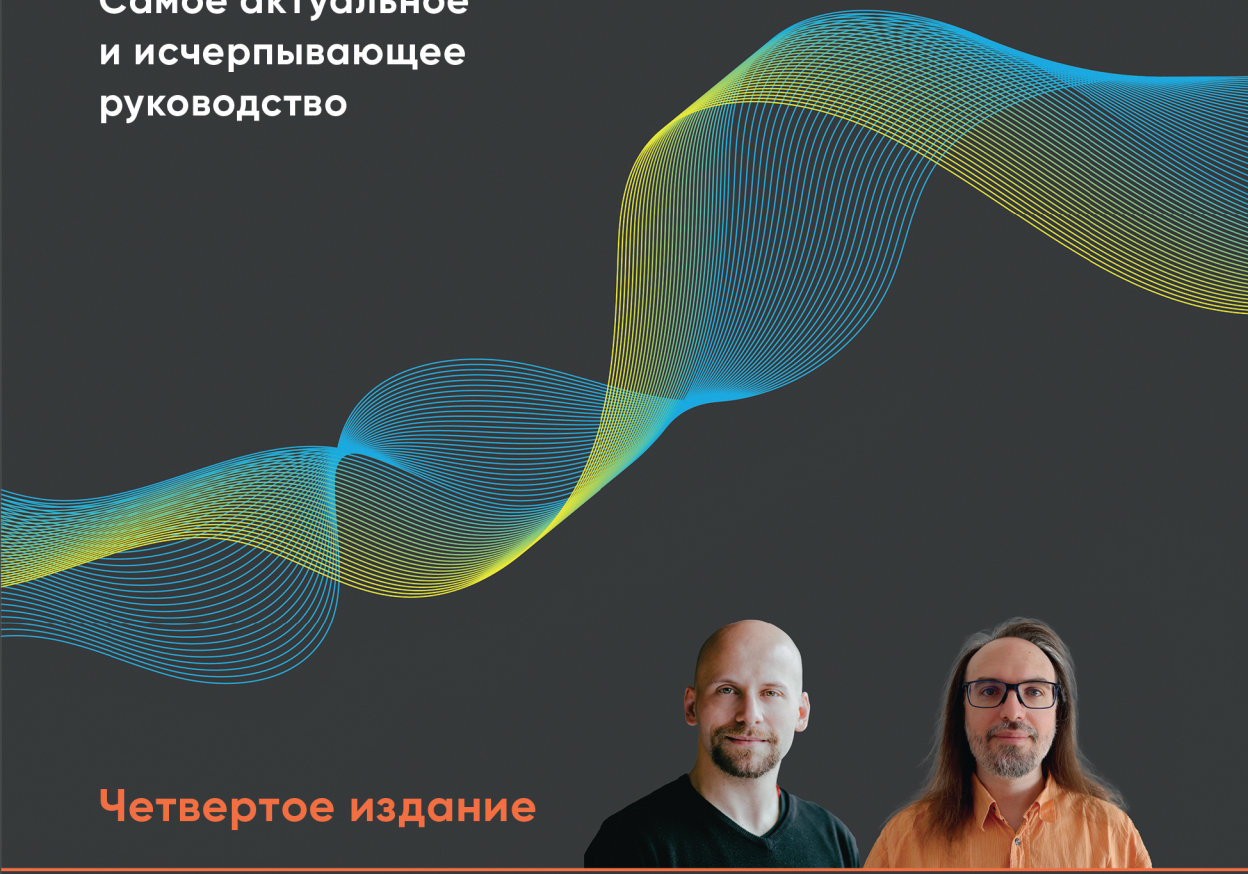
EXPERT INSIGHT

python™

Весь Python

Самое актуальное и исчерпывающее руководство



Четвертое издание

Фабрицио Романо
Генрих Крюгер

SPRINT book

<packt>

Learn Python Programming

Fourth Edition

A Comprehensive, Up-to-Date,
and Definitive Guide to Learning Python

Fabrizio Romano
Heinrich Kruger

Весь Python

Четвертое издание

Самое актуальное
и исчерпывающее руководство

Фабрицио Романо
Генрих Крюгер

SPRINT book 2026

Фабрицио Романо, Генрих Крюгер

Весь Python. Самое актуальное и исчерпывающее руководство

Перевел с английского С. Черников

ББК 32.973.2-018.1я7
УДК 004.43(07)

Романо Фабрицио, Крюгер Генрих

Р69 Весь Python. Самое актуальное и исчерпывающее руководство. — Астана: «Спринт Бук», 2026. — 512 с.: ил.

ISBN 978-601-12-4594-4

Всеобъемлющее современное руководство по программированию на Python, охватывающее фундаментальные идеи и практические приемы!

Вы научитесь писать собственные программы и получите четкое представление о том, куда двигаться дальше и как использовать полученные знания. Изучение Python подкреплено практикой — огромным количеством примеров приложений, к концу книги вы будете готовы применить полученные знания и создать несколько реальных проектов. Вы научитесь эффективно использовать Python в анализе данных, веб-разработке и автоматизации задач.

Книга включает описание новейших возможностей, появившихся в версиях Python 3.9–3.12, в том числе главы об аннотациях типов и консольных приложениях, а также примеры, демонстрирующие современные практики веб-разработки на Python.

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

ISBN 978-601-12-4594-4

Изготовлено в России. Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес: 010000, Казахстан, город Астана, район Алматы, проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 12.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 12.11.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 1000. Заказ 0000.

Краткое содержание

Об авторах 16
О научных редакторах 17
Предисловие 18

Глава 1. Плавное погружение в Python 23
Глава 2. Встроенные типы данных 57
Глава 3. Условные конструкции и циклы 100
Глава 4. Функции — строительные блоки кода 127
Глава 5. Генераторы и включения 164
Глава 6. ООП, декораторы и итераторы 197
Глава 7. Исключения и контекстные менеджеры 236
Глава 8. Работа с файлами и хранение данных 258
Глава 9. Криптография и токены 296
Глава 10. Тестирование 314
Глава 11. Отладка и профилирование 343
Глава 12. Аннотации типов 362
Глава 13. Введение в Data Science 387
Глава 14. Введение в разработку API 417
Глава 15. Консольные приложения (CLI) 451
Глава 16. Упаковка и публикация приложений 469
Глава 17. Задачи по программированию 494

Оглавление

Об авторах ... 16

О научных редакторах ... 17

Предисловие ... 18
- Для кого эта книга ... 19
- Структура издания ... 20
- Как получить максимум пользы от книги ... 21
  - Цветные иллюстрации ... 21
  - Исходный код примеров ... 21
  - Условные обозначения ... 21
- От издательства ... 22
  - О научных редакторах русскоязычного издания ... 22

Глава 1. Плавное погружение в Python ... 23
- Основы программирования ... 24
- Добро пожаловать в мир Python ... 26
- Коротко о Python ... 26
  - Переносимость ... 26
  - Логичность ... 27
  - Скорость разработки ... 27
  - Обширная библиотека ... 27
  - Качество программ ... 27
  - Интеграция с другим ПО ... 28
  - Использование в Data Science ... 28
  - Удовольствие от работы ... 28
- Недостатки Python ... 28
- Как Python используют сегодня ... 29
- Настройка окружения ... 31
  - Установка Python ... 31
  - Консоль Python ... 34

Несколько слов о виртуальных окружениях ... 35
Установка сторонних библиотек ... 39
Консоль ... 40
Как запустить программу на Python ... 40
Запуск скриптов на Python ... 40
Запуск интерактивной оболочки Python ... 41
Запуск Python в качестве сервиса ... 42
Запуск Python в качестве графического приложения ... 42
Структура кода Python ... 43
Как использовать модули и пакеты ... 45
Модель выполнения кода в Python ... 47
Имена и пространства имен ... 47
Области видимости ... 49
Рекомендации по написанию хорошего кода ... 52
Культура Python ... 53
Несколько слов об IDE ... 55
Несколько слов об ИИ ... 55
Резюме ... 56

Глава 2. Встроенные типы данных ... 57
В Python всё — объекты ... 57
Изменяемость ... 58
Числа ... 60
Целые числа (int) ... 60
Логический тип (bool) ... 62
Вещественные числа (float) ... 64
Комплексные числа ... 65
Дроби и десятичные числа ... 66
Неизменяемые последовательности ... 67
Строки и байты ... 67
Кортежи ... 72
Изменяемые последовательности ... 73
Списки ... 73
Массивы байтов (bytearrays) ... 77
Множества ... 78
Отображения: словари ... 80
Типы данных ... 84
Дата и время ... 84
Модуль collections ... 89
Перечисления ... 93

Заключительные замечания ... 94
Кеширование малых значений ... 95
Как выбрать подходящую структуру данных ... 95
Об индексации и срезах ... 97
Об именах ... 98
Резюме ... 99

Глава 3. Условные конструкции и циклы ... 100
Условное программирование ... 100
Оператор if ... 101
Особый случай else: elif ... 101
Вложенные операторы if ... 103
Тернарный оператор ... 104
Сопоставление с шаблоном ... 105
Циклы ... 106
Цикл for ... 106
Итераторы и итерируемые объекты ... 109
Перебор нескольких последовательностей ... 109
Цикл while ... 111
Операторы break и continue ... 113
Особый блок else после цикла ... 115
Выражения присваивания ... 117
Операторы и выражения ... 117
Моржовый оператор (:=) ... 118
Предупреждение ... 119
Примеры программ ... 119
Генератор простых чисел ... 120
Пример со скидками ... 122
Модуль itertools ... 124
Бесконечные итераторы ... 124
Итераторы с остановкой по короткой последовательности ... 124
Комбинаторные генераторы ... 125
Резюме ... 126

Глава 4. Функции — строительные блоки кода ... 127
Зачем нужны функции ... 128
Уменьшение дублирования кода ... 129
Разбиение сложной задачи ... 129
Сокрытие деталей реализации ... 130
Улучшение читабельности ... 131
Улучшение отслеживаемости ... 132

Области видимости и разрешение имен....132
Операторы global и nonlocal....134
Входные параметры....136
Передача аргументов....136
Присваивание имени параметра....137
Изменение изменяемого объекта....138
Передача аргументов....139
Определение параметров....142
Возврат значений....151
Возврат нескольких значений....152
Несколько полезных советов....153
Рекурсивные функции....154
Анонимные функции....155
Атрибуты функций....156
Встроенные функции....158
Документирование кода....158
Импорт объектов....159
Относительный импорт....161
Заключительный пример....162
Резюме....163

Глава 5. Генераторы и включения....164
Функции map(), zip() и filter()....166
map()....166
zip()....169
filter()....170
Включения....171
Вложенные включения....172
Фильтрация во включении....173
Словарные включения....175
Включения множеств....176
Генераторы....177
Генераторные функции....177
Другие инструменты, помимо функции next()....180
Выражение yield from....182
Генераторные выражения....183
Несколько слов о производительности....185
Не увлекайтесь включениями и генераторами....188
Локализация имен....192
Поведение встроенных функций, напоминающее работу генераторов....193
Еще один пример напоследок....194
Резюме....196

Глава 6. ООП, декораторы и итераторы 197
Декораторы 197
Фабрика декораторов 203
Объектно-ориентированное программирование 205
Наипростейший класс в Python 205
Пространства имен классов и объектов 206
Затенение атрибутов 207
Аргумент self 208
Инициализация экземпляра 209
ООП подразумевает повторное использование кода 210
Обращение к базовому классу 215
Множественное наследование 217
Статические методы и методы класса 221
Приватные методы и искажение имен 225
Декоратор property 228
Декоратор cached_property 229
Перегрузка операторов 231
Полиморфизм (краткий обзор) 232
Классы данных 233
Создание собственного итератора 234
Резюме 235

Глава 7. Исключения и контекстные менеджеры 236
Исключения 236
Генерация исключений 238
Определение собственных исключений 239
Трассировки 239
Обработка исключений 240
Группы исключений 245
Исключения как инструмент управления потоком выполнения 249
Контекстные менеджеры 250
Контекстные менеджеры на базе классов 253
Контекстные менеджеры на базе генераторов 255
Резюме 257

Глава 8. Работа с файлами и хранение данных 258
Управление файлами и каталогами 258
Открытие файлов 259
Чтение из файла и запись в файл 261
Проверка существования файлов и каталогов 263

Управление файлами и каталогами 264
Временные файлы и каталоги 267
Содержимое каталогов 267
Сжатие файлов и каталогов 268
Форматы обмена данными 269
Работа с JSON 270
Ввод/вывод, потоки и запросы 277
Управление объектами в памяти 277
Выполнение HTTP-запросов 278
Сохранение данных на диске 281
Сериализация данных с помощью модуля pickle 281
Сохранение данных с помощью модуля shelve 283
Сохранение данных в базу данных 284
Файлы конфигурации 291
Популярные форматы 291
Резюме 295

Глава 9. Криптография и токены 296
Зачем нужна криптография 296
Полезные рекомендации 297
Модуль hashlib 297
Алгоритм HMAC 301
Модуль secrets 302
Случайные объекты 302
Генерация токенов 302
Сравнение дайджестов 304
Токены JWT 305
Зарегистрированные утверждения 308
Применение асимметричных (публичных) алгоритмов 311
Полезные источники 313
Резюме 313

Глава 10. Тестирование 314
Тестирование приложения 315
Структура теста 317
Рекомендации по написанию тестов 318
Модульное тестирование 320
Тестирование генератора CSV 322
Разработка через тестирование 339
Резюме 342

Глава 11. Отладка и профилирование 343
Приемы отладки 344
Отладка с помощью функции print() 344
Отладка с помощью специальной функции 345
Отладка с помощью отладчика Python 347
Анализ логов 349
Другие приемы 353
Поиск справочной информации 354
Рекомендации по устранению неполадок 354
Где искать 354
Использование тестов для отладки 355
Мониторинг 355
Профилирование кода на Python 356
Когда стоит профилировать 359
Измерение времени выполнения 360
Резюме 361

Глава 12. Аннотации типов 362
Система типов в Python 362
Утиная типизация 363
История появления аннотаций типов 364
Преимущества аннотаций типов 366
Использование аннотаций типов 366
Аннотирование функций 367
Тип Any 368
Псевдонимы типов 368
Специальные формы 369
Обобщенные типы 370
Аннотирование переменных 371
Аннотирование контейнеров 372
Аннотирование кортежей 373
Абстрактные базовые классы 375
Особые формы типизации 378
Аннотация переменных параметров 379
Протоколы 380
Статический анализатор типов mypy 383
Полезные ресурсы 386
Резюме 386

Глава 13. Введение в Data Science 387
IPython и Jupyter Notebook 388
Использование Anaconda 390
Запуск Jupyter Notebook 390
Работа с данными 391
Подготовка блокнота 392
Подготовка данных 392
Очистка данных 396
Создание DataFrame 397
Вычисление метрик 403
Сохранение DataFrame в файл 407
Визуализация результатов 407
Что дальше 415
Резюме 416

Глава 14. Введение в разработку API 417
Протокол передачи гипертекста (HTTP) 417
Как работает HTTP 418
Коды состояния 419
Знакомство с API 420
Что такое API 420
Зачем нужны API 421
Протоколы API 422
Форматы обмена данными в API 422
API для железнодорожной системы 423
Проектирование базы данных 424
Основная настройка и конфигурация 430
Конечные точки Station 431
Аутентификация пользователя 445
Документирование API 448
Что дальше 449
Резюме 450

Глава 15. Консольные приложения (CLI) 451
Аргументы командной строки 452
Позиционные аргументы 452
Параметры 452
Подкоманды 453
Анализ аргументов 454

Создание CLI-клиента для работы с API железнодорожной системы 456
Взаимодействие с API железнодорожной системы 458
Создание интерфейса командной строки 458
Конфигурационные файлы и секреты 460
Создание подкоманд 463
Реализация подкоманд 466
Дополнительные ресурсы и инструменты 467
Резюме 468

Глава 16. Упаковка и публикация приложений 469
Python Package Index 469
Упаковка с помощью Setuptools 471
Структура проекта 472
Метаданные пакета 476
Доступ к метаданным в коде 483
Создание и публикация пакетов 485
Сборка 485
Публикация 487
Советы по запуску новых проектов 489
Другие файлы 490
Альтернативные инструменты 490
Дополнительные материалы 492
Резюме 492

Глава 17. Задачи по программированию 494
Проект Advent of Code 495
Camel Cards 496
Cosmic Expansion 501
Заключение 508
Сайты с задачами по программированию 509
Резюме 510

Моим дорогим Маргерите и Грациано. Спасибо вам за любовь, помощь и советы, которые вы дарили мне на протяжении последних двадцати лет.

Фабрицио Романо

Моей жене Деби — без ее любви, поддержки и бесконечного терпения я бы не смог это осуществить.

Генрих Крюгер

Об авторах

Фабрицио Романо родился в Италии в 1975 году. Получил степень магистра в области информатики и вычислительной техники в Падуанском университете. С 1999 года профессионально занимается разработкой программного обеспечения. С 2016-го работает в компании Sohonet, где в настоящее время занимает должность менеджера по развитию бизнеса. В 2020 году Академия телевидения вручила ему и его команде премию «Эмми» за инженерные достижения — за вклад в развитие технологий удаленной работы.

Хочу поблагодарить всех сотрудников Packt, особенно Амишу Ватаре за ее профессионализм и доброту. Спасибо Стефано и Хавьеру, рецензентам, за их полезные замечания и, конечно, Генриху. Моя глубочайшая благодарность — жене Элизе. Спасибо за твою любовь и поддержку.

Генрих Крюгер родился в Южной Африке в 1981 году. Получил степень магистра в области информатики в Утрехтском университете (Нидерланды). С 2014 года профессионально занимается разработкой программного обеспечения. С 2017-го работает вместе с Фабрицио в команде, отвечающей за развитие продуктов в Sohonet. В 2020 году он и его коллеги были награждены премией «Эмми» за инженерные достижения в области развития технологий удаленной работы.

Хочу поблагодарить Фабрицио за приглашение поработать над этой книгой. Это был замечательный опыт — работать с тобой, друг мой. Благодарю также Стефано Чинеллато и Хавьера Наварро за их полезные комментарии, а всех сотрудников Packt — за помощь. И самое главное, я благодарю свою жену Деби за ее любовь, поддержку и веру в меня.

О научных редакторах

Хавьер Муньос родился в Вальядолиде (Испания) и с раннего возраста увлекся компьютерами. Получил диплом инженера в области телекоммуникаций со специализацией в телематике, а также степень магистра в области Big Data Science в Вальядолидском университете, с отличием окончив обучение.

После окончания учебы Хавьер переехал в Дублин (Ирландия), где работал инженером-программистом и аналитиком данных. Его карьерный путь продолжился в компании Optum — технологическом гиганте в сфере здравоохранения из списка Fortune 5. В Optum Хавьер сосредоточился на разработке ИИ-решений: создавал и внедрял модели искусственного интеллекта для поддержки медицинской диагностики и лечения, сотрудничая с врачами и соблюдая требования таких стандартов, как HIPAA и GDPR.

Параллельно с работой в Optum Хавьер основал стартап Saoi Tech Solutions, занимавшийся созданием масштабируемых решений на базе машинного обучения для международных компаний. Среди его проектов — исследования в области распознавания лиц и практическая реализация ИИ-моделей с использованием современных облачных и контейнерных технологий.

В настоящее время Хавьер работает в Великобритании программистом в компании Sohonet, где разрабатывает Python-приложения на базе Django для кино- и телепроизводства. Его задача — создавать инструменты, упрощающие совместную работу и улучшающие производственные процессы в индустрии развлечений.

Я искренне благодарю своих друзей и коллег — Фабрицио и Генриха, авторов этой книги, — за поддержку, сотрудничество и предоставленную возможность выступить в роли научного редактора. Я также признателен всей команде Sohonet и бывшим коллегам и наставникам, поддержка которых была для меня неоценима на всем профессиональном пути. Этот опыт стал полезным и вдохновляющим одновременно, и я очень благодарен всем, кто был его частью.

Стефано Кинеллато — инженер-программист из Италии. Получил степень магистра в области инженерии компьютерных наук в Падуанском университете. Более 15 лет занимается разработкой программного обеспечения. Специализируется на веб-приложениях и особенно интересуется алгоритмами и статистикой.

Предисловие

С радостью представляем вам четвертое издание нашей книги. Первое вышло в 2015 году, и с тех пор эта книга стала одним из мировых бестселлеров по Python.

Десять лет назад программирование означало работу с определенными инструментами и следование устоявшимся парадигмам. Сегодня все иначе: разработчики становятся все более узкоспециализированными, а внимание все чаще сосредоточено на API и распределенных приложениях. Мы постарались уловить актуальные тенденции и предоставить базовые знания, основанные на нашем опыте в стремительно развивающейся отрасли.

При каждом переиздании в книгу вносились изменения: устаревшие главы удалялись, появлялись новые, а некоторые перерабатывались так, чтобы отражать современные подходы к разработке программного обеспечения.

В четвертом издании добавлены три новые главы. Первая посвящена аннотациям типов. Их появление в Python не новость, однако мы убеждены, что они уже стали неотъемлемой частью языка и без их описания книга была бы неполной.

Во второй новой главе рассказывается о создании CLI-приложений, она заменяет собой старую главу о графических интерфейсах. В настоящее время львиная доля взаимодействия с программами происходит через браузер, а многие классические настольные приложения создаются с использованием веб-компонентов. Поэтому мы посчитали, что глава, посвященная GUI, стала неактуальной.

Наконец, в третьей новой главе рассматривается тема соревновательного программирования.

Остальные главы были обновлены с учетом последних изменений в языке, а также переработаны, чтобы изложение стало еще более простым и плавным, но при этом оставалось насыщенным и полезным для читателя.

Душа книги осталась неизменной. Это не «еще одна книга по Python». Прежде всего она посвящена программированию. Мы старались дать как можно больше информации. А в тех случаях, когда ограниченный объем книги не позволял раскрыть тему полностью, мы указывали, где искать дополнительную информацию, чтобы вы могли расширять ваши знания.

Эта книга написана на совесть. В ней мы постарались объяснить концепции и идеи так, чтобы они оставались актуальными как можно дольше. В нее вложено много труда, размышлений и обсуждений — все ради того, чтобы она выдержала проверку временем.

При этом четвертое издание кардинально отличается от первого. Книга стала гораздо более продуманной, профессиональной и теперь больше сосредоточена на самом языке, а чуть меньше — на проектах. Мы считаем, что нам удалось найти правильный баланс между ними.

Работа с книгой потребует сосредоточенности и усилий. Весь код доступен для скачивания. При желании вы можете клонировать репозиторий с GitHub. Обязательно загляните туда. Это поможет вам лучше усвоить материал. Ведь код — не что-то статичное. Он динамичен: меняется, развивается. Вы усвоите гораздо больше, если будете не просто читать, а пробовать, экспериментировать, менять и даже ломать. Мы оставили в книге немало подсказок, которые помогут вам в этом.

Напоследок хочу выразить особую благодарность моему соавтору Генриху. В каждом разделе чувствуются его талант, креативность и глубина знаний. У него феноменальная память: он замечает повторы в главах, разделенных сотнями страниц. Я так не умею.

Как и я, Генрих провел множество ночей и выходных, работая над тем, чтобы подача материала была максимально качественной. Именно благодаря тому, что мы прошли этот путь вместе, я абсолютно уверен: у нас получилось достойное издание.

Наш совет — вдумчиво читайте, изучайте и обязательно экспериментируйте с исходным кодом. А когда почувствуете уверенность в своих знаниях Python, не останавливайтесь. Выходите за его пределы. Хороший разработчик должен понимать принципы и обладать навыками, которые не сводятся к какому-то одному языку программирования. Изучение других языков поможет вам различать: что относится именно к Python, а что является общими концепциями программирования. Надеемся, эта книга поможет двигаться в этом направлении.

Наслаждайтесь путешествием. И пожалуйста, делитесь тем, чему научились.

Фабрицио

Для кого эта книга

Книга рассчитана на читателей, уже обладающих неким опытом программирования, пусть даже не на Python. Базовые познания в области создания кода не помешают, хотя это и не строгое требование.

Даже если вы уже знакомы с Python, книга все равно может вам пригодиться — как справочник по основам языка и как источник практических советов и наблюдений, основанных на более чем сорокалетнем совокупном опыте ее авторов.

Структура издания

В главе 1 вы познакомитесь с основами программирования и базовыми конструкциями языка Python, а также узнаете, как установить и запустить его на вашем компьютере.

В главе 2 вы узнаете, что Python обладает богатым набором собственных типов данных, и найдете описание и примеры использования каждого из них.

В главе 3 рассказывается о том, как управлять потоком выполнения программы с помощью условий, логики и циклических конструкций.

Глава 4 посвящена функциям — важнейшему инструменту, который позволяет использовать код повторно, сокращать время отладки и повышать качество программ.

Глава 5 знакомит с функциональными возможностями Python. Вы узнаете, как писать генераторы и включения — мощные средства создания компактного, быстрого и экономного по памяти кода.

В главе 6 излагаются основы объектно-ориентированного программирования на Python. Здесь вы ознакомитесь с ключевыми концепциями и узнаете, какие возможности дает этот подход. Кроме того, прочтете об одной из самых полезных особенностей языка — декораторах.

В главе 7 описывается механизм исключений (ошибок, возникающих во время выполнения) и представлены способы их обработки. Здесь же рассматриваются контекстные менеджеры, особенно полезные при работе с ресурсами.

В главе 8 показывается, как работать с файлами, потоками, форматами обмена данными и базами данных, а также описываются способы хранения информации.

Глава 9 посвящена вопросам безопасности, хеш-функциям, шифрованию и токенам. Все это необходимо для создания безопасных программ.

В главе 10 рассказывается об основах тестирования и приводятся примеры того, как проверять код, чтобы он стал более надежным, стабильным и быстрым.

В главе 11 описываются основные методы отладки и профилирования, а также показано их применение на практике.

В главе 12 вы познакомитесь с синтаксисом и основными понятиями аннотаций типов. В последние годы типизация стала важной частью Python и его экосистемы.

Глава 13 представляет собой введение в анализ данных. Здесь мы разберем несколько ключевых концепций на примере практической задачи с использованием среды Jupyter Notebook.

В главе 14 рассказывается об основах создания API с помощью фреймворка FastAPI.

Глава 15 посвящена разработке консольных приложений (command-line interface applications, CLI), которые запускаются в терминале и активно используются разработчиками в повседневной работе.

В главе 16 показано, как подготовить проект к публикации и загрузить его в Python Package Index (PyPI).

В главе 17 вы познакомитесь с концепцией соревновательного программирования на примере двух задач с сайта Advent of Code.

Как получить максимум пользы от книги

Мы рекомендуем выполнять все приведенные здесь примеры. Для этого вам понадобится компьютер, подключение к Интернету и браузер. Книга написана под версию Python 3.12, однако в большинстве случаев примеры будут работать и в других актуальных версиях Python 3. В книге даны инструкции по установке Python в вашей операционной системе. Однако такие инструкции быстро устаревают, поэтому мы советуем обращаться к наиболее актуальным руководствам в Интернете, чтобы получить точные пошаговые инструкции. Кроме того, мы объяснили, как установить все дополнительные библиотеки, использующиеся в разных главах.

Для кодинга не требуется специальный редактор, но если вы хотите работать с примерами плодотворно, то имеет смысл использовать полноценную среду разработки. Несколько рекомендаций на этот счет приведены в главе 1.

Цветные иллюстрации

Мы подготовили PDF-файл с цветными изображениями всех снимков экрана и рисунков, приведенных в оригинальной книге. Его можно скачать по адресу https://packt.link/gbp/9781835882948.

Исходный код примеров

Код примеров из этой книги размещен на GitHub по адресу https://github.com/PacktPublishing/Learn-Python-Programming-Fourth-Edition.

Условные обозначения

В книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и важные понятия.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные окружения, операторы и ключевые слова, папки и каталоги, пути, имена файлов и их расширений.

Жирный моноширинный шрифт

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

Шрифт без засечек

Используется для обозначения элементов интерфейса, URL и адресов электронной почты.

Так оформляется предупреждение или примечание.

Так оформляется совет или предложение.

От издательства

Мы выражаем огромную благодарность Группе Arenadata за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство SprintBook, компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научных редакторах русскоязычного издания

Сергей Метелёв — руководитель команды инженеров по разработке и эксплуатации в Группе Arenadata. В IT 15 лет, из них 8 — в DevOps. Прошел путь от начинающего админа в региональном филиале сотового оператора до руководителя группы в департаменте разработки коммерческого продукта. Участвовал в запуске крупных региональных медицинских систем, работал в международной команде финтехпроекта.

Дмитрий Скорых — руководитель команды разработки и backend-разработчик в Группе Arenadata. Начинал путь с программирования микроконтроллеров и встраиваемых систем, разрабатывал продукты, работающие на Embedded Linux, в которых применял связку Python и C++. Участвовал в запуске продуктов с полным циклом разработки, начиная от электроники и заканчивая облачными сервисами. В последнее время работает с инфраструктурным платформенным ПО, активно используя Python для написания большей части микросервисов.

1 Плавное погружение в Python

Дай человеку рыбу — и он будет сыт один день. Научи человека рыбачить — и он будет сыт всю жизнь.

Китайская пословица

Программирование — или, как его еще называют, кодинг — это процесс, в ходе которого вы даете компьютеру инструкции на понятном для него языке.

Компьютеры — это мощные устройства, которые не умеют самостоятельно мыслить. Им нужно объяснить все: как выполнять задачи, как проверять условия, чтобы выбрать нужный путь, как обрабатывать данные, поступающие, например, с жесткого диска или из сети, и как реагировать на непредвиденные ситуации — скажем, если что-то отсутствует или вышло из строя.

Писать код можно по-разному, используя различные языки. Это трудно? И да и нет. Это похоже на письмо: научиться может каждый. Но если вы хотите стать поэтом — одного умения писать недостаточно. Придется развить целый набор других навыков, и это займет больше времени и сил.

В конечном счете все зависит от того, как далеко вы готовы пойти. Кодинг — это не просто составление работающих инструкций. Это нечто гораздо большее!

Хороший код — это короткий, быстрый, элегантный, понятный и удобный для чтения код. Его легко менять, расширять, масштабировать, тестировать и рефакторить. Чтобы научиться писать такой код, потребуется время. Но хорошая новость в том, что, читая эту книгу, вы уже делаете первый шаг. И мы не сомневаемся, что у вас получится. На самом деле это под силу каждому: ведь все мы программируем — просто не всегда осознаем это.

Допустим, вы хотите заварить растворимый кофе. Вам нужно взять кружку, банку с кофе, чайную ложку, воду и чайник. Даже если вы не задумываетесь об этом, в процессе вы анализируете множество факторов. Вы проверяете, есть ли вода в чайнике, включен ли он в розетку, чистая ли кружка и достаточно ли кофе в банке. Затем вы кипятите воду и, возможно, параллельно насыпаете кофе в кружку. Когда вода закипит, вы наливаете ее в кружку и размешиваете.

При чем здесь программирование?

Ну, мы собрали необходимые ресурсы (чайник, кофе, воду, ложку и кружку) и проверили условия, связанные с ними (чайник подключен, кружка чистая, кофе достаточно). Затем запустили два действия: начали кипятить воду и засыпали кофе в кружку. А когда оба действия завершились, мы закончили процедуру — налили воду в кружку и размешали.

Видите параллель? Мы только что описали высокоуровневую «функциональность» программы для приготовления кофе. Это было несложно, поскольку наш мозг и так делает это постоянно: оценивает условия, принимает решения, выполняет действия, повторяет какие-то из них и в какой-то момент завершает выполнение.

Теперь вам нужно лишь научиться раскладывать такие действия, которые вы обычно выполняете автоматически, на шаги, понятные компьютеру. А еще — выучить язык, на котором можно давать компьютеру такие инструкции.

Именно в этом и поможет наша книга. Мы покажем один из возможных подходов к успешному программированию и используем для этого множество простых, но конкретных примеров (мы такие особенно любим).

Основы программирования

Когда мы обучаем программированию, стараемся использовать примеры из реальной жизни. Мы уверены, что такие параллели помогают лучше запоминать новые понятия. Однако сейчас пришло время подойти к вопросу более строго и взглянуть на программирование с технической точки зрения.

Когда мы пишем код, мы даем компьютеру инструкции: что и как он должен делать. Где все это происходит? В разных частях системы: в оперативной памяти, на жестких дисках, в сетевых кабелях, процессоре и т. д. Это целый мир, который, как правило, представляет собой модель части реального мира.

Если вы пишете программу, с помощью которой люди могут покупать одежду онлайн, то вам нужно будет отразить в ней реальные сущности: людей, одежду, бренды, размеры — все это должно существовать внутри программы.

Для этого вы будете создавать и использовать объекты. Человек — это объект. Автомобиль — тоже объект, как и брюки. К счастью, Python отлично работает с объектами.

Любой объект в программе обладает двумя основными характеристиками: *свойствами* и *методами*. Например, если мы представляем человека как объект, то в программе это может быть клиент или сотрудник. Он может обладать такими свойствами, как имя, номер социального страхования, возраст, наличие водительских прав, адрес электронной почты и т. д. В коде мы сохраняем всю информацию, необходимую для использования объекта в соответствии с его назначением. Например, если вы разрабатываете сайт по продаже одежды, то вам, скорее всего,

нужно будет хранить данные о росте и весе клиента, а также другие параметры, которые позволят подбирать подходящие для него вещи.

Свойства — это характеристики объекта. Мы постоянно пользуемся ими в жизни.

— Подай мне ручку.

— Какую?

— Черную.

Здесь мы использовали свойство цвета (черная), чтобы указать на нужную ручку, — вероятно, потому, что рядом лежали и другие, отличающиеся по цвету.

Методы — это действия, которые объект может выполнять. Как человек, я могу *говорить*, *ходить*, *спать*, *просыпаться*, *есть*, *мечтать*, *писать*, *читать* и т. д. Все мои действия можно представить как методы объекта, который меня описывает.

Теперь, зная, что такое объекты, что у них есть методы (действия) и свойства (характеристики), вы готовы к программированию. Ведь, по сути, программирование — это управление объектами, которые существуют в модели мира, реализованной в вашем программном обеспечении. Вы можете создавать объекты, использовать их (в том числе повторно) и удалять по своему усмотрению.

В разделе *Data Model* официальной документации Python (https://docs.python.org/3/reference/datamodel.html) сказано:

> «Объекты — это абстракция, с помощью которой Python работает с данными. Все данные в Python представлены объектами или связями между ними».

К объектам в Python мы еще вернемся в главе 6. Пока же вам нужно знать только одно: каждый объект в этом языке имеет *идентификатор* (ID), *тип* и *значение*.

Объектно-ориентированное программирование (ООП) — лишь один из множества подходов к разработке. Python позволяет писать код в функциональном, императивном и объектно-ориентированном стиле. Однако, как мы уже говорили, в Python все является объектом — поэтому мы так или иначе постоянно работаем с объектами, вне зависимости от выбранного стиля программирования.

После создания объекта его идентификатор никогда не меняется. Это его уникальный номер, который Python использует «за кадром», чтобы находить объект при обращении к нему. Не изменяется и тип объекта. Он определяет, какие операции поддерживает объект и какие значения ему можно присваивать. С самыми важными типами данных в Python мы познакомимся в главе 2. А вот значение некоторых объектов может меняться. Такие объекты называют *изменяемыми* (mutable). Если же значение изменить нельзя — объект является *неизменяемым* (immutable).

Чтобы использовать объект, надо дать ему имя! Вы сможете обращаться к нему по имени и использовать его в программе. В более общем смысле объекты — такие как числа, строки (текст) и коллекции — связываются с именами. В языках

программирования такие имена обычно называют *переменными*. Переменную можно представить себе как коробку, в которую вы помещаете данные.

Объекты — это представление данных. Данные хранятся в базах или передаются по сети. Это то, что вы видите, открывая веб-страницу или работая с документом. Компьютерные программы обрабатывают эти данные: выполняют с ними действия, управляют их потоком, проверяют условия, реагируют на события и т. п.

Чтобы делать все это, нужен язык. Именно для этого и существует Python. В книге мы будем рассматривать, как с его помощью давать компьютеру понятные инструкции.

Добро пожаловать в мир Python

Python — замечательное творение Гвидо ван Россума, нидерландского ученого и математика, который решил подарить миру проект, над которым он экспериментировал во время рождественских каникул 1989 года. Публике Python был представлен примерно в 1991 году и с тех пор превратился в один из самых популярных языков программирования.

Мы, авторы книги, начали программировать еще в юности. Фабрицио начал в семь лет — на компьютере Commodore VIC-20, который позже был заменен на его более мощного «старшего брата» — Commodore 64. В нем использовался язык *BASIC*. Генрих познакомился с программированием в школе, когда начал учить Pascal. Вместе мы успели поработать с Pascal, Assembly, C, C++, Java, JavaScript, Visual Basic, PHP, ASP, ASP.NET, C# и множеством других языков, названия которых уже и не вспомнить. Но только когда мы впервые столкнулись с Python, у нас появилось то самое чувство, которое бывает, когда находишь в магазине нужный диван — *все внутри говорит: бери этот, он идеален!*

Мы привыкли к Python примерно за день. Синтаксис оказался непривычным — не таким, как в языках, с которыми мы работали раньше. Но, преодолев это легкое ощущение неловкости, мы сразу же влюбились в Python. По-настоящему. А почему — расскажем далее.

Коротко о Python

Прежде чем погрузиться в технические детали, разберемся, почему вообще стоит использовать Python. Ниже описаны его ключевые достоинства.

Переносимость

Python работает везде. Перенос программы из Linux в Windows или macOS обычно сводится к корректировке путей и настроек. Язык изначально спроектирован как переносимый, и он сам заботится о различиях между *операционными*

системами (ОС), пряча их за удобными интерфейсами. Вам не нужно писать специальный код для каждой платформы.

Логичность

Python отличается высокой логичностью и внутренней согласованностью. Сразу видно, что этот язык создал талантливый специалист. Очень часто название метода можно просто угадать, даже если вы его не знаете.

Вы можете пока не осознавать, насколько важны логичность и согласованность (особенно если у вас еще не так много опыта), но они дают огромное преимущество. Благодаря им уменьшается необходимость поисков в документации и снижается когнитивная нагрузка при программировании.

Скорость разработки

Как пишет Марк Лутц в пятом издании книги *Learning Python*[1], код Python обычно в пять раз короче, чем эквивалентный код на Java или C++. А значит, работа выполняется быстрее. А быстрее — это хорошо. Это значит, что вы можете быстрее реагировать на изменения рынка. Меньше строк означает снижение объемов кода, который нужно не только писать, но и читать (а профессиональные программисты читают код гораздо чаще, чем пишут), а также сопровождать, отлаживать и рефакторить.

Кроме того, Python не требует долгих этапов компиляции и обработки зависимостей. Вам не нужно ждать, чтобы увидеть результат своей работы, — код запускается сразу.

Обширная библиотека

Python обладает невероятно богатой стандартной библиотекой — недаром о нем говорят, что он поставляется *«в комплекте с батарейками»*. Но и это еще не все: международное сообщество Python поддерживает огромное количество сторонних библиотек, ориентированных на самые разные задачи. Эти библиотеки свободно доступны через репозиторий *Python Package Index (PyPI)*. Когда вы пишете код и понимаете, что вам нужна некая функциональность, почти всегда найдется хотя бы одна библиотека, в которой она уже реализована.

Качество программ

В Python большое внимание уделено удобству чтения, логичности и качеству кода. Благодаря единообразию синтаксиса программы на Python легко читать, а это особенно важно сейчас, когда разработкой занимаются коллективы,

[1] *Лутц М.* Изучаем Python. — М., 2024.

а не один человек. Кроме того, Python — язык с мультипарадигменной основой. Вы можете использовать его как скриптовый язык, а можете программировать в объектно-ориентированном, императивном или функциональном стиле. Python — по-настоящему универсальный язык.

Интеграция с другим ПО

У Python есть еще одно важное преимущество: его можно интегрировать в другие языки программирования. Это означает, что, даже если основная разработка в компании ведется на другом языке, Python может стать связующим звеном между разными частями сложной системы, которым нужно «общаться» друг с другом. Это уже более сложный уровень, но в реальной практике такая гибкость играет важную роль.

Использование в Data Science

Python — один из самых популярных (если не *самый* популярный) языков в таких областях, как анализ данных, машинное обучение и искусственный интеллект. Поэтому знать Python сегодня практически обязательно всем, кто хочет строить карьеру в этих направлениях.

