыдущим процессом и доступны новому процессу. Это удобный способ обмена данными без лишней сложности общей памяти или серверного процесса.

Использование объекта Pipe

Объект Pipe можно сравнить с каналом между двумя процессами для обмена данными. Он особенно удобен, если нужна двусторонняя связь. Когда мы создаем Pipe, он предоставляет два объекта соединения, находящихся по обе стороны канала. Каждый из них предоставляет методы send и recv для отправки и получения данных соответственно.

Для демонстрации создадим две функции, которые будут прикреплены к двум отдельным процессам:

- Первая будет отправлять сообщение через соединение объекта Pipe; мы отправим несколько сообщений и завершим взаимодействие сообщением BYE.
- Вторая функция будет получать сообщение через другое соединение объекта Pipe; она будет выполняться бесконечным циклом, пока не получит сообщение BYE.

Обе функции (или процесса) обеспечиваются двумя объектами соединения. Полный код выглядит следующим образом:

#process6.py обмен данными через Pipe

```
from multiprocessing import Process, Pipe

def mysender (s_conn):
    s_conn.send({100, "Maths"})
    s_conn.send({200, "Science"})
    s_conn.send("BYE")

def myreceiver(r_conn):
    while True:
        msg = r_conn.recv()
        if msg == "BYE":
            break
        print("Received message : ", msg)

def main():
    sender_conn, receiver_conn = Pipe()

    p1 = Process(target=mysender, args=(sender_conn, ))
    p2 = Process(target=myreceiver, args=(receiver_conn,))

    p1.start()
    p2.start()
```

```
    p1.join()
    p2.join()

    print("Exiting the main process")

if __name__ == '__main__':
    main()
```

Стоит отметить, что данные в объекте `Pipe` легко могут повредиться, когда два процесса попытаются одновременно выполнить чтение или запись, используя один объект соединения. Поэтому многопроцессорные очереди являются лучшим вариантом, они обеспечивают надлежащую синхронизацию между процессами.

Синхронизация процессов

Синхронизация процессов гарантирует, что два или более процесса не будут обращаться к одному ресурсу или программному коду (критической секции) одновременно. Эта ситуация может привести к состоянию гонки и повреждению данных. Вероятность возникновения этого не очень высока, но все же возможна. Этих ситуаций можно избежать, используя либо подходящие объекты со встроенной синхронизацией, либо объект `Lock`, как в случае многопоточности.

Мы рассмотрели пример использования типов данных `queues` и `ctype` с включенной блокировкой (`Lock = True`), что делает процесс безопасным. В следующем примере мы продемонстрируем использование `Lock` с целью убедиться, что один процесс получает доступ к консольному выводу за раз. Для этого мы создали процессы, используя объект `Pool`. Для передачи одного и того же объекта `Lock` всем процессам мы использовали его из объекта `Manager`, а не из пакета `multiprocessing`. Мы также использовали функцию `partial` для привязки `Lock` к каждому процессу вместе со списком, который передан каждой функции процесса. Полный пример кода представлен ниже:

```
#process7.py демонстрация синхронизации и блоокировки

from functools import partial
from multiprocessing import Pool, Manager

def printme (lock, msg):
    lock.acquire()
    try:
        print(msg)
    finally:
        lock.release()

def main():
    with Pool(3) as proc:
```

```
        lock = Manager().Lock()
        func = partial(printme,lock)
        proc.map(func, ["Orange", "Apple", "Banana", "Grapes","Pears"])

    print("Exiting the main process")

if __name__ == '__main__':
    main()
```

Без использования объекта `Lock` выходные данные от разных процессов могут быть перепутаны.

Практический пример: многопроцессорное приложение для загрузки файлов с Google Диска

Здесь мы реализуем пример, аналогичный предыдущему, но вместо многопоточности используем многопроцессорность. Требования и задачи будут такие же, как для многопоточного приложения.

Мы будем использовать тот же код, что и в случае многопоточности, за исключением использования множества процессов вместо потоков. Еще одно отличие — задействование объекта `JoinableQueue` из модуля `multiprocessing`, который обеспечивает ту же функциональность, что и объект `Queue`. Код для определения структуры данных ресурса и для функции загрузки файлов с Google Диска показан далее:

#processes_casestudy.py

```
import time
from multiprocessing import Process, JoinableQueue
from getfilelistpy import getfilelist
import gdown
PROCESSES_POOL_SIZE = 5

resource = {
    "api_key": "AIzaSyDYKmm85keqnk4bF1Da2bxddKrGns4z0",
    "id": "0B8TxHW2Ci6dbckVwetTlV3RUU",
    "fields": "files(name,id,webContentLink)",
}

def mydownloader( queue):
    while True:
        #получаем id и имя файла из очереди
        item1 =  queue.get()
        try:
            gdown.download(item1['webContentLink'],
                           './files/{}'.format(item1['name']),
                           quiet=False)
        finally:
            queue.task_done()
```

Мы получаем метаданные файлов, такие, как имя, и HTTP-ссылку из каталога, используя структуру данных ресурса, как показано ниже:

```
def get_files(resource):
    res = getfilelist.GetFileList(resource)
    files_list = res['fileList'][0]
    return files_list
```

В функции `main` мы создаем объект `JoinableQueue` и вставляем метаданные файлов в очередь. Очередь передается пулу процессов, которые загрузят файлы. Мы задействуем класс `time` для измерения времени, необходимого для загрузки всех файлов из Google Диска. Код функции `main` выглядит так:

```
def main ():

    files = get_files(resource)
    #добавляем информацию о файлах в очередь
    myqueue = JoinableQueue()
    for item in files['files']:
        myqueue.put(item)

    processes = []
    for id in range(PROCESSES_POOL_SIZE):
        p = Process(target=mydownloader, args=(myqueue,))
        p.daemon = True
        p.start()

    start_time = time.monotonic()
    myqueue.join()
    total_exec_time = time.monotonic() - start_time

    print(f'Time taken to download: {total_exec_time:.2f} seconds')

if __name__ == '__main__':
    main()
```

Мы запустили приложение с разным числом процессов: 3, 5, 7 и 10. Время загрузки одних и тех же файлов немного быстрее, чем в многопоточном приложении. Время выполнения зависит от компьютера. В нашем случае (MacBook Pro: Intel Core i5 с 16 ГБ ОЗУ) оно заняло около 5 секунд при 5 процессах и 3 секунды при 10 процессах, работающих параллельно. Это на 1 секунду быстрее, чем в многопоточном приложении, как и ожидалось, поскольку многопроцессорная обработка обеспечивает настоящий параллелизм.

Асинхронное программирование для адаптивных систем

Многопроцессорность и многопоточность относятся к синхронному программированию, где отправляется запрос и ожидается ответ, прежде чем будет выполнен следующий блок кода. И если какое-либо переключение контекста и применяется, то это обеспечивается операционной системой. Асинхронное программирование в Python отличается двумя аспектами:

- Задачи должны быть созданы для асинхронного выполнения; это означает, что родительскому вызывающему объекту не нужно ждать ответа от другого процесса; тот ответит, как только закончит выполнение.
- Операционная система больше не управляет переключением контекста между процессами и потоками; асинхронная программа получит только один поток в процессе, но его возможности будут разнообразны; при таком стиле выполнения каждый процесс или задача добровольно передают контроль в случае простоя или ожидания другого ресурса; таким образом, другие задачи тоже могут выполняться; эта концепция называется кооперативной многозадачностью.

Кооперативная многозадачность (**Cooperative Multitasking**) — это эффективный инструмент для достижения параллелизма на уровне приложения. При таком подходе создаются не процессы или потоки, а задачи, которые включают *тасклеты* (**Tasklet**), *корутины* (**Coroutine**) и *зеленые потоки* (**Green Thread**). Они координируются одной функцией, которая называется *циклом событий* (**Event Loop**). Он регистрирует задачи и обрабатывает поток управления между ними. Прелесть заключается в том, что цикл событий в Python реализуется с помощью генераторов. А, значит, они могут выполнять функцию и приостанавливать ее в определенной точке (с помощью `yield`), сохраняя при этом контроль над стеком объектов до возобновления работы.

Для систем, основанных на кооперативной многозадачности, всегда стоит вопрос, когда возвращать управление планировщику или циклу событий. Наиболее популярная логика заключается в использовании операций ввода-вывода в качестве событий для освобождения управления, поскольку для этих операций всегда требуется время ожидания.

Возникает вопрос, разве не эту же логику мы использовали для многопоточности? Она безусловно повышает производительность приложения при работе с операциями ввода-вывода, но есть разница. В случае многопоточности ОС управляет переключением контекста между потоками и может вытеснить любой запущенный поток по любой причине и передать управление другому. Но при асинхронном программировании или кооперативной многозадачности операционной системе не видны задачи и корутины. Фактически они не могут быть вытеснены главным циклом событий. Но это не означает, что ОС не может вытеснить весь процесс Python. Он по-прежнему конкурирует за ресурсы с другими приложениями и процессами на уровне ОС.

Далее изучим некоторые понятия асинхронного программирования, которое предоставляется модулем asyncio, а затем завершим подробным анализом конкретного примера.

Модуль asyncio

Модуль asyncio доступен, начиная с Python 3.5 и более поздних версиях. Он предназначен для написания конкурентных программ с помощью синтаксиса async/await. Но для сложных приложений рекомендуется использовать Python, начиная с версии 3.7. Библиотека богата множеством возможностей, поддерживает создание и выполнение корутинов, выполнение сетевых операций ввода-вывода, распределение задач по очередям и синхронизацию параллельного кода.

Для начала рассмотрим, как писать и выполнять корутины и задачи.

Корутины и задачи

Корутины — это функции, которые должны выполняться асинхронно. Простой пример вывода строки на консоль с помощью них выглядит так:

```
#asyncio1.py создание простого корутина
import asyncio
import time

async def say(delay, msg):
    await asyncio.sleep(delay)
    print(msg)

print("Started at ", time.strftime("%X"))
asyncio.run(say(1,"Good"))
asyncio.run(say(2, "Morning"))
print("Stopped at ", time.strftime("%X"))
```

Ключевые моменты кода:

- Корутин принимает аргументы delay и msg; delay позволяет добавить задержку перед отправкой строки msg на консоль.
- Здесь используется функция asyncio.sleep вместо традиционной time.sleep, поскольку последняя не возвращает контроль циклу событий; вот почему важно использовать совместную функцию asyncio.sleep.
- Корутин выполняется дважды с двумя разными значениями аргумента delay с помощью метода run, который не будет выполнять корутины совместно.

Консольный вывод показывает, что корутины выполняются один за другим, поскольку общая задержка составляет 3 секунды:

```
Started at 15:59:55
Good
Morning
Stopped at 15:59:58
```

Параллельное выполнение корутинов можно запустить функцией `create_task` из модуля `asyncio`. Она создает задачу, с помощью которой можно планировать совместное выполнение корутинов.

В следующем примере изменим `asyncio1.py` и обернем корутин в задачу, используя функцию `create_task`. А если точнее, создадим две задачи, которые служат обертками для `say`. Дождемся выполнения обеих задач с помощью `await`:

```
#asyncio2.py создание и параллельное выполнение корутинов
import asyncio
import time

async def say(delay, msg):
    await asyncio.sleep(delay)
    print(msg)

async def main ():
    task1 = asyncio.create_task( say(1, 'Good'))
    task2 = asyncio.create_task( say(1, 'Morning'))
    print("Started at ", time.strftime("%X"))
    await task1
    await task2
    print("Stopped at ", time.strftime("%X"))

asyncio.run(main())
```

Консольный вывод будет следующим:

```
Started at 16:04:40
Good
Morning
Stopped at  16:04:41
```

Вывод показывает, что обе задачи выполнились за 1 секунду. Это доказывает, что они выполнялись параллельно.

Объекты, ожидающие результатов

Объект является *ожидающим* (**awaitable**), если можно применить к нему оператор `await`. Множество функций и модулей внутри `asyncio` предназначено для работы с ожидающими результатов объектами. Но большинство объектов Python и сторонних библиотек не создано для асинхронного программирования. При разработке таких приложений важно выбирать совместимые библиотеки, которые могут предоставить ожидающие результатов объекты.

Такие объекты делятся на три типа: корутины, задачи и *фьючеры* (**Future**). С корутинами и задачами мы уже знакомы. Фьючер — это низкоуровневый объект, аналогичный механизму *колбэк* (**Callback**), который используется для обработки результата, поступающего от `async/await`. Объекты `Future` обычно не предоставляются для программирования на уровне пользователей.

Одновременное выполнение задач

Если нужно запустить несколько задач параллельно, можно использовать `await`, как в предыдущем примере. Но есть способ лучше — функция `gather`. Она запустит ожидающие объекты в указанной последовательности. Если какой-то из них является корутином, он будет запланирован как задача. Использование функции `gather` мы увидим в следующем подразделе.

Распределение задач с помощью очередей

Объект `Queue` в пакете `asyncio` похож на модуль `Queue`, но не является потокобезопасным. Модуль `asyncio` предоставляет различные реализации очередей, например FIFO, LIFO и приоритетные очереди. Очереди из модуля `asyncio` могут быть использованы для распределения рабочих нагрузок по задачам.

Для демонстрации напишем небольшую программу, которая будет имитировать выполнение реальной функции, засыпая на случайный промежуток времени. Неопределенное время ожидания рассчитывается для 10 выполнений и добавляется к объекту `Queue` основным процессом в качестве рабочих элементов. Объект `Queue` передается в пул из трех задач. Каждая из них выполняет назначенный корутин, который потребляет время выполнения в соответствии с доступной ему записью в очереди. Полный пример кода:

```
#asyncio3.py распределение рабочих нагрузок через очередь
import asyncio
import random
import time

async def executer(name, queue):
    while True:
```

```
        exec_time = await queue.get()
        await asyncio.sleep(exec_time)
        queue.task_done()
        #print(f'{name} has taken  {exec_time:.2f} seconds')

async def main ():
    myqueue = asyncio.Queue()
    calc_exuection_time = 0
    for _ in range(10):
        sleep_for = random.uniform(0.4, 0.8)
        calc_exuection_time += sleep_for
        myqueue.put_nowait(sleep_for)

    tasks = []
    for id in range(3):
        task = asyncio.create_task(executer(f'Task-{id+1}', myqueue))
        tasks.append(task)

    start_time = time.monotonic()
    await myqueue.join()
    total_exec_time = time.monotonic() - start_time

    for task in tasks:
        task.cancel()

    await asyncio.gather(*tasks, return_exceptions=True)

    print(f"Calculated execution time {calc_exuection_time:0.2f}")
    print(f"Actual execution time {total_exec_time:0.2f}")

asyncio.run(main())
```

Мы использовали функцию `put_no_wait` объекта `Queue`, поскольку она является неблокирующей операцией. Консольный вывод будет следующим:

```
Calculated execution time 5.58
Actual execution time 2.05
```

Вывод указывает, что задачи выполняются параллельно. Это в 3 раза быстрее, чем при последовательном выполнении.

Пока что мы рассмотрели базовые концепции пакета `asyncio`. Прежде чем завершить тему, вернемся к уже знакомому примеру из подраздела про многопоточность и реализуем его с помощью `asyncio`.

Практический пример: асинхронное приложение для загрузки файлов с Google Диска

Мы реализуем уже знакомое приложение, но на этот раз будем использовать модуль `asyncio` с функциями `async`, `await` и `async queue`. Требования остаются такие же, за исключением использования библиотек `aiohttp` и `aiofiles` вместо `gdown`. Причина заключается в том, что `gdown` не создана как асинхронный модуль и не подходит для такого программирования. Это важно учитывать при выборе библиотек для работы с асинхронными приложениями.

Для этого приложения мы создали корутин `mydownloader`, который загружает файлы с Google Диска, используя модули `aiohttp` и `aiofiles`. Полный пример кода с выделенными отличиями показан ниже:

#asyncio_casestudy.py

```
import asyncio
import time

import aiofiles, aiohttp
from getfilelistpy import getfilelist

TASK_POOL_SIZE = 5

resource = {
    "api_key": "AIzaSyDYKmm85keqnk4bF1DpYa2dKrGns4z0",
    "id": "0B8TxHW2Ci6dbckVwetTlV3RUU",
    "fields": "files(name, id, webContentLink)",
}

async def mydownloader(name, queue):
    while True:
        #получаем id и имя файла из очереди
        item = await queue.get()
        try:
            async with aiohttp.ClientSession() as sess:
                async with sess.get(item['webContentLink']) as resp:
                    if resp.status == 200:
                        f = await aiofiles.open('./files/{}'
                          .format(item['name']), mode='wb')
                        await f.write(await resp.read())
                        await f.close()
        finally:
            print(f"{name}: Download completed for ",item['name'])
            queue.task_done()
```

Процесс загрузки списка файлов из общей папки аналогичен предыдущим примерам с многопоточностью и многопроцессорной обработкой. Здесь мы создали пул задач (настраиваемый) на основе корутина `mydownloader`. Затем эти задачи планируются для совместного выполнения, и наш родительский процесс ждет завершения всех задач. Код выглядит следующим образом:

```
def get_files(resource):
    res = getfilelist.GetFileList(resource)
    files_list = res['fileList'][0]
    return files_list

async def main ():
    files = get_files(resource)
    #добавляем информацию о файлах в очередь
    myqueue = asyncio.Queue()
    for item in files['files']:
        myqueue.put_nowait(item)

    tasks = []
    for id in range(TASK_POOL_SIZE):
        task = asyncio.create_task(mydownloader(f'Task-{id+1}', myqueue))
        tasks.append(task)

    start_time = time.monotonic()
    await myqueue.join()
    total_exec_time = time.monotonic() - start_time

    for task in tasks:
        task.cancel()

    await asyncio.gather(*tasks, return_exceptions=True)

    print(f'Time taken to download: {total_exec_time:.2f} seconds')

asyncio.run(main())
```

Мы запустили приложение, варьируя количество задач (3, 5, 7 и 10). Время загрузки файлов с помощью `asyncio` требует меньше времени, чем с помощью многопоточности или многопроцессорности.

Время выполнения меняется в зависимости от компьютера, но в нашем случае (MacBook Pro Intel Core i5 с 16 ГБ ОЗУ) потребовалось около 4 секунд при 5 задачах и 2 секунды при 10 задачах. Это ощутимо быстрее, чем в двух предыдущих примерах. Такой результат ожидаем, поскольку `asyncio` обеспечивает лучшую реализацию параллелизма для задач ввода-вывода, когда их нужно осуществить с использованием подходящего набора объектов программирования.

На этом мы завершаем изучение асинхронного программирования. Мы представили все ключевые компоненты для создания асинхронного приложения с помощью пакета `asyncio`.

Заключение

В этой главе мы рассмотрели различные варианты параллельного программирования на Python с использованием стандартных библиотек. Мы начали с многопоточности и основных принципов конкурентного выполнения. Затем узнали о такой проблеме многопоточности, как глобальная блокировка интерпретатора, которая разрешает доступ к объектам Python только одному потоку за раз. На практических примерах изучили концепции блокировки и синхронизации. Мы также обсудили типы задач, для которых многопоточное программирование наиболее эффективно.

Мы узнали, как достичь параллелизма с помощью нескольких процессов в Python. Увидели, как обмениваться данными между процессами, используя общую память и серверный процесс, а также как безопасно обмениваться объектами с помощью `Queue` и `Pipe`. Наконец, написали пример с несколькими процессами, аналогичный многопоточному приложению. Затем познакомились с совершенно другим подходом к параллелизму с помощью асинхронного программирования. Мы начали с базовых концепций и ключевых слов `async` и `await`. Увидели, как создавать задачи или корутины с помощью пакета `asyncio`. И завершили главу примером с использованием принципов асинхронного программирования.

В этой главе представлено много практических примеров реализации параллельных приложений. Эти знания пригодятся всем, кто хочет создавать многопоточные или асинхронные приложения с помощью стандартных библиотек, доступных в Python.

В следующей главе мы поговорим про использование сторонних библиотек для создания параллельных приложений.

Вопросы

1. Что координирует потоки Python? Это интерпретатор Python?
2. Что такое GIL в Python?
3. Когда следует использовать потоки-демоны?
4. Что лучше использовать в системе с ограниченной памятью для создания процессов: объект `Process` или объект `Pool`?
5. Что такое `Futures` в пакете `asyncio`?
6. Что такое цикл событий в асинхронном программировании?
7. Как написать асинхронный корутин или функцию на Python?

Дополнительные ресурсы

- *«Learning Concurrency in Python»*, автор: Эллиот Форбс (Elliot Forbes).
- *«Python. Лучшие практики и инструменты»* (Expert Python Programming) авторы: Михал Яворски (Michał Jaworski) и Тарек Зиаде (Tarek Ziadé).
- *«Python 3 Object-Oriented Programming»*, автор: Дасти Филлипс (Dusty Phillips).
- *«Mastering Concurrency in Python»*, автор: Гуан Нгуен (Quan Nguyen).
- *«Python Concurrency with asyncio»*, автор: Мэтью Фаулер (Mathew Fowler).

Ответы

1. Потоки и процессы координируются ядром операционной системы.
2. GIL (Глобальная блокировка интерпретатора) в Python — это механизм блокировки для одновременного выполнения только одного потока за раз.
3. Потоки-демоны используются, когда завершение потока не является проблемой после завершения основного потока.
4. Объект `Pool` является лучшим выбором, поскольку хранит в памяти только один активный процесс.
5. `Futures` похожи на механизм callback, который используется для обработки результатов от вызовов `async/await`.
6. Объект цикл событий отслеживает задачи и контролирует поток управления между ними.
7. Асинхронный корутин можно начать писать с `async def`.

8

Масштабирование Python с помощью кластеров

В предыдущей главе мы обсуждали параллельную обработку на одном компьютере с использованием потоков и процессов. Здесь мы поговорим о параллельной обработке на нескольких машинах в кластере. *Кластер* (**Cluster**) — это группа вычислительных устройств, работающих вместе для выполнения ресурсоемких задач вроде обработки данных. В частности, мы рассмотрим возможности Python в области интенсивных вычислений, которые обычно используют кластеры для параллельной обработки больших объемов информации. Для таких задач доступно множество инструментов, но мы сосредоточимся на **Apache Spark** (как на механизме обработки данных) и на **PySpark** (как на библиотеке Python для создания таких приложений).

Если Apache Spark правильно настроен и реализован, можно существенно повысить производительность приложений и значительно превзойти платформы конкурентов, наподобие **Hadoop MapReduce**. Мы также рассмотрим, как *распределенные наборы данных* используются в кластерной среде. Эта глава поможет понять, как использовать платформы кластерных вычислений для обработки больших объемов данных, а также как реализовать подобные приложения на Python. Для демонстрации практического применения языка в области кластеров рассмотрим два учебных примера: первый — расчет значения числа π, а второй — генерирование облака слов из файла данных.

Темы этой главы:

- Возможности кластеров для параллельной обработки.
- Устойчивые распределенные наборы данных.
- PySpark для параллельной обработки.
- Практические примеры использования Apache Spark и PySpark.

К концу главы вы узнаете, как писать приложения для обработки данных, которые можно выполнять на рабочих узлах в кластере Apache Spark.

Технические требования

В этой главе понадобится:

- Python 3.7 или более поздней версии.
- Кластер Apache Spark с одним узлом.
- PySpark поверх Python для разработки программ-драйверов.

> ***Примечание***
>
> Версия, используемая для Apache Spark, должна совпадать с версией Python для запуска драйверов.

Пример кода для этой главы можно найти по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter08.

Начнем с рассмотрения опций кластера, доступных для параллельной обработки.

Возможности кластеров для параллельной обработки

При обработке большого объема информации нам может не хватить одного компьютера с несколькими ядрами. Особенно, если речь идет о потоковой обработке в режиме реального времени. В таких ситуациях есть необходимость использования нескольких систем, способных обрабатывать информацию распределенным образом и выполнять задачи параллельно. Использование нескольких компьютеров для подобных ресурсоемких задач называется *кластерными вычислениями* (**Cluster computing**). Есть несколько фреймворков распределенной обработки больших данных, доступных для координации выполнения заданий в кластере. Самые популярные из них: **Hadoop MapReduce** и **Apache Spark**. Оба фреймворка имеют открытый исходный код и поставляются Apache. Существуют разные варианты (например, **Databricks**) их исполнения с дополнительными возможностями, а также поддержкой обслуживания, но основные принципы остаются прежними.

Если взглянуть на рынок, количество развертываний Hadoop MapReduce может быть больше, но рост популярности Apache Spark в конечном итоге меняет ситуацию. Поскольку Hadoop MapReduce по-прежнему очень актуален, важно понять, что он собой представляет и почему Apache Spark является лучшим выбором. Кратко рассмотрим их далее.

Hadoop MapReduce

Hadoop — платформа распределенной обработки общего назначения, которая выполняет задания по обработке больших объемов данных на сотнях или тысячах вычислительных узлов в кластере Hadoop. На рис. 8.1 описаны три ключевых компонента:

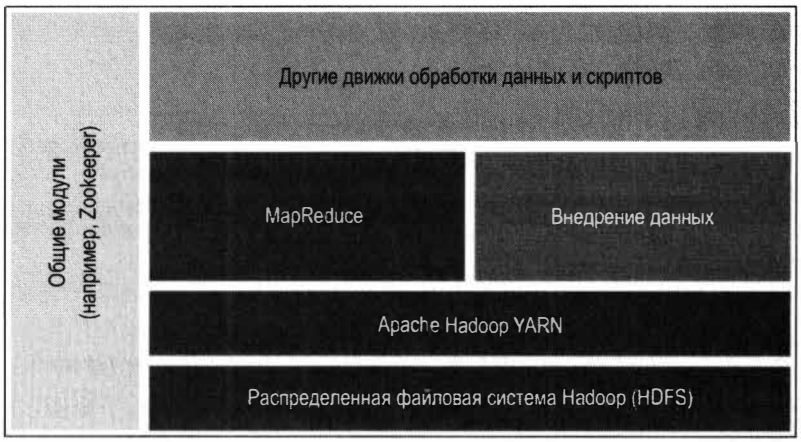


Рис. 8.1. Экосистема Apache Hadoop MapReduce

Рассмотрим подробнее:

- **Распределенная файловая система Hadoop (Hadoop Distributed File System, HDFS)** — нативная файловая система для хранения файлов таким образом, что их можно распараллелить в кластере.
- **YARN (Yet Another Resource Negotiator)** — система, которая обрабатывает хранящиеся в HDFS данные и планирует выполнение заданий системой; может использоваться для обработки графов, потоков или пакетов.
- **MapReduce** — фреймворк, который позволяет обрабатывать большие наборы данных, распределяя их на несколько более мелких наборов; он обрабатывает данные с помощью двух функций: `map` и `reduce`; их роли аналогичны тем, что мы рассматривали в главе 6 (*«Расширенные советы и приемы Python»*); ключевое отличие заключается в том, что мы используем множество функций `map` и `reduce` параллельно для обработки нескольких наборов данных одновременно.
- Разделив большой набор данных на несколько маленьких, можно передать их в качестве входных множеству функций `map` для обработки на разных узлах кластера. Каждая функция `map` принимает на входе один набор данных, обрабатывает его в соответствии с требованиями и выдает результат в виде пар *«ключ-значение»*. Как только будут доступны выходные данные от всех функций `map`, одна или несколько функций `reduce` объединят результаты в соответствии с требованиями.

- Для более подробного объяснения возьмем пример подсчета определенных слов, вроде *«attack»* или *«weapon»*, в большом наборе текста. Текстовую информацию можно разделить на небольшие наборы данных (например, восемь). Таким образом, мы будем иметь восемь функций `map` для подсчета двух слов в предоставленном наборе. Каждая функция `map` на выходе предоставляет количество слов *«attack»* и *«weapon»* в своем наборе. На следующем этапе выходные данные всех функций `map` передаются двум функциям `reduce`, по одной для каждого слова. Каждая функция `reduce` агрегирует результаты функций `map` для своего слова и предоставляет совокупный результат в качестве вывода. Схема работы фреймворка `MapReduce` для этого примера представлена на рис. 8.2. Обратите внимание, что в Python есть одноименные функции `map` и `reduce`:

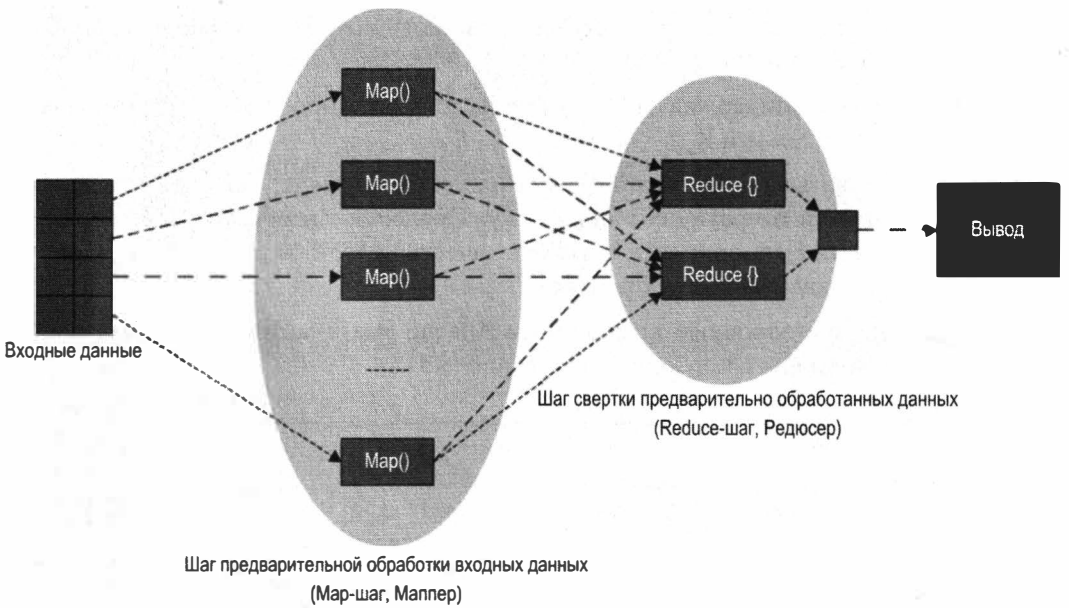


Рис. 8.2. Работа фреймворка MapReduce

Остальные компоненты мы пропустим, поскольку они не имеют отношения к обсуждению в этой главе. Hadoop, в основном, построен на Java, но писать компоненты `map` и `reduce` для модуля `MapReduce` можно на любом языке программирования, включая Python.

Hadoop MapReduce хорошо подходит для обработки больших фрагментов данных путем их разделения на небольшие блоки. Узлы кластера обрабатывают их по отдельности, а перед отправкой запрашивающему объекту результаты объединяются. Hadoop MapReduce обрабатывает данные из файловой системы, поэтому не очень эффективен с точки зрения производительности. Однако он работает очень хорошо, когда скорость не является критичным требованием, например, если обработка выполняется ночью.

Apache Spark

Apache Spark — фреймворк кластерных вычислений с открытым исходным кодом для потоковой и пакетной обработки данных в режиме реального времени. Главная его особенность заключается в обработке информации прямо в оперативной памяти, что разгружает CPU или GPU, обеспечивает низкую задержку и подходит для многих реальных сценариев благодаря следующим дополнительным факторам:

- Он обеспечивает быстрое получение результатов для критически важных и чувствительных ко времени приложений (например, для обработки в режиме реального времени).
- Он хорошо подходит для многократного выполнения задач благодаря обработке непосредственно в оперативной памяти.
- Он позволяет использовать алгоритмы машинного обучения прямо «из коробки» (без дополнительной настройки).
- В нем доступна поддержка дополнительных языков программирования, таких, как Java, Python, Scala и R.

Apache Spark охватывает широкий спектр рабочих нагрузок, включая пакетные и потоковые данные и итеративную обработку. Прелесть заключается в его возможности использовать Hadoop (через YARN) в качестве кластера развертывания, но он имеет и собственный менеджер кластеров.

На высоком уровне основные компоненты Apache Spark можно разделить на три уровня, как показано на следующей схеме (рис. 8.3).

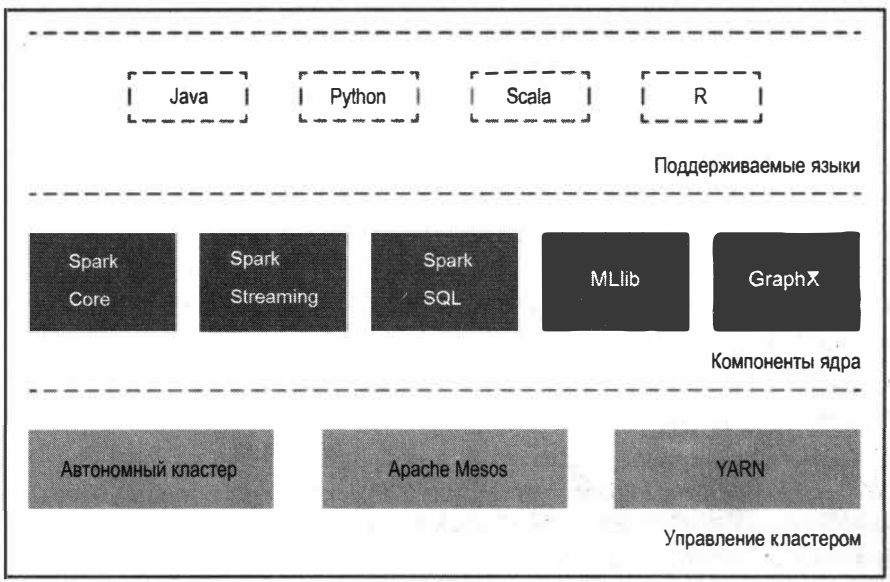


Рис. 8.3. Экосистема Apache Spark

Рассмотрим уровни подробнее.

Поддержка языков

Scala — нативный язык Apache Spark, поэтому он широко применяется для разработки. Фреймворк также предоставляет высокоуровневые API для Java, Python и R. Многоязычная поддержка обеспечивается с помощью интерфейса *удаленного вызова процедур* (**Remote Procedure Call, RPC**). Для каждого языка есть RPC-адаптер, написанный на Scala. Он преобразует клиентские запросы, написанные на другом языке, в нативные запросы на Scala. Что облегчает принятие языка сообществом разработчиков.

Основные компоненты

Кратко рассмотрим основные компоненты Apache Spark:

- **Spark Core** и **RDD**: Spark Core — это движок, который отвечает за обеспечение абстракции для RDD, планирование и распределение заданий по кластеру, взаимодействует с системами хранения, такими, как **HDFS**, **Amazon S3** или **RDBMS**, а также управляет памятью и восстановлением после сбоя; RDD (**Resilient Distributed Dataset**) — это устойчивый распределенный набор данных, который представляет собой неизменяемую коллекцию; RDD разделены на сегменты для выполнения на разных узлах кластера; в следующем подразделе мы подробно на них остановимся.
- **Spark SQL**: модуль для запроса данных (хранящихся как в RDD, так и во внешних источниках) с помощью абстрактных интерфейсов, использование которых позволяет разработчикам сочетать команды SQL с инструментами аналитики.
- **Spark Streaming**: модуль для обработки данных в режиме реального времени, что очень важно для анализа потоков с низкой задержкой.
- **MLlib** (**Machine Learning Library**): используется для применения алгоритмов машинного обучения в Apache Spark.
- **GraphX**: этот модуль предоставляет API для параллельных вычислений на основе графов; он поставляется с различными алгоритмами построения графов; *Граф* — это математическая концепция, основанная на вершинах и ребрах, которая показывает, как объекты связаны или зависят друг от друга; объекты представлены вершинами, а их связи — ребрами.

Управление кластером

Apache Spark поддерживает несколько *менеджеров кластера* (**Cluster Manager**), таких, как **Standalone**, **Mesos**, **YARN** и **Kubernetes**. Их ключевой функцией является планирование и выполнение заданий на узлах кластера, а также управление ресурсами. Для взаимодействия с одним или несколькими менеджерами в основной программе или в программе-драйвере используется специальный объект `SparkSession`. До версии 2.0 точкой входа считался объект `SparkContext`, но сейчас его

API-интерфейс является частью объекта SparkSession. На следующей схеме (рис. 8.4) показано взаимодействие между менеджером кластера, `SparkSession` (`SparkContext`) и *рабочими узлами* **(Worker node)** в кластере:

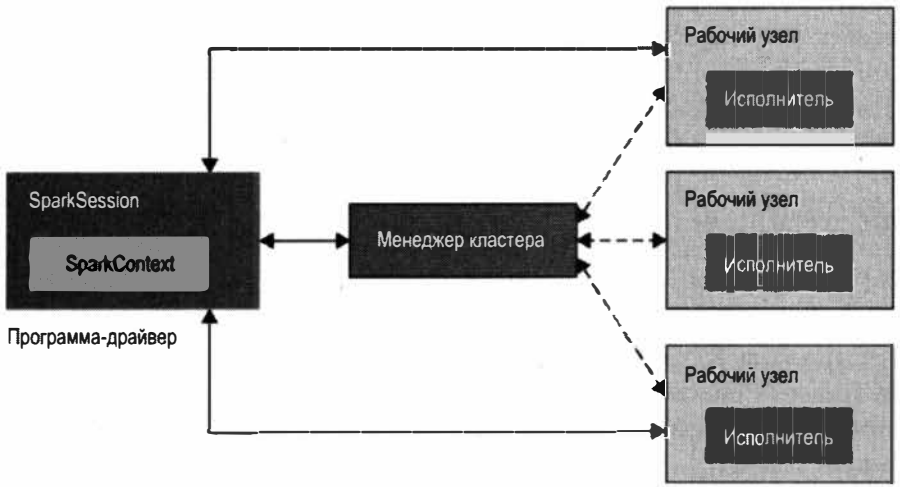


Рис. 8.4. Экосистема Apache Spark

Объект `SparkSession` может подключаться к различным типам менеджеров кластера. После подключения узлы кластера через менеджеров получают исполнителей. *Исполнители* — это процессы Spark, которые выполняют задания и хранят результаты работ. Менеджер кластера на главном узле отвечает за отправку кода приложения в процессы-исполнители на рабочем узле. Когда код и данные приложения перемещаются на рабочие узлы, объект `SparkSession` в программе-драйвере напрямую взаимодействует с процессами-исполнителями для выполнения задач.

В Apache Spark 3.1 поддерживаются следующие менеджеры кластеров:

1. **Standalone:** простой менеджер, входящий в состав Spark Core. Основан на главных (мастеры) и рабочих (подчиненные) процессах. Мастер является, по сути, менеджером кластера, а подчиненные размещают в себе исполнителей. Главные и рабочие процессы могут размещаться на одной машине, но в реальности так никто не делает. Для наилучшего результата рабочие процессы рекомендуется распределять по разным машинам. Standalone прост в настройке и предоставляет большинство необходимых кластеру возможностей.
2. **Apache Mesos**: еще один менеджер общего назначения с поддержкой Hadoop MapReduce. Он предпочтителен в крупных кластерных средах. Идея Apache Mesos заключается в объединении множества физических ресурсов в один виртуальный, который действует как кластер и обеспечивает абстракцию на уровне узла. Задуман как распределенный менеджер.
3. **Hadoop YARN**: менеджер кластера для Hadoop. По своей природе также является распределенным.

4. **Kubernetes**: менеджер находится на экспериментальной стадии. Его назначением является развертывание и масштабирования контейнерных приложений. Последний выпуск Apache Spark включает планировщик Kubernetes.

Перед завершением темы также стоит упомянуть фреймворк **Dask**, который представляет собой библиотеку с открытым исходным кодом, написанную на Python для параллельных вычислений. Dask работает напрямую с распределенными аппаратными платформами вроде Hadoop. Фреймворк использует проверенные библиотеки и проекты Python, такие, как **NumPy**, **Pandas** и **scikit-learn**. Dask меньше и легче, и может работать с кластерами малого и среднего размера. Тогда как Apache Spark поддерживает множество языков и рекомендуется для крупных кластеров.

Мы рассмотрели возможности кластеров для параллельных вычислений и можем перейти к главной структуре данных Apache Spark — устойчивым распределенным наборам данных.

Устойчивые распределенные наборы данных (RDD)

Устойчивые распределенные наборы данных (**Resilient Distributed Datasets**, **RDD**) — это основная структура данных в Apache Spark. Представляет собой не только распределенную коллекцию объектов, но и разделен таким образом, что каждый набор может обрабатываться на разных узлах кластера. Это делает RDD ключевым элементом распределенной обработки. Более того, он очень гибок с точки зрения отказоустойчивости и способен восстановить данные после сбоя. При создании объекта RDD главный узел реплицирует его на несколько исполнителей или рабочих узлов. Если исполнитель или рабочий узел выходит из строя, главный узел обнаруживает сбой и передает выполнение исполнителю на другом узле. Новый узел уже будет иметь копию объекта RDD и сможет немедленно начать выполнение. Данные, обработанные исходным узлом до сбоя, будут потеряны и заново обработаны новым исполнителем.

Далее рассмотрим две главные операции с RDD и узнаем, как создавать объект RDD из разных источников данных.

Операции с RDD

RDD является неизменяемым объектом, а, значит, после создания его нельзя изменить. Но с данными в нем можно выполнять два типа операций: *преобразования* (**Transformation**) и *действия* (**Action**).

Преобразования

Эти операции применяются к RDD и приводят к созданию нового объекта. Они принимают распределенный набор в качестве входных данных и создают один или

несколько наборов на выходе. Важно помнить, преобразования ленивы по своей природе. Это означает, что они будут выполняться, только когда над ними совершается действие. Для лучшего понимания концепции ленивых вычислений представим, что нам нужно преобразовать числовые данные в RDD, вычитая единицу из каждого элемента, а затем арифметически складывая (операция *действие*) все элементы в выходном RDD из этапа преобразования. Из-за ленивых вычислений операция преобразования не выполнится, пока не будет вызвана операция действия.

В Apache Spark доступно несколько встроенных функций преобразования. Самые распространенные из них:

- `map`: проходит по каждому элементу или строке объекта RDD и применяет заданную функцию к каждому из них;
- `filter`: фильтрует данные из исходного RDD и предоставляет новый RDD с отфильтрованными результатами;
- `union`: применяется к двум RDD, если они одного типа, и создает новый RDD, который является объединением входных RDD.

Действия

Действия — это вычислительные операции, применяемые к RDD. Их результаты должны быть возвращены в программу-драйвер (например, `SparkSession`). В Apache Spark доступно несколько встроенных функций-действий. Самые распространенные из них:

- `count`: возвращает количество элементов в RDD;
- `collect`: возвращает в программу-драйвер весь RDD;
- `reduce`: свертывает элементы в RDD; простой пример — операция сложения в наборе данных.

С полным списком преобразований и действий можно ознакомиться в официальной документации Apache Spark. Далее узнаем, как создать RDD.

Создание RDD

Существует три подхода к созданию RDD.

Распараллеливание коллекции

Это один из самых простых подходов к созданию RDD. В этом случае коллекция создается или загружается в программу, а затем передается методу `parallelize` объекта `SparkContext`. Этот вариант не используется за пределами разработки и тестирования, поскольку он требует наличия всего набора данных на одной машине, что очень неудобно при большом объеме информации.

Внешние наборы данных

Apache Spark поддерживает распределенные наборы данных из локальной файловой системы, HDFS, HBase или даже Amazon S3. При таком подходе данные загружаются напрямую из внешнего источника. Для объекта `SparkContext` доступны удобные методы загрузки всех типов данных в RDD. Например, метод `textFile` можно использовать для загрузки текстовых данных из локальных или удаленных источников с указанием соответствующего URL-адреса (например, `file://`, `hdfs://` или `s3n://`).

Загрузка из существующих RDD

RDD можно создавать с помощью операций преобразования. Это одно из главных отличий платформы от Hadoop MapReduce. Входной набор изменить нельзя, но можно создавать новые RDD из существующих. Несколько примеров такого создания с помощью функций `map` и `filter` мы уже видели.

Далее рассмотрим дополнительные детали с примерами кода, используя библиотеку PySpark.

PySpark для параллельной обработки данных

Как уже упоминалось, Apache Spark написан на Scala, что означает отсутствие нативной поддержки Python. Существует также большое сообщество аналитиков и специалистов по data science, которые предпочитают использовать Python благодаря большому выбору библиотек. Было бы неудобно переходить на другой язык только для распределенной обработки данных. Таким образом, интеграция Python с Apache Spark полезна не только сообществу data science, но и многим другим разработчикам, которые хотели бы внедрить Apache Spark без изучения нового языка программирования.

Сообщество Apache Spark разработало библиотеку **PySpark** для облегчения работы с платформой. Для работы кода Python с Apache Spark (написанным на Scala и Java), была разработана Java-библиотека — **Py4J**. Она идет в комплекте PySpark и позволяет коду Python взаимодействовать с объектами **JVM** (**Java Virtual Machine**). По этой причине перед установкой PySpark нужно установить JVM.

PySpark предлагает почти те же возможности и преимущества, что и Apache Spark. К ним относятся вычисления в оперативной памяти, возможность распараллелить рабочие нагрузки, использование ленивых вычислений и поддержка нескольких менеджеров кластера, например, Spark, YARN и Mesos.

Установка PySpark (и Apache Spark) выходит за рамки этой книги. В этой главе основное внимание будет уделяться использованию PySpark для раскрытия силы Apache Spark, а не их установке. Однако все же стоит упомянуть некоторые варианты установки и зависимости.

В Интернете доступно множество руководств по установке для любой версии Apache Spark/PySpark и различных целевых платформ (Linux, macOS и Windows). PySpark включен в официальный релиз Apache Spark, который можно загрузить с веб-сайта (*https://spark.apache.org/*). PySpark также доступен через утилиту `pip` от PyPI, которую можно использовать для локальной установки или для подключения к удаленному кластеру. Другой вариант установки PySpark — **Anaconda**, еще одна популярная система управления пакетами и средами. Если PySpark устанавливается вместе с Apache Spark, на целевой машине требуются следующие компоненты:

- JVM;
- Scala;
- Apache Spark.

Для примеров кода, которые будут обсуждаться позже, мы установили Apache Spark версии 3.1.1 на macOS с включенным PySpark. PySpark поставляется вместе с командной оболочкой **PySpark shell** — интерфейсом командной строки для PySpark API. При запуске PySpark shell автоматически инициализирует объекты `SparkSession` и `SparkContext`, которые можно использовать для взаимодействия с основным движком Apache Spark. На рис. 8.5 показана инициализация PySpark shell:

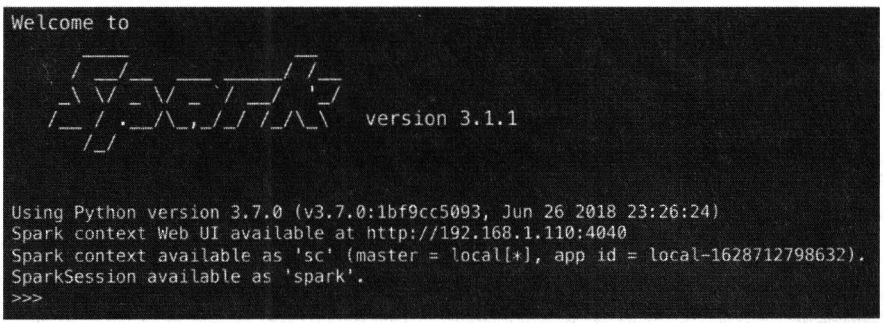


Рис. 8.5. PySpark shell

На этапах инициализации PySpark shell можно наблюдать следующее:

- Объект `SparkContext` уже создан, его экземпляр доступен в командной оболочке как `sc`.
- Также создан объект `SparkSession`, его экземпляр доступен как `spark`; теперь `SparkSession` является точкой входа в PySpark для динамического создания объектов RDD и DataFrame; объект `SparkSession` также может быть создан программно, и позже мы рассмотрим это на примере.
- Apache Spark имеет пользовательский веб-интерфейс и веб-сервер для его размещения, и запускается по адресу `http://192.168.1.110:4040` для локальной установки; обратите внимание, что IP-адрес в этом URL является частным и принадлежит нашему компьютеру; порт `4040` выбран как порт по умолчанию; если он занят, будет выбран следующий доступный порт, например, `4041` или `4042`.

В следующих подразделах мы узнаем, как создавать объекты SparkSession, изучим использование PySpark для операций с RDD и познакомимся с PySpark DataFrames и PySpark SQL. Для начала создадим сеанс Spark, используя Python.

Создание программ SparkSession и SparkContext

До выпуска Spark 2.0 SparkContext использовался как точка входа в PySpark. Начиная с версии 2.0 в качестве точки входа был введен объект SparkSession (для работы с RDD и DataFrame). SparkSession также включает все API, доступные в SparkContext, SQLContext, StreamingContext и HiveContext. Теперь SparkSession также может быть создан с помощью класса SparkSession, используя метод builder, как показано в следующем примере:

```
import pyspark
from pyspark.sql import SparkSession
spark1 = SparkSession.builder.master("local[2]")
    .appName('New App').getOrCreate()
```

Когда этот код выполняется в PySpark shell, в которой уже есть объект SparkSession по умолчанию (созданный как spark), он вернет тот же сеанс в качестве вывода метода builder. Вывод консоли показывает расположение двух объектов SparkSession (spark и spark1), это подтверждает, что они указывают на один объект SparkSession:

```
>>> spark
<pyspark.sql.session.SparkSession object at 0x1091019e8>
>>> spark1
<pyspark.sql.session.SparkSession object at 0x1091019e8>
```

Несколько ключевых понятий касательно метода builder:

- getOrCreate: метод гарантирует, что мы получим уже созданный сеанс в случае PySpark shell; если нового сеанса еще не существует, метод его создаст; в противном случае он вернет уже существующий сеанс;
- master: если нужно создать сеанс, подключенный к кластеру, следует указать имя главного процесса (master), которым может быть имя экземпляра менеджеров Spark, YARN или Mesos; если используется локально развернутый Apache Spark, можно использовать local[n], где n — целое число больше нуля; параметр n будет определять количество разделов, которые будут созданы для RDD и DataFrame; для локальной установки n может быть числом ядер ЦП; если установить для него значение local[*], что является распространенной практикой, это создаст столько рабочих потоков, сколько логических ядер есть в системе.

Если нужно создать новый объект SparkSession, можно использовать метод newSession, доступный на уровне экземпляра существующего объекта SparkSession.

Пример кода для создания нового объекта SparkSession:

```
import pyspark
from pyspark.sql import SparkSession
spark2 = spark.newSession()
```

Консольный вывод для объекта spark2 подтверждает, что это сеанс, отличный от ранее созданных объектов SparkSession:

```
>>> spark2
<pyspark.sql.session.SparkSession object at 0x10910df98>
```

Объект SparkContext также можно создать программно. Самый простой способ получить объект SparkContext из экземпляра SparkSession — использовать атрибут sparkContext. В библиотеке PySpark также есть класс SparkContext, который можно использовать для создания объекта SparkContext напрямую. Такой подход был распространен до Spark версии 2.0.

> ***Примечание***
>
> У нас может быть несколько объектов SparkSession, но только один объект SparkContext для каждой JVM.

Класс SparkSession предлагает еще несколько полезных методов и атрибутов:

- getActiveSession: метод возвращает активный SparkSession в текущем потоке Spark;
- createDataFrame: метод создает объект DataFrame из RDD, списка объектов или объекта pandas DataFrame;
- conf: атрибут возвращает интерфейс конфигурации для сеанса Spark;
- catalog: атрибут предоставляет интерфейс для создания, обновления или запроса связанных баз данных, функций и таблиц.

Полный список методов и атрибутов приводится в документации PySpark для класса SparkSession по адресу:

https://spark.apache.org/docs/latest/api/python/reference/api/.

PySpark для операций с RDD

В подразделе *«Устойчивые распределенные наборы данных (RDD)»* мы уже рассмотрели ключевые функции и операции с RDD. Далее мы расширим эти знания в контексте PySpark с помощью примеров.

Создание RDD из коллекций Python или внешнего файла

Некоторые способы создания RDD мы уже рассматривали. Теперь исследуем, как создавать их из коллекций Python в памяти или из внешнего файла.

Оба подхода описаны ниже:

1. Создать RDD из коллекции данных можно методом `parallelize`, доступным в экземпляре `sparkContext`. Он распределяет коллекцию для формирования объекта RDD. Метод принимает коллекцию в качестве параметра. Второй параметр (необязательный) задает количество создаваемых разделов. По умолчанию этот метод создает разделы по количеству ядер, доступных на локальном компьютере или заданных при создании объекта `SparkSession`.
2. Создать RDD из внешнего файла можно методом `textFile`, доступным в экземпляре `sparkContext`. Он может загрузить файл как RDD из HDFS или из локальной файловой системы (таким образом, он будет доступен на всех узлах кластера). Для локального развертывания на основе системы можно указать абсолютный и/или относительный путь. С помощью этого метода можно задать минимальное количество разделов, которые будут созданы для RDD.

Далее приведен пример (`rddcreate.py`) для демонстрации синтаксиса операторов PySpark, которые используются для создания нового RDD:

```
data = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7, 8, 4, 4, 8, 2, 7, 8, 9, 6, 9]
rdd1 = spark.sparkContext.parallelize(data)
print(rdd1.getNumPartitions())
rdd2 = spark.sparkContext.textFile('sample.txt')
print(rdd2.getNumPartitions())
```

Обратите внимание, файл `sample.txt` содержит произвольные текстовые данные, поскольку его содержимое неважно для нашего примера.

Операции преобразования RDD с использованием PySpark

PySpark предлагает несколько встроенных операций преобразования. Для демонстрации, как реализовать такую операцию, как `map`, возьмем текстовый файл в качестве входных данных и используем функцию `map`, доступную с RDD, с целью преобразовать его в другой RDD. Пример кода `rddtranform1.py` показан ниже:

```
rdd1 = spark.sparkContext.textFile('sample.txt')
rdd2 = rdd1.map(lambda lines: lines.lower())
rdd3 = rdd1.map(lambda lines: lines.upper())

print(rdd2.collect())
print(rdd3.collect())
```

В этом примере мы применили две лямбда-функции с операцией `map` для преобразования текста в объекте RDD в нижний и верхний регистры. В конце мы использовали операцию `collect` для получения содержимого объектов RDD.