Удовольствие от работы

И последнее, но не менее важное. Работать с Python — действительно в радость. Мы можем писать код восемь часов подряд и уходить из офиса с ощущением удовлетворения и легкости. Нас не изматывают трудности, с которыми сталкиваются другие разработчики, использующие языки с менее удобными структурами и инструментами. Нет сомнений, что Python делает программирование приятным занятием. А удовольствие, как известно, — один из лучших стимулов к обучению и продуктивной работе.

Вот основные причины, по которым мы рекомендуем Python. Конечно, у него есть и множество других технических особенностей и расширенных возможностей, о которых мы могли бы рассказать. Но все они выходят за рамки вводной главы. Вы будете с ними знакомиться по мере изучения Python, читая эту книгу глава за главой.

А теперь рассмотрим ограничения Python.

Недостатки Python

Если не учитывать вопрос личных предпочтений, то главный недостаток Python — скорость выполнения. Как правило, этот язык работает медленнее, чем его компилируемые собратья. Стандартная реализация Python при запуске приложения сначала компилирует исходный код в байт-код (файлы с расширением `.pyc`),

а затем передает его на выполнение интерпретатору. Преимущество такого подхода — переносимость, но достигается она ценой более медленной работы, поскольку Python не компилируется в машинный код напрямую, как, например, C или C++.

Тем не менее в большинстве случаев низкая скорость Python не становится проблемой, и именно поэтому язык используется так широко. В реальной жизни стоимость оборудования уже давно некритична, и во многих случаях повысить производительность можно за счет распараллеливания задач. Кроме того, многие программы тратят бóльшую часть времени на завершение операций ввода-вывода, поэтому чистая скорость выполнения нередко оказывается второстепенным фактором по отношению к общей производительности.

Надо отметить, что разработчики ядра Python в течение последних нескольких лет приложили немало усилий, стремясь оптимизировать работу со стандартными структурами данных, и во многих случаях эти улучшения дали ощутимый результат.

Если же скорость действительно критична, то можно рассмотреть альтернативные реализации Python, такие как *PyPy*, которая в среднем дает четырехкратное ускорение за счет более сложных приемов компиляции (см. https://pypy.org). Кроме того, для участков кода, где важна максимальная производительность, можно использовать такие языки, как C или C++, и затем интегрировать эти компоненты с кодом Python. В библиотеках *pandas* и *NumPy*, предназначенных для обработки и анализа данных, активно используется такой подход.

Существует несколько различных реализаций Python. Мы будем использовать основную — CPython. Перечень альтернатив доступен на сайте https://www.python.org/download/alternatives/.

Если вся эта информация кажется недостаточно убедительной, то напомним: Python лежит в основе серверной логики таких гигантов, как Spotify и Instagram, где производительность имеет огромное значение. Так что этот язык отлично справляется со своей задачей — даже там, где требования весьма высоки.

Как Python используют сегодня

Python используется в самых разных областях: в системном программировании, веб-разработке и работе с API, в графических приложениях, играх и робототехнике, при быстром прототипировании, системной интеграции, анализе данных, в базах данных, работе с потоками в реальном времени и многом другом. Кроме того, несколько ведущих университетов выбрали Python в качестве основного языка в курсах по информатике.

Далее приведен список крупных компаний и организаций и дано пояснение, в каких целях они используют Python в своей технологической инфраструктуре, при разработке продуктов, анализе данных или автоматизации процессов.

- Технологические компании:
  - Google — для серверной логики, анализа данных и систем искусственного интеллекта;
  - Facebook — для управления инфраструктурой и автоматизации операций;
  - Instagram — в значительной степени построен на Python, активно использует фреймворк Django;
  - Spotify — для анализа данных и серверной разработки;
  - Netflix — для анализа данных, автоматизации и обеспечения безопасности.
- Финансовый сектор:
  - JP Morgan Chase — для финансового моделирования, анализа данных и алгоритмической торговли;
  - Goldman Sachs — в различных финансовых приложениях;
  - Bloomberg — для анализа финансовых данных и интерфейса Bloomberg Terminal.
- Технологии и ПО:
  - IBM — для решений в области ИИ, машинного обучения и кибербезопасности;
  - Intel — для тестирования оборудования и в процессах разработки;
  - Dropbox — клиентская часть классического приложения в значительной степени написана на Python.
- Космос и научные исследования:
  - NASA — для анализа данных и интеграции систем;
  - CERN — для обработки и анализа данных в физических экспериментах.
- Розничная и электронная торговля:
  - Amazon — для анализа данных, рекомендаций товаров и автоматизации операций;
  - eBay — для серверной логики и анализа данных.
- Индустрия развлечений и медиа:
  - Pixar — для разработки инструментов анимации и написания сценариев для производственного процесса;
  - Industrial Light & Magic (ILM) — в визуальных эффектах и обработке изображений.

- Образование и онлайн-платформы:
  - Coursera — для веб-разработки и серверных решений;
  - Khan Academy — для доставки учебного контента и серверной логики.
- Государственный сектор и некоммерческие организации:
  - Федеральное правительство США — различные департаменты используют Python для анализа данных, кибербезопасности и автоматизации;
  - Фонд Raspberry Pi — использует Python как основной язык программирования для образовательных проектов.

Настройка окружения

На наших компьютерах (MacBook Pro) установлена последняя версия Python:

```
>>> import sys
>>> print(sys.version)
3.12.2 (main, Feb 14 2024, 14:16:36) [Clang 15.0.0 (clang-1500.1.0.2.5)]
```

Как видите, это версия 3.12.2, выпущенная 2 октября 2023 года[1]. А перед вами — небольшой фрагмент кода на Python, введенный в консоли. Скоро мы поговорим об этом подробнее.

Все примеры из книги запускаются с использованием Python версии 3.12. Если вы хотите выполнять примеры и работать с исходным кодом из книги, то, пожалуйста, убедитесь, что используете ту же версию.

Установка Python

Процесс установки языка зависит от операционной системы, которую вы используете. Он уже установлен по умолчанию в большинстве дистрибутивов Linux. В современных версиях macOS он тоже, скорее всего, уже есть. А вот в Windows его, как правило, нужно устанавливать вручную.

Даже если Python уже установлен в системе, важно убедиться, что это именно версия 3.12.

Чтобы проверить установленную версию Python, введите в командной строке одну из следующих команд: python --version или python3 --version. (Об этом мы еще поговорим.)

[1] На самом деле в этот день была выпущена версия 3.12.0, а 3.12.2 — 6 февраля 2024 года. Время в выводе команды показывает не дату релиза, а дату компиляции интерпретатора. — *Примеч. науч. ред.*

Первое место, куда стоит заглянуть, — официальный сайт Python https://www.python.org. Там вы найдете официальную документацию и множество полезных ресурсов.

Полезные ссылки по установке

На сайте Python собраны инструкции по установке языка для различных операционных систем. Выберите ту, которая соответствует вашей системе.

Windows и macOS:

- https://docs.python.org/3/using/windows.html;
- https://docs.python.org/3/using/mac.html.

Linux:

- https://docs.python.org/3/using/unix.html;
- https://ubuntuhandbook.org/index.php/2023/05/install-python-3-12-ubuntu/.

Установка Python в Windows

В качестве примера рассмотрим установку Python в Windows. Перейдите на сайт https://www.python.org/downloads/ и скачайте установочный файл, подходящий для архитектуры вашего процессора.

После этого дважды щелкните на скачанном файле, чтобы запустить установку (рис. 1.1).

Рис. 1.1. Запуск установки в Windows

Рекомендуем выбрать стандартную установку и снять флажок Add python.exe to PATH, чтобы избежать конфликтов с другими версиями Python[1], которые могут быть уже установлены на вашем компьютере — например, другими пользователями. Более подробные инструкции вы найдете по ссылке, приведенной в списке выше.

После нажатия кнопки Install Now начнется процесс установки (рис. 1.2).

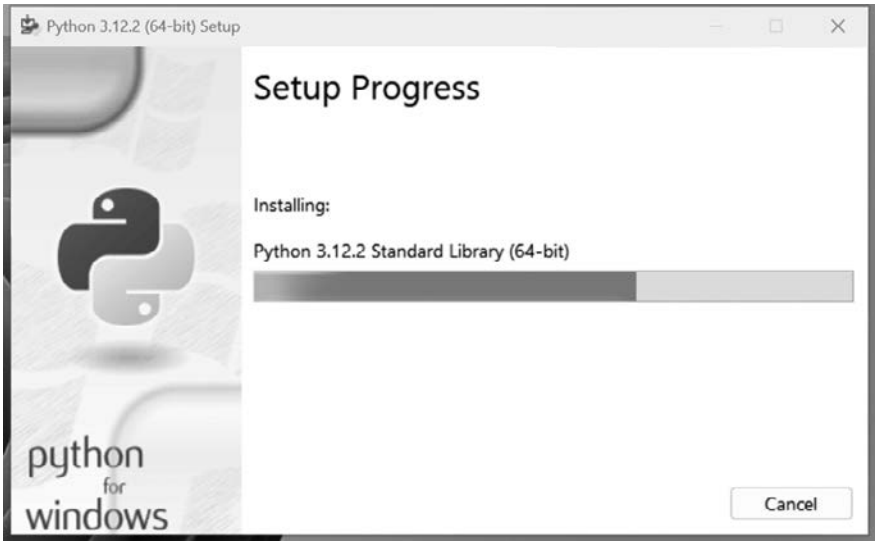


Рис. 1.2. Процесс установки

Когда установка завершится, появится финальное окно (рис. 1.3).

Нажмите кнопку Close, чтобы закрыть установщик.

Python установлен. Откройте командную строку и введите команду `py`. Тем самым вы запустите *интерактивную оболочку Python*, и автоматически будет выбрана последняя версия языка, установленная на вашем компьютере. На момент написания книги это 3.12. Если на вашей машине установлена более новая версия, то вы можете указать нужную, набрав `py -3.12`.

---

[1] Для удобства работы с несколькими версиями Python рекомендуем использовать pyenv https://github.com/pyenv/pyenv, у проекта также есть форк под Windows. Либо пакетный менеджер uv https://github.com/astral-sh/uv. Помимо управления зависимостями, он предлагает функциональность по управлению версиями Python, публикации пакетов. Является самым популярным пакетным менеджером в 2025 году. Подробнее о пакетных менеджерах и управлении зависимостями рассказывается в главе 16. — *Примеч. науч. ред.*

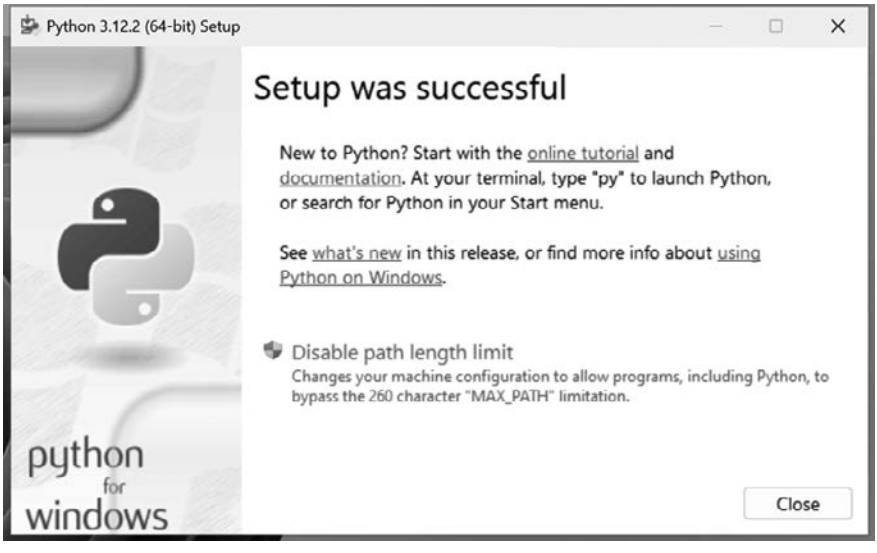


Рис. 1.3. Установка завершена

Чтобы открыть командную строку в Windows, откройте меню Пуск и введите `cmd` в строку поиска. Кроме того, можно воспользоваться PowerShell.

Установка Python в macOS

В macOS этот процесс похож на установку в Windows. Скачав установщик, подходящий для вашей системы, выполните инсталляцию. Затем откройте терминал: Программы ▸ Утилиты ▸ Терминал. Или установите Python через Homebrew.

В открывшемся окне терминала можно попробовать ввести команду `python`. Если откроется не та версия, попробуйте выполнить команду `python3` или `python3.12`.

Установка Python в Linux

Установить язык в Linux обычно немного сложнее, чем в Windows или macOS. Лучший способ — найти в Интернете актуальные инструкции для вашей версии дистрибутива. Они могут заметно различаться, поэтому привести универсальный пример сложно. Подробные ссылки на инструкции можно найти в пункте «Полезные ссылки по установке» (см. выше).

Консоль Python

В книге мы будем использовать слово *«консоль»* для обозначения терминала в Linux, командной строки или PowerShell в Windows, а также терминала в macOS. В примерах приглашение командной строки в стиле Linux оформляется вот так:

```
$ sudo apt-get update
```

Если вы не знакомы с консолью, то стоит уделить немного времени, чтобы разобраться в работе с ней. Если коротко, то после символа `$` вы будете вводить команды. Обращайте внимание на прописные буквы и пробелы — они влияют на результат.

Независимо от операционной системы, откройте консоль и введите `python` (для Linux и macOS) или `py` (для Windows). Убедитесь, что запускается интерактивная оболочка Python (она же Python REPL). Чтобы выйти из нее, введите `exit()`. Если в вашей системе установлено несколько версий Python, то может понадобиться вводить `python3` или `python3.12`.

В книге мы часто будем называть интерактивную оболочку Python просто консолью Python.

Вот пример того, что вы увидите при запуске Python (конкретный вывод зависит от версии и операционной системы):

```
$ python
Python 3.12.2 (main, Feb 14 2024, 14:16:36)
[Clang 15.0.0 (clang-1500.1.0.2.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Теперь, когда Python установлен и запускается, пора проверить, есть ли у вас еще один важный инструмент, который понадобится для работы с примерами из книги, — виртуальное окружение.

Несколько слов о виртуальных окружениях

При работе с Python очень часто используются виртуальные окружения. Что это такое и зачем они нужны? Разберем на простом примере.

Допустим, вы установили Python на свой компьютер и начали разрабатывать сайт для клиента X. Вы создаете папку с проектом и начинаете писать код. По ходу работы устанавливаете несколько библиотек — например, фреймворк Django версии 4.2.

Ваш сайт оказался настолько хорош, что вскоре к вам обращается другой клиент — Y. Он тоже хочет сайт, и вы начинаете проект Y. И снова вам нужен Django. Но к этому времени уже вышла версия 5.0, и вы хотите установить ее. И тут возникает проблема: при установке новой версии заменится старая, которую вы использовали в проекте X. Это может привести к проблемам с совместимостью, и вы не хотите рисковать. У вас есть два варианта: продолжать использовать старую версию Django, которая уже установлена, или обновить ее до новой версии и надеяться, что проект X при этом продолжит работать без ошибок.

Согласитесь, оба варианта не радуют, верно? Конечно, нет. Но решение есть — виртуальные окружения!

Виртуальное окружение — это изолированная среда Python, представляющая собой папку, в которой находятся все нужные исполняемые файлы и библиотеки для конкретного проекта.

Проще говоря, каждый проект получает собственную копию нужных пакетов, независимо от других проектов.

Итак, вы создаете виртуальное окружение для проекта X, устанавливаете в нем все необходимые библиотеки, затем создаете отдельное окружение для проекта Y и без малейших проблем устанавливаете в нем зависимости. Все, что вы устанавливаете, остается внутри соответствующего окружения. В нашем примере проект X использует Django 4.2, а проект Y — Django 5.0.

Очень важно: никогда не устанавливайте библиотеки глобально, на уровне всей операционной системы. Например, Linux зависит от Python, который используется для многих внутренних процессов. Если вы начнете менять системную установку Python, то рискуете нарушить стабильность и работу всей ОС. Запомните правило: всегда создавайте для каждого проекта отдельное виртуальное окружение.

Существует несколько способов создать виртуальное окружение в вашей операционной системе. Начиная с Python 3.5, рекомендуется использовать модуль venv, входящий в стандартную библиотеку Python. Официальная документация модуля доступна по адресу https://docs.python.org/3/library/venv.html.

Если вы применяете дистрибутив Linux на базе Debian (например, Ubuntu), то использование модуля venv может потребовать его установки:

```
$ sudo apt-get install python3.12-venv
```

Другой популярный способ создания виртуальных окружений — использование стороннего пакета `virtualenv`. Его официальный сайт — https://virtualenv.pypa.io.

В книге мы будем использовать рекомендованный способ создания окружения — с помощью модуля `venv` из стандартной библиотеки Python.

Ваше первое виртуальное окружение

Создать окружение очень просто, но команда, которую нужно ввести, может немного отличаться — в зависимости от того, как настроена ваша операционная система и с какой версией Python вы хотите работать. Кроме того, чтобы использовать создаваемое окружение, его нужно активировать. В ходе активации запускаемые в терминале Python библиотеки будут использоваться не из ОС, а изнутри среды. «За кулисами» будет происходить перенастройка путей.

Ниже приведен полный пример для macOS и Windows (в Linux все будет почти так же, как в macOS).

1. Откройте терминал и перейдите в папку, в которой хранятся проекты (в нашем случае это папка `code`). Создайте папку `my-project` и перейдите в нее.
2. Создайте виртуальное окружение `lpp4ed`.
3. Активируйте окружение (способы немного различаются для Linux/macOS и Windows).
4. Убедитесь, что запустилась нужная версия Python (3.12.X), — используя интерактивную оболочку.
5. Деактивируйте окружение, чтобы выйти из него.

Некоторые разработчики предпочитают давать виртуальным окружениям одно и то же имя — например, .venv. Это удобно: вы всегда знаете, как называется папка с окружением, и можно легко настраивать инструменты или запускать скрипты, зная только путь.

В Linux/macOS точка в начале .venv делает папку скрытой, то есть она не будет отображаться в списках по умолчанию.

Эти шаги — все, что вам нужно выполнить, чтобы начать новый проект.

Сейчас мы покажем пример на macOS. Результат может немного различаться в зависимости от вашей системы, версии Python и других факторов. В коде ниже знак # обозначает начало комментария (пояснение для человека, а не для машины); дополнительные пробелы добавлены для удобства чтения; а стрелка → указывает, что строка разделена из-за нехватки места на странице:

```
fab@m1:~/code$ mkdir my-project        # шаг 1
fab@m1:~/code$ cd my-project

fab@m1:~/code/my-project$ which python3.12    # проверяем, где находится
/usr/bin/python3.12                           # установленный Python 3.12

fab@m1:~/code/my-project$ python3.12 -m venv lpp4ed        # шаг 2
fab@m1:~/code/my-project$ source ./lpp4ed/bin/activate     # шаг 3

# Проверяем, какой Python теперь используется, — должен быть из окружения:
(lpp4ed) fab@m1:~/code/my-project$ which python
/Users/fab/code/my-project/lpp4ed/bin/python

(lpp4ed) fab@m1:~/code/my-project$ python     # шаг 4
Python 3.12.2 (main, Feb 14 2024, 14:16:36)
→ [Clang 15.0.0 (clang-1500.1.0.2.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

(lpp4ed) fab@m1:~/code/my-project$ deactivate     # шаг 5
fab@m1:~/code/my-project$
```

Каждый шаг снабжен соответствующим комментарием, так что выполнить инструкцию будет легко.

Обратите внимание: чтобы активировать виртуальное окружение, нужно запустить скрипт `lpp4ed/bin/activate` (или `Scripts\activate` в Windows). Имейте в виду, что этот скрипт нужно запускать с помощью команды `source` (в Linux/macOS) или просто как `.bat/.ps1` в PowerShell. Это означает, что он выполняется в текущем терминале, а не в новом процессе — и его эффект сохраняется после выполнения, что и нужно при активации окружения. Стоит также отметить, как меняется приглашение командной строки после активации среды: слева в строке появляется имя окружения (например, `(lpp4ed)`), а когда вы его отключаете — оно исчезает.

Пример создания виртуального окружения в Windows 11 (PowerShell):

```
PS C:\Users\H\Code> mkdir my-project          # шаг 1
PS C:\Users\H\Code> cd .\my-project\

# Проверяем доступные версии Python:
PS C:\Users\H\Code\my-project> py --list-paths
 -V:3.12 * C:\Users\H\AppData\Local\Programs\Python\Python312\python.exe

PS C:\Users\H\Code\my-project> py -3.12 -m venv lpp4ed     # шаг 2
PS C:\Users\H\Code\my-project> .\lpp4ed\Scripts\activate   # шаг 3

# Проверяем, какая версия Python в работе, теперь используется
# виртуальное окружение
(lpp4ed) PS C:\Users\H\Code\my-project> py --list-paths
* C:\Users\H\Code\my-project\lpp4ed\Scripts\python.exe
-V:3.12 C:\Users\H\AppData\Local\Programs\Python\Python312\python.exe

(lpp4ed) PS C:\Users\H\Code\my-project> python   # шаг 4
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb 6 2024, 21:26:36)
→ [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

(lpp4ed) PS C:\Users\H\Code\my-project> deactivate  # шаг 5
PS C:\Users\H\Code\my-project>
```

Обратите внимание, что в Windows после активации виртуального окружения можно использовать как команду `py`, так и `python`.

На данном этапе вы уже должны уметь создавать и активировать виртуальные окружения. Пожалуйста, попробуйте создать еще одно самостоятельно. Освойте эту процедуру — вы будете выполнять ее постоянно. *Мы никогда не работаем с Python на уровне всей системы* — помните? Виртуальные окружения — крайне важный инструмент.

В архиве с исходным кодом для книги примеры из каждой главы находятся в отдельной папке. Если в главе используется сторонняя библиотека, то в соответ-

ствующей папке вы найдете файл `requirements.txt` (или папку `requirements` с несколькими файлами), с помощью которого можно установить все необходимые зависимости. Мы рекомендуем при работе с кодом из главы создавать отдельное виртуальное окружение. Так вы не только обезопасите основной интерпретатор Python, но и попрактикуетесь в создании виртуальных окружений и установке сторонних библиотек.

Установка сторонних библиотек

Для установки сторонних библиотек используется менеджер пакетов Python — pip. Скорее всего, он уже установлен в вашем виртуальном окружении. Если нет, то всю необходимую информацию можно найти в официальной документации по адресу https://pip.pypa.io.

В примере ниже показано, как создать виртуальное окружение и установить несколько сторонних библиотек из файла `requirements.txt`:

```
fab@m1:~/code$ mkdir my-project
fab@m1:~/code$ cd my-project
fab@m1:~/code/my-project$ python3.12 -m venv lpp4ed
fab@m1:~/code/my-project$ source ./lpp4ed/bin/activate
(lpp4ed) fab@m1:~/code/my-project$ cat requirements.txt
django==5.0.3
requests==2.31.0
# следующая команда показывает, как установить библиотеки из файла
# с помощью pip
(lpp4ed) fab@m1:~/code/my-project$ pip install -r requirements.txt
Collecting django==5.0.3 (from -r requirements.txt (line 1))
  Using cached Django-5.0.3-py3-none-any.whl.metadata (4.2 kB)
Collecting requests==2.31.0 (from -r requirements.txt (line 2))
  Using cached requests-2.31.0-py3-none-any.whl.metadata (4.6 kB)
  ... more collecting omitted ...
Installing collected packages: ..., requests, django
Successfully installed ... django-5.0.3 requests-2.31.0
(lpp4ed) fab@m1:~/code/my-project$
```

Как видно в конце вывода `pip`, были установлены обе библиотеки, указанные в `requirements.txt`, а также несколько дополнительных. Это произошло потому, что `django` и `requests` сами зависят от других библиотек, и `pip` автоматически устанавливает все необходимые зависимости.

Использование файла requirements.txt — лишь один из способов установки сторонних пакетов в Python. Указать, что именно нужно установить, можно и напрямую: pip install django.

Полное руководство пользователя pip можно скачать по адресу https://pip.pypa.io/en/stable/user_guide/.

Теперь, когда мы закончили с настройкой окружения и установкой библиотек, можно подробнее поговорить о самом языке Python. Но прежде уделим немного времени консоли.

Консоль

В эпоху графических интерфейсов и сенсорных экранов идея использовать такой инструмент, как консоль, может показаться немного странной — ведь сегодня все необходимое находится на расстоянии в один щелчок кнопкой мыши.

Но правда в том, что каждый раз, когда вы убираете руку с клавиатуры, чтобы взять мышь и навести указатель на нужное место, вы теряете время. Работа через консоль, какой бы неудобной ни казалась поначалу, на деле приводит к увеличению скорости и продуктивности. Поверьте — мы знаем, о чем говорим. Вам придется просто довериться нам.

Скорость и эффективность важны, но дело не только в них. Умение работать с консолью полезно и по другой причине: когда вы пишете код, который в итоге окажется на сервере, консоль может быть единственным способом получить к нему доступ.

Если вы освоите консоль, то никогда не растеряетесь в ситуациях, где особенно важно действовать быстро — например, когда сайт перестал работать и нужно срочно понять, что произошло.

Если вы все еще сомневаетесь, то просто попробуйте. Работать с консолью гораздо проще, чем кажется, и вы точно не пожалеете. Нет ничего печальнее, чем опытный разработчик, который теряется в SSH-сессии на сервере, поскольку привык только к своей удобной среде и не умеет работать иначе.

Ну а теперь вернемся к Python.

Как запустить программу на Python

Существует несколько способов запуска программ на Python.

Запуск скриптов на Python

Python часто используется как скриптовый язык — и в этом качестве он невероятно полезен. *Скрипты* — это файлы (обычно небольшие), которые запускаются для выполнения конкретной задачи. У многих разработчиков со временем появляется целый набор скриптов, которые они используют для автоматизации типичных задач. Например, можно иметь скрипт для парсинга данных из одного формата в другой, скрипт для работы с файлами и папками, скрипт для создания или изменения конфигурационных файлов. Технически почти любую задачу можно решить с помощью скрипта.

Очень часто скрипты запускаются автоматически по расписанию на сервере. Допустим, база данных вашего сайта требует регулярной очистки каждые 24 часа (например, нужно удалять просроченные пользовательские сессии). В этом случае можно настроить задание Cron, которое будет запускать скрипт ежедневно в 01:00.

Cron — это утилита-планировщик заданий во времени, применяемая в Unix-подобных системах. С ее помощью системные администраторы и разработчики назначают выполнение команд и скриптов на определенное время, конкретную дату или с заданной периодичностью.

У нас самих есть целый набор скриптов на Python, которые берут на себя рутинные задачи, при выполнении вручную отнимавшие у нас по несколько минут или даже больше. В какой-то момент мы поняли: пора их автоматизировать.

Запуск интерактивной оболочки Python

Еще один способ запустить Python — использовать интерактивную оболочку. Вы уже сталкивались с ней, когда вводили команду `python` в консоли.

Откройте консоль, активируйте виртуальное окружение (что, надеемся, вы уже делаете автоматически) и введите команду `python`. Вы увидите несколько строк, примерно таких:

```
(lpp4ed) fab@m1 ~/code/lpp4ed$ python
Python 3.12.2 (main, Feb 14 2024, 14:16:36)
[Clang 15.0.0 (clang-1500.1.0.2.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Символы `>>>` — это приглашение интерактивной оболочки. Они означают, что Python ждет от вас команды. Если вы введете простую инструкцию из одной строки, результат появится на следующей строке. Но если вы начнете писать многострочную конструкцию (например, цикл `for` или определение функции), то приглашение изменится на `...`, указывая, что вы пока не закончили ввод.

Попробуйте сами. Выполним простые математические действия:

```
>>> 3 + 7
10
>>> 10 / 4
2.5
>>> 2 ** 1024
179769313486231590772930519078902473361797697894230657273430081157732675805500
963132708477322407536021120113879871393357658789768814416622492847430639474124
237776789342486548527630221960124609411945308295208500576883815068234246288147
391311054082723716335051068458629823994724593847971630483535632962422413721616
```

Последняя операция — по-настоящему впечатляющая: мы возвели `2` в степень `1024`, и Python легко справился с этим. Попробуйте сделать то же самое в Java, C++ или C# — если у вас нет специальных библиотек для обработки больших чисел, то у вас ничего не получится. В отличие от этих языков Python поддерживает произвольную точность для целых чисел прямо из коробки.

Мы используем интерактивную оболочку каждый день. Она очень удобна, чтобы быстро что-то протестировать: проверить, поддерживает ли структура данных нужную операцию, посмотреть, как работает фрагмент кода, и т. д. Вы быстро поймете, что оболочка Python станет вашим надежным помощником в работе.

Есть и другой вариант, с дружелюбным графическим интерфейсом, — *IDLE (Integrated Development and Learning Environment)*. Это простейшая среда разработки (IDE), которая предназначена в основном для новичков, но имеет более широкие возможности, чем обычная консольная оболочка, так что можете попробовать поработать с ней. Она бесплатно входит в установщики Python для Windows и macOS и легко ставится в других системах. Подробнее об IDLE можно узнать на сайте Python по адресу https://www.python.org.

Гвидо ван Россум назвал Python в честь британской комик-группы Monty Python, а IDLE якобы назвали в честь одного из ее участников — Эрика Айдла (Eric Idle).

Запуск Python в качестве сервиса

Помимо выполнения в виде скрипта или в интерактивной оболочке, Python можно запускать как полноценное приложение. В книге вы не раз увидите примеры такого подхода. Мы рассмотрим эту тему более подробно чуть позже, когда будем говорить о структуре кода Python и о том, как его запускать.

Запуск Python в качестве графического приложения

С помощью Python можно также создавать *графические пользовательские интерфейсы* (Graphical User Interfaces, GUIs). Для этого существует множество библиотек: одни из них кросс-платформенные, другие предназначены для работы в определенной операционной системе. Один из самых популярных вариантов — *Tkinter*, объектно-ориентированная надстройка над GUI-библиотекой `Tk` (собственно, Tkinter расшифровывается как Tk interface).

Tk — инструмент для создания графических интерфейсов, предлагающий более высокий уровень абстракции, чем традиционные подходы. Он изначально был создан для языка Tool Command Language (Tcl), но поддерживается и в других динамических языках, включая Python. Tk позволяет создавать нативные настольные приложения, которые без проблем работают в Windows, Linux, macOS и других системах.

Tkinter входит в стандартную поставку Python, поэтому с него удобно начинать работу с графическим интерфейсом.

Среди других популярных фреймворков для создания GUI-приложений можно отметить:

- PyQt/PySide;
- wxPython;
- Kivy.

Подробное описание этих библиотек выходит за рамки книги, но всю необходимую информацию можно найти на сайте Python: https://docs.python.org/3/faq/gui.html.

На этом сайте обратите внимание на раздел *What GUI toolkits exist for Python?*. Если вы хотите заняться графическим интерфейсом, то при выборе библиотеки проверьте, соответствует ли она таким критериям, как:

- поддержка всех нужных вам инструментов для разработки;
- совместимость с нужными платформами;
- активность и широта сообщества;
- возможность простого доступа и установки необходимых компонентов.

Структура кода Python

Пора немного поговорить о том, из чего состоит код Python. В этом разделе мы заглянем в пресловутую «кроличью нору» и обсудим некоторые технические нюансы и концепции.

Начнем с основ. Код Python, как и в любом языке, вы пишете в текстовых файлах. Когда файл сохраняется с расширением `.py`, он называется *модулем Python*.

В Windows или macOS расширения файлов скрыты по умолчанию. Если вы работаете в этих системах, то рекомендуем изменить данную настройку, чтобы отображать полные имена файлов. Это не строго обязательное требование, но часто оно помогает различать файлы, особенно при работе с большим количеством модулей.

Хранить весь код программы в единственном файле — не самое практичное решение. Такой подход уместен разве что для простых скриптов, которые обычно состоят из нескольких сотен строк (а то и меньше).

Полноценное приложение на Python может насчитывать сотни тысяч строк кода, и, конечно, приходится разбивать его на отдельные модули. Это уже лучше, но все еще недостаточно удобно. Поэтому предпочтительнее использовать такую структуру Python, как *пакет*. Она позволяет группировать модули по папкам.

Пакет в Python — это просто папка. В старых версиях языка, чтобы обозначить папку как пакет, в ней нужно было создать специальный файл `__init__.py`. Этот файл может быть пустым и, начиная с Python 3.3, является необязательным, но на практике его все равно стоит добавлять — по ряду технических причин.

Как обычно, объяснить все куда проще на примере. Мы подготовили демонстрационную структуру в нашем проекте. И когда в консоли вводим следующую команду:

```
$ tree -v example
```

мы получаем древовидное представление содержимого папки `ch1/example`, в которой хранится код для примеров из этой главы. Структура простой программы на Python может выглядеть так:

```
example
├── core.py
├── run.py
└── util
    ├── __init__.py
    ├── db.py
    ├── maths.py
    └── network.py
```

В корне проекта находятся два модуля: `core.py` и `run.py`, а также один пакет — `util`. В `core.py`, скорее всего, содержится основная логика приложения. В `run.py`, как можно догадаться, размещается код для запуска приложения. А в пакете `util` хранятся вспомогательные модули, и даже по их названиям можно понять, для чего они предназначены: `db.py` — утилиты для работы с базами данных, `maths.py` — математические функции (возможно, приложение связано с финансами), а `network.py` — функции для отправки и получения данных по сети.

В папке `util` находится файл `__init__.py`. Как уже говорилось ранее, он позволяет Python понять, что это не просто папка, а пакет.

А теперь сравните предыдущий код с версией, в которой все организовано только в виде *отдельных модулей*, без использования пакетов. Такой пример мы разместили в папке `ch1/files_only/` — посмотрите сами:

```
$ tree -v files_only
```

В этом случае структура выглядит так:

```
files_only
├── core.py
├── db.py
├── maths.py
├── network.py
└── run.py
```

Согласитесь, понять назначение каждого модуля уже сложнее. А теперь представьте, что это лишь упрощенный пример. В реальном приложении, где код не разделен на пакеты и модули, разобраться в структуре было бы намного труднее.

Как использовать модули и пакеты

При разработке приложения часто возникает ситуация, когда один и тот же фрагмент логики нужен в разных частях программы. Например, если вы пишете обработчик формы, которую пользователь заполняет на сайте, то вам может понадобиться проверить, является ли введенное значение числом. И независимо от того, как написана логика, подобная проверка может потребоваться сразу для нескольких полей.

Представьте, что вы создаете приложение для опросов. Пользователю нужно ответить на вопросы, и некоторые ответы будут числовыми.

- Сколько вам лет?
- Сколько у вас домашних животных?
- Сколько у вас детей?
- Сколько раз вы были женаты/замужем?

В каждом из этих случаев нужно выполнить одну и ту же проверку: является ли введенное значение числом. Плохая практика — копировать эту проверку вручную в каждый участок кода, где она требуется. Такой подход нарушает принцип DRY (Don"t Repeat Yourself — «Не повторяйся»), который гласит: «Один и тот же фрагмент кода не должен дублироваться в приложении. Никогда». Да, мы осознаем, что это звучит строго, но хотим подчеркнуть: не повторяйте код, даже если он кажется коротким и простым.

Почему дублирование логики — плохая идея?

- В коде может быть ошибка — и вам придется исправлять ее в каждой копии.
- Вы можете захотеть изменить реализацию механизма проверки — и снова нужно будет менять все вручную.
- При внесении изменений можно пропустить одну из копий, и в результате приложение будет работать неправильно или непоследовательно.
- Код станет длиннее и запутаннее, а пользы это не принесет.

Python — удивительный язык. Он предоставляет все инструменты, необходимые для соблюдения хороших практик программирования. В примере выше нам нужно повторно использовать один и тот же фрагмент кода. Чтобы делать это грамотно, необходим некий механизм, который содержит нужный код и который

можно вызывать всякий раз, когда требуется выполнить ту же самую логику. Такой механизм есть — и это функция.

Пока мы не будем углубляться в детали, просто запомните следующую информацию. *Функция* — это блок организованного, повторно используемого кода, предназначенный для выполнения определенной задачи. Функции бывают разными — и по форме, и по названию, в зависимости от контекста, в котором применяются. Но в данный момент эти тонкости неважны, и мы вернемся к ним позже. Функции лежат в основе модульности любого приложения. Без них почти не обойтись. За исключением случаев, когда вы пишете простейший скрипт, функции будут использоваться постоянно. К их подробному описанию мы перейдем в главе 4.

Как уже упоминалось ранее, Python поставляется с очень богатой стандартной библиотекой. Сейчас как раз подходящий момент, чтобы уточнить, что *библиотека* — это набор функций и объектов, которые расширяют возможности языка. Например, в стандартной библиотеке `math` вы найдете множество математических функций, включая `factorial`, которая вычисляет факториал числа.

В математике факториалом неотрицательного целого числа N, обозначаемым как N!, называется произведение всех положительных целых чисел от 1 до N. Например, факториал числа 5 вычисляется так:

$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$

А в соответствии с соглашением факториал нуля равен единице: 0! = 1. Это называется пустым произведением, и оно всегда считается равным 1.

Если вы хотите использовать такую функцию в своем коде, то все, что нужно сделать, — импортировать ее и вызвать, используя правильные входные значения. Возможно, вам пока не до конца понятно, что такое «входные значения» или «вызов функции». Не волнуйтесь на этот счет. Сейчас для вас главное — понять принцип импорта. Подключая библиотеку, мы импортируем из нее нужные компоненты, а затем используем их по назначению. Например, чтобы в Python вычислить 5!, достаточно этого кода:

```
>>> from math import factorial
>>> factorial(5)
120
```

При работе в интерактивной оболочке Python все, что можно отобразить, выводится автоматически. В данном случае это результат вызова функции — 120.

Помните наш пример со структурой, состоящей из модулей `core.py`, `run.py` и папки `util`? Так вот, это наша собственная библиотека утилит, своего рода пояс с инструментами, где лежат нужные функции, которые можно импортировать

и использовать повторно. Как вы помните, это `db.py` — функции для работы с базами данных, `network.py` — сетевые функции, `maths.py` — функции для математических расчетов (они не входят в стандартную библиотеку Python, поэтому мы пишем их самостоятельно).

О том, как импортировать функции из модулей и использовать их в коде, мы еще поговорим. А сейчас рассмотрим другое ключевое понятие — *модель выполнения кода в Python*.

Модель выполнения кода в Python

В этом разделе вы познакомитесь с несколькими важными понятиями, такими как область видимости, имена и пространства имен. Полное описание модели выполнения кода можно найти в официальной документации языка по адресу https://docs.python.org/3/reference/executionmodel.html. Но если честно, оно довольно сухое и техническое, поэтому мы начнем с менее формального объяснения на понятном языке.

Имена и пространства имен

Представьте, что вы ищете книгу, приходите в библиотеку и спрашиваете, где она. Вам отвечают: «Второй этаж, сектор X, третий ряд». Вы поднимаетесь по лестнице, находите нужный сектор и идете в указанный ряд — все логично. А теперь вообразите другую библиотеку, где все книги свалены в одну кучу в огромной комнате. Никаких этажей, секторов, полок, никакого порядка. Найти нужную книгу в таком хаосе почти невозможно.

С кодом — такая же история. Если не организовать структуру программы, разобраться в ней будет очень сложно, особенно тому, кто видит ее впервые. Хорошая структура упрощает навигацию, способствует повторному использованию кода и помогает избежать дублирования логики.

Допустим, у нас есть книга. Мы ссылаемся на нее по названию. В терминах Python это *имя*. В нем имена — это ближайший аналог того, что в других языках называется переменными. Имя ссылается на объект и создается в момент *привязки*, то есть когда вы присваиваете значение. Ниже приведен короткий пример (напоминаем, что текст, указанный после символа #, — это комментарий, то есть пояснение для человека, а не для машины):

```
>>> n = 3  # целое число
>>> address = "221b Baker Street, NW1 6XE, London"  # адрес Шерлока Холмса
>>> employee = {
...     'age': 45,
...     'role': 'CTO',
...     'SSN': 'AB1234567',
... }
```

```
>>> # теперь выведем их на экран
>>> n
3
>>> address
'221b Baker Street, NW1 6XE, London'
>>> employee
{'age': 45, 'role': 'CTO', 'SSN': 'AB1234567'}
>>> other_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'other_name' is not defined
>>>
```

Как вы уже знаете, у каждого объекта в Python есть идентификатор, тип и значение. В примере мы создали три объекта. Посмотрим, какого они типа и какие значения содержат.

- `n` — целое число (тип: `int`; значение: `3`).
- `address` — строка (тип: `str`; значение: адрес Шерлока Холмса).
- `employee` — словарь (тип: `dict`; значение: объект-словарь с тремя парами «ключ — значение»).

Если вы пока не знаете, что такое словарь, то не волнуйтесь об этом. В главе 2 мы подробно рассмотрим его — словарь там представлен как главная структура данных в Python.

Вы заметили, что приглашение Python поменялось с >>> на ..., когда мы начали вводить employee? Это потому, что определение заняло несколько строк и Python визуально показывает, что вы все еще вводите одну команду.

Итак, что такое `n`, `address` и `employee`? Это *имена*. Их нужно где-то хранить, чтобы с их помощью можно было получить доступ к объектам в программе и извлечь нужные данные. Таким местом хранения является пространство имен.

Пространство имен сопоставляет имена и объекты, на которые эти имена указывают. Проще говоря, это структура, в которой Python записывает, какие имена к каким объектам привязаны. Примеры пространств имен: встроенное пространство имен — содержит функции и объекты, доступные в любом коде Python; глобальное пространство имен в модуле; локальное пространство имен в функции; даже набор атрибутов объекта тоже считается пространством имен.

Преимущество пространств имен в том, что они позволяют четко определять и организовывать имена, не приводя к конфликтам и дублированию имен. Например, пространство имен, связанное с книгой, которую мы ищем в библиотеке, можно использовать для импорта самой книги, например, так:

```
from library.second_floor.section_x.row_three import book
```

Мы начинаем с пространства имен `library` и с помощью оператора точки (`.`) входим в него. Внутри мы ищем пространство имен `second_floor` и снова переходим внутрь, используя оператор точки. Затем повторяем то же действие с пространством `section_x` и, наконец, в последнем пространстве имен — `row_three` — находим нужное имя: `book`.

Переход по пространствам имен станет более понятным, когда мы начнем работать с реальными примерами кода. А пока просто запомните: пространства имен — это области, в которых имена сопоставлены с объектами.

Есть еще одно понятие, тесно связанное с пространствами имен, о котором мы хотим кратко упомянуть. Это *область видимости*.

Области видимости

Согласно документации Python,

> «область видимости — это фрагмент текста программы на Python, в котором пространство имен доступно напрямую».

«Доступно напрямую» означает, что при обращении к имени без уточнения (при использовании неполной ссылки) Python будет искать его в соответствующем пространстве имен.

Области видимости определяются статически (то есть по исходному коду), но используются динамически во время выполнения программы. Это значит, что по коду можно заранее понять, в какой области видимости находится имя. Python предоставляет четыре уровня областей видимости (не все из них обязательно присутствуют одновременно).

- *Локальная область*. Внутренняя область, содержащая локальные имена — то есть те, которые определены внутри текущей функции или блока.
- *Область замыкающей функции*. Область видимости любой внешней функции, если та вложена. Содержит нелокальные и неглобальные имена.
- *Глобальная область*. Область модуля, в котором выполняется код. Содержит глобальные имена — те, что определены на уровне файла.
- *Встроенная область*. Python поставляется с набором встроенных функций, которые готовы к использованию. К их числу относятся `print`, `abs`, `all` и др. Все они находятся во встроенной области видимости.

Правило следующее: когда вы обращаетесь к имени, Python начинает искать его в текущем пространстве имен. Если имя не найдено, то поиск продолжается во внешней области видимости, затем — в глобальной, и наконец — во встроенной. Если даже в последней имя не найдено, то Python выбрасывает *исключение* `NameError`, которое означает, что такое имя не определено (вы могли видеть его исключение в предыдущем примере).

Таким образом, Python ищет имя по четырем уровням, от внутреннего к внешнему, соблюдая такой порядок: локальная область, замыкающая, глобальная, встроенная (local, enclosing, global, built-in, LEGB).

Отложим теорию и рассмотрим пример. Чтобы вы могли увидеть локальные и внешние области, нужно определить несколько функций. Не беспокойтесь, если пока не знакомы с их синтаксисом — мы подробно разберем его в главе 4. На данный момент просто знайте: если в коде встречается ключевое слово `def`, значит, мы определяем функцию.

```
# scopes1.py
# Локальная и глобальная области видимости

# определяем функцию local
def local():
    age = 7
    print(age)

# определяем переменную age в глобальной области видимости
age = 5

# вызываем (или "выполняем") функцию local
local()

print(age)
```

В данном примере имя `age` задается и в глобальной, и в локальной областях видимости. Когда вы запустите код (не забыли активировать виртуальное окружение?) такой командой:

```
$ python scopes1.py
```

в консоли появятся два числа: 7 и 5.

Что происходит? Интерпретатор Python начинает читать файл сверху вниз. Он пропускает комментарии, затем находит определение функции `local`. Будучи вызванной, она создаст имя `age`, связанное с числом `7`, в локальной области и выведет его на экран. Интерпретатор продолжает обработку кода и видит еще одну привязку имени `age`, но теперь уже в глобальной области — со значением `5`. Затем вызывается функция `local()` — и она выводит 7. После этого вызывается `print(age)` — и теперь выводится глобальное значение, то есть `5`.

Стоит подчеркнуть один особенно важный нюанс: в коде, который входит в тело функции `local`, строки внутри функции сдвинуты вправо на четыре пробела. Именно так Python определяет область видимости: вход в нее происходит с помощью добавления отступа, а выход — с помощью его устранения. Одни программисты используют два пробела, другие — три, но рекомендуется использовать четыре. Это количество считается оптимальным с точки зрения читабельности кода. О стандартах оформления мы подробно поговорим позже.

В других языках — таких как Java, C# или C++ — области видимости обозначаются фигурными скобками ({ … }). Значит, в Python установка отступа соответствует открывающей фигурной скобке, а его удаление — закрывающей.

А что произойдет, если удалить из функции строку `age = 7`? Вспомним правило LEGB. Python начнет искать `age` в локальной области и, не найдя, перейдет в замыкающую. Но в данном коде ее нет, поэтому он сразу перейдет в глобальную — и найдет `age = 5`. В результате программа дважды выведет `5`. Вот как будет выглядеть такой код:

```
# scopes2.py
# Локальная и глобальная области видимости

def local():
    # age не принадлежит области видимости, определенной функцией local,
    # поэтому Python продолжит поиск во внешней (замыкающей) области.
    # В итоге переменная age будет найдена в глобальной области.
    print(age, 'printing from the local scope')

age = 5
print(age, 'printing from the local scope')

local()
```

Если запустить файл `scopes2.py`, то мы увидим на экране следующее:

```
$ python scopes2.py
5 printing from the global scope
5 printing from the local scope
```

Как и ожидалось, Python сначала выводит значение переменной `age`. Затем вызывается функция `local()`, в которой переменная `age` не определена локально, поэтому Python продолжает искать имя, следуя цепочке LEGB, — и находит `age` в глобальной области видимости. Теперь рассмотрим пример, в котором появляется дополнительный уровень — замыкающая область:

```
# scopes3.py
# Локальная, замыкающая и глобальная области видимости

def enclosing_func():
    age = 13

    def local():
        # age не принадлежит области, определенной функцией local,
        # поэтому Python продолжит поиск во внешней (замыкающей) области.
        # На этот раз имя переменной age будет найдено в замыкающей
        # области enclosing_func
        print(age, 'printing from the local scope')

    # вызываем функцию local
    local()
```

```
# определяем переменную age в глобальной области
age = 5
print(age, 'printing from the global scope')

# вызываем функцию enclosing_func
enclosing_func()
```

Если запустить файл `scopes3.py`, то будет выведено:

```
$ python scopes3.py
5, 'вывод из глобальной области видимости'
13, ''
```

Как видите, команда `print` внутри функции `local` снова обращается к имени `age`. Это имя все еще не определено внутри самой функции, поэтому Python начинает обход областей в соответствии с порядком LEGB. На этот раз он находит `age` в *замыкающей* области.

Не переживайте, если все это пока немного путает. По мере изучения других примеров логика станет более понятной. Рекомендуем прочитать раздел *Classes* в официальном учебнике Python по адресу https://docs.python.org/3/tutorial/classes.html. Там есть отличный параграф, посвященный областям видимости и пространствам имен, — обязательно прочтите его, если хотите лучше понять эти темы.

Рекомендации по написанию хорошего кода

Писать хороший код не так просто, как кажется. Как мы уже говорили, качественный код сочетает в себе множество достоинств и достичь баланса между ними — настоящее искусство. Каким бы ни был ваш прогресс в этой области, есть то, что стоит освоить как можно раньше, — *PEP 8*.

PEP (Python Enhancement Proposal) — это документ, описывающий предлагаемое улучшение в языке Python. Помимо новых возможностей языка, в PEP также могут описываться процессы разработки и содержаться рекомендации и стандарты. Полный список PEP можно найти по адресу https://www.python.org/dev/peps.

PEP 8 — самый известный из всех PEP-документов. Он содержит четкие и простые рекомендации по оформлению кода Python, чтобы тот был читабельным, понятным и соответствовал стилю языка. Если вы запомните из этой главы только один совет, то пусть он будет таким: используйте PEP 8. Следуйте ему. Впоследствии вы еще не раз поблагодарите себя за это.

В наши дни разработка — это не работа в одиночку. Код создается совместными усилиями, и над одним проектом часто работают несколько разработчиков. С помощью таких инструментов, как Git и Mercurial, множество людей могут писать и редактировать один и тот же код одновременно.

Git и Mercurial — самые популярные распределенные системы контроля версий. Они помогают командам разработчиков совместно работать над одним и тем же проектом.

В таких условиях как никогда важно, чтобы весь код писался в едином стиле. Когда все разработчики в компании придерживаются PEP 8, бывает так, что, открыв чей-то код, один из разработчиков подумает: «Погодите... это я писал?» (Так постоянно происходит с Фабрицио; просто он быстро забывает код, который пишет.)

У этого подхода есть огромное преимущество: если вы читаете код, который могли бы написать сами, то читаете его легко. Не будь соглашений, каждый разработчик писал бы код в собственном стиле — так, как ему привычно или как его когда-то научили. В результате каждую строчку пришлось бы интерпретировать с учетом чужих привычек, а на это уходило бы гораздо больше времени. Благодаря PEP 8 этого можно избежать. Мы сами придерживаемся его настолько сильно, что в нашей команде ревью не принимаются, если код не соответствует PEP 8. Так что, пожалуйста, найдите время и изучите этот документ — он действительно важен.

Разработчики на Python могут воспользоваться разными инструментами, которые автоматически форматируют код в соответствии с PEP 8. Вот несколько популярных: black и ruff — автоматическое форматирование, а flake8 и PyLint — так называемые линтеры: они проверяют стиль и выдают предупреждения, подсказывая, что нужно исправить. Мы искренне рекомендуем использовать эти инструменты — они упрощают задачу написания красивого и аккуратного кода.

В примерах мы будем стараться максимально соблюдать PEP 8. Правда, мы не всегда сможем позволить себе строки кода длиной 79 символов (рекомендация PEP 8), поэтому иногда будем жертвовать пустыми строками и отступами. Но мы обещаем: весь код будет оформлен так, чтобы его было как можно легче читать.

Культура Python

Python получил широкое распространение в индустрии программного обеспечения. Его используют компании самого разного профиля и масштаба для решения различных задач. При этом Python остается и прекрасным языком для обучения: он прост, благодаря чему его легко освоить; он поощряет хорошие привычки — например, писать код, удобный для чтения; он независим от платформы; он поддерживает современные объектно-ориентированные принципы программирования.

Одна из причин популярности Python — огромное, живое и вдохновляющее сообщество. По всему миру проводятся конференции и митапы, посвященные Python или его фреймворкам — например, Django.

Исходный код Python открыт, и те, кто его использует, тоже часто открыты к идеям и диалогу. Посмотрите страницу сообщества на официальном сайте Python, и если хочется, то присоединяйтесь!

Есть еще один важный аспект культуры Python — это понятие *«питонично»* (Pythonic). Оно связано с тем, что данный язык позволяет выражать идеи лаконично, естественно и в своем стиле — таком, которого нет в других языках или который там сложнее реализовать. Иногда после Python становится почти тесно в других языках — настолько удобно здесь все устроено.

Со временем появилось целое понятие: *«писать питонично»*, то есть *так, как принято в Python: просто, понятно и элегантно*.

Чтобы прочувствовать эту культуру, взглянем на «Дзен Python» — это небольшое «пасхальное яйцо», но в нем заключено много смысла. Откройте интерактивную оболочку Python и введите команду `import this`. Появится текст — своего рода манифест стиля Python:

```
>>> import 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!
```

Этот текст можно воспринимать на двух уровнях. Более простой — как набор советов, изложенных иронично и тепло. Более сложный — как философию, отражающую принципы Python, которые стоит осмыслить, если вы действительно хотите писать питонично. Начните с легкого уровня. А затем осваивайте глубины.

Несколько слов об IDE

Для выполнения примеров из книги IDE необязательна — достаточно любого нормального текстового редактора. Если же вам нужны дополнительные возможности — такие как подсветка синтаксиса, автодополнение, поиск по проекту, — то стоит выбрать подходящую среду IDE. Полный список открытых и бесплатных IDE для Python можно найти на сайте Python — просто введите поисковый запрос *Python IDE*.

Фабрицио пользуется программой Microsoft Visual Studio Code. Она бесплатна, нативно поддерживает множество функций и легко расширяется с помощью плагинов.

После многих лет работы с редакторами (в том числе с Sublime Text) именно VS Code показалась ему самой продуктивной.

Генрих, в свою очередь, ярый поклонник Neovim. Освоение этого редактора требует определенных усилий[1], но он исключительно мощный и гибко настраивается с помощью плагинов. Еще один плюс: Neovim совместим со своим предшественником — Vim, который установлен практически в любой системе, с которой регулярно взаимодействует разработчик.

Дадим два очень важных совета.

- Какую бы IDE вы ни выбрали, постарайтесь основательно изучить ее, чтобы использовать все ее сильные стороны. Но при этом *не становитесь зависимыми от нее*. Иногда тренируйтесь работать в Vim (или любом простом редакторе), чтобы уметь ориентироваться на любой платформе и в любых условиях.
- Независимо от IDE/редактора, в коде на Python *отступ равен четырем пробелам*. Никаких знаков табуляций. Никаких сочетаний этих знаков с пробелами. Не два пробела. Не три. И не пять. Только четыре. Так делает весь мир Python. Не стоит становиться изгнанником только потому, что вам приглянулся стиль с тремя пробелами.

Несколько слов об ИИ

В последние несколько лет мир стал свидетелем бурного развития искусственного интеллекта. Сейчас на рынке появилось довольно много решений, в том числе инструменты, помогающие программистам.

Да, ИИ уже умеет писать фрагменты кода, но это ни в коей мере не отменяет причин, по которым стоит учить язык программирования. ИИ-инструменты еще очень далеки от того, что способен сделать человек. Они несовершенны и на

[1] Есть множество шуток о том, как непросто выйти из vim. — *Примеч. науч. ред.*

момент написания этой книги применяются в основном для решения повторяющихся и рутинных задач.

Многие IDE уже поддерживают интеграцию с ИИ, например с GitHub Copilot и другими аналогичными решениями. Visual Studio Code, Zed, IntelliJ IDEA и PyCharm поддерживают расширение возможностей с помощью ИИ-плагинов. Появились даже новые IDE, разработанные специально для функций ИИ, — например, Cursor.

Мы тоже используем такие инструменты в своей работе, но считаем важным подчеркнуть: старайтесь понимать код из этой книги самостоятельно. Попробуйте разобраться в примерах вручную, без помощи ИИ-ассистента — это неотъемлемая часть процесса обучения.

Резюме

В этой главе мы начали знакомить вас с программированием и языком Python. Пока мы дали только вводный материал, но уже упомянули концепции, которые подробно разберем в следующих главах.

Мы обсудили основные характеристики Python, кто его использует и зачем, какие есть способы написания программ на этом языке.

В финальной части главы мы прошлись по базовым понятиям: пространства имен и области видимости. Кроме того, мы показали, как организовать структуру кода, используя модули и пакеты.

Что касается практики, то вы установили Python и убедились, что у вас есть нужные инструменты (например, `pip`). Помимо этого, вы создали и активировали свое первое виртуальное окружение. Теперь вы можете работать в изолированном окружении, не затрагивая системную установку Python. Это безопасно и удобно.

Теперь вы готовы начать путешествие по миру Python вместе с нами! Все, что вам нужно, — энтузиазм, активированное виртуальное окружение, эта книга, ваши пальцы и, возможно, небольшая чашка кофе.

Старайтесь выполнять примеры вручную — они короткие и простые. Если вы воспользуетесь ими на практике, то усвоите материал намного лучше, чем если бы просто читали книгу.

В следующей главе мы рассмотрим богатый набор встроенных типов данных Python. Это очень интересная тема, и вы сможете многому научиться.

2 Встроенные типы данных

> «Данные! Данные! Данные! — нетерпеливо воскликнул он. — Я не могу делать кирпичи без глины».
>
> *Перефразированная цитата из книги «Медные буки» Артура Конана Дойля*

Все, что вы делаете на компьютере, связано с обработкой данных. Они бывают самых разных видов: музыка, которую вы слушаете, фильмы, которые вы смотрите, PDF-файлы, которые вы открываете. Даже глава, которую вы сейчас читаете, — всего лишь файл. А файл — это данные.

Данные могут быть простыми, как целое число для хранения возраста, или сложными, как структура заказа в интернет-магазине; единичными, описывающими один объект, или составными, описывающими набор объектов; и даже *метаданными*, то есть данными о данных: о структуре, контексте, других данных приложения. В Python *объекты — это абстракции для представления данных*. И язык предоставляет богатый набор встроенных структур данных, а также возможность комбинировать их и создавать собственные типы.

В Python всё — объекты

Прежде чем перейти к конкретике, немного подробнее поговорим об объектах в Python. *В этом языке объектом является все*, и каждый объект имеет *идентификатор (ID)*, *тип* и *значение*. Но что именно происходит, когда вы пишете, например, инструкцию `age = 42` на Python?

Если вы зайдете на сайт https://pythontutor.com, то сможете ввести эту строку в окно с кодом и увидеть, как Python визуально представляет происходящее. Сохраните эту ссылку — она отлично помогает разобраться в том, что происходит «за кадром».

Итак, когда выполняется строка `age = 42`, происходит следующее: создается *объект*, ему присваивается идентификатор, его тип становится `int` (целое число), а значение — `42`. Имя `age` добавляется в глобальное пространство имен и указывает на этот объект. Поэтому, пока мы находимся в глобальной области, мы можем обратиться к этому объекту просто по имени: `age`.

Представьте, что вы переезжаете: вы складываете ножи, вилки и ложки в коробку и наклеиваете на нее этикетку *«столовые приборы»*. Та же концепция применима и к коду: имя — это «ярлык на коробке», который указывает на конкретный объект с типом, идентификатором и значением. На рис. 2.1 показано, как имя указывает на объект (возможно, вам придется немного изменить настройки сайта, чтобы получить такой же вид).

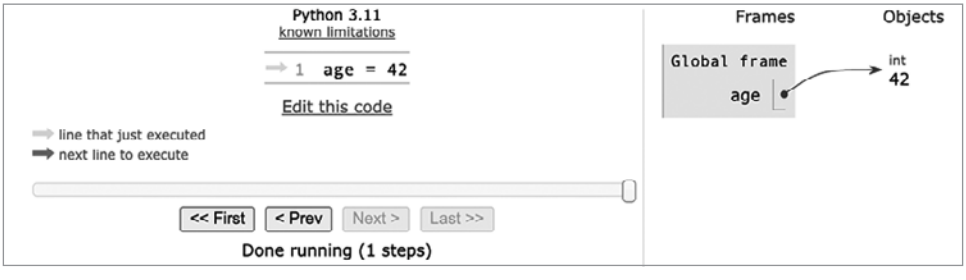


Рис. 2.1. Имя указывает на объект

Поэтому, встречая в главе конструкции вида `имя = значение`, представляйте такую картину: имя помещается в пространство имен, связанное с текущей областью видимости, и стрелка от него указывает на объект, у которого есть *идентификатор*, *тип* и *значение*. В описанном механизме есть еще пара тонкостей, но объяснить их проще на конкретных примерах. Мы обязательно поговорим об этом позже.

Изменяемость

Первое важное различие, которое Python вводит для данных, — может ли значение объекта изменяться. Если да, то объект называют *изменяемым* (mutable). Если нет, то он считается *неизменяемым* (immutable).

Это различие крайне важно понимать, поскольку оно влияет на процесс написания кода. Рассмотрим такой пример:

```
>>> age = 42
>>> age
42
>>> age = 43  #A
>>> age
43
```

В примере в строке с #A изменили ли мы значение age? На самом деле нет. Хотя, конечно, вы можете сказать: «Но теперь оно равно 43...» Да, 43. Но 42 — это объект типа int, а целые числа в Python — неизменяемые. То есть мы не изменили 42 на 43, а просто присвоили age новое значение. Произошло вот что: в первой строке age указывает на объект int, значение которого равно 42; во второй строке создается новый объект int со значением 43; имя age теперь указывает на этот новый объект; объект 42 никуда не делся — просто age его больше не «видит»; у этих объектов будут разные идентификаторы. То есть мы не изменили значение объекта, а переназначили имя — присвоили его другому объекту. Теперь посмотрим на ID этих объектов:

```
>>> age = 42
>>> id(age)
4377553168
>>> age = 43
>>> id(age)
4377553200
```

Мы вызвали встроенную функцию id(), чтобы вывести идентификаторы объектов. Как видно из вывода, они разные — и это вполне ожидаемо. Помните: имя age в каждый момент времени указывает только на один объект: сначала на имеющий значение 42, потом на имеющий значение 43 — никогда на оба сразу.

Если вы повторите эти примеры у себя на компьютере, то, скорее всего, увидите другие идентификаторы. Это нормально — Python назначает их динамически и они меняются от запуска к запуску.

Теперь рассмотрим аналогичный пример, в котором будет использоваться изменяемый объект. Для этого возьмем встроенный тип set:

```
>>> numbers = set()
>>> id(numbers)
4368427136
>>> numbers
set()

>>> numbers.add(3)
>>> numbers.add(7)
>>> id(numbers)
4368427136
>>> numbers
{3, 7}
```

Что здесь происходит: мы создаем объект numbers, представляющий математическое множество; сразу после создания видим его идентификатор и его

значение — `set()` (то есть пустое); затем добавляем в него два элемента: `3` и `7`; повторно выводим `id()` и само множество. Результат: идентификатор остался тем же — это все тот же объект; значение изменилось — теперь в множестве есть два элемента. Это и есть типичное поведение изменяемого объекта. О множествах (`set`) более подробно мы поговорим далее в этой главе.

Об изменяемости мы будем говорить еще не раз — она действительно важна.

Числа

Начнем со встроенных числовых типов данных в Python. Язык был создан человеком с ученой степенью по математике и информатике, поэтому поддержка чисел здесь мощная и продуманная.

Все числовые типы в Python — неизменяемые объекты.

Целые числа (int)

В Python целые числа (`int`) имеют неограниченную точность. Единственное ограничение — доступная оперативная память. То есть насколько бы большое число вам ни понадобилось — если оно помещается в память, то Python его обработает.

Целые числа могут быть положительными, отрицательными и равными нулю (`0`). Их тип — `int`. Они поддерживают все стандартные арифметические операции. Ниже приведен пример:

```
>>> a = 14
>>> b = 3
>>> a + b   # сложение
17
>>> a - b   # вычитание
11
>>> a * b   # умножение
42
>>> a / b   # деление с плавающей запятой
4.666666666666667
>>> a // b  # целочисленное деление
4
>>> a % b   # остаток от деления (операция по модулю)
2
>>> a ** b  # возведение в степень
2744
```

Этот код должен быть понятен. Но обратите внимание на важный момент: в Python два оператора деления: `/` — *обычное деление* (возвращает результат с плавающей запятой) и `//` — *целочисленное деление* (возвращает результат, *округленный* вниз).

Историческая справка: в Python 2 оператор / работал иначе, чем в Python 3. Теперь поведение более очевидное и согласованное.

Посмотрим, как ведет себя деление, если в выражении появляются отрицательные числа:

```
>>> 7 / 4     # деление с плавающей запятой
1.75
>>> 7 // 4    # целочисленное деление, усечение дает 1
1
>>> -7 / 4    # снова деление с плавающей запятой,
              # результат – противоположность предыдущего
-1.75
>>> -7 // 4   # целочисленное деление, результат не является
              # противоположностью предыдущего
-2
-2
```

Это довольно любопытный пример. Если вы ожидали увидеть в последней строке `-1`, то не переживайте, так думают многие. Просто в Python целочисленное деление *всегда округляется вниз — в сторону минус бесконечности*. Если же вам нужно не такое округление, а отсечение дробной части, то используйте встроенную функцию `int()`:

```
>>> int(1.75)
1
>>> int(-1.75)
-1
```

Как видите, `int()` отсекает дробную часть в сторону нуля.

Функция int() также может преобразовывать строки в числа в заданной системе счисления:

```
>>> int('10110', base=2)
22
```

Оператор возведения в степень ** имеет встроенный аналог — функцию `pow()`:

```
>>> pow(10, 3)
1000
>>> 10 ** 3
1000
>>> pow(10, -3)
0.001
>>> 10 ** -3
0.001
```

Существует и оператор остатка от деления — `%`, он называется *оператором остатка от деления (модуля)*:

```
>>> 10 % 3   # остаток от деления 10 // 3
1
>>> 10 % 4   # остаток от деления 10 // 4
2
```

Интересно, что функция `pow()` принимает третий аргумент — и в этом виде выполняет *возведение в степень по модулю*.

Если основание и модуль — взаимно простые, то форма с отрицательной степенью возвращает обратный по модулю элемент (или соответствующую ему степень, если показатель меньше -1), по модулю третьего аргумента.

Ниже приведен пример:

```
>>> pow(123, 4)
228886641
>>> pow(123, 4, 100)
41                  # обратите внимание: 228886641 % 100 == 41
>>> pow(37, -1, 43)  # обратный элемент для 37 по модулю 43
7
>>> (7 * 37) % 43    # проверка правильности предыдущего результата
1
```

Начиная с версии Python 3.6 появилась возможность добавлять подчеркивания внутри числовых литералов — между цифрами или в префиксах систем счисления (подчеркивания нельзя ставить в начале или конце числа — они работают только как визуальные разделители внутри записи). Это помогает сделать большие числа, такие как `1_000_000_000`, более удобными для чтения:

```
>>> n = 1_024
>>> n
1024
>>> hex_n = 0x_4_0_0 # 0x400 == 1024
>>> hex_n
1024
```

Логический тип (bool)

Булева алгебра — это раздел алгебры, в котором переменные могут принимать только два значения: *истина* и *ложь*. В Python такие значения обозначаются ключевыми словами `True` и `False`. Логический тип (`bool`) в Python — это подкласс `int`, поэтому `True` и `False` ведут себя как `1` и `0` соответственно. Тип `bool` является

логическим эквивалентом `int` и возвращает одно из двух значений: `True` или `False`. Каждый встроенный объект в Python может быть приведен к логическому значению — то есть при передаче в функцию `bool()` он даст `True` или `False`.

Логические значения можно комбинировать в булевы выражения с помощью операторов `and` — логическое И, `or` — логическое ИЛИ и `not` — логическое НЕ. Рассмотрим пример:

```
>>> int(True)   # True ведет себя как 1
1
>>> int(False)  # False ведет себя как 0
0
>>> bool(1)     # 1 оценивается как True в булевом контексте
True
>>> bool(-42)   # и любое ненулевое число тоже оценивается как True
True
>>> bool(0)     # 0 оценивается как False
False
>>> # краткий обзор операторов and, or, not
>>> not True
False
>>> not False
True
>>> True and True
True
>>> False or True
True
```

Логические значения чаще всего используются в условных конструкциях, о которых мы подробно поговорим в главе 3.

Вы можете убедиться, что `True` и `False` являются подклассами целых чисел (`int`), если попробуете сложить их. Python автоматически приводит их к типу `int` и выполняет сложение:

```
>>> 1 + True
2
>>> False + 42
42
>>> 7 - True
6
```

Такое преобразование называется восходящим (upcasting), то есть переходом от подкласса к родительскому классу. В данном случае True и False являются объектами класса, унаследованного от int, и при необходимости преобразуются обратно в целые числа. Эта тема относится к наследованию и будет подробно рассмотрена в главе 6.

Вещественные числа (float)

Вещественные числа, или *числа с плавающей запятой*, в Python представлены согласно стандарту IEEE 754 — в формате двойной точности (double precision). Такой формат занимает 64 бита и делится на три части: знак, порядок и мантисса (дробная часть).

Хочется больше информации? Можно прочитать статью в Википедии: http://en.wikipedia.org/wiki/Double-precision_floating-point_format (и https://ru.wikipedia.org/wiki/Число_с_плавающей_запятой).

Во многих языках программирования доступны два формата: одинарной точности (32 бита) и двойной (64 бита). В Python поддерживается только второй. Например:

```
>>> pi = 3.1415926536  # сколько знаков числа Пи вы помните?
>>> radius = 4.5
>>> area = pi * (radius ** 2)
>>> area
63.617251235400005
```

Скобки вокруг radius ** 2 необязательны (оператор ** имеет более высокий приоритет, чем *), но такую запись проще воспринимать визуально — особенно как математическую формулу. Если вы получите немного другое значение, то не волнуйтесь: оно может зависеть от операционной системы, способа компиляции Python и других факторов. Пока совпадают первые несколько знаков после запятой, вы можете считать результат верным.

Сведения о точности и поведении чисел с плавающей запятой можно получить с помощью объекта `sys.float_info`. Он содержит информацию о минимальном и максимальном значениях, точности, порядке и других характеристиках `float` для вашей системы:

```
>>> import sys
>>> sys.float_info
sys.float_info(
    max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
    min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307,
    dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2,
    rounds=1
)
```

Давайте немного поразмышляем. У нас есть 64 бита, чтобы представить число с плавающей запятой. Это значит, что можно представить не более чем `264` (то есть 18 446 744 073 709 551 616) различных значений. Если посмотреть на значения `max` и `epsilon` для чисел с плавающей запятой, то станет очевидно, что представить все возможные значения нельзя — не хватает памяти. Поэтому значения округляются до ближайшего допустимого числа. Возможно, вы думаете, что такие проблемы возникают только с очень большими или очень малыми числами. Если да, то следующий пример вас удивит:

```
>>> 0.3 - 0.1 * 3 # казалось бы, должно получиться 0!!!
-5.551115123125783e-17
```

Что это означает? Даже такие простые числа, как `0.1` и `0.3`, не могут быть представлены точно в формате двойной точности. Это может быть серьезной проблемой, если вы работаете с ценами, финансовыми расчетами или любыми данными, требующими высокой точности. Но не переживайте, в Python есть решение — тип `Decimal`. Он избавлен от этих проблем, и мы познакомимся с ним чуть позже.

Комплексные числа

В Python поддержка этих чисел предусмотрена по умолчанию. На случай, если вы раньше не сталкивались с ними, знайте: *комплексное число* — это число вида `a + ib`, где `a` и `b` — действительные числа, а `i` (или `j` в инженерной записи) — мнимая единица, то есть корень из –1. Части `a` и `b` называются соответственно *действительной* и *мнимой* частями комплексного числа.

Возможно, вам нечасто придется использовать такие числа на практике, но все же рассмотрим небольшой пример:

```
>>> c = 3.14 + 2.73j
>>> c = complex(3.14, 2.73)  # то же, что и выше
>>> c.real         # действительная часть
3.14
>>> c.imag         # мнимая часть
2.73
>>> c.conjugate()  # сопряженное число A + Bj — это A - Bj
(3.14-2.73j)
>>> c * 2          # умножение разрешено
(6.28+5.46j)
>>> c ** 2         # возведение в степень тоже работает
(2.406700000000001+17.1444j)
>>> d = 1 + 1j     # сложение и вычитание также доступны
>>> c - d
(2.14+1.73j)
```

Дроби и десятичные числа

Обзор числовых типов в Python мы завершим кратким описанием дробей и десятичных чисел. Объекты типа `Fraction` представляют рациональные числа в виде числителя и знаменателя в несократимом виде. Рассмотрим простой пример:

```
>>> from fractions import Fraction
>>> Fraction(10, 6)  # странновато, правда?
Fraction(5, 3)       # обратите внимание: дробь была сокращена
>>> Fraction(1, 3) + Fraction(2, 3)  # 1/3 + 2/3 == 3/3 == 1/1
Fraction(1, 1)
>>> f = Fraction(10, 6)
>>> f.numerator      # числитель
5
>>> f.denominator    # знаменатель
3
>>> f.as_integer_ratio()  # представление в виде целочисленного отношения
(5, 3)
```

У чисел типа `int` и логических значений также появился метод `as_integer_ratio()`. Это удобно — вы можете вызывать его, не задумываясь о том, с каким именно типом числа работаете.

Помимо обычной передачи числителя и знаменателя, объекты `Fraction` можно объявлять (инициализировать) и другими способами: строками, десятичными числами, числами с плавающей запятой и, конечно, другими дробями. Вот пример с `float` и строками:

```
>>> Fraction(0.125)
Fraction(1, 8)
>>> Fraction("3 / 7")
Fraction(3, 7)
>>> Fraction("-.250")
Fraction(-1, 4)
```

Объекты `Fraction` временами могут быть полезны, однако в коммерческом программном обеспечении они встречаются редко. Куда чаще используются десятичные числа — особенно там, где точность критична, например в научных или финансовых расчетах.

Важно помнить, что десятичные числа произвольной точности требуют дополнительных ресурсов. Для хранения каждого значения нужно больше памяти, чем для дробей или чисел с плавающей запятой. К тому же интерпретатор Python выполняет с ними больше действий «за кадром».

Рассмотрим пример с десятичными числами:

```
>>> from decimal import Decimal as D  # переименовали для краткости
>>> D(3.14)    # число пи из float, поэтому есть ошибки аппроксимации
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> D("3.14")  # число пи из строки, без ошибок аппроксимации
Decimal('3.14')
>>> D(0.1) * D(3) - D(0.3)       # из float, ошибки все равно есть
Decimal('2.77555756156515654042363166 8E-17')
>>> D("0.1") * D(3) - D("0.3")   # из строки — результат точный
Decimal('0.0')
>>> D("1.4").as_integer_ratio()  # 7/5 = 1.4 (отлично, не так ли?!)
(7, 5)
```

Обратите внимание: если вы создаете `Decimal` из числа с плавающей запятой, то оно унаследует все его проблемы с приближениями. А вот если вы создаете его из строки или целого числа — все будет точно, никаких сюрпризов. Если вы работаете с финансовыми или любыми другими данными, где крайне важны точные расчеты, — используйте `Decimal`.

На этом мы завершаем введение во встроенные числовые типы Python. Пора переходить к последовательностям.

Неизменяемые последовательности

В этом разделе мы рассмотрим неизменяемые последовательности: строки, кортежи и байты.

Строки и байты

Текстовые данные в Python представлены объектами типа `str`, чаще всего называемыми *строками*. Это неизменяемые последовательности кодовых точек Unicode.

Кодовая точка Unicode — это число, присвоенное каждому символу в стандарте Unicode. Он представляет собой универсальную систему кодирования символов, применяемую для представления текста на компьютерах. Unicode присваивает каждому символу уникальное числовое значение, независимо от платформы, программы или языка, что обеспечивает единообразие при хранении и обработке текста в различных системах. В Unicode входят символы латиницы[1], иероглифы китайского, японского и корейского языков, знаки, эмодзи и многое другое.

В отличие от некоторых других языков программирования в Python нет типа `char`, поэтому отдельный символ представляется строкой длиной 1.

[1] И кириллицы. — *Примеч. науч. ред.*

Для внутреннего представления текста в любом приложении следует использовать Unicode. Однако при хранении текстовых данных или их передаче по сети обычно необходимо выполнить кодирование, выбрав подходящую кодировку в зависимости от среды. В результате кодирования получается объект типа `bytes`, синтаксис и поведение которого во многом схожи со строками. Строковые литералы в Python можно записывать с помощью одинарных, двойных или тройных кавычек (повторяющихся трижды одинарных либо двойных). Если строка заключена в тройные кавычки, то может занимать несколько строк. Рассмотрим такой пример:

```
>>> # четыре способа создать строку
>>> str1 = 'This is a string. We built it with single quotes.'
# строка в одинарных кавычках
>>> str2 = "This is also a string, but built with double quotes."
# строка в двойных кавычках
>>> str3 = '''This is built using triple quotes,
... so it can span multiple lines.'''
# тройные одинарные кавычки для многострочного текста
>>> str4 = """This too
... is a multiline one
... built with triple double-quotes."""  # тройные двойные кавычки
                                         # для многострочного текста
>>> str4         #A, вывод строки с символами переноса (в виде \n)
'This too\nis a multiline one\nbuilt with triple double-quotes.'
>>> print(str4)  #Б, вывод строки с реальными переносами строк
This too
is a multiline one
built with triple double-quotes.
```

В строках `#A` и `#Б` мы выводим `str4`: сначала неявно, затем явно — с помощью функции `print()`. Полезным упражнением будет разобраться, почему вывод различается. Готовы решить эту задачу? (Подсказка: поищите информацию о функциях `str()` и `repr()`.)

Строки, как и любые последовательности, имеют длину. Ее можно узнать с помощью функции `len()`:

```
>>> len(str1)
49
```

Начиная с Python 3.9 появилось два новых метода для работы с префиксами и суффиксами строк. Вот пример, который иллюстрирует их работу:

```
>>> s = "Hello There"
>>> s.removeprefix("Hell")
'o There'
>>> s.removesuffix("here")
'Hello T'
>>> s.removeprefix("Ooops")
'Hello There'
```

Главное преимущество этих методов понятно из последней строки: если нужный префикс или суффикс не найден, то метод просто возвращает копию исходной строки. «За кадром» происходит проверка, начинается или заканчивается ли строка переданным аргументом — и если да, то он удаляется.

Кодирование и декодирование строк

С помощью методов `encode()` и `decode()` можно кодировать строки Unicode и декодировать байтовые объекты. Наиболее часто используемая *кодировка символов — UTF-8*. Это кодировка переменной длины, способная представлять все возможные кодовые точки Unicode. Она считается стандартной кодировкой для Интернета. Стоит также отметить, что если перед строкой добавить литерал `b`, то Python создаст объект типа `bytes`. Например:

```
>>> s = "This is üņíc0de"          # строка Unicode: кодовые точки
>>> type(s)
<class 'str'>
>>> encoded_s = s.encode("utf-8")        # версия строки, закодированная в utf-8
>>> encoded_s
b'This is \xc3\xbc\xc5\x8b\xc3\xadc0de'  # результат: объект bytes
>>> type(encoded_s)                      # еще один способ проверить тип
<class 'bytes'>
>>> encoded_s.decode("utf-8")            # возвращаемся к исходной строке
'This is üņíc0de'
>>> bytes_obj = b"A bytes object"        # объект bytes
>>> type(bytes_obj)
<class 'bytes'>
```

Индексация и срезы строк

При работе с последовательностями часто требуется обратиться к элементу по индексу *(индексация)* или получить подпоследовательность *(срез)*. Для неизменяемых последовательностей обе операции доступны только для чтения.

Индексация всегда одинакова: нумерация начинается с нуля, и вы можете обратиться к любому элементу по позиции. Срезы, напротив, бывают разных видов. Чтобы получить срез последовательности, можно указать *начальную* и *конечную* позиции, а также *шаг*. Эти параметры разделяются двоеточиями (`:`), как в следующем выражении: `my_sequence[start:stop:step]`. Все параметры являются необязательными; `start` добавляется в результат, а `stop` — нет.

Проще всего увидеть это на примере:

```
>>> s = "The trouble is you think you have time."
>>> s[0]       # индексация: символ на позиции 0, то есть первый символ
'T'
>>> s[5]       # индексация: символ на позиции 5, то есть шестой символ
'r'
```

```
>>> s[:4]      # срез: указана только конечная позиция
'The '
>>> s[4:]      # срез: указана только начальная позиция
'trouble is you think you have time.'
>>> s[2:14]    # срез: указаны начальная и конечная позиции
'e trouble is'
>>> s[2:14:3]  # срез: начальная, конечная позиции и шаг (каждый третий символ)
'erb '
>>> s[:]       # быстрый способ сделать копию строки
'The trouble is you think you have time.'
```

Последняя строка особенно интересна. Если вы не укажете ни один из параметров, то Python подставит значения по умолчанию. В этом случае `start` — начало строки, `stop` — конец строки, а `step` задан по умолчанию — 1. Такой прием позволяет быстро получить копию строки `s` (значение то же, но объект другой). Сможете ли вы придумать, как получить инвертированную копию строки с помощью среза? (Не ищите готовый ответ, попробуйте догадаться сами.)