Другой популярной операцией преобразования является `filter`, которую можно использовать для фильтрации некоторых записей.

Ниже приведен пример (rddtranform2.py), который отфильтровывает все четные числа из RDD:

```
data = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7, 8, 4, 4, 8, 2, 7, 8, 9, 6, 9]
rdd1 = spark.sparkContext.parallelize(data)
rdd2 = rdd1.filter(lambda x: x % 2 !=0 )
print(rdd2.collect())
```

При выполнении этого кода на консоль выводятся элементы 3, 5, 7 и 9. Далее рассмотрим несколько примеров с операциями действия с PySpark.

Операции действия с RDD с использованием PySpark

Для демонстрации операции используем RDD, созданный из коллекции, а затем применим несколько встроенных операций действия из библиотеки PySpark. Пример кода rddaction1.py:

```
data = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7, 8, 4, 4, 8, 2, 7, 8, 9, 6, 9]
rdd1 = spark.sparkContext.parallelize(data)

print("RDD contents with partitions:" + str(rdd1.glom().collect()))
print("Count by values: " +str(rdd1.countByValue()))
print("reduce function: " + str(rdd1.glom().collect()))
print("Sum of RDD contents:"+str(rdd1.sum()))
print("top: " + str(rdd1.top(5)))
print("count: " + str(rdd1.count()))
print("max: "+ str(rdd1.max()))
print("min" + str(rdd1.min()))

time.sleep(60)
```

Некоторые операции очевидны и не требуют подробностей (count, max, min и sum). Остальные действия, которые стоит пояснить, рассмотрим ниже:

- glom: приводит к созданию RDD путем объединения в список всех записей из каждого раздела;
- collect: метод возвращает все элементы RDD в виде списка;
- reduce: сокращает число элементов в RDD; в нашем случае мы использовали лямбда-функцию для объединения двух элементов в один и так далее; это приведет к добавлению всех элементов в RDD;
- top(x): возвращает первые x элементов массива, если они упорядочены.

Мы рассмотрели, как создавать RDD с помощью PySpark и реализовывать операции преобразования и действия с RDD. В следующем подразделе поговорим о **PySpark DataFrame**, еще одной популярной структуре данных, которая используется для аналитики.

PySpark DataFrames

PySpark DataFrame — это табличная структура данных, состоящая из строк и столбцов, подобно таблицам в реляционной базе данных и pandas DataFrame, которые мы рассматривали в главе 6 (*«Расширенные советы и приемы Python»*). Основное отличие от pandas DataFrame в том, что объекты PySpark DataFrames распределены в кластере, то есть хранятся на разных его узлах. Использование DataFrame в основном предназначено для распределенной обработки большого набора структурированных и неструктурированных данных, объем которых может измеряться петабайтами. Как и RDD, объекты PySpark DataFrame являются неизменяемыми и основаны на ленивых вычислениях (будут отложены, пока не понадобятся).

В DataFrame можно хранить как числовые, так и строковые данные. Столбцы не могут быть пустыми. Они должны содержать один тип данных и быть одинаковой длины. Строки могут содержать данные разных типов. Имена строк должны быть уникальными.

Далее узнаем, как создать DataFrame, и рассмотрим некоторые ключевые операции, используя PySpark.

Создание объекта DataFrame

PySpark DataFrame можно создать с помощью одного из следующих источников данных:

- Коллекции (списки, кортежи и словари).
- Файлы (CSV, XML, JSON, Parquet и т. д.).
- RDD с использованием методов `toDF` или `createDataFrame`.
- Потоковые сообщения **Apache Kafka** можно преобразовать в PySpark DataFrame с помощью метода `readStream` объекта `SparkSession`.
- Таблицы базы данных (например, **Hive** и **HBase**) можно запрашивать с помощью традиционных команд SQL, и выходные данные будут преобразованы в PySpark DataFrame.

Для начала создадим DataFrame из коллекции, что является самым простым и удобным для демонстрации подходом. Следующий фрагмент показывает, как создать PySpark DataFrame из набора данных о сотрудниках:

```
data = [('James','','Bylsma','HR','M',40000),
  ('Kamal','Rahim','','HR','M',41000),
  ('Robert','','Zaine','Finance','M',35000),
  ('Sophia','Anne','Richer','Finance','F',47000),
  ('John','Will','Brown','Engineering','F',65000)
]
```

```
columns = ["firstname","middlename","lastname","department","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
print(df.printSchema())
print(df.show())
```

Сначала мы создали строку данных в виде списка сотрудников, а затем создали схему с именами столбцов. Когда схема представляет собой список имен столбцов, тип каждого их них определяется данными в нем. Каждый столбец по умолчанию помечается как допускающий пустое значение. Для определения схемы DataFrame вручную можно использовать более сложный API (`StructType` или `StructField`), который включает установку типа данных и пометку столбца как допускающего или не допускающего пустое значение. В консольном выводе будет сначала показана схема, а затем содержимое DataFrame в виде таблицы:

```
root
  |-- firstname: string (nullable = true)
  |-- middlename: string (nullable = true)
  |-- lastname: string (nullable = true)
  |-- department: string (nullable = true)
  |-- gender: string (nullable = true)
  |-- salary: long (nullable = true)
+---------+----------+--------+-----------+------+-------+
|firstname|middlename|lastname| department|gender|salary|
+---------+----------+--------+-----------+------+-------+
|    James|          |  Bylsma|         HR|     M|  40000|
|    Kamal|     Rahim|        |         HR|     M|  41000|
|   Robert|          |   Zaine|    Finance|     M|  35000|
|   Sophia|      Anne|  Richer|    Finance|     F|  47000|
|     John|      Will|   Brown|Engineering|     F|  65000|
+---------+----------+--------+-----------+------+-------+
```

В следующем примере создадим DataFrame из CSV-файла, в котором будут те же записи, что и в предыдущем примере. В `dfcreate2.py` мы также определили схему вручную, используя объекты `StructType` и `StructField`:

```
schemas = StructType([ \
    StructField("firstname",StringType(),True), \
    StructField("middlename",StringType(),True), \
    StructField("lastname",StringType(),True), \
    StructField("department", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True) \
  ])
df = spark.read.csv('df2.csv', header=True, schema=schemas)
print(df.printSchema())
print(df.show())
```

Консольный вывод будет аналогичным выводу из предыдущего примера. Импорт JSON, XML и текстовых файлов в DataFrame поддерживается методом `read` с помощью схожего синтаксиса. Информацию о поддержке других источников, таких, как RDD и базы данных, вы можете изучить самостоятельно.

Работа с PySpark DataFrame

После создания DataFrame из некоторых данных, независимо от их источника, мы готовы проанализировать его, преобразовать и выполнить некоторые действия для получения требуемого результата. Большинство операций, поддерживаемых PySpark DataFrame, аналогичны RDD и pandas DataFrame. В целях демонстрации загрузим в объект DataFrame те же данные, что и в предыдущем примере, а затем выполним следующие операции:

1. Выберем один или несколько столбцов, используя метод `select`.
2. Заменим значения в столбце, используя словарь и метод `replace`. В библиотеке PySpark доступны и другие варианты замены данных в столбце.
3. Добавим новый столбец со значениями на основе данных имеющегося столбца.

Полный пример кода (`dfoperations.py`) показан ниже:

```
data = [('James','','Bylsma','HR','M',40000),
  ('Kamal','Rahim','','HR','M',41000),
  ('Robert','','Zaine','Finance','M',35000),
  ('Sophia','Anne','Richer','Finance','F',47000),
  ('John','Will','Brown','Engineering','F',65000)
]
columns = ["firstname","middlename","lastname",
           "department","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)

#отображаем 2 столбца
print(df.select([df.firstname, df.salary]).show())

заменяем значения в столбце
myDict = {'F':'Female','M':'Male'}
df2 = df.replace(myDict, subset=['gender'])

#добавляем новый столбец Pay Level на основе имеющегося столбца values
df3 = df2.withColumn("Pay Level", when((df2.salary < 40000), lit("10")) \
     .when((df.salary >= 40000) & (df.salary <= 50000), lit("11")) \
     .otherwise(lit("12")) \
  )
print(df3.show())
```

Консольный вывод будет следующим (рис. 8.6):

```
+---------+------+
|firstname|salary|
+---------+------+
|    James| 40000|
|    Kamal| 41000|
|   Robert| 35000|
|   Sophia| 47000|
|     John| 65000|
+---------+------+

+---------+----------+--------+-----------+------+------+---------+
|firstname|middlename|lastname| department|gender|salary|Pay Level|
+---------+----------+--------+-----------+------+------+---------+
|    James|          | Bylsma|         HR|  Male| 40000|       11|
|    Kamal|     Rahim|       |         HR|  Male| 41000|       11|
|   Robert|          |  Zaine|    Finance|  Male| 35000|       10|
|   Sophia|      Anne| Richer|    Finance|Female| 47000|       11|
|     John|      Will|  Brown|Engineering|Female| 65000|       12|
+---------+----------+--------+-----------+------+------+---------+
```

Рис. 8.6. Консольный вывод программы dfoperations.py

В первой таблице показан результат операции `select`. В следующей таблице показан результат операции `replace` для столбца `gender` и нового столбца `Pay Level`.

Для работы с PySpark DataFrame доступно множество встроенных операций. Многие из них аналогичны тем, которые мы обсуждали в pandas DataFrame. Более подробную информацию можно найти в официальной документации Apache Spark.

На этом этапе каждый может задать закономерный вопрос: *зачем использовать PySpark DataFrame, когда уже есть pandas DataFrame, предлагающий такие же операции*? Ответ прост: *PySpark предлагает распределенные объекты DataFrame*. И операции с такими объектами предназначены для параллельного выполнения в кластере узлов. Это делает производительность выше, чем у pandas DataFrame.

До этого момента мы наблюдали, что нам, как разработчикам, не нужно ничего программировать касательно делегирования распределенных объектов RDD и DataFrame различным исполнителям в автономном или распределенном кластере. Наше внимание сосредоточено только на аспекте написания кода для обработки данных. Координация и связь с локальным или удаленным кластером узлов автоматически обеспечивается `SparkSession` и `SparkContext`. В этом прелесть Apache Spark и PySpark: они позволяют нам сосредоточиться на решении реальных проблем и не беспокоиться о том, как будут выполняться рабочие нагрузки.

PySpark SQL

Spark SQL — это один из ключевых модулей Apache Spark, который используется для обработки структурированных данных и действует как механизм распределенных SQL-запросов. Spark SQL обладает хорошей масштабируемостью, поскольку является механизмом распределенной обработки. Обычно источником для Spark SQL выступает база данных, но SQL-запросы также можно выполнять к временным представлениям, которые можно создать из RDD и DataFrame.

Для демонстрации работы PySpark со Spark SQL будем использовать тот же DataFrame, что и в предыдущем примере, используя данные о сотрудниках для создания экземпляра `TempView` для SQL-запросов. В примере кода сделаем следующее:

1. Создадим PySpark DataFrame для данных о сотрудниках из коллекции Python, как в предыдущем примере.
2. Создадим экземпляр `TempView` из PySpark DataFrame, используя метод `createOrReplaceTempView`.
3. Используя метод `sql` объекта `SparkSession`, выполним стандартные SQL-запросы к экземпляру `TempView` вроде записей обо всех сотрудниках, сотрудниках с зарплатой более 45 000, количестве сотрудников по половому типу, а также использование SQL-команды `group by` для столбца `gender`.

Полный пример кода (`sql1.py`) выглядит следующим образом:

```
data = [('James','','Bylsma','HR','M',40000),
  ('Kamal','Rahim','','HR','M',41000),
  ('Robert','','Zaine','Finance','M',35000),
  ('Sophia','Anne','Richer','Finance','F',47000),
  ('John','Will','Brown','Engineering','F',65000)
]

columns = ["firstname","middlename","lastname",
           "department","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)

df.createOrReplaceTempView("EMP_DATA")

df2 = spark.sql("SELECT * FROM EMP_DATA")
print(df2.show())

df3 = spark.sql("SELECT firstname,middlename,lastname,
    salary FROM EMP_DATA WHERE SALARY > 45000")
print(df3.show())

df4 = spark.sql(("SELECT gender, count(*) from EMP_DATA
    group by gender"))
print(df4.show())
```

Консольный вывод всех трех SQL-запросов следующий:

```
+---------+----------+--------+-----------+------+------+
|firstname|middlename|lastname| department|gender|salary|
+---------+----------+--------+-----------+------+------+
|    James|          | Bylsma|         HR|     M| 40000|
|    Kamal|     Rahim|        |         HR|     M| 41000|
|   Robert|          |  Zaine|    Finance|     M| 35000|
|   Sophia|      Anne| Richer|    Finance|     F| 47000|
|     John|      Will|  Brown|Engineering|     F| 65000|
+---------+----------+--------+-----------+------+------+
+---------+----------+--------+------+
|firstname|middlename|lastname|salary|
+---------+----------+--------+------+
|   Sophia|      Anne| Richer| 47000|
|     John|      Will|  Brown| 65000|
+---------+----------+--------+------+
+------+--------+
|gender|count(1)|
+------+--------+
|     F|       2|
|     M|       3|
+------+--------+
```

Spark SQL — это обширная тема. Мы рассмотрели в общих чертах только введение в нее с целью показать возможности использования команд SQL поверх структур данных Spark без знания источника данных. На этом мы завершим обсуждение использования PySpark для обработки и анализа данных. В следующем подразделе рассмотрим несколько учебных примеров реальных приложений.

Практические примеры использования Apache Spark и PySpark

В предыдущем подразделе мы познакомились с основными концепциями и архитектурой Apache Spark и PySpark. Здесь мы рассмотрим два примера реализации приложений.

Пример 1: калькулятор числа π в Apache Spark

Мы будем рассчитывать число π с помощью кластера Apache Spark, который запущен на локальной машине. Значение π равно значению площади круга, если его радиус равен 1. Перед обсуждением алгоритма и программы-драйвера важно обговорить настройку кластера для этого примера.

Настройка кластера Apache Spark

До этого момента мы использовали PySpark, установленный локально на компьютере без кластера. Для примера настроим кластер с использованием нескольких виртуальных машин. Существует множество инструментов виртуализации (например, **VirtualBox**), любой из которых подойдет для настройки.

Мы использовали **Ubuntu Multipass** (*https://multipass.run/*) для создания виртуальных машин поверх macOS. Multipass работает как в Linux, так и в Windows. Это облегченный менеджер виртуализации, созданный специально для разработчиков, и позволяющий создавать виртуальные машины одной командой. Менеджер имеет мало команд, что упрощает его использование. Если вы решите его использовать, рекомендуем ознакомиться с официальной документацией по установке и конфигурации. С помощью Multipass в нашем примере были созданы следующие виртуальные машины (рис. 8.7):

Name	State	IPv4	Image
vm1	Running	192.168.64.2	Ubuntu 20.04 LTS
vm2	Running	192.168.64.3	Ubuntu 20.04 LTS
vm3	Running	192.168.64.4	Ubuntu 20.04 LTS

Рис. 8.7. Виртуальные машины, созданные для кластера Apache Spark

Мы установили Apache Spark 3.1.1 на каждой виртуальной машине, используя утилиту `apt-get`. Запустили Apache Spark как главный процесс на виртуальной машине `vm1`, а затем запустили Apache Spark как рабочий процесс на `vm2` и `vm3`, указав URI-адрес главного процесса. В нашем случае это `Spark://192.168.64.2.7077`. Полная настройка кластера выглядит следующим образом (рис. 8.8):

Node Name	Role	Web UI
192.168.64.2	Master (Spark://192.168.64.2:7077)	http://192.168.64.2:8080/
192.168.64.3	Worker	http://192.168.64.3:8081/
192.168.64.4	Worker	http://192.168.64.4:8081/

Рис. 8.8. Кластер Apache Spark

Пользовательский веб-интерфейс главного узла представлен на рис. 8.9.

Краткое описание веб-интерфейса главного узла будет следующим:

- Веб-интерфейс предоставляет имя узла с URL-адресом; в нашем случае мы использовали IP-адрес в качестве имени хоста, поэтому IP-адрес виден в URL.
- Есть 2 рабочих узла, каждый из которых использует 1 ядро ЦП и 1 ГБ памяти.
- Веб-интерфейс также предоставляет информацию о запущенных и завершенных приложениях.

Веб-интерфейс рабочих узлов показан на рис. 8.10.

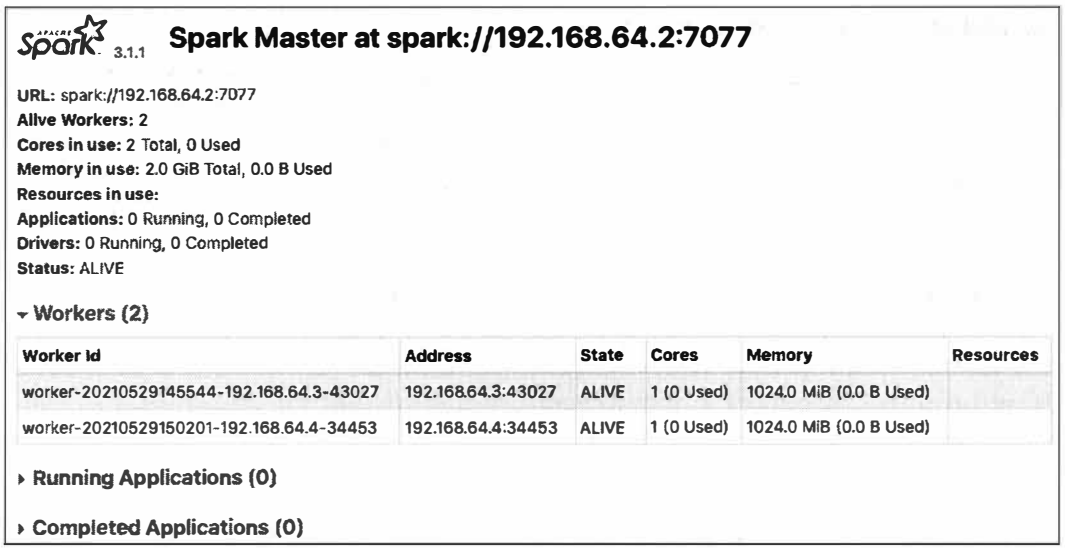


Рис. 8.9. Пользовательский веб-интерфейс главного узла в кластере Apache Spark

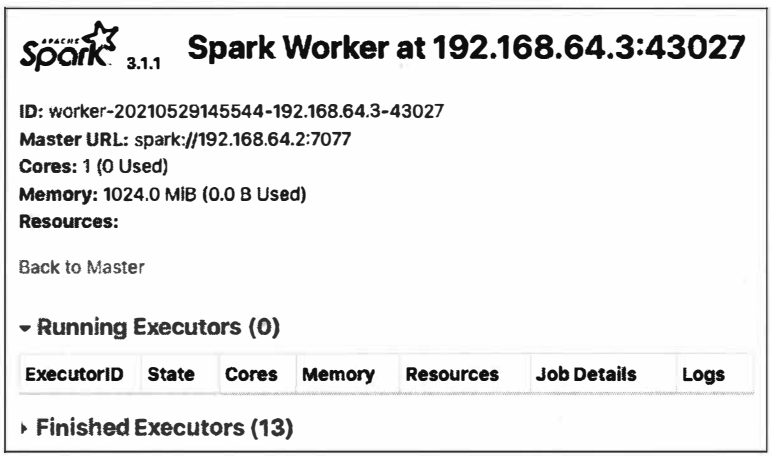


Рис. 8.10. Пользовательский веб-интерфейс рабочих узлов в кластере Apache Spark

Кратко подытожим про веб-интерфейс для рабочих узлов:

- Веб-интерфейс предоставляет идентификаторы рабочих процессов (ID), а также имена узлов и порты, на которых рабочие процессы прослушивают запросы.
- URL главного узла также предоставляется в веб-интерфейсе.
- Доступна информация о ядре ЦП и памяти, выделенной рабочим узлам.
- Веб-интерфейс предоставляет сведения о *выполняемых* (**Running Executor**) и уже *завершенных заданиях* (**Finished Executor**).

Написание программы-драйвера для расчета числа π

Для вычисления числа π мы применим популярный алгоритм (**Монте-Карло**), который предполагает, что квадрат имеет площадь 4, и в него вписан круг с радиусом 1. Идея состоит в генерации огромного количества случайных чисел в области квадрата. Внутри квадрата находится круг с тем же диаметром, что и длина стороны квадрата. Это означает, что круг вписан в квадрат. Значение π рассчитывается как соотношение числа точек внутри круга к общему числу точек.

Ниже приведен полный пример кода для программы-драйвера. Мы используем два раздела, поскольку мы имеем два рабочих узла. Для каждого из них зададим 10 000 000 точек. Также важно отметить, URL главного узла Spark указан в качестве атрибута `master` при создании сеанса:

```
#casestudy1.py: калькулятор Pi
from operator import add
from random import random

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("spark://192.168.64.2:7077") \
    .appName("Pi claculator app") \
    .getOrCreate()

partitions = 2
n = 10000000 * partitions

def func(_):
    x = random() * 2 - 1
    y = random() * 2 - 1
    return 1 if x ** 2 + y ** 2 <= 1 else 0

count = spark.sparkContext.parallelize(range(1, n + 1),
    partitions).map(func).reduce(add)
print("Pi is roughly %f" % (4.0 * count / n))
```

Консольный вывод будет следующим:

```
Pi is roughly 3.141479
```

Веб-интерфейс Spark предоставит статус приложения во время выполнения и после его завершения. На скриншоте, представленном на рис. 8.11, видно, что в выполнении задания участвовало два рабочих узла.

Можно кликнуть на имя приложения для перехода к следующему уровню детализации, как показано на рис. 8.12. На скриншоте видно, какие рабочие узлы участвуют в выполнении задач и какие ресурсы используются (если выполнение еще не завершено).

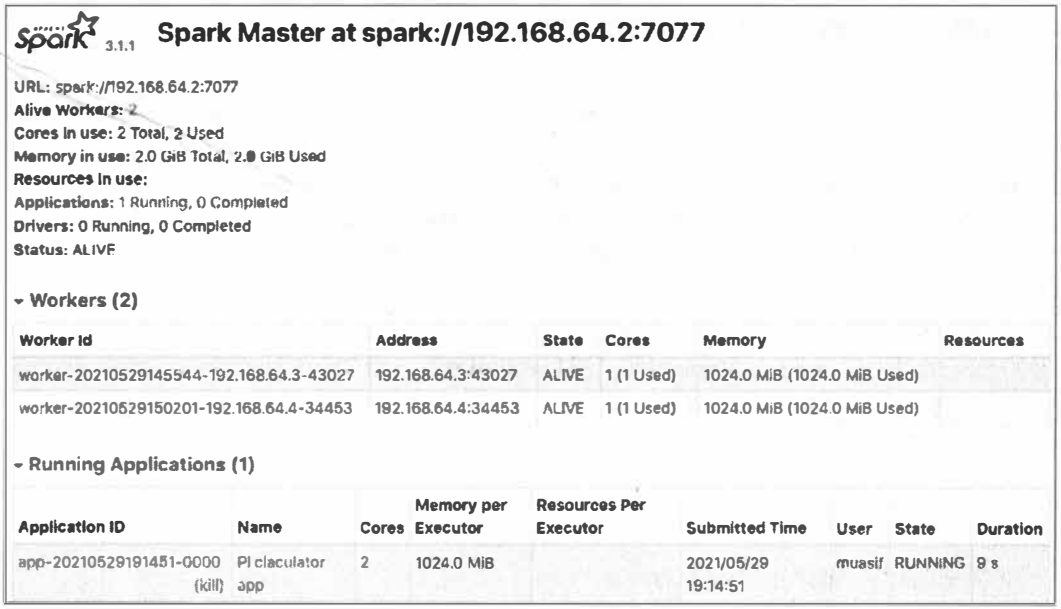


Рис. 8.11. Статус калькулятора числа π в пользовательском веб-интерфейсе Spark

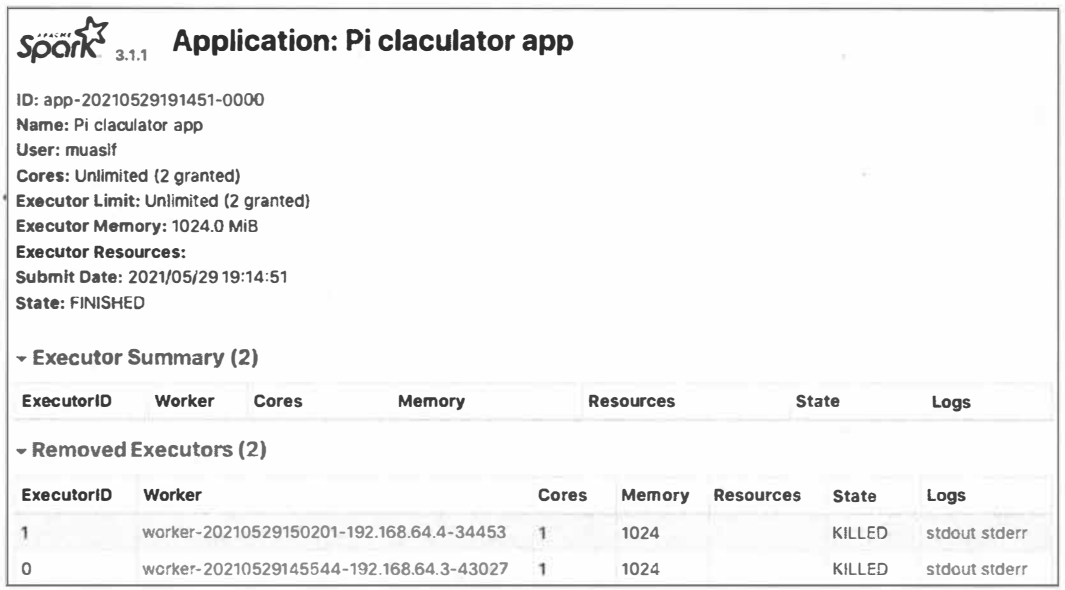


Рис. 8.12. Информация о калькуляторе π на уровне исполнителей

Мы рассмотрели, как возможно настроить кластер Apache Spark в целях тестирования и экспериментов, а также как создать программу-драйвер на Python с помощью библиотеки PySpark для подключения к Apache Spark и отправки задания на два разных узла кластера.

В следующем примере создадим облако слов с помощью библиотеки PySpark.

Пример 2: создание облака слов с помощью PySpark

Облако слов — это визуальное представление частоты слов, встречающихся в определенных текстовых данных. Проще говоря, чем чаще слово встречается в тексте, тем более крупным и жирным шрифтом оно выделяется в облаке слов. Такая визуализация также известна как *облако тегов* или *текстовое облако*. Это очень полезный инструмент, который помогает определить наиболее важные темы в тексте. Например, облако слов используется для анализа контента в социальных сетях в целях маркетинга, бизнес-аналитики и безопасности.

Для демонстрации напишем приложение для создания облака слов, которое читает текстовый файл из локальной файловой системы. Файл импортируется в объект RDD, который затем обрабатывается для подсчета количества повторений каждого слова. Затем идет обработка этих данных с целью отфильтровать слова, которые встречаются меньше двух раз, а также длина которых меньше четырех букв. Данные о частоте слов передаются объекту библиотеки `WordCloud`. Для отображения облака слов используется библиотека `matplotlib`. Полный пример кода показан ниже:

```
#casestudy2.py: приложения для подсчета слов
import matplotlib.pyplot as plt
from pyspark.sql import SparkSession
from wordcloud import WordCloud

spark = SparkSession.builder.master("local[*]")\
    .appName("word cloud app")\
    .getOrCreate()

wc_threshold = 1
wl_threshold = 3
textRDD = spark.sparkContext.textFile('wordcloud.txt',3)
flatRDD = textRDD.flatMap(lambda x: x.split(' '))
wcRDD = flatRDD.map(lambda word: (word, 1)).\
    reduceByKey(lambda v1, v2: v1 + v2)

#отфильтровываем слова с числом вхождений меньше порогового значения
filteredRDD = wcRDD.filter(lambda pair: pair[1] >= wc_threshold)
filteredRDD2 = filteredRDD.filter(lambda pair: len(pair[0]) > wl_threshold)

word_freq = dict(filteredRDD2.collect())
#создаем объект wordcloud
wordcloud = WordCloud(width=480, height=480, margin=0).\
    generate_from_frequencies(word_freq)

#выводи получившееся изображения облака слов
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```

Вывод программы отображается как оконное приложение и выглядит следующим образом (на основе текста из файла `wordcloud.txt`) (рис. 8.13):

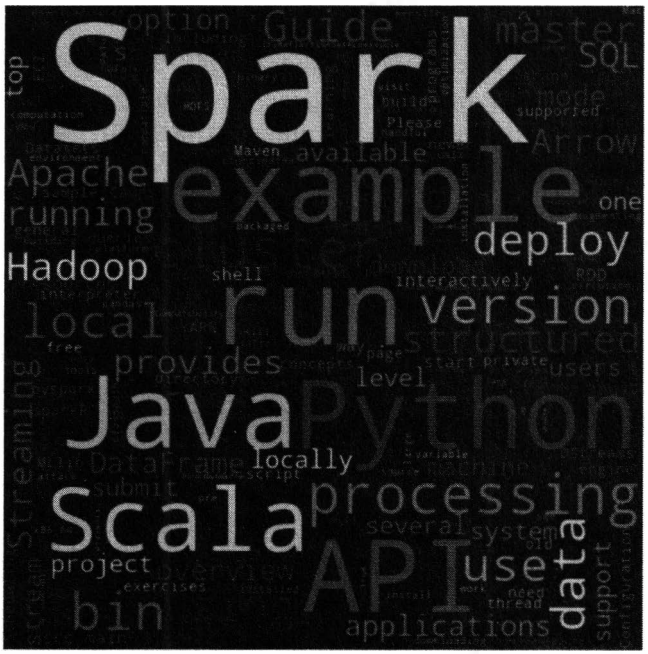


Рис. 8.13. Облако слов, созданное с помощью PySpark RDD

Обратите внимание, что в примере была использована не очень большая выборка текста. В реальном мире объем данных может быть огромным, поэтому использование кластера Apache Spark для обработки является оправданным.

Оба практических примера дают навыки использования Apache Spark для масштабной обработки данных. Они будут особенно полезны разработчикам, которые интересуются *обработкой естественного языка* (**Natural Language Processing, NLP**), анализом текста и анализом тональности (эмоционального окраса текста). Эти навыки важны для специалистов data science и всех, кто занимается обработкой данных для аналитики и построением алгоритмов для NLP.

Заключение

В этой главе мы рассмотрели параллельную обработку огромных объемов данных в кластере узлов. Сначала мы изучили различные возможности кластеров, доступные для обработки. Провели сравнительный анализ Hadoop MapReduce и Apache Spark. Это показало, что Apache Spark поддерживает больше языков и систем управления кластерами, а также превосходит Hadoop MapReduce при работе в режиме реального времени за счет обработки в оперативной памяти.

Рассмотрели главную структуру данных Apache Spark — RDD. Узнали, как создавать RDD из разных источников данных, и познакомились с двумя типами операций: преобразования и действия.

Основная часть главы была посвящена созданию RDD с помощью PySpark и управлению ими с помощью Python. Затем мы разобрали несколько примеров с операциями преобразования и действия. Познакомились с PySpark DataFrame для распределенной обработки данных. И завершили тему знакомством с PySpark SQL, используя несколько примеров.

В конце мы рассмотрели два реальных примера: калькулятор числа π и приложение для создания облака слов из текстовых данных.

Эта глава дает большой опыт в настройке Apache Spark как локально, так и с использованием виртуализации. Здесь также представлено множество примеров, которые помогут улучшить ваши практические навыки. Эти знания понадобятся всем, кто хочет решать задачи по обработке больших данных, используя кластеры для повышения эффективности и масштабирования.

В следующей главе мы рассмотрим варианты использования таких фреймворков, как **Apache Beam**, а также поговорим об использовании публичных облаков для обработки данных.

Вопросы

1. Чем Apache Spark отличается от Hadoop MapReduce?
2. Чем преобразования отличаются от действий в Apache Spark?
3. Что такое ленивые вычисления в Apache Spark?
4. Что такое `SparkSession`?
5. Чем PySpark DataFrame отличается от pandas DataFrame?

Дополнительные ресурсы

- «*Spark in Action*», второе издание, автор: Жан-Жорж Перрин (Jean-Georges Perrin).
- «*Learning PySpark*», авторы: Томаш Драбас (Tomasz Drabas), Денни Ли (Denny Lee).
- «*PySpark Recipes*», автор: Раджу Кумар Мишра (Raju Kumar Mishra).
- Документация по Apache Spark для вашего релиза (*https://spark.apache.org/docs/rel#*).
- Документация по Multipass (*https://multipass.run/docs*).

Ответы

1. Apache Spark обрабатывает данные в памяти, тогда как Hadoop MapReduce выполняет их чтение и запись в файловой системе.
2. Преобразование переводит данные из одной формы в другую, а результаты остаются в кластере. Действия — это функции, которые применяются к данным для получения результатов, возвращаемых программе-драйверу.
3. Ленивые вычисления применяются, в основном, к операциям преобразования. Это означает, что они не выполняются, пока не будет инициировано действие с объектом данных.
4. `SparkSession` — это точка входа в приложение Spark для подключения к одному или нескольким менеджерам кластера и работы с исполнителями для выполнения задач.
5. PySpark DataFrame является распределенным объектом и должен быть доступен на нескольких узлах кластера Apache Spark для параллельной обработки

9

Программирование на Python для облака

Облачные вычисления (**Cloud Computing**) — это обширное понятие, которое охватывает различные области применения, включая физические и виртуальные вычислительные платформы, инструменты разработки программного обеспечения, платформы обработки больших данных, хранилища, сетевые функции, программные сервисы и т. д. В этой главе мы рассмотрим использование Python для облачных вычислений в двух взаимосвязанных аспектах. Во-первых, изучим варианты использования Python при разработке приложения для облачных сред выполнения. Во-вторых, продолжим обсуждение обработки больших объемов данных, начатое в главе 8 (*«Масштабирование Python с помощью кластеров»*), используя облака вместо кластеров. Основное внимание будет уделено трем общедоступным облачным платформам: **Google Cloud Platform** (**GCP**), **Amazon Web Services** (**AWS**) и **Microsoft Azure**.

Темы этой главы:

- Знакомство с облачными возможностями для приложений Python.
- Создание веб-сервисов Python для облачного развертывания.
- Использование Google Cloud Platform для обработки данных.

К концу главы вы научитесь разрабатывать и развертывать приложения на облачной платформе, а также использовать Apache Beam в целом и для Google Cloud Platform в частности.

Технические требования

Для этой главы понадобится:

- Python 3.7 или более поздней версии.
- Аккаунт Google Cloud Platform (подойдет бесплатная версия).

- Google Cloud SDK, установленный на вашем компьютере.
- Apache Beam, установленный на вашем компьютере.

Примеры кода для этой главы можно найти по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter09.

Начнем с изучения возможностей облачных платформ, доступных при разработке приложений для облачных развертываний.

Знакомство с облачными возможностями для приложений Python

Сегодня облачные вычисления являются крайним рубежом программистов. В этом подразделе мы изучим, как создавать приложения, используя облачные среды разработки или специальный набор средств разработки для облачного развертывания — **SDK** (**Software Development Kit**). Также увидим, как выполнять код в облачной среде. В дополнение, рассмотрим различные опции в задачах с большими объемами данных (Apache Spark в облаке). Начнем с облачных сред разработки.

Среды разработки Python для облака

Для подготовки сред разработки в случае большой тройки общедоступных облаков, перечисленных выше, доступны две модели:

- Полностью облачная *интегрированная среда разработки* (**Integrated Development Environment**, **IDE**).
- Локально установленная **IDE** с возможностью интеграции с облаком.

Познакомимся с каждым вариантом поподробнее.

Облачная IDE

Рассмотрим наборы инструментов, которые могут предложить AWS, GCP и Azure. Существуют и другие облачные среды разработки. К ним относятся **PythonAnyWhere**, **Repl.it**, **Trinket** и **Codeanywhere**. Большинство из них предлагает как платные, так и бесплатные лицензии. Но мы остановимся на большой тройке:

1. **AWS** предлагает сложную облачную IDE AWS Cloud9, которая доступна через браузер. Она дает разработчикам широкие возможности и поддерживает несколько языков программирования, включая Python. Важно отметить, что Cloud9 поставляется как приложение, размещенное на виртуальной машине Amazon EC2. Сама IDE предоставляется бесплатно, но за использование базового экземпляра Amazon EC2 и хранилища может взиматься плата, причем весьма условная при ограниченном использовании. Платформа также предлагает инструмен-

ты для создания и тестирования кода при *непрерывной интеграции* (**CI**) и *непрерывной доставке* (**CD**).

2. AWS CodeBuild — еще один доступный сервис, который компилирует исходный код, выполняет тесты и собирает пакеты программного обеспечения для развертывания. Это сервер сборки, похожий на Bamboo. Вместе с AWS Cloud9 часто используется **AWS** CodeStar, который предлагает платформу на основе проектов для разработки, создания и развертывания ПО. CodeStar предлагает предопределенные шаблоны проектов для определения всей цепочки инструментов непрерывной поставки вплоть до выпуска кода.
3. **Microsoft Azure** предлагает встроенную среду Visual Studio, которая доступна в облаке, если вы используете платформу Azure DevOps. Онлайн-доступ к Visual Studio основан по платной подписке. Эта IDE известна своим богатым функционалом для совместной работы. Microsoft Azure предлагает *конвейеры Azure* (**Azure Pipelines**) для создания, тестирования и развертывания кода на любой платформе вроде Azure, AWS и GCP. Azure Pipelines поддерживают множество языков, включая Node.js, Python, Java, PHP, Ruby, C/C++, .NET и даже инструменты мобильной разработки.
4. **Google** предлагает Cloud Code для написания, тестирования и развертывания кода, который может быть написан как в вашем браузере (например, через AWS Cloud9), так и в любой локальной IDE. Cloud Code поставляется с плагинами для самых популярных IDE, например, IntelliJ IDE, Visual Studio Code и JetBrains PyCharm. Google Cloud Code доступен бесплатно и предназначен для сред запуска контейнеров. Как AWS CodeBuild и Azure Pipelines, Google также предлагает эквивалентный сервис, известный как Cloud Build, для непрерывного создания, тестирования и развертывания ПО в различных средах, включая виртуальные машины и контейнеры. Google также предлагает Colaboratory (или Google Colab), с удаленным использованием Jupyter Notebooks. Он популярен у специалистов data science.
5. Помимо прочего, Google Cloud предлагает сервисы Tekton и Jenkins для построения моделей разработки и поставки CI/CD.

В дополнение ко всем перечисленным инструментам и сервисам эти облачные платформы предлагают *среды-оболочки* (**Shell environment**), как онлайн, так и установленные локально. Это быстрый способ управления кодом, пусть и в условиях ограниченных возможностей.

Далее рассмотрим варианты локальных IDE для использования Python в облаке.

Локальная IDE для облачной разработки

Облачная среда разработки является отличным инструментом для нативных вариантов интеграции с остальной частью облачной экосистемы. Это позволяет удобно создавать экземпляры ресурсов по запросу и развертывать их, а также не требует никаких токенов аутентификации. Однако есть нюансы. Например, инструменты

могут быть бесплатными, но используемые ими ресурсы — нет. Кроме того, их доступность в офлайн-режиме не является идеальной. Разработчики любят писать код без привязки к онлайн, таким образом, они имеют возможность работать в поезде или парке.

Из-за подобных недостатков разработчики предпочитают сначала задействовать локальные редакторы или IDE, а затем уже используют дополнительные инструменты развертывания на облачных платформах. Среды Microsoft Azure (Visual Studio и Visual Studio Code) доступны на локальных машинах. Платформы AWS и Google предлагают свои собственные SDK и плагины для интеграции с любой IDE. Рассмотрим эти модели позже в этой главе.

Облачные среды выполнения для Python

Среда выполнения (**Runtime environment**) — это исполнительная платформа, на которой выполняется код. Самый простой способ сделать такую среду для Python — создать виртуальную машину Linux или контейнер с установленным Python. Версия языка может быть установлена по нашему выбору. Для интенсивной обработки больших объемов данных кластер Apache Spark можно настроить на вычислительных узлах в облаке. Но это налагает ответственность за задачи, связанные с обслуживанием платформы, на случай если что-то пойдет не так. Почти все публичные облачные платформы предлагают более элегантные решения, которые позволяют упростить работу разработчикам и ИТ-администраторам. Они дают уже готовые среды в зависимости от типов приложений. Мы обсудим несколько сред выполнения, предлагаемых Amazon AWS, GCP и Microsoft Azure.

Среды выполнения от Amazon AWS

Amazon AWS предлагает следующие варианты:

- **AWS Beanstalk**: предоставляется в виде «*платформа как услуга*» (**Platform-as-a-Service, PaaS**) и может использоваться для развертывания веб-приложений, разработанных на Java, .NET, PHP, Node.js, Python и многих других языках; этот сервис также предлагает возможность использования Apache, Nginx, Passenger или IIS в качестве веб-сервера; сервис обеспечивает гибкое управление базовой инфраструктурой, которая требуется для развертывания сложных приложений.
- **AWS App Runner**: используется для запуска контейнерных веб-приложений и микросервисов с API; это полностью управляемый сервис; таким образом, у вас нет административных обязанностей, а также нет доступа к базовой инфраструктуре.
- **AWS Lambda**: это бессерверная вычислительная среда, которая позволяет запускать код, не беспокоясь об управлении какими-либо базовыми серверами; сервис поддерживает множество языков, включая Python; несмотря на то что Lambda-код можно выполнять непосредственно из приложения, этот вариант

хорошо подходит для ситуаций, если необходимо выполнить определенный фрагмент кода, в случае когда другие сервисы AWS инициируют событие.

- **AWS Batch**: этот вариант используется для запуска вычислительных задач в больших объемах в виде пакетов; это альтернатива кластерам Apache Spark и Hadoop MapReduce.
- **AWS Kinesis**: также используется для обработки данных, но в потоковом виде и в режиме реального времени.

Среды выполнения от GCP

Ниже приведены варианты сред выполнения, предлагаемые GCP:

- **App Engine**: это вариант PaaS от GCP для разработки и размещения веб-приложений в большом масштабе; приложения развертываются как контейнеры, но исходный код упакован в них инструментом развертывания (эта сложность скрыта от разработчиков).
- **CloudRun**: используется для размещения любого кода, созданного в виде контейнера; контейнер-приложения должны иметь конечные точки HTTP для развертывания в CloudRun; в отличие от App Engine за упаковку приложений в контейнер отвечает сам разработчик.
- **Cloud Function**: это бессерверное и одноцелевое решение для размещения облегченного кода Python; код обычно запускается через прослушивание событий в других службах GCP или через прямые HTTP-запросы; похож на сервис AWS Lambda.
- **Dataflow**: еще один бессерверный вариант, но в основном предназначен для обработки данных с минимальной задержкой; упрощает жизнь специалистам data science, устраняя сложность базовой платформы обработки и предоставляя конвейер данных на основе Apache Beam.
- **Dataproc**: этот сервис предлагает вычислительную платформу на базе Apache Spark, Apache Flink, Presto и других инструментов; подходит для задач обработки данных, зависящих от экосистем Spark или Hadoop; сервис требует ручной подготовки кластеров.

Среды выполнения от Microsoft Azure

Microsoft Azure предлагает следующие варианты сред выполнения:

- **App Service**: используется для создания и развертывания веб-приложений в большом масштабе; веб-приложение можно развернуть как контейнер или запустить в Windows или Linux.
- **Azure Functions**: бессерверная среда выполнения, которой управляют события; используется для выполнения кода на основе определенного события или прямого запроса, как AWS Lambda и GCP CloudRun.

- **Batch**: используется для выполнения облачных задач, которым требуются сотни или тысячи виртуальных машин.
- **Azure Databricks**: Microsoft совместно с Databricks предлагает эту платформу на базе Apache Spark для обработки больших объемов данных.
- **Azure Data Factory**: бессерверный сервис для обработки потоковых данных и преобразования их в осмысленные результаты.

Как можем видеть, большая тройка облачных поставщиков предлагает различные среды выполнения в зависимости от доступных приложений и рабочих нагрузок. На облачных платформах можно развернуть следующие варианты использования:

- Разработка веб-сервисов и веб-приложений.
- Обработка данных с помощью облачной среды выполнения.
- Приложения на базе микросервисов (контейнеров) с использованием Python.
- Бессерверные функции или приложения для облака.

В следующих подразделах мы рассмотрим первые два варианта. Остальные обсудим в последующих главах, поскольку они требуют более детального обсуждения. Далее мы начнем создавать веб-сервис, используя Python, а также узнаем, как развернуть его в среде выполнения GCP App Engine.

Создание веб-сервисов Python для облачного развертывания

Разработка приложений для облака отличается от разработки для локальной среды. Существуют три ключевых требования, которые необходимо учитывать, при создании и развертывании облачного приложения:

1. **Веб-интерфейс**: для большинства облачных развертываний подходят приложения с *графическим пользовательским интерфейсом* (**Graphical User Interface**, **GUI**) или *программным интерфейсом приложения* (**Application Programming Interface**, **API**). Приложения с командной строкой не могут использовать облачную среду, если они не развернуты на выделенном экземпляре виртуальной машины, тогда мы можем запускать их с помощью SSH или Telnet. Поэтому в качестве примера мы выбрали приложение с веб-интерфейсом.
2. **Настройка среды**: все публичные облака поддерживают разные языки программирования и разные их версии. Например, GCP App Engine поддерживает Python 3.7, 3.8 и 3.9 (по состоянию на июнь 2021 года). Иногда облачные сервисы позволяют использовать свою версию для развертывания. Для веб-приложений также важно задать точку входа для доступа к коду и настройкам на уровне проекта. Обычно они определяются в одном файле (YAML в случае GCP App Engine).

3. **Управление зависимостями**: главная сложность, связанная с портативностью приложений, заключается в зависимостях от сторонних библиотек. Для приложений GCP App Engine мы документируем все зависимости в текстовом файле (`requirements.txt`) вручную или с помощью команды `PIP freeze`. Существуют и другие элегантные способы решить эту проблему. Например, упаковать все сторонние библиотеки с приложениями в один файл для облачного развертывания вроде веб-архива Java (файл `.war`). Другой подход заключается в объединении всех зависимостей, содержащих код приложения и целевую платформу, в контейнер с последующим его развертыванием непосредственно на хостинг-платформе. Развертывание на основе контейнеров мы рассмотрим в главе 11 (*«Python для разработки микросервисов»*).

Существует, как минимум, три варианта развернуть приложение веб-сервиса Python в GCP App Engine:

- Используя Google Cloud SDK через интерфейс CLI (командная строка).
- Используя веб-консоли GCP (портал) вместе с Cloud Shell (интерфейс CLI).
- Используя стороннюю IDE, например, PyCharm.

Подробно остановимся на первом варианте и кратко опишем два остальных.

> ***Важное примечание***
>
> Общие процессуальные действия для развертывания приложений Python в AWS и Azure являются одинаковыми, но детали будут различаться в зависимости от поддержки SDK и API, доступных у каждого облачного поставщика.

Использование Google Cloud SDK

Здесь мы обсудим, как использовать Google Cloud SDK для создания и развертывания примера. Приложение будет развернуто на платформе Google App Engine (**GAE**). Это платформа PaaS, которая лучше всего подходит для развертывания веб-приложений с использованием разных языков программирования, включая Python.

Для использования Google Cloud SDK необходимо выполнить предварительную подготовку на локальном компьютере:

- Установим и инициализируем Cloud SDK; после установки можно получить к нему доступ через CLI и проверить версию с помощью команды ниже. Обратите внимание, что почти все команды Cloud SDK начинаются с `gcloud`:

```
gcloud -version
```

- Установим компоненты Cloud SDK для добавления расширения App Engine для Python 3; это можно выполнить следующей командой:

```
gcloud components install app-engine-python
```

- GCP CloudBuild API должен быть включен для облачного проекта GCP.

- Также должен быть включен биллинг и привязан к вашему аккаунту для выставления платежных счетов, даже если используется бесплатный пробный аккаунт.
- Права пользователя GCP для настройки нового приложения App Engine и включения сервисов API должны быть предоставлены на уровне *владельца (***owner***)*.

Далее опишем, как настроить облачный проект GCP, создать веб-сервис и развернуть его в GAE.

Создание облачного проекта GCP

Концепция облачного проекта GCP аналогична большинству IDE. Он состоит из набора параметров, которые управляют взаимодействием кода с сервисами GCP и отслеживают используемые ресурсы. Проект должен быть связан с платежным аккаунтом (биллинг-аккаунтом). Это необходимое условие для отслеживания, сколько ресурсов и сервисов потребляется каждым проектом.

Далее рассмотрим, как создать проект с помощью Cloud SDK:

1. Авторизуемся в Cloud SDK, используя команду ниже. Если вход еще не выполнен, это перебросит нас в веб-браузер:

   ```
   gcloud init
   ```

2. Создаем новый проект с именем `time-wsproj`. Название проекта должно быть коротким и содержать только буквы и цифры. Для лучшей читаемости допускается использование дефиса (-):

   ```
   gcloud projects create time-wsproj
   ```

3. Переключаем область Cloud SDK по умолчанию на новый проект следующей командой (если это еще не сделано):

   ```
   gcloud config set project time-wsproj
   ```

4. Это позволит Cloud SDK использовать данный проект по умолчанию для любой команды, которую мы отправляем через CLI.

5. Создаем экземпляр App Engine в проекте по умолчанию или в любом другом, используя атрибут `project` с помощью одной из следующих команд:

   ```
   gcloud app create                    #для проекта по умолчанию
   gcloud app create --project=time-wsproj     #для конкретного проекта
   ```

6. Обратите внимание, что эта команда зарезервирует облачные ресурсы (в основном, вычислительные и хранилище) и предложит выбрать регион и зону для размещения ресурсов. Можно выбрать ближайшие к вам или наиболее подходящие с точки зрения вашей аудитории.

7. Включаем сервис Cloud Build API для текущего проекта. Как уже упоминалось, сервис Google Cloud Build используется для сборки приложения перед его развертыванием в среде выполнения Google вроде App Engine. Сервис Cloud Build API легче всего включить через веб-консоль GCP, поскольку это требует всего несколько кликов мышью. Для включения его с помощью Cloud SDK, нужно знать точное название сервиса. Список доступных сервисов можно получить с помощью команды `gcloud services list`.

8. Команда выдаст длинный список, в котором можно найти нужный сервис, связанный с Cloud Build. Можно также использовать атрибут `format` с любой командой для получения читаемого вывода Cloud SDK. Можно сделать еще удобнее, используя утилиту `grep` (для Linux или macOS) вместе с командой для фильтрации результатов, а затем включив сервис командой `enable`:

```
gcloud services list --available | grep cloudbuild
#Вывод будет таким - NAME: cloudbuild.googleapis.com
#Команда Cloud SDK для включения сервиса
gcloud services enable cloudbuild.googleapis.com
```

9. Для включения Cloud Billing API в проект сначала нужно связать проект с биллинг-аккаунтом. Поддержка таких аккаунтов в Cloud SDK пока невозможна для *общедоступной версии* (**General Availability**, **GA**) в релизе Cloud SDK 343.0.0. Привязать биллинг-аккаунт к проекту можно через веб-консоль GCP. Но существует также бета-версия команд Cloud SDK, дублирующая те же функции. Сначала нужно узнать идентификатор аккаунта, который будет использоваться. Информацию о биллинг-аккаунтах, связанных с авторизованным пользователем, можно получить с помощью команды `beta`:

```
gcloud beta billing accounts list
#Вывод будет включать следующую информацию
#billingAccounts/0140E8-51G144-2AB62E
#Включение биллинга для текущего проекта
gcloud beta billing projects link time-wsproj --billing-
account 0140E8-51G144-2AB62E
```

10. Обратите внимание, если вы впервые используете команды `beta`, вам будет предложено установить *beta-компонент*. Необходимо дать согласие на установку. Если вы уже используете версию Cloud SDK с биллинг-компонентом, включенным для GA, можно пропустить использование `beta` или использовать соответствующие команды в соответствии с документацией для вашего релиза Cloud SDK.

11. Включаем сервис Cloud Billing API для текущего проекта, выполнив те же действия, что и для Cloud Build API. Сначала нужно найти имя сервиса, а затем включить его следующим набором команд:

```
gcloud services list --available | grep cloudbilling
#Вывод будет следующим - NAME: cloudbilling.googleapis.com
#команда для включения сервиса
gcloud services enable cloudbilling.googleapis.com
```

Шаги по настройке проекта просты для опытного пользователя облачных сервисов и займут всего несколько минут. После создания можно узнать информацию о конфигурации следующей командой:

```
gcloud projects describe time-wsproj
```

В консольном выводе будет статус жизненного цикла, имя, идентификатор и номер проекта. Ниже приведен пример вывода:

```
createTime: '2021-06-05T12:03:31.039Z'
lifecycleState: ACTIVE
name: time-wsproj
projectId: time-wsproj
projectNumber: '539807460484'
```

Теперь, когда проект создан и настроен, можем приступать к разработке веб-приложения.

Создание приложения Python

Для облачных развертываний можно создать приложение с помощью IDE или системного редактора, а затем локально эмулировать среду выполнения App Engine с помощью Cloud SDK и компонента *app-engine-python*, который мы предварительно установили. В качестве примера создадим приложение на базе веб-сервиса, которое будет предоставлять время и дату через REST API. Приложение можно запускать через API-клиент или веб-браузер. Для упрощения примера мы не будем включать аутентификацию.

Создадим виртуальную среду с помощью пакета `venv`. Она будет использоваться в качестве обертки для интерпретатора, ядра, сторонних библиотек и скриптов, отделяя их как от системного, так и других виртуальных Python-окружений. Данный функционал поддерживается в Python, начиная с версии 3.3. Для создания виртуальных окружений существуют и другие инструменты, например, `virtualenv` и `pipenv`. PyPA рекомендует использовать `venv`, поэтому мы выбрали его для большинства примеров в этой книге.

Для начала создадим каталог проекта с именем `time-wsproj`, который содержит следующие файлы:

- `app.yaml`;
- `main.py`;
- `requirements.txt`.

Для удобства мы назвали каталог тем же именем, что и облачный проект, но это не обязательное действие. Рассмотрим файлы подробнее.

Файл `app.yaml` содержит параметры развертывания и среды выполнения, например, номер версии среды. В Python 3 файл `app.yaml` должен иметь, как минимум, параметр среды выполнения (runtime-параметр):

```
runtime: python38
```

Каждый сервис веб-приложения может иметь собственный YAML-файл, но для простоты мы будем использовать только один. В нашем случае он будет содержать только атрибут среды выполнения (runtime-атрибут). Мы добавили еще несколько атрибутов в файл YAML для иллюстрации.

Модуль `main.py` является точкой входа в приложение. Для примера мы выбрали библиотеку Flask. Она используется для веб-разработки и популярна, в основном, благодаря своей простоте и мощному функционалу. Мы рассмотрим ее более подробно в следующей главе. Полный код представлен ниже:

```
from flask import Flask
from datetime import date, datetime
# Если точка входа не определена в app.yaml, App Engine будет искать
# переменную app, как в нашем YAML-файле
app = Flask(__name__)
@app.route('/')
def welcome():
    return 'Welcome Python Geek! Use appropriate URI for date and time'

@app.route('/date')
def today():
    today = date.today()
    return "{date:" + today.strftime("%B %d, %Y") + '}'
@app.route('/time')
def time():
    now = datetime.now()
    return "{time:" + now.strftime("%H:%M:%S") + '}'

if __name__ == '__main__':
    # Для локального тестирования
    app.run(host='127.0.0.1', port=8080, debug=True)
```

Основные возможности этого модуля:

- Здесь определена точка входа по умолчанию, которая имеет имя `app`; она используется для перенаправления запросов, отправляемых этому модулю.
- С помощью аннотации Flask мы определили обработчики для трех URL-адресов:
  - Корневой URL (`/`) вызывает функцию `welcome`, которая возвращает приветственное сообщение в виде строки.
  - URL `/date` вызывает функцию `today`, которая возвращает текущую дату в формате JSON.
  - URL `/time` вызывает функцию `time`, которая возвращает текущее время в формате JSON.
- В конце модуля мы добавили функцию `__main__` для запуска локального веб-сервера, который поставляется с Flask в целях тестирования.

Файл `requirements.txt` содержит список зависимостей проекта от сторонних библиотек. С помощью его содержимого App Engine сделает необходимые библиотеки доступными приложению. В нашем случае потребуется библиотека Flask. Содержимое этого файла будет следующим:

```
Flask==2.0.1
```

Подготовив каталог проекта и описанные выше файлы, мы должны создать виртуальную среду внутри или вне каталога проекта и активировать его командой `source`:

```
python -m venv myenv
source myenv/bin/activate
```

После активации мы должны установить необходимые зависимости в соответствии с файлом `requirements.txt`. Используем утилиту `pip` из того же каталога, где находится `requirements.txt`:

```
pip install -r requirements.txt
```

После установки Flask и зависимостей структура каталога в PyCharm IDE будет выглядеть следующим образом (рис. 9.1):

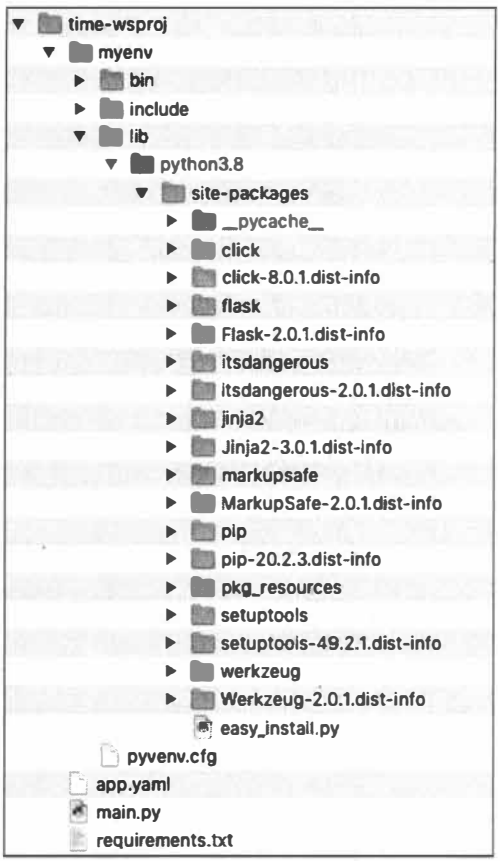


Рис. 9.1. Структура каталога для примера веб-приложения

После подготовки файла проекта и зависимостей локально запускаем веб-сервер следующей командой:

```
python main.py
```

Сервер запустится с отладочными сообщениями, из которых ясно, что сервер предназначен только для тестирования и не подходит в качестве рабочей среды:

```
* Serving Flask app 'main' (lazy loading)
* Environment: production
   WARNING: This is a development server. Do not use it in a
 production deployment.
   Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 668-656-035
```

Приложение веб-сервиса доступно по следующим URI:

- `http://localhost:8080/`;
- `http://localhost:8080/date`;
- `http://localhost:8080/time`.

Ответ от веб-серверов для этих URI будет следующий (рис. 9.2):

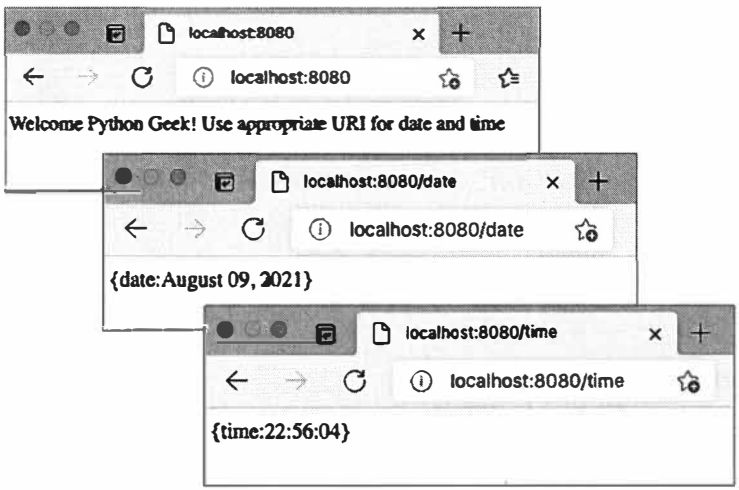


Рис. 9.2. Ответ от примера веб-сервиса в браузере

Веб-сервер будет остановлен прежде, чем мы перейдем к следующему этапу, а именно, к развертыванию приложения в Google App Engine.

Развертывание в Google App Engine

Развернуть приложение в GAE можно, выполнив следующую команду из каталога проекта:

```
gcloud app deploy
```

Cloud SDK прочитает файл `app.yaml`, который предоставляет входные данные для создания экземпляра App Engine. Во время развертывания создается образ контейнера с помощью Cloud Build, который затем загружается в хранилище GCP. После успешного развертывания можно получить доступ к веб-сервису следующей командой:

```
gcloud app browse
```

Команда откроет приложение, используя браузер по умолчанию на вашем компьютере. URL-адрес будет отличаться в зависимости от региона и зоны, выбранных при создании проекта.

Важно понимать, что при каждом выполнении команды `deploy` создается новая версия приложения в App Engine. Это означает, что будет потребляться больше ресурсов. Мы можем проверить версии, которые были установлены для веб-приложения, следующей командой:

```
gcloud app versions list
```

Старые версии приложения по-прежнему могут быть в состоянии обслуживания (URL-адреса будут немного изменены). Их можно остановить, запустить и удалить с помощью команды `gcloud app versions` в Cloud SDK и идентификатора версии. Приложение можно останавливать и запускать командами `stop` и `start`, как показано ниже:

```
gcloud app versions stop <version id>
gcloud app versions start <version id>
gcloud app versions delete <version id>
```

Идентификатор версии доступен, когда мы запускаем команду `gcloud app versions list`. На этом можно завершить обсуждение, как создавать и развертывать веб-приложение Python в Google Cloud. Далее мы обсудим, как развернуть то же приложение с помощью консоли GCP.

Использование веб-консоли GCP

Консоль GCP представляет собой простой в использовании веб-портал для доступа к проектам GCP и управления ими, а также онлайн-версию Google Cloud Shell. Консоль к тому же предлагает настраиваемые панели инструментов, отслеживание облачных ресурсов, потребляемых проектами, сведения о биллинге, логирование активности и многие другие возможности. Некоторые задачи можно выполнять непосредственно в веб-интерфейсе, но для большинства действий потребуется Cloud Shell. Это тот же Cloud SDK, но доступный онлайн в любом браузере.

Тем не менее Cloud Shell дает больше возможностей, чем Cloud SDK, по нескольким причинам:

- Он дает доступ к таким CLI, как `gcloud` и `kubectl`; последний используется для управления ресурсами в движке GCP Kubernetes.

- Он позволяет разрабатывать, делать отладку, создавать и развертывать приложения с использованием *редактора Cloud Shell* (**Cloud Shell Editor**).
- Cloud Shell также предлагает онлайн-сервер разработки для тестирования приложения перед его развертыванием в App Engine.
- Он обладает инструментами для обмена файлами между платформой Cloud Shell и локальным компьютером.
- Предусмотрена возможность предварительного просмотра веб-приложения через порт 8080 или любой другой по вашему выбору.

В Cloud Shell для создания нового проекта, сборки приложения и его развертывания в App Engine используются те же команды, что и в Cloud SDK. Поэтому это останется вам для самостоятельного изучения. Необходимо будет просто повторить инструкции из предыдущего подраздела. Обратите внимание, проект можно создать с помощью консоли GCP. Интерфейс Cloud Shell можно включить, нажав значок **Cloud Shell** в верхнем меню справа. После включения в нижней части вебстраницы консоли откроется интерфейс командной строки, как показано на скриншоте (рис. 9.3):

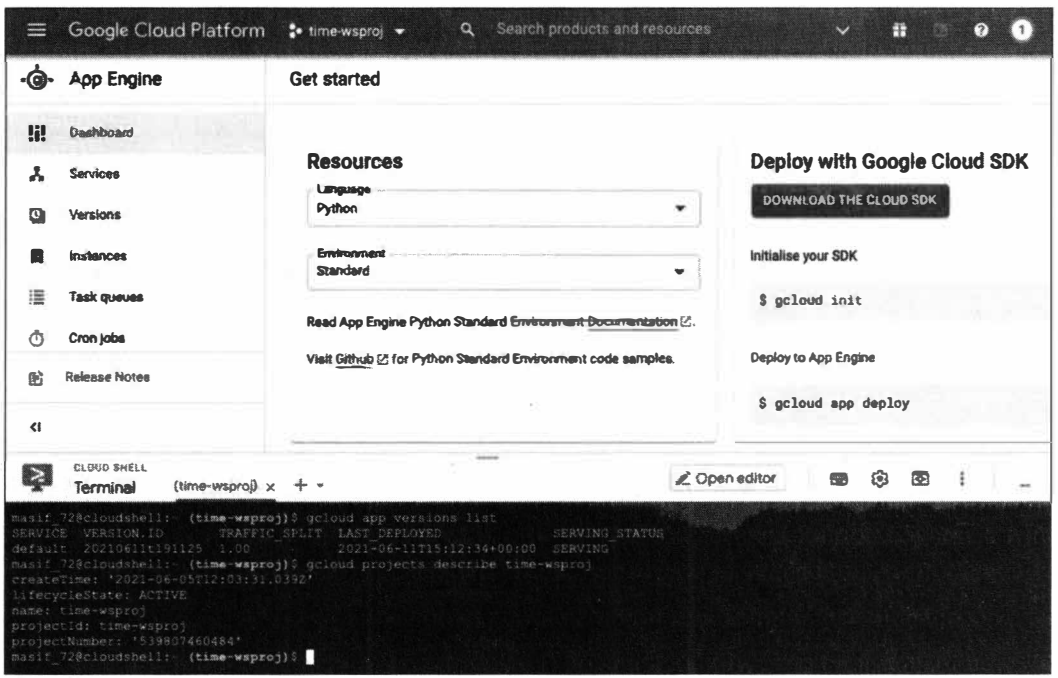


Рис. 9.3. Консоль GCP с Cloud Shell.

Как уже говорилось, Cloud Shell имеет инструмент редактирования, который можно запустить с помощью кнопки **Open editor**. На следующем скриншоте, приведенном на рис. 9.4, показан файл Python, открытый в редакторе Cloud Shell.

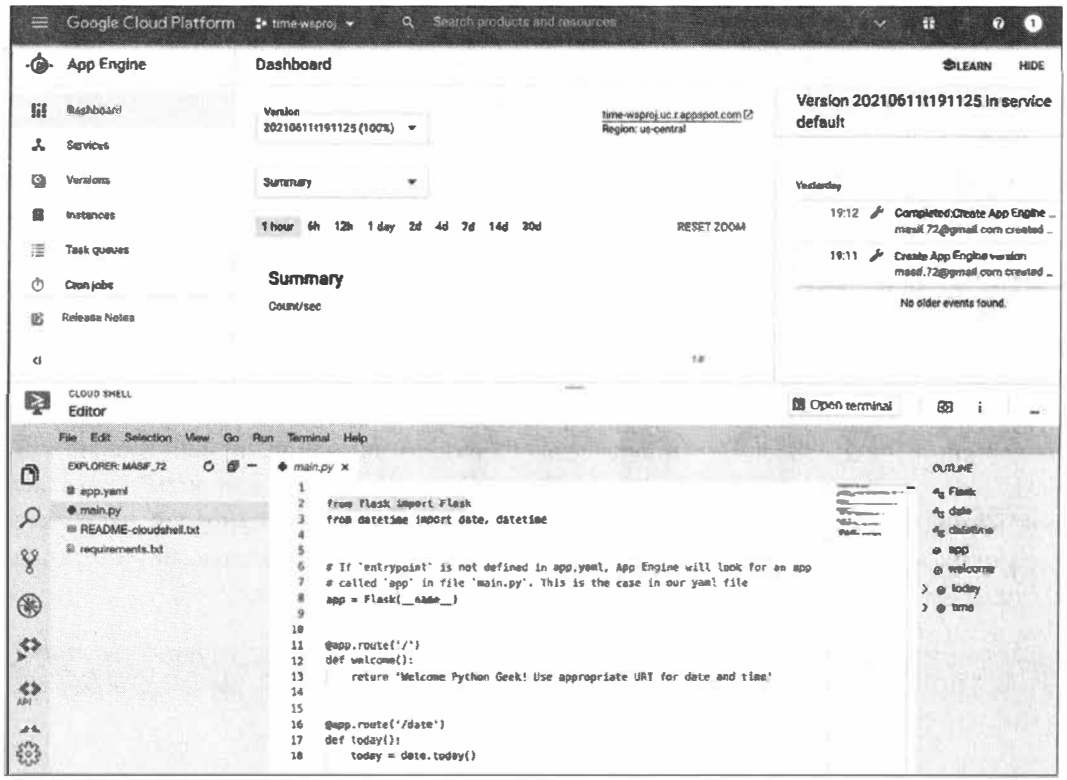


Рис. 9.4. Консоль GCP с редактором Cloud Shell

Создавать и развертывать веб-приложения можно, используя также сторонние IDE с подключаемыми плагинами Google App Engine. Как показывает практика, плагины для популярных IDE, вроде PyCharm и Eclipse, в основном, созданы для Python 2 и устаревших библиотек. Прямая интеграция IDE с GCP требует дополнительной работы и развития. На момент написания книги лучшим вариантом было использование Cloud SDK или Cloud Shell напрямую совместно с редактором или с IDE по вашему выбору.

Мы рассмотрели разработку веб-приложений с помощью Python и их развертывание на платформе GCP App Engine. Amazon для этого предлагает сервис AWS Beanstalk, в котором процесс развертывания почти совпадает с действиями для GCP App Engine, за исключением того, что Beanstalk не требует предварительной настройки проекта. Поэтому в решении от Amazon можно быстрее развертывать приложения.

Для развертывания нашего приложения в AWS Beanstalk мы должны предоставить следующую информацию, используя консоль AWS или AWS CLI:

- Имя приложения.
- Платформа (в нашем случае Python 3.8).
- Версия исходного кода.
- Исходный код вместе с файлом `requirements.txt`.

Мы рекомендуем использовать AWS CLI для веб-приложений, которые зависят от сторонних библиотек. Можно загрузить исходный код в виде ZIP-файла или веб-архива (файл `WAR`) с локального компьютера или скопировать его из хранилища Amazon S3.

Детальные инструкции по развертыванию в AWS Beanstalk доступны по адресу: *https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-flask.html*.

Azure для сборки и развертывания веб-приложений предлагает App Service. Инструкции по созданию и развертыванию в Azure доступны по адресу: *https://docs.microsoft.com/en-us/azure/app-service/quickstart-python*.

Далее мы рассмотрим создание программ-драйверов для обработки данных с помощью облачных платформ.

Использование Google Cloud Platform для обработки данных

Google Cloud Platform предлагает Cloud Dataflow для пакетной и потоковой обработки данных в режиме реального времени. Сервис позволяет создавать конвейеры обработки для data science и аналитики. Cloud Dataflow использует Apache Beam, который был разработан Google, но сейчас развивается как проект Apache с открытым исходным кодом. Он предлагает модель программирования для создания конвейеров обработки данных. Такие конвейеры можно создавать с помощью Apache Beam, а затем выполнять с помощью сервиса Cloud Dataflow.

Сервис Google Cloud Dataflow похож на Amazon Kinesis, Apache Storm, Apache Spark и Facebook Flux. Прежде чем перейти к обсуждению, как использовать Google Dataflow с Python, познакомимся с Apache Beam и его концепцией конвейера.

Введение в основы Apache Beam

Сегодня данные являются ценным активом для многих организаций. Приложения, устройства и люди генерируют огромные объемы информации, которые необходимо обрабатывать, прежде чем их можно будет использовать. В номенклатуре Apache Beam действия по обработке данных обычно называются *конвейерами*. Иными словами, это ряд действий, которые выполняются над необработанными данными из различных источников, после чего они поступают в место назначения для дальнейшего использования в аналитике и других бизнес-приложениях.

Apache Beam используется для разделения задачи на небольшие пакеты, которые можно обрабатывать параллельно. Один из основных вариантов применения сервиса заключается в операциях *извлечения, преобразования и загрузки* (**Extract, Transform, Load — ETL**). Три шага ETL лежат в основе конвейера, когда необходимо преобразовать необработанные данные в полезную информацию.

Ключевые концепции и компоненты описаны ниже:

- **Конвейер** — это схема преобразования данных из одной формы в другую в рамках процесса обработки.
- **PCollection (Parallel Collection)** — аналог RDD из Apache Spark; это распределенный набор данных, который содержит неизменяемую и неупорядоченную группу элементов; размер набора может быть фиксированным (ограниченным), подобно пакетной обработке, когда мы знаем, сколько заданий следует обрабатывать в одном пакете; размер также может быть гибким (неограниченным) в зависимости от постоянно меняющегося потокового источника данных.
- **PTransform** — операции по преобразованию данных, определенные в конвейере; они выполняются над объектами Pcollection.
- **SDK** — набор инструментов разработки ПО для конкретного языка (доступен для Java, Python и Go), с помощью которого мы создаем конвейеры и отправляем их *исполнителю* (**Runner**) для выполнения.
- **Runner** (исполнитель) — платформа выполнения для конвейеров Apache Beam; исполнитель обязан реализовать единственный метод — `run (Pipeline)`, который по умолчанию является асинхронным; несколько доступных исполнителей — Apache Flink, Apache Spark и Google Cloud Dataflow.
- **Пользовательские функции (user-defined functions, UDF)**: Apache Beam предлагает несколько типов; наиболее часто используемой является `DoFn`, которая работает поэлементно; предоставленная реализация `DoFn` обернута в объект `ParDo`, предназначенный для параллельного выполнения.

Простой конвейер выглядит следующим образом (рис. 9.5):

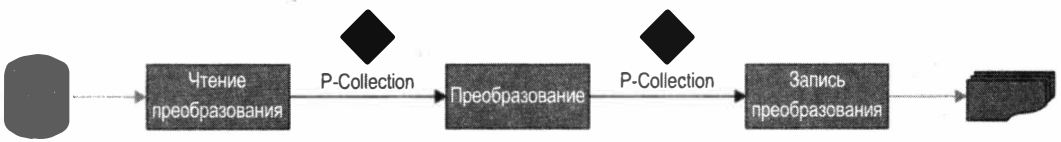


Рис. 9.5. Конвейер с тремя операциями PTransform

При проектировании конвейера обычно необходимо учитывать три элемента:

1. Сначала нужно понять источник данных: файл, база данных или поток. В зависимости от ответа мы определим, какой тип операции *Чтение преобразования* (**Read transform**) нужно реализовать. Нам также необходимо понимать формат и структуру данных: как часть операции чтения или как отдельную операцию.
2. Следующий шаг — определить и спроектировать, что делать с данными. Это наша основная операция *Преобразования* (**Transform**). Их может быть несколько последовательно или параллельно с одними и теми же данными. Apache Beam SDK предоставляет несколько предварительно созданных преобразований, которые можно использовать. Это также позволяет нам писать свои преобразования с использованием функций `ParDo/DoFn`.

3. Наконец, нужно определить, каким будет вывод конвейера и где хранить его результаты. На схеме выше этот шаг показан как операция *Запись преобразования* (**Write transform**).

Мы рассмотрели структуру простого конвейера для разъяснения концепций, связанных с Apache Beam. На практике конвейеры могут быть относительно сложными и включать несколько источников данных и выводов. Операции PTransform могут привести к созданию нескольких объектов PCollection, тогда возникнет необходимость выполнять их параллельно.

Далее мы изучим, как создать новый конвейер и выполнить его с помощью исполнителей Apache Beam или Cloud Dataflow.

Конвейеры Apache Beam

Как мы уже обсуждали, конвейер — это набор действий или операций для достижения определенных целей по обработке информации. Для этого требуется источник входных данных, например, память, локальные/облачные файлы или потоковые данные. Псевдокод типичного конвейера будет выглядеть следующим образом:

```
[Final PColletcion] = ([Initial Input PCollection] | [First
PTransform] | [Second PTransform] | [Third PTransform])
```

`Initial Input PCollection` используется как входные данные для операций PTransform. Выходная PCollection операции `First PTransform` будет использоваться как входные данные для `Second PTransform` и т. д. Итоговый вывод PCollection из последней операции PTransform будет записан как `Final PCollection` и использован для экспорта результатов в место назначения.

Для демонстрации создадим несколько конвейеров разного уровня сложности. Эти примеры предназначены показать роли различных компонентов и библиотек Apache, которые используются при создании и выполнении конвейера. В конце мы создадим конвейер приложения по подсчету слов, которое также упоминается в документации по Apache Beam и GCP Dataflow. Важно отметить, что мы должны установить библиотеку Python `apache-beam` с помощью утилиты `pip` для всех примеров кода в этой главе.

Пример 1: Создание конвейера со строковыми данными в оперативной памяти

В этом примере мы создадим входной объект PCollection из коллекции строк в оперативной памяти, применим пару операций преобразования, а затем выведем результаты на консоль. Ниже приведен полный код примера:

```
#pipeline1.py: отдельные строки из PCollection
import apache_beam as beam

with beam.Pipeline() as pipeline:
  subjects = (
```

```
    pipeline
    | 'Subjects' >> beam.Create(['English Maths Science French Arts',])
    | 'Split subjects' >> beam.FlatMap(str.split)
    | beam.Map(print))
```

На что нужно обратить внимание:

- Мы использовали символ «|» для записи различных операций PTransform в конвейере; это перегруженный оператор, который больше похож на применение PTransform к PCollection для создания другого объекта Pcollection.
- Мы использовали оператор «>>» для именования всех PTransform в целях логирования и отслеживания; строка между «|» и «>>» используется для отображения и логирования.
- Мы использовали три операции преобразования; все они являются частью библиотеки Apache Beam:
  - Первая создает объект PCollection, который представляет собой строку, содержащую пять названий учебных предметов.
  - Вторая разделяет строковые данные и создает новую PCollection с помощью встроенного метода `split` объекта `String`.
  - Третья выводит каждую запись PCollection на консоль.

Вывод консоли будет содержать список предметов, по одному в строке.

Пример 2: Создание и обработка конвейера с кортежем данных в оперативной памяти

В этом примере создадим PCollection из кортежей. Каждый кортеж будет содержать название учебного предмета и связанную с ним оценку. Основная операция PTransform в этом конвейере заключается в отделении предмета и оценки от данных. Пример кода:

```
#pipeline2.py: отделение предметов с оценками от PCollection
import apache_beam as beam

def my_format(sub, marks):
    yield '{}\t{}'.format(sub,marks)

with beam.Pipeline() as pipeline:
  plants = (
      pipeline
      | 'Subjects' >> beam.Create([
      ('English','A'),
      ('Maths', 'B+'),
      ('Science', 'A-'),
      ('French', 'A'),
```

```
    ('Arts', 'A+'),
    ])
    | 'Format subjects with marks' >> beam.FlatMapTuple(my_format)
    | beam.Map(print))
```

В отличие от первого примера здесь использована операция преобразования `FlatMapTuple` с пользовательской функцией для форматирования данных кортежа. Вывод консоли покажет название каждого предмета с оценкой в отдельной строке.

Пример 3: Создание конвейера из данных текстового файла

В первых двух примерах мы реализовали простой конвейер для анализа строковых данных из большой строки и кортежей из PCollection. На практике же мы чаще работаем с большим объемом информации, которая либо загружается из файла или хранилища, либо поступает в потоковом режиме. В этом примере будем использовать данные из локального текстового файла в исходном объекте PCollection, а затем выведем результаты в выходной файл. Полный пример будет выглядеть следующим образом:

```
#pipeline3.py: чтение данных из файла и запись результатов в другой файл
import apache_beam as beam
from apache_beam.io import WriteToText, ReadFromText

with beam.Pipeline() as pipeline:
    lines = pipeline | ReadFromText('sample1.txt')

    subjects = (
      lines
      | 'Subjects' >> beam.FlatMap(str.split))

    subjects | WriteToText(file_path_prefix='subjects',
                           file_name_suffix='.txt',
                           shard_name_template='')
```

Мы применили операцию PTransform для чтения текстовых данных из файла перед запуском любых PTransform, связанных с непосредственной обработкой. В конце применили PTransform для записи данных в выходной файл. В примере мы использовали две новые функции — `ReadFromText` и `WriteToText`:

1. `ReadFromText`: входит в модуль Apache Beam I/O и используется для чтения из текстовых файлов в PCollection, состоящую из строк. Путь к файлу или шаблон файла могут быть предоставлены в качестве входного аргумента для чтения по локальному пути. Также можно использовать `gs://` для доступа к любому файлу в хранилище GCS.
2. `WriteToText`: записывает PCollection в текстовый файл. Для этого, как минимум, требуется указать аргумент `file_path_prefix`. Также можно указать аргумент `file_path_suffix` и, таким образом, задать расширение файла. Для

`shard_name_template` задано пустое значение, это позволит создать файл с именем, используя аргументы префикса и суффикса. Apache Beam поддерживает шаблон имени *шарда* (**shard**) для определения имени файла на основе шаблона.

Когда конвейер выполняется локально, создается файл `subjects.txt` с названиями учебных предметов в соответствии с операцией PTransform.

Пример 4: Создание конвейера для исполнителя Apache Beam с аргументами

Итак, мы научились создавать простой конвейер, формировать объект PCollection из текстового файла и записывать результаты в новый файл. В дополнение к этим основным шагам нам нужно сделать еще несколько действий. После этого можно будет убедиться, что программа-драйвер готова отправить задание исполнителю GCP Dataflow или любому другому облачному исполнителю:

- В предыдущем примере мы указали имя входного и шаблон выходного файлов, заданные в программе-драйвере; на практике такие параметры передаются через аргументы командной строки; для *парсинга* (**Parsing**) и контроля аргументов командной строки мы будем использовать библиотеку `argparse`.
- Добавим дополнительные аргументы, например, для установки исполнителя; он будет устанавливать целевой исполнитель конвейера с помощью исполнителей DirectRunner или GCP Dataflow; обратите внимание, DirectRunner — это конвейерная среда выполнения для локального компьютера; она гарантирует, что конвейеры максимально соответствуют модели Apache Beam.
- Мы также реализуем функцию `ParDo`, которая будет использовать пользовательскую функцию для парсинга строк из текстовых данных; этого можно добиться, используя функции `String`, но здесь они были добавлены для демонстрации, как использовать `ParDo` и `DoFn` с PTransform.

Наши шаги будут следующими:

1. Сначала создадим *парсер* (**Parser**) аргументов и определим, какие из них мы ожидаем от командной строки. Установим для них значения по умолчанию и зададим дополнительный текст справки, которая будет отображаться по ключевому слову `help`. Атрибут `dest` важен, поскольку он используется для идентификации аргументов, которые будут использоваться в операторах. Также определим функцию `ParDo`, которая будет использоваться для выполнения конвейера. Пример кода представлен ниже:

```
#pipeline4.py(часть 1): использование аргумента для конвейера
import re, argparse, apache_beam as beam
from apache_beam.io import WriteToText, ReadFromText
from apache_beam.options.pipeline_options import PipelineOptions

class WordParsingDoFn(beam.DoFn):
  def process(self, element):
    return re.findall(r'[\w\']+', element, re.UNICODE)
```

```
def run(argv=None):
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--input',
        dest='input',
        default='sample1.txt',
        help='Input file to process.')

    parser.add_argument(
        '--output',
        dest='output',
        default='subjects',
        help='Output file to write results to.')

    parser.add_argument(
        '--extension',
        dest='ext',
        default='.txt',
        help='Output file extension to use.')

    known_args, pipeline_args = parser.parse_known_args(argv)
```

2. Теперь зададим `DirectRunner` в качестве среды выполнения конвейера и обозначим задание, которое нужно выполнить. Пример кода для этого шага выглядит так:

```
#pipeline4.py(part 2): с методом run
    pipeline_args.extend([
        '--runner=DirectRunner',
        '--job_name=demo-local-job',
    ])
    pipeline_options = PipelineOptions(pipeline_args)
```

3. Наконец, реализуем конвейер с помощью объекта `pipeline_options`, который мы создали на предыдущем шаге. Конвейер будет считывать данные из входного текстового файла, преобразовывать их в соответствии с функцией `ParDo`, а затем сохранять результаты в качестве вывода:

```
#pipeline4.py(part 3): с методом run

    with beam.Pipeline(options=pipeline_options) as pipeline:
        lines = pipeline | ReadFromText(known_args.input)
        subjects = (
                lines
                | 'Subjects' >> beam.ParDo(WordParsingDoFn()).
                with_output_types(str))
        subjects | WriteToText(known_args.output, known_args.ext)
```

4. Когда мы выполняем эту программу напрямую через IDE, используя значения по умолчанию для аргумента, или инициируем ее из интерфейса командной строки, используя следующую команду, мы получим тот же результат:

   ```
   python pipeline4.py --input sample1.txt --output myoutput --extension .txt
   ```

5. Здесь происходит парсинг слов из входного текстового файла (`sample1.txt`). Затем они записываются по одному в каждую строку выходного файла.

Apache Beam — обширная тема, поэтому невозможно охватить все его возможности в пределах одной главы. Тем не менее мы рассмотрели основы с примерами, позволяющие начать писать простые конвейеры, которые можно развернуть в GCP Cloud Dataflow. Это станет темой следующего подраздела.

Создание конвейеров для Cloud Dataflow

Примеры, которые обсуждалимь до сих пор, были сосредоточены на создании простых конвейеров и их выполнении с помощью DirectRunner. В этом подразделе мы создадим программу-драйвер и развернем конвейер для подсчета слов в Google Cloud Dataflow. Эта программа важна, поскольку мы задаем все параметры, связанные с облаком, внутри нее. Благодаря этому не придется использовать Cloud SDK или Cloud Shell для выполнения дополнительных команд.

Конвейер подсчета слов будет представлять собой расширенную версию примера `pipeline4.py`. Дополнительные компоненты и шаги для развертывания конвейера приведены далее:

1. Создадим новый облачный проект GCP, используя шаги, аналогичные тем, что мы выполняли в приложении веб-сервиса для развертывания App Engine. Для этой задачи можно использовать Cloud SDK, Cloud Shell или консоль GCP.
2. Включим Dataflow Engine API для нового проекта.
3. Затем создадим сегмент для хранения входного и выходного файлов, а также предоставим временные и промежуточные каталоги для Cloud Dataflow. Для этой задачи тоже можно использовать консоль GCP, Cloud Shell или Cloud SDK. В Cloud Shell или Cloud SDK сегмент можно создать следующей командой:

   ```
   gsutil mb gs://<bucket name>
   ```

4. Может потребоваться связать аккаунт сервиса с созданным сегментом, если он не находится в том же проекте, что и задача конвейера. В этом случае также нужно будет выбрать роль *администратора объекта хранилища* для контроля доступа.
5. Нужно установить Apache Beam с необходимыми библиотеками `gcp` с помощью утилиты `pip`:

   ```
   pip install apache-beam[gcp]
   ```

6. Необходимо создать ключ аутентификации для аккаунта сервиса, который используется для облачного проекта GCP. Этого не потребуется, если мы будем выполнять программу-драйвер на платформе GCP, например, Cloud Shell. Ключ

аккаунта должен быть загружен на локальной машине. Сделать ключ доступным для Apache Beam SDK можно через указание пути к файлу ключа (файл JSON) в переменной окружения `GOOGLE_APPLICATION_CREDENTIALS`.

Перед обсуждением, как выполнить конвейер в Cloud Dataflow, кратко рассмотрим пример программы-драйвера для подсчета слов. В этой программе мы определим аргументы командной строки, похожие на те, что мы использовали в предыдущем примере (`pipeline4.py`), но с некоторыми отличиями:

- Зададим переменную окружения `GOOGLE_APPLICATION_CREDENTIALS` не через операционную систему, а через программу-драйвер (для упрощения).
- Загрузим файл `sample.txt` в хранилище Google, в нашем случае это каталог `gs//muasif/input`; этот путь будет использоваться в качестве значения по умолчанию для аргумента `input`.

Полный пример кода выглядит так:

```
#wordcount.py(часть 1): подсчет слов в текстовом файле

import argparse, os, re, apache_beam as beam
from apache_beam.io import ReadFromText, WriteToText
from apache_beam.options.pipeline_options import PipelineOptions
from apache_beam.options.pipeline_options import SetupOptions

def run(argv=None, save_main_session=True):
  os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "some folder/key.json"
  parser = argparse.ArgumentParser()
  parser.add_argument(
      '--input',
      dest='input',
      default='gs://muasif/input/sample.txt',
      help='Input file to process.')
  parser.add_argument(
      '--output',
      dest='output',
      default='gs://muasif/input/result',
      help='Output file to write results to.')
  known_args, pipeline_args = parser.parse_known_args(argv)
```

Далее зададим расширенные аргументы для параметров конвейера, таким образом, он будет работать в среде выполнения Cloud Dataflow. Эти аргументы приведены далее:

- Платформа среды выполнения (исполнитель) для работы конвейера (в данном случае DataflowRunner).
- Идентификатор GCP Cloud Project.
- Регион GCP.

- Путь к сегментам хранилища Google для хранения входного, выходного и временных файлов.
- Имя задания для отслеживания.

На основе этих аргументов создадим объект параметров конвейера, который будет использоваться при выполнении. Фрагмент кода для этих задач будет выглядеть следующим образом:

```
#wordcount.py (часть 2): с методом run
  pipeline_args.extend([
      '--runner=DataflowRunner',
      '--project=word-count-316612',
      '--region=us-central1',
      '--staging_location=gs://muasif/staging',
      '--temp_location=gs://muasif/temp',
      '--job_name=my-wordcount-job',
  ])

  pipeline_options = PipelineOptions(pipeline_args)
  pipeline_options.view_as(SetupOptions).save_main_session = save_main_session
```

Наконец, реализуем конвейер с уже определенными параметрами и добавим наши операции PTransform. В этом примере мы реализовали дополнительную операцию PTransform для добавления в пару каждому слову единицы (1). В еще одной операции PTransform мы сгруппируем пары и применим операцию sum для подсчета их частоты. Так мы узнаем количество каждого слова в текстовом файле:

```
#wordcount.py (часть 3): с методом run
  with beam.Pipeline(options=pipeline_options) as p:

    lines = p | ReadFromText(known_args.input)
    #Считаем количеств вхождений каждого слова
    counts = (
        lines
        | 'Split words' >> (
            beam.FlatMap(
                lambda x: re.findall(r'[A-Za-z\']+', x)).
                with_output_types(str))
        | 'Pair with 1' >> beam.Map(lambda x: (x, 1))
        | 'Group & Sum' >> beam.CombinePerKey(sum))

    def format_result(word_count):
      (word, count) = word_count
      return '%s: %s' % (word, count)

    output = counts | 'Format' >> beam.Map(format_result)
    output | WriteToText(known_args.output)
```

Мы задаем значения по умолчанию для каждого аргумента в программе-драйвере. Это означает, что можно выполнить программу напрямую командой `python wordcount.py` или с помощью другой команды для передачи аргументов через CLI:

```
python wordcount.py \
    --project word-count-316612 \
    --region us-centrall \
    --input gs://muasif/input/sample.txt \
    --output gs://muasif/output/results \
    --runner DataflowRunner \
    --temp_location gs://muasif/temp \
    --staging_location gs://muasif/staging
```

Выходной файл содержит результаты и количество каждого слова. GCP Cloud Dataflow предоставляет дополнительные инструменты для мониторинга хода выполнения заданий, а также для оценки потребления ресурсов. На следующем скриншоте консоли GCP отображен список заданий, отправленных в Cloud Dataflow. В сводке отображаются их статусы и несколько ключевых метрик (рис. 9.6):

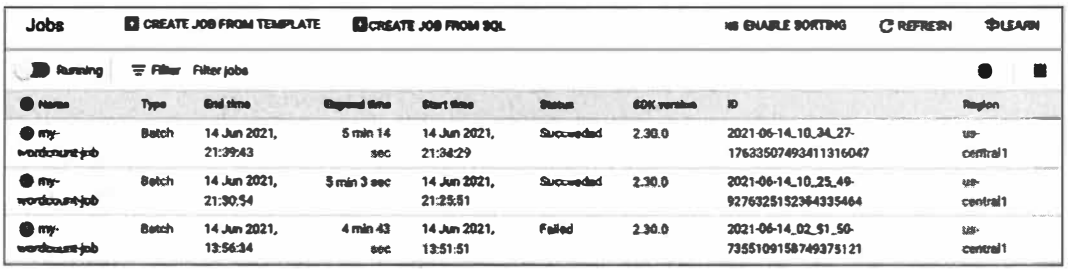


Рис. 9.6. Сводка по заданиям Cloud Dataflow

Можно перейти к подробной информации о любом задании (кликнув на его имя), как показано на следующем скриншоте. С правой стороны показаны сведения о задании и окружении, а также ход выполнения различных операций PTransform, которые мы определили для нашего конвейера. Пока задание выполняется, статус каждой операции PTransform обновляется в реальном времени, как показано на рис. 9.7.

Важно отметить, что операции PTransform называются в соответствии со строками, которые использованы с оператором «>>». Это помогает визуально выделить операции.

На этом можно завершить обсуждение, как создавать и развертывать конвейер для Google Dataflow. По сравнению с Apache Spark решение Apache Beam отличается большей гибкостью. Благодаря возможностям облачной обработки можно полностью сосредоточиться на моделировании конвейеров и не думать об их выполнении.

Как уже упоминалось, Amazon предлагает аналогичный сервис (AWS Kinesis) для развертывания и выполнения конвейеров, но он больше ориентирован на потоковые

данные в реальном времени. Как и AWS Beanstalk, AWS Kinesis не требует предварительной подготовки проекта. Руководство пользователя по обработке данных с помощью AWS Kinesis доступны по адресу: *https://docs.aws.amazon.com/kinesis/*.

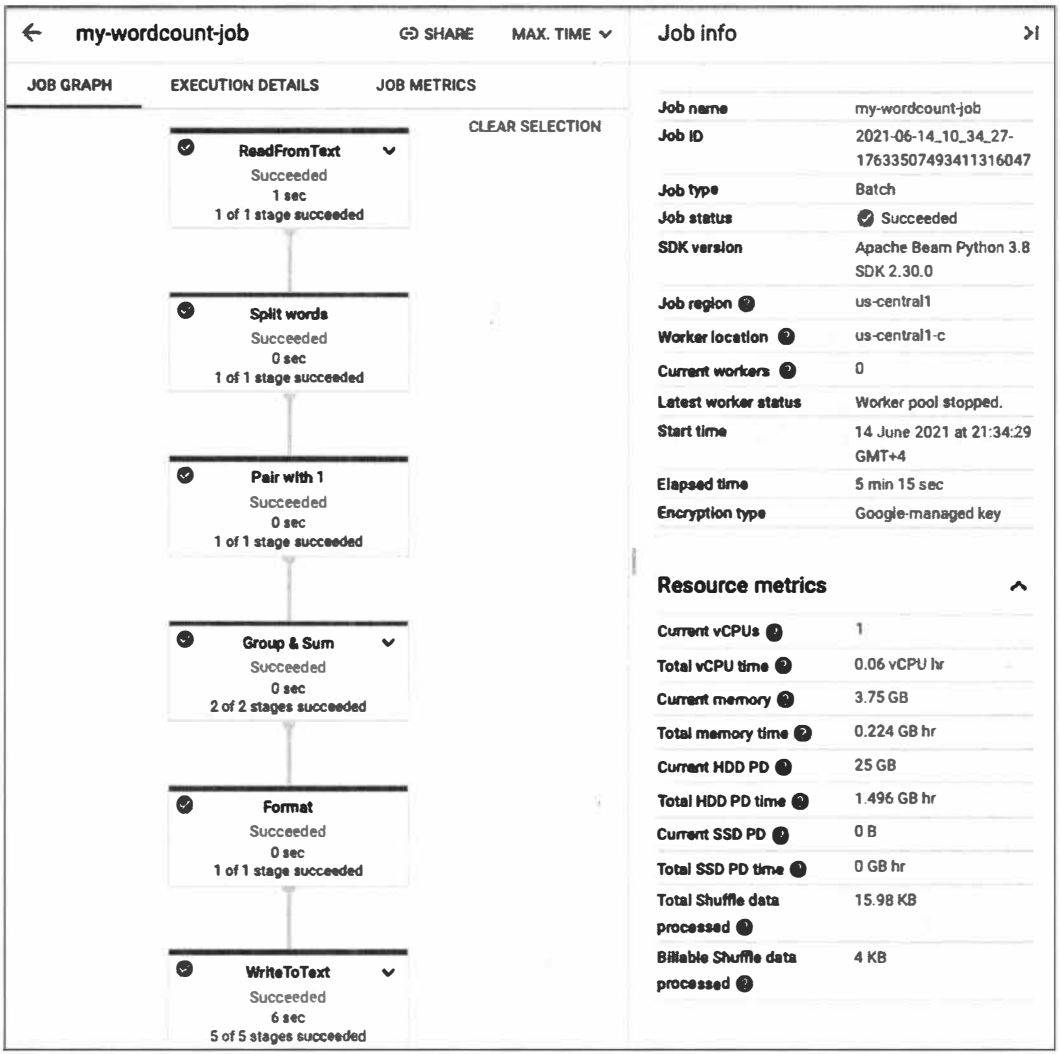


Рис. 9.7. Подробная информация о задании Cloud Dataflow со схемой и метриками

Заключение

В этой главе мы обсудили роль Python в разработке приложений для облачного развертывания в целом, а также использование Apache Beam для развертывания конвейеров обработки данных в Google Cloud Dataflow в частности. Мы начали со

знакомства с тремя крупными поставщиками облачных услуг и с продуктами разработки, которые они предлагают. Также познакомились с доступными вариантами сред выполнения каждого из них и сравнили их. Узнали, что существуют отдельные среды выполнения для классических веб-приложений, контейнерных приложений и бессерверных функций. На практике попытались изучить эффективность Python для облачных веб-приложений, создав пример и развернув его в Google App Engine с помощью Cloud SDK. В конце главы мы расширили свое понимание процесса обработки данных, начатое в предыдущей главе. Познакомились с конвейерами Apache Beam. На примерах научились создавать их для Cloud Dataflow.

Примеры кода в этой главе помогут вам приступить к созданию облачных проектов и написанию кода для Apache Beam. Эти знания понадобятся всем, кто хочет решать задачи обработки больших данных с помощью облачных сервисов.

В следующей главе мы рассмотрим возможности Python для разработки веб-приложений с использованием фреймворков Flask и Django.

Вопросы

1. Чем AWS Beanstalk отличается от AWS App Runner?
2. Что делает сервис GCP Cloud Function?
3. Какие сервисы предлагает GCP для обработки данных?
4. Что такое конвейер Apache Beam?
5. Зачем нужен объект PCollection в конвейере обработки данных?

Дополнительные ресурсы

- *«Разработка веб-приложений с использованием Flask на языке Python»* (Flask Web Development), автор: Мигель Гринберг (Miguel Grinberg).
- *«Advanced Guide to Python 3 Programming»*, автор: Джон Хант (John Hunt).
- *«Apache Beam: A Complete Guide»*, автор: Герардус Блокдик (Gerardus Blokdyk).
- *«Google Cloud Platform for Developers»*, авторы: Тэд Хантер (Ted Hunter) и Стивен Портер (Steven Porter).
- Документация по Google Cloud Dataflow: *https://cloud.google. com/dataflow/docs.*
- Документация по AWS Elastic Beanstalk: *https://docs.aws. amazon.com/elastic-beanstalk.*
- Документация по Службе приложений Azure: *https://docs.microsoft. com/en-us/azure/app-service/.*
- Документация по AWS Kinesis: *https://docs.aws.amazon.com/ kinesis/.*

Ответы

1. AWS Beanstalk — это универсальное решение PaaS для развертывания веб-приложений, а AWS App Runner — полностью управляемый сервис для развертывания веб-приложений на основе контейнеров.
2. GCP Cloud Function — это бессерверный сервис, управляемый событиями, для выполнения программы. Событие также может поступать от другого сервиса GCP или HTTP-запроса.
3. Cloud Dataflow и Cloud Dataproc — два популярных сервиса обработки данных от GCP.
4. Конвейер Apache Beam — это набор действий, определенных для загрузки, преобразования и записи данных.
5. PCollection похож на RDD в Apache Spark и содержит элементы данных. При обработке данных в конвейере операция PTransform принимает один или несколько объектов PCollection в качестве входных данных и выдает результаты в виде одного или нескольких объектов PCollection.

Раздел 4

Python для веб-разработки, облака и сети

Еще рано заканчивать наше путешествие, пока мы на практике не применили полученные знания. Это главная часть книги, где мы бросим вызов нашему прогрессу в обучении, решая реальные задачи. Во-первых, рассмотрим, как создавать веб-приложения и интерфейсы REST API с использованием фреймворка Flask. Во-вторых, подробно изучим реализацию бессерверных облачных приложений, используя архитектуру микросервисов и бессерверных функций. Коснемся как облачных, так и локальных вариантов развертывания. В конце рассмотрим использование Python для создания моделей *машинного обучения* (**Machine Learning**) и развертывания их в облаке. И завершим обсуждение, исследуя роль Python в автоматизации сети с реальными примерами.

Раздел состоит из следующих глав:

- «Глава 10: Использование Python для разработки веб-приложений и REST API».
- «Глава 11: Использование Python для разработки микросервисов».
- «Глава 12: Создание бессерверных функций с помощью Python».
- «Глава 13: Python и машинное обучение».
- «Глава 14: Использование Python для автоматизации сетей».

10

Использование Python для разработки веб-приложений и REST API

Веб-приложение — это программа, которая размещается и выполняется на сервере в локальной сети или Интернете и доступна через браузер на устройстве пользователя. Это позволяет получить доступ к приложению из любого места без установки дополнительного программного обеспечения на локальный компьютер. Благодаря такой простоте веб-приложения популярны уже более двух десятилетий. Они имеют широкий диапазон применения, будь то загрузка статического/динамического контента, интернет-магазины, онлайн-игры, социальные сети, обучение, поставка мультимедийного контента, опросы, блоги и даже сложные приложения для планирования ресурсов предприятия.

Веб-приложения по своей природе являются многоуровневыми (в основном трехуровневыми): пользовательский интерфейс, бизнес-логика и доступ к базе данных. Таким образом, они взаимодействует с веб-серверами (пользовательский интерфейс), сервером приложений (бизнес-логика) и системами управления базами данных (доступ к БД). С развитием мобильных технологий интерфейс стал чаще представлять собой мобильное приложение, которому требуется доступ к уровню бизнес-логики через REST API, наличие которого (или других интерфейсов) считается фундаментальным требованием для веб-приложений. В этой главе мы обсудим, как использовать Python для написания многоуровневых веб-приложений.

Для этого существует несколько фреймворков, но в рамках этой главы мы выбрали Flask за его функциональность и легкость. Важно понимать, что не стоит путать веб-приложения с мобильными приложениями для небольших устройств.

Темы этой главы:

- Требования к веб-разработке.
- Знакомство с фреймворком Flask.
- Взаимодействие с базами данных с помощью Python.

- Создание REST API с помощью Python.
- Пример создания веб-приложения с помощью REST API.

К концу главы вы научитесь использовать Flask для разработки веб-приложений, взаимодействия с БД и создания REST API и веб-сервисов.

Технические требования

Для этой главы понадобится:

- Python 3.7 или более поздней версии.
- Библиотека Python Flask 2.x с расширениями.

Пример кода к этой главе можно найти по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter10.

Начнем обсуждение с ключевых требований к разработке веб-приложений и REST API.

Требования к веб-разработке

Разработка веб-приложения включает в себя создание *пользовательского интерфейса* (**User Interface, UI**), маршрутизацию пользовательских запросов или действий к конечным точкам приложения, преобразование входных данных, написание бизнес-логики для запросов, взаимодействие с уровнем данных для чтения/записи информации и предоставление результатов пользователю. Для реализации всех этих компонентов могут потребоваться разные платформы и даже разные языки программирования. В этом подразделе мы рассмотрим, какие компоненты и инструменты требуются для веб-разработки, начиная с веб-фреймворков.

Веб-фреймворки

Разработка с нуля утомительна. Сделать ее удобной для программистов позволили фреймворки, существовавшие уже на ранних этапах зарождения веб-разработки. Они предоставляют набор библиотек, структуры каталогов, многоразовые компоненты и инструменты развертывания. Веб-фреймворки обычно следуют архитектуре, которая позволяет разработчикам создать сложное приложение за меньшее время и оптимальным способом.

Для Python существует несколько веб-фреймворков, из которых наиболее популярны **Flask** и **Django**. Оба предоставляются бесплатно и с открытым исходным кодом. Flask — легкий фреймворк со стандартными возможностями для создания веб-приложений, также имеет возможность использования дополнительных библиотек и расширений. Django — это фреймворк с полным стеком, который уже содержит

все необходимое «из коробки» и не требует дополнительных библиотек. Оба варианта имеют свои плюсы и минусы, но в конечном итоге приложение может быть полностью разработано с использованием любого из них.

Flask является лучшим выбором, когда мы хотим полностью контролировать приложение с возможностью использования сторонних библиотек по мере необходимости. Также это хороший вариант, если требования проекта часто меняются. Django полезен в ситуациях, когда нужны сразу все инструменты и библиотеки, и вы хотите сосредоточиться только на реализации бизнес-логики. Он хорошо подходит для масштабных проектов, но для небольших он может оказаться излишним. Использование Django требует предварительного изучения и опыта в веб-разработке. Если вы только начинаете в этой области, присмотритесь к Flask. А после того как освоитесь с ним, можете легко перейти на Django для создания более сложных проектов.

При создании веб- или любых других приложений с пользовательским интерфейсом мы часто сталкиваемся с понятием *«модель-представление-контроллер»* (**Model View Controller, MVC**). Это шаблон проектирования, который разделяет приложение на три уровня:

- **Модель**: этот уровень представляет собой данные, которые обычно хранятся в БД.
- **Представление**: этот уровень представляет собой пользовательский интерфейс, с которым взаимодействует пользователь.
- **Контроллер**: предназначен для обеспечения логики взаимодействия пользователя с приложением через пользовательский интерфейс; например, может возникнуть необходимость создать новый объект или обновить уже существующий; на уровне контроллера реализуется логика того, какой интерфейс (представление) дать пользователю для создания или обновления объекта с моделью (данными) или без нее.

Flask не обеспечивает прямой поддержки шаблонов проектирования MVC, но их можно реализовать с помощью кода. Django обеспечивает достаточно близкую реализацию MVC, но не полностью. Контроллер в Django управляется самим Django и недоступен для написания в нем собственного кода. Он и многие другие веб-фреймворки Python следуют шаблону проектирования *«модель-представление-шаблон»* (**Model View Template, MVT**). Он похож на MVC, за исключением уровня *шаблона*, который предоставляет специально отформатированные паттерны для создания ожидаемого пользовательского интерфейса с возможностью вставки динамического содержимого в HTML.

Пользовательский интерфейс

UI — это уровень представления приложения, который иногда включается как часть веб-фреймворков. Мы уделяем этому отдельное внимание с целью выделить

ключевые технологии и варианты, доступные для этого уровня. Прежде всего, пользователь взаимодействует через браузер, используя *язык гипертекстовой разметки* (**HyperText Markup Language**, **HTML**) и *каскадную таблицу стилей* (**Cascading Style Sheets**, **CSS**). Можно создавать интерфейсы напрямую на HTML и CSS, но это довольно утомительно и не подходит для быстрой доставки динамического контента. Существует несколько технологий, упрощающих создание UI:

1. **UI-фреймворки**: в основном это библиотеки HTML и CSS, которые предоставляют различные классы (стили) для создания пользовательского интерфейса. В них по-прежнему есть необходимость писать или генерировать ключевые HTML-части интерфейса, но без необходимости беспокоиться о внешнем виде страниц. Популярным примером является Bootstrap, построенный поверх CSS. Он был создан Twitter для собственного использования, но позже его исходный код был открыт для всех желающих. Еще один популярный вариант — ReactJS от Facebook, но это скорее библиотека, чем фреймворк.
2. **Шаблонизатор** (**Template engine**): еще один популярный механизм для динамического создания контента. Шаблон похож на определение желаемого результата, который содержит как статические данные, так и заглушки для динамического контента, которые представляют собой маркированные строки, заменяемые реальными значениями во время выполнения. Выходные данные могут иметь любой формат: HTML, XML, JSON или PDF. Один из самых популярных шаблонизаторов Python — Jinja2. Он также входит в состав фреймворка Flask. Django имеет собственный шаблонизатор.
3. **Скрипты на стороне клиента**: это программы, которые загружаются с веб-сервера и выполняются браузером. JavaScript — самый популярный язык для создания клиентских скриптов. Для него существует множество библиотек, упрощающих жизнь программистам.

Для разработки веб-интерфейсов можно использовать несколько технологий. В стандартном проекте все три варианта используются в разной степени.

Веб-сервер/сервер приложений

Веб-сервер — это программное обеспечение, которое прослушивает клиентские запросы через HTTP и предоставляет контент (веб-страницы, скрипты, изображения) в соответствии с типом запроса. Основная задача его — обслуживать только статические ресурсы. Он не может выполнять код.

Сервер приложений больше подходит для языка программирования. Его главная задача — предоставить доступ к реализации бизнес-логики, написанной с помощью языка программирования. В большинстве производственных сред веб-сервер и сервер приложений объединены в одно ПО для простоты развертывания. Flask имеет собственный веб-сервер **Werkzeug** для этапа проектирования, но в производственной среде его не рекомендуется использовать. Для этого есть более подходящие варианты: Gunicorn, uWSGI и движки от GCP.

База данных

Это не обязательный компонент, но практически необходимый для любого интерактивного веб-приложения. Python предлагает несколько библиотек для доступа к распространенным *системам управления базами данных* (**СУБД**), например, MySQL, MariaDB, PostgreSQL и Oracle. Язык также оснащен облегченным сервером БД — SQLite.

Безопасность

Безопасность имеет основополагающее значение, главным образом потому, что целевой аудиторией обычно являются пользователи Интернета, а конфиденциальность данных — главное требование в такой области. Протоколы **SSL** (**Secure Sockets Layer**) и недавно представленный **TLS** (**Transport Layer Security**) являются минимальными стандартами безопасности для защиты передачи данных между клиентами и сервером. Требования безопасности на транспортном уровне обычно выполняются веб-сервером, а иногда и прокси-сервером. Безопасность на уровне пользователей — следующее фундаментальное требование безопасности, минимальной необходимостью которого являются *имя пользователя* и *пароль*. Безопасность реализуется на уровне приложения, и ответственность за ее реализацию несут разработчики.

API

Уровень бизнес-логики в веб-приложениях может использоваться дополнительными клиентами. Например, мобильное приложение может использовать одну и ту же бизнес-логику для похожего набора возможностей. В сфере **Business to Business** (**B2B**) удаленное приложение может напрямую отправлять запросы на уровень бизнес-логики. Все это возможно, если мы предоставим стандартные интерфейсы, такие, как REST API. Сейчас реализация доступа через API является лучшей практикой, позволяющей сделать API готовым для использования с самого начала.

Документация

Документация также важна, как и написание программного кода. Когда мы говорим, что у нас есть API для приложения, то его потребители в первую очередь задаются вопросом, можем ли мы поделиться документацией по нему. Самый удобный способ создания API-документации — использовать встроенные инструменты, которые поставляются с веб-фреймворками. **Swagger** — популярный инструмент для автоматического создания документации из комментариев, добавленных при написании кода.

Итак, мы обсудили ключевые требования и в следующем подразделе можем углубиться в разработку веб-приложения с помощью Flask.

Знакомство с фреймворком Flask

Flask — это микрофреймворк для веб-разработки на Python. Термин «микро» указывает, что ядро Flask имеет облегченную конструкцию, но с возможностью расширения. Простой пример — взаимодействие с СУБД. Для сравнения, Django в своем составе имеет библиотеки для взаимодействия с самыми популярными базами данных. Flask, в свою очередь, для достижения тех же целей позволяет использовать расширения в зависимости от типа БД или подхода к интеграции. Философия Flask заключается в использовании *соглашений о конфигурации* (**Convention over configuration**). Это означает, если мы будем следовать стандартным соглашениям веб-разработки, мы потратим меньше усилий на настройку. Поэтому Flask является лучшим выбором для новичков, изучающих веб-разработку на Python. Он также позволяет пошагово внедрять различные концепции разработки.

В этом подразделе мы изучим следующие аспекты:

- Создание базового веб-приложения с маршрутизацией.
- Обработка запросов с разными типами HTTP-методов.
- Отображение статического и динамического контента с помощью Jinja2.
- Извлечение аргументов из HTTP-запроса.
- Взаимодействие с СУБД.
- Обработка ошибок и исключений.

Прежде чем перейти к примерам, нужно установить Flask 2.x в виртуальной среде. Начнем с создания базового веб-приложения.

Создание базового веб-приложения с маршрутизацией

В предыдущей главе (*«Программирование на Python для облака»*) мы уже использовали Flask при создании простого приложения для развертывания в GCP App Engine. Освежим наши знания и начнем с понимания, как строится веб-приложение и как в нем работает маршрутизация. Код выглядит следующим образом:

```
#app1.py: маршрутизация в приложении Flask
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello World!'
@app.route('/greeting')
def greeting():
    return 'Greetings from Flask web app!'

if __name__ == '__main__':
    app.run()
```

Пошагово проанализируем пример:

1. **Инициализация**: сначала мы должны создать экземпляр приложения (в нашем случае `app`). Веб-сервер будет передавать этому экземпляру все запросы от клиентов с помощью протокола **WSGI (Web Server Gateway Interface)**. Экземпляр приложения создается с помощью оператора `app = Flask(__name__)`.
2. Важно передать имя модуля в качестве аргумента конструктору `Flask`. Это позволит узнать о расположении приложения, которое станет отправной точкой для определения расположения других файлов (статические ресурсы, шаблоны и изображения). Переменная `__name__` соответствует *соглашению* (о конфигурации) и передается конструктору `Flask`, а он делает все остальное.
3. **Маршрут**: как только экземпляр `app` получает запрос, он становится ответственным за выполнение фрагмента кода для обработки запроса. Этот фрагмент кода, который обычно является функцией Python, называется *обработчиком*. Хорошая новость в том, что каждый запрос обычно (но не всегда) сопоставлен с одним URL-адресом, такая связь называется *маршрутом*. В примере мы выбрали простой подход к определению этого сопоставления, используя декоратор `route`. Для примера, URL-адрес `/hello` сопоставляется с функцией `hello`, а `/greeting` — с функцией `greeting`. Если есть предпочтение определять все маршруты в одном месте, можно использовать `add_url_rule` в экземпляре приложения.
4. **Функция обработчика**: после обработки запроса эта функция должна отправить ответ клиенту, который может быть простой строкой с HTML (или без него) или же сложной статической или динамической веб-страницей на основе шаблона. Для демонстрации мы вернули простую строку.
5. **Веб-сервер**: Flask имеет сервер разработки, который можно запустить методом `app.run()` или с помощью команды `flask run` в оболочке. При запуске сервер по умолчанию ищет модуль `app.py` или `wsgi.py`, и он будет загружен автоматически с сервером, если указать имя `app.py` для файла модуля (опять же по соглашению о конфигурации). Если назвать модуль по-другому (как в нашем случае), нужно задать переменную среды `FLASK_APP = <имя модуля>`, с помощью которой веб-сервер загрузит модуль.
6. Если проект Flask создан с помощью IDE, вроде PyCharm Pro, переменная окружения будет установлена как часть проекта. Если используется оболочка командной строки, можно установить переменную окружения следующим выражением в зависимости от операционной системы:

```
export FLASK_APP = app1.py.    #для macOS и Linux
set FLASK_APP = app1.py.       #для MS windows
```

7. При запуске сервер прослушивает клиентские запросы по адресу `http://localhost:5000/` и по умолчанию доступен только на локальном компьютере. Если необходимо запустить сервер с другим именем хоста и портом, можно выполнить следующую команду (или эквивалентный оператор Python):

```
Flask run --host <ip_address> --port <port_num>
```

8. **Веб-клиент**: можно протестировать приложение через браузер, введя URL в адресную строку или используя утилиту `curl` для простых HTTP-запросов. В нашем случае мы можем сделать это с помощью следующих команд `curl`:

```
curl -X GET http://localhost:5000/
curl -X GET http://localhost:5000/greeting
```

Когда мы рассмотрели основы Flask, можем перейти к изучению тем, связанных с запросами и отправкой динамических ответов клиентам.

Обработка запросов с разными типами HTTP-методов

HTTP использует модель *«запрос-ответ»* между клиентом и сервером. Клиент (например, веб-браузер) может вызывать методы для определения типа запроса к серверу: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `PATCH` или `OPTIONS`. Методы `GET` и `POST` — самые часто используемые, поэтому для демонстрации принципов веб-разработки мы рассмотрим подробно только их.

Также важно понять два ключевых компонента — это HTTP-запрос и HTTP-ответ. HTTP-запрос состоит из трех частей:

- **Строка запроса**: включает используемый HTTP-метод, URI-адрес запроса и используемый HTTP-протокол (версию):

```
GET /home HTTP/1.1
```

- **Поля заголовков**: они содержат метаданные с информацией о запросе; каждая запись заголовка получает пару *«ключ-значение»*, разделенные двоеточием (`:`).
- **Тело** (опционально): это заглушка, куда можно добавить дополнительные данные; для веб-приложения можно отправлять информацию из формы в POST-запросе внутри тела HTTP-запроса; для REST API можно отправлять данные для запросов `PUT` или `POST` внутри тела.

При отправке HTTP-запроса веб-серверу в качестве результата мы получаем HTTP-ответ. Он состоит из таких же частей, что и HTTP-запрос:

- **Строка состояния**: указывает на успешное выполнение или ошибку; здесь указывается код:

```
HTTP/1.1 200 OK
```

- Код состояния в диапазоне 200–299 указывает на успешное выполнение; коды в диапазоне 400–499 указывают на ошибки на стороне клиента, а в диапазоне 500–599 — на ошибки на стороне сервера.
- **Поля заголовков**: аналогичны заголовкам HTTP-запроса.
- **Тело** (опционально): хоть и не обязательная, но ключевая часть HTTP-ответа; может включать HTML-страницы для веб-приложений или данные в любом другом формате.

`GET` используется для отправки запроса на определенный ресурс, указанный в URL, с возможностью добавления строки запроса как части адреса. Символ «`?`» добавля-

ется в URL для отделения строки запроса от основного адреса. Например, если в поиске Google мы ищем «Python», то в адресной строке увидим следующий URL:

```
https://www.google.com/search?q=Python
```

В этом адресе `q=Python` — это строка запроса. Она используется для переноса данных в виде пар *«ключ-значение»*. Такой вариант доступа к ресурсам популярен благодаря своей простоте, но имеет ограничения. Данные из строки запроса видны в URL, поэтому недопустимо отправлять конфиденциальную информацию, например, имя пользователя и пароль. К тому же, длина строки запроса не может превышать 255 символов. Однако из-за своей простоты этот метод нашел свою реализацию в поисковых системах вроде Google и YAHOO. В методе `POST` данные отправляются в теле HTTP-запроса, что позволяет обойти ограничения метода `GET`. Данные не отображаются как часть адреса, и нет ограничений на их размер. Также нет ограничений на типы данных.

Flask предоставляет несколько удобных способов определить, с помощью какого метода отправлен запрос: `GET`, `POST` или какой-то другой. В следующем примере мы рассмотрим два подхода. В первом используется декоратор `route` с точным списком ожидаемых методов, во втором — специфичные для каждого HTTP-метода декораторы `get` и `post`. Следующий пример демонстрирует оба варианта:

```
#app2.py: сопоставление запроса с типом метода
from flask import Flask, request
app = Flask(__name__)
@app.route('/submit', methods=['GET'])
def req_with_get():
    return "Received a get request"

@app.post('/submit')
def req_with_post():
    return "Received a post request"

@app.route('/submit2', methods = ['GET', 'POST'])
def both_get_post():
    if request.method == 'POST':
        return "Received a post request 2"
    else:
        return "Received a get request 2"
```

Разберем три определения маршрута и соответствующие функции:

1. В первом определении (`@app.route('/submit', methods=['GET'])`) мы использовали декоратор `route` для сопоставления URL-адреса в запросах типа `GET` с функцией Python. С этим декоратором функция будет обрабатывать запросы методом `GET` только для адреса `/submit`.
2. Во втором определении маршрута (`@app.post('/submit')`) мы использовали декоратор `post` и указали вместе с ним только URL запроса. Это упрощенная версия

сопоставления запроса в методе POST с функцией Python. Новый параметр эквивалентен первому определению, но с методом POST в упрощенном виде вместо GET. То же самое можно сделать для метода GET с помощью декоратора get.

3. В третьем определении (@app.route('/submit2', methods = ['GET', 'POST'])) мы сопоставили один URL в запросах для методов POST и GET с одной функцией Python. Этот способ удобен, когда ожидается, что любой метод запроса будет обрабатываться одним обработчиком (функцией Python). Внутри функции мы использовали атрибут method из объекта запроса для определения типа: GET или POST. Обратите внимание, веб-сервер делает объект запроса доступным для приложения Flask после импорта пакета request в программу. Этот подход позволяет клиентам отправлять запросы любым из двух методов, используя один и тот же URL. Как разработчики, мы сопоставили оба варианта с одной функцией Python.

Будет удобнее протестировать код с помощью утилиты curl, поскольку непросто отправлять POST-запросы без HTML-формы. Следующие команды можно использовать для отправки HTTP-запросов в веб-приложение:

```
curl -X GET http://localhost:5000/submit
curl -X POST http://localhost:5000/submit
curl -X GET http://localhost:5000/submit2
curl -X POST http://localhost:5000/submit2
```

Далее рассмотрим, как отобразить ответ от статических страниц и шаблонов.

Отображение статического и динамического контента

Статический контент важен, поскольку включает в себя файлы CSS и JavaScript. Они могут обслуживаться напрямую веб-сервером. Flask также может взять на себя этот функционал, если создать каталог с именем static в проекте и перенаправить на него клиента.

Динамический контент можно создать с помощью Python, но это трудоемкая задача, которая требует значительных усилий для обслуживания такого кода. Рекомендуемый подход — использовать шаблонизатор, например Jinja2. Эта библиотека уже присутствует в составе Flask, поэтому она не требует дополнительной настройки. Ниже приведен пример с двумя функциями, одна из которых обрабатывает запрос для статического контента, а другая — для динамического:

```
#app3.py: отображение статического и динамического контента
from flask import Flask, render_template, url_for, redirect

app = Flask(__name__)

@app.route('/hello')
def hello():
    hello_url = url_for ('static', filename='app3_s.html')
    return redirect(hello_url)
```

```
@app.route('/greeting')
def greeting():
    msg = "Hello from Python"
    return render_template('app3_d.html', greeting=msg)
```

Ключевые моменты кода:

1. Мы импортируем дополнительные модули из Flask, такие, как `url_for`, `redirect` и `render_template`.
2. Для маршрута `/hello` мы создаем URL-адрес с помощью функции `url_for`, указав каталог `static` и имя HTML-файла в качестве аргументов. Отправляем ответ, который является инструкцией для браузера о перенаправлении клиента на URL со статическим файлом. Инструкции по перенаправлению обозначены кодом состояния в диапазоне 300–399, который автоматически задается для Flask при использовании функции `redirect`.
3. Для маршрута `/greeting` мы отображаем шаблон `app3_d.html` с помощью функции `render_template`. Мы также передаем строку приветствия (`greeting`) как значение переменной для шаблона. Переменная `greeting` доступна шаблону, как показано в следующем фрагменте из файла `app3_d.html`:

```
<!DOCTYPE html>
<body>
{% if greeting %}
  <h1> {{greeting}}!</h1>
{% endif %}
</body>
</html>
```

4. Это простейший Jinja-шаблон с оператором `if`, заключенным в `{% %}`, и переменной в двойных фигурных скобках `{{}}`. Мы не будем подробно рассматривать шаблоны Jinja2, но настоятельно рекомендуем это сделать вам с помощью документации (*https://jinja.palletsprojects.com/*).

Получить доступ к этому примеру приложения можно с помощью браузера и утилиты `curl`. Далее обсудим, как извлекать параметры из разных типов запросов.

Извлечение параметров из HTTP-запроса

Веб-приложения, в отличие от веб-сайтов, взаимодействуют с пользователем, а это невозможно без обмена данными между клиентом и сервером. Поэтому в этом подразделе мы обсудим, как извлекать данные из запроса. В зависимости от HTTP-метода мы будем использовать разные подходы. Рассмотрим три типа запроса:

- Параметры как часть URL-адреса запроса.
- Параметры как строка запроса с методом GET.
- Параметры в виде HTML с методом POST.

Пример кода с тремя разными маршрутами для трех типов запросов приведен ниже. Мы отображаем шаблон (app4.html), который аналогичен использованному ранее в app3_d.html, за исключением того что переменная имеет имя name вместо greeting:

```
#app4.py: извлечение параметров из разных запросов
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/hello')
@app.route('/hello/<fname> <lname>')
def hello_user(fname=None, lastname=None):
    return render_template('app4.html', name=f"{fname}{lname}")

@app.get('/submit')
def process_get_request_data():
    fname = request.args['fname']
    lname = request.args.get('lname', '')
    return render_template('app4.html', name=f"{fname}{lname}")

@app.post('/submit')
def process_post_request_data():
    fname = request.form['fname']
    lname = request.form.get('lname','')
    return render_template('app4.html', name=f"{fname}{lname}")
```

Рассмотрим все подходы подробнее:

1. Для первого набора (app.route) мы определили маршрут таким образом, что любой текст после /hello/ считается параметрами запроса. Мы можем задать один или два параметра, или не задать ни одного, и наша функция способна обработать любую комбинацию и в качестве ответа вернуть имя шаблону (которое может быть пустым). Это подход хорош для простых случаев передачи параметров программе сервера и является популярным выбором при разработке REST API для доступа к одному экземпляру ресурса.
2. Для второго маршрута (app.get) мы извлекаем параметры строки запроса из объекта словаря args. Получить значение можно по имени в качестве ключа словаря или с помощью метода GET со вторым аргументом в качестве значения по умолчанию. Мы использовали пустую строку как значение по умолчанию с GET-методом. Из двух рассмотренных вариантов лучше использовать GET, если в запросе нет значения по умолчанию и есть необходимость его установить.
3. Для третьего маршрута (app.post) параметры поступают в виде данных формы как часть тела HTTP-запроса, и мы будем использовать объект словаря для их извлечения. Мы также использовали имя параметра в качестве ключа словаря и метод GET в целях демонстрации.

4. Для тестирования этих сценариев рекомендуется использовать утилиту `curl`, особенно для `POST`-запросов. Мы протестировали приложение следующими командами:

```
curl -X GET http://localhost:5000/hello
curl -X GET http://localhost:5000/hello/jo%20so
curl -X GET 'http://localhost:5000/submit?fname=jo&lname=so'
curl -d "fname=jo&lname=so" -X POST http://localhost:5000/submit
```

Далее обсудим, как работать с базой данных в Python.

Взаимодействие с системами управления базами данных

Полноценное веб-приложение требует хранения структурированных данных, поэтому знания и опыт работы с БД необходимы для веб-разработки. Python и Flask интегрируются с большинством СУБД, будь то SQL или NoSQL. Сам Python поставляется с облегченной базой данных SQLite, расположенной в модуле `sqlite3`. Мы выбрали ее, поскольку она не требует настройки отдельного сервера и хорошо работает для небольших приложений. В производственных средах рекомендуется использовать другие СУБД, например, MySQL, MariaDB или PostgreSQL. Для доступа и взаимодействия с БД используется одно из расширений Flask — `Flask-SQLAlchemy`. Оно основано на библиотеке Python `SQLAlchemy`. Она предоставляет инструмент *объектно-реляционного связывания* (**Object Relational Mapper**, **ORM**), который сопоставляет таблицы БД с объектами Python. Использование библиотеки ORM не только ускоряет цикл разработки, но и обеспечивает гибкость переключения основной СУБД без изменения кода. Поэтому для работы с СУБД рекомендуется использовать SQLAlchemy или аналогичную библиотеку.

Как обычно, для взаимодействия с любой БД нужно создать экземпляр приложения. Поэтому следующий шаг — настройка этого экземпляра с URL-адресом для расположения БД (в случае с SQLite3 это файл). После создадим экземпляр `SQLAlchemy`, передав его экземпляру приложения. SQLite3 достаточно инициировать один раз. Это также можно сделать из программы, но мы не рекомендуем такой подход из-за сброса БД при каждом запуске приложения. Лучше инициализировать БД один раз из командной строки, используя экземпляр `SQLAlchemy`. Обсудим шаги инициализации БД подробнее после примера.

Для демонстрации работы `SQLAlchemy` создадим простое приложение с операциями `add`, `list` и `delete` объектов `student` в таблице БД. Пример инициализации приложения и экземпляра базы данных (`SQLAlchemy`), а также создания объекта `Model` класса `Student`, представлен ниже:

```
#app5.py (часть 1): взаимодействие с БД для операций create, delete и list с
  объектами
from flask import Flask, request, render_template, redirect
from flask_sqlalchemy import SQLAlchemy
```

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///student.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    grade = db.Column(db.String(20), nullable=True)

    def __repr__(self):
        return '<Student %r>' % self.name
```

Создав db (экземпляр SQLAlchemy), можно работать с объектами БД. Прелесть библиотек ORM состоит в том, что в Python они позволяют определять схему БД как класс, в терминологии ORM она называется *моделью* (**Model**). В нашем примере мы создали класс Student, унаследованный от базового класса db.Model. В этом классе мы определили атрибуты id, name и grade, которые соответствуют трем столбцам в таблице Student экземпляра базы данных SQLite3. Для каждого атрибута определен его тип данных с максимальной длиной, наличие *первичного ключа* (**Primary key**) и возможность обнуления. Эти дополнительные определения атрибутов важны для оптимальной настройки таблиц БД.

В следующем фрагменте кода продемонстрируем функцию list_students для получения списка учеников из базы данных. Эта функция сопоставлена с URL-адресом /list нашего примера и возвращает все объекты Student из таблицы, используя метод all в экземпляре query (атрибут экземпляра db). Обратите внимание, что экземпляр query и его методы доступны из базового класса db.Model:

```
#app5.py (часть 2)
@app.get('/list')
def list_students():
    student_list = Student.query.all()
    return render_template('app5.html', students=student_list)
```

В следующем фрагменте напишем функцию (add_student) для добавления учеников в таблицу. Она связывается с URL-адресом /add и ожидает, что имя учащегося и его класс будут переданы в качестве параметров запроса, используя GET-метод. Для добавления нового объекта в БД создадим новый экземпляр класса Student с необходимыми значениями атрибутов, а затем используем экземпляр db.Session для добавления его в слой ORM с помощью функции add. Эта функция сама по себе не добавит экземпляр в базу данных. Мы будем использовать метод commit для отправки данных в таблицу. Как только новый ученик добавлен, мы передаем управление URL-адресу /list. Здесь используется перенаправление на этот URL-адрес, поскольку нам нужно вернуть последний список учеников после добавления новой записи, а также повторно использовать функцию list_students, которую мы уже реализовали.

Полный код функции add_student выглядит следующим образом:

```
#app5.py(часть 3)
@app.get('/add')
def add_student():
    fname = request.args['fname']
    lname = request.args.get('lname', '')
    grade = request.args.get('grade','')
    student = Student(name=f"{fname} {lname}", grade=grade)
    db.session.add(student)
    db.session.commit()
    return redirect("/list")
```

В последней части примера напишем функцию delete_student для удаления учеников из таблицы. Она связывается с URL-адресом /delete<int:id>. Обратите внимание, мы ожидаем, что клиент отправит идентификатор ученика, который мы передали со списком учеников, используя запрос list. Для удаления конкретного ученика мы запрашиваем его экземпляр по идентификатору. Для этого мы используем метод filter_by с экземпляром query. Получив экземпляр Student, мы применяем метод delete экземпляра db.Session и фиксируем изменения. Как и при использовании функции add_student, мы перенаправили клиента на URL-адрес /list с целью вернуть актуальный список учеников в шаблон Jinja:

```
#app5.py (часть 4)
@app.get('/delete/<int:id>')
def del_student(id):
    todelete = Student.query.filter_by(id=id).first()
    db.session.delete(todelete)
    db.session.commit()
    return redirect("/list")
```

Для вывода списка учеников в браузере, мы создали простой шаблон (app5.html), в котором находится список учеников в виде таблицы. Важно отметить, что мы используем цикл for для динамического создания строк HTML-таблицы, как показано в следующем примере:

```
<!DOCTYPE html>
<body>
<h2>Students</h2>
    {% if students|length > 0 %}
        <table>
            <thead>
              <tr>
                <th scope="col">SNo</th>
                <th scope="col">name</th>
                <th scope="col">grade</th>
              </tr>
```

```
            </thead>
            <tbody>
              {% for student in students %}
                <tr>
                    <th scope="row">{{student.id}}</th>
                    <td>{{student.name}}</td>
                    <td>{{student.grade}}</td>
                </tr>
              {% endfor %}
            </tbody>
        </table>
    {% endif %}
</body>
</html>
```

Перед запуском приложения нужно однократно инициализировать схему БД. Это можно сделать, используя программу, но нам необходимо убедиться, что код выполнится только один раз, когда база данных еще не инициализирована. Лучший вариант — сделать это вручную с помощью оболочки Python. Она позволяет импортировать экземпляр `db` из модуля приложения, а затем использовать метод `db.create_all` для инициализации БД в соответствии с классами модели, определенными в программе. Ниже приведены примеры команд, которые используются в приложении для инициализации БД:

```
>>> from app5 import db
>>> db.create_all()
```

Эти команды создадут файл `student.db` в том же каталоге, где находится программа. Для сброса БД можно удалить файл `student.db` и повторно выполнить команды инициализации или использовать метод `db.drop_all` в оболочке Python.

Протестировать приложение можно с помощью утилиты `curl` или через браузер, используя следующие URL-адреса:

- `http://localhost:5000/list`;
- `http://localhost:5000/add?fname=John&Lee=asif&grade=9`;
- `http://localhost:5000/delete/<id>`.

Далее мы узнаем, как обрабатывать ошибки в веб-приложениях на базе Flask.

Обработка ошибок и исключений в веб-приложениях

До этого момента в наших примерах мы не обращали внимания на ситуации, когда пользователь может вводить неверный URL-адрес или отправлять недопустимый набор аргументов. Мы не рассматривали такие сценарии намеренно с целью сначала сосредоточиться на ключевых компонентах веб-приложений. Прелесть веб-фреймворков в том, что они обычно поддерживают обработку ошибок по умолча-

нию. В случае возникновения какой-либо ошибки автоматически возвращается соответствующий код состояния. Коды четко определены как часть HTTP-протокола. Например, коды с 400 по 499 указывают на ошибки с клиентскими запросами, а коды с 500 по 599 — на проблемы с сервером. Ниже приведены наиболее распространенные ситуации (табл. 10.1):

Таблица 10.1. Распространенные ошибки HTTP

Код ошибки	Имя	Описание
400	Bad Request	Указывает на неверный URL или содержание запроса
401	Unauthorized	Имя пользователя или пароль в запросе не указаны или указаны неверно
403	Forbidden	Пользователь пытается обратиться к ресурсу, но ему запрещен доступ
404	Not Found	Ресурс, к которому мы пытаемся обратиться, недоступен. Обычно из-за неверно указанного URL
500	Internal Server Error	Запрос верный, но что-то случилось на стороне сервера

Полный список кодов состояний и ошибок доступен по адресу: *https://httpstatuses.com/*.

Flask также поставляется с фреймворком обработки ошибок в своем составе. При обработке клиентских запросов, если программа прерывает свою работу, по умолчанию возвращается ошибка `500 Internal Server Error`. Если клиент запрашивает URL-адрес, не связанный ни с одной функцией, Flask возвращает ошибку `404 Not Found`. Эти типы ошибок реализованы как подклассы класса `HTTPException` из библиотеки Flask.

Обрабатывать эти исключения можно с помощью пользовательского поведения или пользовательских сообщений, для этого нужно зарегистрировать наш обработчик в приложении Flask. Обратите внимание, что обработчик — это функция Flask, которая срабатывает только при возникновении ошибки, и можно связывать с ним конкретные или общие исключения. Рассмотрим пример для демонстрации концепции в общих чертах, как это работает. Сначала мы создадим простое веб-приложение с двумя функциями (`hello` и `greeting`) для обработки двух URL-адресов, как показано в следующем фрагменте:

```
#app6.py (часть 1): обработка ошибок и исключений
import json
from flask import Flask, render_template, abort
from werkzeug.exceptions import HTTPException

app = Flask(__name__)
```

```
@app.route('/')
def hello():
    return 'Hello World!'

@app.route('/greeting')
def greeting():
    x = 10/0
    return 'Greetings from Flask web app!'
```

Для обработки ошибок мы регистрируем обработчик в экземпляре приложения с помощью декоратора `errorHandler`. Для примера, который приведен ниже, мы зарегистрировали обработчик `page_not_found` для кода ошибки `404`. Для кода `500` мы зарегистрировали обработчик `internal_error`. В конце мы зарегистрировали `generic_handler` для класса `HTTPException`. Этот универсальный обработчик будет перехватывать ошибки и исключения, отличные от `404` и `500`. Пример кода со всеми тремя обработчиками:

```
#app6.py(часть 2)
@app.errorhandler(404)
def page_not_found(error):
    return render_template('error404.html'), 404
@app.errorhandler(500)
def internal_error(error):
    return render_template('error500.html'), 500

@app.errorhandler(HTTPException)
def generic_handler(error):
    error_detail = json.dumps({
        "code": error.code,
        "name": error.name,
        "description": error.description,
    })
    return render_template('error.html',
        err_msg = error_detail), error.code
```

Для демонстрации мы также написали базовые шаблоны с пользовательскими сообщениями (`error404.html`, `error500.html` и `error.html`). Шаблоны `error404.html` и `error500.html` используют жестко закодированное сообщение, а шаблон `error.html` ожидает пользовательское сообщение от веб-сервера. Для тестирования примеров, сделаем следующие запросы через браузер или утилиту `curl`:

- `GET http://localhost:5000/`: в этом случае мы ожидаем нормального ответа.
- `GET http://localhost:5000/hello`: мы ожидаем ошибку `404`, поскольку с этим адресом не связана функция Python, и приложение отобразит шаблон `error404.html`.

нию. В случае возникновения какой-либо ошибки автоматически возвращается соответствующий код состояния. Коды четко определены как часть HTTP-протокола. Например, коды с 400 по 499 указывают на ошибки с клиентскими запросами, а коды с 500 по 599 — на проблемы с сервером. Ниже приведены наиболее распространенные ситуации (табл. 10.1):

Таблица 10.1. Распространенные ошибки HTTP

Код ошибки	Имя	Описание
400	Bad Request	Указывает на неверный URL или содержание запроса
401	Unauthorized	Имя пользователя или пароль в запросе не указаны или указаны неверно
403	Forbidden	Пользователь пытается обратиться к ресурсу, но ему запрещен доступ
404	Not Found	Ресурс, к которому мы пытаемся обратиться, недоступен. Обычно из-за неверно указанного URL
500	Internal Server Error	Запрос верный, но что-то случилось на стороне сервера

Полный список кодов состояний и ошибок доступен по адресу: *https://httpstatuses.com/*.

Flask также поставляется с фреймворком обработки ошибок в своем составе. При обработке клиентских запросов, если программа прерывает свою работу, по умолчанию возвращается ошибка `500 Internal Server Error`. Если клиент запрашивает URL-адрес, не связанный ни с одной функцией, Flask возвращает ошибку `404 Not Found`. Эти типы ошибок реализованы как подклассы класса `HTTPException` из библиотеки Flask.

Обрабатывать эти исключения можно с помощью пользовательского поведения или пользовательских сообщений, для этого нужно зарегистрировать наш обработчик в приложении Flask. Обратите внимание, что обработчик — это функция Flask, которая срабатывает только при возникновении ошибки, и можно связывать с ним конкретные или общие исключения. Рассмотрим пример для демонстрации концепции в общих чертах, как это работает. Сначала мы создадим простое веб-приложение с двумя функциями (`hello` и `greeting`) для обработки двух URL-адресов, как показано в следующем фрагменте:

```
#app6.py (часть 1): обработка ошибок и исключений
import json
from flask import Flask, render_template, abort
from werkzeug.exceptions import HTTPException

app = Flask(__name__)
```

том концепции операций CRUD (**Create**, **Read**, **Update**, **Delete**). Именно здесь HTTP-методы напрямую сопоставляются с операциями, например, `GET` для операции `Read`, `POST` для операции `Create`, `PUT` для операции `Update`, `DELETE` для операции `Delete`.

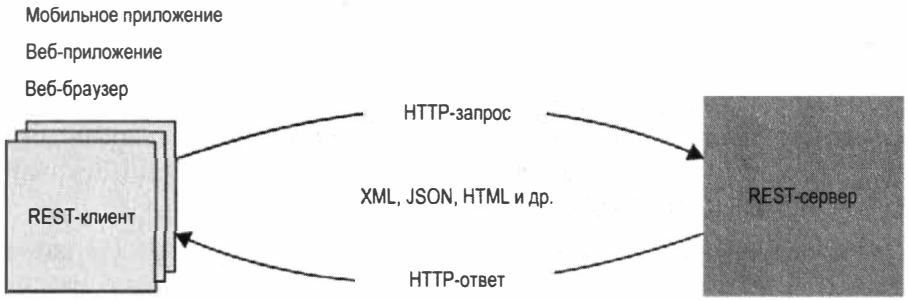


Рис. 10.1. Взаимодействие между клиентом и сервером на основе REST API

При создании REST API с HTTP-методами нужно быть осторожным при выборе метода, учитывая его *идемпотентность*. В математике операция считается идемпотентной, если дает одинаковый результат при многократном повторении выполнения. С точки зрения проектирования метод `POST` не является идемпотентным. Это означает, мы должны убедиться, что клиенты не инициируют `POST`-запрос несколько раз для одного набора данных. Методы `GET`, `PUT` и `DELETE` являются идемпотентными, хотя вполне есть возможность получить ошибку `404`, если попытаться удалить один и тот же ресурс во второй раз. Но с точки зрения идемпотентности это является допустимым поведением.

Использование Flask для REST API

REST API в Python можно создать с помощью различных библиотек и фреймворков. Самые популярные из них — это Django, Flask (с расширением Flask-RESTful) и FastAPI. Каждый из них имеет свои достоинства и недостатки. Django отлично подходит для создания REST API, если веб-приложение разрабатывается также с использованием Django. Однако задействовать его только для разработки API было бы излишним. Расширение Flask-RESTful без проблем работает с веб-приложением Flask. И Django, и Flask имеют сильную поддержку сообщества, что иногда является важным фактором при выборе. FastAPI считается лучшим по производительности и хорошо подходит, когда необходимо создать только REST API для приложения. Однако FastAPI не имеет такой же поддержки сообщества, как Django и Flask.

Мы остановили свой выбор на расширении RESTful с целью продолжить изучение Flask. Обратите внимание, мы можем создать простой веб-API, используя только Flask, что мы и сделали в предыдущей главе, когда разрабатывали приложение на базе веб-сервиса для развертывания в Google Cloud. В этом подразделе мы сосредоточимся на использовании архитектурного стиля REST при создании API. Это оз-

- `GET http://localhost:5000/greeting`: мы ожидаем ошибку `500` из-за попытки разделить число на ноль для вызова ошибки `ZeroDivisionError`; поскольку ошибка на стороне сервера, она активирует обработчик `internal_error`, который отображает общий шаблон `error500.html`.
- `POST http://localhost:5000/`: для эмуляции роли универсального обработчика отправим запрос, который вызывает код ошибки, отличный от `404` и `500`; для этого достаточно отправить запрос `POST` на адрес, который ожидает `GET`, и сервер выдаст ошибку `405` (неподдерживаемый метод HTTP); у нас в приложении нет специального обработчика для этого кода ошибки, но мы зарегистрировали универсальный обработчик из класса `HTTPException`, который обработает эту ошибку и отобразит универсальный шаблон `error.html`.

На этом можно завершить тему использования фреймворка Flask для разработки веб-приложений. Далее рассмотрим создание REST API с помощью расширений Flask.

Создание REST API

REST (или **ReST**) — это аббревиатура от **Representational State Transfer**; представляет собой архитектуру, позволяющую клиентским машинам запрашивать информацию о ресурсах на удаленном компьютере. **API** расшифровывается как **Application Programming Interface** (интерфейс прикладного программирования) и представляет собой набор правил и протоколов для взаимодействия с прикладным ПО приложения, запущенным на разных машинах. Потребность во взаимодействии между различными программными сущностями появилась достаточно давно. За последние несколько десятилетий было предложено и изобретено множество технологий для эффективного и беспрепятственного взаимодействия на программном уровне, включая *удаленный вызов процедур* (**Remote Procedure Call**, **RPC**), *удаленный вызов методов* (**Remote Method Invocation**, **RMI**), CORBA и веб-сервисы SOAP. Эти технологии имеют ограничения из-за привязки к определенному языку программирования, привязки к проприетарному транспортному механизму или использованию только определенного типа формата данных. Эти ограничения были почти полностью устранены в RESTful API, который широко известен как REST API.

Благодаря своей простоте и гибкости протокол HTTP удобно использовать в качестве транспортного механизма. Еще одно преимущество заключается в поддержке разных форматов данных для обмена (текст, XML, JSON). REST API не привязан к одному языку, что делает его лучшим выбором при создании API для веб-взаимодействия. На рис. 10.1 приведено архитектурное представление вызова REST API от REST-клиента к REST-серверу с помощью HTTP.

REST API основан на HTTP-запросах и использует собственные методы, например `GET`, `PUT`, `POST` и `DELETE`. Использование HTTP-методов упрощает реализацию клиентского и серверного ПО с точки зрения проектирования. REST API разработан с уче-

хож на argparse, который является популярным инструментом для парсинга аргументов командной строки.

Далее рассмотрим разработку REST API для доступа к данным в БД.

Разработка REST API для доступа к базе данных

Для демонстрации работы Flask и расширения Flask-RESTful при создания REST API мы изменим наше приложение app5.py и предоставим доступ к объекту Student (объект Resource) с помощью архитектурного стиля REST. Ожидается, что аргументы, отправленные методам PUT и POST, находятся в теле запроса, и API отправит ответ в формате JSON. Измененный код приведен ниже:

```
#api_app.py: приложение REST API для ресурса student
from flask_sqlalchemy import SQLAlchemy
from flask import Flask
from flask_restful import Resource, Api, reqparse
app = Flask(__name__)
api = Api(app)

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///student.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

В предыдущем фрагменте мы начали с инициализации приложения и экземпляра БД. Далее создали экземпляр API с помощью экземпляра Flask, выполнив выражение api = Api(app). Этот экземпляр API является ключом для разработки остальной части приложения, и мы будем использовать его в дальнейшем.

Затем нужно настроить экземпляр reqparse, зарегистрировав аргумент, который мы ожидаем для парсинга от HTTP-запроса. В нашем примере мы зарегистрировали два аргумента строкового типа (name и grade), как показано в следующем фрагменте:

```
parser = reqparse.RequestParser()
parser.add_argument('name', type=str)
parser.add_argument('grade', type=str)
```

На следующем шаге мы создадим объект Student, аналогичный тому, что в примере app5.py, но добавив в него метод serialize для преобразования объекта в формат JSON. Это важный шаг для сериализации ответа JSON перед его отправкой обратно клиентам API. Существуют и другие решения данной задачи, но мы выбрали этот вариант из-за его простоты. Пример кода выглядит следующим образом:

```
class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    grade = db.Column(db.String(20), nullable=True)
```

начает, что мы будем использовать HTTP-метод для выполнения операций с ресурсом, который будет представлен объектом Python.

> ***Важное примечание***
>
> Поддержка Flask-RESTful уникальна тем, что предоставляет удобный способ задать код и заголовок ответа как часть оператора `return`.

Для использования Flask с расширением Flask-RESTful нужно сначала его установить. Это можно сделать в виртуальной среде с помощью команды `pip`:

```
pip install Flask-RESTful
```

Прежде чем обсуждать реализацию, познакомимся с некоторыми терминами и концепциями.

Ресурс

Это ключевой элемент REST API, и он поддерживается расширением Flask-RESTful. Объект ресурса определяется путем расширения нашего класса из базового класса `Resource`, который доступен в библиотеке Flask-RESTful. Базовый класс `Resource` предлагает несколько *магических функций* для помощи в разработке и автоматически связывает HTTP-методы с методами Python, определенными в объекте ресурса.

Конечная точка API

Это точка входа для установки соединения между ПО клиента и ПО сервера. Говоря простыми словами, конечная точка — это альтернативная терминология для URL-адреса сервера или службы, где программа прослушивает API-запросы. С помощью Flask-RESTful мы определяем конечную точку, связывая определенный URL-адрес (или несколько адресов) с объектом ресурса. Мы реализуем объект, расширяя базовый класс `Resource`.

Маршрутизация

Концепция маршрутизации для API аналогична маршрутизации для веб-приложений с Flask с той лишь разницей, что в первом случае необходимо сопоставить объект `Resource` с одним или несколькими URL-адресами конечных точек.

Парсинг аргументов

Парсинг аргументов запроса возможен с использованием строки запроса или данных в HTML-форме. Однако это не лучший вариант, поскольку ни строка запроса, ни HTML-форма не предназначены для использования с API. Рекомендуемый подход заключается в извлечении аргументов напрямую из HTTP-запроса. Для этой задачи расширение Flask-RESTful предлагает специальный класс `reqparse`. Он по-

```
    grade = args['grade']
    student = Student(name=name, grade=grade)
    db.session.add(student)
    db.session.commit()
    return student, 200
```

Для метода `post` класса `StudentListDao` мы использовали парсер `reqparse` для извлечения аргументов `name` и `grade` из запроса. Остальная часть реализации `POST`-метода совпадает с реализацией в примере `app5.py`.

В следующих двух строках мы сопоставили URL-адреса с объектами `Resource`. Все запросы, поступающие для `/students/<student_id >`, будут перенаправлены в класс ресурсов `StudentDao`. Все запросы для `/students` будут перенаправлены в класс ресурса `StudentListDao`:

```
api.add_resource(StudentDao, '/students/<student_id>')
api.add_resource(StudentListDao, '/students')
```

Обратите внимание, мы пропустили реализацию метода `PUT` из класса `StudentDao`, но для полноты картины его можно найти в исходном коде к этой главе. Для примера мы не добавляли обработку ошибок и исключений с целью избежать усложнения кода, но настоятельно рекомендуется включить это в окончательную реализацию.

На данном этапе мы рассмотрели базовые концепции и принципы реализации в разработке REST API. Далее мы расширим наши знания и создадим полноценное веб-приложение.

Пример: создание веб-приложения с помощью REST API

В этой главе мы узнали, как создать простое веб-приложение с помощью Flask и как добавить REST API к уровню бизнес-логики с помощью расширения Flask. В реальном мире веб-приложения обычно состоят из трех уровней: веб-уровень, уровень бизнес-логики и уровень доступа к данным. С ростом популярности мобильных приложений архитектура эволюционировала, и стало возможным использовать REST API в качестве строительного блока для бизнес-логики. Это позволяет создавать мобильные и веб-приложения, используя один и тот же уровень бизнес-логики. Более того, тот же API можно использовать для B2B-взаимодействия с другими поставщиками. Схема такой архитектуры приведена на рис. 10.2.

В этом примере мы разработаем веб-приложение поверх REST API, которое мы ранее создали для объекта модели `Student`. В общих чертах приложение будет включать компоненты, показанные на рис. 10.3.

```
    def serialize(self):
        return {
            'id': self.id,
            'name': self.name,
            'grade': self.grade
        }
```

Далее мы создали два класса `Resource` для доступа к объектам учеников в базе данных (`StudentDao` и `StudentListDao`):

- `StudentDao` предлагает методы `get` и `delete` для отдельных экземпляров ресурса; они связаны с методами `GET` и `DELETE` протокола HTTP.
- `StudentListDao` предлагает методы `get` и `post`; первый предоставляет список всех ресурсов типа `Student` с помощью HTTP-метода `GET`, а второй добавляет новый объект ресурса с помощью HTTP-метода `POST`; это стандартный шаблон проектирования при реализации функционала CRUD для веб-ресурса.

Что касается методов, реализованных для классов `StudentDao` и `StudentListDao`, мы возвращаем код состояния и сам объект в одном операторе. Эту удобную возможность предоставляет расширение Flask-RESTful.

Пример кода для `StudentDao`:

```
class StudentDao(Resource):
    def get(self, student_id):
        student = Student.query.filter_by(id=student_id).\
            first_or_404(description='Record with id={} is
                not available'.format(student_id))
        return student.serialize()

    def delete(self, student_id):
        student = Student.query.filter_by(id=student_id).\
            first_or_404(description='Record with id={} is
                not available'.format(student_id))
        db.session.delete(student)
        db.session.commit()
        return '', 204
```

Пример кода для `StudentListDao`:

```
class StudentListDao(Resource):
    def get(self):
        students = Student.query.all()
        return [Student.serialize(student) for student in students]

    def post(self):
        args = parser.parse_args()
        name = args['name']
```

Приложение webapp.py будет основано на Flask. Оно (далее webapp) не будет зависеть от api_app.py (далее apiapp) в том смысле, что оба приложения будут работать отдельно как два экземпляра Flask в идеале на двух отдельных машинах. Но, если мы запускаем оба экземпляра на одном компьютере в целях тестирования, мы должны задействовать разные порты и использовать IP-адрес локальной машины в качестве хоста. Flask использует адрес 127.0.0.1 для встроенного веб-сервера, что может помешать запустить на нем два экземпляра. Оба приложения будут взаимодействовать только через REST API. Кроме того, мы разработаем несколько шаблонов Jinja для отправки запросов на выполнение операций *создания*, *обновления* и *удаления*. Мы будем повторно использовать код api_py.py как есть и напишем приложение webapp.py с такими функциями, как *просмотр списка учеников*, *добавление* и *удаление учеников*, а также *изменение* данных о них. Для каждой из них мы добавим функции Python:

1. Начнем с инициализации экземпляра Flask, как мы это делали в предыдущих примерах. Код выглядит следующим образом:

```
#webapp.py: взаимодействие с уровнем бизнес-логики через REST API
#для создания, удаления и просмотра объектов
from flask import Flask, render_template, redirect, request
import requests, json

app = Flask(__name__)
```

2. Далее добавим функцию list для обработки запросов с URL-адреса «/»:

```
@app.get('/')
def list():
    response = requests.get('http://localhost:8080/students')
    data = json.loads(response.text)
    return render_template('main.html', students=data)
```

3. Во всех функциях используется библиотека requests для отправки запроса REST API в приложение apiapp, которое размещено на том же компьютере в нашей тестовой среде.

4. Реализуем функцию add для обработки запроса на добавление нового ученика в БД. С этой функцией будет сопоставляться только запрос с методом POST. Пример кода выглядит следующим образом:

```
@app.post('/')
def add():
    fname = request.form['fname']
    lname = request.form['lname']
    grade = request.form['grade']
    payload = {'name': f"{fname} {lname}", 'grade':
        grade}
    respone = requests.post('http://localhost:8080
        /students', data=payload)
    return redirect("/")
```



Рис. 10.2. Архитектура веб- и мобильных приложений

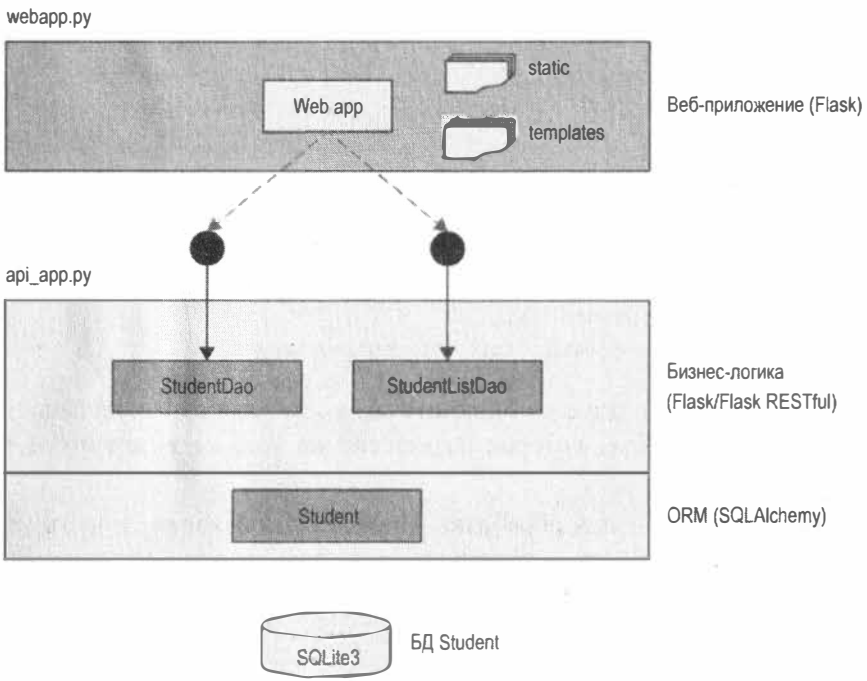


Рис. 10.3. Пример веб-приложения на базе REST API

Мы уже разработали уровни бизнес-логики и доступа к данным (ORM), а также предоставили функционал через две конечные точки API (ранее в подразделе *«Использование Flask для REST API»*). Теперь разработаем часть приложения для веб-доступа и будем использовать API, предлагаемый бизнес-логикой.

2. Корневой URL-адрес «/» запустит главную страницу `main.html`, которая позволит добавить нового ученика, а также предоставит список имеющихся учеников. На рис. 10.4 приведен скриншот главной страницы, которая будет отображаться с помощью шаблона `main.html`:

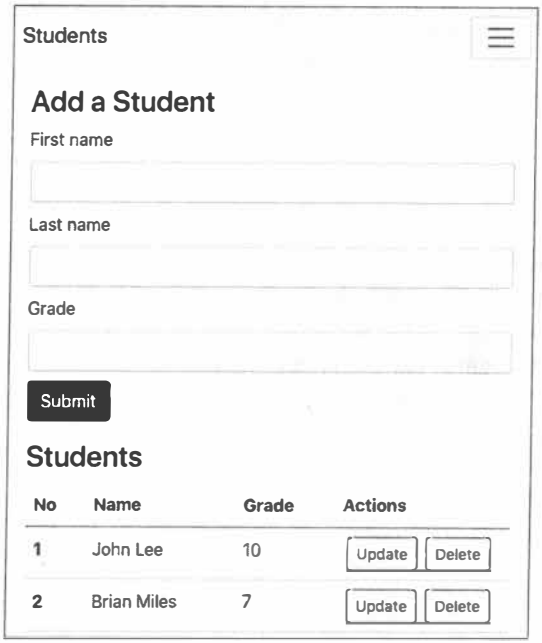


Рис 10.4. Главная страница приложения webapp

3. Если мы введем имя, фамилию и класс ученика и нажмем кнопку **Submit**, выполнится `POST`-запрос с данными этих трех полей. Веб-приложение передаст этот запрос в функцию `add`, которая использует соответствующий REST API приложения `apiapp` для добавления нового ученика, и снова отобразит главную страницу с обновленным списком.
4. На главной странице `webapp` (`main.html`) мы добавили две кнопки (**Update** и **Delete**) к каждой записи об ученике. При нажатии на **Delete** браузер выполняет `GET`-запрос с URL-адресом `/delete/<id>`, который делегируется функции `delete`. Она, в свою очередь, будет использовать соответствующий REST API для удаления ученика из БД и снова отобразит главную страницу с обновленным списком учеников.
5. При нажатии на **Update** браузер выполнит `GET`–запрос с URL-адресом `/update/<id>`. который будет передан функции `load_student_for_update`. Она загрузит информацию об ученике с помощью REST API, затем передаст данные в ответ и отобразит шаблон `update.html`. Он покажет пользователю HTML-форму с данными ученика, которые можно изменять. Форма для этого сценария показана на рис. 10.5.

5. Обратите внимание, для вызова в приложении apiapp мы создали объект payload и передали его как атрибут data в POST-метод модуля requests.

6. Далее добавим функцию DELETE для обработки запроса на удаление ученика из БД. Ожидается, что тип запроса, связанный с этим методом, предоставит идентификатор ученика как часть URL-адреса:

```
@app.get('/delete/<int:id>')
def delete(id):
    response = requests.delete('http://localhost:8080
        /students/'+str(id))
    return redirect("/")
```

7. Добавим две функции для реализации изменения данных. Первая (update) обновит данные ученика таким же образом, как и функция post. Перед запуском update приложение webapp предложит пользователю форму с текущими данными объекта student. Вторая функция (load_student_for_update) получит объект student и отправит его в шаблон для редактирования пользователем. Код обеих функций выглядит следующим образом:

```
@app.post('/update/<int:id>')
def update(id):
    fname = request.form['fname']
    lname = request.form['lname']
    grade = request.form['grade']
    payload = {'name' : f"{fname} {lname}",'grade':grade}
    respone = requests.put('http://localhost:8080
        /students/' + str(id), data = payload)
    return redirect("/")

@app.get('/update/<int:id>')
def load_student_for_update(id):
    response = requests.get('http://localhost:8080
        /students/'+str(id))
    student = json.loads(response.text)
    fname = student['name'].split()[0]
    lname = student['name'].split()[1]
    return render_template('update.html', fname=fname,
        lname=lname, student= student)
```

Код внутри этих функций не отличается от предыдущих примеров, поэтому мы не будем подробно рассматривать каждую строку, а выделим основные моменты веб-приложения и его взаимодействия с приложением REST API:

1. Мы используем два шаблона (main.html и update.html). Мы также используем общий для них шаблон base.html. Он в основном создается, с помощью UI-фреймворка bootstrap. Здесь мы не будем подробно обсуждать шаблоны Jinja и bootstrap, но вы можете ознакомиться с ними по ссылкам, предложенным в конце этой главы. Там также можно найти примеры этих шаблонов.

SQLAlchemy. В дополнение мы узнали роль веб-API для мобильных, B2B- и веб-приложений. На простом примере изучили расширение Flask для разработки REST API. В конце мы реализовали пример веб-приложения со списком учеников, которое состоит из двух независимых приложений. Одно из них предлагает REST API для уровня бизнес-логики поверх СУБД. Другое предоставляет пользователям веб-интерфейс и использует интерфейс REST API первого приложения для обеспечения доступа к таблице учеников.

Примеры кода в этой главе позволят приступить к созданию собственных веб-приложений и написанию REST API. Знания будут полезны всем, кто хочет стать веб-разработчиком и работать над созданием REST API.

В следующей главе мы увидим, как использовать Python для разработки микросервисов, которые являются новой парадигмой разработки ПО.

Вопросы

1. В чем цель TLS?
2. В каких случаях Flask лучше Django?
3. Какие HTTP-методы наиболее часто используются?
4. Что такое CRUD и как это связано с REST API?
5. Использует ли REST API только JSON в качестве формата данных?

Дополнительные ресурсы

- «*Разработка веб-приложений с использованием Flask на языке Python*» (Flask Web Development), автор: Мигель Гринберг (Miguel Grinberg).
- «*Advanced Guide to Python 3 Programming*», автор: Джон Хант (John Hunt).
- «*REST APIs with Flask and Python*», авторы: Джейсон Маерз (Jason Myers) и Рик Копланд (Rick Copeland).
- «*Essential SQLAlchemy*», 2-е издание, автор: Хосе Сальватьерра (Jose Salvatierra).
- «*Bootstrap 4 Quick Start*», автор: Джейкоб Летт (Jacob Lett).
- Документация по Jinja: *https://jinja.palletsprojects.com/*.

Ответы

1. Главная цель TLS — обеспечить шифрование данных, которыми обмениваются две системы в Интернете.

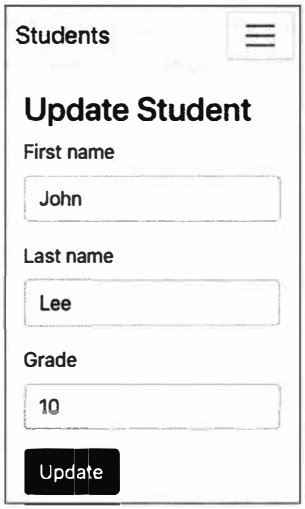


Рис. 10.5. Пример формы для изменения данных ученика

После внесения изменений, если пользователь отправит форму, нажав на кнопку **Update,** браузер выполнит `POST`-запрос с URL-адресом `/update/<id>`. Мы зарегистрировали функцию `update` для этого запроса. Она будет извлекать данные и передавать их в REST API. После обновления информации об ученике мы снова отображаем страницу `main.html` с обновленными сведениями.

В этой главе мы не рассматривали подробные детали *чистых* веб-технологий, таких, как HTML, Jinja, CSS и UI-фреймворки. Прелесть веб-фреймворков в том, что они позволяют использовать любые веб-технологии пользовательских интерфейсов, особенно если мы создаем приложения с помощью REST API.

На этом мы завершаем обсуждение создания веб-приложений и разработку REST API с помощью Flask и его расширений. Разработка не ограничивается одним языком или фреймворком. Ключевые принципы и архитектуры остаются неизменными для всех фреймворков и языков. Принципы, которые мы рассмотрели, помогут лучше понять другие веб-инструменты для Python, а также другие языки.

Заключение

В этой главе мы узнали, как использовать Python и веб-фреймворки вроде Flask для разработки веб-приложений и REST API. Мы начали с анализа требований к веб-разработке, которые включают веб-фреймворк, UI-фреймворк, веб-сервер, СУБД, поддержку API, безопасность и документацию. Затем на примерах узнали, как использовать Flask для создания веб-приложений. Обсудили типы запросов с разными HTTP-методами и способы парсинга данных в запросе. Мы также изучили, как использовать Flask для взаимодействия с СУБД, используя ORM-библиотеки вроде

11

Разработка микросервисов на Python

Монолитные приложения с одноуровневым ПО много лет были популярным вариантом для разработки, однако развертывать их на облачных платформах неэффективно с точки зрения резервирования и использования ресурсов. Также дела обстоят и при развертывании крупномасштабных монолитных приложений на физических машинах. Разрабатывать и обслуживать такие приложения всегда дорого. Разделение приложений на несколько уровней в какой-то мере решило проблему, но настоящим спасением стала *микросервисная архитектура*, которая обеспечивает динамическое выделение ресурсов и сокращает затраты на разработку и обслуживание. Она позволяет создавать приложения из слабо связанных сервисов и развертывать их на динамически масштабируемых платформах, таких, как *контейнеры*. Организации вроде Amazon, Netflix и Facebook уже перешли от монолитной модели к микросервисам. Иначе они не смогли бы обслуживать огромное количество клиентов.

Темы этой главы:

- Введение в микросервисы.
- Практические рекомендации по созданию микросервисов.
- Создание приложений на базе микросервисов.

В этой главе вы познакомитесь с микросервисами и научитесь создавать приложения на их основе.

Технические требования

В этой главе понадобится:

- Python 3.7 или более поздней версии.
- Библиотека Flask с расширением RESTful.

2. Flask — лучший выбор для малых и средних приложений, особенно если требования проекта меняются часто.
3. `GET` и `POST`.
4. **CRUD** расшифровывается как **Create**, **Read**, **Update** и **Delete**. Это операции, которые используются в разработке ПО. С точки зрения API каждая операция CRUD сопоставляется с одним из HTTP-методов (`GET`, `POST`, `PUT` и `DELETE`).
5. REST API поддерживает любой строковый формат, например, JSON, XML или HTML. Поддержка формата данных больше связана со способностью HTTP передавать данные как часть тела HTTP.

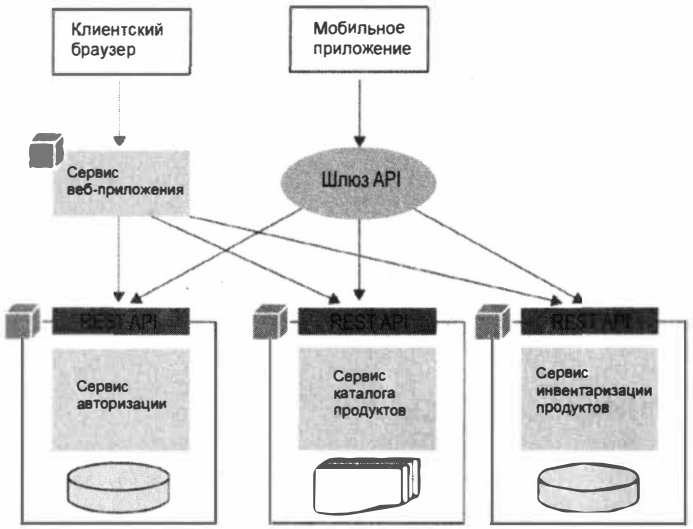


Рис. 11.1. Пример приложения с микросервисной архитектурой

2. Возможность разрабатывать, тестировать и обслуживать отдельные микросервисы небольшими независимыми командами. Крайне важно иметь независимые и автономные команды для разработки крупномасштабных приложений.
3. Одна из проблем монолитных приложений — управление конфликтующими версиями библиотек, без которых не обойтись. При использовании микросервисов конфликты версий сведены к минимуму.
4. Возможность развертывать и исправлять отдельные микросервисы независимо друг от друга. Благодаря этому можно использовать CI/CD для сложных приложений. Это также позволяет применять патчи или обновлять только один компонент приложения. В случае с монолитным приложением приходится повторно развертывать его целиком, а значит, есть вероятность сломать другие части приложения. При использовании микросервисов только один или два сервиса будут развернуты без риска нарушить работу остальных.
5. Возможность изолировать ошибки и сбои на уровне микросервиса, а не всего приложения. Если есть сбой в одном сервисе, можно выполнить отладку и исправить ошибку или остановить сервис для обслуживания, не затронув остальной функционал приложения. Ошибки в монолитных приложениях могут вывести из строя все приложение.

Несмотря на преимущества микросервисная архитектура имеет недостатки:

1. Повышенная сложность создания приложений на базе микросервисов. В основном это связано с тем, что каждый микросервис должен предоставлять API для взаимодействия с ним клиентских сервисов и программ. Также разработку усложняет необходимость обеспечивать безопасность каждого микросервиса.
2. Повышенная потребность в ресурсах по сравнению с монолитными приложениями. Каждый микросервис требует независимого размещения дополнительной

- Библиотека Django с фреймворком Django REST.
- Аккаунт Docker, а также установленные на вашей машине Docker Engine и Docker Compose.
- Аккаунт GCP для развертывания микросервиса в GCP Cloud Run (достаточно бесплатной пробной версии).

Пример кода для этой главы можно найти по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter11.

Начнем обсуждение со знакомства с микросервисами.

Введение в микросервисы

Микросервис — это независимая программная сущность, которая должна иметь следующие характеристики:

- **Слабая связь** с другими сервисами и независимость от других программных компонентов.
- **Простота разработки и обслуживания** небольшой командой, не зависящей от других команд.
- **Независимая установка** в виде отдельной сущности, предпочтительно в контейнере.
- **Простые в использовании** интерфейсы с синхронными (REST API) или асинхронными протоколами (Kafka или RabbitMQ).

Ключевые слова с точки зрения микросервиса: *независимое развертывание*, *слабое связывание* и *простота обслуживания*. Каждый микросервис может иметь собственные серверы БД для избегания совместного использования ресурсов с другими микросервисами. Это обеспечит устранение зависимостей между ними.

Микросервисная архитектура — это парадигма разработки ПО, которое состоит исключительно из микросервисов. Такая архитектура включает даже объект главного интерфейса, например, веб-приложение. Пример ПО на базе микросервисов показан на рис. 11.1.

В этом примере есть отдельные микросервисы, такие, как *Сервис авторизации*, *Сервис каталога продуктов* и *Сервис инвентаризации продуктов*. Мы создали веб-приложение тоже в виде микросервиса, который использует три отдельных микросервиса через REST API. Мобильное приложение может быть создано с использованием тех же микросервисов через шлюз API. Очевидное преимущество микросервисной архитектуры — возможность повторного использования компонентов. Есть также и другие преимущества:

1. Гибкость при выборе любых технологий и языков программирования в соответствии с индивидуальными требованиями микросервисов. Есть даже возможность использовать устаревший код на любом языке, если предоставить к нему доступ через API-интерфейс.

4. **Интерфейсы взаимодействия**: для взаимодействия следует применять четко определенные интерфейсы микросервисов, желательно REST API или событийно-ориентированные API. Микросервисы должны избегать прямых вызовов друг друга.
5. **Использование шлюза API**: микросервисы и их клиентские *приложения-потребители* должны взаимодействовать друг с другом через шлюз API. Он помогает позаботиться об аспектах безопасности, таких, как аутентификация и балансировка нагрузки, «из коробки». Более того, при появлении новой версии микросервиса можно использовать шлюз API для перенаправления на него клиентских запросов, не влияя на клиентское ПО.
6. **Ограниченный стек технологий**: хотя архитектура позволяет использовать любые фреймворки и языки программирования в отдельных сервисах, не рекомендуется задействовать при разработке разные технологии без веских на то причин. Многообразие стека технологий может быть привлекательно в учебных целях, но на практике усложнит обслуживание и устранение неполадок.
7. **Модель развертывания**: микросервисы не обязательно развертывать в контейнерах, но это рекомендуемый подход. Контейнеры имеют много встроенных возможностей, например, автоматизированное развертывание, кросс-платформенная поддержка и совместимость. Кроме того, с помощью них можно выделять ресурсы в зависимости от требований и обеспечивать справедливое их распределение между микросервисами.
8. **Контроль версий**: для каждого микросервиса требуется отдельная система контроля версий.
9. **Организация команд**: архитектура микросервисов позволяет создавать выделенные команды программистов для каждого микросервиса. Следует помнить об этом принципе при организации команд разработки для крупномасштабного проекта. Количество людей должно основываться на *правиле двух пицц*, согласно которому должно быть столько инженеров, что их можно накормить двумя большими пиццами. Одна команда может заниматься одним или несколькими микросервисами в зависимости от их сложности.
10. **Централизованное логирование/мониторинг**: как уже упоминалось, в приложениях с микросервисной архитектурой поиск неполадок может отнимать много времени, особенно если сервисы работают в контейнерах. Следует использовать открытый исходный код или профессиональные инструменты мониторинга и устранения неполадок для снижения подобных расходов эксплуатации. Несколько примеров таких инструментов: Splunk, Grafana, Elk и App Dynamics.

Далее углубимся в создание приложения с помощью микросервисов.

Создание приложений на базе микросервисов

Прежде чем вдаваться в детали, важно проанализировать несколько фреймворков и вариантов развертывания.

памяти в контейнере или на виртуальной машине, даже если это **Java Virtual Machine (JVM)**.

3. Дополнительные усилия для отладки и устранения неполадок в разных микросервисах, которые могут быть развернуты в отдельных контейнерах или системах.

Далее рассмотрим лучшие практики по созданию микросервисов.

Практические рекомендации по созданию микросервисов

Начиная работу над новым приложением, мы должны ответить на главный вопрос, стоит ли использовать в нем микросервисную архитектуру. Начинать нужно с анализа требований к приложению и возможности разделить их на независимые компоненты. Если прослеживается частая зависимость компонентов друг от друга, это указывает, что разделение приложения на части стоит доработать или вовсе отказаться от микросервисной архитектуры.

Это решение важно принять на начальном этапе. Существует мнение, что лучше начинать разработку с использованием монолитной архитектуры во избежание дополнительных затрат на микросервисы в самом начале. Однако этот подход не рекомендуется. После создания монолитное приложение уже трудно преобразовать в микросервисы, особенно если оно уже развернуто в рабочей среде. Такие компании, как Amazon и Netflix уже перешли на микросервисную архитектуру, но они сделали это в рамках своей эволюции технологий и, разумеется, они имеют достаточно человеческих и технологических ресурсов для такой трансформации.

Если вы уже четко решили задействовать микросервисы, следующие рекомендации помогут в проектировании и развертывании:

1. **Независимость и слабая связь**: эти требования входят в само определение понятия микросервисов. Каждый из них должен быть построен независимо от остальных и иметь максимально слабые связи.
2. **Предметно-ориентированное проектирование (Domain-Driven Design, DDD)**: цель архитектуры не в разделении на максимальное количество небольших микросервисов. Нужно помнить, что каждый сервис имеет свои издержки. Мы должны создавать столько микросервисов, сколько требуется бизнесу (предметной области). Рекомендуем изучить DDD, представленное Эриком Эвансом в 2004 году.
3. Если мы пытаемся применить DDD к микросервисам, это предполагает сначала разработку стратегического проектирования для определения различных контекстов путем объединения связанных бизнес-областей и их подобластей. За стратегическим проектированием может следовать тактическое проектирование, которое фокусируется на разбиении основных предметных областей на детализированные строительные блоки и сущности. Это разделение предоставит четкие рекомендации для сопоставления требований с возможными микросервисами.

Варианты развертывания микросервисов

Как только микросервис создан, следует задать еще один важный вопрос — как его развернуть в качестве изолированной и независимой сущности. В целях обсуждения предположим, что микросервисы созданы с интерфейсами HTTP/REST. Можно развернуть их все на одном веб-сервере как разные веб-приложения или выделить веб-сервер для каждого микросервиса. Один микросервис можно развернуть на одной машине (физической или виртуальной) или на разных машинах и даже в отдельных контейнерах. На следующей схеме (рис. 11.2) описаны все возможные модели развертывания:

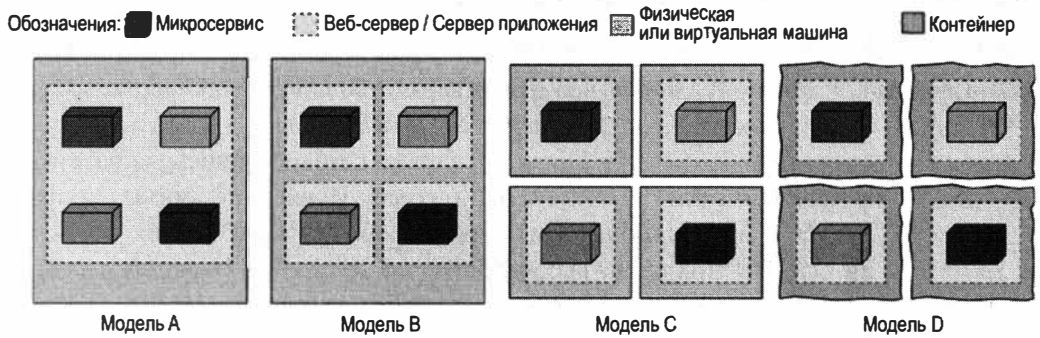


Рис. 11.2. Модели развертывания микросервисов

Рассмотрим подробнее:

- **Модель A**: здесь мы развертываем четыре разных микросервиса на одном веб-сервере. В этом случае есть большая вероятность, что они используют общие библиотеки, которые также расположены на одном веб-сервере. Это может привести к конфликтам библиотек, поэтому такая модель не рекомендуется.
- **Модель B**: здесь четыре микросервиса развернуты на одной машине, но каждому из них выделен свой веб-сервер с целью сделать их независимыми. Эта модель подходит для сред разработки, но может не подходить для производственных сред.
- **Модель C**: здесь для каждого из четырех микросервисов задействованы четыре отдельных виртуальных машины. На каждой машине размещается только один микросервис с веб-сервером. Эта модель подходит для производственных сред, если нет возможности использовать контейнеры. Основным предостережением такой модели являются дополнительные издержки, которые принесет каждая из виртуальных машин.
- **Модель D**: здесь каждый микросервис развернут как контейнер на одной или нескольких машинах. Это не только экономично, но также позволяет соблюсти соответствие спецификациям микросервисов. Такая модель рекомендована, если есть возможность ее использовать.

приложений. Сейчас в этот файл включено только приложение `admin`. Файлы `asgi.py` и `wsgi.py` доступны для запуска веб-сервера ASGI или WSGI, а в файле `settings.py` задается, какой из них будет использоваться.

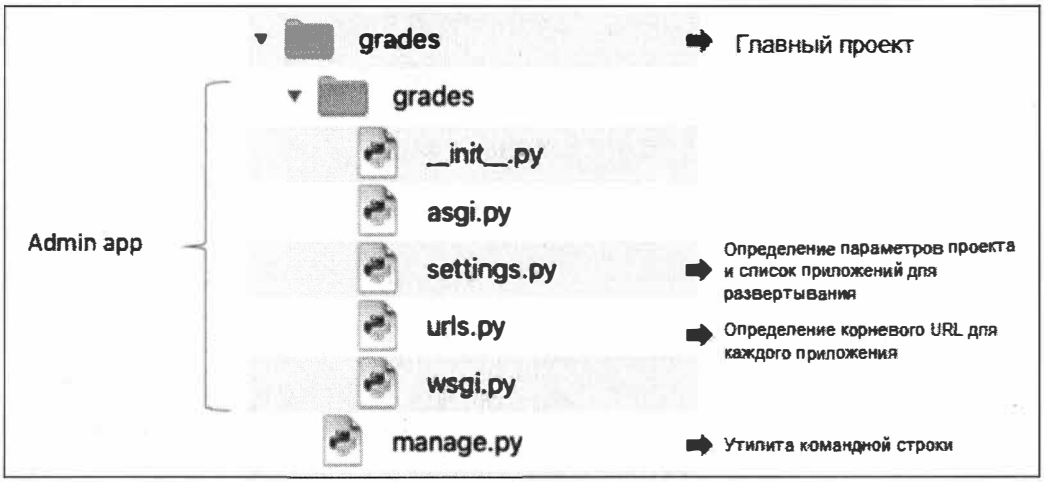


Рис. 11.4. Файловая структура нового проекта Django

7. Создаем новое приложение Django (микросервис `Grades`), выполнив следующую команду в главном каталоге проекта `grades`:

```
python3 manage.py startapp grades_svc
```

8. Команда создаст новое приложение (с веб-компонентами) в отдельном каталоге с именем `grades_svc`, указанным в команде. Также будет создан экземпляр БД `SQLite3` по умолчанию. Параметр, что `SQLite3` будет использоваться по умолчанию, указан в файле `settings.py`, но его можно изменить, если нужно использовать другую БД.

9. К автоматически созданным файлам в каталоге `grades_svc` добавим еще два файла — `urls.py` и `serializers.py`. Полная структура каталогов проекта с двумя дополнительными файлами показана на рис. 11.5 вместе с описанием назначения каждого файла.

10. Далее в эти файлы поочередно добавим код для микросервисов. Начнем с определения класса `Grade`, расширив класс `Model` из пакета БД `models`. Полный код файла `models.py` выглядит так:

```
from django.db import models
class Grade(models.Model):
    grade_id = models.CharField(max_length=20)
    building = models.CharField(max_length=200)
    teacher = models.CharField(max_length=200)

    def __str__(self):
        return self.grade_id
```

- Повторно используем приложение `apiapp` из предыдущей главы; для текущего примера оно будет называться `Students`; этот модуль будет без изменений.
- Обновим приложение `webapp` из предыдущей главы с целью использовать микросервис `Grades` и добавим дополнительные атрибуты `Grade` для каждого объекта `Student`; это также потребует небольших изменений в шаблонах Jinja.

Начнем с создания микросервиса `Grades` с помощью Django.

Создание микросервиса Grades

Для разработки микросервиса мы будем использовать Django Rest Framework (**DRF**). Django использует различные компоненты для создания REST API и микросервисов. Этот пример также даст вам представление о разработке с помощью Django.

Поскольку мы уже знакомы с Flask и основными концепциями веб-разработки, нам будет легко начать использовать Django. Разберем некоторые этапы:

1. Сначала создадим каталог проекта или новый проект в IDE с виртуальной средой. Если вы не используете IDE, можете создать и активировать виртуальную среду в каталоге проекта следующими командами:

```
python -m venv myenv
source myenv/bin/activate
```

2. Жизненно важно создать отдельную виртуальную среду для каждого веб-приложения. Использование глобальной среды для зависимостей библиотек может привести к ошибкам, которые сложно устранить.
3. Для разработки с Django понадобятся, как минимум, две библиотеки, которые можно установить следующими командами `pip`:

```
pip install django
pip install django-rest-framework
```

4. После установки Django можно создать проект с помощью утилиты командной строки `django-admin`. Следующая команда создаст проект `grades` для нашего микросервиса:

```
django-admin startproject grades
```

5. Команда создаст веб-приложение `admin` в каталоге `grades` и добавит файл `manage.py` в проект. Веб-приложение `admin` включает в себя скрипты запуска встроенного веб-сервера, файлы настроек и маршрутизации URL. Как и `django-admin`, `manage.py` также является утилитой командной строки и предлагает аналогичные функции, но в контексте проекта Django. Файловая структура каталога нового проекта будет выглядеть, как представлено на рис. 11.4.
6. Как показано на рис. 11.4, файл `settings.py` содержит параметры на уровне проекта, включая список приложений для развертывания на веб-сервере. Файл `urls.py` содержит информацию о маршрутизации для различных развернутых

```
        serializer = GradeSerializer(grades_list,
            many=True)
        return Response(serializer.data)
    def create(self, request):
        pass:
    def retrieve(self, request, id=None):
        pass:
```

13. Обратите внимание, для полноты микросервиса мы написали методы для добавления нового объекта `Grade` и получения объекта `Grade` по идентификатору в фактической реализации. В примере мы приводим только метод `list`, поскольку это единственный подходящий метод для нашего примера. Также важно подчеркнуть, что объекты представления должны быть реализованы как классы, и в них не следует размещать логику приложения.

После реализации основных методов в `grades_svc` добавим наше приложение в проект Django для развертывания, а также укажем маршруты на уровне приложения и API:

1. Добавим приложение `grades_svc`, а также `rest-framework` в список установленных приложений `INSTALLED_APPS` в файле `settings.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'grades_svc',
    'rest_framework',
]
```

2. Распространенной ошибкой разработчиков является добавление новых компонентов в один файл `settings.py`, поскольку его сложно поддерживать в большом проекте. Лучше разделить его на несколько файлов и загружать их в главный файл настроек.
3. Следующим шагом является добавление конфигурации URL-адреса на уровне приложения `admin`, а затем на уровне нашего приложения. Сначала мы добавим URL-адрес для нашего приложения в файл `urls.py` в приложении `admin`. Это также обеспечит видимость приложения в `admin`:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('grades_svc.urls')),
]
```

4. В файле `urls.py` приложения `admin` мы перенаправляем каждый запрос в наш микросервис, кроме запросов с URL-адреса `admin/`.

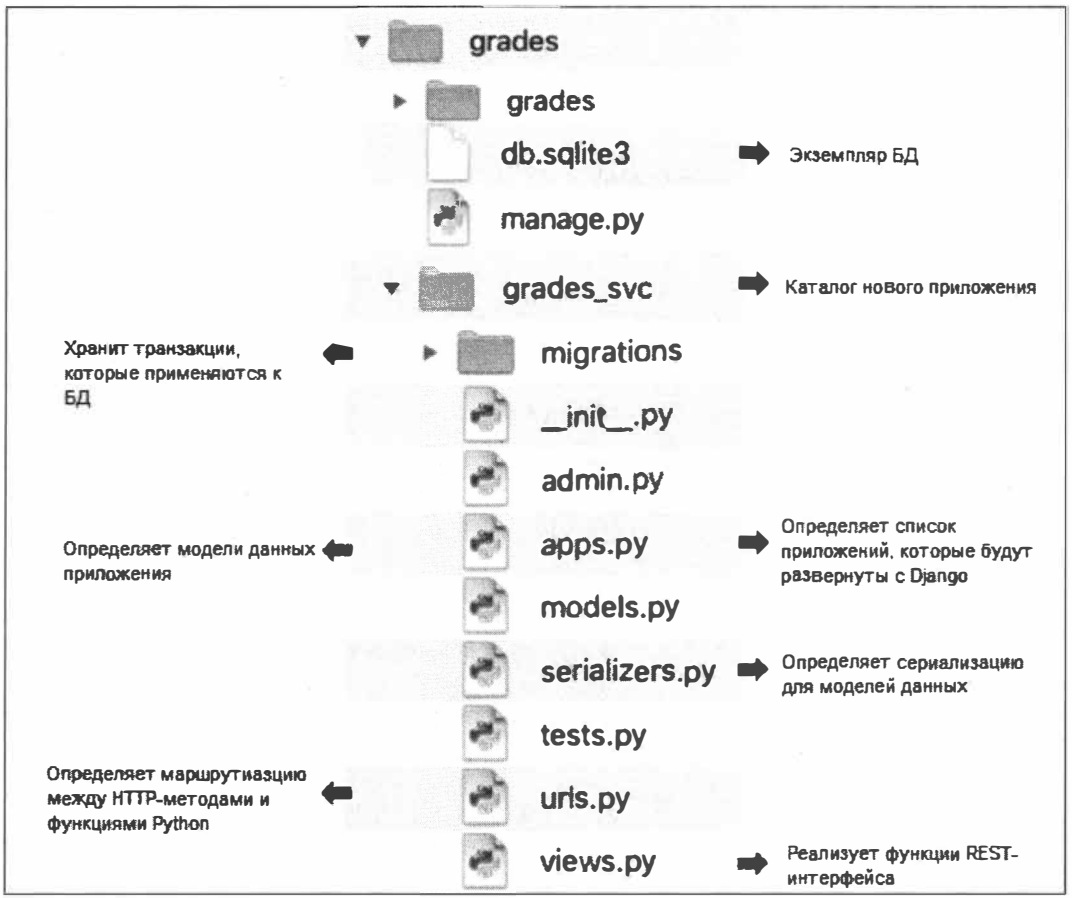


Рис. 11.5. Полная структура каталога приложения grades_svc

11. Нужно сделать модель видимой на панели инструментов приложения `admin`, зарегистрировав класс `Grade` в файле `admin.py`:

```
from django.contrib import admin
from .models import Grade
admin.site.register(Grade)
```

12. Реализуем метод для извлечения списка объектов `Grade` из базы данных. Добавим класс `GradeViewSet`, расширив класс `ViewSet` в файле `views.py`:

```
from rest_framework import viewsets, status
from rest_framework.response import Response
from .models import Grade
from .serializers import GradeSerializer

class GradeViewSet(viewsets.ViewSet):

    def list(self, request):
        grades_list = Grade.objects.all()
```

его в контейнер и запустим как контейнерное приложение. Этому посвящен следующий подраздел.

Контейнеризация микросервиса

Контейнеризация (**Containerization**) — это тип виртуализации ОС, при котором приложения выполняются в отдельных пользовательских пространствах, но используют одну ОС. Отдельное пользовательское пространство называется *контейнером*. Самая популярная платформа для создания, администрирования и выполнения приложений в виде контейнеров — **Docker**. Он занимает более 80% рынка, но существуют также и другие среды выполнения, например, CoreOS rkt, Mesos, lxc и containerd. Прежде чем использовать Docker, рассмотрим основные его компоненты:

- **Docker Engine**: основное приложение Docker для создания, сборки и запуска контейнерных приложений.
- **Образ Docker**: это файл, который используется для запуска приложения в контейнерной среде; приложения, разработанные с помощью Docker Engine, хранятся в виде Docker-образов, которые представляют собой набор кода, библиотек, файлов ресурсов и других зависимостей, необходимых для выполнения приложения.
- **Docker Hub:** онлайн-репозиторий образов, которым можно поделиться с командой или сообществом; также известен как **реестр Docker**.
- **Docker Compose:** инструмент для создания и запуска контейнерных приложений с помощью YAML-файла вместо CLI-команд в Docker Engine; Docker Compose предлагает простой способ развертывания и выполнения нескольких контейнеров с атрибутами конфигурации и зависимостями; для создания и запуска контейнеров рекомендуется использовать Docker Compose или аналогичную технологию.

Для использования Docker Engine и Docker Compose необходимо иметь аккаунт в реестре Docker. Кроме того, перед выполнением следующих шагов нужно загрузить и установить Docker Engine и Docker Compose на компьютер:

1. Сначала создадим список зависимостей проекта:

```
pip freeze -> requirements.txt
```

2. Эта команда создаст список зависимостей и экспортирует их в файл `requirements.txt`, который будет использоваться Docker Engine для загрузки библиотек внутри контейнера поверх интерпретатора Python. Содержимое файла для нашего проекта будет следующим:

```
asgiref==3.4.1
Django==3.2.5
django-rest-framework==0.1.0
djangorestframework==3.12.4
pytz==2021.1
sqlparse==0.4.1
```

5. Следующий шаг — установка маршрутов в нашем приложении на основе различных HTTP-методов. Для этого нужно добавить файл `urls.py` в каталог `grades_svc` со следующими определениями маршрутов:

```
from django.urls import path
from .views import GradeViewSet

urlpatterns = [
    path(grades/', GradeViewSet.as_view({
        'get':'list', #relevant for our sample
            application
        'post':'create'

        }),
    path('grades/<str:id>', GradeViewSet.as_view({
        'get': 'retrieve'
    }))
]
```

6. В этом файле мы присоединяем методы `GET` и `POST` HTTP-запросов с URL-адресом `grades/` к методам `list` и `create` класса `GradeViewSet`, который мы ранее реализовали в файле `views.py`. Точно так же мы присоединяем запрос `GET` с URL-адресом `grades/<str:id>` к методу `retrieve` класса `GradeViewSet`. Используя этот файл, можно добавить дополнительные связывания URL-адресов с функциями/методами Python.

На этом мы завершаем реализацию микросервиса `Grades`. Следующим шагом будет его запуск на веб-сервере Django для проверки работоспособности. Но перед этим убедимся, что объекты модели переданы в БД. Это действие эквивалентно инициализации БД в случае с Flask. В случае с Django нужно выполнить следующие команды для подготовки изменений и последующего их выполнения:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

Часто разработчики пропускают этот важный шаг, и при попытке запустить приложение возникают ошибки. Нужно убедиться, что все изменения выполнены, прежде чем запускать веб-сервер следующей командой:

```
python3 manage.py runserver
```

Команда запустит веб-сервер (порт `8000` по умолчанию) на нашем локальном хосте. Обратите внимание, настройки по умолчанию, включая БД и веб-сервер с атрибутами хоста и порта, могут быть изменены в файле `settings.py`. Кроме того, рекомендуется настроить пользовательский аккаунт для приложения `admin` следующей командой:

```
python3 manage.py createsuperuser
```

Эта команда предложит выбрать имя пользователя, e-mail адрес и пароль для аккаунта администратора. Когда микросервис начнет выполнять функции, мы упакуем

Здесь мы не будем подробно рассматривать работу Docker Engine и Docker Compose, но рекомендуем вам познакомиться с ними поближе в документации по адресу: *https://docs.docker.com/*.

Повторное использование API-приложения Students

Мы будем повторно использовать API-приложение `Students`, которое разработали в предыдущей главе. Оно запустится вместе со своим встроенным сервером и будет называться микросервисом `Students`. В нем не будет никаких изменений.

Обновление веб-приложения Students

Приложение `webapp`, разработанное в предыдущей главе, использует только `apiapp` через REST API. В новой его версии будут использованы микросервисы `Grades` и `Students` для получения списка объектов `Grade` и списка объектов `Student`. Функция `list` (файл `webapp.py`) будет объединять два списка объектов для предоставления веб-клиентам дополнительной информации. Функция будет выглядеть следующим образом:

```
STUDENTS_MS = http://localhost:8080/students
GRADES_MS = "http://localhost:8000/grades"
@app.get('/')
def list():
    student_svc_resp = requests.get(STUDENTS_MS)
    students = json.loads(student_svc_resp.text)

    grades_svc_resp = requests.get(GRADES_MS)
    grades_list = json.loads(grades_svc_resp.text)
    grades_dict = {cls_item['grade']:
                   cls_item for cls_item in grades_list}

    for student in students:
       student['building'] = grades_dict[student['grade']]['building']
       student['teacher'] = grades_dict[student['grade']]['teacher']

    return render_template('main.html', students=students)
```

В новом коде мы создали словарь `grades`, используя *словарное включение* из списка объектов `Grades`. Этот словарь будет использоваться для вставки атрибутов `grade` в объекты `Student` перед их отправкой в шаблон Jinja для отображения. В главном шаблоне (`main.html`) добавлены два дополнительных столбца (`Building` и `Teacher`) в таблицу `Students`, как показано на рис. 11.6.

Мы рассмотрели создание микросервиса, а также его развертывание как в контейнере Docker, так и в качестве веб-приложения на веб-сервере. Кроме того, мы объединили результаты двух микросервисов в веб-приложении.

3. Создадим файл `Dockerfile`, который будет использоваться Docker Engine для создания нового образа контейнера. В нашем случае его содержимое будет следующим:

```
FROM python:3.8-slim
ENV PYTHONUNBUFFERED 1
WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt
COPY . /app
CMD python manage.py runserver 0.0.0.0:8000
```

4. Первая строка устанавливает базовый образ (`Python:3.8-slim`) для контейнера, который уже доступен в репозитории Docker. Вторая строка задает переменную окружения для удобства логирования. Остальные строки говорят сами за себя, поскольку это команды Unix.

5. Далее создадим файл Docker Compose (`docker-compose.yml`):

```
version: '3.7'
services:
  gradesms:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 8000:8000
    volumes:
     - .:/app
```

6. Это YAML-файл, в котором контейнеры определены как сервисы. Поскольку у нас только один контейнер, мы определили сервис `gradesms`. Обратите внимание, `build` указывает на только что созданный `Dockerfile` и предполагает, что тот находится в том же каталоге, что и файл `docker-compose.yml`. Порт контейнера `8000` сопоставляется с портом веб-сервера `8000`. Это важный шаг, который позволяет разрешить трафик в приложение внутри контейнера.

7. Помещаем текущий каталог (`.`) в каталог `/app` внутри контейнера. Это позволит отобразить сделанные в системе изменения в контейнер и наоборот. Этот шаг важен, если мы создаем контейнеры во время цикла разработки.

8. Запустить контейнер можно следующей командой Docker Compose:

```
docker-compose up
```

9. Команда создаст новый образ контейнера. Для загрузки базового образа из реестра Docker потребуется доступ в Интернет. После создания образ автоматически будет запущен.

```
RUN pip install Flask gunicorn
#Запуск веб-сервера при старте контейнера. Мы будем использовать
#gunicorn и привяжем наше api_app как основное приложение
CMD exec gunicorn --bind:$PORT --workers 1 --threads 8
api_app:app
```

3. Этого файла будет достаточно для развертывания приложения в GCP Cloud Run. Но сначала нужно создать образ контейнера с помощью GCP Cloud SDK. Для этого требуется создать проект GCP с помощью Cloud SDK или консоли GCP. Шаги по созданию проекта GCP и привязке к нему биллинг-аккаунта для выставления счетов мы рассматривали в предыдущих главах. Будем предполагать, что у нас уже есть проект с именем `students-run` в GCP.
4. Когда проект готов, можно создать образ контейнера API-приложения `Students` следующей командой:

   ```
   gcloud builds submit --tag gcr.io/students-run/students
   ```

5. Обратите внимание, `gcr` расшифровывается как Google Container Registry (Реестр контейнеров Google).
6. Для создания образа необходимо указать атрибут `tag` в следующем формате:

   ```
   <hostname>/<Project ID>/<Image name>
   ```

7. В нашем случае имя хоста будет `gcr.io`, и он находится в США. Можно также использовать локально созданный образ, но сначала нужно задать атрибут `tag` в указанном выше формате, а затем отправить его в реестр Google. Это можно сделать, выполнив следующие команды Docker:

   ```
   docker tag SOURCE_IMAGE <hostname>/<Project ID>/<Image name>:tagid
   docker push <hostname>/<Project ID>/<Image name>
   #или, если мы хотим отправить конкретный тег
   docker push <hostname>/<Project ID>/<Image name>:tag
   ```

8. Как видно, команда `gcloud build` может выполнить два шага в одной команде.
9. Далее запустим загруженный образ контейнера. Это можно сделать следующей командой Cloud SDK:

   ```
   gcloud run deploy --image gcr.io/students-run/students
   ```

10. Выполнение образа также можно запустить из консоли GCP. Как только конвейер будет успешно развернут и запущен, выходные данные этой команды (или консоли GCP) будут включать URL-адрес микросервиса.

Для использования новой версии микросервиса `Students` из GCP Cloud Run следует обновить веб-приложение, переключившись на URL-адрес нового развернутого сервиса в GCP Cloud Run. Если протестировать веб-приложение с помощью локально развернутого микросервиса `Grades` и удаленно развернуть микросервис `Students`, мы получим те же результаты, что и на рис. 11.6, и сможем выполнить все операции, как если бы микросервис `Students` был развернут локально.

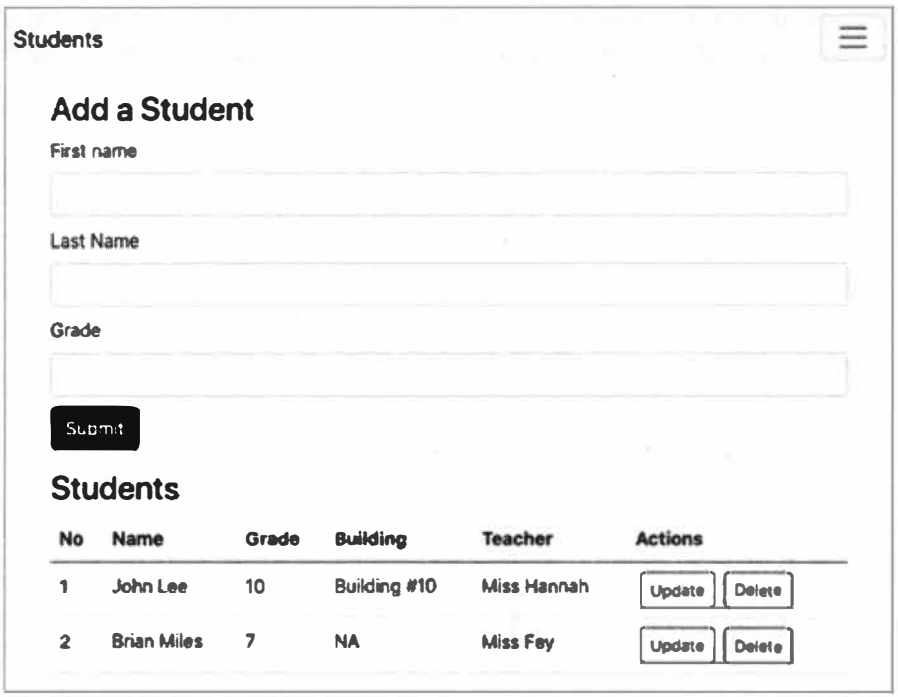


Рис. 11.6. Обновленная главная страница с данными о корпусе (Building) и учителе (Teacher)

Развертывание микросервиса Students в GCP Cloud Run

Пока что мы использовали микросервис `Students` в качестве веб-приложения с REST API, размещенного на сервере Flask. Настало время поместить его в контейнер и развернуть в GCP. Google Cloud Platform имеет движок среды выполнения (Cloud Run) для развертывания контейнеров и запуска их в качестве сервисов (микросервисов). Необходимые для этого шаги описаны ниже:

1. Для упаковки кода `Students` в контейнер сначала нужно определить список зависимостей и экспортировать их в файл `requirements.txt`, выполнив следующую команду в виртуальной среде проекта:

```
pip freeze -> requirements.txt
```

2. Затем создадим `Dockerfile` в корневом каталоге проекта, подобно тому который мы подготовили для микросервиса `Grades`. Содержимое `Dockerfile` будет следующим:

```
FROM python:3.8-slim
ENV PYTHONUNBUFFERED True
WORKDIR /app
COPY . ./
#Установка производственных )рабочих) зависимостей.
RUN pip install -r requirements.txt
```

Ответы

1. Можно, но лучше развернуть его в контейнере.
2. Технически это осуществимо, но это не лучший вариант. Сбой базы данных приведет к остановке обоих микросервисов.
3. Docker Compose — это инструмент для развертывания и запуска контейнерных приложений с помощью файла YAML. Он предоставляет простой формат для определения различных сервисов (контейнеров) с атрибутами развертывания и выполнения.
4. REST API — самый популярный интерфейс обмена данными, но не единственный. Микросервисы также могут использовать RPC и протоколы на основе событий для обмена данными.

На этом можно завершить обсуждение, как создавать микросервисы с помощью различных фреймворков Python, развертывать их локально и в облаке, а также как использовать их в веб-приложении.

Заключение

В этой главе мы познакомились с микросервисной архитектурой и обсудили ее достоинства и недостатки. Рассмотрели рекомендации по созданию, развертыванию и эксплуатации микросервисов. А также проанализировали варианты развертывания, доступные в Python, включая Flask, Django, Falcon, Nameko, Bottle и Tornado. Мы выбрали Flask и Django для создания микросервисов. Для реализации нового примера был использован Django вместе с его Django REST Framework (DRF). В этой реализации микросервиса мы также увидели, как в целом работает фреймворк Django. Затем мы подробно изучили, как упаковать созданный микросервис в контейнер с помощью Docker Engine и Docker Compose. В конце мы преобразовали API-приложение `Students` в образ Docker и развернули его в GCP Cloud Run. Кроме того, мы обновили веб-приложение `Students` и задействовали два микросервиса, развернутых в разных уголках мира.

Примеры кода для этой главы наглядно демонстрируют создание и развертывание микросервисов для разных сред. Эти знания полезны всем, кто хочет создавать приложения на базе микросервисов. В следующей главе мы рассмотрим, как использовать Python для разработки бессерверных функций, которые являются еще одной парадигмой разработки ПО для облака.

Вопросы

1. Можно ли развернуть микросервис без контейнера?
2. Могут ли два микросервиса использовать одну базу данных, но с разной схемой?
3. Что такое Docker Compose и как он помогает развертывать микросервисы?
4. Является ли REST единственным форматом обмена данными для микросервисов?

Дополнительные ресурсы

- *«Hands-On Docker for Microservices with Python»*, автор: Джейми Буэльта (Jaime Buelta).
- *«Python Microservices Development»*, автор: Тарек Зиаде (Tarek Ziade).
- *«Предметно-ориентированное проектирование: структуризация сложных программных систем»*, автор: Эрик Эванс (Eric Evans).
- Краткие руководства по созданию и развертыванию микросервисов в Google Cloud Run: *https://cloud.google.com/run/docs/quickstarts/*.

Технические требования

В этой главе понадобится:

- Python 3.7 или более поздней версии.
- Аккаунт GCP (достаточно бесплатной пробной версии) для развертывания бессерверной функции в Cloud Functions.
- Аккаунт (бесплатный) в SendGrid для отправки электронной почты.

Пример кода для этой главы находится по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter12.

Сначала узнаем, что такое бессерверная функция.

Знакомство с бессерверными функциями

Бессерверная функция — это модель, которая позволяет разрабатывать и выполнять программные компоненты или модули, не беспокоясь о платформе хостинга. Эти модули и компоненты известны как *лямбда-функции* или *облачные функции*. Amazon стал первым провайдером, предложившим их на такой платформе, как AWS Lambda. За ним последовали Google и Microsoft, представив Google Cloud Functions и Azure Functions соответственно.

Как правило, бессерверная функция состоит из четырех компонентов (рис. 12.1):

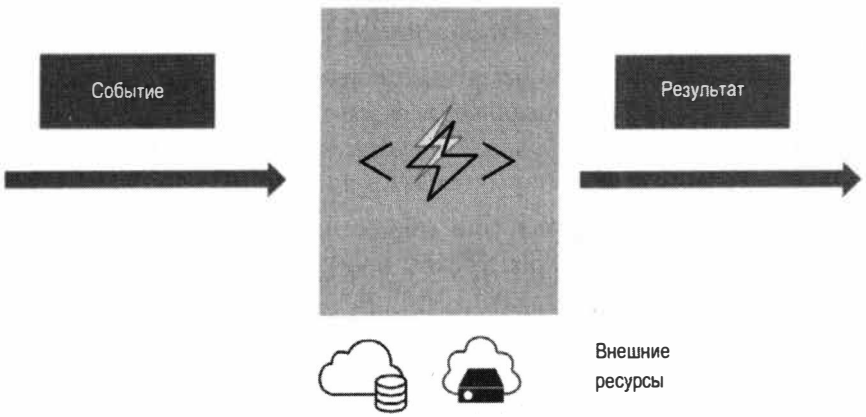


Рис. 12.1. Компоненты бессерверной функции

Рассмотрим компоненты подробнее:

- **Функциональный код**: программный модуль, который выполняет определенные задачи в соответствии с бизнес-целями или функциональными требованиями; например, можно написать функцию для обработки входного потока данных

12

Создание бессерверных функций на Python

Бессерверные вычисления (**Serverless Computing**) — это новая модель облачных вычислений, в которой управление физическими или виртуальными серверами и инфраструктурным ПО (например, СУБД) отделено от самого приложения. Разработчики могут сосредоточиться исключительно на написании кода, пока ресурсами инфраструктуры управляет кто-то другой. Облачные провайдеры — лучший вариант для внедрения этой модели. Контейнеры не только подходят для сложных развертываний, но и являются прорывной технологией для эры бессерверных вычислений. В дополнение к контейнерам существует еще одна форма бессерверных вычислений — *функция как услуга* (**Function as a Service , FaaS**). Согласно этой парадигме, облачные провайдеры предлагают платформу для разработки и запуска функций приложения или *бессерверных функций*, которые выполняются обычно в ответ на событие, или прямой их вызов. Этот сервис предоставляют все публичные облачные провайдеры, например, Amazon, Google, Microsoft, IBM и Oracle. В этой главе основное внимание будет уделено пониманию и созданию бессерверных функций с использованием Python.

Темы этой главы:

- Знакомство с бессерверными функциями.
- Варианты развертывания бессерверных функций.
- Создание бессерверных функций на практическом примере.

Из этой главы вы узнаете о роли бессерверных функций в облачных вычислениях и научитесь писать их на Python.

режиме в реальном времени. В частности, бессерверные функции можно интегрировать с датчиками *Интернета вещей* (**Internet of Things, IoT**), которые исчисляются тысячами. Бессерверные функции способны эффективно обрабатывать запросы от такого количества источников. В мобильном приложении они могут использоваться в качестве *бэкенд-сервиса* (**Backend Service**) для выполнения определенных задач или обработки данных без ущерба для ресурсов мобильного устройства. Еще одним из примеров является виртуальный ассистент Amazon Alexa. Невозможно вместить абсолютно все навыки и интеллект в само устройство, поэтому здесь используются функции Amazon Lambda. Кроме того, они обеспечивают возможность масштабирования в зависимости от спроса. Некоторые функции могут использоваться чаще других, например, запросы на прогноз погоды.

В следующем разделе рассмотрим различные варианты развертывания бессерверных функций.

Варианты развертывания бессерверных функций

Использование виртуальной машины или другого ресурса среды выполнения в публичных облаках для приложений, к которым обращаются нерегулярно, может оказаться коммерчески непривлекательным решением. В таких ситуациях на помощь приходят бессерверные функции. Облачный провайдер предлагает динамически управляемые ресурсы для приложения и взимает плату только за время их фактического использования в ответ на событие. Иными словами, бессерверная функция — это метод бэкенд-вычислений, который предоставляется по запросу, оплачивается по факту использования и предлагается только в публичных облаках. Мы рассмотрим несколько вариантов развертывания бессерверных функций в облаке:

1. **AWS Lambda**: это считается одним из первых подобных сервисов. Функции AWS Lambda можно писать на Python, Node.js, PowerShell, Ruby, Java, C# и Go. Они могут выполняться в ответ на такие события, как загрузка файлов в хранилище Amazon S3, уведомление от Amazon SNS или прямой вызов API. Функции AWS Lambda не хранят состояние.
2. **Azure Functions:** были представлены почти через два года после запуска AWS Lambda. Эти функции можно привязывать к событиям в облачной инфраструктуре. Microsoft поддерживает их создание и отладку с помощью Visual Studio, Visual Studio Code, IntelliJ и Eclipse. Функции Azure можно писать на C#, F#, Node.js, PowerShell, PHP и Python. Кроме того, Microsoft предлагает *устойчивые функции* (**Durable Functions**), которые позволяют писать функции с отслеживанием состояния в бессерверном окружении.
3. **Google Cloud Functions:** GCP предлагает Google Cloud Functions как бессерверные функции, которые могут быть написаны на Python, Node.js, Go, .NET, Ruby и PHP. Как и конкуренты, облачные функции от Google могут запускаться

или создать запланированное действие для проверки определенных ресурсов в целях мониторинга.

- **События**: бессерверные функции не предназначены для использования в качестве микросервисов; они запускаются триггером, который может быть инициирован событием в системе *«издатель-подписчик»* (**Publish-Subscribe system, Pub/Sub system**), или поступать как HTTP-вызов на основе внешнего события, такого, как срабатывание датчика.
- **Результат**: когда бессерверная функция запущена и выполняет свою задачу, результатом может быть или простой ответ вызывающему объекту, или запуск другого действия для смягчения последствий изначального события; примером такого результата может служить запуск другого облачного сервиса, например, службы СУБД, или отправка электронной почты.
- **Ресурсы**: иногда функциональный код использует дополнительные ресурсы для выполнения задачи, например, сервис БД или облачное хранилище для доступа к файлам.

Преимущества бессерверных функций

Бессерверные функции позволяют использовать все преимущества бессерверных вычислений:

- **Простота разработки**: они избавляют разработчиков от инфраструктурных сложностей, позволяя сосредоточиться на функциональном аспекте программы.
- **Встроенная масштабируемость**: они поставляются со встроенной масштабируемостью, позволяющей справиться с любым ростом трафика в любое время.
- **Эффективность затрат**: они не только снижают расходы на разработку, но также предлагают оптимизированное развертывание и рабочий режим; как правило, это модель *оплаты по факту использования*, то есть плата взимается только за время, в течение которого выполняется функция.
- **Независимость от технологий**: они совместимы с любыми технологиями. их можно создавать на множестве языков программирования с использованием различных облачных ресурсов.

Обратите внимание, бессерверные функции имеют также некоторые ограничения. Например, на уровне системы контроль будет меньше, а устранение неполадок — сложным.

Варианты использования

Существует несколько возможных вариантов применения бессерверных функций. Например, их можно использовать для обработки данных, когда получено событие загрузки файла в облачное хранилище или когда данные поступают в потоковом

Создание облачной функции на основе HTTP с помощью консоли GCP

Сначала создадим простую облачную функцию, которая предоставляет текущую дату и время для HTTP-триггера. Обратите внимание, HTTP-триггер — это самый простой способ вызова облачной функции. Для начала нам потребуется проект GCP. Можно создать новый на консоли GCP или использовать уже имеющийся. Инструкции по созданию проекта и привязке биллинг-аккаунта приведены в главе 9 (*«Программирование на Python для облака»*). После подготовки проекта создание новой облачной функции будет состоять из трех шагов.

Настройка атрибутов облачной функции

При инициировании рабочего процесса **Create Function** на консоли GCP нам предлагается указать определение облачной функции следующим образом (рис. 12.2):

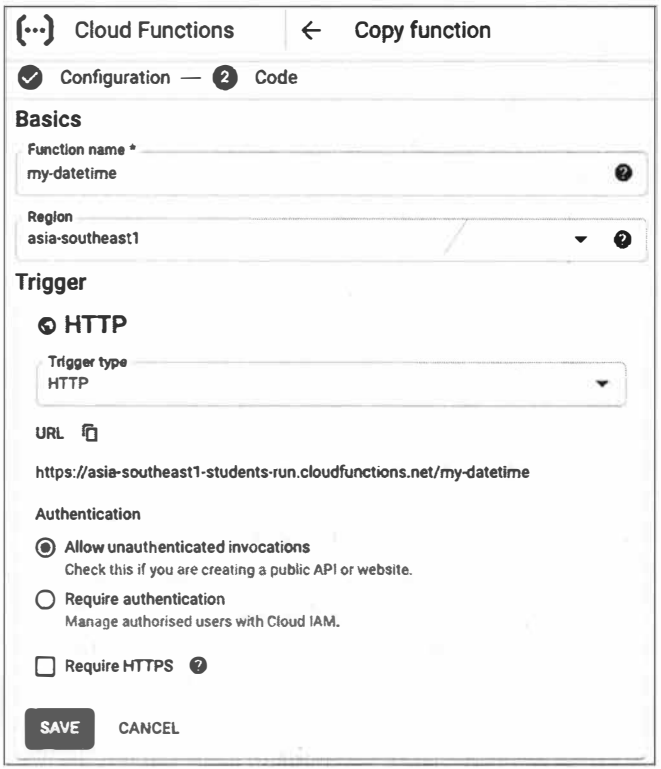


Рис. 12.2. Действия по созданию облачной функции на консоли GCP (1/2)

Определение облачной функции в общих чертах:

1. Мы указываем **Function Name** (в нашем случае это `my-datetime`) и выбираем **Region** для размещения функции.

HTTP-запросами или событиями в инфраструктуре Google Cloud. Google позволяет использовать Cloud Build для автоматического тестирования и развертывания Cloud Functions.

Существуют и другие облачные провайдеры, которые предлагают свои решения для бессерверных функций. Например, IBM предлагает Cloud Functions на базе проекта Apache OpenWhisk с открытым исходным кодом. Oracle предлагает свою платформу бессерверных вычислений на основе проекта Fn, который также имеет открытый исходный код. Прелесть таких проектов в возможности разрабатывать и тестировать код локально. Кроме того, они позволяют без изменений переносить код из одного облака в другое или даже в локальное развертывание.

Стоит упомянуть еще один известный фреймворк для бессерверных вычислений — Serverless Framework. Это не платформа развертывания, а программный инструмент, который можно использовать локально при создании и упаковке кода для бессерверного развертывания, а затем использовать этот пакет для развертывания в любом публичном облаке. Этот фреймворк поддерживает такие языки программирования, как Python, Java, Node.js, Go, C#, Ruby и PHP.

В следующем подразделе мы напишем несколько бессерверных функций на Python.

Написание бессерверных функций

Здесь мы узнаем, как создавать бессерверные функции в одном из публичных облаков. Несмотря на то что пионером бессерверных функций стал Amazon AWS, представив в 2014 году AWS Lambda, в наших примерах мы будем использовать платформу Google Cloud Functions. Такой выбор обусловлен тем, что мы уже хорошо знакомы с этой платформой из предыдущих глав и можно использовать тот же аккаунт GCP для дальнейшего развертывания бессерверных функций. Однако мы настоятельно рекомендуем изучить и другие платформы, особенно если вы планируете использовать их бессерверные функции в дальнейшем. Основные принципы построения и развертывания на разных платформах аналогичны.

GCP Cloud Functions предлагает несколько способов разработки и развертывания бессерверных функций (в дальнейшем в контексте GCP мы будем называть их *облачными функциями*). В нашем примере мы рассмотрим два типа событий, которые можно описать следующим образом:

- Первая облачная функция будет построена и развернута с помощью консоли GCP Console; она будет активироваться HTTP-вызовом (или событием).
- Вторая облачная функция станет частью примера приложения, которое отслеживает событие в облачной инфраструктуре и в ответ на событие выполняет отправку электронной почты; в этом примере мы будем использовать Cloud SDK.

Начнем создание облачной функции с помощью консоли GCP.

Создание облачной функции на основе HTTP с помощью консоли GCP

Сначала создадим простую облачную функцию, которая предоставляет текущую дату и время для HTTP-триггера. Обратите внимание, HTTP-триггер — это самый простой способ вызова облачной функции. Для начала нам потребуется проект GCP. Можно создать новый на консоли GCP или использовать уже имеющийся. Инструкции по созданию проекта и привязке биллинг-аккаунта приведены в главе 9 (*«Программирование на Python для облака»*). После подготовки проекта создание новой облачной функции будет состоять из трех шагов.

Настройка атрибутов облачной функции

При инициировании рабочего процесса **Create Function** на консоли GCP нам предлагается указать определение облачной функции следующим образом (рис. 12.2):

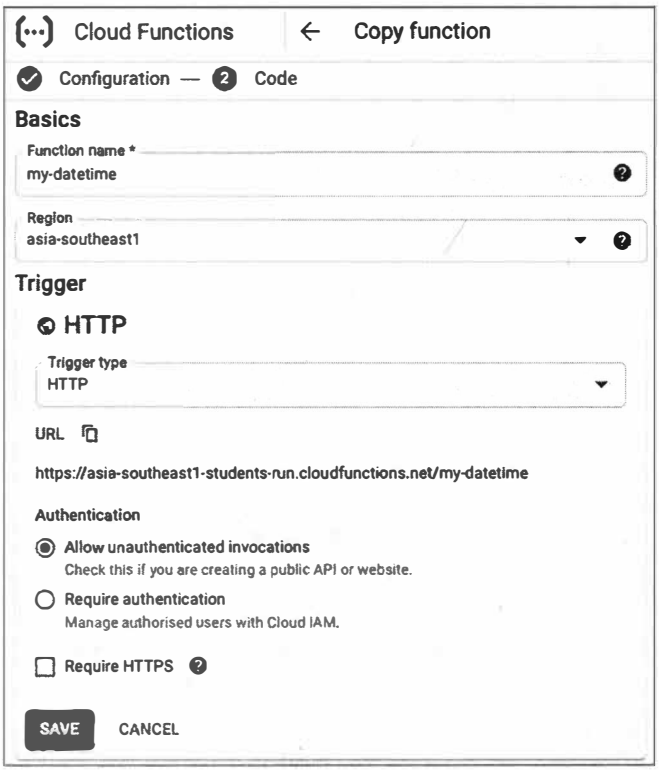


Рис. 12.2. Действия по созданию облачной функции на консоли GCP (1/2)

Определение облачной функции в общих чертах:

1. Мы указываем **Function Name** (в нашем случае это `my-datetime`) и выбираем **Region** для размещения функции.

2. Для параметра **Trigger type** выбираем `HTTP`. Выбор триггера для функции — самый важный шаг. Есть и другие доступные триггеры, например, **Cloud Pub/Sub** и **Cloud Storage**. На момент написания книги в GCP были добавлены еще несколько триггеров для ознакомления.
3. Для простоты мы не будем включать аутентификацию доступа к нашей функции.

После нажатия на кнопку **Save** будет предложено ввести параметры **RUNTIME, BUILD AND CONNECTIONS SETTINGS** (для настройки *выполнения*, *сборки* и *подключений* соответственно) (рис. 12.3):

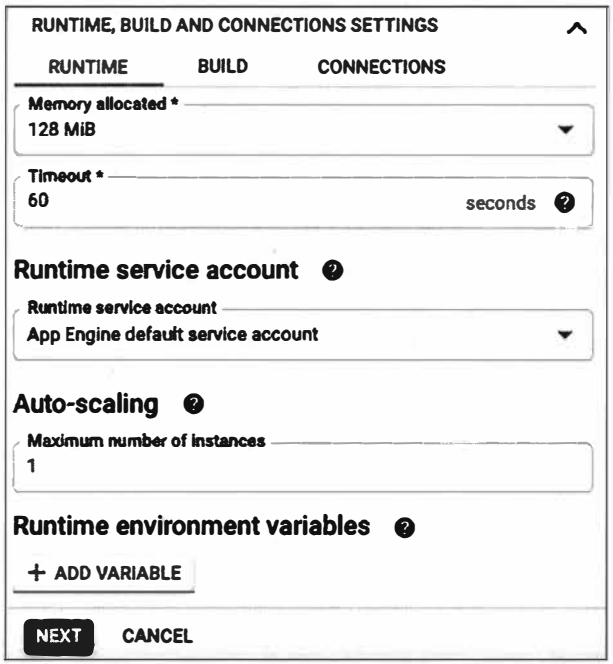


Рис. 12.3. Действия по созданию облачной функции на консоли GCP (2/2)

Эти параметры можно указать следующим образом:

1. **RUNTIME**-атрибуты можно оставить по умолчанию, кроме **Memory allocated**, который будет уменьшен до **128 MiB**. Мы выбрали **App Engine default service account** в параметре **Runtime service account**. Для **Auto-scaling** мы оставим значение по умолчанию, но можно задать максимальное количество экземпляров.
2. На вкладке **RUNTIME** можно добавить *переменные окружения* (**Runtime environment variables**), если это необходимо. Для нашей функции мы не будем их добавлять.
3. На вкладке **BUILD** можно добавить переменные окружения для сборки (**Build environment variables**), но мы также не будем ничего добавлять.
4. На вкладке **CONNECTIONS** можно оставить параметры по умолчанию и, таким образом, разрешить всему трафику доступ к облачной функции.

После настройки выполнения, сборки и подключений следующим шагом будет добавление кода реализации.

Добавление кода Python в облачную функцию

После нажатия на кнопку **Next**, как показано на рис. 12.3, GCP Console предложит определить или добавить детали реализации функции (рис. 12.4):

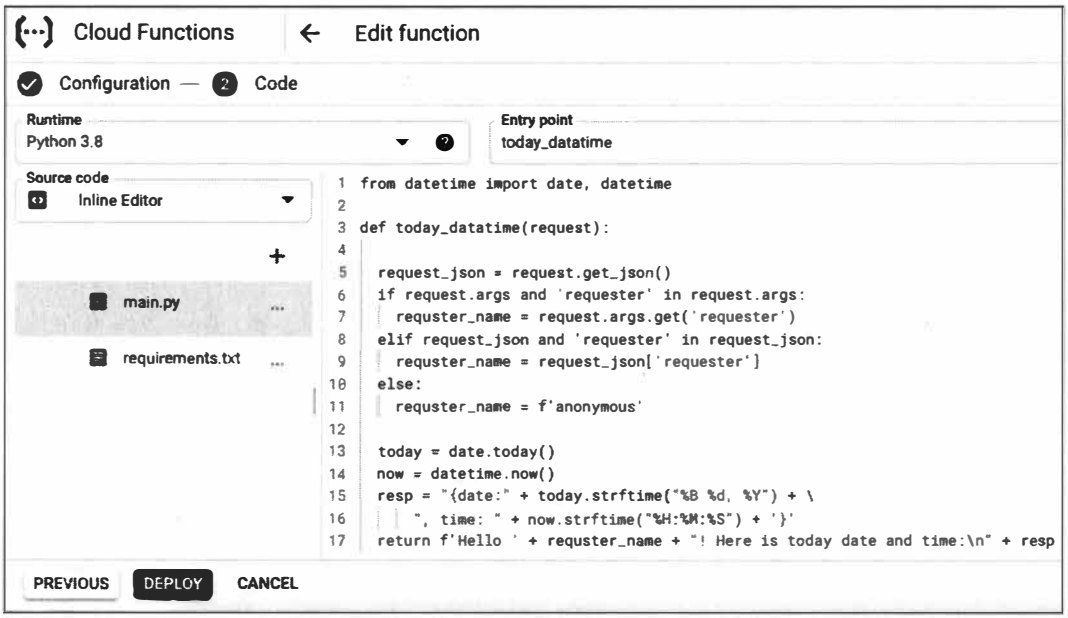


Рис. 12.4. Реализация облачной функции на консоли GCP

Варианты для добавления кода Python:

1. Можно выбрать несколько вариантов среды выполнения (параметр **Runtime**), например, Java, PHP, Node.js или различные версии Python; мы выбрали **Python 3.8**.
2. Атрибут **Entry point** должен быть именем функции в коде. Google Cloud Functions будет вызывать функцию в коде на основе этого атрибута.
3. Исходный код Python может быть добавлен с помощью встроенного редактора **Inline Editor** справа. Его также можно загрузить, используя ZIP-файл с локального компьютера или из облачного хранилища. Или указать расположение репозитория GCP **Cloud Source** для исходного кода. В примере мы будем использовать **Inline Editor**;
4. Для Python платформа GCP Cloud Functions автоматически создает два файла: `main.py` и `requirements.txt`. Первый будет содержать код, а второй — зависимости от сторонних библиотек.

5. Пример кода, показанный в **Inline Editor**, сначала проверяет, отправил ли вызывающий объект атрибут `requester` в HTTP-запросе. На основе значения этого атрибута будет отправлено сообщение с текущей датой и временем. Мы реализовали аналогичный пример кода с двумя отдельными веб-API, используя Flask в главе 9 (*«Программирование на Python для облака»*) для демонстрации возможностей GCP App Engine.

Как только код написан, развернем функцию на платформе Google Cloud Functions.

Развертывание облачной функции

Следующий шаг — развернуть функцию, нажав на кнопку **Deploy**, как показано на рис. 12.4. GCP начнет развертывание немедленно. Это займет несколько минут. Важно понимать, Google Cloud Functions развертываются с помощью контейнеров так же как и микросервисы в GCP Cloud Run. Ключевое отличие в том, что их можно вызывать с помощью разных типов событий, и они используют модель оплаты по мере использования.

После развертывания функцию можно продублировать, протестировать или удалить из списка Cloud Functions, как показано на скриншоте (рис. 12.5):



Рис. 12.5. Главное представление Google Cloud Functions

Далее кратко рассмотрим, насколько удобно тестировать и исправлять неполадки облачной функции с помощью GCP Console. Если нажать на кнопку **Test function**, консоль откроет тестовую страницу на вкладке **TESTING** (рис. 12.6). Для тестирования функции передадим атрибут `requester` в формате JSON:

```
{"requester":"John"}
```

Нажав на **[...]TEST THE FUNCTION**, можно просмотреть результаты в разделе **Output** и записи журнала в разделе **Logs** в нижней части экрана (рис. 12.6). По-

скольку используется HTTP-триггер, также можно протестировать функцию с помощью браузера или утилиты `CURL` прямо из Интернета. Однако следует убедиться, что облачная функция включает члена `allUsers` с ролью `Cloud Functions Invoker`. Эти параметры можно настроить на вкладке **PERMISSIONS**, но не рекомендуется этого делать без настройки механизма аутентификации:

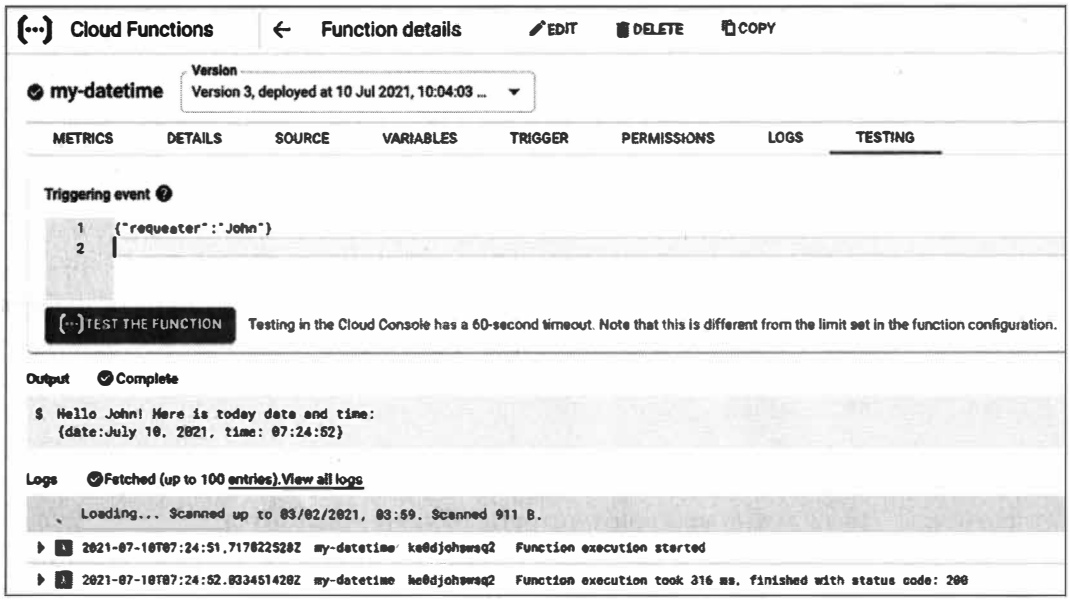


Рис. 12.6. Тестирование облачной функции с помощью консоли GCP

Создать облачную функцию с помощью консоли GCP довольно просто. Далее рассмотрим пример реального применения облачных функций.

Практический пример: создание приложения для уведомлений о событиях в облачном хранилище

В этом примере мы создадим облачную функцию, которая запускается для событий в **Google Storage bucket**. При получении события функция отправит уведомление по электронной почте списку адресатов. Схема приложения выглядит следующим образом (рис. 12.7):



Рис. 12.7. Облачная функция, ожидающая события в Google Storage bucket

Обратите внимание, облачная функция может быть настроена на прослушивание одного или нескольких событий. Google Cloud Functions поддерживает следующие события в Google Storage:

- **finalize**: это событие создается при добавлении или замене нового файла в хранилище;
- **delete**: это событие представляет собой удаление файла из хранилища; это относится к сегментам без контроля версий; обратите внимание, файл на самом деле не удаляется, а архивируется, если для сегмента хранилища все же настроен контроль версий;
- **archive**: это событие возникает при архивировании файла; операция архивации запускается, когда файл удаляется или перезаписывается для сегментов хранилища с контролем версий;
- **metadata update**: это событие возникает при обновлении метаданных файла.

Получив событие от Google Storage bucket, облачная функция извлекает атрибуты из объектов `context` и `event`, переданных ей в качестве аргументов. Затем она использует стороннюю службу электронной почты (например, SendGrid от Twilio) для отправки уведомлений.

Предварительно у вас уже должен быть создан бесплатный аккаунт SendGrid (*https://sendgrid.com/*). В аккаунте должен быть создан хотя бы один пользователь-отправитель. Также нужно настроить секретный ключ API, который можно использовать вместе с облачной функцией для отправки электронных писем. Twilio SendGrid позволяет бесплатно отправлять до 100 сообщений в день, этого будет достаточно для тестирования.

В примере мы напишем код для облачной функции локально, а затем развернем его на платформе Google Cloud Functions с помощью Cloud SDK. Мы будем делать это пошагово, как показано ниже:

1. Создадим сегмент хранения для привязки к нашей облачной функции и будем загружать или удалять файлы из него для генерации событий. Создать новый сегмент можно следующей командой Cloud SDK:

```
gsutil mb gs://<bucket name>
gsutil mb gs://muasif-testcloudfn    #Образец сегмента создан
```

2. Для упрощения генерации событий отключим контроль версий в этом сегменте следующей командой:

```
gsutil versioning set off gs://muasif-testcloudfn
```

3. Когда сегмент хранилища будет готов, создадим локальный каталог проекта и настроим виртуальную среду:

```
python -m venv myenv
source myenv/bin/activate
```

4. Затем установим пакет `sendgrid` для Python с помощью утилиты `pip`:

```
pip install sendgrid
```

5. После установки сторонних библиотек необходимо создать файл с зависимостями requirements.txt:

```
pip freeze -> requirements.txt
```

6. Далее создадим новый файл main.py с функцией handle_storage_event. Она будет точкой входа для облачной функции. Код в ней выглядит следующим образом:

```
#main.py
from sendgrid import SendGridAPIClient
from sendgrid.helpers.mail import Mail, Email, To, Content
def handle_storage_event(event, context):
    from_email = Email("abc@domain1.com")
    to_emails = To("zyz@domain2.com")
    subject = "Your Storage Bucket Notification"
    content = f"Bucket Impacted:{event['bucket']} \n" + \
              f"File Impacted: {event['name']} \n " + \
              f"Event Time: {event['timeCreated']} \n" + \
              f"Event ID: {context.event_id} \n" + \
              f"Event Type: {context.event_type}"
  mail = Mail(from_email, to_emails, subject, content)
  sg = SendGridAPIClient()
  response = sg.send(mail)
  print(response.status_code) # for logging purpose
  print(response.headers)
```

7. Ожидается, что функция handle_storage_event будет получать объекты event и context в качестве входных аргументов. Объект event — это словарь, который содержит данные события. Доступ к этим данным можно получить по ключам: bucket, name и timeCreated. Объект context предоставляет контекст события, например, event_id и event_type. Кроме того, мы используем библиотеку sendgrid для подготовки содержимого электронных сообщений, а затем отправки их с информацией о событии списку контактов.

8. Подготовив файлы main.py и requirements.txt, можно инициировать операцию развертывания следующей командой Cloud SDK:

```
gcloud functions deploy handle_storage_create \
--entry-point handle_storage_event --runtime python38 \
--trigger-resource gs://muasif-testcloudfn/\
--trigger-event google.storage.object.finalize
--set-env-vars SENDGRID_API_KEY=<Your SEND-GRID KEY>
```

9. Эту команду нужно выполнить в проекте GCP с включенным биллингом, как обсуждалось ранее. Мы указали имя облачной функции handle_storage_create, а атрибуту entry-point задали функцию handle_storage_event в коде Python. Для события finalize задали trigger-event. С помощью set-env-vars задали SENDGRID_API_KEY для сервиса SendGrid.

10. Команда `deploy` упакует код Python из текущего каталога, подготовит целевую платформу с учетом зависимостей из файла `requirements.txt`, а затем развернет код на платформе GCP Cloud Functions. В нашем случае мы также можем создать файл `.gcloudignore` для исключения файлов и каталогов, которые команда `deploy` должна игнорировать.
11. После развертывания нашей функции мы можем протестировать ее, загрузив файл с локального компьютера в сегмент хранилища следующей командой:

```
gsutil cp test1.txt gs://muasif-testcloudfn
```

12. Как только копирование будет завершено, событие `finalize` запустит облачную функцию. В результате мы получим электронное письмо с информацией о событии. Также можно проверить логи Cloud Functions следующей командой:

```
gcloud functions logs read --limit 50
```

В этом приложении облачная функция привязана только к событию `finalize`. Это обусловлено тем, что можно привязать одну облачную функцию только к одному событию. Однако функция — это объект развертывания, а не сам программный код. Нам не нужно снова писать или дублировать код Python для обработки событий другого типа. Если мы хотим получать уведомления о других типах событий (например, `Delete`), можно создать новую облачную функцию, используя тот же код, но уже для нового события:

```
gcloud functions deploy handle_storage_delete \
--entry-point handle_storage_event --runtime python38 \
--trigger-resource gs://muasif-testcloudfn/ \
--trigger-event google.storage.object.delete
--set-env-vars SENDGRID_API_KEY=<Your SEND-GRID KEY>
```

Обратите внимание на эту версию команды `deploy`. Мы изменили только имя облачной функции и тип события-триггера. Команда создаст новую облачную функцию, которая будет работать параллельно с предыдущей , но будет запускаться уже по другому событию (в этом случае `delete`).

Можно протестировать событие `delete` с новой облачной функцией, удалив загруженный файл (или любой другой файл) из сегмента хранилища следующей командой Cloud SDK:

```
gsutil rm gs://muasif-testcloudfn/test1.txt
```

Используя тот же код Python, можно создавать больше облачных функций для других событий.

На этом можно завершить обсуждение, как создавать облачные функции для событий в хранилище. Все описанные для Cloud SDK шаги можно реализовать и на GCP Console.

Заключение

В этой главе мы познакомились с бессерверными вычислениями и моделью FaaS, а также проанализировали основные составляющие бессерверных функций. Мы рассмотрели их ключевые преимущества и недостатки. Кроме того, изучили некоторые варианты развертывания для бессерверных функций: AWS Lambda, Azure Functions, Google Cloud Functions, Oracle Fn и IBM Cloud Functions. В заключительной части главы мы создали простую облачную функцию Google Cloud Functions на основе HTTP-триггера с помощью GCP Console. В конце мы создали приложение для уведомлений о событиях в хранилище Google и облачную функцию с помощью Cloud SDK.

Примеры кода к этой главе помогут понять, как использовать GCP Console и Cloud SDK для создания и развертывания облачных функций. Эти знания понадобятся всем, кто хочет развиваться в сфере бессерверных вычислений.

В следующей главе мы рассмотрим, как использовать Python с *машинным обучением* (**Machine Learning**).

Вопросы

1. Чем бессерверные функции отличаются от микросервисов?
2. Какое практическое применение бессерверных функций в реальных примерах?
3. Что такое устойчивые функции и какой облачный провайдер их предлагает?
4. Может ли одна облачная функция быть привязана к нескольким триггерам?

Дополнительные ресурсы

- *«Serverless Computing with Google Cloud»*, автор: Ричард Роуз (Richard Rose).
- *«Mastering AWS Lambda»*, автор: Йохан Вадия (Yohan Wadia).
- *«Mastering Azure Serverless Computing»*, авторы: Лоренцо Барбьери (Lorenzo Barbieri) и Массимо Бонанни (Massimo Bonanni).
- Руководства по созданию облачных функций на платформе Google Cloud Functions *https://cloud.google.com/functions/docs/quickstarts*.

Ответы

1. Это разные варианты бессерверных вычислений. Обычно бессерверные функции запускаются событием и оплачиваются по мере использования. Микросервисы

обычно используются через вызовы API, и модель оплаты по мере использования к ним не применяется.

2. Amazon Alexa использует функции AWS Lambda для предоставления своим пользователям различных возможностей.
3. Устойчивые функции — это расширение Azure Functions от Microsoft, которое предлагает возможность отслеживать состояния в бессерверном окружении.
4. Нет, не может. Одну облачную функцию можно привязать только к одному триггеру.

13

Python и машинное обучение

Машинное обучение **(Machine Learning, ML)** — это построение моделей, основанное на *искусственном интеллекте* (ИИ, **Artificial Intelligence**, **AI**) и на изучении закономерностей в данных, а также использование этих моделей для прогнозирования. Это одна из самых популярных методик ИИ, которая используется во многих сферах, например, в медицинской диагностике, обработке изображений, распознавании речи, прогнозировании угроз, сборе данных, классификации и многих других. Важность и польза машинного обучения очевидны. Python, будучи лаконичным, но очень мощным языком, широко используется для реализации моделей ML. Язык имеет в своем арсенале такие библиотеки обработки и подготовки данных, как NumPy, Pandas и PySpark, что делает его предпочтительным выбором для разработчиков при создании и обучении моделей.

В этой главе мы рассмотрим, как использовать Python для задач машинного обучения оптимизированным способом. Это особенно важно, поскольку обучение моделей требует больших вычислительных ресурсов, а оптимизация кода имеет основополагающее значение при выборе Python для задач ML.

Темы этой главы:

- Введение в машинное обучение.
- Использование Python для машинного обучения.
- Тестирование и оценка моделей машинного обучения.
- Развертывание моделей машинного обучения в облаке.

Из этой главы вы узнаете, как использовать Python для создания, обучения и оценки моделей ML, а также развертывать их в облаке и использовать для прогнозирования.

Технические требования

Для этой главы понадобятся:

- Python 3.7 или более поздней версии.
- Дополнительные библиотеки машинного обучения — SciPy, NumPy, Pandas и scikit-learn.
- Аккаунт GCP для развертывания модели на платформе AI Platform (достаточно будет бесплатной пробной версии).

Примеры кода для этой главы находятся по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter13.

Начнем со знакомства с машинным обучением.

Введение в машинное обучение

В традиционном программировании мы предоставляем коду данные и набор правил их обработки для получения желаемого результата. Машинное обучение — принципиально иной подход, в котором мы предоставляем данные и желаемый результат в качестве входных данных для получения набора правил. В машинном обучении это называется *моделью*. Концепция представлена на следующей схеме (рис. 13.1):

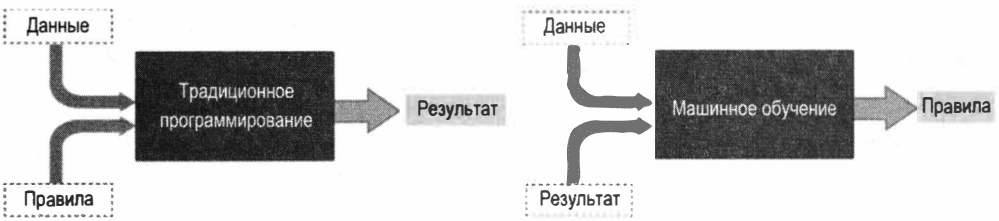


Рис. 13.1. Традиционное программирование и машинное обучение

Для понимания, как работает ML, необходимо познакомиться с его основными компонентами:

1. **Набор данных:** без хорошего набора ничего не получится. Данные — настоящая сила ML. Они должны иметь широкое разнообразие и охватывать самые разные ситуации для представления модели, наиболее близкой к реальному процессу или системе. Еще одно требование — данных должно быть много (порядка нескольких тысяч записей). Кроме того, они должны быть максимально точными и содержать значимую информацию. Данные используются для обучения системы, а также оценки ее точности. Информацию можно собирать из многих источников, и в большинстве из них они представлены в необработанном виде (*сырые данные*). Но их можно обработать, используя библиотеки вроде Pandas, как уже упоминалось ранее.

2. **Извлечение признаков**: Перед использованием данных для создания модели нужно понять, какого они типа и как структурированы. Как только это сделано, можно выбрать, какие признаки будут использованы алгоритмом для построения модели. Также можно вычислить дополнительные признаки на основе исходного набора. Например, если есть необработанное изображение в виде пикселей, само по себе оно может не подойти для обучения. Но можно использовать длину или ширину формы внутри изображения в качестве признаков для формирования правил.
3. **Алгоритм**: Это программа, которая используется для создания модели из доступных данных. С математической точки зрения алгоритм пытается изучить целевую функцию *f(X)*, которая выражает отношение входных данных (*X*) к выходным данным (*y*) следующим образом:

$$y=f(X).$$

4. Существует несколько алгоритмов для разных типов задач и ситуаций, поскольку универсального подхода нет. Среди популярных можно выделить *линейную регрессию*, *деревья классификации и регрессии*, а также *метод опорных векторов*. В этой книге мы не будем рассматривать математические подробности работы алгоритмов. Но советуем для ознакомления изучить источники из раздела дополнительных ресурсов в конце главы.
5. **Модель**: это математическое или машинное представление процесса, происходящего в повседневной жизни. С точки зрения машинного обучения это результат работы алгоритма, который мы применяем к набору данных. Результат (модель) может представлять собой набор правил или определенную структуру данных, которые можно использовать для прогнозирования любых реальных событий.
6. **Обучение**: это не новый компонент или шаг машинного обучения. Говоря про обучение, мы имеем в виду применение алгоритма ML к набору данных для создания модели. Говорят, что модель, которую мы получаем на выходе, обучена на определенном наборе данных. Существуют три способа обучения:

   - **Обучение с учителем:** мы предоставляем не только набор данных, но и желаемый результат. Цель состоит в определении, как вход (*X*) сопоставлен с выходом (*y*) на основе имеющихся данных. Такой подход используется в *классификации* и *регрессии*. Примерами таких задач являются классификация изображений или прогнозирование цен на жилье (регрессия). В первом случае можно научить модель определять вид животного на изображении (например, кошка или собака), на основе формы, длины и ширины объекта. Для этого надо пометить каждое изображение в обучающем наборе названием животного. Во втором случае для прогнозирования цен необходимо предоставить данные о домах в районе, где они находятся, например, расположение, количество комнат и санузлов и т. д.

   - **Обучение без учителя**: в этом случае мы обучаем модель, не зная желаемого результата. Обучение без учителя обычно применяется в задачах *кластериза-*

ции и *ассоциации*. Этот тип обучения, в основном, строится на наблюдениях и определении групп или кластеров точек данных со схожими характеристиками. Обучение без учителя широко используется интернет-магазинами, вроде Amazon, для деления клиентов на группы (кластеризация) по их покупательскому поведению, а затем для предложения товаров, которые могут их заинтересовать. Кроме того, интернет-магазины пытаются найти связь (ассоциация) между различными покупками. Например, насколько вероятно, что клиент, купивший товар А, купит и товар Б.

- **Обучение с подкреплением**: модель вознаграждается за принятие правильного решения в конкретной ситуации. В этом случае не используется никаких данных, модель учится на собственном опыте. Популярный пример — беспилотные автомобили.

7. **Тестирование**: для тестирования модели необходим набор данных, который не использовался для обучения. В традиционном подходе задействуются две трети имеющихся данных для обучения и одна треть — для тестирования.

Кроме описанных трех подходов существует также *глубокое обучение*. Это более сложный вид машинного обучения, основанный на процессе получения человеческим мозгом определенных знаний с помощью алгоритмов нейронной сети. В этой главе мы будем строить образцы моделей, используя обучение с учителем.

В следующем подразделе рассмотрим доступные в Python библиотеки для решения задач машинного обучения.

Использование Python для машинного обучения

Python пользуется популярностью в сообществе data science из-за его простоты, кросс-платформенной совместимости и богатой поддержки анализа и обработки данных с помощью библиотек. Одним из ключевых этапов в машинном обучении является подготовка данных для построения моделей, и Python в этом плане выигрывает. Единственный минус состоит в том, что это интерпретируемый язык, поэтому скорость выполнения кода медленнее по сравнению, например, с C. Но это не является серьезной проблемой, поскольку существуют библиотеки, позволяющие увеличить скорость языка за счет параллельного использования нескольких ядер центрального (ЦП) или графического (ГП) процессоров.

Далее рассмотрим несколько библиотек Python для ML.

Библиотеки машинного обучения в Python

Существует несколько ML-библиотек в Python. Некоторые из них уже упоминались, например, NumPy, SciPy и Pandas. Они широко используются для уточнения, анализа и обработки данных. В этой главе мы кратко рассмотрим самые популярные из них для создания ML-моделей машинного обучения.

scikit-learn

Эта библиотека пользуется популярностью, поскольку имеет множество встроенных алгоритмов машинного обучения и инструментов оценки их производительности. Она включает в себя алгоритмы классификации и регрессии для обучения с учителем, а также кластеризации и ассоциации для обучения без учителя. К тому же scikit-learn, в основном, написана на Python и использует библиотеку NumPy для многих операций. Новичкам рекомендуется начать именно, с нее, а затем переходить к более продвинутым, например, TensorFlow. Мы будем использовать scikit-learn в целях демонстрации самой концепции, а именно, создания, обучения и оценки ML-моделей.

Кроме того, scikit-learn предлагает алгоритмы *градиентного бустинга* (**Gradient Boost, GB**). Они основаны на математическом понятии *градиента*, который представляет собой *наклон* функции. В контексте машинного обучения он измеряет изменения в ошибках. Идея алгоритмов GB заключается в итеративной *тонкой настройке* параметров для нахождения *локального минимума функции* (минимизация ошибок для ML-моделей). Эти алгоритмы используют ту же стратегию для итеративного улучшения новой модели, учитывая производительность предыдущей. Это происходит с помощью тонкой настройки параметров для новой модели, а также задания в качестве цели принятие новой модели, если она лучше минимизирует ошибки, чем предыдущая.

XGBoost

XGBoost (**eXtreme Gradient Boosting**) — это библиотека алгоритмов, которая использует *деревья решений* с градиентным бустингом. Она популярна, поскольку чрезвычайно быстра и предлагает наилучшую производительность по сравнению с другими реализациями алгоритмов GB и традиционными алгоритмами ML. Как и XGBoost, scikit-learn также предлагает алгоритмы градиентного бустинга, которые принципиально не отличаются, но первая работает значительно быстрее. В основном, это достигается за счет максимального использования параллелизма между ядрами одной машины или в распределенном кластере узлов. Библиотека также может упорядочить деревья решений во избежание подгонки модели к данным. XGBoost не является полноценным фреймворком ML, она предлагает, в основном, алгоритмы (модели). Ее можно задействовать, но придется также использовать и scikit-learn для остальных служебных функций и инструментов, таких, как анализ и подготовка данных.

TensorFlow

TensorFlow — еще одна очень популярная ML-библиотека с открытым исходным кодом, разработанная командой Google Brain для высокопроизводительных вычислений. Она особенно полезна при обучении и работе с глубокими нейронными сетями, поэтому является популярным выбором в сфере глубокого обучения.

Keras

Это API с открытым исходным кодом для глубокого обучения нейронных сетей в Python. Keras — более высокоуровневый API поверх TensorFlow. Работать через него удобнее, чем использовать TensorFlow напрямую, поэтому он рекомендован на начальном этапе работы при разработке моделей ML с помощью Python. Keras может работать как с центральными, так и графическими процессорами.

PyTorch

PyTorch — библиотека с открытым исходным кодом. Представляет собой реализацию для Python популярной библиотеки Torch на C.

В следующем подразделе мы кратко обсудим рекомендации и практики по использованию Python в ML.

Рекомендации по обучающим данным

Ранее мы уже подчеркивали, как важны данные при обучении моделей. В этом подразделе изучим некоторые передовые методы и рекомендации по подготовке и использованию данных для обучения модели ML:

- Большой набор данных (несколько тысяч записей или хотя бы несколько сотен) имеет ключевое значение; чем больше данных, тем точнее модель.
- Перед началом любого обучения данные необходимо очистить и уточнить; в них не должно быть отсутствующих или неточных полей; для таких задач удобно использовать библиотеки Python, например, Pandas.
- Важно использовать набор данных без нарушения конфиденциальности и безопасности информации; необходимо убедиться, что не используются данные каких-либо других организаций без соответствующего разрешения.
- ГП отлично подходят для обработки больших объемов данных, их использование даст более быстрый результат; такие библиотеки, как XGBoost, TensorFlow и Keras, задействуют графические процессоры для обучения.
- При большом обучающем наборе важно эффективно использовать системную память; следует загружать данные порционно или использовать распределенные кластеры; рекомендуем использовать функцию генератора как можно чаще.
- Также при выполнении ресурсоемких задач рекомендуется следить за потреблением памяти и периодически освобождать ее, заставляя сборщик мусора удалять неиспользуемые объекты.

После рассмотрения библиотек и рекомендаций для машинного обучения перейдем к работе с реальными примерами.

Создание и оценка модели машинного обучения

Прежде чем приступить к написанию программы, исследуем процесс создания модели.

Процесс построения модели машинного обучения

Различные компоненты ML мы уже обсуждали в подразделе *«Введение в машинное обучение»*. Процесс обучения использует эти элементы как входные данные для тренировки модели. Он состоит из трех этапов, а каждый этап из нескольких шагов, как показано на рис. 13.2:

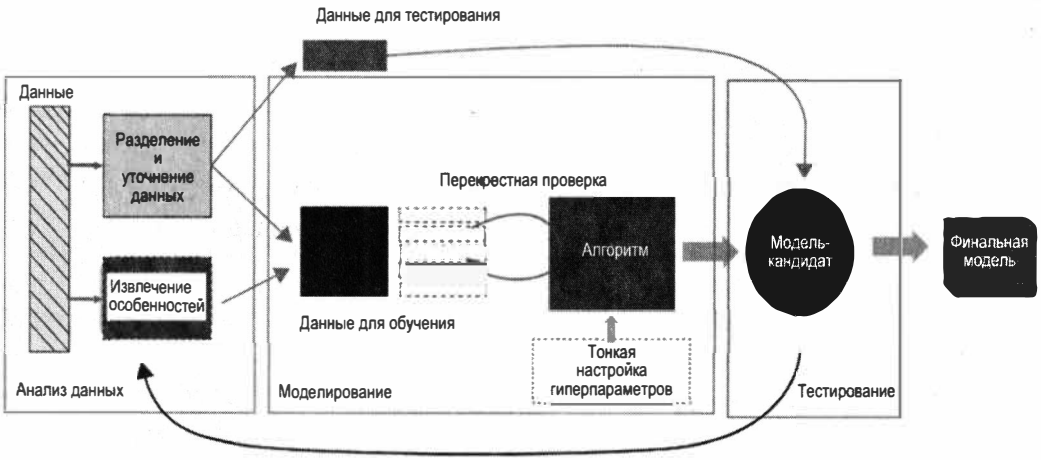


Рис. 13.2. Шаги по созданию модели машинного обучения при использовании классического подхода

Рассмотрим все этапы и шаги подробнее:

1. **Анализ данных**: на этом этапе происходит сбор необработанных данных и преобразование их в форму, которую можно анализировать и использовать в дальнейшем для моделирования и тестирования. Некоторые данные могут быть отброшены, например, записи с пустыми значениями. Анализ позволяет выбрать признаки (атрибуты), которые можно использовать для выявления закономерностей в данных. Извлечение признаков — важный шаг. От них зависит успешность построения модели. Во многих случаях приходится корректировать признаки после тестирования, с целью убедиться что набор характерных признаков верный. Традиционно данные разделяют на два набора, один из которых используется для обучения на этапе моделирования, а другой — для проверки точности обученной модели на этапе тестирования. Последний этап может быть пропущен, если оценка модели выполняется другим подходом, например, с помощью *кросс-валидации* (**Cross-validation**). Мы рекомендуем не пропускать этап тести-

рования и оставить часть данных (скрытых от модели), как показано на схеме выше.

2. **Создание модели**: этот этап посвящен обучению модели на основе данных и признаков, извлеченных на предыдущем шаге. В традиционном подходе обучающие данные используются как есть, но для повышения точности можно задействовать следующие дополнительные методы:
   - Можно разделить обучающие данные на срезы и использовать один для оценки модели, а остальные — для обучения; затем повторить процесс для другой комбинации обучающих и оценочных срезов; такой подход к оценке называется кросс-валидацией.
   - ML-алгоритмы имеют несколько параметров, также известных как *гиперпараметры* (**Hyperparameters**), которые можно использовать для тонкой настройки модели для наилучшего соответствия данным; обычно такая настройка выполняется вместе с кросс-валидацией на этапе моделирования.
   - Значения признаков в данных могут использовать разные шкалы измерения, что затрудняет создание правил; в таких ситуациях нужно привести данные (значения признаков) к общей или нормализованной шкале (например, от 0 до 1); этот этап называется *масштабированием данных* или *нормализацией*; все шаги по нормализации и оценке или только их часть могут быть добавлены в конвейер (например, Apache Beam) и выполняться вместе при оценке разных комбинаций для выбора лучшей модели; результатом этого этапа будет *модель-кандидат*.
3. **Тестирование**: на этом этапе используются данные из отложенного набора для проверки точности модели-кандидата, созданной на предыдущем этапе. Результаты тестирования можно использовать для добавления/удаления некоторых признаков и тонкой настройки модели, пока не будет достигнута приемлемая точность.

После достижения необходимой точности можно приступать к реализации модели для прогнозирования на основе реальных данных.

Создание примера машинного обучения

В этом разделе мы создадим пример модели машинного обучения на Python, который будет распознавать три типа растения *Ирис*. Для построения модели используем общедоступный набор данных с четырьмя признаками (длина и ширина чашелистиков и лепестков) и три типа цветков ириса.

Для упражнения задействуем следующие компоненты:

♦ Набор данных об ирисах из репозитория **UC Irvine Machine Learning Repository** (*http://archive.ics.uci.edu/ml/*), который содержит 150 записей и три ожидаемых шаблона для идентификации; это уточненный набор данных, в котором уже определены необходимые признаки.

- Различные библиотеки Python:
  - Pandas и matplotlib для анализа данных.
  - scikit-learn для обучения и тестирования модели ML.

Сначала напишем программу на Python для анализа набора данных.

Анализ набора данных

Для простоты мы загрузили два файла с набором данных (`iris.data` и `iris.names`) из репозитория *https://archive.ics.uci.edu/ml/machine-learning-databases/iris/*.

Можно напрямую получить доступ к файлу данных из этого репозитория через Python, но в примере мы будем использовать его локальную копию. Кроме того, scikit-learn предоставляет несколько наборов данных как часть библиотеки и может напрямую использоваться для оценки. Мы решили использовать фактические файлы, поскольку это близко к реальным сценариям, когда мы сами собираем данные и используем их.

Файл данных содержит 150 записей, отсортированных по ожидаемому результату. В нем предоставлены значения для четырех признаков, которые описаны в файле `iris.names` (`sepal-length`, `sepal-width`, `petal-length` и `petal-width`). Согласно файлу, выходными результатами являются три вида растения ирис: `Iris-setosa` (ирис щетинистый), `Iris-versicolor` (ирис разноцветный) и `Iris-virginica` (ирис виргинский). Мы загрузим данные в pandas DataFrame и проанализируем их на наличие интересующих нас атрибутов. Пример кода для анализа выглядит следующим образом:

```
#iris_data_analysis.py
from pandas import read_csv
from matplotlib import pyplot

data_file = "iris/iris.data"
iris_names = ['sepal-length', 'sepal-width', 'petal-length',
  'petal-width', 'class']
df = read_csv(data_file, names=iris_names)

print(df.shape)
print(df.head(20))
print(df.describe())
print(df.groupby('class').size())

# ящик с усами
df.plot(kind='box', subplots=True, layout=(3,3), sharex=False, sharey=False)
pyplot.show()

# проверка гистограмм
df.hist()
pyplot.show()
```

В первой части анализа мы проверили несколько метрик в данных с помощью функций библиотеки Pandas:

- Мы использовали метод `shape` для получения размера DataFrame, который должен составлять [150, 5], поскольку мы имеем 150 записей и пять столбцов (четыре с признаками и один с ожидаемым результатом); на этом шаге выполняется проверка, что все данные корректно загружены в DataFrame.
- Мы проверили фактические данные методами `head` или `tail`; это применимо только для визуального осмотра данных, особенно если не видно, что находится в файле.
- Метод `describe` предоставил различные статистические *ключевые показатели эффективности* (**Key Performance Indicators**, **KPI**, **КПЭ**) для данных:

	sepal-length	sepal-width	petal-length	petal-width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

- Эти KPI могут помочь выбрать подходящий алгоритм для набора данных.
- Метод `groupby` используется при определении числа записей для каждого `class` (столбец с ожидаемыми результатами); в выводе будет указано, что для каждого вида ирисов имеется 50 записей:

```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
```

Во второй части анализа мы попытались использовать *блочную диаграмму* (также известную, как *Ящик с усами*) и *гистограмму*. Ящик с усами — это визуальный способ отображения KPI, полученных методом `describe`: минимальное значение, первый квартиль, второй квартиль (медиана), третий квартиль и максимальное значение. Этот график показывает, распределены ли данные симметрично, сгруппированы в определенном диапазоне или смещены в одну сторону от распределения. Для нашего набора данных мы получим диаграммы, показанные на рис. 13.3.

На графиках видно, что данные о длине (**petal-length**) и ширине (**sepal-length**) чашелистиков сгруппированы между первым и третьим квартилями. В этом можно убедиться, проанализировав распределение данных с помощью гистограммы, представленной на рис. 13.4.

Проанализировав данные и выбрав подходящий алгоритм (модель), можно перейти к следующему шагу — обучению модели.

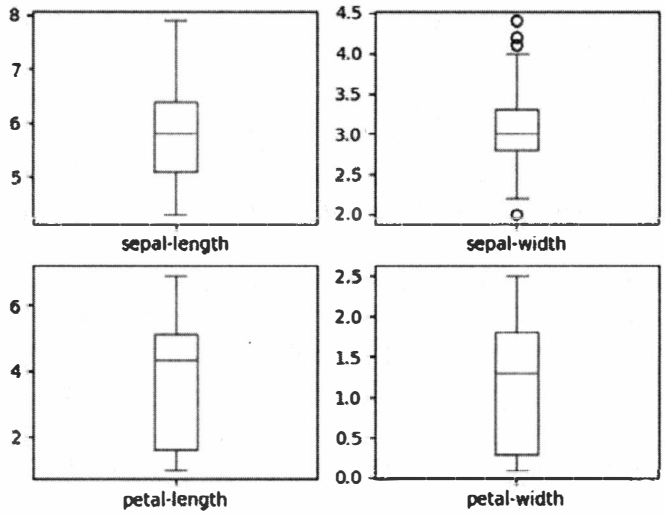


Рис. 13.3. Диаграммы «ящик с усами» для признаков из набора данных об ирисах

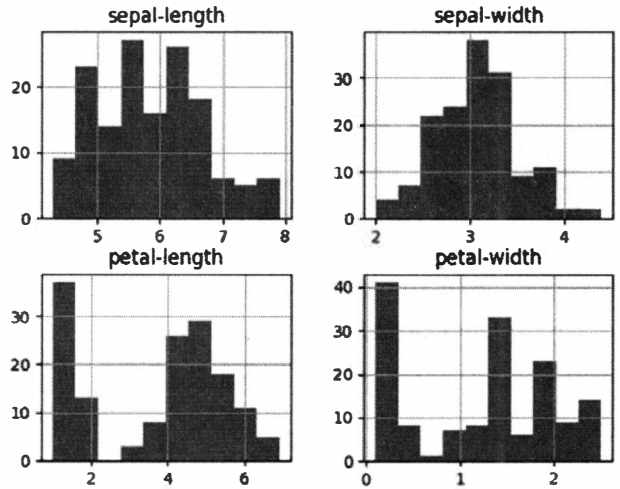


Рис. 13.4. Гистограмма с признаками из набора данных об ирисах

Обучение и тестирование модели машинного обучения

Для обучения и тестирования модели необходимо выполнить следующие шаги:

1. Сначала нужно разделить набор данных на две части: одна для обучения, другая — для тестирования. Такой подход называется методом *удержания* **(Holdout)**. Библиотека scikit-learn имеет функцию `train_test_split` для удобного разделения:

```
#iris_build_svm_model.py (#1)
# Разделяем набор данных
```

```
X = df.drop('class', axis = 1))
y = df['class']
X_train, X_test, y_train, y_test = train_test_split(X,
  y, test_size=0.20, random_state=1, shuffle=True)
```

2. Перед вызовом `train_test_split` набор данных разделяется на набор признаков (в номенклатуре ML обычно обозначается `X` и должен быть в верхнем регистре) и набор ожидаемых результатов (обозначается `y` и должен быть в нижнем регистре). Затем оба набора данных (`X` и `y`) разделяются функцией `train_test_split` в соответствии с размером тестового набора `test_size` (в нашем случае 20% от всего набора данных). Мы также разрешаем перемешивание данных перед разделением. В результате получим четыре набора (`X_train`, `y_train`, `X_test` и `y_test`) для обучения и тестирования.
3. Далее создадим модель и предоставим для ее обучения данные `X_train` и `y_train`. Выбор алгоритма не так важен в этом упражнении. Для набора данных будем использовать алгоритм под названием *машина опорных векторов* (**Support Vector Machine**, **SVC**), известный также как *метод опорных векторов*. Пример кода Python:

```
#iris_build_svm_model.py (#2)
# делаем предсказания
model = SVC(gamma='auto')
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

4. Для обучения модели мы использовали метод `fit`. В следующем выражении мы сделали прогнозы на основе данных тестирования (`X_test`). Они будут использоваться для оценки эффективности обученной модели.
5. В конце прогнозы будут сравниваться с ожидаемыми результатами (`y_test`) с использованием функций `accuracy_score` и `classification_report` из библиотеки scikit-learn:

```
#iris_build_svm_model.py (#3)
# оценка предсказаний
print(accuracy_score(y_test, predictions))
print(classification_report(y_test, predictions))
```

Консольный вывод будет следующим:

```
0.9666666
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        11
Iris-versicolor       1.00      0.92      0.96        13
 Iris-virginica       0.86      1.00      0.92         6

       accuracy                           0.97        30
      macro avg       0.95      0.97      0.96        30
   weighted avg       0.97      0.97      0.97        30
```

Значение точности очень высокое (0.966), что указывает на способность модели прогнозировать вид ириса на тестовых данных с точностью 96%. Модель превосходно справляется с определением видов Iris-setosa и Iris-versicolor, но менее эффективно с видом Iris-virginica (точность 86%). Есть несколько способов повысить эффективность модели, и мы обсудим их далее.