Форматирование строк

У строк есть полезное свойство: их можно использовать как шаблоны. Это означает, что строка может содержать заполнители, которые заменяются произвольными значениями с помощью операций форматирования. Существует несколько способов форматировать строки. Полный список возможностей вы найдете в документации; здесь же приведены несколько распространенных примеров:

```
>>> greet_old = "Hello %s!"
>>> greet_old % 'Fabrizio'
'Hello Fabrizio!'
>>> greet_positional = "Hello {}!"
>>> greet_positional.format("Fabrizio")
'Hello Fabrizio!'
>>> greet_positional = "Hello {} {}!"
>>> greet_positional.format("Fabrizio", "Romano")
'Hello Fabrizio Romano!'
>>> greet_positional_idx = "This is {0}! {1} loves {0}!"
>>> greet_positional_idx.format("Python", "Heinrich")
'This is Python! Heinrich loves Python!'
>>> greet_positional_idx.format("Coffee", "Fab")
'This is Coffee! Fab loves Coffee!'
>>> keyword = "Hello, my name is {name} {last_name}"
>>> keyword.format(name="Fabrizio", last_name="Romano")
'Hello, my name is Fabrizio Romano'
```

В примере показаны четыре различных способа форматирования строк. Первый из них — с использованием оператора `%` — может привести к неожиданным ошибкам и требует осторожности. Более современный способ — использовать метод строки `format()`. Как видно из примера, пара фигурных скобок в строке играет роль заполнителя. Когда вызывается метод `format()`, в него передаются данные, которые подставляются на место этих заполнителей. Внутри фигурных скобок

можно указать индексы (и не только), а также имена переменных — в последнем случае метод `format()` вызывается с именованными аргументами.

Обратите внимание, как строка `greet_positional_idx` форматируется по-разному в зависимости от переданных данных.

Одна из функциональностей, появившихся в Python 3.6, — *форматированные строковые литералы*, или *f-строки*. Она действительно удобна (и работает быстрее, чем метод `format()`): строка начинается с символа `f` и содержит выражения в фигурных скобках, которые вычисляются во время выполнения и форматируются по протоколу `format`.

```
>>> name = "Fab"
>>> age = 48
>>> f"Hello! My name is {name} and I'm {age}"
"Hello! My name is Fab and I'm 48"
>>> from math import pi
>>> f"No arguing with {pi}, it's irrational..."
"No arguing with 3.141592653589793, it's irrational..."
```

Интересное расширение f-строк появилось в Python 3.8: теперь внутри выражения можно указать знак равенства. Это приводит к тому, что выражение разворачивается в текст выражения, знак равенства и результат его вычисления. Такой метод отлично подходит для самодокументирования и отладки. Ниже показан пример, иллюстрирующий разницу в поведении:

```
>>> user = "heinrich"
>>> password = "super-secret"
>>> f"Log in with: {user} and {password}"
'Log in with: heinrich and super-secret'
>>> f"Log in with: {user=} and {password=}"
"Log in with: user='heinrich' and password='super-secret'"
```

В версии Python 3.12 синтаксис f-строк был усовершенствован — новшества описаны в PEP 701 (https://peps.python.org/pep-0701/). Одно из таких улучшений — повторное использование кавычек:

```
>>> languages = ["Python", "JavaScript "]
>>> f"Two very popular languages: {", ".join(languages)}"
'Two very popular languages: Python, JavaScript
```

Обратите внимание: мы использовали двойные кавычки внутри фигурных скобок, и код прекрасно работает. Здесь вызывается метод `join()` у строки `","`, чтобы объединить элементы списка `languages` в строку с запятой и пробелом. В предыдущих версиях Python пришлось бы использовать одинарные кавычки внутри скобок: `', '`.

В число других нововведений входит возможность писать многострочные выражения, добавлять комментарии и использовать обратные косые черты (\) внутри f-строк — раньше это было запрещено.

Полный список возможностей и примеров форматирования строк можно найти в официальной документации. Это мощный инструмент, который стоит освоить.

```
>>> f"Who knew f-strings could be so powerful? {"\N{shrug}"}"
'Who knew f-strings could be so powerful? 🤷'
```

Кортежи

Последний тип неизменяемых последовательностей, который мы рассмотрим, — это кортеж. *Кортеж* (tuple) — это последовательность произвольных объектов Python. При объявлении кортежа элементы разделяются запятыми. Кортежи повсеместно используются в Python. Они позволяют реализовывать конструкции, которые в других языках программирования было бы сложно воспроизвести. Иногда кортежи используются без скобок — например, при множественном присваивании в одной строке или при возвращении нескольких объектов из функции (во многих языках функция может вернуть только один объект). Кроме того, в консоли Python кортежи можно использовать неявно, чтобы вывести несколько элементов одной инструкцией. Рассмотрим примеры для всех этих случаев:

```
>>> t = ()                              # пустой кортеж
>>> type(t)
<class 'tuple'>
>>> one_element_tuple = (42, )         # нужна запятая!
>>> three_elements_tuple = (1, 3, 5)   # скобки здесь необязательны
>>> a, b, c = 1, 2, 3                  # кортеж для множественного присваивания
>>> a, b, c                            # неявный кортеж для вывода одной инструкцией
(1, 2, 3)
>>> 3 in three_elements_tuple          # проверка вхождения
True
```

Мы используем оператор `in`, чтобы проверить, содержится ли значение в кортеже. Вдобавок этот оператор принадлежности можно применять к спискам, строкам, словарям, а также ко всем объектам коллекций и последовательностей в целом.

Чтобы создать кортеж из одного элемента, нужно поставить запятую после этого элемента. Без нее такой элемент в скобках будет считаться просто выражением, а не кортежем. Обратите внимание еще вот на что: при присваивании скобки можно опустить, то есть my_tuple = 1, 2, 3 — это то же самое, что my_tuple = (1, 2, 3).

Одна интересная возможность кортежей — это *обмен значениями двух переменных в одной строке* без использования временной переменной. Сначала рассмотрим традиционный способ:

```
>>> a, b = 1, 2
>>> c = a  # нужны временная переменная c и три строки
>>> a = b
```

```
>>> b = c
>>> a, b   # a и b поменялись местами
(2, 1)
```

А теперь — как это делается в Python:

```
>>> a, b = 0, 1
>>> a, b = b, a # питонично: меняем значения местами
>>> a, b
(1, 0)
```

Обратите внимание на строку, в которой показан питоничный способ обмена значениями двух переменных. Помните, что мы писали в главе 1? Программа на Python обычно занимает от одной пятой до одной трети размера аналогичного кода на Java или C++, и такие возможности, как обмен значениями в одной строке, тоже способствуют этому. Python элегантен, а под этим качеством в данном случае подразумевается и лаконичность.

Поскольку кортежи неизменяемы, их можно использовать в качестве ключей словаря (мы скоро поговорим об этом). На наш взгляд, кортежи — это тип встроенных данных в Python, который больше всего напоминает математический вектор. Это не означает, что они были придуманы именно для этого. Кортежи, как правило, содержат разнородную последовательность элементов, тогда как списки чаще используются для хранения однородных данных. Кроме того, кортежи обычно обрабатываются с помощью распаковки или индексации, а списки чаще перебираются в цикле.

Изменяемые последовательности

Изменяемые последовательности (mutable sequences) отличаются от неизменяемых тем, что после создания их можно модифицировать. В Python есть два типа изменяемых последовательностей: *списки* (lists) и *массивы байтов* (bytearrays).

Списки

Списки в Python похожи на кортежи, но не имеют ограничений, связанных с неизменяемостью. Обычно списки используются для хранения однородных объектов, хотя ничто не мешает вам хранить в них и неоднородные коллекции. Списки можно создавать множеством способов. Рассмотрим пример:

```
>>> []  # пустой список
[]
>>> list()                      # то же, что []
[]
>>> [1, 2, 3]                   # как и в кортежах, элементы разделены запятыми
[1, 2, 3]
```

```
>>> [x + 5 for x in [2, 3, 4]]  # Python — это магия
[7, 8, 9]
>>> list((1, 3, 5, 7, 9))       # список из кортежа
[1, 3, 5, 7, 9]
>>> list("hello")               # список из строки
['h', 'e', 'l', 'l', 'o']
```

Здесь показано, как создавать списки с помощью различных техник. Особенно обратите внимание на строку с комментарием `Python — это магия` — мы не ожидаем, что вы поймете ее сразу, особенно если не знакомы с Python. В этой строке находится выражение, которое называется *генератором списка* (list comprehension), — это мощная функциональная возможность Python, которую мы подробно разберем в главе 5. Пока мы просто хотим разжечь ваше любопытство.

Создавать списки — хорошо, но настоящая магия возникает, когда вы начинаете их использовать. Взглянем на основные методы списков:

```
>>> a = [1, 2, 1, 3]
>>> a.append(13)      # добавляем в конец списка любой элемент
>>> a
[1, 2, 1, 3, 13]
>>> a.count(1)        # сколько раз в списке встречается число 1?
2
>>> a.extend([5, 7])  # расширяем список другим списком
                      # (или любой последовательностью)
>>> a
[1, 2, 1, 3, 13, 5, 7]
>>> a.index(13)       # индекс числа 13 в списке (нумерация с нуля)
4
>>> a.insert(0, 17)   # вставляем число 17 в позицию 0
>>> a
[17, 1, 2, 1, 3, 13, 5, 7]
>>> a.pop()           # удаляем и возвращаем последний элемент
7
>>> a.pop(3)          # удаляем и возвращаем элемент на позиции 3
1
>>> a
[17, 1, 2, 3, 13, 5]
>>> a.remove(17)      # удаляем первое вхождение числа 17 из списка
>>> a
[1, 2, 3, 13, 5]
>>> a.reverse()       # меняем порядок элементов на обратный
>>> a
[5, 13, 3, 2, 1]
>>> a.sort()          # сортируем список по возрастанию
>>> a
[1, 2, 3, 5, 13]
>>> a.clear()         # очищаем список — удаляем все элементы
>>> a
[]
```

Этот код дает общее представление об основных методах списков. Мы хотим продемонстрировать их возможности на примере метода `extend()`. Вы можете расширять списки, используя любой тип последовательностей:

```
>>> a = list("hello")    # превращаем строку в список символов
>>> a
['h', 'e', 'l', 'l', 'o']
>>> a.append(100)        # добавляем в список число — список теперь может
                         # содержать разные типы
>>> a
['h', 'e', 'l', 'l', 'o', 100]
>>> a.extend((1, 2, 3))  # расширяем список элементами из кортежа
>>> a
['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3]
>>> a.extend('...')      # расширяем список символами из строки — каждый
                         # символ добавляется отдельно
>>> a
['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3, '.', '.', '.']
```

Теперь посмотрим на часто используемые операции со списками:

```
>>> a = [1, 3, 5, 7]
>>> min(a)   # минимальное значение в списке
1
>>> max(a)   # максимальное значение в списке
7
>>> sum(a)   # сумма всех значений в списке
16
>>> from math import prod
>>> prod(a)  # произведение всех значений в списке
105
>>> len(a)   # количество элементов в списке
4
>>> b = [6, 7, 8]
>>> a + b    # знак + для списков означает конкатенацию
[1, 3, 5, 7, 6, 7, 8]
>>> a * 2    # знак * тоже имеет специальное значение
[1, 3, 5, 7, 1, 3, 5, 7]
```

Обратите внимание, как легко можно вычислить сумму и произведение всех значений в списке. Функция `prod()` из модуля `math` — лишь одно из многих нововведений, появившихся в Python 3.8. Даже если вы не планируете пользоваться ею часто, будет полезно ознакомиться с модулем `math` и его функциями — они могут оказаться весьма полезными.

Последние две строки в приведенном выше примере тоже довольно интересны: они вводят понятие *перегрузки операторов*. Проще говоря, такие операторы, как `+`, `-`, `*`, `%` и др., могут означать разные действия в зависимости от контекста, в котором они применяются. Казалось бы, сложение двух списков — не самая логичная операция, но здесь знак `+` используется для объединения списков.

Аналогично знак * используется для многократного повторения содержимого списка — столько раз, сколько указано в правом операнде.

А теперь перейдем кое к чему еще более интересному. Мы хотим показать вам, насколько мощным может быть метод `sorted()` и как легко в Python добиться таких результатов, которые в других языках потребовали бы гораздо больше усилий:

```
>>> from operator import itemgetter
>>> a = [(5, 3), (1, 3), (1, 2), (2, -1), (4, 9)]
>>> sorted(a)
[(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]
>>> sorted(a, key=itemgetter(0))
[(1, 3), (1, 2), (2, -1), (4, 9), (5, 3)]
>>> sorted(a, key=itemgetter(0, 1))
[(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]
>>> sorted(a, key=itemgetter(1))
[(2, -1), (1, 2), (5, 3), (1, 3), (4, 9)]
>>> sorted(a, key=itemgetter(1), reverse=True)
[(4, 9), (5, 3), (1, 3), (1, 2), (2, -1)]
```

Код требует некоторых пояснений. Обратите внимание: `a` — это список кортежей, то есть каждый элемент `a` — кортеж (в данном случае — из двух элементов). Вызывая `sorted(my_list)`, мы получаем отсортированную версию `my_list`. В случае с кортежами из двух элементов сортировка идет сначала по первому элементу кортежа, а затем по второму, если первые совпадают. Это поведение видно в результате вызова `sorted(a)`, который дает `[(1, 2), (1, 3), ...]`. Кроме того, в Python можно указывать, по каким элементам кортежа необходимо сортировать. Обратите внимание: когда мы указываем функции `sorted`, что нужно сортировать по первому элементу каждого кортежа (`key=itemgetter(0)`), то результат получается иным: `[(1, 3), (1, 2), ...]`. Сортировка выполняется только по элементу с индексом 0 в каждом кортеже. Если мы хотим воспроизвести поведение обычного вызова `sorted(a)`, то нужно использовать `key=itemgetter(0, 1)`, что дает Python указание сортировать сначала по элементам с индексом 0, а затем по элементам с индексом 1. Сравните результаты — они совпадают.

Для полноты картины мы добавили пример сортировки только по элементам с индексом 1, а затем — ту же сортировку, но в обратном порядке. Это вас, скорее всего, впечатлит, если вы уже сталкивались с сортировкой в других языках.

Алгоритм сортировки в Python действительно очень мощный. Его написал Тим Питерс (автор «Дзен Python»). Алгоритм называется *Timsort*, является гибридом *сортировок слиянием* и *вставками* и по времени выполнения опережает большинство алгоритмов, используемых в популярных языках программирования. Timsort — устойчивый алгоритм сортировки, то есть если при сравнении несколько элементов равны, их исходный порядок сохраняется. Мы видим это в результате `sorted(a, key=itemgetter(0))`, который дал `[(1, 3), (1, 2), ...]` — порядок этих двух кортежей остался прежним, так как значение в позиции 0 у них одинаковое.

Массивы байтов (bytearrays)

Чтобы завершить обзор типов изменяемых последовательностей, уделим немного внимания типу `bytearray`. *Массивы байтов* представляют собой изменяемую версию объектов `bytes`. Они поддерживают большинство стандартных методов изменяемых последовательностей, а также большинство методов типа `bytes`. Элементы в `bytearray` — это целые числа в диапазоне [0, 256).

Для обозначения интервалов используется стандартная нотация открытых и закрытых границ. Квадратная скобка означает добавление значения, круглая — исключение. Точность обычно определяется типом граничных элементов. Например, в интервал [3, 7] входят все целые числа от 3 до 7 включительно. Интервал (3, 7) содержит только 4, 5 и 6. Элементы типа bytearray — это целые числа от 0 до 255 включительно, 256 не входит в диапазон.

Причина, по которой интервалы часто записывают таким образом, — удобство программирования. Если разбить интервал [a, b) на N последовательных интервалов, то его легко представить как объединение:

$[a, k_1) + [k_1, k_2) + [k_2, k_3) + \ldots + [k_{N-1}, b)$

Исключение точки разбиения (k_i) на одном конце и добавление на другом позволяет просто объединять и разделять диапазоны.

Рассмотрим пример с типом `bytearray`:

```
>>> bytearray()               # пустой объект bytearray
bytearray(b'')
>>> bytearray(10)             # экземпляр заданной длины, дополненный нулями
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> bytearray(range(5))       # bytearray из итерируемого объекта с целыми числами
bytearray(b'\x00\x01\x02\x03\x04')
>>> name = bytearray(b"Lina") #A – bytearray из байтов
>>> name.replace(b"L", b"l")
bytearray(b'lina')
>>> name.endswith(b'na')
True
>>> name.upper()
bytearray(b'LINA')
>>> name.count(b'L')
1
```

Как видите, существует несколько способов создать объект `bytearray`. Они полезны во многих ситуациях, например при получении данных через сокет. В таких случаях можно обойтись без постоянной конкатенации данных во время опроса, что делает `bytearray` весьма удобным. В строке `#A` мы создали массив байтов `name` из байтового литерала `b'Lina'`, чтобы показать, как `bytearray` объединяет методы и последовательностей, и строк, — это крайне полезно. Если подумать, такие объекты можно считать изменяемыми строками.

Множества

В Python предусмотрены два типа множеств: `set` и `frozenset`. Тип `set` изменяемый, тогда как `frozenset` — нет. Оба представляют собой неупорядоченные коллекции неизменяемых объектов. При выводе они отображаются как значения, разделенные запятыми и заключенные в фигурные скобки.

Хешируемость — свойство, благодаря которому объект можно использовать в качестве элемента множества или ключа словаря. Мы поговорим об этом более подробно чуть позже.

Согласно официальной документации (https://docs.python.org/3.12/glossary.html#term-hashable),

> «объект считается *хешируемым*, если его хеш-значение остается неизменным в течение всей жизни объекта и если его можно сравнивать с другими объектами. [...] Хешируемость делает объект пригодным для использования в качестве ключа словаря или элемента множества, поскольку эти структуры данных используют хеш-значение внутри себя. Большинство встроенных неизменяемых объектов Python хешируемы; изменяемые контейнеры (такие как списки или словари) — нет; неизменяемые контейнеры (такие как кортежи и `frozenset`) хешируемы только в том случае, если их элементы тоже хешируемы. Объекты, являющиеся экземплярами пользовательских классов, по умолчанию хешируемы. Они считаются неравными друг другу (кроме как по отношению к самим себе), а их хеш-значение вычисляется на основе их `id()`».

Объекты, которые считаются равными, должны иметь одинаковое хеш-значение. Множества часто используются для проверки принадлежности. В следующем примере используется оператор `in`, уже знакомый вам по подразделу «Кортежи» (см. выше):

```
>>> small_primes = set()              # пустое множество
>>> small_primes.add(2)               # добавляем по одному элементу
>>> small_primes.add(3)
>>> small_primes.add(5)
>>> small_primes
{2, 3, 5}
>>> small_primes.add(1)               # 1 — не простое число!
>>> small_primes
{1, 2, 3, 5}
>>> small_primes.remove(1)            # поэтому удаляем его
>>> 3 in small_primes                 # проверка вхождения
True
>>> 4 in small_primes
False
>>> 4 not in small_primes             # проверка отсутствия
True
>>> small_primes.add(3)               # пытаемся снова добавить 3
>>> small_primes
{2, 3, 5}                             # без изменений, дубликаты не допускаются
```

```
>>> bigger_primes = set([5, 7, 11, 13]) # более быстрый способ создания множества
>>> small_primes | bigger_primes        # оператор объединения |
{2, 3, 5, 7, 11, 13}
>>> small_primes & bigger_primes        # оператор пересечения &
{5}
>>> small_primes - bigger_primes        # оператор вычитания -
{2, 3}
```

В примере показаны два способа создания множества. В одном случае сначала создается пустое множество, а затем в него поочередно добавляются элементы. Во втором множество создается сразу, путем передачи списка чисел в конструктор, который выполняет всю работу за нас. Конечно, можно создать множество из списка, кортежа или любого другого итерируемого объекта, а затем при необходимости добавлять или удалять элементы.

О том, что такое итерируемые объекты и как устроен процесс итерации, мы поговорим в следующей главе. Пока вам достаточно знать, что итерируемые объекты — это те, которые можно перебрать по порядку.

Существует еще один способ создать множество — с помощью фигурных скобок:

```
>>> small_primes = {2, 3, 5, 5, 3}
>>> small_primes
{2, 3, 5}
```

Обратите внимание, что в примере встречаются повторяющиеся элементы — это сделано специально, чтобы акцентировать ваше внимание на том, что в результате множество не будет содержать дубликатов.

Теперь посмотрим на пример с неизменяемым вариантом типа `set` — типом `frozenset`:

```
>>> small_primes = frozenset([2, 3, 5, 7])
>>> bigger_primes = frozenset([5, 7, 11])
>>> small_primes.add(11)            # в frozenset нельзя добавлять элементы
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>> small_primes.remove(2)          # удалять тоже нельзя
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'remove'
>>> small_primes & bigger_primes    # пересечение, объединение и т. п. разрешены
frozenset({5, 7})
```

Как видите, объекты типа `frozenset` ограничены куда более сильно по сравнению с изменяемыми множествами. Тем не менее они отлично подходят для проверки принадлежности, операций объединения, пересечения и вычитания, а также используются по соображениям производительности.

Отображения: словари

Из всех встроенных типов данных Python словарь, пожалуй, самый интересный. Это единственный стандартный сопоставляющий тип, и он лежит в основе каждого объекта в Python.

Словарь сопоставляет ключи со значениями. Ключи должны быть хешируемыми объектами, а значения могут иметь любой тип. Словари относятся к изменяемым типам данных. Существует несколько способов создать словарь. Ниже показаны простые примеры пяти разных способов:

```
>>> a = dict(A=1, Z=-1)
>>> b = {"A": 1, "Z": -1}
>>> c = dict(zip(["A", "Z"], [1, -1]))
>>> d = dict([("A", 1), ("Z", -1)])
>>> e = dict({"Z": -1, "A": 1})
>>> a == b == c == d == e # Все ли они одинаковы?
True                      # Да, действительно одинаковы
```

Во всех этих примерах с ключом `"A"` сопоставлено значение `1`, а с ключом `"Z"` — значение `-1`.

Обратили внимание на двойной знак равенства? Одинарный знак используется для присваивания, а двойной — для проверки, равны ли два объекта (в примере проверяется сразу пять таких элементов). Есть и другой способ сравнения объектов — с помощью оператора `is`. Он проверяет, являются ли два объекта буквально одним и тем же объектом в памяти (то есть имеют ли одинаковый идентификатор, а не просто равны по значению). Но если нет веской причины использовать `is`, то лучше предпочесть обычное двойное равенство.

Кроме того, в приведенном выше коде использовалась одна полезная функция — `zip()`. Она, как застежка-молния, соединяет две части, поочередно выбирая по одному элементу из каждой.

```
>>> list(zip(["h", "e", "l", "l", "o"], [1, 2, 3, 4, 5]))
[('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]
>>> list(zip("hello", range(1, 6))) # тот же способ, но более питоничный
[('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]
```

Здесь мы создали одинаковые списки двумя способами: один — более явный, другой — более питоничный. На минуту забудем о том, что пришлось обернуть вызов функции `zip()` в конструктор `list()` (дело в том, что она возвращает итератор, а не список, поэтому, если мы хотим увидеть результат, нужно «истощить» итератор, преобразовав его во что-то — в данном случае в список), и сосредоточимся на результате. Обратите внимание, как `zip()` объединяет сначала первые элементы двух последовательностей, затем вторые, потом третьи и т. д. Функция `zip()` в Python работает аналогично молнии на чемодане или сумке.

А теперь вернемся к словарям и посмотрим, какие полезные методы предоставляются, чтобы облегчить работу с ними. Начнем с базовых операций:

```
>>> d = {}
>>> d["a"] = 1 # зададим пару (ключ – значение)
>>> d["b"] = 2
>>> len(d)     # сколько пар?
2
>>> d["a"]     # какое значение у ключа "a"?
1
>>> d          # как теперь выглядит словарь d?
{'a': 1, 'b': 2}
>>> del d["a"] # удалим ключ "a"
>>> d
{'b': 2}
>>> d["c"] = 3 # добавим пару "c": 3
>>> "c" in d   # наличие проверяется по ключам
True
>>> 3 in d     # значения не проверяются
False
>>> "e" in d
False
>>> d.clear()  # очистим словарь полностью
>>> d
{}
```

Обратите внимание, что доступ к элементам словаря, независимо от операции, выполняется с помощью квадратных скобок. Помните строки, списки и кортежи? Мы получали доступ к элементам по индексу, используя квадратные скобки, — это еще один пример внутренней согласованности Python.

Теперь рассмотрим три специальных объекта, называемых *представлениями словаря*: `keys`, `values` и `items`. Они обеспечивают динамическое представление содержимого словаря и автоматически обновляются при его изменении. Метод `keys()` возвращает все ключи, `values()` — все значения, а `items()` — все пары *(ключ — значение)* в виде списка кортежей из двух элементов.

Попробуем все эти операции в коде:

```
>>> d = dict(zip("hello", range(5)))
>>> d
{'h': 0, 'e': 1, 'l': 3, 'o': 4}
>>> d.keys()
dict_keys(['h', 'e', 'l', 'o'])
>>> d.values()
dict_values([0, 1, 3, 4])
>>> d.items()
dict_items([('h', 0), ('e', 1), ('l', 3), ('o', 4)])
>>> 3 in d.values()
True
>>> ("o", 4) in d.items()
True
```

Здесь есть несколько важных моментов. Обратите внимание, как создается словарь: путем итерации по результату функции `zip()`, объединяющему строку `'hello'` и числа `0`, `1`, `2`, `3`, `4`. В строке `'hello'` два символа `'l'`, и они попадают в пары с числами `2` и `3`. Однако в словаре вторая пара с ключом `'l'` (где значение `3`) перезаписывает первую (где значение `2`). Это связано с тем, что в словаре каждый ключ должен быть уникальным. Кроме того, стоит отметить: при получении любого представления словаря — `keys()`, `values()` или `items()` — порядок добавления элементов сохраняется.

Позже вы увидите, насколько важны эти представления при переборе коллекций. А пока рассмотрим и другие полезные методы, доступные благодаря словарям Python:

```
>>> d
{'h': 0, 'e': 1, 'l': 3, 'o': 4}
>>> d.popitem()         # удаляет последний добавленный элемент
('o', 4)
>>> d
{'h': 0, 'e': 1, 'l': 3}
>>> d.pop("l")          # удаляем элемент с ключом l
3
>>> d.pop("not-a-key")  # попытка удалить отсутствующий ключ вызывает KeyError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not-a-key'
>>> d.pop("not-a-key", "default-value") # с заданным по умолчанию значением
'default-value'                         # возвращается значение по умолчанию
>>> d.update({"another": "value"})      # обновляем словарь таким образом
>>> d.update(a=13)                      # или так (как вызов функции)
>>> d
{'h': 0, 'e': 1, 'another': 'value', 'a': 13}
>>> d.get("a")          # то же, что d['a'], но без ошибки при отсутствии ключа
13
>>> d.get("a", 177)     # значение по умолчанию используется,
                        # если ключ отсутствует
13
>>> d.get("b", 177)     # в данном случае возвращается значение по умолчанию
177
>>> d.get("b")          # ключа нет, возвращается None
```

Все эти методы достаточно просты для понимания, но стоит сказать несколько слов о значении `None`. Каждая функция в Python возвращает его, если явно не указан оператор `return` с другим значением. Мы подробно разберем это в главе 4. Значение `None` часто используется для указания на отсутствие значения. Кроме того, его часто добавляют в качестве значения по умолчанию для аргументов в определении функции. Менее опытные разработчики иногда пишут функции, которые возвращают либо `False`, либо `None`. Оба этих значения интерпретируются как ложные в логическом контексте, поэтому может показаться, что между ними нет разницы. На самом деле разница существенная: `False` означает, что у нас есть информация и она ложная.

`None` означает *отсутствие информации*. А это совсем не то же самое, что `False`. Представьте, что вы спрашиваете автомеханика: «Готова ли моя машина?» Между ответами «Нет, не готова» (`False`) и «Понятия не имею» (`None`) — большая разница.

Еще один полезный метод словаря, о котором стоит упомянуть, — `setdefault()`. Он работает аналогично методу `get()`, но имеет важное отличие: при вызове не только возвращает значение по ключу, но и добавляет пару «ключ — значение» в словарь, если такого ключа в нем еще нет. Ниже приведен пример:

```
>>> d = {}
>>> d.setdefault("a", 1) # ключа "a" нет, возвращается значение по умолчанию
1
>>> d
{'a': 1}          # при этом пара "ключ – значение" ("a", 1) добавлена в словарь
>>> d.setdefault("a", 5) # попробуем изменить значение
1
>>> d
{'a': 1}          # изменение не произошло, как и ожидалось
```

На этом наше небольшое путешествие по словарям заканчивается. Проверьте, насколько хорошо вы поняли данную тему, — попробуйте предсказать, как будет выглядеть словарь `d` после выполнения такой строки:

```
>>> d = {}
>>> d.setdefault("a", {}).setdefault("b", []).append(1)
```

Не переживайте, если ответ не сразу очевиден. Главное — мы хотим побудить вас экспериментировать со словарями.

Начиная с версии 3.9 в Python появился новый оператор объединения для словарей, описанный в *PEP 584*. При объединении словарей важно помнить: данная операция не является коммутативной. Это особенно заметно, если объединяемые словари содержат одинаковые ключи. Рассмотрим такой пример:

```
>>> d = {"a": "A", "b": "B"}
>>> e = {"b": 8, "c": "C"}
>>> d | e
{'a': 'A', 'b': 8, 'c': 'C'}
>>> e | d
{'b': 'B', 'c': 'C', 'a': 'A'}
>>> {**d, **e}
{'a': 'A', 'b': 8, 'c': 'C'}
>>> {**e, **d}
{'b': 'B', 'c': 'C', 'a': 'A'}
>>> d |= e
>>> d
{'a': 'A', 'b': 8, 'c': 'C'}
```

Здесь оба словаря, `d` и `e`, содержат ключ `'b'`. В словаре `d` ему соответствует значение `'B'`, а в `e` — число `8`. Это значит, что при объединении словарей с помощью оператора `|`, когда `e` стоит справа, его значение переопределяет значение из `d`. А если поменять словари местами, то произойдет обратное.

В примере также показано, как выполнить объединение словарей с помощью оператора **, то есть *распаковки словарей*. Стоит отметить, что объединение можно выполнить, используя оператор дополняющего присваивания: `d |= e`. В этом случае операция не требует промежуточной переменной и `d` изменяется напрямую. Подробности об этом нововведении можно найти в PEP 584.

На этом мы завершаем обзор встроенных типов данных. Прежде чем закончить главу, коротко обсудим другие типы данных, предоставляемые стандартной библиотекой.

Типы данных

В Python предусмотрено множество специализированных типов данных: для работы с датой и временем, контейнеры и перечисления. В стандартной библиотеке даже есть целый раздел *Data Types*, с которым стоит познакомиться поближе, — он содержит множество интересных и полезных инструментов для решения самых разных задач. Найти его можно по адресу https://docs.python.org/3/library/datatypes.html.

Далее мы кратко рассмотрим типы для работы с датой и временем, коллекции и перечисления.

Дата и время

В стандартной библиотеке Python предусмотрено несколько типов данных, предназначенных для работы с датами и временем. На первый взгляд задача может показаться простой, но временные зоны, переход на летнее и зимнее время, високосные годы и другие особенности способны сбить с толку даже опытного программиста. К тому же существует огромное количество способов, позволяющих форматировать и локализовать информацию о дате и времени, что, в свою очередь, усложняет задачу разбора таких данных. Именно поэтому многие профессиональные разработчики на Python часто используют сторонние библиотеки, чтобы получить дополнительные возможности при работе с датами и временем.

Стандартная библиотека

Начнем со стандартной библиотеки, а позже рассмотрим, какие сторонние библиотеки можно использовать для работы с датами и временем.

Основные модули стандартной библиотеки, предназначенные для работы с датами и временем, — это `datetime`, `calendar`, `zoneinfo` и `time`. Начнем с импортов, которые понадобятся нам для примеров всего этого раздела:

```
>>> from datetime import date, datetime, timedelta, timezone, UTC
>>> import time
>>> import calendar as cal
>>> from zoneinfo import ZoneInfo
```

Первый пример связан с датами:

```
>>> today = date.today()
>>> today
datetime.date(2024, 3, 19)
>>> today.ctime()
'Tue Mar 19 00:00:00 2024'
>>> today.isoformat()
'2024-03-19'
>>> today.weekday()
1
>>> cal.day_name[today.weekday()]
'Tuesday'
>>> today.day, today.month, today.year
(19, 3, 2024)
>>> today.timetuple()
time.struct_time(
    tm_year=2024, tm_mon=3, tm_mday=19,
    tm_hour=0, tm_min=0, tm_sec=0,
    tm_wday=1, tm_yday=79, tm_isdst=-1
)
```

Сначала мы получаем текущую дату с помощью `datetime.date.today()`. Она представляет собой экземпляр класса `datetime.date`. Затем получаем два разных строковых представления этой даты: одно — в формате *C* (`ctime()`), второе — в формате *ISO 8601* (`isoformat()`). Далее мы узнаем день недели с помощью метода `weekday()`, который возвращает число от 0 до 6 (от понедельника до воскресенья). Получив число, мы используем его как индекс, чтобы взять соответствующее название дня из `calendar.day_name`. Обратите внимание: в коде мы в целях удобства дали модулю `calendar` короткое имя `cal`.

Две последние инструкции показывают, как извлечь подробную информацию из объекта даты. Можно обратиться к его атрибутам `day`, `month` и `year` или вызвать метод `timetuple()`, который возвращает структуру с множеством деталей. Поскольку это объект даты, а не даты и времени, все поля, относящиеся к времени, будут равны нулю.

Теперь немного поэкспериментируем с временем:

```
>>> time.ctime()
'Tue Mar 19 21:15:23 2024'
>>> time.daylight
1
>>> time.gmtime()
time.struct_time(
    tm_year=2024, tm_mon=3, tm_mday=19,
    tm_hour=21, tm_min=15, tm_sec=53,
```

```
    tm_wday=1, tm_yday=79, tm_isdst=0
)
>>> time.gmtime(0)
time.struct_time(
    tm_year=1970, tm_mon=1, tm_mday=1,
    tm_hour=0, tm_min=0, tm_sec=0,
    tm_wday=3, tm_yday=1, tm_isdst=0
)
>>> time.localtime()
time.struct_time(
    tm_year=2024, tm_mon=3, tm_mday=19,
    tm_hour=21, tm_min=16, tm_sec=6,
    tm_wday=1, tm_yday=79, tm_isdst=0
)
>>> time.time()
1710882970.789991
```

Этот пример очень похож на предыдущий, только теперь мы работаем с временем. Сначала получаем строковое представление времени в формате C (`ctime()`), а затем проверяем, действует ли переход на летнее время (`daylight`). Функция `gmtime()` преобразует указанное количество секунд с начала эпохи в объект `struct_time` в формате UTC. Если аргумент не передан, используется текущее время.

Эпоха (epoch) — значение, от которого операционная система ведет отсчет времени. В большинстве систем, включая Unix и POSIX, за эпоху принимается 1 января 1970 года. Этот день используется как точка отсчета времени. UTC (Coordinated Universal Time — Всемирное координированное время) — основной международный стандарт, по которому регулируются часы и время в мире.

В завершение мы получаем объект `struct_time`, представляющий текущее локальное время (`localtime()`), и количество секунд, прошедших с начала эпохи (`time.time()`), в виде числа с плавающей запятой.

Теперь посмотрим пример с объектами `datetime`, которые объединяют дату и время:

```
>>> now = datetime.now()
>>> utcnow = datetime.now(UTC)
>>> now
datetime.datetime(2024, 3, 19, 21, 16, 56, 931429)
>>> utcnow
datetime.datetime(
    2024, 3, 19, 21, 17, 53, 241072,
    tzinfo=datetime.timezone.utc
)
>>> now.date()
datetime.date(2024, 3, 19)
>>> now.day, now.month, now.year
(19, 3, 2024)
>>> now.date() == date.today()
True
```

```
>>> now.time()
datetime.time(21, 16, 56, 931429)
>>> now.hour, now.minute, now.second, now.microsecond
(21, 16, 56, 931429)
>>> now.ctime()
'Tue Mar 19 21:16:56 2024'
>>> now.isoformat()
'2024-03-19T21:16:56.931429'
>>> now.timetuple()
time.struct_time(
    tm_year=2024, tm_mon=3, tm_mday=19,
    tm_hour=21, tm_min=16, tm_sec=56,
    tm_wday=1, tm_yday=79, tm_isdst=-1
)
>>> now.tzinfo
>>> utcnow.tzinfo
datetime.timezone.utc
>>> now.weekday()
1
```

Пример говорит сам за себя. Мы создаем два объекта, представляющих текущее время. Один соответствует времени UTC (`utcnow()`), второй — локальному времени (`now()`).

Из объекта `datetime`, как и из объекта `date`, можно извлекать дату, время и отдельные атрибуты. Обратите внимание на различие в значении атрибута `tzinfo`: объект `now` — *наивный* (naïve), то есть не содержит информации о временной зоне, а `utcnow` — осведомленный (aware), то есть временная зона учтена.

Объекты даты и времени классифицируются как осведомленные, если содержат информацию о временной зоне, и как наивные, если не содержат ее.

Теперь посмотрим, как в этом контексте представляются интервалы времени:

```
>>> f_bday = datetime(
    1975, 12, 29, 12, 50, tzinfo=ZoneInfo('Europe/Rome')
)
>>> h_bday = datetime(
    1981, 10, 7, 15, 30, 50, tzinfo=timezone(timedelta(hours=2))
)
>>> diff = h_bday - f_bday
>>> type(diff)
<class 'datetime.timedelta'>
>>> diff.days
2109
>>> diff.total_seconds()
182223650.0
>>> today + timedelta(days=49)
datetime.date(2024, 5, 7)
>>> now + timedelta(weeks=7)
datetime.datetime(2024, 5, 7, 21, 16, 56, 931429)
```

Здесь создаются два объекта, представляющих дни рождения Фабрицио и Генриха. На этот раз, чтобы показать альтернативный способ, мы создали *осведомленные* объекты (`aware`), содержащие информацию о временной зоне.

Существует несколько способов указать временную зону при создании объекта `datetime`. В примере выше показаны два из них: один использует объект `ZoneInfo` из модуля `zoneinfo` (введен в Python 3.9), второй — объект `timedelta`, представляющий собой продолжительность.

Затем создается объект `diff`, которому присваивается результат вычитания одного `datetime` из другого. Эта операция возвращает объект типа `timedelta`. Мы можем узнать у `diff`, сколько дней между двумя датами, а также получить общее количество секунд, соответствующее всей продолжительности. Для этого вызывается метод `total_seconds()`, который возвращает общее количество секунд. А атрибут `seconds` хранит только остаток секунд, не считая целые дни. Например, у `timedelta(days=1)` значение `seconds` будет равно `0`, а `total_seconds()` — `86400` (количество секунд в сутках).

Если прибавить объект `timedelta` к `datetime`, то результатом будет дата и время с учетом сдвига. Если же прибавить продолжительность к объекту `date`, получится новый объект `date` — логично, не правда ли?

Одна из самых непростых задач при работе с датами и временем — *синтаксический анализ строки (парсинг)*. Рассмотрим короткий пример:

```
>>> datetime.fromisoformat('1977-11-24T19:30:13+01:00')
datetime.datetime(
    1977, 11, 24, 19, 30, 13,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=3600))
)
>>> datetime.fromtimestamp(time.time())
datetime.datetime(2024, 3, 19, 21, 26, 56, 785166)
```

Мы можем легко создавать объекты `datetime` из строк в формате ISO, а также из временных меток (`timestamp`). Однако в общем случае парсинг даты из строки в произвольном формате может оказаться сложной задачей.

Сторонние библиотеки

Завершая этот раздел, хочется упомянуть несколько сторонних библиотек, с которыми вы наверняка столкнетесь при работе с датами и временем в Python:

- *dateutil* — продвинутое расширение модуля `datetime` (https://dateutil.readthedocs.io/);
- *Arrow* — делает работу с датой и временем более удобной (https://arrow.readthedocs.io/);

- *Pendulum* — упрощает работу с датами и временем (https://pendulum.eustace.io/);
- *Maya* — «библиотека для людей» (https://github.com/kennethreitz/maya);
- *Delorean* — упрощает путешествие во времени (https://delorean.readthedocs.io/);
- *pytz* — помогает определять часовые пояса мира (https://pythonhosted.org/pytz/).

Это одни из самых популярных решений, и с ними точно стоит познакомиться.

Теперь взглянем на последний пример, в котором используется сторонняя библиотека Arrow:

```
>>> import arrow
>>> arrow.utcnow()
<Arrow [2024-03-19T21:29:15.076737+00:00]>
>>> arrow.now()
<Arrow [2024-03-19T21:29:26.354786+00:00]>

>>> local = arrow.now("Europe/Rome")
>>> local
<Arrow [2024-03-19T22:29:40.282730+01:00]>
>>> local.to("utc")
<Arrow [2024-03-19T21:29:40.282730+00:00]>
>>> local.to("Europe/Moscow")
<Arrow [2024-03-20T00:29:40.282730+03:00]>
>>> local.to("Asia/Tokyo")
<Arrow [2024-03-20T06:29:40.282730+09:00]>
>>> local.datetime
datetime.datetime(
    2024, 3, 19, 22, 29, 40, 282730,
    tzinfo=tzfile('/usr/share/zoneinfo/Europe/Rome')
)
>>> local.isoformat()
'2024-03-19T22:29:40.282730+01:00'
```

Библиотека Arrow представляет собой обертку над структурами данных стандартной библиотеки и предлагает множество методов и вспомогательных функций, упрощающих работу с датами и временем. В примере видно, как просто получить текущие дату и время в итальянском часовом поясе *(Europe/Rome)*, а затем перевести их в формат UTC, в московское или токийское время. Две последние строки примера показывают, как извлечь базовый объект `datetime` из объекта Arrow, а также получить дату и время в формате ISO — очень удобно для сериализации или передачи в API.

Модуль collections

Когда встроенных универсальных контейнеров Python (таких как `tuple`, `list`, `set` и `dict`) становится недостаточно, можно воспользоваться специализированными типами контейнеров из модуля `collections` (табл. 2.1).

Таблица 2.1. Типы данных модуля collections

Тип данных	Описание
`namedtuple()`	Фабричная функция для создания подклассов кортежей с именованными полями
`deque`	Контейнер, похожий на список, с быстрым добавлением в оба конца и удалением оттуда
`ChainMap`	Класс, похожий на словарь, создает единое представление нескольких отображений
`Counter`	Подкласс словаря для подсчета хешируемых объектов
`OrderedDict`	Подкласс словаря с поддержкой переупорядочения элементов
`defaultdict`	Подкласс словаря, вызывающий готовую функцию для создания отсутствующих значений
`UserDict`	Обертка над словарем, упрощающая создание подклассов словаря
`UserList`	Обертка над списком, упрощающая создание подклассов списка
`UserString`	Обертка над строкой, упрощающая создание подклассов строки

Подробное описание каждого из этих типов и множество примеров можно найти в официальной документации по адресу https://docs.python.org/3/library/collections.html. Далее мы приведем небольшой пример, чтобы вы могли познакомиться с типами `namedtuple`, `defaultdict` и `ChainMap`.

Тип namedtuple

Тип `namedtuple` представляет собой объект, похожий на кортеж, но с возможностью обращаться к его полям по имени как к атрибутам. При этом он остается индексируемым и итерируемым, поскольку на самом деле является подклассом `tuple`. Это нечто среднее между полноценным объектом и обычным кортежем. Такой подход бывает полезен, когда вам не нужны все возможности пользовательского класса, но хочется сделать код более удобным для чтения, избавившись от обращения к полям по позициям.

Другой случай, когда `namedtuple` оказывается удобным, — если есть вероятность, что позиции элементов кортежа будут меняться при рефакторинге. Это потребует корректировки всей логики, завязанной на эти позиции, что может быть сложно и рискованно.

Например, вы работаете с информацией о левом и правом глазе пациента. Вы сохраняете значение для левого глаза на позиции 0, а для правого — на позиции 1, в обычном кортеже. Вот как это может выглядеть:

```
>>> vision = (9.5, 8.8)
>>> vision
(9.5, 8.8)
```

```
>>> vision[0]  # левый глаз (неявное позиционное обращение)
9.5
>>> vision[1]  # правый глаз (неявное позиционное обращение)
8.8
```

А теперь представьте, что регулярно работаете с объектами, описывающими зрение, и в какой-то момент дизайнер решает расширить их, добавив данные об общей (бинокулярной) остроте зрения. В результате структура объекта меняется и теперь выглядит так: *левый глаз*, *зрение в целом*, *правый глаз*.

Видите, в чем проблема? Возможно, у вас уже есть немалый объем кода, в котором предполагается, что `vision[0]` — это левый глаз (так и есть), а `vision[1]` — правый глаз (а вот это уже не так). Теперь `vision[1]` — зрение в целом, а правый глаз оказался на позиции `2`. Это означает, что нужно пересмотреть и переписать все участки кода, где вы обращаетесь к элементам по индексам, и заменить `vision[1]` на `vision[2]`. Такая переработка может быть довольно болезненной. Возможно, стоило бы изначально подойти к задаче иначе — например, использовать `namedtuple`. Сейчас покажем, что мы имеем в виду:

```
>>> from collections import namedtuple
>>> Vision = namedtuple('Vision', ['left', 'right'])
>>> vision = Vision(9.5, 8.8)
>>> vision[0]
9.5
>>> vision.left   # то же, что vision[0], но явно
9.5
>>> vision.right  # то же, что vision[1], но явно
8.8
```

Если в коде мы обращаемся к глазам как `vision.left` и `vision.right`, то при изменении структуры данных достаточно лишь обновить фабрику и способ создания экземпляров. Остальная часть кода останется неизменной:

```
>>> Vision = namedtuple('Vision', ['left', 'combined', 'right'])
>>> vision = Vision(9.5, 9.2, 8.8)
>>> vision.left      # все еще корректно
9.5
>>> vision.right     # тоже корректно (теперь это vision[2])
8.8
>>> vision.combined  # новое vision[1]
9.2
```

Вы можете видеть, что обращаться к полям по именам, а не по позициям намного удобнее. Как говорил один мудрец (отсылка к «Дзен Python»), *явное лучше неявного*. Пример может показаться немного утрированным — в конце концов, маловероятно, что опытный разработчик станет хранить такие данные в обычном кортеже. И все же вы бы удивились, узнав, насколько часто подобные ситуации встречаются в профессиональной среде — и насколько сложно исправлять и переписывать такой код.

Тип defaultdict

Один из наших любимых типов данных — `defaultdict`. Он избавляет от необходимости проверять вручную наличие ключа в словаре: при первом обращении к отсутствующему ключу он автоматически создается со значением по умолчанию, которое определяется типом, переданным при создании. В некоторых ситуациях это может оказаться очень удобным и сократить объем кода. Рассмотрим небольшой пример. Допустим, мы хотим увеличить возраст (`age`) на один год. Если значения еще нет, то мы считаем, что оно равно `0`, и прибавляем `1`:

```
>>> d = {}
>>> d["age"] = d.get("age", 0) + 1   # ключа "age" нет, берем 0 + 1
>>> d
{'age': 1}
>>> d = {"age": 39}
>>> d["age"] = d.get("age", 0) + 1   # ключ "age" есть, получаем 40
>>> d
{'age': 40}
```

Теперь посмотрим, как упростить первую часть кода с помощью `defaultdict`:

```
>>> from collections import defaultdict
>>> dd = defaultdict(int)     # int — тип значения по умолчанию (значение 0)
>>> dd["age"] += 1            # сокращенная запись для dd['age'] = dd['age'] + 1
>>> dd
defaultdict(<class 'int'>, {'age': 1})  # 1, как и ожидалось
```

Обратите внимание: при создании экземпляра `defaultdict` нужно лишь указать, что для новых ключей следует использовать тип `int`. Это значит, что при обращении к отсутствующему ключу он будет автоматически создан со значением `0` — именно такое значение по умолчанию дает тип `int`. В этом примере количество строк кода не изменилось, но читабельность явно улучшилась, а это крайне важно. Кроме того, можно использовать собственные функции, чтобы задавать значение для отсутствующих ключей, — поведение можно легко адаптировать под любые задачи. Подробнее об этом типе данных можно прочитать в официальной документации: https://docs.python.org/3/library/collections.html#collections.defaultdict.

Тип ChainMap

Тип данных `ChainMap`, появившийся в Python 3.3, работает как обычный словарь, но, как сказано в официальной документации по адресу https://docs.python.org/3/library/collections.html#collections.ChainMap, *предназначен для быстрого объединения нескольких отображений (словари, другие отображающие структуры) в единое представление*. Использовать этот способ, как правило, намного быстрее, чем создавать один словарь и выполнять на нем несколько вызовов `update()`. Тип `ChainMap` может использоваться для эмуляции вложенных областей видимости,

а также полезен в шаблонизации. Внутренние отображения хранятся в списке, доступном через атрибут `maps`. Этот список открыт: его можно просматривать и изменять напрямую. При поиске ключа `ChainMap` последовательно проходит по всем отображениям, пока не найдет нужный. А вот операции записи, изменения и удаления затрагивают только первое отображение в цепочке.

Один из типичных сценариев применения `ChainMap` — объединение пользовательских настроек с настройками по умолчанию. Рассмотрим пример:

```
>>> from collections import ChainMap
>>> default_connection = {'host': 'localhost', 'port': 4567}
>>> connection = {'port': 5678}
>>> conn = ChainMap(connection, default_connection)  # создание ChainMap
>>> conn['port']                     # порт найден в первом словаре
5678
>>> conn['host']                     # хост взят из второго словаря
'localhost'
>>> conn.maps                        # видим отображаемые словари
[{'port': 5678}, {'host': 'localhost', 'port': 4567}]
>>> conn['host'] = 'packtpub.com'    # добавляем ключ host
>>> conn.maps
[{'port': 5678, 'host': 'packtpub.com'},
{'host': 'localhost', 'port': 4567}]
>>> del conn['port']                 # удаляем информацию о порте
>>> conn.maps
[{'host': 'packtpub.com'}, {'host': 'localhost', 'port': 4567}]
>>> conn['port']                     # теперь порт берется из второго словаря
4567
>>> dict(conn)          # удобно объединить и преобразовать в обычный словарь
{'host': 'packtpub.com', 'port': 4567}
```

Это еще один пример, как Python упрощает жизнь разработчику. Вы работаете с объектом `ChainMap`, настраиваете первое (активное) отображение так, как вам нужно, а затем, когда потребуется итоговый словарь, содержащий и пользовательские, и стандартные значения, передаете `ChainMap` в конструктор `dict`. Если вы когда-нибудь писали код на Java или C++, то наверняка сможете оценить, насколько это удобно и до какой степени Python упрощает такие задачи.

Перечисления

Перечисления, находящиеся в модуле `enum`, определенно заслуживают упоминания. Они были добавлены в Python начиная с версии 3.4, и мы решили включить в книгу пример с ними, чтобы обзор типов данных был полным.

Согласно официальному определению, перечисление — это *набор символических имен (элементов), связанных с уникальными неизменяемыми значениями. Внутри перечисления элементы можно сравнивать по идентификатору и само перечисление можно перебирать в цикле.*

Предположим, вам нужно представить сигналы светофора. В коде это часто делают так:

```
>>> GREEN = 1
>>> YELLOW = 2
>>> RED = 4
>>> TRAFFIC_LIGHTS = (GREEN, YELLOW, RED)
>>> # или через словарь
>>> traffic_lights = {"GREEN": 1, "YELLOW": 2, "RED": 4}
```

На первый взгляд, в этом нет ничего необычного. Такой подход действительно довольно часто встречается. Но теперь рассмотрим альтернативу:

```
>>> from enum import Enum
>>> class TrafficLight(Enum):
...     GREEN = 1
...     YELLOW = 2
...     RED = 4
...
>>> TrafficLight.GREEN
<TrafficLight.GREEN: 1>
>>> TrafficLight.GREEN.name
'GREEN'
>>> TrafficLight.GREEN.value
1
>>> TrafficLight(1)
<TrafficLight.GREEN: 1>
>>> TrafficLight(4)
<TrafficLight.RED: 4>
```

Если на минуту отвлечься от (относительной) сложности определения класса, то становится ясно, насколько удобнее этот подход. Структура данных становится чище, а получаемый интерфейс — более мощным и выразительным. Мы советуем заглянуть в официальную документацию модуля `enum` по адресу https://docs.python.org/3/library/enum.html, чтобы изучить все возможности, — модуль точно стоит того, чтобы хотя бы познакомиться с ним.

Заключительные замечания

Итак, вы познакомились с основной частью структур данных, с которыми вам предстоит работать в Python. Мы настоятельно рекомендуем вам самостоятельно поработать с каждым из типов, представленных в этой главе. Кроме того, будет полезно просмотреть официальную документацию, чтобы хотя бы в общих чертах представить, какие еще возможности дает Python. Это знание может очень пригодиться в ситуации, где привычных типов данных окажется недостаточно для удобного представления информации.

Прежде чем перейти к главе 3, хочется уделить немного внимания некоторым важным аспектам, которые, на наш взгляд, не стоит упускать из виду.

Кеширование малых значений

В начале главы, обсуждая объекты, мы говорили о том, что при присваивании имени значению Python создает объект, устанавливает его значение и привязывает к нему имя. Мы также можем присваивать одно и то же значение разным переменным и ожидать, что будут созданы разные объекты. Например:

```
>>> a = 1000000
>>> b = 1000000
>>> id(a) == id(b)
False
```

Здесь переменные `a` и `b` ссылаются на два объекта типа `int` с одинаковым значением, но это разные объекты — что видно по разным значениям `id`. Теперь попробуем то же самое, но с меньшими значениями:

```
>>> a = 5
>>> b = 5
>>> id(a) == id(b)
True
```

Ой! Этого мы не ожидали. Почему теперь это один и тот же объект? Ведь мы явно не делали `a = b = 5`; мы присваивали значения отдельно. Ответ кроется в механизме, называемом интернированием объектов.

Интернирование объектов (object interning) — это прием оптимизации памяти, используемый в первую очередь для неизменяемых типов данных, таких как строки и числа. Суть в том, чтобы повторно использовать существующий объект, если снова требуется значение, которое уже где-то представлено в памяти.

Такой подход может заметно снизить потребление памяти и повысить производительность: нагрузка на сборщик мусора снижается, а сравнивать значения можно по идентификаторам объектов, что быстрее, чем сравнивать сами значения.

Все эти действия выполняются автоматически, скрытым образом, и в большинстве случаев вам не нужно беспокоиться о них. Но важно знать об этом поведении, особенно если вы работаете на уровне идентификаторов объектов (`id`), а не только значений.

Как выбрать подходящую структуру данных

Как вы уже увидели, Python предлагает множество встроенных типов данных. И если ваш опыт пока небольшой, то выбор подходящего типа — особенно когда речь идет о коллекциях — может оказаться непростой задачей. Представьте, что вам нужно сохранить много словарей, каждый из которых представляет клиента. В каждом словаре есть уникальный идентификатор клиента с ключом `"id"`. В какой тип коллекции стоит поместить такие словари?

Если не знать дополнительных нюансов, то ответ будет неочевиден. Нужно задать себе несколько вопросов.

- Какой доступ нам нужен?
- Какие операции придется выполнять над элементами? Как часто?
- Будет ли коллекция изменяться со временем?
- Нужно ли будет модифицировать словари клиентов?
- Какая операция с коллекцией будет выполняться чаще всего?

Если вы сможете ответить на эти вопросы, то выбор структуры данных станет понятным процессом. Например, если коллекция не будет изменяться (клиенты после создания не будут добавляться или удаляться) и перетасовываться, то можно использовать кортежи. Если же коллекция будет меняться, хорошим вариантом станет список. Но, поскольку у каждого клиента есть уникальный идентификатор, можно рассмотреть и словарь, в котором идентификаторы клиентов выступают как ключи. Вот возможные варианты:

```
customer1 = {"id": "abc123", "full_name": "Master Yoda"}
customer2 = {"id": "def456", "full_name": "Obi-Wan Kenobi"}
customer3 = {"id": "ghi789", "full_name": "Anakin Skywalker"}
# собираем их в кортеж
customers = (customer1, customer2, customer3)
# или собираем их в список
customers = [customer1, customer2, customer3]
# или, может быть, в словаре, ведь у них есть уникальный идентификатор
customers = {
    "abc123": customer1,
    "def456": customer2,
    "ghi789": customer3,
}
```

Клиенты у нас, конечно, интересные! Скорее всего, мы бы не стали использовать кортеж, если только не хотели бы подчеркнуть, что коллекция не должна изменяться, или хотя бы навести других разработчиков на эту мысль. Обычно же лучше выбрать список — он куда более гибкий.

Стоит помнить еще один важный нюанс: и кортежи, и списки — это упорядоченные коллекции. Если вы решите использовать, скажем, множество, то потеряете порядок элементов. Поэтому важно заранее понимать, насколько порядок имеет значение для вашей задачи.

Теперь о производительности. Например, в списке вставка и проверка наличия элемента (`x in my_list`) в среднем требуют O(n) времени, а в словаре обе операции занимают O(1). Но использовать словарь можно не всегда, а только в том случае, если вы можете однозначно идентифицировать каждый элемент по какому-то его свойству и это свойство хешируемо, то есть может быть ключом в `dict`.

Если вы задаетесь вопросом, что означают обозначения O(*n*) и O(1), то поищите информацию о нотации «О большое». В контексте этой книги достаточно знать следующее: если операция O*p* на структуре данных выполняется за O(*f*(*n*)), это означает, что время ее выполнения (*t*) не превышает *c* * *f*(*n*), где *c* — некая положительная константа, *n* — размер входных данных, *f* — некая функция, определяющая зависимость времени от размера входа. Проще говоря, O(...) можно рассматривать как верхнюю границу времени выполнения операции. Эту нотацию можно использовать и для оценки других ресурсов — например, объема памяти.

Еще один способ понять, подходит ли выбранная структура данных, — это посмотреть на тот код, который приходится писать для ее обработки. Если логика пишется легко, естественно и понятно, то, скорее всего, структура выбрана правильно. А вот если вы чувствуете, что код получается излишне громоздким, то, возможно, стоит переосмыслить выбор. Не имея конкретного примера, трудно давать универсальные советы. Поэтому при выборе структуры данных учитывайте как удобство работы, так и производительность и отдавайте приоритет тому, что важнее именно для вашей задачи.

Об индексации и срезах

В начале этой главы вы уже видели, как работают срезы, на примере строк. Срезы в Python можно применять ко всем последовательностям: кортежам, спискам, строкам и т. д. В случае со списками срезы можно использовать и для присваивания, хотя на практике такой прием используется редко, по крайней мере в нашем опыте. Словари и множества не поддерживают срезы.

А теперь чуть подробнее поговорим об индексации.

Есть одна особенность индексации в Python, о которой мы до сих пор не упоминали. Покажем ее на примере: как обратиться к последнему элементу коллекции?

```
>>> a = list(range(10))  # a содержит 10 элементов. Последний — 9.
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(a)               # его длина — 10 элементов
10
>>> a[len(a) - 1]        # индекс последнего равен len(a) - 1
9
>>> a[-1]                # но len(a) не нужен! Python велик!
9
>>> a[-2]                # эквивалентно len(a) - 2
8
>>> a[-3]                # эквивалентно len(a) - 3
7
```

Если в списке `a` десять элементов, то (поскольку в Python используется нумерация с нуля) первый элемент находится на позиции `0`, а последний — на позиции `9`.

В данном примере значения удобно расположены на позиции, равной своему значению: `0` — на позиции `0`, `1` — на позиции `1` и т. д.

Чтобы получить последний элемент, нужно знать длину коллекции и вычесть 1: `len(a) - 1`. Такая операция встречается настолько часто, что в Python предусмотрен специальный механизм — *обратная*, или *отрицательная индексация*. Она очень удобная, поскольку упрощает код. На рис. 2.2 показана наглядная схема того, как работает индексация в строке `"HelloThere"` (это саркастическая фраза Оби-Вана Кеноби, которой он приветствовал генерала Гривуса в фильме «Звездные войны. Эпизод III: Месть ситхов»).

Рис. 2.2. Индексация в Python

Если вы попытаетесь обратиться к индексу больше 9 или меньше –10, то Python вызовет исключение `IndexError` — чего и следовало ожидать.

Об именах

Вы, вероятно, заметили, что во многих примерах мы давали объектам простые однобуквенные имена наподобие `a`, `b`, `c`, `d` и т. д. Это вполне допустимо при отладке в консоли или когда нужно просто показать, что `a + b == 7`. Но применительно к профессиональному коду (а в принципе, и к любому другому типу кода) подобный подход считается плохой практикой. Надеемся, вы отнесетесь к таким случаям в этой книге с пониманием: мы стремились сделать примеры максимально компактными и сосредоточиться на сути.

В реальной разработке, когда вы присваиваете имена данным, стоит делать это вдумчиво — имя должно отражать смысл. Например, если у вас есть коллекция объектов `Customer`, то имя `customers` — отличный выбор. А как насчет `customers_list`, `customers_tuple` или `customers_collection`? Задумайтесь. Стоит ли привязывать имя переменной к ее типу? Мы считаем, что не стоит, если только на это нет серьезной причины. Почему? Если переменная `customers_tuple` используется в разных частях кода, а затем вы решите заменить `tuple` на `list`, то имя останется прежним, но больше не будет отражать тип, что приведет к путанице и, возможно, к необходимости рефакторинга. В качестве имен для данных лучше использовать существительные, а для функций — глаголы. Имена должны быть максимально

выразительными. Python — отличный пример языка, в котором имена подобраны удачно и последовательно. Во многих случаях, зная, что делает функция, вы можете предсказать ее имя.

Глава 2 книги *Clean Code*[1] Роберта Мартина полностью посвящена именам. Это замечательная книга, которая помогла нам улучшить стиль кода. Если вы хотите вывести свои навыки на новый уровень — обязательно прочтите ее.

Резюме

В этой главе мы познакомились со встроенными типами данных в Python. Вы увидели, как их много и чего можно достичь, просто комбинируя их.

Мы рассмотрели числовые типы, последовательности, множества, отображения, типы для работы с датами и временем, структуры из модуля `collections` и перечисления. Кроме того, мы показали вам, что все в Python — объект, и очертили разницу между изменяемыми и неизменяемыми типами. Кроме того, вы познакомились со срезами и индексацией.

Мы приводили простые примеры, но возможностей у этих типов намного больше. Поэтому обязательно загляните в официальную документацию и исследуйте все сами!

Но самое главное — практикуйтесь. Пробуйте выполнять все упражнения самостоятельно: дайте пальцам привыкнуть к коду, наработайте мышечную память и экспериментируйте как можно больше. Посмотрите, что произойдет, если разделить число на ноль, объединить разные типы, поработать со строками. Исследуйте все типы данных, применяйте их, ломайте, изучайте их методы — и освойте их в совершенстве. Ведь данные — основа всего. Если фундамент хлипкий — насколько прочным будет здание кода?

Чем дальше вы будете изучать материал книги, тем вероятнее, что время от времени встретите неточности или опечатки — в нашем коде или в своем. Что-то не заработает, появится ошибка. И это прекрасно! В процессе программирования довольно часто что-то ломается, и это нужно чинить. Лучше сразу смиритесь с тем, что ошибки будут появляться. Воспринимайте их не как проблему, а как полезный повод узнать что-то новое о языке.

Следующая глава посвящена условным конструкциям и циклам. Вы увидите, как использовать коллекции на практике и принимать решения на основе данных. Темп повествования немного ускорится. Поэтому прежде, чем переходить к главе 3, убедитесь, что уверенно ориентируетесь в темах этой главы. Повторим: работайте с удовольствием, исследуйте код, ломайте его — это отличный способ учиться.

---

[1] *Мартин Р.* Чистый код. Создание, анализ и рефакторинг. — СПб.: Питер, 2025.

3 Условные конструкции и циклы

— Скажите, пожалуйста, куда мне отсюда идти?

— Это во многом зависит от того, куда ты хочешь прийти.

Льюис Кэрролл, «Приключения Алисы в Стране чудес»

В предыдущей главе вы познакомились со встроенными типами данных Python. Теперь пришло время узнать, как программа может их использовать.

Согласно Википедии:

> «в императивном программировании *порядок выполнения* (или *управляющий поток*) — это способ упорядочения операторов, инструкций или вызовов функций программы в процессе ее выполнения».

В Python (как и в других языках) два основных способа управления потоком: *условное программирование* (его также называют *ветвлением*) и *циклы* (повторение действий). Эти способы можно комбинировать, создавая бесконечное разнообразие программ. Мы не будем описывать все возможные сочетания ветвлений и циклов, а вместо этого познакомим вас с основными конструкциями управления потоком в Python, а затем разберем несколько примеров программ. Это поможет вам лучше понять особенности практического применения условных конструкций и циклов.

Условное программирование

Условное программирование, или ветвление, — это то, чем люди, по сути, занимаются постоянно и ежедневно. Оно сводится к тому, чтобы оценить условие и принять решение о следующем действии. Например: *если горит зеленый свет — можно переходить*; *если идет дождь — беру зонт*; *если опаздываю на работу — звоню шефу*.

Оператор if

Основной инструмент условного программирования в Python — оператор `if`. Его задача — оценить выражение и, в зависимости от результата, выбрать, какой участок кода выполнять. Рассмотрим самый простой пример:

```
# conditional.1.py
late = True
if late:
    print("I need to call my manager!")
```

Здесь оператор `if` оценивает переменную `late` в логическом контексте — так же, как если бы мы написали `bool(late)`. Если результат равен `True`, то выполняется тело оператора `if`, то есть инструкции, расположенные с отступом под ним. Обратите внимание: строка `print()` смещена вправо — это значит, что она входит в область действия оператора `if`. В результате выполнения мы получим:

```
$ python conditional.1.py
I need to call my manager!
```

Поскольку `late` равно `True`, то инструкция `print()` была выполнена. Можно расширить базовый оператор `if`, добавив к нему блок `else`. Он задает альтернативный набор инструкций, которые выполняются, если выражение в `if` оценивается как `False`.

```
# conditional.2.py
late = False
if late:
    print("I need to call my manager!")    # 1
else:
    print("no need to call my manager...") # 2
```

Теперь мы задали `late = False`, поэтому результат другой:

```
$ python conditional.2.py
no need to call my manager...
```

В зависимости от значения переменной `late` выполняется либо блок `# 1`, либо блок `# 2` — *но никогда оба сразу*. Блок `# 1` выполняется, если значение `late` равно `True`; блок `# 2` — если значение этой переменной равно `False`. Поэкспериментируйте: присваивайте `late` значения `True` и `False` и наблюдайте, как меняется поведение программы.

Особый случай else: elif

Материала, который мы рассмотрели до сих пор, вполне достаточно, если нужно проверить одно условие и выбрать один из двух путей: либо выполнить блок `if`, либо альтернативный блок `else`. Но в некоторых ситуациях приходится проверять несколько условий и выбирать один из нескольких вариантов. Чтобы

показать вам подобный случай, мы используем пример с несколькими возможными исходами.

Предположим, мы создаем простой калькулятор для расчета налогов. Условия налогообложения таковы: если доход меньше 10 000 долларов, то налог платить не нужно; при доходе от 10 000 до 30 000 долларов ставка налога — 20 %; при доходе от 30 000 до 100 000 долларов ставка — 35 %; если доход превышает 100 000 долларов, то ставка — 45 %. Запишем это на Python:

```
# taxes.py
income = 15000
if income < 10000:
    tax_coefficient = 0.0   # 1
elif income < 30000:
    tax_coefficient = 0.2   # 2
elif income < 100000:
    tax_coefficient = 0.35  # 3
else:
    tax_coefficient = 0.45  # 4

print(f"You will pay: ${income * tax_coefficient} in taxes")
```

При запуске кода мы получим следующий результат:

```
$ python taxes.py
You will pay: $3000.0 in taxes
```

Разберем пример построчно. Сначала мы задаем значение переменной `income`. В примере оно равно 15 000 долларов. Далее переходим к оператору `if`. Обратите внимание: в этом примере впервые появляется `elif` — сокращение от `else if`. В отличие от обычного `else` этот блок имеет собственное условие. В выражении `income < 10000` результат — `False`, поэтому блок `# 1` не выполняется.

Управление передается к следующему условию: `elif income < 30000`. Выражение оказалось истинным, поэтому выполняется блок `# 2`. Затем Python переходит к выполнению кода за пределами всей конструкции `if/elif/elif/else` (которую мы теперь просто будем называть оператором `if`). После оператора `if` остается только одна инструкция: вызов `print()`. Она выводит информацию о том, что в этом году сумма налогов составит 3000 долларов (20 % от 15 000 долларов). Важно помнить: порядок следования блоков обязателен: сначала идет `if`, затем (при необходимости) любое количество блоков `elif`, и в конце — не более одного блока `else`.

Независимо от того, сколько строк содержится в каждом блоке, выполняется только первый, условие которого дает `True`. После этого выполнение продолжается сразу после всей конструкции. Если же ни одно из условий не дало `True` (например, `income = 200000`), то будет выполнено тело блока `else` (блок `# 4`). Этот пример помогает лучше понять, как работает блок `else`. Код внутри него выполняется только в том случае, если ни одно из предыдущих выражений `if` или `elif` не оказалось истинным.

Попробуйте менять значение переменной `income`, пока не сможете намеренно активировать любой из блоков. Кроме того, проверьте поведение программы на *граничных значениях* — там, где результат логических выражений в `if` и `elif` меняется. Тщательное тестирование границ критически важно, поскольку позволяет убедиться в правильной работе кода. Например: с какого возраста разрешено садиться за руль — с 18 или 17 лет? Мы проверяем возраст с помощью `age < 18` или `age <= 18`? Вы не представляете, сколько раз нам приходилось исправлять труднонаходимые ошибки, вызванные неправильно выбранным оператором сравнения. Поэтому обязательно поэкспериментируйте с кодом: замените в некоторых местах `<` на `<=`, подставьте в `income` граничные значения — такие как 10 000, 30 000 или 100 000, а также попробуйте любые промежуточные значения. Посмотрите, как меняется результат, и хорошо разберитесь в этой теме, прежде чем двигаться дальше.

Вложенные операторы if

Операторы `if` можно вкладывать друг в друга. Рассмотрим пример, который поможет вам понять, как работает это вложение. Предположим, что программа обнаружила ошибку. Дальнейшие действия зависят от системы оповещения. Если сообщение об ошибке должно появиться в консоли, то выводим его с помощью функции `print()`. Если оно должно прийти на электронную почту, то выбор адреса зависит от уровня серьезности ошибки. Если в качестве системы оповещения не выступают ни консоль, ни почта, то мы не знаем, что делать, и просто ничего не предпринимаем. Запишем эту логику на языке Python:

```
# errorsalert.py
alert_system = "console"     # другое значение может быть "email"
error_severity = "critical"  # другие значения: "medium" или "low"
error_message = "Something terrible happened!"

if alert_system == "console":                                  # внешний
    print(error_message)                                       # 1
elif alert_system == "email":
    if error_severity == "critical":                           # внутренний
        send_email("admin@example.com", error_message)         # 2
    elif error_severity == "medium":
        send_email("support.1@example.com", error_message)  # 3
    else:
        send_email("support.2@example.com", error_message)  # 4
```

Здесь мы видим *вложенный* оператор `if`, находящийся в теле блока `elif` *внешнего* оператора `if`. Обратите внимание: вложенность реализуется с помощью отступов — вложенный `if` сдвинут вправо относительно внешнего `elif`.

Разберем работу кода пошагово. Сначала мы задаем значения переменным `alert_system`, `error_severity` и `error_message`. Если выражение `alert_system == "console"` оказывается истинным, то выполняется блок `# 1`, и на этом выполнение кода заканчивается. Если `alert_system == "email"` оказывается истинным,

то происходит переход к вложенному `if`, где поведение определяется значением `error_severity`. Во вложенном операторе `if` серьезность ошибки влияет на то, кому отправить сообщение: если ошибка критическая — отправляем администратору (блок `# 2`); если уровень серьезности — 1, то письмо уходит на первую линию поддержки (блок `# 3`); в остальных случаях — на вторую (блок `# 4`). В этом примере функция `send_email()` не определена, поэтому при запуске кода возникнет ошибка. Но в модуле `errorsalert.py`, который можно найти в сопроводительном коде для этой книги, мы использовали небольшую хитрость: вызов `send_email()` перенаправляется на обычную функцию `print()`, чтобы вы могли экспериментировать прямо в консоли, не отправляя настоящих писем. Измените значения переменных и посмотрите, как ведет себя программа.

Тернарный оператор

В Python этот оператор также называют *условным выражением*. Он выглядит как короткая однострочная форма оператора `if` и ведет себя аналогично. Если вам просто нужно выбрать одно из двух значений в зависимости от условия, то иногда удобнее использовать тернарный оператор вместо полной конструкции `if`. Вместо такого варианта:

```
# ternary.py
order_total = 247   # фунтов стерлингов

# классическая форма if/else
if order_total > 100:
    discount = 25   # фунтов стерлингов
else:
    discount = 0    # фунтов стерлингов

print(order_total, discount)
```

можно написать так:

```
# ternary.py
# тернарный оператор
discount = 25 if order_total > 100 else 0
print(order_total, discount)
```

В простых случаях такой подход позволяет выразить логику в одной строке вместо четырех. А это бывает очень кстати: как программист, вы проводите гораздо больше времени за чтением кода, чем за его написанием, и краткость Python — действительно ценное качество.

В некоторых языках (например, в C или JavaScript) тернарный оператор выглядит еще компактнее. Код выше можно записать в такой форме:

```
discount = order_total > 100 ? 25 : 0;
```

Версия Python немного длиннее, но мы считаем, что она компенсирует это читабельностью и понятностью кода.

Понятно ли вам, как работает тернарный оператор? На самом деле все очень просто. Выражение вида `значение1 if условие else значение2` означает следующее: если условие истинно, то результатом будет `значение1`; если ложно, то результатом будет `значение2`.

Сопоставление с шаблоном

Структурное сопоставление с шаблоном — относительно новая возможность, появившаяся в Python 3.10 благодаря PEP 634 (https://peps.python.org/pep-0634). Ее разработчики вдохновлялись механизмами сопоставления, которые уже давно существуют в таких языках, как Haskell, Erlang, Scala, Elixir и Ruby.

Проще говоря, оператор `match` позволяет сравнивать значение с одним или несколькими шаблонами, а затем выполнять блок кода, связанный с первым совпавшим вариантом. Рассмотрим простой пример:

```
# match.py
day_number = 4
match day_number:
    case 1 | 2 | 3 | 4 | 5:
        print("Weekday")
    case 6:
        print("Saturday")
    case 7:
        print("Sunday")
    case _:
        print(f"{day_number} is not a valid day number")
```

Сначала мы объявляем переменную `day_number`, а затем переходим к оператору `match`. Оператор `match` пытается сопоставить значение `day_number` с одним из шаблонов, каждый из которых задается через ключевое слово `case`. В нашем примере заданы четыре шаблона. Первый: `1 | 2 | 3 | 4 | 5` — это *OR-шаблон*, он соответствует любому из указанных значений: `1`, `2`, `3`, `4` или `5`. Такой шаблон состоит из нескольких вариантов, разделенных символом `|`. Он срабатывает, если хотя бы один из вариантов совпадает (в данном случае — одно из указанных чисел). Второй и третий шаблоны — это просто целочисленные литералы `6` и `7`. Последний шаблон, `_`, — это *шаблон-подстановщик* (wildcard). Он служит «ловушкой по умолчанию», срабатывающей при любом значении, если ни один из предыдущих шаблонов не подошел. В конструкции `match` может быть только один шаблон-подстановщик, и если он есть, то должен стоять последним.

Выполняется тело только первого совпавшего блока `case`. После этого интерпретатор переходит к выполнению кода ниже конструкции `match`, не проверяя остальные шаблоны. Если ни один шаблон не совпал, то выполнение продолжается и после `match`, но ни один блок `case` не будет выполнен. В нашем примере значение `day_number` совпадает с первым шаблоном, поэтому выполняется `print("Weekday")`. Попробуйте поэкспериментировать с этим примером: измените значение переменной `day_number` и посмотрите, как ведет себя программа.

Оператор `match` внешне напоминает операторы `switch/case` таких языков, как C++ и JavaScript. Однако он гораздо мощнее. Благодаря разнообразию доступных видов шаблонов и возможности комбинировать их `match` позволяет выполнять намного больше действий, чем простой оператор `switch` в C++. Так, в Python вы можете сопоставлять последовательности (например, списки или кортежи), словари и даже пользовательские классы. Более того, прямо в шаблоне можно извлекать значения и присваивать их переменным. В рамках этой главы мы не сможем описать все, на что способен механизм сопоставления с шаблоном, но очень рекомендуем изучить учебное руководство в PEP 636 (https://peps.python.org/pep-0636) — там все показано на практических примерах.

Теперь, когда вы умеете управлять порядком выполнения кода, пора перейти к следующей теме — *циклам*.

Циклы

Если вы уже работали с циклами в других языках программирования, то можете заметить, что в Python подход к ним немного другой. Цикл позволяет повторять выполнение блока кода несколько раз, в зависимости от заданных условий. Существуют разные конструкции циклов, предназначенные для разных задач. Python свел их всего к двум основным операторам, с помощью которых можно выразить любые сценарии повторения, — `for` и `while`.

С технической точки зрения любую задачу можно реализовать с помощью любого из этих операторов, но в Python они все же предназначены для разных целей. Далее мы подробно разберем, чем различаются циклы `for` и `while` и когда их лучше использовать.

Цикл for

Этот цикл используется, когда нужно перебирать элементы последовательности — например, списка, кортежа или любой другой коллекции объектов. Начнем с простого примера и постепенно развернем его, чтобы вы могли понять, какие возможности предоставляет синтаксис Python:

```
# simple.for.py
for number in [0, 1, 2, 3, 4]:
    print(number)
```

Этот небольшой фрагмент при запуске выводит числа от 0 до 4. Тело цикла `for` (в нашем случае — строка с `print()`) выполняется один раз для каждого значения из последовательности `[0, 1, 2, 3, 4]`. В первой итерации переменной `number` присваивается первое значение — `0`, во второй — `1` и так далее до последнего значения в списке. После обработки всех элементов цикл завершается и выполнение программы продолжается со строки, следующей за циклом.

Перебор диапазона значений

Задача перебрать диапазон чисел возникает часто, и каждый раз вручную задавать список было бы утомительно. В таких случаях можно задействовать функцию `range()`. Посмотрим, как переписать предыдущий пример, используя ее:

```
# simple.for.py
for number in range(5):
    print(number)
```

Функция `range()` широко используется для создания последовательностей в программах на Python. Ее можно вызывать с одним аргументом — он будет трактоваться как граница окончания (в этом случае счет начинается с 0); с двумя аргументами — это будут начало и конец диапазона; с тремя аргументами — добавляется шаг. Ниже приведен пример:

```
>>> list(range(10))          # одно значение: от 0 до значения (не включая его)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(3, 8))        # два значения: от начала до конца (не включая его)
[3, 4, 5, 6, 7]
>>> list(range(-10, 10, 4))  # три значения: добавлен шаг
[-10, -6, -2, 2, 6]
```

Пока не стоит задумываться о том, почему мы оборачиваем `range(...)` в `list()`, — мы объясним это в главе 5, посвященной генераторам и включениям. Важно понять, что поведение функции `range()` аналогично срезам, о которых шла речь в главе 2: начальное значение добавляется, конечное — исключается, шаг — необязательный параметр, по умолчанию равен 1.

Попробуйте поэкспериментировать с параметрами вызова функции `range()` в файле `simple.for.py` и понаблюдайте за результатом.

Перебор элементов последовательности

Теперь у нас есть все необходимое, чтобы перебирать элементы последовательности, так что расширим предыдущий пример:

```
# simple.for.2.py
surnames = ["Rivest", "Shamir", "Adleman"]
for position in range(len(surnames)):
    print(position, surnames[position])
```

Этот код немного усложняет ситуацию. При запуске он выводит следующий результат:

```
$ python simple.for.2.py
0 Rivest
1 Shamir
2 Adleman
```

Разберем его по частям, *изнутри наружу* — это способ анализа, при котором мы начинаем с самого вложенного выражения и постепенно расширяем контекст. `len(surnames)` возвращает длину списка `surnames`, то есть `3`. Выражение `range(len(surnames))` превращается в `range(3)`, то есть диапазон `[0, 3)`. Это дает последовательность `(0, 1, 2)` — позиции элементов в списке. Значит, цикл `for` выполнится три раза: на первой итерации переменная `position` получит значение `0`, на второй — `1`, на третьей — `2`. А это как раз индексы списка `surnames`: на позиции `0` — `"Rivest"`, на позиции `1` — `"Shamir"`, на позиции `2` — `"Adleman"`. Если вы хотите узнать, что объединяет этих трех мужчин, то попробуйте изменить строку `print(position, surnames[position])` на `print(surnames[position][0], end="")` и добавьте вызов `print()` вне цикла. Запустите код — результат может вас удивить!