Оценка модели с помощью кросс-валидации и тонкой настройки гиперпараметров

В предыдущем примере мы акцентировали внимание именно на простом процессе обучения для лучшего понимания основных этапов создания ML-модели. Для производственных развертываний недостаточно полагаться на столь маленький набор данных, содержащий всего 150 записей. Кроме того, необходимо оценить модель на эффективность прогнозов, используя следующие методы:

1. **k-блочная кросс-валидация:** в предыдущей модели мы перемешивали данные перед разделением их на обучающий и тестовый наборы методом удержания. Из-за этого модель может выдавать разные результаты при каждом обучении, что приводит к нестабильности. Довольно сложно выбрать обучающие данные из небольшого набора так, что они будут похожи на реальные данные. Если мы хотим сделать модель более стабильной на небольшом наборе, рекомендуется использовать кросс-валидацию по k блокам. При таком подходе набор разделяется на k блоков или срезов, где k–1 блок используются для обучения, а последний k-й — для оценки или тестирования. Процесс повторяется, пока в тестировании не поучаствует каждый блок. Это эквивалентно повторению метода удержания k раз с использованием разных срезов данных для тестирования.
2. В дальнейшем мы разделим наш набор на пять фрагментов (k=5 для 5-кратной кросс-валидации). В первой итерации можно использовать первый срез (20%) для тестирования, а остальные четыре (80%) — для обучения. Во второй итерации будет использован второй срез для тестирования, а остальные четыре — для обучения, и т. д. Так мы сможем оценить модель по всем пяти обучающим наборам и выбрать лучший вариант. Схема выбора данных для обучения и тестирования показана на рис. 13.5:

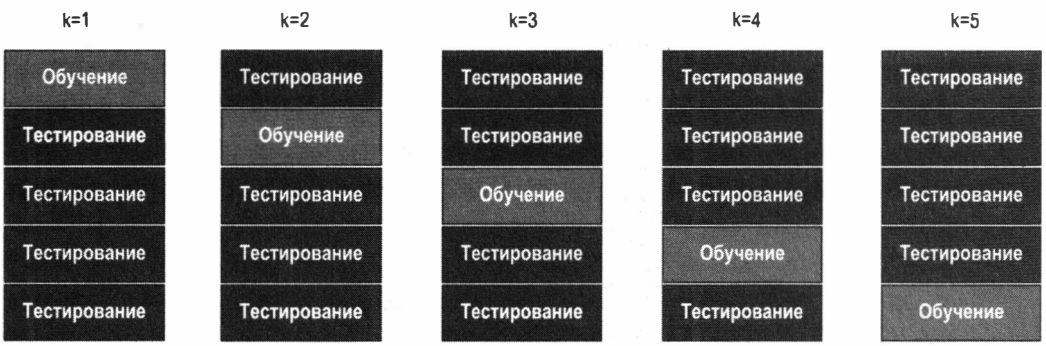


Рис. 13.5. Схема кросс-валидации для 5 блоков данных

3. Точность кросс-валидации рассчитывается как средняя точность каждой модели, созданной на каждой итерации в соответствии со значением k.
4. **Оптимизация гиперпараметров:** в предыдущем примере был использован алгоритм машинного обучения с параметрами по умолчанию. Каждый алгоритм имеет множество гиперпараметров, которые можно изменять для настройки модели в соответствии с набором данных. Некоторые параметры можно задавать вручную, анализируя распределение данных, но выявлять влияние комбинаций этих параметров утомительно. Необходимо оценить модель, используя различные значения гиперпараметров, что может помочь в выборе наилучшей их комбинации. Этот метод называется *тонкой настройкой* или *оптимизацией* гиперпараметров.

Кросс-валидация и тонкая настройка сложны в реализации даже с помощью программного кода. К счастью, scikit-learn содержит инструменты для решения таких задач парой строк кода на Python. Библиотека предлагает два типа инструментов: `GridSearchCV` и `RandomizedSearchCV`. Рассмотрим каждый из них.

GridSearchCV

Инструмент `GridSearchCV` оценивает любую заданную модель, используя кросс-валидацию для всех возможных комбинаций значений, предоставленных для гиперпараметров. Каждая комбинация значений будет оцениваться с помощью кросс-валидации на срезах набора.

В следующем примере используем класс `GridSearchCV` из библиотеки scikit-learn для оценки SVC-модели для комбинации параметров `C` и `gamma`. `C` — это параметр регуляризации, который помогает найти компромисс между низкой ошибкой обучения и низкой ошибкой тестирования. Чем выше значение `C`, тем большее количество ошибок допускается. Мы будем использовать значения 0.001, 0.01, 1, 5, 10 и 100 для `C`. Параметр `gamma` определяет нелинейные *гиперплоскости* или нелинейные *линии* для классификации. Чем выше `gamma`, тем больше данных модель может попытаться уместить. Для `gamma` мы также укажем значения 0,001, 0,01, 1, 5, 10 и 100. Полный код `GridSearchCV` выглядит следующим образом:

```
#iris_eval_svc_model.py (часть 1 из 2)
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV
from sklearn.datasets import load_iris
from sklearn.svm import SVC
iris= load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test=train_test_split(X,y,test_size=0.2)
params = {"C": [0.001, 0.01, 1, 5, 10, 100],
          "gamma": [0.001, 0.01, 0.1, 1, 10, 100]}
model=SVC()
grid_cv=GridSearchCV(model, params,  cv=5)
```

```
grid_cv.fit(X_train,y_train)
print(f"GridSearch- best parameter:{grid_cv.best_params_}")
print(f"GridSearch- accuracy: {grid_cv.best_score_}")
print(classification_report(y_test,
  grid_cv.best_estimator_.predict( X_test)))
```

Ключевые моменты кода:

- Мы загрузили данные напрямую из библиотеки scikit-learn в целях демонстрации; этот код также можно использоваться для загрузки данных из локального файла.
- На первом шаге важно определить словарь `params` для тонкой настройки; в этом словаре мы задаем значения параметров `C` и `gamma`.
- Мы указали `cv=5`, это позволит оценить каждую комбинацию параметров с помощью кросс-валидации на пяти срезах.

Результат программы даст наилучшее сочетание `C` и `gamma`, а также точность модели с помощью кросс-валидации. Консольный вывод для лучшей комбинации параметров и лучшей точности модели будет следующий:

```
GridSearch- best parameter: {'C': 5, 'gamma': 0.1}
GridSearch- accuracy: 0.9833333333333334
```

За счет оценки разных сочетаний параметров и использования кросс-валидации с помощью `GridSearchCV` общая точность модели повышается с 96% до 98%. Отчет о классификации (в выходных данных отсутствует) показывает, что точность для трех видов ирисов составляет 100% для наших тестовых данных. Однако этот инструмент невозможно использовать, когда есть большое число значений параметров и большой набор данных.

RandomizedSearchCV

Инструмент `RandomizedSearchCV` позволяет оценивать модель только для случайно выбранных значений гиперпараметров, а не для всех комбинаций. В качестве входных данных можно указать значения параметров и количество случайных итераций для выполнения. `RandomizedSearchCV` случайным образом выберет комбинацию параметров в соответствии с количеством предоставленных итераций. Этот инструмент полезен, когда мы имеем дело с большим набором данных, а также при большом количестве комбинаций, оценка которых может потребовать очень много времени и вычислительных ресурсов.

Код Python для `RandomizedSearchCV` будет аналогичным, как и для `GridSearchCV`, за исключением нескольких дополнительных строк:

```
#iris_eval_svc_model.py (часть 2 из 2)
rand_cv=RandomizedSearchCV(model, params, n_iter = 5, cv=5)
rand_cv.fit(x_train,y_train)
print(f" RandomizedSearch - best parameter: {rand_cv.best_params_}")
print(f" RandomizedSearch - accuracy: {rand_cv.best_score_}")
```

Поскольку мы указали `n_iter=5`, `RandomizedSearchCV` выберет только пять комбинаций параметров `C` и `gamma` и оценит по ним модель.

После выполнения этого кода консольный вывод будет следующим:

```
RandomizedSearch- best parameter: {'gamma': 10, 'C': 5}
RandomizedSearch- accuracy: 0.9333333333333333
```

Обратите внимание, результат может отличаться, поскольку этот инструмент может выбрать другие комбинации значений для оценки. Если увеличить количество итераций (`n_iter`) для объекта `RandomizedSearchCV`, точность в выводе повысится. Если `n_iter` не задать, оценка будет выполнена для всех комбинаций. Это означает, что результат будет аналогичен `GridSearchCV`.

Как видите, лучшие выбранные комбинации от `GridSearchCV` и `RandomizedSearchCV` отличаются. Такой результат ожидаем, поскольку мы запускали инструменты с разным числом итераций.

На этом обсуждение, как создавать пример модели машинного обучения с помощью библиотеки scikit-learn, можно завершить. Мы рассмотрели основные этапы и концепции, необходимые для построения и оценки моделей. На практике для нормализации данных мы также выполнили их масштабирование, которое может быть достигнуто с помощью встроенных классов scikit-learn (например, `StandardScaler`) или с помощью собственных классов. Масштабирование — это операция преобразования данных, которую можно объединять с задачей обучения в одном конвейере. Библиотека scikit-learn поддерживает объединение в конвейер нескольких операций или задач с помощью класса `Pipeline`. Он также может быть использован напрямую с `RandomizedSearchCV` или `GridSearchCV`. Узнать больше о работе *масштабаторов* и конвейеров с библиотекой scikit-learn можно в онлайн-документации (*https://scikit-learn.org/stable/user_guide.html*).

Далее обсудим, как сохранить модель в файл и восстановить ее из файла.

Сохранение ML-модели в файл

Оценив модель и выбрав лучшую в соответствии с набором данных, на следующем шаге можно реализовать ее для будущих прогнозов. Модель может быть реализована как часть любого приложения Python, включая веб-приложения Flask или Django, микросервисы и даже облачные функции. Главная задача заключается в переносе модели из одной программы в другую. Для этого можно использовать такие библиотеки, как pickle или `joblib`, которые можно задействовать для сериализации модели в файл. Затем этот файл можно использовать в любом приложении для повторной загрузки модели в Python и создания прогнозов методом `predict`.

Для демонстрации этой концепции сохраним одну из ранее созданных ML-моделей (объект `model` в программе `iris_build_svm_model.py`) в файл `model.pkl`. Затем загрузим модель из этого файла с помощью pickle и сделаем прогноз на основе новых дан-

ных для эмуляции использования модели в любом приложении. Полный пример кода выглядит следующим образом:

```
#iris_save_load_predict_model.py
# создаем модель, используя код из iris_build_svm_model.py
# сохраняем модель в файл
with open("model.pkl", 'wb') as file:
    pickle.dump(model, file)

# загружаем модель из файла(в другом приложении)
with open("model.pkl", 'rb') as file:
    loaded_model = pickle.load(file)
    x_new = [[5.6, 2.6, 3.9, 1.2]]
    y_new = loaded_model.predict(x_new)
    print("X=%s, Predicted=%s" % (x_new[0], y_new[0]))
```

Использовать библиотеку joblib проще, чем pickle, но может потребоваться ее установка в качестве зависимости scikit-learn, если это еще не было сделано. В следующем примере показано, как использовать joblib для сохранения лучшей модели (в соответствии с оценкой инструмента `GridSearchCV`), а затем для загрузки модели из файла:

```
#iris_save_load_predict_gridmodel.py
# создаем и обучаем grid_cv, используя код из  iris_eval_svm_model.py

joblib.dump(grid_cv.best_estimator_, "model.joblib")
loaded_model = joblib.load("model.joblib")
x_new = [[5.6, 2.5, 3.9, 1.1]]
y_new = loaded_model.predict(x_new)
print("X=%s, Predicted=%s" % (x_new[0], y_new[0]))
```

Код библиотеки joblib лаконичен и прост. Прогнозирующая часть аналогична предыдущему примеру с библиотекой pickle.

Теперь, когда мы знаем, как сохранить модель в файл, можно перенести ее в любое приложение в локальной и даже облачной среде (например, GCP AI Platform). Далее обсудим, как развернуть нашу модель на платформе GCP.

Развертывание и прогнозирование ML-модели в GCP Cloud

Провайдеры публичных облаков предлагают различные ИИ-платформы для обучения как встроенных, так и пользовательских моделей для развертывания и дальнейшего прогнозирования. Google предлагает Vertex AI, Amazon предлагает Amazon SageMaker, а Azure — Azure ML. Мы остановили свой выбор на Google,

поскольку, предположительно, вы уже имеете аккаунт GCP и знакомы с основными принципами работы с ним. AI Platform является частью Vertex AI Platform и предназначена для обучения и развертывания ML-моделей в масштабе. GCP AI Platform поддерживает такие библиотеки, как scikit-learn, TensorFlow и XGBoost. В этом подразделе мы рассмотрим, как развернуть уже обученную модель в GCP, а затем сделать прогноз результата с ее помощью.

Google AI Platform предлагает сервер прогнозирования (вычислительный узел) через глобальную (`ml.googleapis.com`) или региональную (`<регион>-ml.googleapis.com`) конечные точки. Глобальная API-точка рекомендована для пакетных прогнозов, доступных для TensorFlow на Google AI Platform. Региональные обеспечивают дополнительную защиту от сбоев в других регионах. Мы развернем ML-модель, используя региональную конечную точку.

Нам понадобится проект GCP. Можно создать новый или использовать существующий из предыдущих упражнений. Действия по созданию проекта GCP и привязке биллинг-аккаунта приведены в главе 9 (*«Программирование на Python для облака»*). Когда проект готов, можно развернуть модель `model.joblib` из предыдущего подраздела:

1. Сначала создадим сегмент хранилища, где будет находиться файл модели. Можно использовать следующую команду Cloud SDK:

```
gsutil mb gs://<bucket name>
gsutil mb gs://muasif-svc-model #Образец сегмента создан
```

2. Когда сегмент готов, можно загрузить в него файл модели (`model.joblib`) следующей командой:

```
gsutil cp model.joblib gs://muasif-svc-model
```

3. Обратите внимание, имя файла модели должно иметь вид `model.*`. Это означает, что файл должен иметь имя `model` с расширениями `pkl`, `joblib` или `bst` в зависимости от библиотеки, используемой для упаковки модели.

4. Теперь можно инициировать рабочий процесс для создания модели на AI Platform, выполнив следующую команду. Обратите внимание, имя модели может содержать только буквы, цифры и нижнее подчеркивание:

```
gcloud ai-platform models create my_iris_model - region=us-central1
```

5. Теперь можно создать версию нашей модели следующей командой:

```
gcloud ai-platform versions create v1 \
  --model=my_iris_model\
  --origin=gs://muasif-svc-model \
  --framework=scikit-learn \
  --runtime-version=2.4 \
  --python-version=3.7 \
  --region=us-central1 \
  --machine-type=n1-standard-2
```

6. Рассмотрим атрибуты этой команды:

   - `model`: указывает на имя модели, созданное на предыдущем шаге;

- `origin`: указывает на сегмент хранилища, где находится файл; мы укажем только расположение каталога, а не путь к файлу;
- `framework`: указывает на используемую библиотеку ML; GCP поддерживает scikit-learn, TensorFlow и XGBoost;
- `runtime-version`: указывает на среду выполнения (в нашем случае на версию библиотеки scikit-learn);
- `python-version`: указывает на версию Python (в нашем случае выбрана максимальная версия, которую поддерживает GCP AI Platform на момент написания книги — 3.7);
- `region`: указывает регион, выбранный для модели;
- `machine-type` (опциональный): указывает на тип вычислительного узла, который будет использоваться для развертывания модели; если атрибут не указан, используется тип `nl-standard-2`;
- команде `versions create` может потребоваться несколько минут для развертывания новой версии. Как только она выполнится, мы получим примерно следующий результат:

```
Using endpoint [https://us-central1-ml.googleapis.com/]
Creating version (this might take a few minutes)......done.
```

7. Убедиться, что модель и версия корректно развернуты, можно командой `describe` в контексте `versions`:

```
gcloud ai-platform versions describe v1 ^
  model=my_iris_model
```

8. Когда модель вместе с версией развернуты, можно использовать новые данные для прогнозирования. В целях тестирования были добавлены несколько записей данных, отличных от исходного набора (в файл `input.json`):

```
[5.6, 2.5, 3.9, 1.1]
[3.2, 1.4, 3.0, 1.8]
```

9. Для прогнозирования результата на основе записей в файле `input.json` можно использовать следующую команду:

```
gcloud ai-platform predict --model my_iris_model
--version v1 --json-instances input.json
```

10. Вывод консоли покажет спрогнозированный класс для каждой записи, а также следующие данные:

```
Using endpoint [https://us-central1-ml.googleapis.com/]
['Iris-versicolor', 'Iris-virginica']
```

Для использования развернутой модели в нашем приложении (локальном или облачном) можно задействовать Cloud SDK или Cloud Shell, но рекомендуется выбрать Google AI API для предсказаний.

Мы рассмотрели развертывание в облаке и варианты предсказаний с помощью Google AI Platform. Однако модель можно перенести и на другие платформы, например, Amazon SageMaker или Azure ML. Дополнительную информацию по Amazon SageMaker можно найти по адресу: *https://docs.aws.amazon.com/sagemaker/*, а документацию по Azure ML по адресу: *https://docs. microsoft.com/en-us/azure/machine-learning/*.

Заключение

В этой главе мы познакомились с машинным обучением и его основными компонентами, включая наборы данных, алгоритмы и модели, а также обучение и тестирование моделей. После мы обсудили популярные фреймворки и библиотеки ML для Python, включая scikit-learn, TensorFlow, PyTorch и BGBoost. Мы также изучили рекомендации по очистке и разделению данных для обучения моделей. Для изучения библиотеки scikit-learn на практике мы создали пример модели с использованием алгоритма SVC. Далее обучили модель и оценили ее с помощью методов k-блочной кросс-валидации и тонкой настройки гиперпараметров. Научились сохранять модель в файл, а затем загружать ее обратно в программу. В конце мы продемонстрировали, как развернуть модель и использовать ее для прогнозов с помощью Google AI Platform и нескольких команд GCP Cloud SDK.

Эта глава будет полезна всем, кто хочет использовать Python для машинного обучения.

В следующей главе мы изучим, как использовать Python для автоматизации сетей.

Вопросы

1. Чем обучение с учителем отличается от обучения без учителя?
2. Что такое k-блочная кросс-валидация и как она используется для оценки модели?
3. Что такое `RandomizedSearchCV` и чем он отличается от `GridSearchCV`?
4. Какие библиотеки позволяют сохранить модель в файл?
5. Почему для Google AI Platform региональные конечные точки предпочтительнее глобальных?

Дополнительные ресурсы

- *«Machine Learning Algorithms»*, автор: Джузеппе Бонаккорсо (Giuseppe Bonaccorso).
- *«40 Algorithms Every Programmer Should Know»*, автор: Имран Ахмад (Imran Ahmad).

- *«Mastering Machine Learning with scikit-learn»*, автор: Гевин Хэкелинг (Gavin Hackeling).
- *«Python и машинное обучение: Машинное и глубокое обучение с использованием Python, scikit-learn и TensorFlow 2»*, авторы: Себастьян Рашка (Sebastian Raschka) и Вахид Мирджалили (Vahid Mirjalili).
- Руководство по scikit-learn: *https://scikit-learn.org/stable/ user_guide.html*.
- Руководство по Google AI Platform для обучения и развертывания моделей машинного обучения: *https://cloud.google.com/ai-platform/docs*.

Ответы

1. При обучении с учителем мы предоставляем желаемый результат с обучающими данными. При обучении без учителя мы предоставляем только обучающие данные без результата.
2. Кросс-валидация — это статистический метод, используемый для измерения эффективности модели машинного обучения. При k-блочной кросс-валидации данные делятся на *k* срезов или блоков. Для обучения используется k–1 срезов, а для проверки точности — k-й блок. Процесс повторяется, пока каждый срез не поучаствует в проверке. Точность кросс-валидации модели рассчитывается как средняя точность всех моделей, созданных за k итераций.
3. `RandomizedSearchCV` — инструмент библиотеки scikit-learn для применения кросс-валидации к модели ML со случайно выбранными гиперпараметрами. `GridSearchCV` предлагает аналогичные возможности, за исключением того что он проверяет модель для всех комбинаций значений гиперпараметров.
4. Pickle и Joblib.
5. Региональные конечные точки обеспечивают дополнительную защиту от сбоев в других регионах, а доступность вычислительных ресурсов для них выше, чем для глобальных.

14

Python для автоматизации сети

Традиционно за создание и обслуживание сетей всегда отвечали соответствующие специалисты, и для сферы телекоммуникаций этот подход также актуален. Однако управление и эксплуатация сетей вручную были медленными и иногда приводили к дорогостоящим сбоям из-за человеческих ошибок. Кроме того, для получения онлайн-услуги (например, подключение смены тарифа Интернета) клиенту приходилось ждать по несколько дней. С появлением смартфонов и мобильных приложений, где достаточно нажать на кнопку для получения услуги, пользователи привыкли, что сетевые сервисы будут предоставлять все необходимое в течение нескольких минут или даже секунд. Традиционный подход к управлению сетями этого не позволяет. Более того, он является препятствием для внедрения новых продуктов и услуг поставщиками.

Автоматизация сети (**Network Automation**) может улучшить эту ситуацию с помощью ПО для автоматизированного управления и рабочих аспектов сети. Такой подход исключает человеческий фактор при настройке сетевых устройств и позволяет значительно сократить эксплуатационные расходы благодаря автоматизации повторяющихся задач. Эта концепция помогает быстрее предоставлять услуги и позволяет поставщикам внедрять новые предложения.

Python хорошо подходит для подобных задач. В этой главе мы познакомимся с возможностями языка для автоматизации сети. Python предоставляет такие библиотеки, как Paramiko, Netmiko и NAPALM, которые можно использовать для взаимодействия с сетевыми устройствами. Если они конфигурируются *системой управления сетями* (**Network Management System, NMS**) или сетевым *контроллером/оркестратором*, Python может взаимодействовать с ними по протоколам **REST** и **RESTCONF**. Сквозная (**End-to-end**) автоматизация сети невозможна без прослушивания событий, происходящих в ней в режиме реального времени. Обычно эти события (или потоковые данные) доступны через такие системы, как **Apache Kafka**. Мы также рассмотрим работу с подобной *событийно-ориентированной системой* (**Event-driven system**).

Темы этой главы:

- Введение в автоматизацию сети.
- Взаимодействие с сетевыми устройствами.
- Интеграция с системами управления сетью (NMS).
- Работа с событийно-ориентированными системами.

В этой главе вы научитесь использовать библиотеки Python для получения информации с сетевого устройства и отправки конфигурационных данных на них. Это базовые действия для любого процесса по автоматизации сети.

Технические требования

В этой главе понадобится:

- Python 3.7 или более поздней версии.
- Установленные библиотеки Paramiko, Netmiko, NAPALM, ncclient и requests поверх Python.
- Доступ к одному или нескольким сетевым устройствам по протоколу SSH.
- Доступ к лаборатории для разработчиков Nokia для работы с их системой управления сетями, известной как **Network Services Platform (NSP)**.

Пример кода для этой главы можно найти по адресу:

https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter14.

> ***Важное примечание***
>
> В этой главе потребуется доступ к физическим или виртуальным сетевым устройствам и NMS. У вас такой возможности может не быть. Но вы можете использовать любое сетевое устройство с аналогичным функционалом. Мы сосредоточимся больше на реализации кода Python и сделаем удобным его повторное использование с другими устройствами или NMS.

Начнем обсуждение с введения в базовые принципы автоматизации сети.

Введение в автоматизацию сети

Автоматизация сети — это использование технологий и ПО для автоматизации процессов по управлению сетями и их эксплуатации. Ключевым словом здесь является *автоматизация процесса*. Это означает, что речь идет не только о развертывании и конфигурировании сети, но и о шагах для достижения этого. Например, иногда этапы автоматизации требуют одобрения от различных заинтересованных сторон перед отправкой конфигурации в сеть. Автоматизация такого этапа является частью автоматизации сети. Таким образом, процесс может меняться от одной организации к другой в зависимости от их внутреннего распорядка. Это затрудняет

создание единой платформы, способной выполнять автоматизацию сразу для многих клиентов.

В настоящее время поставщики (вендоры) сетевых устройств предпринимают значительные усилия по предоставлению необходимых платформ, которые могут помочь клиентам при построении индивидуальной автоматизации с минимальными усилиями. Например, Cisco предлагает Network Services Orchestrator (NSO), Juniper Networks предлагает платформу Paragon Automation, а Nokia — NSP.

Одна из проблем таких платформ заключается в привязке к поставщику. Это означает, что платформа конкретного вендора может также управлять устройствами других производителей, но на практике такое взаимодействие будет утомительным и дорогим. Поэтому операторы телекоммуникационных услуг ищут и другие решения. Python и Ansible — два популярных языка программирования, которые используются для автоматизации в отрасли телекоммуникаций. Прежде чем перейти к возможностям Python для таких задач, рассмотрим преимущества и недостатки сетевой автоматизации.

Плюсы и минусы автоматизации сети

В самом начале мы уже затронули несколько преимуществ. Все основные достоинства можно резюмировать следующим образом:

- **Быстрое предоставление услуг**: чем быстрее оператор предоставит услугу новому клиенту, тем быстрее начнет получать доход, а клиент останется доволен скоростью обслуживания.
- **Снижение эксплуатационных расходов**: за счет автоматизации повторяющихся задач и мониторинга сети инструментами с обратной связью, можно сократить затраты.
- **Исключение человеческого фактора**: большинство сетевых сбоев происходит из-за человеческих ошибок; их можно избежать, используя при настройке сети стандартные шаблоны, которые тщательно тестируются перед реализацией.
- **Единообразие при настройке сети**: человек не способен учесть все шаблоны и соглашения об именовании, которые важны при управлении сетью; автоматизация обеспечивает согласованность при настройке, поскольку используется один и тот же скрипт или шаблон.
- **Мониторинг сети**: инструменты и платформы позволяют получить доступ к возможностям мониторинга сети и визуализировать в ней все процессы; это позволяет реагировать на пиковые нагрузки еще до того, как из-за нехватки ресурсов возникнут проблемы.

Цифровая трансформация невозможна без автоматизации, но она имеет и свои недостатки:

- **Стоимость**: создание или настройка ПО для автоматизации всегда требует затрат, которые должны быть учтены заранее.

- **Сопротивление сотрудников**: во многих организациях специалисты считают автоматизацию угрозой их рабочему месту, поэтому сопротивляются нововведениям, особенно в эксплуатационных бригадах.
- **Структура организации**: автоматизация обеспечивает реальную *окупаемость инвестиций* (**Return On Investment**, **ROI**), когда она используется на разных сетевых уровнях и доменах; проблема заключается в принадлежности этих доменов разным отделам, каждый из которых имеет собственные стратегии автоматизации и предпочтения в платформах.
- **Выбор инструмента/платформы автоматизации**: выбор платформы от поставщиков является непростым решением, поскольку вариантов очень много (Cisco, Nokia, HP, Accenture и многие др.); часто поставщики сетевых услуг используют решения разных вендоров, и это создает новый набор проблем, связанных с их взаимодействием.
- **Обслуживание**: обслуживание инструментов и скриптов автоматизации так же важно, как их создание; это требует заключения контрактов на техническое обслуживание у поставщиков или создания своей команды для этих целей.

Далее рассмотрим, какие задачи можно автоматизировать в реальности.

Варианты использования

Python и другие инструменты помогают автоматизировать некоторые однообразные действия по управлению сетью. Но настоящие их преимущества проявляются в задачах, которые повторяются, подвержены ошибкам или являются трудоемкими при выполнении вручную. Лучшие варианты применения автоматизации сети (с точки зрения поставщика телекоммуникационных услуг) следующие:

- Повседневная конфигурация сетевых устройств, например, создание новых IP-интерфейсов и услуг по подключению к сети; выполнение этих задач вручную занимает много времени.
- Настройка правил и политик брандмауэра для экономии времени; эти задачи являются утомительными, а любые ошибки могут привести к потере времени на их поиск и устранение.
- Когда в сети тысячи устройств, обновление ПО становится масштабной задачей, которая может затянуться на 1–2 года; автоматизации позволяет ускорить эти процессы и обеспечить удобную проверку устройств до и после обновления.
- Подключение новых устройств в сеть; если устройство должно быть установлено на территории клиента, можно сэкономить время, автоматизировав процесс подключения; этот процесс также известен как **Zero Touch Provisioning** (**ZTP**).

После знакомства с основами автоматизации сети можно перейти к взаимодействию с сетевыми устройствами с помощью разных протоколов.

Взаимодействие с сетевыми устройствами

Python является популярным выбором при автоматизации сети, поскольку он прост в изучении и может использоваться для интеграции с сетевыми устройствами как напрямую, так и через NMS. На самом деле многие вендоры, вроде Nokia и Cisco, обеспечивают поддержку Python, поскольку возможность выполнения кода на самом устройстве полезна для автоматизации задач. В этой части книги мы сосредоточимся на выполнении кода вне устройства, что даст возможность работать с несколькими устройствами одновременно.

> ***Важное примечание***
>
> Для всех примеров кода в этой главе мы будем использовать виртуальное сетевое устройство от Cisco (IOS XR версии 7.1.2). Для интеграции с NMS будем использовать систему Nokia NSP.

Прежде чем приступить к взаимодействию с сетевыми устройствами, рассмотрим сначала, какие протоколы связи нам доступны.

Протоколы для взаимодействия с сетевыми устройствами

Для прямого взаимодействия с сетевыми устройствами можно использовать несколько протоколов, например, **SSH (Secure Shell Protocol)**, **SNMP (Simple Network Management Protocol)** и **NETCONF (Network Configuration)**. Некоторые из них работают поверх друг друга. Остановимся на самых популярных из них.

SSH

SSH — это сетевой протокол для защищенного обмена данными между любыми двумя устройствами или компьютерами. Весь трафик между объектами будет зашифрован перед отправкой по транспортному каналу. Обычно SSH-клиент задействуется для подключения к сетевому устройству с помощью команды `ssh`. При этом клиент использует в команде имя пользователя, вошедшего в систему:

```
ssh <server ip or hostname>
```

Можно использовать другого пользователя, отличного от вошедшего в систему, указав имя следующим образом:

```
ssh username@<server IP or hostname>
```

После установки соединения можно отправлять команды через командную строку (CLI) либо для получения конфигурации или оперативной информации, либо для настройки устройства. SSH версии 2 (**SSHv2**) — это популярный вариант при взаимодействии с устройствами для управления и даже автоматизации.

Использование протокола SSH с Python-библиотеками Paramiko, Netmiko и NAPALM мы обсудим в подразделе *«Взаимодействие с сетевыми устройствами с помощью протоколов на основе SSH»*. Он также является основным транспортным

протоколом для многих других (расширенных) протоколов управления сетью, например, NETCONF.

SNMP

Этот протокол был стандартом больше 30 лет и по-прежнему широко используется для управления сетью. Однако сейчас его заменяют более продвинутые и масштабируемые протоколы вроде NETCONF и gNMI. SNMP можно использовать как для конфигурации, так и для мониторинга сети, где обычно он и применяется. Сейчас он считается устаревшим протоколом, который появился в конце 1980-х годов исключительно для управления сетью.

SNMP использует *базу управляющей информации* **(Management Information Base, MIB)**, которая является моделью устройства, которая была написана с использованием языка **SMI (Structure of Management Information)**.

NETCONF

Протокол NETCONF, представленный *Инженерным советом Интернета* (**Internet Engineering Task Force, IETF**), пришел на смену SNMP. Он, в основном, используется для настройки и должен поддерживаться всеми новыми сетевыми устройствами. NETCONF состоит из четырех уровней:

- **Содержимое**: это уровень данных, основанный на языке моделирования YANG; каждое устройство предлагает несколько YANG-моделей для разных модулей; дополнительную информацию можно найти по ссылке *https://github.com/YangModels/yang*.
- **Операции**: действия или инструкции, которые отправляются клиентом на сервер (также имеет название **NETCONF Agent**); эти операции заключены в сообщения *запросов* и *ответов*; примеры операций NETCONF: `get`, `get-config`, `edit-config` и `delete-config`.
- **Сообщения**: это сообщения RPC, которыми обмениваются клиенты и NETCONF Agent; операции NETCONF и данные, закодированные как XML, упаковываются в сообщения RPC.
- **Транспорт**: этот уровень обеспечивает канал связи между клиентом и сервером; сообщения могут использовать NETCONF поверх SSH или TLS с поддержкой сертификатов SSL.

Протокол NETCONF основан на обмене XML-сообщениями по протоколу SSH через порт по умолчанию `830`. Базы конфигураций для сетевых устройств обычно делятся на два типа. Первый тип называется *активной* БД (или хранилищем) с текущей конфигурацией, включая рабочие данные. Это обязательная база для каждого устройства. Второй тип называется *база-кандидат* (или хранилище-кандидат), которая хранит *возможную* конфигурацию до отправки в *активную*. Когда существует кандидат, изменения конфигурации нельзя напрямую вносить в активную БД.

Как работать с NETCONF, используя Python, мы рассмотрим в подразделе *«Взаимодействие с сетевыми устройствами с помощью NETCONF»*.

RESTCONF

RESTCONF — это тоже стандарт IETF, который предлагает подмножество функций NETCONF через интерфейс RESTful. Вместо RPC-вызовов с кодировкой XML он предлагает REST-вызовы на основе HTTP/HTTPS с возможностью использования сообщений XML или JSON. Если сетевое устройство поддерживает RESTCONF, можно использовать HTTP-методы (`GET`, `PATCH`, `PUT`, `POST` и `DELETE`) для управления сетью. Нужно понимать, что RESTCONF предоставляет ограниченный функционал NETCONF через HTTP/HTTPS. RESTCONF не поддерживает такие операции NETCONF, как *коммиты* (**Commit**), *откаты* (**Rollback**) или *блокировка конфигурации*.

gRPC/gNMI

gNMI — это *интерфейс сетевого взаимодействия* (**Network Management Interface**, **NMI**) от Google (g в начале), основанный на gRPC. gRPC — это удаленный вызов процедур от Google для извлечения данных с низкой задержкой и высокой масштабируемостью. Изначально протокол gRPC был разработан для мобильных клиентов, которым требовалось обмениваться данными с облачными серверами со строгими требованиями к задержкам. Он очень эффективен для передачи структурированных данных через *протокольные буферы* (**Protocol buffer**, **Protobuf**), которые являются ключевым компонентом протокола. При использовании Protobuf данные упаковываются в двоичный формат вместо текстового (JSON или XML). Этот формат не только уменьшает размер данных, но и очень эффективен для сериализации и десериализации данных по сравнению с JSON и XML. Более того, данные передаются по HTTP/2 вместо HTTP 1.1. HTTP/2 предлагает как модель *«запрос-ответ»*, так и модель *двунаправленной связи*, которая позволяет клиентам устанавливать долгосрочные соединения, что значительно ускоряет передачу данных. Эти две технологии делают протокол gRPC в 7–10 раз быстрее, чем REST API.

Другими словами, gNMI — это специфическая реализация протокола gRPC для управления сетью и телеметрии. Как и NETCONF, он использует YANG и предлагает мало операций (`Get`, `Set` и `Subscribe`). gNMI все чаще используется для сбора данных в телеметрии, чем для управления сетью. Главная причина в том, что он не обеспечивает такой гибкости, как NETCONF, зато он оптимизирован для сбора данных из удаленных систем, особенно в режиме реального или близкого к реальному времени.

Далее мы рассмотрим библиотеки Python для взаимодействия с сетевыми устройствами.

Взаимодействие с сетевыми устройствами с помощью библиотек Python на основе SSH

Существует несколько библиотек для взаимодействия с сетевыми устройствами по SSH. Paramiko, Netmiko и NAPALM — три популярных доступных варианта, которые мы рассмотрим далее.

Paramiko

Paramiko — это абстракция протокола SSHv2, которая включает функционал, как на стороне сервера, так и на стороне клиента. Здесь мы сосредоточимся только на возможностях клиента.

При взаимодействии с сетевым устройством мы пытаемся либо получить данные конфигурации, либо задать новую конфигурацию для определенных объектов. В первом случае мы используем CLI-команды типа *show* в зависимости от ОС устройства. А во втором может потребоваться специальный режим выполнения CLI-команд. При работе с библиотеками Python эти два типа команд обрабатываются по-разному.

Для подключения к сетевому устройству (которое прослушивает как SSH-сервер) потребуется использовать экземпляр класса `paramiko.SSHClient` или напрямую использовать низкоуровневый класс `paramiko.Transport`. Класс `Transport` предлагает низкоуровневые методы для прямого управления коммуникациями на основе сокетов. Класс `SSHClient` служит оберткой и использует класс `Transport` для управления сеансом с помощью SSH-сервера, реализованного на сетевом устройстве.

Можно использовать Paramiko для установки соединения с сетевым устройством (Cisco IOS XR в нашем случае) и выполнения *show*-команды (в нашем случае `show ip int brief`):

```
#show_cisco_int_pmk.py
import paramiko

host='HOST_ID'
port=22
username='xxx'
password='xxxxxx'

# команда cisco ios для получения списка IP-интерфейсов
cmd= 'show ip int brief \n'
def main():
    try:
        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.
 AutoAddPolicy())
        ssh.connect(host, port, username, password)
        stdin, stdout, stderr = ssh.exec_command(cmd)
        output_lines = stdout.readlines()
```

```
        response = ''.join(output_lines)
        print(response)
    finally:
        ssh.close()
if __name__ == '__main__':
    main()
```

Ключевые моменты кода:

- Мы создали экземпляр `SSHClient` и открыли соединение с SSH-сервером.
- Поскольку мы не используем ключ хоста, мы применили метод `set_missing_host_key_policy` во избежание каких-либо предупреждений и ошибок.
- Установив SSH-соединение, мы отправили *show*-команду (`show ip int brief`) на хост-компьютер по каналу SSH и получили вывод в виде SSH-ответа.
- Результатом работы программы будет кортеж объектов `stdin`, `stdout` и `stderr`; если команда выполнена успешно, мы получим вывод из объекта `stdout`.

При выполнении на устройстве Cisco IOS XR вывод программы будет следующим:

```
Mon Jul 19 12:03:41.631 UTC
Interface                      IP-Address      Status     Protocol
Loopback0                      10.180.180.10   Up         Up
GigabitEthernet0/0/0/0         10.1.10.2       Up         Up
GigabitEthernet0/0/0/0.100     unassigned      Up         Down
GigabitEthernet0/0/0/1         unassigned      Up         Up
GigabitEthernet0/0/0/1.100     150.150.150.1   Up         Up
GigabitEthernet0/0/0/2         unassigned      Shutdown   Down
```

Если нужно выполнить эту программу на устройстве другого типа, следует изменить команду, заданную в качестве переменной `cmd`, в соответствии с типом устройства.

Библиотека Paramiko обеспечивает низкоуровневый контроль над сетевым взаимодействием. Но иногда она может неправильно работать из-за нестандартной или неполной реализации протокола SSH на многих сетевых устройствах. Если вы столкнулись с такими проблемами, дело не на вашей стороне или в Paramiko, а в том, как устройство воспринимает ваши с ним взаимодействия. Низкоуровневый транспортный канал может решить эти проблемы, но для этого требуется немного сложного кода. В такой ситуации на помощь приходит библиотека Netmiko.

Netmiko

Netmiko — абстрактная библиотека для управления сетью, созданная на основе Paramiko. Она помогает устранить проблемы Paramiko за счет разного взаимодействия с разными сетевыми устройствами. Netmiko скрывает многие детали коммуникации на уровне устройства и поддерживает несколько устройств от разных производителей (Cisco, Arista, Juniper и Nokia).

Для подключения к сетевому устройству с помощью CLI-команд типа *show* нужно задать определение `device_type`, которое используется для подключения к целевому устройству. Определение `device_type` представляет собой словарь, который должен включать тип устройства, IP-адрес хоста или *полное доменное имя* (**Fully Qualified Domain Name, FQDN**) устройства, имя пользователя и пароль для подключения. Можно задать номер порта для SSH-соединения, если целевой компьютер прослушивает порт, отличный от 22. Следующий код можно использовать для выполнения той же команды *show*, которую мы выполняли с библиотекой Paramiko:

```
#show_cisco_int_nmk.py
from netmiko import ConnectHandler

cisco_rtr = {
    "device_type": "cisco_ios",
    "host": "HOST_ID",
    "username": "xxx",
    "password": "xxxxxxx",
    #"global_delay_factor": 2,
}

def main():
    command = "show ip int brief"
    with ConnectHandler(**cisco_rtr) as net_connect:
        print(net_connect.find_prompt())
        print(net_connect.enable())
        output = net_connect.send_command(command)
    print(output)
```

Ключевые моменты кода:

- Мы создали сетевое соединение с помощью класса `ConnectHandler`, используя менеджер контекста, который будет управлять жизненным циклом подключения.
- Netmiko предлагает простой метод `find_prompt` для захвата *запроса* (**prompt**) целевого устройства, который будет полезен для парсинга вывода от многих сетевых устройств; для Cisco IOS XR этого делать необязательно, но мы задействовали метод, поскольку это рекомендованная практика.
- Netmiko также позволяет войти в режим *Enable* (запрос командной строки, `#`) на устройствах Cisco IOS с помощью метода `enable`; этого также делать необязательно, но является рекомендованной практикой, особенно если мы отправляем конфигурационные CLI-команды как часть одного скрипта.
- Мы выполнили команду `show ip int brief`, используя метод `send_command`, и получили тот же результат, что и для программы `show_cisco_int_pmk.py`.

Если сравнить два примера кода с одинаковой реализацией команды *show*, можно увидеть, что работать с Netmiko намного удобнее, чем с Paramiko.

> ***Важное примечание***
>
> Важно установить верный тип устройства для получения согласованных результатов, даже если используются устройства одного производителя. Это особенно важно при выполнении команд по конфигурации. Неправильный тип устройства может привести к конфликтующим ошибкам.

Иногда выполняются команды, которые требуют больше времени, чем обычные команды *show*. Например, необходимо скопировать большой файл из одного места на устройстве в другое, и известно, что это займет несколько сотен секунд. По умолчанию Netmiko ожидает выполнения команды около 100 секунд. Можно добавить глобальный коэффициент задержки при определении устройства:

```
"global_delay_factor": 2
```

Время ожидания для всех команд этого устройства будет увеличено в 2 раза. В качестве альтернативы можно задать коэффициент задержки для отдельной команды с помощью метода `send_command`, передав следующий аргумент:

```
delay_factor=2
```

Коэффициент необходимо указывать, если заранее известно, что потребуется большое время задержки. Когда мы его добавляем, следует также добавить еще один атрибут в качестве аргумента метода `send_command`, который прервет цикл ожидания досрочно, если появится командный запрос на данное действие (например, `#` для устройств Cisco IOS). Это можно сделать с помощью атрибута:

```
expect_string=r'#'
```

В следующем примере мы отправим конфигурацию на устройство. Процесс настройки с помощью Netmiko аналогичен выполнению команд *show*, поскольку библиотека позаботится о включении терминала конфигурации (при необходимости, в зависимости от типа устройства) и корректном выходе из него.

Для примера будет задано описание интерфейса (`description`) следующим образом:

```
#config_cisco_int_nmk.py
from netmiko import ConnectHandler

cisco_rtr = {
    "device_type": "cisco_ios",
    "host": "HOST_ID",
    "username": "xxx",
    "password": "xxxxxx",
}

def main():
    commands = ["int Lo0", "description custom_description", "commit"]
    with ConnectHandler(**cisco_rtr) as net_connect:
        output = net_connect.send_config_set(commands)

    print(output)
    print()
```

Ключевые моменты кода:

- Мы создали список из трех команд (`int <id интерфейса>`, `description <новое описание>` и `commit`); первые две команды также можно отправлять в качестве одной, но для наглядности мы их разделили; команда `commit` используется для сохранения изменений.
- Когда мы отправляем команду на устройство, мы используем метод `send_config_set` для установки соединения в целях настройки; успешное выполнение этого шага зависит от правильной установки типа устройства; это связано с тем, что разные типы устройств по-разному могут реагировать на команды конфигурации.
- Этот набор из трех команд добавит или обновит атрибут `description` для указанного интерфейса.

От программы не ожидается никакого специфического вывода, кроме запросов конфигурации устройства. Вывод консоли будет следующим:

```
Mon Jul 19 13:21:16.904 UTC
RP/0/RP0/CPU0:cisco(config)#int Lo0
RP/0/RP0/CPU0:cisco(config-if)#description custom_description
RP/0/RP0/CPU0:cisco(config-if)#commit
Mon Jul 19 13:21:17.332 UTC
RP/0/RP0/CPU0:cisco(config-if)#
```

Netmiko предлагает гораздо больше возможностей, но мы оставим это для самостоятельного изучения в официальной документации (*https://pypi.org/project/netmiko/*). Примеры кода, которые обсуждались в этом подразделе, использовались для устройства Cisco, но ту же программу можно использовать, изменив тип устройства и команды для любого другого устройства, если оно поддерживается библиотекой.

Netmiko упрощает код для взаимодействия с сетевыми устройствами, но для получения и передачи конфигурации по-прежнему нужны CLI-команды. В Netmiko нелегко программировать, но здесь может помочь другая библиотека — NAPALM.

NAPALM

NAPALM — это аббревиатура от **Network Automation and Programmability Abstraction Layer with Multivendor**. Библиотека обеспечивает следующий уровень абстракции поверх Netmiko, предлагая набор функций в виде единого API для взаимодействия с несколькими сетевыми устройствами. NAPALM поддерживает меньше устройств, чем Netmiko. В версии 3 доступны основные драйверы для сетевых устройств Arista EOS, Cisco IOS, Cisco IOS-XR, Cisco NX-OS и Juniper JunOS. Однако существуют драйверы, созданные сообществом, для многих других устройств, например, Nokia SROS, Aruba AOS-CX и Ciena SAOS.

Как и в случае с Netmiko, мы создадим примеры для взаимодействия с сетевым устройством с помощью NAPALM. В первом примере мы получим список IP-

интерфейсов, а во втором добавим или обновим атрибут `description` для IP-интерфейса. Оба примера будут выполнять те же операции, что и в примерах с Paramiko и Netmiko.

Для получения конфигурации сетевого устройства сначала нужно установить соединение. Мы будем делать это в обоих примерах. Установка соединения состоит из трех шагов:

1. Мы должны узнать класс драйвера, который поддерживает тип устройства . Для этого можно использовать функцию `get_network_driver`.
2. Узнав класс драйвера, можно создать объект устройства, предоставив аргументы `host id`, `username` и `password` конструктору класса.
3. Далее подключимся к устройству с помощью метода `open` объекта устройства. Все эти шаги могут быть реализованы на Python:

```
from napalm import get_network_driver
driver = get_network_driver('iosxr')
device = driver('HOST_ID', 'xxxx', 'xxxx')
device.open()
```

Установив соединение, можно вызывать такие методы, как `get_interfaces_ip` (эквивалент CLI-команды `show interfaces`) или `get_facts` (эквивалент `show version`). Полный код для двух методов выглядит следующим образом:

```
#show_cisco_int_npm.py
from napalm import get_network_driver
import json

def main():
    driver = get_network_driver('iosxr')
    device = driver('HOST_ID', 'root', 'rootroot')
    try:
        device.open()
        print(json.dumps(device.get_interfaces_ip(), indent=2))
        #print(json.dumps(device.get_facts(), indent=2))

    finally:
        device.close()
```

Что интересно, вывод программы по умолчанию будет представлен в формате JSON. NAPALM по умолчанию преобразует вывод CLI-команд в словарь, который легко использовать в Python. Фрагмент выходных данных показан ниже:

```
{
  "Loopback0": {
    "ipv4": {
      "10.180.180.180": {
        "prefix_length": 32
      }
```

```
    }
  },
  "MgmtEth0/RP0/CPU0/0": {
    "ipv4": {
      "172.16.2.12": {
        "prefix_length": 24
      }
    }
  }
}
```

В следующем примере мы попробуем добавить или обновить атрибут description для существующего IP-интерфейса:

```
#config_cisco_int_npm.py
from napalm import get_network_driver
import json

def main():

    driver = get_network_driver('iosxr')
    device = driver('HOST_ID', 'xxx', 'xxxx')
    try:
        device.open()
        device.load_merge_candidate(config='interface Lo0 \n
           description napalm_desc \n end\n')
        print(device.compare_config())
        device.commit_config()
    finally:
        device.close()
```

Ключевые моменты кода:

- Для настройки IP-интерфейса мы использовали метод load_merge_candidate и передали ему тот же набор CLI-команд, что и в примере с Netmiko.
- Затем мы сравнили конфигурации *до* и *после* выполнения команд, используя метод compare_config; это позволяет посмотреть, какая конфигурация добавлена, а какая удалена;
- Мы зафиксировали все изменения, используя метод commit_config.

Вывод примера будет отображать только разницу между изменениями:

```
---
+++
@@ -47,7 +47,7 @@
  !
 !
 interface Loopback0
- description my custom description
```

```
+ description napalm added new desc
  ipv4 address 10.180.180.180 255.255.255.255
 !
 interface MgmtEth0/RP0/CPU0/0
```

Любая строка, которая начинается с «-» указывает на удаляемую конфигурацию, а любая строка с «+» — на добавляемую.

Оба примера демонстрируют базовый набор возможностей NAPALM для одного типа устройства. Библиотека может использоваться для одновременной настройки нескольких устройств и работать с различными наборами конфигураций.

В следующем подразделе мы обсудим взаимодействие с сетевыми устройствами по протоколу NETCONF.

Взаимодействие с сетевыми устройствами с помощью NETCONF

Протокол NETCONF был создан для модельно-ориентированного (или объектно-ориентированного) управления сетью, особенно в целях конфигурирования. При работе с сетевым устройством через NETCONF важно понимать две возможности устройства:

- Для отправки сообщений в правильном формате вы должны понимать YANG-модели устройств; полезный ресурс с YANG-моделями от разных вендоров можно найти по адресу *https://github. com/YangModels/yang*.
- Можно включить порты NETCONF и SSH для протокола NETCONF на сетевом устройстве; в нашем случае это будет виртуальное устройство Cisco IOS XR, как и в предыдущих примерах.

Прежде чем переходить к управлению сетью, нужно сначала проверить NETCONF-возможности устройства и сведения о конфигурации источника данных NETCONF. Для всех примеров в этом подразделе будет использоваться клиентская библиотека NETCONF для Python — `ncclient`. Она предлагает удобные методы для отправки запросов RPC. Ниже показан пример программы, использующий `ncclient` для получения возможностей и полной конфигурации устройства:

```
#check_cisco_device.py
from ncclient import manager

with manager.connect(host='device_ip, username=xxxx,
 password=xxxxxx, hostkey_verify=False) as conn:
   capabilities = []
   for capability in conn.server_capabilities:
       capabilities.append(capability)
   capabilities = sorted(capabilities)
   for cap in capabilities:
       print(cap)

   result = conn.get_config(source="running")
   print (result)
```

Объект `manager` из библиотеки `ncclient` используется для подключения к устройству по SSH, но через NETCONF-порт `830` (по умолчанию). Сначала мы получаем список возможностей сервера через экземпляр соединения, а затем выводим их в отформатированном виде для удобного чтения. Дальше мы инициировали NETCONF-операцию `get-config`, используя метод `get_config` библиотеки класса `manager`. Вывод программы очень объемный и включает все возможности и конфигурацию устройства. Мы оставим возможность самостоятельно изучить результат и узнать о возможностях вашего устройства.

Важно понимать, целью этого подраздела является не изучение NETCONF, а понимание, как использовать Python и `ncclient` для работы с NETCONF. Для этого напишем два примера: один — для получения конфигурации интерфейсов устройства, другой — для обновления описания интерфейса, как мы делали с предыдущими библиотеками Python.

Получение интерфейсов через NETCONF

Ранее мы узнали, что наше устройство (Cisco IOS XR) поддерживает интерфейсы с помощью реализации **OpenConfig**, которая доступна по адресу *http://openconfig.net/yang/interfaces?module=openconfig-interfaces*.

Мы также можем проверить формат XML конфигурации интерфейса, который был получен в выводе метода `get_config`. В этом примере мы просто передаем XML-фильтр с конфигурацией интерфейса в качестве аргумента методу `get_config`:

```
#show_all_interfaces.py
from ncclient import manager
with manager.connect(host='device_ip', username=xxx,
                password='xxxx', hostkey_verify=False) as conn:
    result = conn.get_config("running", filter=('subtree',
      '<interfaces xmlns= "http://openconfig.net/yang/interfaces"/>'))
    print (result)
```

Результат программы будет содержать список интерфейсов. Ниже показан только фрагмент для демонстрации:

```
<rpc-reply message-id="urn:uuid:f4553429-
ede6-4c79-aeea-5739993cacf4"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <data>
  <interfaces xmlns="http://openconfig.net/yang/interfaces">
   <interface>
    <name>Loopback0</name>
    <config>
     <name>Loopback0</name>
     <description>Configured by NETCONF</description>
    </config>
<!--rest of the output is skipped -->
```

Для получения выборочного набора интерфейсов будем использовать расширенную версию XML-фильтра на основе YANG-модели интерфейса. Для следующего примера определим XML-фильтр со свойствами name интерфейсов в качестве критериев фильтрации. Поскольку этот фильтр занимает несколько строк, он будет определен отдельно как строковый объект. Код с XML-фильтром выглядит следующим образом:

```
#show_int_config.py
from ncclient import manager
# Создаем шаблон фильтра для интерфейса
filter_temp = """
<filter>
    <interfaces xmlns="http://openconfig.net/yang/interfaces">
        <interface>
            <name>{int_name}</name>
        </interface>
    </interfaces>
</filter>"""

with manager.connect(host='device_ip', username=xxx,
               password='xxxx', hostkey_verify=False) as conn:
    filter = filter_temp.format(int_name = "MgmtEth0/RP0/CPU0/0")
    result = m.get_config("running", filter)
    print (result)
```

Выводом программы будет один интерфейс (в соответствии с конфигурацией устройства):

```
<?xml version="1.0"?>
<rpc-reply message-id="urn:uuid:c61588b3-
1bfb-4aa4-a9de-2a98727e1e15"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <data>
  <interfaces xmlns="http://openconfig.net/yang/interfaces">
   <interface>
    <name>MgmtEth0/RP0/CPU0/0</name>
    <config>
     <name>MgmtEth0/RP0/CPU0/0</name>
    </config>
    <ethernet xmlns="http://openconfig.net/yang/interfaces/ethernet">
     <config>
      <auto-negotiate>false</auto-negotiate>
     </config>
    </ethernet>
    <subinterfaces>
```

```
    <@!- ommitted sub interfaces details to save space -->
   </subinterfaces>
  </interface>
 </interfaces>
</data>
</rpc-reply>
```

Также можно определить XML-фильтры в XML-файле, а затем считать его содержимое в строковый объект. Еще один вариант — использовать шаблоны Jinja, если планируется много работать с фильтрами.