Кстати, такой стиль — с перебором индексов — напоминает код на Java или C и в Python встречается довольно редко. Почему? В Python можно напрямую перебирать элементы последовательности, не извлекая их по индексам. Перепишем пример, придав ему более питоничный вид:

```
# simple.for.3.py
surnames = ["Rivest", "Shamir", "Adleman"]
for surname in surnames:
    print(surname)
```

Цикл `for` может напрямую перебирать элементы списка `surnames` и на каждой итерации возвращает следующий элемент в порядке следования. При выполнении этот код выводит три фамилии, по одной за раз, что намного удобнее читать, чем индексы.

А если вы хотите выводить и позицию элемента? Или — что важнее — если позиция действительно нужна? Нужно ли возвращаться к форме `range(len(...))`? Нет. В таких случаях вы можете воспользоваться встроенной функцией `enumerate()`, вот так:

```
# simple.for.4.py
surnames = ["Rivest", "Shamir", "Adleman"]
for position, surname in enumerate(surnames):
    print(position, surname)
```

Этот код тоже очень показателен. Обратите внимание: `enumerate()` возвращает кортеж из двух значений — `(position, surname)` — на каждой итерации. Но даже так он остается более читабельным и эффективным, чем если бы использовалась `range(len(...))`. Вы можете передать функции `enumerate()` параметр `start`, например так: `enumerate(iterable, start=1)`. И тогда отсчет будет начинаться не с нуля, а с заданного значения. Это мелочь, но она хорошо отражает то, насколько сильно Python стремится сделать работу разработчика проще и понятнее.

Цикл `for` можно использовать для перебора списков, кортежей и вообще всего, что в Python считается *итерируемым*. Это важное понятие, и далее мы разберем его подробнее.

Итераторы и итерируемые объекты

Согласно официальной документации Python (https://docs.python.org/3.12/glossary.html#term-iterable), *итерируемый объект* (iterable) — это:

> «объект, способный возвращать свои элементы по одному. Примеры итерируемых объектов — все типы последовательностей (например, `list`, `str`, `tuple`) и некоторые непоследовательностные типы, такие как `dict` и файловые объекты».

Говоря проще, когда вы пишете:

```
for k in sequence:
    ... тело цикла ...
```

происходит следующее: цикл `for` запрашивает у `sequence` следующий элемент, получает значение, присваивает его переменной `k`, выполняет тело цикла. Затем все повторяется: снова запрашивается следующий элемент, снова выполняется тело и т. д., пока элементы не закончатся. Если последовательность пуста, то тело цикла не выполняется.

Некоторые структуры данных при итерации сохраняют порядок элементов — например, списки, кортежи, строки и словари. А вот множества порядок не гарантируют. Python позволяет перебирать любые итерируемые объекты с помощью специального типа — итераторов. *Итератор* — это объект, представляющий поток данных, из которого можно получать значения одно за другим.

На практике все это работает «за кадром». Если вы не создаете собственные итерируемые объекты или итераторы, то все происходит автоматически и вам не придется задумываться о нюансах. Если вы понимаете, как Python работает с итерациями, то писать читабельные и надежные циклы становится гораздо проще.

К теме итерации мы еще вернемся в главах 5 и 6.

Перебор нескольких последовательностей

Теперь посмотрим, как перебирать сразу две последовательности одинаковой длины, обрабатывая их элементы парами. Предположим, у нас есть два списка — с именами людей и с их возрастами. Наша задача — вывести имя и возраст для каждого человека на отдельной строке. Начнем с базового примера, который затем немного улучшим:

```
# multiple.sequences.py
people = ["Nick", "Rick", "Roger", "Syd"]
ages = [23, 24, 23, 21]
for position in range(len(people)):
    person = people[position]
    age = ages[position]
    print(person, age)
```

На данном этапе код должен быть понятен. Мы перебираем позиции (`0, 1, 2, 3`), поскольку хотим извлекать элементы из двух разных списков одновременно. При выполнении программа выдает такой результат:

```
$ python multiple.sequences.py
Nick 23
Rick 24
Roger 23
Syd 21
```

Код работает, но написан не слишком питонично. Приходится вычислять длину `people`, строить диапазон, перебирать индексы и вручную доставать значения по ним. Для некоторых структур данных извлечение по позиции может быть дорогой операцией, особенно если это, например, итератор или поток. Было бы гораздо лучше, если бы мы могли перебирать оба списка напрямую, как делаем с одной последовательностью. Попробуем улучшить пример, используя функцию `enumerate()`:

```
# multiple.sequences.enumerate.py
people = ["Nick", "Rick", "Roger", "Syd"]
ages = [23, 24, 23, 21]
for position, person in enumerate(people):
    age = ages[position]
    print(person, age)
```

Получилось лучше: теперь мы нормально перебираем список `people`, но все еще обращаемся к `ages` по индексу. А этого как раз и хочется избежать. Исправить ситуацию можно с помощью функции `zip()`, которую мы уже встречали в предыдущей главе. Применим ее:

```
# multiple.sequences.zip.py
people = ["Nick", "Rick", "Roger", "Syd"]
ages = [23, 24, 23, 21]
for person, age in zip(people, ages):
    print(person, age)
```

Такой подход намного элегантнее, чем исходный вариант. Когда цикл `for` запрашивает у `zip(sequenceA, sequenceB)` следующий элемент, то получает кортеж, который автоматически распаковывается в переменные `person` и `age`. Размер этого кортежа зависит от того, сколько последовательностей передано в `zip()`. Немного усложним предыдущий пример:

```
# multiple.sequences.unpack.py
people = ["Nick", "Rick", "Roger", "Syd"]
ages = [23, 24, 23, 21]
instruments = ["Drums", "Keyboards", "Bass", "Guitar"]
for person, age, instrument in zip(people, ages, instruments):
    print(person, age, instrument)
```

Здесь мы добавили список `instruments`. Теперь, передавая три последовательности в `zip()`, на каждой итерации цикл получает *кортеж из трех элементов*, который распаковывается в переменные `person`, `age` и `instrument`. Обратите внимание: порядок элементов в кортеже соответствует порядку последовательностей в вызове `zip()`. Код выведет следующее:

```
$ python multiple.sequences.unpack.py
Nick 23 Drums
Rick 24 Keyboards
Roger 23 Bass
Syd 21 Guitar
```

Обратите внимание, что распаковывать кортежи не обязательно. Иногда удобно обрабатывать весь кортеж целиком, не разбивая его на части. Это абсолютно допустимо:

```
# multiple.sequences.tuple.py
people = ["Nick", "Rick", "Roger", "Syd"]
ages = [23, 24, 23, 21]
instruments = ["Drums", "Keyboards", "Bass", "Guitar"]
for data in zip(people, ages, instruments):
    print(data)
```

Этот код почти идентичен предыдущему примеру. Единственное различие — вместо распаковки мы присваиваем весь кортеж переменной `data`.

Цикл while

На предыдущих страницах мы подробно рассмотрели цикл `for`. Он удобен, когда нужно перебирать элементы последовательности или коллекции. Главное, что стоит помнить при выборе конструкции цикла: цикл `for` подходит лучше, когда есть объект-контейнер (например, список) или любой другой итерируемый объект, по которому можно пройтись.

Однако бывают и другие случаи: когда нужно повторять действия, пока выполняется некоторое условие, или даже бесконечно, пока приложение не будет остановлено. В таких ситуациях не по чему итерироваться, и цикл `for` здесь не подойдет. Лучше использовать `while`.

По своей сути `while` похож на `for`: оба выполняют тело цикла многократно. Но есть ключевое различие: `for` перебирает значения, а `while` повторяет действия, пока условие истинно. Как только условие перестает выполняться, цикл завершается.

Как обычно, начнем с примера — он сразу прояснит суть. Предположим, мы хотим вывести двоичное представление положительного числа. Для этого можно воспользоваться простым алгоритмом: делим число на 2, запоминаем остаток от деления, продолжаем, пока не дойдем до нуля, а затем разворачиваем список

остатков в обратном направлении — это и будет двоичная запись числа. Рассмотрим пример для числа 6.

1. 6 / 2 = 3, остаток 0.
2. 3 / 2 = 1, остаток 1.
3. 1 / 2 = 0, остаток 1.
4. Остатки — это 0, 1, 1.
5. Разворачиваем: 1, 1, 0 — это и есть двоичное представление числа 6 — 110.

Теперь запишем этот алгоритм на Python и вычислим двоичное представление числа 39 (в двоичном виде это 100111):

```
# binary.py
n = 39
remainders = []
while n > 0:
    remainder = n % 2             # остаток от деления на 2
    remainders.append(remainder)  # сохраняем остатки
    n //= 2                       # делим n на 2
remainders.reverse()
print(remainders)
```

Здесь ключевое условие — `n > 0`: цикл продолжается, пока число больше нуля. Обратите внимание: код полностью соответствует описанному алгоритму. Пока `n` положительно, делим его на 2 и сохраняем остаток в список. Когда `n` достигает 0 — разворачиваем список остатков и получаем двоичное представление исходного значения `n`.

Код можно сделать чуть короче (и более питоничным), используя встроенную функцию `divmod()`. Функция `divmod(x, y)` возвращает кортеж из двух чисел: результат целочисленного деления `x // y` и остаток от деления `x % y`. Например, `divmod(13, 5)` вернет `(2, 3)`, поскольку `5 * 2 + 3 = 13`:

```
# binary.2.py
n = 39
remainders = []
while n > 0:
    n, remainder = divmod(n, 2)
    remainders.append(remainder)
remainders.reverse()
print(remainders)
```

Теперь мы в одной строке присваиваем `n` результат деления на 2, а остаток добавляем в список.

Если вам не нужен ручной расчет, то можете использовать встроенную функцию bin(), которая возвращает двоичное представление числа. Поэтому, если не считать учебные примеры или упражнения, в реальной практике такой код обычно не нужен.

Напомним: условие в `while` задает, продолжать ли цикл. Пока оно истинно — выполняется тело цикла, затем снова проверяется условие и все повторяется. Когда условие становится ложным, то цикл немедленно завершается и тело больше не выполняется. Если условие никогда не становится ложным, то возникает *бесконечный цикл*. Подобные циклы применяются, например, при опросе сетевых устройств: вы запрашиваете данные у сокета, если они есть — обрабатываете, немного «спите», снова запрашиваете и так до бесконечности, пока приложение работает.

Чтобы вы могли увидеть наглядное сравнение цикла `for` с `while`, адаптируем один из предыдущих примеров (`multiple.sequences.py`), добавив в него `while`:

```
# multiple.sequences.while.py
people = ["Nick", "Rick", "Roger", "Syd"]
ages = [23, 24, 23, 21]
position = 0
while position < len(people):
    person = people[position]
    age = ages[position]
    print(person, age)
    position += 1
```

Здесь мы отдельно выделили *объявление*, *условие* и *обновление* переменной `position`. Именно эти три элемента позволяют вручную реализовать логику, аналогичную `for`, с помощью `while`. Фактически все, что можно сделать с циклом `for`, можно реализовать и через `while`. Но, как видно, приходится писать немного больше «обвязки», чтобы добиться того же результата. Обратное тоже верно: конструкции `while` можно переписать с использованием `for`. Но если нет особой причины, то всегда стоит выбирать инструмент, который больше подходит для решения задачи.

Итак, подведем итог: используйте `for`, когда нужно перебрать итерируемый объект (например, список, строку, словарь и т. п.); `while` больше подойдет для случаев, когда цикл должен выполняться, пока соблюдается условие. Если вы запомните это простое различие в назначении, то никогда не ошибетесь с выбором конструкции цикла.

Теперь посмотрим, как можно изменить стандартный порядок выполнения цикла.

Операторы break и continue

Иногда в программе нужно изменить стандартный порядок выполнения цикла. Вы можете пропустить текущую итерацию (нужное количество раз) или выйти из цикла досрочно. Простой пример пропуска итерации — когда вы перебираете список, но хотите обрабатывать только те элементы, которые соответствуют какому-то условию. А если вы ищете элемент в коллекции, то логично выйти из цикла сразу, как только нужный элемент будет найден, — не выполняя лишних

итераций. Возможных сценариев множество, и далее мы разберем несколько практических примеров.

Допустим, вы хотите сделать скидку 20 % на все продукты, срок годности которых истекает сегодня. Для этого можно использовать оператор `continue`, который прерывает текущую итерацию цикла и переходит к следующей, если такая есть:

```
# discount.py
from datetime import date, timedelta
today = date.today()
tomorrow = today + timedelta(days=1)  # сегодня + 1 день — это завтра
products = [
    {"sku": "1", "expiration_date": today, "price": 100.0},
    {"sku": "2", "expiration_date": tomorrow, "price": 50},
    {"sku": "3", "expiration_date": today, "price": 20},
]
for product in products:
    print("Processing sku", product["sku"])
    if product["expiration_date"] != today:
        continue
    product["price"] *= 0.8           # то же самое, что применить скидку 20%
    print("Sku", product["sku"], "price is now", product["price"])
```

Мы импортируем объекты `date` и `timedelta`. Далее создаем список `products`. Срок годности товаров со `sku 1` и `3` истекает сегодня, то есть к ним и должна применяться скидка. Затем запускаем цикл по списку продуктов. Для каждого товара: если срок годности не истекает сегодня, выполняется `continue` и остальная часть тела цикла пропускается; если срок годности истекает сегодня, то цикл продолжает выполняться и скидка применяется. Запуск модуля `discount.py` даст следующий результат:

```
$ python discount.py
Processing sku 1
Sku 1 price is now 80.0
Processing sku 2
Processing sku 3
Sku 3 price is now 16.0
```

Как видите, две последние строки тела цикла не были выполнены для товара с `sku` номер 2, поскольку сработал `continue`.

Теперь рассмотрим пример с досрочным выходом из цикла. Допустим, вы хотите определить, есть ли хотя бы один элемент списка, который в логическом контексте оценивается как `True` (то есть `bool(element)` дает `True`). Нам нужен только один такой элемент, поэтому, как только он будет найден, дальнейший перебор можно остановить. В Python такое поведение реализуется с помощью оператора `break`. Код может выглядеть так:

```
# any.py
items = [0, None, 0.0, True, 0, 7]  # True и 7 интерпретируются как True
found = False                       # это называется флаг
for item in items:
    print("scanning item", item)
    if item:
        found = True                # обновляем флаг
        break
if found:                           # проверяем флаг
    print("At least one item evaluates to True")
else:
    print("All items evaluate to False")
```

В данном коде используется распространенный прием: сначала мы объявляем *флаг* до начала цикла, затем перебираем элементы. Если находим подходящий — обновляем флаг и выходим из цикла с помощью оператора `break`. После цикла проверяем значение флага и действуем в зависимости от результата. При запуске программа выведет:

```
$ python any.py
scanning item 0
scanning item None
scanning item 0.0
scanning item True
At least one item evaluates to True
```

Обратите внимание: как только найдено значение `True`, выполнение цикла останавливается и остальные элементы не проверяются. Оператор `break` похож на `continue` тем, что немедленно завершает выполнение тела цикла, но в отличие от `continue` полностью прерывает цикл, не переходя к следующей итерации.

Разумеется, в реальном коде нет необходимости писать такую проверку вручную — для этого есть встроенная функция any(), которая делает то же самое.

Вы можете использовать столько `continue` или `break`, сколько необходимо, — в любом месте тела цикла (`for` или `while`). И даже оба оператора в одном цикле, если того требует логика.

Особый блок else после цикла

Одна из возможностей, уникальных для Python, — это наличие блока `else` после цикла. Такой блок используется довольно редко, но иногда может быть очень полезным. Рассмотрим, как работает `else` после цикла. Если цикл завершился нормально: в `for` — по причине исчерпания итерируемого объекта,

а в `while` — потому что условие перестало выполняться, то выполняется блок `else`, если он есть. Если же цикл был прерван оператором `break`, то блок `else` не выполняется.

Рассмотрим пример. Предположим, мы перебираем список людей и ищем хотя бы одного, кто может водить автомобиль. Если никто не подходит, то мы выбросим *исключение* — то есть остановим обычное выполнение программы и сообщим об ошибке. Исключения будут описаны в главе 7, поэтому пока не беспокойтесь, если этот механизм вам незнаком. Главное — запомнить, что исключения меняют обычный порядок выполнения кода.

Сначала посмотрим, как можно реализовать такую проверку без использования `for...else`:

```
# for.no.else.py
class DriverException(Exception):
    pass
people = [("James", 17), ("Kirk", 9), ("Lars", 13), ("Robert", 8)]
driver = None
for person, age in people:
    if age >= 18:
        driver = (person, age)
        break
if driver is None:
    raise DriverException("Driver not found.")
```

Обратите внимание: здесь снова используется *флаг*-переменная. Вначале мы присваиваем переменной `driver` значение `None`, затем, если водитель найден, обновляем значение `driver`, после завершения цикла проверяем, был ли найден нужный элемент. Если водитель не найден, то выбрасывается исключение `DriverException`, которое сигнализирует, что выполнение программы не может продолжаться (водителя нет).

Теперь посмотрим, как реализовать ту же логику, но с использованием блока `else` после цикла `for`:

```
# for.else.py
class DriverException(Exception):
    pass
people = [("James", 17), ("Kirk", 9), ("Lars", 13), ("Robert", 8)]
for person, age in people:
    if age >= 18:
        driver = (person, age)
        break
else:
    raise DriverException("Driver not found.")
```

Теперь *флаг* нам больше не нужен. Исключение выбрасывается в блоке `else` как часть логики самого цикла. Это вполне логично — ведь цикл проверяет наличие подходящего элемента. Если подходящий водитель найден, то мы просто создаем

объект `driver` и остальной код может использовать его для дальнейшей обработки. Обратите внимание: код стал короче и элегантнее, поскольку логика собрана в одном месте, где ей и положено быть.

В своем выступлении Transforming Code into Beautiful, Idiomatic Python Рэймонд Хеттингер предложил удачное альтернативное название для блока else после цикла: nobreak — «если break не сработал». Если вам сложно запомнить, как работает else после for, то просто вспомните слово nobreak — и все сразу встанет на свои места.

Выражения присваивания

Прежде чем перейти к более сложным примерам, кратко обсудим одну интересную возможность, которая появилась в Python начиная с версии 3.8 благодаря PEP 572 (https://peps.python.org/pep-0572). Выражения присваивания позволяют привязать значение к имени прямо внутри выражения, в тех местах, где обычные операторы присваивания (`=`) использовать нельзя. Для этого используется не `=`, а специальный оператор `:=`, который получил прозвище *«моржовый оператор»*, поскольку визуально напоминает морду этого животного.

Операторы и выражения

Чтобы понять разницу между обычным присваиванием и выражением с присваиванием, нужно разобраться, чем оператор отличается от выражения. Согласно документации Python (https://docs.python.org/3.12/glossary.html#term-statement), *оператор* — это:

> «...часть набора инструкций (то есть «блока» кода). Оператором может быть либо выражение, либо одна из конструкций с ключевыми словами, такими как `if`, `while` или `for`».

А *выражение*, в свою очередь, — это:

> «синтаксическая конструкция, которую можно вычислить и получить результат. Другими словами, выражение — это сочетание элементов, таких как литералы, имена, доступ к атрибутам, операторы или вызовы функций, которые возвращают некое значение».

Главная особенность выражения в том, что оно возвращает значение. Выражение может быть оператором, но не всякий оператор является выражением. Например, присваивание наподобие `name = "heinrich"` — это оператор, но не выражение, то есть оно не возвращает значение. Поэтому его нельзя использовать в условии цикла `while`, в конструкции `if` или в любом другом месте, где требуется значение.

Именно поэтому консоль Python не выводит результат, когда вы присваиваете значение имени:

```
>>> name = "heinrich"
>>>
```

Присваивание — это оператор, не имеющий возвращаемого значения, которое можно было бы вывести.

Моржовый оператор (:=)

Если бы в языке не было выражений с присваиванием, то вам пришлось бы использовать два отдельных оператора: один — чтобы присвоить значение имени, другой — чтобы использовать это значение в выражении. Например, часто можно встретить такой код:

```
# walrus.if.py
remainder = value % modulus
if remainder:
    print(f"Not divisible! The remainder is {remainder}.")
```

Используя выражение с присваиванием, его можно переписать так:

```
# walrus.if.py
if remainder := value % modulus:
    print(f"Not divisible! The remainder is {remainder}.")
```

Выражения с присваиванием помогают сократить количество строк. При грамотном использовании они делают код более чистым и понятным. Далее мы разберем чуть более крупный пример, чтобы увидеть, как такое выражение упрощает цикл `while`.

Интерактивные сценарии часто просят пользователя выбрать один из вариантов. Допустим, мы пишем интерактивный сценарий для магазина мороженого, где покупатель должен выбрать вкус. Чтобы избежать путаницы при оформлении заказов, нужно убедиться, что пользователь выбирает только из предложенных вариантов. Без выражений с присваиванием код мог бы выглядеть так:

```
# menu.no.walrus.py
flavors = ["pistachio", "malaga", "vanilla", "chocolate"]
prompt = "Choose your flavor: "
print(flavors)
while True:
    choice = input(prompt)
    if choice in flavors:
        break
    print(f"Sorry, '{choice}' is not a valid option.")
print(f"You chose '{choice}'.")
```

Внимательно прочитайте этот код. Обратите внимание на условие в цикле: `while True` означает бесконечный цикл, а нам это не нужно. Мы хотим, чтобы цикл завершался, когда пользователь введет допустимый вкус (`choice in flavors`). Чтобы

добиться этого, мы используем в теле цикла условие `if` и оператор `break`. Логика управления циклом не сразу бросается в глаза. Тем не менее такая структура часто встречается, когда нужное для цикла значение можно получить только внутри самого цикла.

Функция input() очень полезна в интерактивных сценариях. Она запрашивает ввод у пользователя и возвращает введенное значение в виде строки.

Можно ли упростить этот код? Попробуем применить выражение с присваиванием:

```
# menu.walrus.py
flavors = ["pistachio", "malaga", "vanilla", "chocolate"]
prompt = "Choose your flavor: "
print(flavors)
while (choice := input(prompt)) not in flavors:
    print(f"Sorry, '{choice}' is not a valid option.")
print(f"You chose '{choice}'.")
```

Теперь условие цикла точно отражает нашу цель и код становится куда более понятным. Кроме того, он сокращается на три строки.

В этом примере вокруг выражения с присваиванием стоят скобки, поскольку оператор := имеет более низкий приоритет, чем not in. Если скобки убрать, то результат будет другим — попробуйте и посмотрите, что произойдет.

Вы уже видели примеры использования выражений с присваиванием в операторах `if` и `while`. Кроме этих случаев, они полезны и в лямбда-выражениях (см. главу 4), а также в выражениях-генераторах и включениях (см. главу 5).

Предупреждение

Появление моржового оператора в Python вызвало споры. Некоторые разработчики опасались, что он упростит написание запутанного, неканоничного для Python кода. Мы считаем, что эти опасения не до конца обоснованны. Как вы уже видели, моржовый оператор может улучшить код и сделать его более понятным. Но, как и любую мощную возможность, его можно использовать во вред — например, чтобы усложнить чтение. Поэтому советуем применять его с осторожностью. Всегда оценивайте, как этот оператор влияет на читабельность вашего кода.

Примеры программ

Теперь, когда мы разобрались с условными конструкциями и циклами, можно перейти к примерам программ, которые мы обещали в начале главы. В них будут сочетаться разные элементы, чтобы вы увидели, как использовать все эти концепции.

Генератор простых чисел

Начнем с написания кода, который формирует список простых чисел до заданного предела (включительно). Обратите внимание: мы собираемся реализовать крайне неэффективный и примитивный алгоритм для поиска простых чисел. Главное здесь — сосредоточиться на тех частях кода, которые относятся к теме этой главы. Согласно Wolfram MathWorld (https://mathworld.wolfram.com/PrimeNumber.html),

> «*простое число* — это положительное целое число $p > 1$, не имеющее других положительных делителей, кроме 1 и самого p. Более кратко: это положительное число, которое делится только на 1 и на само себя, то есть не может быть разложено на множители».

Исходя из этого определения, если взять первые десять натуральных чисел, видно, что 2, 3, 5 и 7 — простые, а 1, 4, 6, 8, 9 и 10 — нет. Чтобы определить, является ли число N простым, можно попробовать разделить его на каждое из чисел в диапазоне от 2 до N (не включая N). Если при каком-либо делении остаток будет равен нулю, то число не простое.

Чтобы сгенерировать последовательность простых чисел, будем рассматривать каждое натуральное число, начиная с 2 и вплоть до заданного предела, и проверять, является ли оно простым. Мы напишем две версии — во второй будет использована конструкция `for...else`.

```
# primes.py
primes = []          # в конце здесь будут простые числа
upto = 100           # верхняя граница, включительно
for n in range(2, upto + 1):
    is_prime = True  # флаг, новый на каждой итерации внешнего цикла
    for divisor in range(2, n):
        if n % divisor == 0:
            is_prime = False
            break
    if is_prime:     # проверка флага
        primes.append(n)
print(primes)
```

Здесь происходит довольно много интересного. Сначала создается пустой список `primes`, в который в итоге будут собраны все простые числа. Верхняя граница установлена равной `100`, и поскольку она должна входить в диапазон, то внешний цикл использует `range(2, upto + 1)` (напомним, что `range(2, upto)` *завершится на* `upto - 1`). Внешний цикл перебирает все потенциальные простые числа, то есть натуральные числа от 2 до заданного предела. Каждая итерация проверяет одно число на простоту. Внутри каждой итерации создается флаг `is_prime`, которому на старте присваивается значение `True`. Затем начинается деление текущего числа `n` на все числа от 2 до `n - 1`. Если находится делитель без остатка, это значит, что `n` — составное число, флаг устанавливается в `False` и внутренний цикл прерывается. При этом внешний цикл продолжает работу, переходя к следующему числу. Почему нужно выходить из внутреннего цикла при первом делителе?

Потому что дальнейшая проверка уже не нужна — достаточно одного делителя, чтобы убедиться, что число не простое.

По завершении внутреннего цикла проверяется флаг `is_prime`. Если он по-прежнему равен `True`, значит, делителей в диапазоне `[2, n)` не нашлось и число действительно простое. Такое число добавляется в список `primes`, и цикл продолжается, пока `n` не достигнет `100`.

При запуске этот код выводит такой результат:

```
$ python primes.py
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97]
```

Прежде чем двигаться дальше, ответим на один вопрос. Одна из итераций внешнего цикла отличается от остальных. Можете определить, какая именно и почему? Подумайте немного, вернитесь к коду, попробуйте разобраться самостоятельно, а затем продолжайте чтение.

Нашли ответ? Не переживайте, если нет. Умение понимать, что делает код, просто глядя на него, — навык, который развивается с опытом. Это важная способность для любого программиста, поэтому старайтесь тренировать ее при каждом удобном случае.

А теперь ответ: первая итерация ведет себя иначе, чем все остальные. Причина в том, что в первой итерации `n` равно `2`. Следовательно, самый внутренний цикл `for` будет выполнен по диапазону `range(2, 2)`, а это пустой диапазон. То есть тело цикла не выполнится ни разу. Попробуйте сами: напишите простой цикл `for` с таким диапазоном, добавьте `print` внутрь тела и посмотрите, что произойдет при запуске.

Мы не будем оптимизировать этот код с точки зрения алгоритма. Но применим кое-что из показанного в этой главе, чтобы немного улучшить читабельность:

```
# primes.else.py
primes = []
upto = 100
for n in range(2, upto + 1):
    for divisor in range(2, n):
        if n % divisor == 0:
            break
        else:
            primes.append(n)
print(primes)
```

Благодаря конструкции `else` после внутреннего цикла можно избавиться от флага `is_prime`. Теперь мы добавляем `n` в список `primes` только в том случае, если внутренний цикл завершился без прерывания, то есть не встретил оператор `break`. Код сократился всего на две строки, но при этом стал более простым и аккуратным, и к тому же теперь он читается легче. А это действительно важно, ведь в программировании простота и читабельность кода имеют очень большое

значение. Всегда стремитесь упростить код и сделать его понятнее. Вы сами себе скажете спасибо, когда через несколько месяцев вернетесь к нему и попробуете вспомнить, что именно тогда написали.

Пример со скидками

В этом примере мы хотим продемонстрировать прием, который очень любим, — воспользуемся *таблицей поиска*. Начнем с простого кода, в котором клиентам назначаются скидки в зависимости от значения их купона. Логику сведем к минимуму — напомним, что в данном случае нас интересуют в первую очередь условные конструкции и циклы:

```
# coupons.py
customers = [
    dict(id=1, total=200, coupon_code="F20"),  # F20: фиксированная скидка, £20
    dict(id=2, total=150, coupon_code="P30"),  # P30: процентная скидка, 30 %
    dict(id=3, total=100, coupon_code="P50"),  # P50: процентная скидка, 50 %
    dict(id=4, total=110, coupon_code="F15"),  # F15: фиксированная скидка, £15
]
for customer in customers:
    match customer["coupon_code"]:
        case "F20":
            customer["discount"] = 20.0
        case "F15":
            customer["discount"] = 15.0
        case "P30":
            customer["discount"] = customer["total"] * 0.3
        case "P50":
            customer["discount"] = customer["total"] * 0.5
        case _:
            customer["discount"] = 0.0
for customer in customers:
    print(customer["id"], customer["total"], customer["discount"])
```

Мы начинаем с того, что создаем список клиентов. У каждого указан общий заказ, код купона и идентификатор. Мы придумали четыре типа купонов: два дают фиксированную скидку, два — процентную. Используется конструкция `match` с отдельными случаями (`case`) для каждого купона и шаблоном подстановки (wildcard) для обработки недействительных кодов. Сумма скидки вычисляется и записывается в словарь клиента под ключом `"discount"`.

Вывод позволяет проверить, что код работает корректно:

```
$ python coupons.py
1 200 20.0
2 150 45.0
3 100 50.0
4 110 15.0
```

Этот код легко понять, но большое количество веток `match` захламляет логику. Чтобы добавить новый купон, нужно дописывать еще один случай и реализовы-

вать расчет скидки заново. При этом в большинстве случаев расчет очень похож. В результате код повторяется и нарушает принцип *DRY* (*Don't Repeat Yourself — «Не повторяйся»*). В таких ситуациях удобно использовать словарь в качестве таблицы поиска. Вот как это может выглядеть:

```
customers = [
    dict(id=1, total=200, coupon_code="F20"),  # F20: фиксированная скидка, £20
    dict(id=2, total=150, coupon_code="P30"),  # P30: процентная скидка, 30 %
    dict(id=3, total=100, coupon_code="P50"),  # P50: процентная скидка, 50 %
    dict(id=4, total=110, coupon_code="F15"),  # F15: фиксированная скидка, £15
]
discounts = {
    "F20": (0.0, 20.0),  # каждое значение — (процент, фиксированная сумма)
    "P30": (0.3, 0.0),
    "P50": (0.5, 0.0),
    "F15": (0.0, 15.0),
}
for customer in customers:
    code = customer["coupon_code"]
    percent, fixed = discounts.get(code, (0.0, 0.0))
    customer["discount"] = percent * customer["total"] + fixed

for customer in customers:
    print(customer["id"], customer["total"], customer["discount"])
```

Результат выполнения будет точно таким же, как и в предыдущем примере. Код сократился на две строки, но самое главное — стал гораздо более удобным для чтения: теперь тело цикла `for` занимает всего три строки и легко воспринимается. Как мы уже сказали, ключевая идея — использовать словарь как таблицу поиска. Иначе говоря, мы извлекаем параметры для расчета скидки из словаря по коду купона. Чтобы учесть случаи, когда код отсутствует в словаре, применяется `dict.get(key, default)`, позволяющий подставить значение по умолчанию.

Помимо читабельности, у этого подхода есть еще одно важное преимущество. Добавлять или удалять купоны теперь можно, не изменяя логику, — достаточно обновить *данные* в таблице поиска. В реальном приложении такую таблицу можно хранить в базе данных и предоставить интерфейс для добавления и удаления купонов прямо во время работы программы.

Чтобы рассчитать скидку, мы воспользовались простой линейной формулой. Каждая скидка представлена в словаре как пара значений: процентная и фиксированная часть (кортеж из двух элементов). Формула `процент * сумма + фиксированная часть` возвращает нужную скидку. Если процент равен 0, то остается только фиксированная сумма; если фиксированная часть равна 0 — скидка вычисляется как процент от суммы.

Этот прием близок к так называемым *таблицам диспетчеризации*, в которых значениями являются функции. Такой подход позволяет сделать код еще более гибким.

Если вам пока не до конца понятно, как работает этот механизм, не спешите читать дальше. Попробуйте поэкспериментировать: измените значения, добавьте `print()` и посмотрите, как программа ведет себя при выполнении.

Модуль itertools

Глава об итерируемых объектах, итераторах, условной логике и циклах была бы неполной без упоминания модуля `itertools`. Согласно официальной документации Python (https://docs.python.org/3.12/library/itertools.html), модуль `itertools`:

> «...реализует набор строительных блоков для итераторов, вдохновленных конструкциями из языков APL, Haskell и SML. Все они адаптированы под стиль Python.
>
> Модуль содержит компактный набор высокопроизводительных и эффективных в плане использования памяти инструментов, полезных как по отдельности, так и в сочетании друг с другом. Вместе они образуют своего рода "алгебру итераторов", позволяющую создавать специализированные инструменты лаконично и действенно — на чистом Python».

В книге мы не можем продемонстрировать все, что предлагает модуль `itertools`, поэтому рекомендуем изучить его подробнее самостоятельно. Но одно мы можем обещать: вам точно понравится. Модуль предлагает три основные категории итераторов. Для начала мы приведем по одному небольшому примеру из каждой категории.

Бесконечные итераторы

Бесконечные итераторы позволяют использовать цикл `for` как бесконечный, перебирая последовательность, которая никогда не заканчивается:

```
# infinite.py
from itertools import count
for n in count(5, 3):
    if n > 20:
        break
    print(n, end=", ")  # вместо переноса строки — запятая и пробел
```

Результат выполнения:

```
$ python infinite.py
5, 8, 11, 14, 17, 20,
```

Класс-фабрика `count` создает итератор, который просто бесконечно считает. В этом примере он начинает с `5` и на каждом шаге прибавляет `3`. Такой цикл нужно прерывать вручную, иначе программа попадет в бесконечный цикл.

Итераторы с остановкой по короткой последовательности

Эта категория особенно интересна. Она позволяет создавать итератор на основе нескольких других, объединяя их значения по определенной логике. Главное здесь то, что итоговый итератор не выдаст ошибку, если одна из входных после-

довательностей окажется короче остальных. Он просто завершится, как только самая короткая из них закончится.

Звучит немного абстрактно, поэтому приведем пример с `compress()`. Этот итератор принимает две последовательности: одну с *данными*, другую — с логическими значениями *(селекторами)*. Он возвращает только те элементы из первой последовательности, которые соответствуют `True` во второй. Например, `compress("ABC", (1, 0, 1))` вернет `"A"` и `"C"`, поскольку только эти позиции помечены единицами:

```
# compress.py
from itertools import compress
data = range(10)
even_selector = [1, 0] * 10
odd_selector = [0, 1] * 10
even_numbers = list(compress(data, even_selector))
odd_numbers = list(compress(data, odd_selector))
print(odd_selector)
print(list(data))
print(even_numbers)
print(odd_numbers)
```

Обратите внимание: `odd_selector` и `even_selector` содержат по 20 элементов, а `data` — только 10. `compress()` завершит работу, как только закончатся элементы в `data`:

```
$ python compress.py
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
[1, 3, 5, 7, 9]
```

Это быстрый и удобный способ выбрать элементы из итерируемого объекта. Код при этом остается простым. Вместо цикла `for`, который перебирал бы возвращаемые значения, используется функция `list()` — она делает то же самое, но не выполняет никаких действий в теле цикла, а просто собирает все элементы в список и возвращает его.

Комбинаторные генераторы

Третья категория итераторов в модуле `itertools` — комбинаторные генераторы. Рассмотрим простой пример с перестановками. Согласно Wolfram MathWorld (https://mathworld.wolfram.com/Permutation.html),

> «*перестановка* (также называемая числом размещений или порядком) — это перестройка элементов упорядоченного набора S, при которой создается взаимно однозначное соответствие между новым набором и самим S».

Например, для строки `"ABC"` возможно шесть перестановок: `ABC`, `ACB`, `BAC`, `BCA`, `CAB` и `CBA`.

Если множество содержит N элементов, то количество всех возможных перестановок равно $N!$ (факториал N). У строки `"ABC"` это $3! = 3 \times 2 \times 1 = 6$. Вот как это выглядит в Python:

```
# permutations.py
from itertools import permutations
print(list(permutations("ABC")))
```

При запуске этого короткого фрагмента мы получаем такой результат:

```
$ python permutations.py
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'),
('C', 'A', 'B'), ('C', 'B', 'A')]
```

Будьте осторожны при работе с перестановками. Их количество быстро растет (пропорционально факториалу числа элементов) и уже при небольших входных данных может стать огромным.

Помимо модуля itertools, существует сторонняя библиотека more-itertools, которая расширяет его возможности. Документацию можно найти по адресу https://more-itertools.readthedocs.io.

Резюме

В этой главе вы сделали еще один шаг к расширению вашего словаря Python. Вы разобрались, как управлять выполнением кода с помощью условий, а также как организовывать циклы и перебирать элементы в последовательностях и коллекциях. Это позволяет контролировать поведение программы во время выполнения и направлять ее логику так, чтобы она реагировала на изменяющиеся данные и выполняла то, что вам нужно.

Кроме того, в простых примерах вы увидели, как объединять все эти конструкции, а в завершение бегло познакомились с модулем `itertools`, который предлагает множество интересных итераторов, расширяющих возможности работы с Python.

Теперь пора переходить к функциям. Вся следующая глава посвящена именно им, и это действительно важная тема. Убедитесь, что хорошо освоили весь предыдущий материал. Мы подготовили для вас интересные примеры, так что движемся дальше!

4 Функции — строительные блоки кода

> Создавать архитектуру — значит упорядочивать. Что именно упорядочивать? Функции и объекты.
>
> *Ле Корбюзье*

В предыдущих главах вы уже видели, что в Python все является объектами, и функции не исключение. Но что же такое функция? Это *блок повторно используемого кода, предназначенный для выполнения определенной задачи или группы связанных задач*. Такие блоки можно выносить за пределы основной программы и подключать там, где они нужны. У использования функций есть множество преимуществ, и мы скоро их разберем.

Мы уверены: выражение *«Лучше один раз увидеть, чем сто раз услышать»* особенно актуально, когда нужно объяснить функции тем, кто только знакомится с этой концепцией. Поэтому взгляните на рис. 4.1.

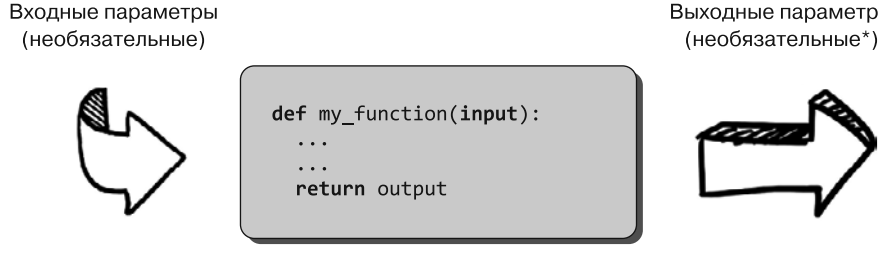


Рис. 4.1. Пример функции

Как видно на схеме, функция — это блок инструкций, оформленный как единое целое, наподобие коробки. Функции могут принимать входные параметры и возвращать значения. Оба этих элемента необязательны, как вы увидите на примерах в этой главе.

В Python функция определяется с помощью ключевого слова `def`. Сразу после него указывается имя функции, за которым следует пара круглых скобок (внутри могут быть параметры, а могут и не быть). Двоеточие (`:`) в конце строки завершает определение. Далее — с отступом в четыре пробела — идет тело функции, то есть набор инструкций, которые будут выполнены при ее вызове.

Отступ в четыре пробела — не строго обязательное требование, но именно такой отступ рекомендован в PEP 8 и на практике используется чаще всего[1].

Функция может возвращать значение, а может и не возвращать. Если она должна вернуть результат, то используется ключевое слово `return`, за которым следует нужное значение. Возможно, на рис. 4.1 вы заметили звездочку после слова «необязательные». Она стоит там потому, что в Python функция всегда что-то возвращает, даже если явно не указан оператор `return`. Если в теле функции его нет или он написан без значения, то она возвращает `None`.

Это решение в языке Python принято по нескольким причинам.

- *Простота и единообразие*. Независимо от того, возвращает функция значение явно или нет, ее поведение всегда предсказуемо.
- *Снижение сложности*. В некоторых языках различают функции (с возвращаемым значением) и *процедуры* (без него). В Python одно и то же определение может использоваться для обоих случаев — не нужны отдельные конструкции.
- *Единообразие в разных ветках кода*. Если в функции есть несколько условных ветвей и ни одна из них не вызывает `return`, то результатом все равно будет `None`. Поэтому `None` — удобное значение по умолчанию.

Этот список показывает, что даже за внешне простым решением стоит множество факторов. Именно такие продуманные нюансы лежат в основе лаконичности, ясности и гибкости Python.

Зачем нужны функции

Функции — одна из ключевых концепций и основ любой языковой конструкции. Вот несколько причин, по которым они нам так важны.

- *Уменьшают дублирование кода*. Инкапсулировав инструкцию в функцию, ее можно вызывать в разных местах программы, не повторяя реализацию.

---

[1] На самом деле четыре раза нажимать пробел довольно неудобно. К счастью, абсолютное большинство современных IDE поддерживает автозамену символа табуляции на четыре пробела. — *Примеч. науч. ред.*

- *Позволяют разбить сложную задачу на части.* Каждая часть становится отдельной функцией, что делает структуру программы более понятной.
- *Скрывают детали реализации.* Пользователю функции не нужно знать, как она устроена внутри.
- *Улучшают читабельность кода.* Программа становится более структурированной и легкой для понимания.
- *Улучшают отслеживаемость.* Легче понять, откуда приходит результат и где искать ошибки.

Теперь рассмотрим несколько примеров, чтобы лучше понять, как все это работает на практике.

Уменьшение дублирования кода

Представьте, что вы создаете программное обеспечение для научных исследований и вам нужно находить простые числа до определенного предела — как в предыдущей главе. У вас есть алгоритм, вы просто копируете его туда, где он нужен. Но однажды коллега показывает более эффективный способ вычисления простых чисел. Теперь придется пройтись по всему проекту и заменить старый код на новый.

Такой подход чреват ошибками. Можно случайно удалить кусок окружающего кода, заменить нужный фрагмент не полностью или вообще пропустить одно из мест, где используется расчет простых чисел. В итоге одна и та же операция может выполняться по-разному в различных частях программы, что ведет к несогласованности. А если речь не о замене на улучшенную версию, а, например, о правке ошибки — последствия могут быть куда более серьезными. К тому же, если переменные в старом и новом коде называются по-разному, замена усложняется еще сильнее.

Чтобы избежать всего этого, проще один раз написать функцию `get_prime_numbers(upto)` и вызывать ее везде, где нужно получить список простых чисел. Когда коллега принесет новую реализацию, будет достаточно заменить содержимое функции — остальная часть программы останется прежней, ведь она просто вызывает одну и ту же функцию.

Код станет короче, а главное — избавится от несоответствий между старыми и новыми реализациями. К тому же вы с меньшей вероятностью допустите ошибку, связанную с копированием и вставкой или с невнимательностью.

Разбиение сложной задачи

Функции позволяют разбивать длинные или сложные задачи на более мелкие части. Это дает сразу несколько преимуществ: код становится проще читать, удобнее тестировать и легче использовать повторно.

Простой пример: вы готовите отчет. Ваш код должен получить данные из источника, проанализировать их, отфильтровать, привести в нужный вид, а затем запустить серию алгоритмов, чтобы получить результат, который и пойдет в отчет. Часто в таких случаях пишут одну большую функцию наподобие `do_report(data_source)`, внутри которой — сотни строк кода, выполняющихся перед тем, как будет создан финальный отчет.

Неопытные программисты, не привыкшие к структурированному коду, могут создавать такие функции длиной несколько сотен строк. Их тяжело читать, трудно отследить, где заканчивается один этап и начинается другой. Намного лучше подойти к задаче иначе:

```
# data.science.example.py
def do_report(data_source):

    # получить и подготовить данные
    data = fetch_data(data_source)
    parsed_data = parse_data(data)
    filtered_data = filter_data(parsed_data)
    polished_data = polish_data(filtered_data)

    # запустить алгоритмы на данных
    final_data = analyse(polished_data)

    # создать и вернуть отчет
    report = Report(final_data)
    return report
```

Конечно, этот пример вымышленный. Но согласитесь — просматривать такой код куда проще. Если что-то пошло не так, то можно быстро проверить результат на каждом этапе внутри функции `do_report()`. А если нужно временно исключить какой-то шаг, то достаточно просто закомментировать соответствующую строку. С таким кодом работать гораздо удобнее.

Сокрытие деталей реализации

Предыдущий пример отлично подходит для того, чтобы обсудить еще один важный момент. Просматривая код функции `do_report()`, можно получить довольно хорошее представление о происходящем, даже не заглядывая в реализацию. Это становится возможным благодаря тому, что функции скрывают детали реализации.

А значит, если нам не нужно вникать в каждую мелочь, то мы и не обязаны этого делать, в отличие от ситуации, когда `do_report()` представляет собой одну длинную монолитную функцию. Тогда, чтобы понять, что происходит, пришлось бы прочитать и осмыслить каждую строчку. А если код разбит на более мелкие функции, то нам уже не нужно вчитываться в каждую из них — достаточно по-

нять, что делает каждая часть в целом[1]. Это приведет к сокращению времени, затрачиваемого на чтение кода, что очень важно, поскольку (как мы уже неоднократно упоминали) профессиональным программистам читать код приходится гораздо чаще, чем писать его.

Улучшение читабельности

Иногда программисты не видят смысла выносить в отдельную функцию код, который занимает всего пару строк. Рассмотрим пример, который показывает, почему все-таки стоит это делать. Представьте, что вам нужно перемножить две матрицы — как в следующем примере:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 & 1 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 9 & 3 \\ 23 & 7 \end{pmatrix}$$

Какой вариант вы бы предпочли прочитать?

```
# matrix.multiplication.nofunc.py
a = [[1, 2], [3, 4]]
b = [[5, 1], [2, 1]]
c = [
    [sum(i * j for i, j in zip(r, c)) for c in zip(*b)] for r in a
]
```

Или такой?

```
# matrix.multiplication.func.py
def matrix_mul(a, b):
    return [
        [sum(i * j for i, j in zip(r, c)) for c in zip(*b)]
        for r in a
    ]
a = [[1, 2], [3, 4]]
b = [[5, 1], [2, 1]]
c = matrix_mul(a, b)
```

Во втором случае сразу понятно, что `c` — это результат перемножения `a` и `b`, и сам код читается гораздо легче. Если не нужно менять логику умножения, то даже не придется вдаваться в детали реализации `matrix_mul()`. В результате читабельность кода улучшается, а вот в первом варианте пришлось бы тратить время на то, чтобы понять, как устроена эта сложная конструкция с включением.

О включениях мы подробно поговорим в главе 5.

[1] Главное при этом не забывать давать функциям лаконичные, но говорящие названия, отражающие назначение, — это существенно упростит чтение кода. — *Примеч. науч. ред.*

Улучшение отслеживаемости

Представьте, что вы пишете код для сайта электронной торговли. На разных страницах отображаются цены товаров. В базе данных цены хранятся без учета НДС, однако на сайте их нужно показывать с НДС 20 %. Вот несколько способов вычислить цену с НДС по цене без НДС:

```
# vat.nofunc.py
price = 100  # цена без НДС
final_price1 = price * 1.2
final_price2 = price + price / 5.0
final_price3 = price * (100 + 20) / 100.0
final_price4 = price + price * 0.2
```

Все четыре варианта расчета — корректны. Мы встречали их в реальных проектах, над которыми работали в разное время.

Теперь представьте, что начинаете продавать товары в других странах — и в некоторых из них действует иной процент НДС. Значит, код по всему сайту нужно переработать, чтобы сделать расчет НДС динамическим.

Как отследить все места, где выполняется расчет НДС? Современная разработка — это командная задача, и мы не можем быть уверены, что расчет НДС во всем проекте выполняется одним и тем же способом. В итоге отследить такие участки кода становится сложно.

Решение простое. Напишем функцию, которая принимает два аргумента, `vat` и `price` (без НДС), и возвращает цену с учетом НДС:

```
# vat.function.py
def calculate_price_with_vat(price, vat):
    return price * (100 + vat) / 100
```

Теперь эту функцию можно импортировать и использовать везде, где нужно показать цену с НДС. А если потребуется найти все такие места, то достаточно выполнить поиск по имени `calculate_price_with_vat`.

В примере предполагается, что price — это цена без НДС, а vat — процентная ставка (например, 19, 20 или 23).

Области видимости и разрешение имен

В главе 1 мы уже обсуждали области видимости и пространства имен. В данном разделе мы расширим эту тему. Наконец-то мы можем говорить о ней в контексте функций — так разобраться будет проще. Начнем с простого примера:

```
# scoping.level.1.py
def my_function():
```

```
    test = 1  # это объявлено в локальной области видимости функции
    print("my_function:", test)

test = 0      # это объявлено в глобальной области видимости
my_function()
print("global:", test)
```

В примере имя `test` определено в двух местах — в разных областях видимости. Одно — в глобальной области (`test = 0`), другое — в локальной области функции `my_function()` (`test = 1`). При выполнении этот код выдает следующий результат:

```
$ python scoping.level.1.py
my_function: 1
global: 0
```

Ясно, что строка `test = 1` в теле функции затеняет глобальное определение `test = 0`. В глобальной области `test` по-прежнему равен `0` — как видно из вывода программы. Но внутри функции мы снова создаем имя `test` и привязываем его к числу `1`. Таким образом, оба имени `test` существуют параллельно: одно в глобальной области, ссылается на `int` со значением `0`, другое — внутри `my_function()`, ссылается на `int` со значением `1`. Теперь закомментируйте строку `test = 1`. Python будет искать имя `test` в ближайшей внешней области видимости (вспомните правило LEGB: локальная, замыкающая, глобальная, встроенная (см. главу 1)). В этом случае программа дважды выведет значение `0`. Попробуйте применить это в своем коде.

А теперь перейдем к более сложному примеру с вложенными функциями:

```
# scoping.level.2.py
def outer():
    test = 1      # внешний уровень области видимости

    def inner():
        test = 2  # внутренний уровень области видимости
        print("inner:", test)

    inner()
    print("outer:", test)

test = 0          # глобальная область видимости
outer()
print("global:", test)
```

В этом коде двойное затенение: один уровень — внутри функции `outer()`, а второй — внутри вложенной функции `inner()`. Если выполнить код, то результат будет такой:

```
$ python scoping.level.2.py
inner: 2
outer: 1
global: 0
```

Закомментируйте строку `test = 1`. Как вы думаете, какой будет результат? Когда выполнение дойдет до строки `print('outer:', test)`, Python начнет искать имя `test` в ближайшей внешней области видимости — и найдет значение `0`, а не `1`. Закомментируйте строку `test = 2` — это поможет лучше понять, как работает правило LEGB. Убедитесь, что правильно понимаете происходящее, прежде чем читать дальше.

Еще один важный момент: в Python можно определять одну функцию внутри другой. Имя `inner()` создается в пространстве имен функции `outer()`, точно так же, как любое другое имя.

Операторы global и nonlocal

В предыдущем примере мы можем изменить поведение затенения имени `test`, воспользовавшись одним из двух специальных операторов — `global` или `nonlocal`. Как вы можете видеть, когда мы пишем `test = 2` внутри функции `inner()`, это не влияет на `test` в функции `outer()` и в глобальной области.

Мы можем получить доступ к этим именам для чтения, если используем их во вложенной области, где они не определены. Но мы не можем их изменить, поскольку при присваивании Python создает новое имя в текущей области видимости.

Чтобы изменить такое поведение, можно использовать оператор `nonlocal`. Согласно официальной документации Python,

> «оператор `nonlocal` указывает, что перечисленные идентификаторы должны ссылаться на ранее связанные переменные в ближайшей замыкающей области видимости, исключая глобальную».

Вставим этот оператор в тело функции `inner()` и посмотрим, что произойдет:

```
# scoping.level.2.nonlocal.py
def outer():
    test = 1      # внешний уровень области видимости

    def inner():
        nonlocal test
        test = 2  # ближайшая внешняя область видимости
                  # (в данном случае – 'outer')
        print("inner:", test)

    inner()
    print("outer:", test)

test = 0          # глобальная область видимости
outer()
print("global:", test)
```

Обратите внимание: в теле `inner()` мы объявили имя `test` как `nonlocal`. Результат выполнения будет таким:

```
$ python scoping.level.2.nonlocal.py
inner: 2
outer: 2
global: 0
```

Объявляя `test` как `nonlocal` внутри функции `inner()`, мы фактически привязываем это имя к переменной `test`, объявленной во внешней функции `outer()`. А если убрать строку `nonlocal test` из `inner()` и попытаться использовать ее в `outer()`, то возникнет ошибка `SyntaxError`, поскольку оператор `nonlocal` работает только с замыкающими областями видимости, но не с глобальной.

А можно ли получить доступ на запись к `test = 0` в глобальной области? Да, для этого нужно использовать оператор `global`:

```
# scoping.level.2.global.py
def outer():
    test = 1      # внешний уровень области видимости

    def inner():
        global test
        test = 2  # глобальная область видимости
        print("inner:", test)

    inner()
    print("outer:", test)

test = 0          # глобальная область видимости
outer()
print("global:", test)
```

Теперь мы объявили имя `test` как глобальное — это означает, что оно будет связано с переменной, определенной в глобальной области (`test = 0`). Запуск этого кода приводит к следующему результату:

```
$ python scoping.level.2.global.py
inner: 2
outer: 1
global: 2
```

Это доказывает, что теперь присваивание `test = 2` влияет именно на имя в глобальной области. И такой подход будет работать и в функции `outer()`, ведь в этом случае мы ссылаемся все на ту же глобальную область.

Попробуйте выполнить этот код сами и посмотрите, что изменится. Изучите тему областей видимости и разрешения имен как следует — она действительно важна. Бонусный вопрос: как вы думаете, что произойдет, если в предыдущем примере определить `inner()` вне `outer()`?

Входные параметры

В начале главы мы уже упоминали, что функция может принимать входные параметры. Прежде чем углубляться в разные типы параметров, стоит убедиться, что вам ясен термин «передача аргумента функции». Есть три ключевых момента, которые важно запомнить:

- передача аргумента — это просто присваивание объекта имени локальной переменной;
- присваивание нового объекта аргументу внутри функции не влияет на вызывающий код;
- а изменение переданного изменяемого объекта — влияет.

Прежде чем двигаться дальше, немного уточним терминологию. Согласно официальной документации Python,

> «параметры — это имена, указанные в определении функции, а аргументы — это значения, которые фактически передаются функции при ее вызове. Параметры определяют, какие типы аргументов может принимать функция».

Мы будем стараться точно различать параметры и аргументы, но стоит понимать, что в разговорной речи и даже в некоторых источниках эти термины могут использоваться как синонимы. А теперь перейдем к примерам.

Передача аргументов

Посмотрите на следующий код. Сначала мы объявляем имя `x` в глобальной области, затем определяем функцию `func(y)` и, наконец, вызываем ее, передавая в нее `x`:

```
# key.points.argument.passing.py
x = 3
def func(y):
    print(y)

func(x) # выводит: 3
```

Когда вызывается `func(x)`, внутри локальной области функции создается имя `y`, которое ссылается на тот же объект, что и `x`. Эта идея наглядно показана на рис. 4.2 (пример написан на Python 3.11, однако не стоит беспокоиться — рассматриваемый функционал языка не изменился).

Правая часть рис. 4.2 отражает состояние программы в момент завершения выполнения — после того, как функция `func()` вернула `None`. Обратите внимание на столбец Frames. В глобальной области (Global frame) видно два имени: `x` и `func`. Первое ссылается на объект типа `int` со значением `3`, второе — на объект функ-

ции. Чуть ниже, в прямоугольнике с заголовком `func`, отображается локальная область видимости функции. Здесь определено только одно имя — `y`. Мы вызвали `func(x)` (строка 6 в левой части рисунка), поэтому имя `y` ссылается на тот же объект, что и `x`.

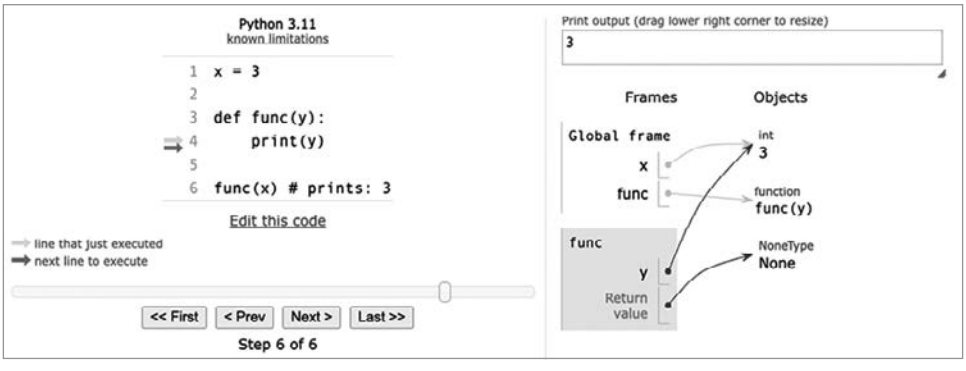


Рис. 4.2. Как работает передача аргументов (визуализация с помощью Python Tutor)

Именно так устроена передача аргументов в Python «за кадром». Если бы мы вместо `y` в теле функции использовали имя `x`, то результат был бы точно таким же. Но в этом случае внутри функции был бы локальный `x`, а снаружи — глобальный `x`, как мы уже обсуждали в разделе «Области видимости и разрешение имен» ранее в этой главе.

Подводя итог: функция при вызове создает в своей локальной области имена, указанные в определении параметров. А вызывая функцию, мы сообщаем Python, на какие объекты эти имена должны ссылаться.

Присваивание имени параметра

Присваивание значения параметру не влияет на вызывающий код. Поначалу это может показаться неочевидным, поэтому рассмотрим пример:

```
# key.points.assignment.py
x = 3

def func(x):
    x = 7  # определение локальной переменной x, глобальная не меняется

func(x)
print(x)   # выводит: 3
```

Здесь при вызове `func(x)` выполняется строка `x = 7`, но внутри локальной области видимости функции `func()`. Имя `x` теперь ссылается на целое число 7, не затрагивая глобальную переменную `x`.

Изменение изменяемого объекта

Изменение изменяемого объекта влияет на вызывающий код. Это важный момент. Может показаться, что Python *ведет* себя по-разному с изменяемыми объектами, однако на самом деле поведение языка строго единообразно. Рассмотрим пример:

```
# key.points.mutable.py
x = [1, 2, 3]

def func(x):
    x[1] = 42  # это изменяет аргумент x!

func(x)
print(x)       # выводит: [1, 42, 3]
```

Здесь мы изменили исходный объект. Но если подумать, то ничего странного в этом поведении нет. Когда вызывается `func(x)`, имя `x` в пространстве имен функции ссылается на тот же объект, что и глобальное имя `x`. В теле функции мы не меняем саму переменную `x` в вызывающем коде, поскольку не переназначаем, на что она указывает. Мы просто получаем доступ к элементу с индексом 1 внутри объекта и меняем значение по этому индексу.

Напомним второй пункт перечня в начале раздела «Входные параметры»: *присваивание нового объекта аргументу внутри функции не влияет на вызывающий код*. Если вы это поняли, то следующий пример не должен удивить:

```
# key.points.mutable.assignment.py
x = [1, 2, 3]

def func(x):
    x[1] = 42             # это изменяет исходный аргумент x!
    x = "something else"  # теперь x ссылается на новый объект строки

func(x)
print(x)                  # все равно выводит: [1, 42, 3]
```

Обратите внимание на две ключевые строки. Сначала, как и раньше, мы получаем доступ к объекту, переданному извне, и меняем его элемент с индексом 1 на число 42. Затем переназначаем имя `x` внутри функции, чтобы оно указывало на строку `'something else'`. При этом вызывающий код не затрагивается и результат получается точно такой же, как в предыдущем примере.

Поэкспериментируйте немного. Добавьте `print()` и вызовы `id()`, чтобы отследить, что именно происходит. Это один из ключевых аспектов Python, и его нужно понимать абсолютно четко — иначе вы рискуете получить трудноуловимые ошибки. Кстати, сайт http://www.pythontutor.com очень поможет — он дает наглядное представление о происходящем в памяти.

А теперь, поговорив о том, как применяются входные параметры, перейдем к разным способам передачи аргументов в функции.

Передача аргументов

Python предоставляет четыре способа передачи аргументов в функции:

- использование позиционных аргументов;
- использование именованных (ключевых) аргументов;
- распаковку итерируемого объекта;
- распаковку словарей.

Рассмотрим каждый из них.

Позиционные аргументы

При вызове функции каждому позиционному аргументу соответствует параметр в том же *порядке*, в котором они указаны в определении функции:

```
# arguments.positional.py
def func(a, b, c):
    print(a, b, c)

func(1, 2, 3) # выводит: 1 2 3
```

Это наиболее распространенный способ передачи аргументов. (А в некоторых языках программирования это вообще единственный допустимый способ.)

Именованные аргументы

Именованные (или ключевые) аргументы передаются в функцию с помощью записи вида `имя=значение`:

```
# arguments.keyword.py
def func(a, b, c):
    print(a, b, c)

func(a=1, c=2, b=3) # выводит: 1 3 2
```

Когда используются именованные аргументы, порядок их указания не обязан совпадать с порядком параметров в определении функции. Это повышает читабельность кода и упрощает его отладку. Нет необходимости запоминать (или искать) порядок параметров — прямо в вызове функции видно, какое значение относится к тому или иному параметру.

Кроме того, позиционные и именованные аргументы можно смешивать:

```
# arguments.positional.keyword.py
def func(a, b, c):
    print(a, b, c)

func(42, b=1, c=2)
```

Но важно помнить: *позиционные аргументы всегда должны указываться перед именованными*. Если попытаться сделать вот так:

```
# arguments.positional.keyword.py
func(b=1, c=2, 42) # позиционный аргумент после именованных аргументов
```

то возникнет ошибка выполнения:

```
$ python arguments.positional.keyword.py
  File "arguments.positional.keyword.py", line 7
    func(b=1, c=2, 42) # позиционный аргумент после именованных аргументов
                   ^
SyntaxError: positional argument follows keyword argument
```

Распаковка итерируемого объекта

Такой способ используется для передачи элементов в виде позиционных аргументов с помощью синтаксиса `*имя_итерируемого_объекта`:

```
# arguments.unpack.iterable.py
def func(a, b, c):
    print(a, b, c)

values = (1, 3, -7)
func(*values) # эквивалент: func(1, 3, -7)
```

Эта возможность особенно полезна, когда нужно программно сформировать список аргументов для передачи в функцию.

Распаковка словаря

Относится к именованным аргументам так же, как распаковка итерируемого объекта — к позиционным. С помощью синтаксиса `**имя_словаря` можно передать пары «ключ — значение» как именованные аргументы:

```
# arguments.unpack.dict.py
def func(a, b, c):
    print(a, b, c)

values = {"b": 1, "c": 2, "a": 42}
func(**values) # эквивалент: func(b=1, c=2, a=42)
```

Комбинирование различных типов аргументов

Вы уже видели, что позиционные и именованные аргументы можно использовать вместе, — главное, чтобы они передавались в правильном порядке. Точно так же можно комбинировать распаковку (и итерируемых объектов, и словарей) с обычными позиционными и именованными аргументами. Более того, можно распаковывать сразу несколько итерируемых объектов и словарей.

Аргументы в вызове функции должны передаваться в следующем порядке:

1) позиционные аргументы: обычные (`имя`) и распакованные (`*имя`);
2) именованные аргументы (`имя=значение`), их можно смешивать с `*имя`;
3) распаковка словарей (`**имя`), которую можно комбинировать с `имя=значение`.

Эта тема станет куда более понятной, если взглянуть на пример:

```
# arguments.combined.py
def func(a, b, c, d, e, f):
    print(a, b, c, d, e, f)

func(1, *(2, 3), f=6, *(4, 5))
func(*(1, 2), e=5, *(3, 4), f=6)
func(1, **{"b": 2, "c": 3}, d=4, **{"e": 5, "f": 6})
func(c=3, *(1, 2), **{"d": 4}, e=5, **{"f": 6})
```

Все приведенные выше вызовы `func()` эквивалентны и выводят: `1 2 3 4 5 6`. Поэкспериментируйте с этим примером, пока не разберетесь в теме. Особое внимание обратите на ошибки, которые возникают при неправильном порядке аргументов.

Возможность распаковывать сразу несколько итерируемых объектов и словарей была добавлена в Python в рамках PEP 448. Вдобавок в этом документе рассматривается расширение области применения распаковки — теперь ее можно использовать не только в вызовах функций. Подробности можно прочитать по адресу https://peps.python.org/pep-0448/.

Важно помнить: при комбинировании позиционных и именованных аргументов каждый параметр может быть указан только один раз:

```
# arguments.multiple.value.py
def func(a, b, c):
    print(a, b, c)

func(2, 3, a=1)
```

Здесь параметру `a` передано два значения: сначала как позиционный аргумент `2`, а затем — как именованный `a=1`. Это недопустимо, и при выполнении возникнет ошибка:

```
$ python arguments.multiple.value.py
Traceback (most recent call last):
  File "arguments.multiple.value.py", line 5, in <module>
    func(2, 3, a=1)
TypeError: func() got multiple values for argument 'a'
```

Определение параметров

Параметры функции можно разделить на пять категорий.

- Позиционные или именованные параметры — можно передавать как позиционно, так и по имени.
- Параметры с переменным количеством позиционных аргументов — собирают произвольное количество позиционных аргументов в кортеж.
- Параметры с переменным количеством именованных аргументов — собирают произвольное количество именованных аргументов в словарь.
- Параметры, передаваемые только позиционно, — их можно указать только как позиционные.
- Параметры, передаваемые только по имени, — их можно передать только как именованные аргументы.

Все параметры, с которыми мы до этого момента работали в главе, относились к обычным позиционным или именованным параметрам. Вы уже видели, что такие параметры можно передавать как позиционно, так и по имени. Больше о них сказать нечего, так что перейдем к другим категориям. Но, прежде чем делать это, коротко рассмотрим необязательные параметры.

Необязательные параметры

Помимо рассмотренных ранее категорий, параметры можно разделить на *обязательные* и *необязательные*. Вторые имеют значение по умолчанию, указанное в определении функции. Синтаксис имеет вид `имя=значение`:

```
# parameters.default.py
def func(a, b=4, c=88):
    print(a, b, c)

func(1)              # выводит: 1 4 88
func(b=5, a=7, c=9)  # выводит: 7 5 9
func(42, c=9)        # выводит: 42 4 9
func(42, 43, 44)     # выводит: 42, 43, 44
```

В этом примере параметр `a` — обязательный, `b` и `c` — необязательные: для `b` задано значение по умолчанию `4`, для `c` — `88`. Важно помнить: за исключением параметров, передаваемых только по имени, все обязательные параметры в определении функции должны располагаться слева от необязательных. Попробуйте убрать значение по умолчанию для `c` — и посмотрите, что произойдет.

Параметры с переменным количеством позиционных аргументов

Иногда бывает удобно не указывать точно, сколько позиционных аргументов будет передано функции. В Python это можно сделать с помощью *параметра с переменным числом позиционных аргументов*. Рассмотрим типичный при-

мер — функцию `minimum()`, которая находит минимальное значение среди переданных:

```
# parameters.variable.positional.py
def minimum(*n):
    # print(type(n))  # n — это кортеж
    if n:             # объяснение после кода
        mn = n[0]
        for value in n[1:]:
            if value < mn:
                mn = value
        print(mn)

minimum(1, 3, -7, 9)  # n = (1, 3, -7, 9) — выводит: -7
minimum()             # n = () — ничего не выводит
```

Как видите, если перед именем параметра мы ставим звездочку *, то сообщаем Python, что параметр собирает произвольное количество позиционных аргументов. Внутри функции переменная `n` будет представлять собой кортеж. Раскомментируйте строку `print(type(n))`, чтобы убедиться в этом, и поэкспериментируйте с функцией.

Обратите внимание, что функция может иметь только один такой параметр. Несколько значений `*имя` не допускаются, поскольку Python не сможет определить, какие аргументы относятся к тому или иному параметру. Кроме того, стандартное значение указать нельзя — по умолчанию такой параметр всегда получает пустой кортеж.

Вы обратили внимание на то, как мы проверили, что переменная `n` не пуста, с помощью простого условия `if n:`? Такой способ работает, поскольку коллекции в Python интерпретируются как True, если не являются пустыми, и как False в противном случае. Это относится к кортежам, спискам, множествам, словарям и другим подобным структурам.

Еще один момент, который стоит иметь в виду: иногда более разумно явно сообщить об ошибке, если функция вызвана без аргументов, а не просто молча пропускать выполнение. В данном случае цель — не создать надежную реализацию, а разобраться, как работают параметры с переменным количеством аргументов.

Кстати, вы обратили внимание, что синтаксис параметров `*имя` очень похож на синтаксис распаковки итерируемых объектов (`*итерируемый`)? Это не случайность. Эти две возможности Python зеркально отражают друг друга. Они часто используются вместе, поскольку параметры с * позволяют не беспокоиться о том, совпадает ли длина распаковываемого объекта с количеством параметров в определении функции.

Параметры с переменным количеством именованных аргументов

Эти параметры очень похожи на параметры с переменным количеством позиционных аргументов. Разница лишь в синтаксисе (** вместо *) и в том, что все переданные аргументы собираются в словарь:

```
# parameters.variable.keyword.py
def func(**kwargs):
    print(kwargs)

func(a=1, b=42)        # выводит {'a': 1, 'b': 42}
func()                 # выводит {}
func(a=1, b=46, c=99)  # выводит {'a': 1, 'b': 46, 'c': 99}
```

Если указать ** перед именем параметра в определении функции, то Python будет использовать это имя, чтобы собрать все переданные именованные аргументы в словарь. Как и в случае с *имя, в функции может быть только один такой параметр, и указать значение по умолчанию для него нельзя.

Как *-параметры напоминают распаковку итерируемых объектов, так и **-параметры напоминают распаковку словарей. Их действительно часто используют вместе: распаковка словаря — удобный способ передать аргументы функции, принимающей переменное количество именованных параметров.

Зачем вообще нужна возможность передавать переменное количество именованных аргументов? На первый взгляд, польза может быть неочевидна. Разберем более реалистичный пример. Предположим, нам нужно определить функцию для подключения к базе данных. Мы хотим, чтобы по умолчанию функция подключалась к стандартной базе, если ее вызвать без параметров, и при этом мы могли подключаться к любой другой базе, передавая нужные параметры путем вызова этой же функции. Прежде чем читать дальше, подумайте пару минут о том, как бы вы это реализовали:

```
# parameters.variable.db.py
def connect(**options):
    conn_params = {
        "host": options.get("host", "127.0.0.1"),
        "port": options.get("port", 5432),
        "user": options.get("user", ""),
        "pwd": options.get("pwd", ""),
    }
    print(conn_params)
    # затем мы подключаемся к базе данных (закомментировано)
    # db.connect(**conn_params)

connect()
connect(host="127.0.0.42", port=5433)
connect(port=5431, user="fab", pwd="gandalf")
```

Заметьте, что внутри функции мы можем сформировать словарь параметров подключения (`conn_params`), задав значения по умолчанию, которые будут использоваться, если их не переопределили при вызове функции. Есть более компактные способы сделать это, но в данный момент важна не краткость кода, а то, чтобы вы поняли, как работать с переменными именованными параметрами. Результат выполнения приведенного выше кода будет следующим:

```
$ python parameters.variable.db.py
{'host': '127.0.0.1', 'port': 5432, 'user': '', 'pwd': ''}
{'host': '127.0.0.42', 'port': 5433, 'user': '', 'pwd': ''}
{'host': '127.0.0.1', 'port': 5431, 'user': 'fab', 'pwd': 'gandalf'}
```

Обратите внимание на соответствие между вызовами функции и результатами, а также на то, как значения по умолчанию переопределяются в зависимости от переданных аргументов.

Параметры, передаваемые только позиционно

Начиная с Python 3.8 в рамках PEP 570 (https://peps.python.org/pep-0570/) в язык была добавлена возможность определять параметры, которые можно передавать только позиционно. Для этого используется новый синтаксис в определении функции — символ `/`, указывающий, что все параметры, находящиеся слева от него, должны передаваться только позиционно и не могут быть заданы по имени. Рассмотрим простой пример:

```
# parameters.positional.only.py
def func(a, b, /, c):
    print(a, b, c)

func(1, 2, 3)   # выводит: 1 2 3
func(1, 2, c=3) # выводит: 1 2 3
```

Функция `func()` принимает три параметра: `a`, `b` и `c`. Символ `/` в сигнатуре означает, что `a` и `b` должны быть переданы только позиционно, то есть нельзя передавать их как `a=1`, `b=2`. В последних двух строках показаны допустимые вызовы: передаем все аргументы позиционно или передаем `c` как именованный параметр — это разрешено, поскольку `c` находится после символа `/`. А вот такой вызов:

```
func(1, b=2, c=3)
```

приведет к ошибке выполнения:

```
Traceback (most recent call last):
  File "arguments.positional.only.py", line 7, in <module>
    func(1, b=2, c=3)
TypeError: func() got some positional-only arguments
passed as keyword arguments: 'b'
```

Пример показывает, что Python теперь строго различает позиционные и именованные аргументы и не разрешает передавать `b` по имени, если тот в сигнатуре функции был обозначен как позиционный.

При этом параметры, передаваемые только позиционно, могут быть необязательными:

```
# parameters.positional.only.optional.py
def func(a, b=2, /):
    print(a, b)

func(4, 5) # выводит: 4 5
func(3)    # выводит: 3 2
```

Теперь посмотрим, как Python определяет параметры, передаваемые только позиционно, на нескольких примерах из официальной документации. Одно из преимуществ использования этого языка — возможность полностью эмулировать поведение функций, реализованных на C:

```
def divmod(a, b, /):
    "Emulate the built in divmod() function"
    return (a // b, a % b)
```

Еще один важный сценарий использования — отказ от именованных аргументов, когда имя параметра негативно сказывается на читабельности кода:

```
len(obj='hello')
```

Здесь параметр `obj`, передаваемый по имени, только ухудшает читабельность. Кроме того, если в будущем параметр `obj` во внутренней реализации функции `len()` придется переименовать, например, в `the_object`, это не повлияет на внешний код. Почему? Вызовов с `obj=` просто не будет — параметры, передаваемые только позиционно, не могут передаваться по имени, так что имя внутри реализации можно менять без риска.

Наконец, если параметры, передаваемые только позиционно, обозначать с помощью `/`, то все параметры слева от данного знака остаются доступными для передачи через `**kwargs`, как видно в следующем примере:

```
def func_name(name, /, **kwargs):
    print(name)
    print(kwargs)

func_name("Positional-only name", name="Name in **kwargs")
# выводит:
# Positional-only name
# {'name': 'Name in **kwargs'}
```

Такая возможность — использовать имена параметров в сигнатуре функции и при этом передавать их через `**kwargs` — может сильно упростить и оптимизировать код.

Параметры, передаваемые только по имени

Возможность объявлять такие параметры появилась в языке начиная с Python 3. Мы рассмотрим их кратко, поскольку на практике они встречаются не очень часто. Существует два способа определить эти параметры: либо после параметра с *имя (переменное количество позиционных аргументов), либо после одинокого * (без имени). В примере ниже показаны оба варианта:

```
# parameters.keyword.only.py
def kwo(*a, c):
    print(a, c)

kwo(1, 2, 3, c=7)   # выводит: (1, 2, 3) 7
kwo(c=4)            # выводит: () 4
# kwo(1, 2)         # вызовет ошибку, неверный синтаксис, с сообщением:
# TypeError: kwo() missing 1 required keyword-only argument: 'c'

def kwo2(a, b=42, *, c):
    print(a, b, c)

kwo2(3, b=7, c=99)  # выводит: 3 7 99
kwo2(3, c=13)       # выводит: 3 42 13
# kwo2(3, 23)       # вызовет ошибку, неверный синтаксис, с сообщением:
# TypeError: kwo2() missing 1 required keyword-only argument: 'c'
```

Как и ожидалось, функция `kwo()` принимает переменное количество позиционных аргументов (`a`) и один именованный параметр `c`, который должен передаваться только по имени. Результаты вызовов очевидны. Можете раскомментировать третий вызов, чтобы увидеть, какую ошибку вернет Python.

То же самое справедливо и для функции `kwo2()`, которая отличается от `kwo` тем, что принимает один позиционный аргумент `a`, один обычный именованный аргумент `b` и затем обязательный именованный параметр `c`. Раскомментируйте третий вызов, чтобы увидеть сообщение об ошибке.

Теперь, когда вы знаете, как задаются разные типы входных параметров, посмотрим, как их можно комбинировать в определениях функций.

Комбинирование входных параметров

В одной функции можно сочетать разные типы параметров — и на практике это даже очень удобно. Как и при передаче аргументов, здесь тоже есть жесткий порядок, которого нужно придерживаться:

1) параметры, передаваемые только позиционно, за ними ставится `/`;
2) обычные параметры, которые можно передавать как позиционно, так и по имени;
3) параметры с переменным количеством позиционных аргументов (`*имя`);
4) параметры, передаваемые только по имени;
5) параметры с переменным количеством именованных аргументов (`**имя`).

Дополнительное правило: для параметров, передаваемых только позиционно, и обычных параметров все обязательные должны быть определены до необязательных. Это значит, что если у вас есть необязательный параметр, передаваемый только позиционно, то все обычные тоже должны быть необязательными. Это ограничение не касается параметров, передаваемых только по имени.

Эти правила могут показаться непростыми, так что посмотрим на несколько фрагментов — примеров кода:

```
# parameters.all.py
def func(a, b, c=7, *args, **kwargs):
    print("a, b, c:", a, b, c)
    print("args:", args)
    print("kwargs:", kwargs)

func(1, 2, 3, 5, 7, 9, A="a", B="b")
```

Обратите внимание на порядок параметров в определении функции. Выполнение этого кода дает следующий результат:

```
$ python parameters.all.py
a, b, c: 1 2 3
args: (5, 7, 9)
kwargs: {'A': 'a', 'B': 'b'}
```

Теперь посмотрим на пример, где параметры передаются только по имени:

```
# parameters.all.pkwonly.py
def allparams(a, /, b, c=42, *args, d=256, e, **kwargs):
    print("a, b, c:", a, b, c)
    print("d, e:", d, e)
    print("args:", args)
    print("kwargs:", kwargs)

allparams(1, 2, 3, 4, 5, 6, e=7, f=9, g=10)
```

Здесь в определении функции используются и параметры, передаваемые только позиционно, и параметры, передаваемые только по имени: `a` передается только позиционно, `d` и `e` — только по имени. Эти параметры передаются после `*args`, то есть переменного количества позиционных аргументов. Аналогично они бы работали, если бы шли сразу после одиночного `*`, — в этом случае не было бы `*args`, но `d` и `e` все так же требовали бы передачу по имени. Результат выполнения будет таким:

```
$ python parameters.all.pkwonly.py
a, b, c: 1 2 3
d, e: 256 7
args: (4, 5, 6)
kwargs: {'f': 9, 'g': 10}
```

Еще одно замечание: имена `args` и `kwargs` в параметрах с `*` и `**` необязательны, вы можете использовать любые другие. Но стоит помнить, что `args` и `kwargs` — общепринятые имена, и в целях улучшения читабельности кода лучше придерживаться их.

Дополнительные примеры сигнатур

Чтобы кратко подытожить все, что касается сигнатур функций с параметрами, передаваемыми только позиционно и только по имени, приведем еще несколько примеров. Мы опустим параметры с `*args` и `**kwargs`, чтобы сосредоточиться на основной структуре:

```
def func_name(positional_only_parameters, /,
    positional_or_keyword_parameters, *,
    keyword_only_parameters):
```

Общая последовательность такая: сначала только позиционные, затем — позиционные или именованные и в конце — только именованные.

Вот несколько корректных сигнатур:

```
def func_name(p1, p2, /, p_or_kw, *, kw):
def func_name(p1, p2=None, /, p_or_kw=None, *, kw):
def func_name(p1, p2=None, /, *, kw):
def func_name(p1, p2=None, /):
def func_name(p1, p2, /, p_or_kw):
def func_name(p1, p2, /):
```

А вот некорректные:

```
def func_name(p1, p2=None, /, p_or_kw, *, kw):
def func_name(p1=None, p2, /, p_or_kw=None, *, kw):
def func_name(p1=None, p2, /):
```

Подробные правила синтаксиса можно найти в официальной документации по адресу https://docs.python.org/3/reference/compound_stmts.html#function-definitions.

Полезное упражнение: реализуйте одну из этих сигнатур, выведите значения параметров (как мы делали ранее) и поэкспериментируйте с разными способами передачи аргументов — это поможет закрепить материал.