Далее обсудим, как обновить описание интерфейса.

Обновление описания интерфейса

Для настройки атрибута интерфейса `description` необходимо использовать модель YANG, доступную по адресу: *http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg*.

Более того, блок XML для настройки интерфейса отличается от блока XML для получения конфигурации. Для обновления интерфейса необходимо использовать следующий шаблон, определенный в отдельном файле:

```
<!--config-template.xml-->
<config xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
 <interface-configurations xmlns="http://cisco.com/ns/yang/
 Cisco-IOS-XR-ifmgr-cfg">
   <interface-configuration>
    <active>act</active>
    <interface-name>{int_name}</interface-name>
    <description>{int_desc}</description>
   </interface-configuration>
 </interface-configurations>
</config>
```

В этом шаблоне заданы заглушки для свойств интерфейса `name` и `description`. Далее мы напишем программу для чтения этого шаблона и вызова NETCONF-операции `edit-config` с помощью метода `edit_config` библиотеки `ncclient`. В результате шаблон будет отправлен в хранилище-кандидат на устройстве:

```
#config_cisco_int_desc.py
from ncclient import manager

nc_template = open("config-template.xml").read()
nc_payload = nc_template.format(int_name='Loopback0',
                          int_desc="Configured by NETCONF")

with manager.connect(host='device_ip, username=xxxx,
                     password=xxx, hostkey_verify=False) as nc:
```

```
netconf_reply = nc.edit_config(nc_payload, target="candidate")
print(netconf_reply)
reply = nc.commit()
print(reply)
```

Здесь важно отметить два момента. Во-первых, устройство Cisco IOS XR настроено на прием новой конфигурации только через *кандидата*. Если попытаться задать для атрибута `target` значение `running`, произойдет сбой. Во-вторых, сделать новую конфигурацию *рабочей* можно, только вызвав метод `commit` после операции `edit-config` в том же сеансе. Вывод программы будет содержать два ответа `OK` от сервера NETCONF:

```
<?xml version="1.0"?>
<rpc-reply message-id="urn:uuid:6d70d758-
6a8e-407d-8cb8-10f500e9f297"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <ok/>
</rpc-reply>
<?xml version="1.0"?>
<rpc-reply message-id="urn:uuid:2a97916bdb5f-427d-9553-de1b56417d89"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <ok/>
</rpc-reply>
```

На этом можно завершить обсуждение, как использовать Python для операций NETCONF. Мы рассмотрели работу двух основных из них (`get-config` и `edit-config`) с библиотекой `ncclient`.

Далее рассмотрим интеграцию с системами управления сетью с помощью Python.

Интеграция с системами управления сетью

Системы управления сетью (**Network Management System, NMS**) или *сетевые контроллеры* — это приложения для управления сетью с *графическим пользовательским интерфейсом* (**Graphical User Interface, GUI**). Они включают в себя инвентаризацию сети, ее подготовку, управление сбоями, а также взаимодействие с сетевыми устройствами, которое происходит с использованием комбинации разных протоколов связи для разных задач, например, SSH/NETCONF для подготовки сети, SNMP для предупреждений и мониторинга устройств и gRPC для сбора данных телеметрии. Эти системы также предлагают возможности автоматизации с помощью механизмов обработки скриптов и рабочих процессов.

Основным ценным аспектом здесь является объединение всего функционала сетевых устройств в единую систему и ее последующее предоставление через *северные интерфейсы* (**North Bound Interface, NBI**), которые обычно являются интерфейсами REST или RESTCONF. Эти системы также могут отправлять уведомления о событиях в реальном времени, например, рассылать оповещения через Apache Kafka или другую *событийно-ориентированную систему* (**Event-driven System**). В этом подразделе мы рассмотрим несколько примеров использования REST API в NMS. Интеграцию с Apache Kafka обсудим дальше в подразделе *«Интеграция с событийно-ориентированными системами»*.

Для работы с NMS мы будем использовать общедоступную лабораторию Nokia на онлайн-портале для разработчиков (*https://network.developer.nokia.com/*). Она имеет несколько IP-маршрутизаторов Nokia и NSP. Лаборатория предоставляется бесплатно на ограниченное время (3 часа в день на момент написания книги). Вам необходимо будет создать бесплатный аккаунт на портале. При бронировании вы получите электронное письмо с инструкциями о подключении к лаборатории, а также необходимые данные для VPN. Если вы являетесь сетевым инженером и имеете доступ к любой другой NMS или контроллеру, можете использовать их для упражнений в этом разделе, внеся необходимые корректировки.

Для использования REST API от Nokia NSP необходимо взаимодействовать со шлюзом REST API, который управляет несколькими конечными точками для Nokia NSP. Можно начать работу со шлюзом REST API, используя сервисы определения местоположения.

Использование конечных точек сервиса определения местоположения

Сначала необходимо понять, какие конечные точки API доступны. Для этого Nokia NSP предлагает конечную точку сервиса определения местоположения, которая предоставляет список всех конечных точек API. Для работы с любым REST API в этом разделе мы будем использовать Python-библиотеку `requests`. Она хорошо известна тем, что отправляет HTML-запросы на сервер с помощью протокола HTTP, и мы уже использовали ее в предыдущих главах. Для получения списка конечных точек из Nokia NSP мы используем следующий код для вызова API сервиса определения местоположения:

```
#location_services1.py
import requests
payload = {}
headers = {}
url = "https://<NSP URL>/rest-gateway/rest/api/v1/location/services"
resp = requests.request("GET", url, headers=headers, data=payload)
print(resp.text)
```

Ответ будет содержать несколько десятков конечных точек в формате JSON. Для понимания, как работает каждый API, можно ознакомиться с онлайн-документацией Nokia NSP по адресу *https://network. developer.nokia.com/api-documentation/*. Если нужно найти определенную конечную точку, можно изменить значение переменной `url` в приведенном выше коде:

```
url = "https://<NSP URL>/rest-gateway/rest/api/v1/location/
services/endpoints?endPoint=/v1/auth/token
```

С помощью этого нового URL-адреса мы пытаемся найти конечную точку для токена авторизации (`/v1/auth/token`). Вывод программы для нового URL будет следующий:

```
{
 "response": {
  "status": 0,
  "startRow": 0,
  "endRow": 0,
  "totalRows": 1,
  "data": {
   "endpoints": [
    {
    "docUrl":"https://<NSP_URL>/rest-gateway/api-docs#!/authent..",
    "effectiveUrl": "https://<NSP_URL>/rest-gateway/rest/api",
    "operation": "[POST]"
    }
   ]
  },
  "errors": null
 }
}
```

Обратите внимание, для использования API сервисов определения местоположения не требуется аутентификации. Однако понадобится токен аутентификации для вызова любого другого API. Далее узнаем, как его получить.

Получение токена аутентификации

На следующем шаге мы используем `effectiveUrl` из вывода предыдущего кода для получения токена аутентификации. Этот API требует передачу `username` и `password` в кодировке Base64 в качестве атрибута `Authorization` заголовка HTTP. Код для вызова API аутентификации выглядит следующим образом:

```
#get_token.py
import requests
from base64 import b64encode
import json
```

```
#получаем кодировку base64
message = 'username'+ ':' +'password'
message_bytes = message.encode('UTF-8')
basic_token = b64encode(message_bytes)
payload = json.dumps({
  "grant_type": "client_credentials"
})
headers = {
  'Content-Type': 'application/json',
  'Authorization': 'Basic {}'.format(str(basic_token,'UTF-8'))
}
url = "https://<NSP SERVER URL>/rest-gateway/rest/api/v1/auth/token"
resp = requests.request("POST", url, headers=headers,data=payload)
token = resp.json()["access_token"]
print(resp)
# При выполнении этого кода Python мы получим токен на один час, который можно
# использовать для любого API NSP.
{
  "access_token": "VEtOLVNBTXFhZDQ3MzE5ZjQtNWUxZjQ0YjNl",
  "refresh_token": "UkVUS04tU0FNcWF5ZlMTmQ0ZTA5MDNlOTY=",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Также доступен *токен обновления*, который можно использовать для обновления текущего токена до истечения срока его действия. Рекомендуется обновлять его каждые 30 минут. Это можно сделать с помощью того же токена аутентификации, но при этом необходимо отправить следующие атрибуты в теле HTTP-запроса:

```
payload = json.dumps({
  "grant_type": "refresh_token",
  "refresh_token": "UkVUS04tU0FNcWF5ZlMTmQ0ZTA5MDNlOTY="
})
```

Другой хорошей практикой является отзыв токена, когда он больше не нужен. Для этого можно использовать следующую конечную точку API:

```
url = "https://<NSP URL>rest-gateway/rest/api/v1/auth/revocation"
```

Получение сетевых устройств и инвентаризация интерфейсов

Получив токен аутентификации, можно использовать REST API для приема настроенных данных и добавления новой конфигурации. Начнем с простого кода для получения списка всех устройств в сети, управляемой NSP. В этом примере используется токен, который уже был извлечен с помощью API токена:

```
#get_network_devices.py
import requests
pload={}
```

```
headers = {
  'Authorization': 'Bearer {token}'.format(token)
}
url = "https://{{NSP_URL}}:8544/NetworkSupervision/rest/api/v1/networkElements"
response = requests.request("GET", url, headers=headers, data=pload)
print(response.text)
```

Вывод программы будет содержать список сетевых устройств с их атрибутами. Мы не будем демонстрировать вывод из-за большого объема данных.

Следующий пример покажет, как получить список портов (интерфейсов) устройства с помощью фильтра. Обратите внимание, фильтры можно применить и к сетевым устройствам. В примере мы попросим NSP API предоставить список портов по имени порта (`Port 1/1/1` в нашем случае):

```
#get_ports_filter.py
import requests
payload={}
headers = {
  'Authorization': 'Bearer {token}'.format(token)
}
url = "https://{{server}}:8544/NetworkSupervision/rest/api/v1/
  ports?filter=(name='Port 1/1/1')
response = requests.request("GET", url, headers=headers, data=payload)
print(response.text)
```

Результатом программы будет список портов с именем `Port 1/1/1` на всех сетевых устройствах. Возможность получить порты на нескольких сетевых устройствах с помощью API — это одно из главных преимуществ работы с NMS.

Далее мы узнаем, как обновить сетевой ресурс с помощью NMS API.

Обновление порта на сетевом устройстве

NMS API позволяет удобно создавать новые объекты или обновлять существующие. Мы попробуем обновить описание порта, как в примерах ранее с библиотеками Netmiko, NAPALM и ncclient. Для обновления порта или интерфейса мы будем использовать другую конечную точку, доступную в модуле **Network Function Manager for Packet (NFMP)**. NFMP — это модуль NMS для сетевых устройств Nokia на платформе Nokia NSP. Пошагово рассмотрим, как обновить описание порта или внести любое другое изменение в сетевой ресурс:

1. Для обновления объекта или создания нового в уже существующем, потребуется *полное имя объекта* (**Object Full Name, OFN**), также известное как *полное различимое имя* (**Fully Distinguishable Name, FDN**). OFN существующего объекта используется для обновления, а для создания нового используется OFN родительского объекта. *Полное имя объекта* выступает как первичный ключ для

уникальной идентификации объекта. В сетевых объектах Nokia, доступных в модулях NSP, каждый объект имеет атрибут OFN или FDN. Для получения OFN для обновляемого порта мы будем использовать API `v1/managedobjects/searchWithFilter` со следующими критериями фильтрации:

#update_port_desc.py (часть 1)

```
import requests
import json

token = <token obtain earlier>
headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer {}'.format(token)
}

url1 = "https://NFMP_URL:8443/nfm-p/rest/api/v1/
  managedobjects/searchWithFilter"

payload1 = json.dumps({
  "fullClassName": "equipment.PhysicalPort",
  "filterExpression": "siteId ='<site id>' AND portName='123",
  "resultFilter":[
    "objectFullName",
    "description"
  ]
})
response = requests.request("POST", url1,
headers=headers, data=payload1, verify=False)
port_ofn = response.json()[0]['objectFullName']
```

2. В этом коде мы задаем имя объекта `fullClassNames`. Полные имена класса объекта доступны в документации по объектной модели Nokia NFMP. Мы задаем `filterExpression` для поиска уникального порта по идентификатору сайта устройства и имени порта. Атрибут `resultFilter` ограничивает атрибуты, возвращаемые API в ответе. Нас интересует атрибут `objectFullName` в ответе этого API.
3. Затем мы используем другую конечную точку `v1/managedobjects/ofn` для обновления атрибута сетевого объекта. В нашем случае мы меняем только атрибут `description`. Для операции обновления необходимо задать атрибут `fullClassName` в полезных данных и новое значение для атрибута `description`. К URL-адресу конечной точки мы добавим переменную `port_ofn`, вычисленную на предыдущем шаге. Пример кода для этой части программы выглядит следующим образом:

#update_port_desc.py (часть 2)

```
payload2 = json.dumps({
  "fullClassName": "equipment.PhysicalPort",
  "properties": {
    "description": "description added by a Python
```

```
    program"
    }
})

url2 = "https:// NFMP_URL:8443/nfm-p/rest/api/v1/managedobjects/"+port_ofn

response = requests.request("PUT", url2, headers=headers, data=payload2,
  verify=False)

print(response.text)
```

Автоматизация сети — это процесс создания и обновления множества сетевых объектов в заданном порядке. Например, можно обновить порт перед созданием службы IP-подключения для объединения двух и более локальных сетей. Такой сценарий использования требует выполнения ряда задач для обновления всех задействованных портов, а также многих других объектов. NMS API позволяет организовать все эти задачи в программе для реализации автоматизированного процесса.

Далее мы узнаем, как выполнить интеграцию с Nokia NSP или аналогичными системами для событийно-ориентированного взаимодействия.

Интеграция с событийно-ориентированными системами

Ранее мы обсуждали, как взаимодействовать с сетевыми устройствами и системами управления сетью с помощью модели *«запрос-ответ»*, когда клиент отправляет запрос серверу, а тот посылает ответ на этот запрос. Протоколы HTTP (REST API) и SSH основаны на этой модели. Она хорошо подходит для настройки системы или получения рабочего состояния сети на разовой или периодической основе. Но что если какое-то событие в сети потребует вмешательства рабочей команды? Предположим, произошел аппаратный сбой оборудования или обрыв кабеля. Сетевые устройства в таких случаях подают сигналы оповещения, которые должны быть немедленно доведены до оператора (электронная почта, SMS-сообщение или панель мониторинга).

Можно использовать модель *«запрос-ответ»* для ежесекундного опроса устройства и проверки, изменилось ли его состояние и возникло ли новое оповещение. Однако это является неэффективным использованием ресурсов, которое приведет к излишнему трафику. Было бы удобнее, если сетевые устройства или NMS сами отправляли сообщение клиентам при изменении состояния важных ресурсов. Такой тип модели называется *событийно-ориентированной* или *управляемой событиями*, и это популярный подход к мониторингу в реальном времени.

Такие системы можно реализовать либо с помощью *вебхуков* (**Webhook**) или *вебсокетов* (**WebSocket**), либо с помощью *потокового подхода*. Веб-сокеты обеспечивают двунаправленный транспортный канал по HTTP 1.1 через сокет TCP/IP. Поскольку здесь не используется традиционная модель *«запрос-ответ»*, веб-сокеты

хорошо подходят, когда нужно установить прямое (**one-to-one**) соединение между двумя системами. Это один из лучших вариантов взаимодействия между двумя программами в реальном времени. Веб-сокеты поддерживаются всеми стандартными браузерами, в том числе на устройствах iPhone и Android. Также это является популярным вариантом для многих социальных сетей, стриминговых приложений и онлайн-игр.

Веб-сокеты — легкое решение для получения событий в реальном времени. Но когда множество клиентов хотят получать события из одной системы, более эффективным и масштабируемым решением будет *потоковая передача*. Такая событийно-управляемая модель обычно соответствует шаблону *«издатель-подписчик»* и включает три основных компонента:

- *Тема* или *топик* (**Topic**): все потоковые сообщения или уведомления о событиях хранятся в *теме*, которая представляет собой что-то вроде каталога; топик помогает подписываться на интересующие нас темы, иначе мы будем получать все происходящие события.
- *Издатель* (**Producer** или **Publisher**): это программа или часть ПО, которая отправляет события или сообщения в тему; в нашем случае это приложение NSP.
- *Подписчик* или *потребитель* (**Consumer** или **Subscriber**): это программа, которая извлекает события или сообщения из темы; в нашем примере это написанная нами программа Python.

Событийно-ориентированные системы можно использовать как с сетевыми устройствами, так и NMS. Платформы NMS используют системы событий (gRPC или SNMP) для получения сообщений от сетевых устройств в реальном времени и предлагают совместные интерфейсы для уровня оркестрации или для приложений по мониторингу и эксплуатации. В нашем примере мы будем взаимодействовать с событийно-управляемой системой на основе **Apache Kafka** от платформы Nokia NSP. Apache Kafka — это ПО с открытым исходным кодом, разработанное на Scala и Java. Оно обеспечивает реализацию шины для программного обмена сообщениями на основе модели *«издатель-подписчик»*. Прежде чем работать с Apache Kafka, перечислим список ключевых *категорий* (это другое название темы/топика в Apache Kafka), предлагаемых в системе Nokia NSP:

- `NSP-FAULT`: категория событий, связанных со сбоями или аварийными оповещениями;
- `NSP-PACKET-ALL`: категория используется для всех событий управления сетью, включая события поддержания активности (**keep-alive**);
- `NSP-REAL-TIME-KPI`: категория событий для потоковой передачи уведомлений в реальном времени;
- `NSP-PACKET-STATS`: категория статистических событий.

Полный список категорий доступен в документации Nokia NSP. Все они предлагают дополнительные фильтры для подписки на события определенного типа. В контексте Nokia NSP мы будем использовать Apache Kafka для создания новой подписки, а затем для обработки событий. Начнем с управления подписками.

Создание подписок для Apache Kafka

Для получения от Apache Kafka событий или сообщений нужно подписаться на тему или категорию. Обратите внимание, что одна подписка действительна только для одной категории. Обычно срок подписки истекает через 1 час, поэтому рекомендуется продлевать ее за 30 минут до окончания срока действия.

Мы создадим новую подписку, используя API `v1/notifications/subscriptions` и следующий код:

```
#subscribe.py
import requests

token = <token obtain earlier>
url = "https://NSP_URL:8544/nbi-notification/api/v1/
  notifications/subscriptions"

def create_subscription(category):
    headers = {'Authorization': 'Bearer {}'.format(token) }
    payload = {
        "categories": [
          {
            "name": "{}".format(category)
          }
        ]
    }
    response = requests.request("POST", url, json=payload,
                                headers=headers, verify=False)
    print(response.text)

if __name__ == '__main__':
    create_subscription("NSP-PACKET-ALL")
```

Вывод этой программы будет включать в себя важные атрибуты, например, `subscriptionId`, `topicId` и `expiresAt`:

```
{
   "response":{
      "status":0,
      "startRow":0,
      "endRow":0,
      "totalRows":1,
      "data": {
         "subscriptionId":"440e4924-d236-4fba-b590-a491661aae14",
         "clientId": null,
         "topicId":"ns-eg-440e4924-d236-4fba-b590-a491661aae14",
         "timeOfSubscription":1627023845731,
         "expiresAt":1627027445731,
```

```
            "stage":"ACTIVE",
            "persisted":true
        },
        "errors":null
    }
}
```

Атрибут `subscriptionId` используется для последующего продления или удаления подписки. Apache Kafka создаст тему специально для нее. Тема предоставляется через атрибут `topicId`, который можно использовать при подключении к Apache Kafka для получения событий. Это объясняет, почему мы называем общие темы категориями в Apache Kafka. Атрибут `expiresAt` указывает время истечения срок действия подписки.

Как только подписка будет готова, можно подключиться к Apache Kafka для получения событий.

Обработка событий от Apache Kafka

Написание базового потребителя в Kafka занимает не более нескольких строк кода Python при использовании библиотеки kafka-python. Для создания клиента Kafka мы будем использовать класс KafkaConsumer из этой библиотеки. Мы можем использовать следующий код для использования событий из темы, на которую мы подписаны:

```
#basic_consumer.py
topicid = 'ns-eg-ff15a252-f927-48c7-a98f-2965ab6c187d'
consumer = KafkaConsumer(topic_id, group_id='120',
                         bootstrap_servers=[host_id],
                         value_deserializer=lambda m: json.loads
                         (m.decode('ascii')),
                          api_version=(0, 10, 1))
try:
    for message in consumer:
        if message is None:
            continue
        else:
            print(json.dumps(message.value, indent=4, sort_keys=True))
except KeyboardInterrupt:
    sys.stderr.write('++++++ Aborted by user ++++++++\n')

finally:
    consumer.close()
```

Важно отметить, если вы используете Python 3.7 или выше, вам необходимо использовать библиотеку `kafka-python`. С более ранней версией — библиотеку `kafka`.

При использовании `kafka` с Python версии 3.7 или выше существует ряд известных проблем. Например, существует ситуация, когда `async` является ключевым словом в Python версии 3.7 или выше, но в библиотеке `kafka` оно используется как переменная. Также существуют проблемы с версией API при использовании библиотеки `kafka-python` с Python 3.7 и более поздними версиями. Этого можно избежать, задав правильную версию API в качестве аргумента (в нашем примере это версия `0.10.0`).

Мы рассмотрели самый простой пример базового потребителя Kafka. В репозитории для этой книги можно найти более сложный пример: *https://github.com/nokia/NSP-Integration-Bootstrap/tree/master/kafka/ kafka_cmd_consumer*.

Продление и удаление подписки

Продлить подписку в Nokia NSP можно через ту же конечную точку API, которая использовалась для создания подписки. Для этого надо в конец URL-адреса добавить атрибут `subscriptionId` и ресурс `renewals` следующим образом:

```
https://{{server}}:8544/nbi-notification/api/v1/notifications/
  subscriptions/<subscriptionId>/renewals
```

Удалить подписку можно через ту же конечную точку с атрибутом `subscriptionId` в конце URL-адреса, но используя HTTP-метод `Delete`. Конечная точка для запроса на удаление будет выглядеть следующим образом:

```
https://{{server}}:8544/nbi-notification/api/v1/notifications/
  subscriptions/<subscriptionId>
```

В обоих случаях мы не будем отправлять никаких аргументов в теле запроса.

На этом можно завершить обсуждение интеграции с NMS и сетевыми контроллерами с помощью модели *«запрос-ответ»* и событийно-ориентированных систем. Оба подхода станут хорошей отправной точкой для интеграции с другими системами управления.

Заключение

В этой главе мы рассмотрели автоматизацию сети, включая ее преимущества и сложности для поставщиков телекоммуникационных услуг. Мы также изучили ключевые варианты использования автоматизации. Затем обсудили транспортные протоколы, доступные для взаимодействия с сетевыми устройствами. Познакомились с разными способами автоматизации сети. Сначала мы увидели, как взаимодействовать с сетевыми устройствами напрямую с помощью протокола SSH. После мы использовали библиотеки Paramiko, Netmiko и NAPALM для передачи конфигурации с сетевого устройства и на него. В дополнение мы изучили, как использовать NETCONF с Python для взаимодействия с сетевым устройством. Мы также рассмотрели примеры кода для работы с NETCONF и использовали библиотеку

`ncclient` сначала для получения конфигурации IP-интерфейса, а затем для его обновления на сетевом устройстве.

В последней части главы мы рассмотрели, как взаимодействовать с системами управления сетью вроде Nokia NSP, используя Python в качестве клиента REST API и подписчика Kafka. Рассмотрели несколько примеров, как получить токен аутентификации, а затем отправили REST API в NMS для получения данных конфигурации и обновления конфигурации сети на устройствах.

В эту главу включено несколько примеров, которые покажут, как использовать Python для взаимодействия с устройствами через SSH, NETCONF и REST API на уровне NMS. Эти знания важны инженерам по автоматизации, которые хотят эффективно решать задачи с помощью Python.

Данная глава завершает книгу. Мы рассмотрели не только основные концепции Python, но также дали представление об использовании языка во многих продвинутых областях, таких, как обработка данных, бессерверные вычисления, веб-разработки, машинное обучение и автоматизация сети.

Вопросы

1. Как в библиотеке Paramiko называется часто используемый класс для подключения к устройству?
2. Из каких четырех уровней состоит NETCONF?
3. Можно ли отправить конфигурацию непосредственно в активную базу данных в NETCONF?
4. Почему gNMI лучше подходит для сбора данных, чем для настройки сети?
5. Предоставляет ли RESTCONF те же возможности, что и NETCONF, но через интерфейсы REST?
6. Что такое издатель и потребитель в Apache Kafka?

Дополнительные ресурсы

- *«Python для сетевых инженеров»*, (*Mastering Python Networking*), автор: Эрик Чоу (Eric Chou).
- *«Practical Network Automation»*, второе издание, автор: Абхишек Ратан (Abhishek Ratan).
- *«Network Programmability and Automation»*, автор: Джейсон Эдельман (Jason Edelman).
- Официальная документация по библиотеке Paramiko: *http://docs.paramiko.org/*.
- Официальная документация по библиотеке Netmiko: *https://ktbyers.github.io/*.

- Официальная документация по библиотеке NAPALM: *https://napalm. readthedocs.io/.*
- Официальная документация по библиотеке ncclient: *https://ncclient. readthedocs.io/.*
- Модели YANG NETCONF: *https://github.com/YangModels/yang.*
- Официальная документация по API Nokia NSP: *https://network.developer. nokia.com/api-documentation/.*

Ответы

1. Класс `paramiko.SSHClient`.
2. Содержимое, Операции, Сообщения и Транспорт.
3. Если сетевое устройство не поддерживает базу-кандидат, тогда обновления конфигурации непосредственно в активную базу допустимы.
4. gNMI основан на gRPC, протоколе от Google для вызовов RPC между мобильными клиентами и облачными приложениями. Протокол оптимизирован для передачи данных, что делает его более эффективным для сбора данных с сетевых устройств, но не для конфигурирования.
5. RESTCONF предоставляет большинство функций NETCONF через интерфейсы REST, но не все.
6. Издатель — клиентская программа, которая отправляет сообщения в *тему* (*категорию*) Kafka в виде событий. Подписчик — это клиентское приложение, которое читает и обрабатывает сообщения из темы Kafka.

Предметный указатель

«

«Дзен Python» (The Zen of Python), 24
«клиент-сервер», 31
«Красный, Зеленый, Рефакторинг» (Red, Green, Refactor), 165
«модель-представление-контроллер» (Model View Controller, MVC), 305
«модель-представление-шаблон» (Model View Template, MVT), 305

A

Agile, 168
Amazon Web Services (AWS), 30
Apache Beam, 30
Apache Kafka, 415
Apache Spark, 242
Application Programming Interface, 321
ASGI (Asynchronous Service Gateway Interface), 339
Asyncio, 234
Atom, 42

B

Business to Business (B2B), 307

C

Chalice, 32
Counter, функция, 171
CRISP-DM (Cross-Industry Standard Process for Data Mining), 29

D

Data science, 43
DataFrame, 192
Development-Operations (DevOps), 168
DevOps (Development-Operations), 40
Docker, 347
Docstring, 32
Dunder (Double Under), 36

E

ETL, 287

F

filter, функция, 177
from, инструкция, 50
FTP, 31

G

Git, 39
GitHub, 40
Google App Engine (GAE), 43
Google Cloud Platform (GCP), 30
Google Container Registry, 351
gRPC, 396

H

Hadoop MapReduce, 242

I

IDE. См. интегрированная среда разработки
IDLE, 42
importlib, библиотека, 52
init, файл, 63

J

Java Virtual Machine (JVM), 337

JSON (JavaScript Object Notation), 139
JVM (Java Virtual Machine), 251

M

map, 175
Microsoft Azure, 30
ML. См. Машинное обучение
MVP. См. Минимально жизнеспособный продукт

N

NETCONF (Network Configuration), 394
NMS, 390
Node.js, 32

P

pandas, библиотека, 192
PCollection, 288
PEP 420: неявные пакеты пространства имен, 63
PEP 8, 35
Portable Operating System Interface (POSIX), 30
Prompt, 399
PTransform, 288
PyCharm, 42
PyDev, 43
PyPA, 62
PySpark, 242
Python, 16
Python Packaging Authority (PyPA), 62

R

RDD (Resilient Distributed Dataset), 247
reduce, функция, 177
REST API, 321
RESTCONF, 31
RESTful API, 321

S

SCP, 31
SDK (Software Development Kit), 272
Secure Shell (SSH), 31
Serverless, 32
Simple Storage Service (S3), 28
SMI (Structure of Management Information), 395
SNMP (Simple Network Management Protocol), 394
Spyder, 43
SSH (Secure Shell Protocol), 394
SSL, 307
str, 86
Sublime Text, 42
Subversion (SVN), 42

T

TCP, 31
Test PyPI, 75
TLS, 307

U

UAT-тестирование (User Acceptance Testing), 146
UDP, 31

V

Visual Studio Code, 43

W

WSGI (Web Service Gateway Interface), 339
WSGI-приложений, 32

Y

YAML (Yet Another Markup Language), 139

Z

Zappa, 32
Zero Touch Provisioning (ZTP), 393
zip, функция, 172

А

Абсолютный путь, 54
Абстрактные методы, 100
Абстрактный класс, 100
Абстракция, 100
Автоматизация сети (Network Automation), 390
Автоматизация тестирования, 143
Алгоритм Монте-Карло, 265
Альфа-тестирование, 146
Анонимная функция, 121
Ассоциативный массив, 114

Б

База управляющей информации (Management Information Base, MIB), 395
Базовый класс. См. родительский класс
Бессерверная функция, 355
Бессерверные вычисления (Serverless Computing), 354
AWS Lambda, 357
Google Cloud Functions, 357, 364
Azure Functions, 357
Бета-тестирование, 146
Блок управления задачей (Task Control Block, TCB), 221
Блок Управления Потоком (Thread Control Block, TCB), 208
Блок управления процессом, 221
Блок управления процессом (Process Control Block, PCB), 221
Блокировка конфигурации, 396
Блочная диаграмма, 378
Бэкенд-сервис (Backend Service), 357

В

Веб-приложение, 303
Веб-сервер, 306
Веб-сокет (WebSocket), 414
Веб-фреймворк, 304
Django, 304
Flask, 304
Вебхук (Webhook), 414
ВерблюжийРегистр (CamelCase), 37
Взаимоблокировка, 216
Включение (Comprehension), 190
Включение словарей (Dictionary comprehension), 191
Включение списков (List comprehension), 190
Включение множеств (Set comprehension), 192
Вложенная функция. См. Внутренняя функция
Вложенный класс, 87
Внутренний класс. См. Вложенный класс
Внутренняя функция, 179
Выборочная сортировка, 100
Выполняемое задание (Running Executor), 264

Г

Гвидо ван Россум, 23, 35
Генератор (Generator), 120
геттер, 91
Гиперпараметры (Hyperparameters), 376
Гистограмма, 378
Глобальная блокировка интерпретатора (Global Interpreter Lock, GIL), 209
Глубокое обучение, 372
Градиентный бустинг (Gradient Boost, GB), 373
Граф, 247
Графический пользовательский интерфейс (Graphical User Interface, GUI), 276
Графический процессор (Graphics Processing Units, GPU), 221

Д

Дандер (Dunder), 86
Декларативное программирование, 79
Декоратор, 84
Дерево решений, 373
Деревья классификации и регрессии, 371
Деструкторы класса, 83
Дзен Python, 24
Динамическая типизация. См. Утиная типизация
Документация Swagger, 307
Дочерний класс, 94

Ж

Жесткий код (Hardcode), 166
Средство логирования, 132

З

Завершенное задание (Finished Executor), 264
Заводские приемочные испытания (Factory Acceptance Testing, FAT), 146
Замóк (Lock), 215
Замыкание (Closure), 180
Запись преобразования (Write transform), 289
Защищенный метод (protected), 89
Зеленый поток (Green Thread), 233

И

Идентификатор процесса (Process ID, PID), 221
Импорт модулей, 46
Инженерный совет Интернета (Internet Engineering Task Force, IETF), 395
Инкапсуляция данных, 87
Инструменты оболочки, 30
Интегрированная среда разработки (Integrated Development Environment, IDE), 42
Интернет вещей (Internet of Things, IoT), 357
Интерпретатор, 52
Интерфейс командной строки (Command Line Interface, CLI), 71
Интерфейс сетевого взаимодействия (Network Management Interface, NMI), 396
Инфраструктура как услуга (Infrastructure-as-a-Service, IaaS), 30
Искатель (finder), 53
Исключение, 127
Искусственный интеллект (Artificial Intelligence, AI), 369
Исполнитель (Runner), 248
Итератор, 116
Итерация (Iteration), 116

К

Каскадная модель, 28
Каскадная таблица стилей (Cascading Style Sheets, CSS), 306
Класс, 80
- ◊ абстрактный, 100
- ◊ вложенный, 87

Кластер (Cluster), 242
Кластерные вычисления (Cluster computing), 243
Ключевй показатель эффективности (Key Performance Indicators, KPI), 378
Колбэк (Callback), 236
Комментарий, 32
Коммит (Commit), 40
Конвейер обработки данных, 287
Конвейеры Azure (Azure Pipelines), 273
Конкурентная программа, 112
Конструктор класса, 83
Контейнер, 347
Контейнер данных, 112
Контейнеризация (Containerization), 347
Контейнеры Docker, 30
Кооперативная многозадачность, 233
Кортеж (Tuple), 114
Корутин (Coroutine), 233
Критическая секция (Critical Section), 214
Кросс-валидация (Cross-validation), 375
Кумулятивная обработка (Cumulative Processing), 177

Л

Линейная регрессия, 371
Логгер, 132
Логирование (Logging), 131
Лямбда-функция (Lambda function), 178

М

Магические методы (Magic Methods), 36
Масштабатор, 384
Масштабирование данных, 376
Машина опорных векторов (Support Vector Machine, SVC), 380

Машинное обучение (Machine Learning), 28
Менеджер кластера (Cluster Manager), 247
Менеджер контекста, 122, 125
Метод, 84
Метод опорных векторов, 371
Метод удержания (Holdout), 379
Микросервис, 335
Микросервисная архитектура, 334
Минимально жизнеспособный продукт (Minimum Viable Product, MVP), 27
Многопоточность, 208
Множество (Set), 115
Модель «запрос-ответ», 310
модель машинного обучения, 369
Модель-кандидат, 376
Модификаторы доступа, 91
Модуль, 58
Модуль ABC (Abstract Base Classes), 100
Мьютекс (Mutex, механизм взаимного исключения), 209

Н

Наследование, 94
Непрерывная доставка (Continuous Delivery, CD), 167
Непрерывная интеграция (Continuous Integration , CI), 167
Нормализация данных, 376

О

Облачная IDE, 272
Облачная функция. См. Бессерверная функция
Облачные вычисления (Cloud Computing), 271
Обработка естественного языка (Natural Language Processing, NLP), 268
Объект класса, 80
Объектно-ориентированное программирование (ООП), 79
Объектно-реляционное связывание (Object Relational Mapper, ORM), 315
Одиночка (Singleton), 48
Ожидающий объект (Awaitable Object), 236
Окупаемость инвестиций (Return On Investment, ROI), 393
Операции CRUD (Create, Read, Update, Delete), 322
Операции извлечения, преобразования и загрузки (Extract, Transform, Load — ETL), 287
Операция действия (Action), 249
Операция преобразования (Transformation), 249
Оркестратор, 390
Откат (Rollback), 396
Очередь, 113

П

Пакет, 62
Парсер (Parser), 292
Парсинг (Parsing), 292
Первичный ключ (Primary key), 316
Перегрузка метода, 97
Переменная, 89
Переменная окружения (Runtime environment variables), 360
Переопределение метода, 97
Платформа как услуга (Platform-as-a-Service, PaaS), 274
Подкласс. См. Дочерний класс
Показатель F1-score, 30
Полиморфизм, 97
Полное доменное имя (Fully Qualified Domain Name, FQDN), 399
Полное имя объекта (Object Full Name, OFN), 412
Полное различимое имя (Fully Distinguishable Name, FDN). См. Полное имя объекта
Пользовательский интерфейс (User Interface, UI), 304
Поставщик облачных сервисов (Cloud Service Provider, CSP), 31
Поток, 208
Поток-демон (Daemon Thread), 212
Правило двух пицц, 338
Предложения по улучшению Python (Python Enhancement Proposal, PEP) 257, 32

Предметно-ориентированное проектирование (Domain-Driven Design, DDD), 337
Преобразование (Transform), 288
Примитив (Primitive), 221
Программный интерфейс приложения (Application Programming Interface, API), 100
Производный класс. См. Дочерний класс
Производственная среда (Production environment, 138
Протокол SSL (Secure Sockets Layer), 307
Протокол TLS (Transport Layer Security), 307
Протокол WSGI (Web Server Gateway Interface), 309
Протокольный буфер (Protocol buffers, Protobuf), 396

P

Рабочий узел (Worker node), 248
Разработка через тестирование (Test-driven development, TDD), 165
Распределенная файловая система Hadoop (Hadoop distributed file system, HDFS), 28
Распределенный набор данных, 242
Регрессионное тестирование, 146
Регулярное выражение (Regular Expression, regex), 31
Редактор Cloud Shell (Cloud Shell Editor), 285
Родительский класс, 94
Ромбовидное наследование (diamond inheritance), 96

C

Сборщик мусора, 83
Северный интерфейс (North Bound Interface, NBI), 409
Семафор (Semaphore), 215
Серверный процесс, 226
Сетевое программирование, 31
Сеттер, 91
Сеть как услуга (Network-as-a-service, NaaS), 31
Система «издатель-подписчик» (Publish-Subscribe system, Pub/Sub system), 356
Система контроля версий, 39
Система управления базами данных (СУБД), 307
Система управления сетью (Network Management System, NMS), 390
Системное программирование, 30
Сквозное тестирование (End-to-End, E2E), 145
Словарь (Dictionary), 114
Событийно-ориентированная система (Event-driven system), 390
Соглашение о конфигурации (Convention over configuration), 308
Соглашение об именовании, 60, См. PEP 8
Сопрограмма, 234
Состояние гонки, 214, 230
Специальный метод, 86
Список (List), 113
Среда выполнения (Runtime environment), 274
Среда разработки (Development environment), 138
Среда-оболочка (Shell environment), 273
Стек, 113
Строка (String), 112
Суперкласс. См. Родительский класс
Сырые данные, 370

T

Тасклет (Tasklet), 233
Тег-идентификатор (ID), 162
Тестирование
Тестирование методом белого ящика (White-box Testing), 145
Тестирование методом черного ящика (Black-box Testing), 145
Тестирование на основе данных (Data-driven Testing, DDT), 161
Тест-кейс (test case), 146
Тестовый исполнитель, 148
Тестовый набор, 148
Тестовые фикстуры (Test Fixture), 147
Тонкая настройка гиперпараметров, 382

У

Удаленный вызов методов (Remote Method Invocation, RMI), 321
Удаленный вызов процедур (Remote Procedure Call, RPC), 247, 321
Устойчивая функция (Durable Functions), 357
Устойчивый распределенный набор данных (Resilient Distributed Datasets, RDD), 249
Утиная типизация, 104

Ф

Фабрика замыканий (Closure factory), 181
Фикстура, 147, 162
Фиктивное тестирование (Mock Testing), 145
Функция полезности, 84
Функция, 84
Функция как услуга (Function as a Service, FaaS), 354
Фьючерс (Future), 236

Х

Хеш-таблица, 114

Ц

Центральный процессор (ЦП), 208
Цикл событий (Event Loop), 233

Ч

Частный метод (Private method), 89
Чтение преобразования (Read transform), 288

Ш

Шаблонизатор, 306
Шард (Shard), 292

Э

Экземпляр класса, 80
Эксплуатационное приемочное тестирование (Operational Acceptance Testing, OAT), 146

Я

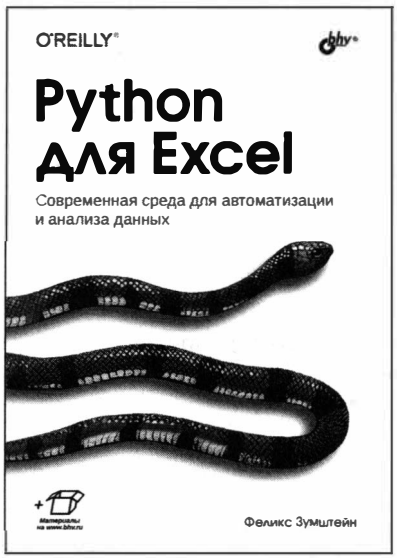
Язык гипертекстовой разметки (HyperText Markup Language, HTML), 306
Ящик с усами, 378

www.bhv.ru

Зумштейн Ф.

Python для Excel

Отдел оптовых поставок:
e-mail: opt@bhv.ru



- Освойте работу с современными инструментами, включая блокноты Jupyter и Visual Studio Code
- Используйте pandas для сбора, очистки и анализа данных и замены типичных вычислений в Excel
- Автоматизируйте рутинные задачи, такие, как объединение рабочих книг Excel и создание отчетов Excel
- Используйте xlwings для создания интерактивных инструментов Excel, использующих Python в качестве механизма вычислений
- Подключайте Excel к базам данных и файлам CSV и получайте данные из Интернета с помощью кода Python
- Используйте Python как единый инструмент для замены VBA, Power Query и Power Pivot

Книга посвящена автоматизации Excel с помощью языка программирования Python. Описаны дистрибутив Anaconda Python и современные средства разработки, такие, как менеджеры пакетов Conda и pip, блокноты Jupyter и Visual Studio Code. Даны необходимые основы языка Python и введение в анализ данных с помощью библиотеки pandas. Приведены приемы чтения и записи файлов Excel без Excel. Рассмотрено программирование приложений Excel с помощью популярного пакета с открытым исходным кодом xlwings: автоматизация Excel, инструменты на основе технологии Python, трекер пакетов Python, а также функции, определяемые пользователем.

Электронный архив на сайте издательства содержит цветные иллюстрации к книге.

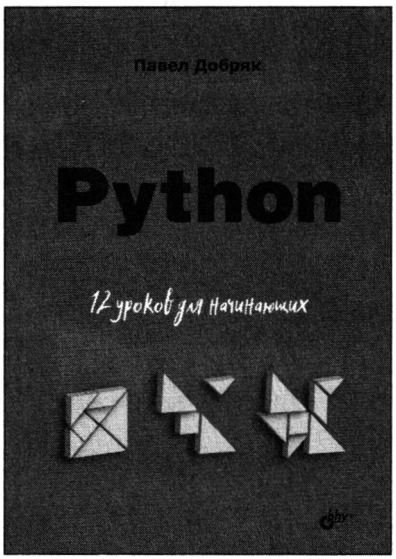
Феликс Зумштейн, создатель xlwings, популярного пакета с открытым исходным кодом, который позволяет автоматизировать Excel с помощью Python в Windows и macOS. Как генеральный директор компании xltrail и менеджер одноименной системы контроля версий для файлов Excel, он имеет глубокое представление о типичных случаях использования и проблемах с Excel в различных областях.

www.bhv.ru

Добряк П.

Python. 12 уроков для начинающих

Отдел оптовых поставок:
e-mail: opt@bhv.ru



- Изучите основные конструкции языка Pyhon
- Освойте программирование в ходе 12 наглядных уроков
- Научитесь понимать и писать код в различных стилях программирования
- Прокачайте алгоритмическое мышление

В 12 уроках показаны основы программирования и базовые конструкции языка Python. Изложены принципы различных стилей программирования. Даны понятия ввода-вывода, переменных, условий, потока чисел, циклов и списков, массивов, функций и рекурсий. Рассмотрены особенности структурного, объектно-ориентированного и функционального программирования. В каждой главе предложены практические задачи и дано их пошаговое решение с подробным описанием алгоритма.

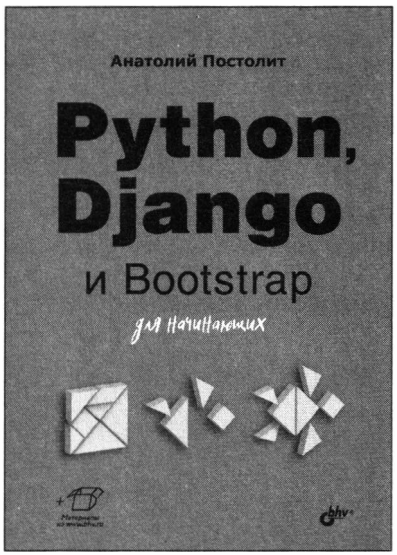
Добряк Павел Вадимович, кандидат технических наук, преподаватель Уральского федерального университета. Проводит занятия по различным языкам программирования, базам данных, искусственному интеллекту и проектированию информационных систем. Репетитор по математике и информатике.

www.bhv.ru

Постолит А.

Python, Django и Bootstrap для начинающих

Отдел оптовых поставок:
e-mail: opt@bhv.ru



- Веб-технологии
- Инструментальные средства для разработки веб-приложений
- Знакомство с фреймворком Django
- Знакомство с фреймворком Bootstrap
- Интерактивная среда разработки PyCharm
- Обработка и маршрутизация запросов
- Шаблоны веб-страниц
- Формы и модели данных
- Веб-сайт и веб-интерфейс для пользователей
- Встроенная панель для администрирования сайта
- Пользовательские формы
- Публикация сайта в Интернете

Книга посвящена вопросам разработки веб-приложений с использованием языка Python, фреймворков Django, Bootstrap и интерактивной среды разработки PyCharm. Рассмотрены основные технологии и рабочие инструменты создания веб-приложений. Описаны фреймворки Django, Bootsrtap и структура создаваемых веб-приложений. На простых примерах показана обработка и маршрутизация запросов пользователей, формирование ответных веб-страниц. Рассмотрено создание шаблонов веб-страниц и форм для пользователей. Показано взаимодействие пользователей с различными типами баз данных через модели. Описана работа с базами данных через встроенные в Django классы без использования SQL-запросов. Приведен пошаговый пример создания сайта — от его проектирования до формирования программных модулей и развертывания сайта в Интернете с базами данных SQLite и MySQL. Электронный архив на сайте издательства содержит коды всех примеров.

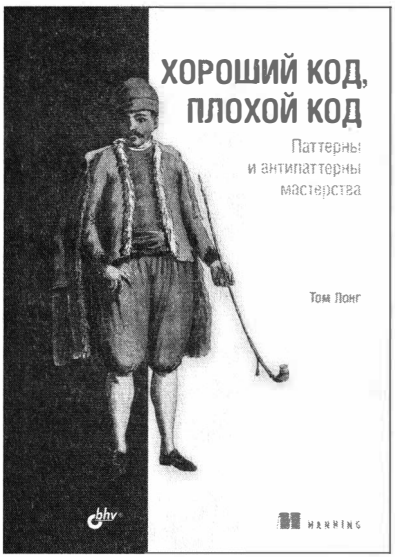
Постолит Анатолий Владимирович, доктор технических наук, профессор, академик Российской академии транспорта, лауреат Всероссийского конкурса «Инженер года». Профессиональный программист, автор книг компьютерной тематики, в том числе «Основы искусственного интеллекта в примерах на Python», «Python, Django и PyCharm для начинающих» и более 100 научных публикаций. Преподавал в Московском государственном автомобильно-дорожном техническом университете (МАДИ). Занимался разработкой и внедрением информационных систем для транспортного комплекса Москвы и Московской области, для транспортного обслуживания зимних Олимпийских игр в г. Сочи, систем оплаты проезда и информирования пассажиров городского общественного транспорта. Специализируется на создании информационных систем на основе MS SQL Server, MS Visual Studio, Bluetooth-технологий, а также инновационных проектов с использованием Python, Django, Bootstrap, PyCharm и различных специализированных библиотек.

www.bhv.ru

Отдел оптовых поставок:
e-mail: opt@bhv.ru

Лонг Т.

Хороший код, плохой код



В частности, здесь рассказано:

- Как писать код, который сложно использовать ненадлежащим образом
- Как правильно организовать обработку ошибок в приложениях
- Как обеспечить модульность кода, оптимизировав его для повторного использования
- Как при необходимости переводить примеры кода с Java на C#
- Как правильно организовать тестирование кода и внедрение зависимостей

В книге изложены приемы и методы, обязательные при грамотном объектно-ориентированном программировании. Следование приведенным паттернам проектирования защитит программу от ошибок, проблем с совместимостью, неожиданных и непонятных отказов, а также обеспечит модульность кода. Рассмотрены примеры на языке Java, легко адаптируемые для C#, пояснены приемы тестирования, развития и адаптации кода с учетом меняющихся требований.

Книга адресована программистам на языках Java и C# и руководителям команд, практикующих объектно-ориентированное программирование.

Том Лонг, программист корпорации Google, техлид. Наряду с решением производственных задач постоянно обучает начинающих программистов секретам профессионального мастерства.

bhv®

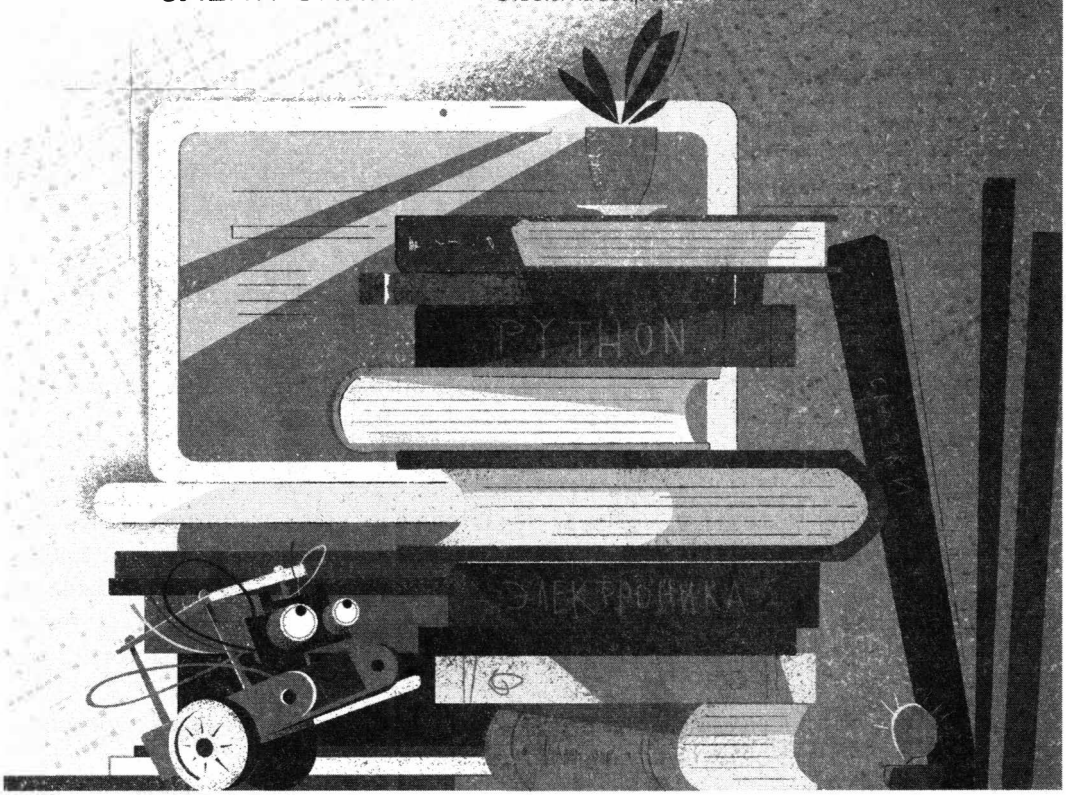
ИНТЕРНЕТ-МАГАЗИН

BHV.RU

КНИГИ, РОБОТЫ,
ЭЛЕКТРОНИКА

Интернет-магазин издательства «БХВ»

- Более 30 лет на российском рынке
- Книги и наборы по электронике и робототехнике по издательским ценам
- Электронные архивы книг и компакт-дисков
- Ответы на вопросы читателей



Интернет-магазин БХВ-Электроника

Скоро открытие!



Гик — человек, глубоко погруженный в мир компьютерных технологий, стремящийся досконально разобраться в наиболее важных мелочах и нюансах. Эта книга написана для гиков, увлеченных программированием на Python.

Книга раскрывает методы оптимального использования Python как с точки зрения проектирования, так и реализации практических задач. В ней подробно описан жизненный цикл крупномасштабного проекта на Python, показаны различные способы создания его модульной архитектуры. Вы изучите лучшие практики и паттерны проектирования, узнаете, как масштабировать приложения на Python, как реализовать многопроцессорность и многопоточность. Вы поймете, как можно использовать Python не только для развертывания на одной машине, но также в частных и публичных облачных средах. Вы изучите методы обработки данных, сосредоточитесь на создании микросервисов и научитесь использовать Python для автоматизации сетей и машинного обучения. Наконец, вы узнаете, как применять описанные методы и практики в веб-разработке.

Вы изучите:

- **Принципы разработки и управления сложными проектами**
- **Способы автоматизации тестирования приложений и разработки через тестирование (TDD)**
- **Многопоточность и многопроцессорность в Python**
- **Написание приложений с использованием кластера Apache Spark для обработки больших данных**
- **Разработку и развертывание бессерверных программ в облаке на примере Google Cloud Platform (GCP)**
- **Создание на Python веб-приложений и REST API, использование среды Flask**
- **Использование Python для извлечения данных с сетевых устройств и систем управления сетью (NMS)**
- **Применение Python для анализа данных и машинного обучения**

Мухаммад Азиф — программный архитектор, обладающий обширным опытом в области веб-разработки, автоматизации сетей и облаков, виртуализации и машинного обучения. Возглавлял многие крупномасштабные проекты в различных коммерческих компаниях. В 2012 году получил степень доктора философии в области компьютерных систем в Карлтонском университете (Оттава, Канада) и в настоящее время работает в компании Nokia в качестве ведущего специалиста.

ISBN 978-5-9775-0956-5

9 785977 509565

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®