Избегайте ловушки! Изменяемые значения по умолчанию

Одна из особенностей Python заключается в том, что значения по умолчанию создаются в момент определения функции. Поэтому повторные вызовы одной и той же функции могут вести себя по-разному — в зависимости от того, изменяемы ли эти значения. Рассмотрим пример:

```
# parameters.defaults.mutable.py
def func(a=[], b={}):
    print(a)
    print(b)
    print("#" * 12)
    a.append(len(a))    # это изменит значение по умолчанию для a
    b[len(a)] = len(a)  # и это изменит значение по умолчанию для b

func()
func()
func()
```

Здесь оба параметра имеют изменяемые значения по умолчанию. Это означает, что если изменить такие объекты, то изменения сохранятся между вызовами функции. Попробуйте проанализировать, почему вывод получается таким:

```
$ python parameters.defaults.mutable.py
[]
{}
############
[0]
{1: 1}
############
[0, 1]
{1: 1, 2: 2}
############
```

Сначала такое поведение может показаться странным, однако оно абсолютно логично и иногда бывает полезным — например, при реализации кэширования *(мемоизации)*. Ситуация становится еще более интересной, если между вызовами с аргументами по умолчанию выполнить один вызов с явно переданными значениями:

```
# parameters.defaults.mutable.intermediate.call.py
func()
func(a=[1, 2, 3], b={"B": 1})
func()
```

Вывод будет таким:

```
$ python parameters.defaults.mutable.intermediate.call.py
[]
{}
############
[1, 2, 3]
{'B': 1}
############
[0]
{1: 1}
############
```

Это подтверждает, что объекты по умолчанию сохраняются, даже если между ними были вызовы с другими аргументами. Но как поступать, если каждый раз нужен новый пустой объект? В Python для этого есть общепринятая практика:

```
# parameters.defaults.mutable.no.trap.py
def func(a=None):
    if a is None:
        a = []
    # делайте с параметром a что угодно ...
```

При использовании такого подхода, если параметр `a` не передан при вызове функции, вы всегда будете получать совершенно новый пустой список.

Теперь, когда мы подробно разобрали тему входных параметров, пора перейти к возврату значений — второй ключевой части определения функции.

Возврат значений

Мы уже говорили, что для возврата значения из функции нужно использовать оператор `return`, за которым указывается то, что мы хотим вернуть. При необходимости тело функции может содержать несколько операторов `return`.

С другой стороны, если в теле функции нет оператора `return` или если он используется без значения, то функция возвращает `None`. Такое поведение безвредно, когда возврат значения не нужен, но при этом открывает интересные возможности и в очередной раз подчеркивает, насколько последовательным остается Python.

Мы говорим, что это «безвредно», поскольку никто не заставляет вас сохранять результат вызова функции. Сейчас мы покажем, что это означает на практике:

```
# return.none.py
def func():
    pass

func()      # результат этого вызова не сохраняется — он теряется
a = func()  # результат этого вызова сохраняется в переменную a
print(a)    # выводит: None
```

Обратите внимание: тело функции содержит только оператор `pass`. Как сказано в официальной документации, `pass` — пустая операция: при ее выполнении ничего не происходит. Она полезна как заглушка, когда синтаксис требует наличия оператора, но в этом месте пока не нужно выполнять какой-либо код. В других языках в таких случаях можно было бы использовать пустые фигурные скобки `{}`, чтобы обозначить *пустую область*. Но в Python область определяется отступами, поэтому необходимо вставить хотя бы `pass`, чтобы тело блока существовало.

Кроме того, заметьте, что в первом вызове `func()` возвращается значение `None`, но мы не сохраняем его. Как мы уже говорили, сохранять результат вызова функции не обязательно.

А теперь перейдем к более интересному примеру. Помните, в главе 1 мы говорили о *факториале*? Теперь напишем собственную реализацию такой функции. (Для простоты предположим, что функция всегда вызывается с корректными значениями, поэтому не будем выполнять проверку аргумента.)

```
# return.single.value.py
def factorial(n):
    if n in (0, 1):
        return 1
    result = n
```

```
    for k in range(2, n):
        result *= k
    return result

f5 = factorial(5) # f5 = 120
```

Обратите внимание: в функции два пути возврата значения. Если `n` присвоено значение `0` или `1`, сразу возвращается `1`. Во всех остальных случаях происходит вычисление, и результат возвращается с помощью оператора `return result`.

В Python часто используется оператор in для проверки принадлежности — как мы сделали в данном примере. Такой код короче и понятнее, чем более громоздкий вариант:

```
if n == 0 or n == 1:
    ...
```

Теперь сделаем реализацию этой функции чуть более компактной:

```
# return.single.value.2.py
from functools import reduce
from operator import mul

def factorial(n):
    return reduce(mul, range(1, n + 1), 1)

f5 = factorial(5) # f5 = 120
```

Этот простой пример демонстрирует, насколько Python может быть лаконичным и выразительным одновременно. Даже если вы ни разу не видели `reduce()` или `mul()`, код остается читабельным. Если сейчас он кажется непонятным — ничего страшного. Загляните в документацию Python и изучите работу этих функций. Умение читать чужой код и находить нужную информацию — важнейший навык для любого разработчика.

Кстати, для изучения функций в интерактивном режиме очень пригодится встроенная функция help(). Она незаменима при исследовании кода в консоли.

Возврат нескольких значений

Возвращать сразу несколько значений в Python очень просто — достаточно использовать кортежи. Рассмотрим простой пример, имитирующий поведение встроенной функции `divmod()`:

```
# return.multiple.py
def moddiv(a, b):
    return a // b, a % b

print(moddiv(20, 7)) # выводит: (2, 6)
```

Мы могли бы обернуть возвращаемое выражение в круглые скобки, но в этом нет необходимости — Python и так понимает, что нужно вернуть кортеж. Эта функция сразу возвращает и результат деления, и остаток — в одном значении.

В исходном коде этого примера оставлена простая функция для тестирования, которая позволяет убедиться, что вычисление работает корректно.

Несколько полезных советов

Когда вы пишете функции, полезно придерживаться определенных рекомендаций — так код получается более чистым и надежным.

- *Функция должна выполнять только одну задачу.* Функцию, которая делает что-то одно, легко описать в одном коротком предложении. Если она решает несколько задач — ее лучше разбить на более мелкие, каждая из которых отвечает только за одну задачу. Код таких функций легче читать и понимать.
- *Функция должна быть небольшой.* Чем она компактнее, тем проще ее протестировать и убедиться, что она действительно делает только то, что должна.
- *Чем меньше входных параметров, тем лучше.* Функции с большим количеством параметров быстро становятся неудобными в использовании и сопровождении (и это не все проблемы).
- *Функция должна возвращать значения последовательно.* `False` и `None` — не одно и то же. В булевом контексте они оба означают «ложь», но по смыслу `False` — это ложный результат, а `None` — отсутствие результата вообще. Пишите функции так, чтобы они возвращали значения одного типа или одного семантического смысла, независимо от хода выполнения.
- *Функции не должны иметь побочных эффектов.* В функциональном программировании есть понятие *чистой функции*, которая соблюдает два принципа.
  - *Детерминированный результат.* При одних и тех же входных данных функция всегда возвращает один и тот же результат. То есть ее поведение не зависит от внешнего состояния, которое может измениться во время выполнения.
  - *Отсутствие побочных эффектов.* Чистая функция не изменяет внешнее состояние системы: не затрагивает глобальные переменные, не выполняет ввод-вывод, не пишет и не читает файлы, не выводит информацию на экран.

Если не удается сделать функцию чистой, то постарайтесь хотя бы исключить побочные эффекты. Функции не должны изменять значения аргументов, с которыми были вызваны. Возможно, это утверждение труднее всего понять на

данном этапе, поэтому приведем пример с использованием списков. В коде ниже функция `sorted()` не изменяет список `numbers`, а возвращает отсортированную копию. В то же время метод `list.sort()` изменяет сам объект `numbers` — и это нормально, поскольку это метод списка, то есть функция, которая принадлежит объекту и имеет право его изменять.

```
>>> numbers = [4, 1, 7, 5]
>>> sorted(numbers)  # не сортирует исходный список numbers
[1, 4, 5, 7]
>>> numbers          # проверим
[4, 1, 7, 5]         # отлично, не изменен
>>> numbers.sort()   # этот метод изменит список на месте
>>> numbers
[1, 4, 5, 7]
```

Если вы будете следовать этим рекомендациям, то автоматически избавите себя от многих трудноуловимых ошибок.

Глава 3 книги *Clean Code* Роберта Мартина посвящена функциям — и это один из лучших сводов практических рекомендаций, которые мы когда-либо читали по этой теме.

Рекурсивные функции

Если функция вызывает саму себя, то называется *рекурсивной*. В некоторых случаях рекурсия оказывается очень удобной: с ее помощью проще выразить нужную логику. Одни алгоритмы гораздо легче записать рекурсивно, другие — наоборот. Любую рекурсивную функцию можно переписать в виде итерационной, поэтому выбор подхода зависит от конкретной задачи и предпочтений программиста.

Есть два варианта выполнения рекурсивной функции. Рекурсивный случай: когда функция для получения результата вызывает саму себя. Базовый случай: когда функция возвращает ответ сразу, без вызова себя.

В качестве примера рассмотрим, вероятно, уже знакомую функцию вычисления *факториала N!*. Базовый случай возникает, когда N равно 0 или 1 — функция просто возвращает `1`, не выполняя дальнейшие вычисления. В общем же случае факториал N вычисляется как произведение:

```
1 * 2 * ... * (N - 1) * N
```

Факториал можно выразить через рекурсивную формулу: $N! = (N - 1)! \times N$. На практике это выглядит так:

```
5! = 1 * 2 * 3 * 4 * 5 = (1 * 2 * 3 * 4) * 5 = 4! * 5
```

А в коде так:

```
# recursive.factorial.py
def factorial(n):
    if n in (0, 1):              # базовый случай
        return 1
    return factorial(n - 1) * n  # рекурсивный случай
```

Рекурсивные функции часто используются в алгоритмах и могут быть по-настоящему интересными. Попрактикуйтесь с ними: попробуйте решить несколько простых задач как рекурсивным, так и итерационным способом. Хорошие кандидаты для таких упражнений — вычисление чисел Фибоначчи или длины строки.

При работе с рекурсией важно помнить, сколько вложенных вызовов происходит. Для этого в Python предусмотрены функции sys.getrecursionlimit() и sys.setrecursionlimit(), с помощью которых можно узнать текущий лимит или изменить его.

Анонимные функции

Последний тип функций, о котором стоит упомянуть, — *анонимные* функции. В Python они называются *лямбда-выражениями* (`lambda`) и обычно применяются в тех случаях, когда полноценная именованная функция является излишней, а нужна короткая однострочная конструкция.

Допустим, нужно получить список всех чисел до заданного значения *N*, которые делятся на пять. Для этого можно воспользоваться функцией `filter()`. Она принимает в качестве аргументов функцию и итерируемый объект. Возвращается объект `filter`, при переборе которого выдаются те элементы исходного итерируемого объекта, для которых функция возвращает `True`. Без использования анонимной функции такой код выглядел бы следующим образом:

```
# filter.regular.py
def is_multiple_of_five(n):
    return not n % 5

def get_multiples_of_five(n):
    return list(filter(is_multiple_of_five, range(n)))
```

Здесь мы используем функцию `is_multiple_of_five()` для фильтрации первых *n* натуральных чисел. Но в данном случае такая реализация выглядит чрезмерной — задача простая и функция `is_multiple_of_five()` больше нигде не пригодится. Перепишем ее с помощью лямбда-выражения:

```
# filter.lambda.py
def get_multiples_of_five(n):
    return list(filter(lambda k: not k % 5, range(n)))
```

Логика осталась прежней, но теперь функция фильтрации записана в виде лямбда-выражения. Объявить его очень просто — синтаксис выглядит так:

```
func_name = lambda [parameter_list]: expression
```

Это лямбда-выражение возвращает объект функции, который полностью эквивалентен следующей записи:

```
def func_name([parameter_list]):
    return expression
```

Обратите внимание: необязательные параметры можно указывать по стандартным правилам — заключая их в квадратные скобки.

Посмотрим еще пару примеров, в которых функции заданы как через `def`, так и через `lambda`:

```
# lambda.explained.py
# пример 1: сложение
def adder(a, b):
    return a + b

# эквивалентно:
adder_lambda = lambda a, b: a + b

# пример 2: преобразование в верхний регистр
def to_upper(s):
    return s.upper()

# эквивалентно:
to_upper_lambda = lambda s: s.upper()
```

Оба примера довольно просты. В первом вычисляется сумма двух чисел, во втором возвращается строка в верхнем регистре. Здесь результат лямбда-выражения сохраняется в переменную (`adder_lambda`, `to_upper_lambda`), но это не обязательно — как видно из примера с `filter()`, лямбда-выражение можно использовать прямо в месте вызова, не сохраняя.

Атрибуты функций

Любая функция в Python — полноценный объект, у которого есть набор атрибутов. Некоторые из них являются специальными и используются для интроспекции, то есть для анализа объекта функции во время выполнения. В примере ниже показаны несколько таких атрибутов и способ их отображения для конкретной функции:

```
# func.attributes.py
def multiplication(a, b=1):
    """Возвращает произведение a и b."""
    return a * b

if __name__ == "__main__":
    special_attributes = [
        "__doc__",
        "__name__",
        "__qualname__",
        "__module__",
        "__defaults__",
        "__code__",
        "__globals__",
        "__dict__",
        "__closure__",
        "__annotations__",
        "__kwdefaults__",
    ]

    for attribute in special_attributes:
        print(attribute, "->", getattr(multiplication, attribute))
```

Здесь используется встроенная функция `getattr()`, с помощью которой мы получаем значения атрибутов. Вызов `getattr(obj, attribute)` эквивалентен `obj.attribute`, но особенно удобен, когда имя атрибута хранится в переменной и становится известно только во время выполнения (как в этом примере). При запуске этот сценарий выдает:

```
$ python func.attributes.py
__doc__ -> Return a multiplied by b.
__name__ -> multiplication
__qualname__ -> multiplication
__module__ -> __main__
__defaults__ -> (1,)
__code__ -> <code object multiplication at 0x102ce1550,
             file "func.attributes.py", line 2>
__globals__ -> {... omitted ...}
__dict__ -> {}
__closure__ -> None
__annotations__ -> {}
__kwdefaults__ -> None
```

Атрибут `__globals__` был опущен, поскольку его содержимое слишком объемное. Пояснение к нему можно найти в разделе *Callable types* на странице *Data Model* документации Python (https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy).

Чтобы получить список всех атрибутов любого объекта, воспользуйтесь встроенной функцией dir().

Обратите внимание на конструкцию, использованную в предыдущем примере:

```
if __name__ == "__main__":
```

Эта строка гарантирует, что следующий за ней код будет выполняться только при непосредственном запуске модуля, а не при его импорте. Когда Python запускает сценарий, переменной `__name__` присваивается значение `"__main__"`. Если же сценарий импортируется как модуль, то `__name__` получает имя данного модуля.

Встроенные функции

В Python доступно множество встроенных функций. Они доступны повсеместно, и их не нужно предварительно импортировать. Чтобы просмотреть их список, можно вызвать `dir(builtins)` или изучить официальную документацию Python. Описать все встроенные функции в этой главе не получится. Некоторые из них вы уже встречали — например, `any`, `bin`, `bool`, `divmod`, `filter`, `float`, `getattr`, `id`, `int`, `len`, `list`, `min`, `print`, `set`, `tuple`, `type`, `zip`. Но, помимо них, есть еще много других функций, о которых стоит хотя бы раз прочитать. Полезно познакомиться с каждой встроенной функцией: попробуйте написать небольшой фрагмент кода для каждой из них, чтобы лучше понять, как она работает. Так вы сможете быстрее их освоить и будете более уверенно применять в нужный момент.

Полный список встроенных функций можно найти в официальной документации Python по адресу https://docs.python.org/3/library/functions.html.

Документирование кода

Нам близка идея кода, который не нуждается в документации. Хорошо написанный, элегантный код, созданный по общепринятым принципам, часто понятен сам по себе — и тогда дополнительные пояснения почти не требуются. Тем не менее добавление строки документации (docstring) к функции или комментария с важной информацией может оказаться весьма полезным.

Рекомендации по документированию кода в Python изложены в PEP 257 — *Docstring conventions* (https://peps.python.org/pep-0257/), но в этом разделе мы приведем основные правила.

В Python для документации используются строки, которые называются *строками документации — docstring*. Документировать можно любой объект. Строки бывают одно- и многострочными. Однострочные максимально просты. В них не следует дублировать сигнатуру функции — достаточно коротко описать, что она делает:

```
# docstrings.py
def square(n):
    """Возвращает квадрат числа n."""
    return n**2
```

```
def get_username(userid):
    """Возвращает имя пользователя по его идентификатору."""
    return db.get(user_id=userid).username
```

Использование строк в тройных двойных кавычках удобно тем, что такую документацию легко расширить позже. Старайтесь писать полными предложениями, завершая их точкой. Пустые строки в начале и в конце строк документации добавлять не нужно.

Многострочные комментарии оформляются схожим образом: сначала идет короткая строка, в которой кратко описывается назначение объекта, а затем — более подробное пояснение. Ниже показан пример документирования вымышленной функции `connect()` с использованием синтаксиса, принятого в *Sphinx*:

```
def connect(host, port, user, password):
    """Подключение к базе данных.
    Выполняет прямое подключение к базе данных PostgreSQL с использованием
    переданных параметров.

    :param host: IP-адрес хоста.
    :param port: Порт подключения.
    :param user: Имя пользователя для подключения.
    :param password: Пароль для подключения.
    :return: Объект подключения.
    """
    # тело функции здесь...
    return connection
```

Sphinx — один из самых популярных инструментов для создания документации в Python. Именно с его помощью написана официальная документация языка. Познакомиться с ним определенно стоит.

Встроенная функция `help()` предназначена для интерактивной работы и позволяет просматривать документацию по объекту, используя соответствующую docstring.

Импорт объектов

Теперь, когда мы разобрали функции, самое время поговорить о том, как их использовать. Основной смысл функций — возможность их повторно использовать. В Python это означает импорт в нужное пространство имен. Способов импортировать объекты несколько, но самые распространенные — `import имя_модуля` и `from имя_модуля import имя_функции`. Это базовые конструкции, но начинать можно и с них.

Форма `import имя_модуля` находит модуль `имя_модуля` и создает в текущем пространстве имен одноименную ссылку на него. Вариант `from имя_модуля import идентификатор` устроен чуть сложнее: он тоже находит модуль, но дополнительно

извлекает из него указанный атрибут (или подмодуль) и сохраняет ссылку на него в текущем пространстве имен. Обе формы позволяют переименовывать импортируемые объекты с помощью конструкции `as`:

```
from mymodule import myfunc as better_named_func
```

Чтобы вы представили, как импорт выглядит на практике, ниже мы привели пример из тестового модуля одного из проектов Фабрицио. Обратите внимание на пустые строки между группами импортов — они соответствуют рекомендациям PEP 8 (https://peps.python.org/pep-0008/#imports): сначала стандартная библиотека, затем сторонние библиотеки, а в конце — модули самого проекта:

```
# imports.py
from datetime import datetime, timezone    # два импорта в одной строке
from unittest.mock import patch            # одиночный импорт

import pytest                              # сторонняя библиотека

from core.models import (                  # многострочный импорт
    Exam,
    Exercise,
    Solution,
)
```

Если в структуре проекта есть корень в виде основной папки, то можно использовать точечную нотацию (`.`), чтобы добраться до нужного объекта — будь то пакет, модуль, класс, функция или что-то другое.

Кроме того, синтаксис `from имя_модуля import` поддерживает так называемый «глобальный» импорт — `from имя_модуля import *`, при котором в текущее пространство имен импортируются все доступные объекты модуля. Однако такой подход не рекомендуется. Он снижает читабельность кода, может привести к неявному переопределению имен и повлиять на производительность. Эти и другие нюансы подробно описаны в официальной документации Python.

Прежде чем завершить разговор об операторах импорта, рассмотрим еще один более наглядный пример.

Представим, что мы определили две функции — `square(n)` и `cube(n)` — в модуле `funcdef.py`, который находится в папке `util`. Эти функции нам понадобятся в двух других модулях — `func_import.py` и `func_from.py`, которые расположены на том же уровне, что и папка `util`. Структура проекта выглядит примерно так:

```
├── func_from.py
├── func_import.py
├── util
│   ├── __init__.py
│   └── funcdef.py
```

Прежде чем перейти к коду самих модулей, напомним: чтобы Python воспринимал папку как пакет, в ней должен находиться файл `__init__.py`.

О файле __init__.py нужно помнить две вещи. Во-первых, это обычный модуль Python и в нем можно писать любой код. Во-вторых, начиная с версии Python 3.3 наличие этого файла не является обязательным, чтобы папка считалась пакетом, но его использование все еще может быть полезным.

Код модулей выглядит так:

```
# util/funcdef.py
def square(n):
    return n**2

def cube(n):
    return n**3

# func_import.py
import util.funcdef

print(util.funcdef.square(10))
print(util.funcdef.cube(10))

# func_from.py
from util.funcdef import square, cube

print(square(10))
print(cube(10))
```

Оба модуля при запуске выводят `100` и `1000`. Разница лишь в том, как именно мы обращаемся к функциям `square` и `cube` — это зависит от способа импорта в каждом конкретном случае.

Относительный импорт

До этого момента мы рассматривали так называемый *абсолютный импорт* — когда указывается полный путь к модулю или объекту, который нужно импортировать. Однако в Python есть и другой способ — *относительный импорт*. При его использовании перед именем модуля указываются точки, количество которых означает, на сколько уровней вверх по структуре каталогов следует подняться, чтобы добраться до нужного объекта. Проще говоря, запись может выглядеть так:

```
from .mymodule import myfunc
```

Относительный импорт особенно полезен при реструктуризации проекта. Отсутствие полного пути в строках импорта позволяет более свободно перемещать модули между папками — не нужно переименовывать пути в каждом

месте, где они использовались. Подробное объяснение относительного импорта приведено в PEP 328: https://peps.python.org/pep-0328/.

В следующих главах мы будем создавать проекты с помощью разных библиотек и применять разные виды импорта, включая относительные, так что вам лучше заранее ознакомиться с их описанием в официальной документации Python.

Заключительный пример

Прежде чем закончить главу, рассмотрим еще один пример. Мы уже писали код для генерации списка простых чисел до заданного предела в главе 3. Теперь оформим его в виде функции и слегка оптимизируем, чтобы было интереснее.

Во-первых, при проверке, является ли число N простым, не обязательно делить его на все числа от 2 до $N - 1$. Достаточно остановиться на $\sqrt{N}$ (квадратном корне из N). Более того, не нужно проверять деление на все числа до $\sqrt{N}$ — достаточно использовать только простые числа в данном диапазоне. Почему это работает — вы можете понять самостоятельно, погрузившись в математику.

Посмотрим, как выглядит такой код:

```
# primes.py
from math import sqrt, ceil

def get_primes(n):
    """Вычислить список простых чисел до n (включительно)."""
    primelist = []
    for candidate in range(2, n + 1):
        is_prime = True
        root = ceil(sqrt(candidate))  # ограничение для деления
        for prime in primelist:       # проверяем только простые числа
            if prime > root:          # дальше проверять не нужно
                break
            if candidate % prime == 0:
                is_prime = False
                break
        if is_prime:
            primelist.append(candidate)
    return primelist
```

По сути, это тот же код, который был в предыдущей главе, но с двумя изменениями: проверка делимости происходит только на уже найденные простые числа; проверка останавливается, как только делитель превышает корень из проверяемого числа. Значение корня мы вычисляем немного «элегантнее» — берем целое значение `ceil(sqrt(candidate))`, что требует импорта функций из модуля `math`. Выражение `int(k ** 0.5) + 1` тоже подошло бы, но нам хотелось показать пример с импортом стандартной библиотеки. Изучите содержимое модуля `math` — в нем действительно много интересных функций!

Резюме

В текущей главе мы подробно разобрали функции — одну из ключевых составляющих Python. С этого момента они будут использоваться практически в каждом примере. Мы обсудили, почему стоит применять функции. Самые важные из причин — повторное использование кода и сокрытие реализации.

Функцию можно представить как «коробочку», в которую можно «положить» входные данные (аргументы) и при необходимости «вытащить» результат. Входные значения можно передавать разными способами: позиционно, по имени, а также через специальные формы записи — `*args` и `**kwargs`.

Теперь вы знаете, как написать функцию, задокументировать ее, импортировать и вызвать в коде.

В следующей главе темп повествования немного ускорится, поэтому стоит закрепить пройденный материал: попробуйте поэкспериментировать с кодом и изучите официальную документацию Python — это точно поможет.

5 Генераторы и включения

> Важно не ежедневное увеличение, а ежедневное уменьшение. Отбросьте ненужное.
>
> *Брюс Ли*

Вторая часть этой цитаты — *«отбросьте ненужное»* — лучше всего передает наше понимание элегантности программирования. Мы постоянно ищем более изящные способы выразить одни и те же действия, чтобы не тратить зря ни время, ни память.

Конечно, не всегда есть смысл выжимать из кода максимум производительности. Иногда микроскопическое улучшение может требовать ухудшения читабельности или сопровождаемости. Вряд ли разумно усложнять код до неузнаваемости ради ускорения загрузки страницы с 1,05 до 1 секунды.

Но бывают и обратные случаи — когда имеет смысл сэкономить даже одну миллисекунду, если функция вызывается тысячи раз. В таком контексте даже малая оптимизация дает заметный выигрыш во времени, и это может серьезно повлиять на производительность приложения.

С учетом этого глава будет посвящена не агрессивным приемам оптимизации *любой ценой*, а тому, как писать эффективный, лаконичный и в то же время удобный для чтения код, который не расходует ресурсы впустую.

Мы выполним несколько замеров, сравним результаты и сделаем осторожные выводы. Помните, что на другом компьютере с иной конфигурацией или ОС результаты могут различаться.

А теперь взглянем на такой фрагмент кода:

```
# squares.py
def square1(n):
    return n**2  # возведение в квадрат с помощью оператора степени

def square2(n):
    return n * n  # возведение в квадрат путем умножения
```

Обе функции возвращают квадрат числа `n`, но какая из них работает быстрее? По результатам простого теста немного выигрывает вторая. Это логично: возведение в степень включает в себя умножение, а значит, какую бы реализацию вы ни использовали, она вряд ли окажется быстрее, чем обычное перемножение, — как в `square2`.

А стоит ли вообще обращать внимание на эту разницу? В большинстве случаев — нет. Если вы пишете код для сайта электронной торговли, то, скорее всего, вам вообще не придется возводить числа в квадрат, а если и придется, то крайне редко. Поэтому экономить доли микросекунды на функции, которая вызывается пару раз, не имеет смысла.

Но когда оптимизация действительно важна? Один из типичных случаев — работа с большими коллекциями данных. Если вы применяете одну и ту же функцию к миллиону объектов (например, покупателей), то логично захотеть, чтобы она работала максимально быстро. Сэкономив десятую долю секунды на каждом вызове, вы в сумме выигрываете 100 000 секунд, а это примерно 27,7 часа. Так что далее мы сосредоточимся на коллекциях и посмотрим, какие средства Python предлагает для эффективной работы с ними.

Многие из понятий, с которыми вы познакомитесь в этой главе, основаны на итераторах и итерируемых объектах — вы уже встречались с ними в главе 3. В главе 6 вы научитесь писать собственные итераторы и итерируемые объекты.

Конструкции, которые мы сейчас будем рассматривать, представляют собой именно итераторы. Они экономят память, поскольку обрабатывают по одному элементу за раз, а не создают измененную копию всей коллекции. Из-за этого при выводе результата на экран может понадобиться дополнительное преобразование. Мы часто будем оборачивать итератор в вызов `list()`. Это нужно для того, чтобы исчерпать итератор и собрать все его элементы в обычный список, который легко вывести и изучить.

Рассмотрим пример с объектом `range`:

```
# list.iterable.txt
>>> range(7)
range(0, 7)
>>> list(range(7))  # поместить все элементы в список, чтобы их увидеть
[0, 1, 2, 3, 4, 5, 6]
```

Если ввести `range(7)` в интерактивной оболочке Python, то содержимое диапазона не отобразится, поскольку сам `range` не создает последовательность целиком в памяти. А вот `list(range(7))` сразу покажет все числа, которые может сгенерировать диапазон.

Пора перейти к инструментам, которые Python предлагает для эффективной работы с коллекциями данных.

Функции map(), zip() и filter()

Начнем с краткого обзора функций `map()`, `filter()` и `zip()` — это основные встроенные инструменты для работы с коллекциями. Затем разберем, как достичь тех же результатов с помощью *включений* и *генераторов*.

map()

Согласно официальной документации Python (https://docs.python.org/3/library/functions.html#map),

> «`map(function, iterable, *iterables)`
>
> возвращает итератор, который применяет указанную функцию к каждому элементу итерируемого объекта, поочередно возвращая результаты. Если передано несколько итерируемых объектов, то функция должна принимать соответствующее количество аргументов и будет применяться ко всем объектам параллельно. Итерация останавливается, как только заканчивается самый короткий из итерируемых объектов».

Пока не будем вдаваться в тему итераторов — к ней мы еще вернемся в текущей главе. А сейчас посмотрим, как описанное работает в коде. Мы воспользуемся лямбда-выражением, которое принимает переменное количество позиционных аргументов и возвращает их в виде кортежа:

```
# map.example.txt
>>> map(lambda *a: a, range(3))                      # 1 итерируемый объект
<map object at 0x7f0db97adae0>                       # Неэффективно! Лучше list
>>> list(map(lambda *a: a, range(3)))                # 1 итерируемый объект
[(0,), (1,), (2,)]
>>> list(map(lambda *a: a, range(3), "abc"))         # 2 итерируемых объекта
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> list(map(lambda *a: a, range(3), "abc", range(4, 7)))  # 3 итерируемых объекта
[(0, 'a', 4), (1, 'b', 5), (2, 'c', 6)]
>>>  # map останавливается на самом коротком итерируемом объекте
>>> list(map(lambda *a: a, (), "abc"))            # пустой кортеж – самый короткий
[]
>>> list(map(lambda *a: a, (1, 2), "abc"))        # (1, 2) – самый короткий
[(1, 'a'), (2, 'b')]
>>> list(map(lambda *a: a, (1, 2, 3, 4), "abc"))  # "abc" – самый короткий
[(1, 'a'), (2, 'b'), (3, 'c')]
```

В примере видно, почему мы оборачиваем вызов `map()` в `list()`. Если мы не делаем это, то получаем строковое представление объекта `map`, которое показывает только тип и адрес в памяти, а в такой форме результат бесполезен.

Кроме того, можно заметить, как именно аргументы из итерируемых объектов подставляются в функцию: сначала берутся первые элементы из каждого итери-

руемого объекта, затем вторые и т. д. Еще один важный момент: функция `map()` завершает работу, как только заканчивается самый короткий из переданных ей объектов. Это очень удобно: не нужно подгонять все итерируемые объекты под одну длину — `map()` сама корректно остановится, не вызвав ошибку.

Рассмотрим более интересный пример. Допустим, у нас есть коллекция словарей, каждый из которых содержит вложенный словарь с успеваемостью студентов по предметам. Мы хотим отсортировать студентов по суммарной успеваемости. Однако текущая структура данных не позволяет применить функцию сортировки напрямую.

Чтобы решить задачу, мы применим прием *«декорирование — сортировка — раздекорирование»*, также известный как *преобразование Шварца*. Он был особенно популярен в ранних версиях Python, когда функция сортировки еще не поддерживала функцию `key`. Сейчас в большинстве случаев в нем уже нет необходимости, но в некоторых задачах он все еще полезен.

Декорировать объект — значит преобразовать его, добавив дополнительную информацию или обернув в другую структуру. *Раздекорировать* объект — значит вернуть его к исходной форме, убрав все добавленное.

Важно: этот прием не имеет отношения к декораторам Python — о них мы поговорим позже. Вот пример применения функции `map()`:

```
# decorate.sort.undecorate.py
from pprint import pprint

students = [
    dict(id=0, credits=dict(math=9, physics=6, history=7)),
    dict(id=1, credits=dict(math=6, physics=7, latin=10)),
    dict(id=2, credits=dict(history=8, physics=9, chemistry=10)),
    dict(id=3, credits=dict(math=5, physics=5, geography=7)),
]

def decorate(student):
    # создаем кортеж из двух элементов: суммы баллов по предметам
    # и данных о студенте
    return (sum(student["credits"].values()), student)

def undecorate(decorated_student):
    # отбрасываем сумму баллов, возвращаем только данные студента
    return decorated_student[1]

print(students[0])
print(decorate(students[0]))

students = sorted(map(decorate, students), reverse=True)
students = list(map(undecorate, students))
pprint(students)
```

Начнем с разбирательства того, что собой представляет объект `student`. Для этого выведем данные о первом студенте:

```
{'id': 0, 'credits': {'math': 9, 'physics': 6, 'history': 7}}
```

Как видите, они представлены словарем с двумя ключами: `id` и `credits`. Значение `credits` — это тоже словарь, в котором содержатся пары «предмет — оценка». Как мы уже обсуждали в главе 2, вызов `dict.values()` возвращает итерируемый объект, содержащий только значения словаря. Значит, выражение `sum(student["credits"].values())` для первого студента эквивалентно `sum((9, 6, 7))`.

Теперь выведем результат работы функции `decorate` для первого студента:

```
(22, {'id': 0, 'credits': {'math': 9, 'physics': 6, 'history': 7}})
```

Если таким образом декорировать данные обо всех студентах, то можно просто отсортировать полученный список кортежей по суммарной успеваемости. Для этого применим `map(decorate, students)`, отсортируем результат и затем раздекорируем список, вернув исходные словари.

Если вывести список `students` после выполнения всей цепочки, то получится следующий результат:

```
[{'credits': {'chemistry': 10, 'history': 8, 'physics': 9}, 'id': 2},
 {'credits': {'latin': 10, 'math': 6, 'physics': 7}, 'id': 1},
 {'credits': {'history': 7, 'math': 9, 'physics': 6}, 'id': 0},
 {'credits': {'geography': 7, 'math': 5, 'physics': 5}, 'id': 3}]
```

Как видите, список студентов действительно отсортирован по их суммарной успеваемости.

Если хотите лучше разобраться в преобразовании Шварца, то обратитесь к разделу Sorting HOW TO в официальной документации Python по адресу https://docs.python.org/3.12/howto/sorting.html#decorate-sort-undecorate.

Есть один важный нюанс, связанный с этапом сортировки. Что произойдет, если суммарная успеваемость двух студентов окажется одинаковой? В таком случае алгоритм сортировки попытается сравнить сами объекты студентов — и это не имеет смысла. В более сложных структурах подобное сравнение может привести к непредсказуемым результатам или даже ошибкам. Простое решение — вместо пары значений (`(сумма, объект)`) использовать три: сумму оценок (суммарную успеваемость), индекс студента в исходном списке и сам объект студента.

Такой подход гарантирует, что даже при равных суммах оценок сортировка будет происходить по уникальному индексу, а значит, порядок будет четко определенным и стабильным.

zip()

Мы уже обсуждали функцию `zip()` в предыдущих главах, так что сейчас просто дадим точное определение и затем покажем, как ее можно комбинировать с `map()`. Согласно официальной документации Python (https://docs.python.org/3/library/functions.html#zip),

> «`zip(*iterables, strict=False)`
>
> ...возвращает итератор кортежей, где каждый *i*-й кортеж содержит *i*-е элементы из всех переданных итерируемых объектов.
>
> Другой способ представить работу `zip()` — преобразование строк в столбцы, а столбцов в строки. Это похоже на транспонирование матрицы».

Рассмотрим пример:

```
# zip.grades.txt
>>> grades = [18, 23, 30, 27]
>>> avgs = [22, 21, 29, 24]
>>> list(zip(avgs, grades))
[(22, 18), (21, 23), (29, 30), (24, 27)]
>>> list(map(lambda *a: a, avgs, grades)) # эквивалентно zip
[(22, 18), (21, 23), (29, 30), (24, 27)]
```

Здесь мы объединяем средний балл каждого студента и его итоговую оценку за последний экзамен, используя `zip()`. Обратите внимание, как просто воспроизвести поведение `zip()` с помощью `map()` — это показано в двух последних строках примера. Как и в случае с `map()`, снова приходится использовать `list()`, чтобы увидеть результат.

По умолчанию `zip()` завершает работу, как только достигает конца самого короткого итерируемого объекта. Такое поведение может скрыть ошибки в данных и привести к багам. Например, допустим, мы хотим создать словарь, сопоставляющий имена студентов с их оценками. Если при вводе данных список оценок окажется короче списка имен, то данный факт останется незамеченным. Ниже приведен пример:

```
# zip.strict.txt
>>> students = ["Sophie", "Alex", "Charlie", "Alice"]
>>> grades = ["A", "C", "B"]
>>> dict(zip(students, grades))
{'Sophie': 'A', 'Alex': 'C', 'Charlie': 'B'}
```

Как видите, для "Alice" запись в словаре отсутствует — поведение zip() скрыло ошибку. Чтобы избежать таких ситуаций, в Python начиная с версии 3.10 в zip() добавлен параметр strict=True. Если указать его, то итерируемые объекты обязаны быть одинаковой длины, иначе будет выброшено исключение:

```
>>> dict(zip(students, grades, strict=True))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is shorter than argument 1
```

Кроме того, в модуле itertools есть функция zip_longest(). Она ведет себя как zip(), но продолжает работу до тех пор, пока не исчерпается самый длинный итерируемый объект. Более короткие итерируемые объекты дополняются значением, которое можно указать аргументом, — по умолчанию это None.

filter()

Согласно официальной документации Python (https://docs.python.org/3/library/functions.html#filter), предназначение функции filter() звучит так:

> «filter(function, iterable)
>
> создает итератор, содержащий только те элементы из iterable, для которых функция function возвращает значение True. Аргумент iterable может быть последовательностью, контейнером, поддерживающим итерацию, или итератором. Если function указана как None, то используется функция идентификации — иными словами, из итерируемого объекта исключаются все элементы, интерпретируемые как False».

Рассмотрим простой пример:

```
# filter.txt
>>> test = [2, 5, 8, 0, 0, 1, 0]
>>> list(filter(None, test))
[2, 5, 8, 1]
>>> list(filter(lambda x: x, test))     # эквивалентно предыдущему
[2, 5, 8, 1]
>>> list(filter(lambda x: x > 4, test)) # keep only items > 4
[5, 8]
```

Обратите внимание: второй вызов filter() делает то же самое, что и первый. Если передать функцию, которая принимает один аргумент и возвращает его же, то она будет возвращать True только для «истинных» значений. Такое поведение эквивалентно передаче None. Данный прием полезен для практики: если вы сможете воспроизвести встроенное поведение Python вручную и получить тот же результат, это будет означать, что вы действительно понимаете, как язык работает в подобной ситуации.

Итак, теперь вы знаете о `map()`, `zip()` и `filter()` (а также о множестве других полезных функций из стандартной библиотеки Python). Эти инструменты дают большую свободу в работе с последовательностями. Но они не единственный вариант. Далее вы познакомитесь с одним из самых мощных средств Python — *включениями*.

Включения

Включение (comprehension) позволяет легко выполнить операцию над каждым элементом коллекции и/или выбрать подмножество элементов, удовлетворяющих какому-либо условию. Эта конструкция заимствована из функционального языка программирования Haskell (https://www.haskell.org) и вместе с итераторами и генераторами придает Python характерные черты функционального стиля.

В Python доступны несколько видов включений: списковые, словарные и включения множеств. В этой главе мы сосредоточимся на списковых (list comprehensions). Поняв их, вы без труда освоите и остальные.

Начнем с простого примера: мы хотим получить список квадратов первых десяти натуральных чисел. Один из способов — использовать цикл `for` и добавлять квадрат в список на каждой итерации:

```
# squares.for.txt
>>> squares = []
>>> for n in range(10):
...     squares.append(n**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Этот способ не слишком элегантен: приходится сначала создавать пустой список. Используя функцию `map()`, можно добиться того же результата с помощью всего одной строки кода:

```
# squares.map.txt
>>> squares = list(map(lambda n: n**2, range(10)))
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

А теперь посмотрим, как получить тот же результат с помощью спискового включения:

```
# squares.comprehension.txt
>>> [n**2 for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Это решение гораздо проще читается, и нам больше не нужно использовать лямбда-выражение. Мы просто поместили цикл `for` внутрь квадратных скобок. Теперь отфильтруем только четные квадраты. Сначала вы увидите, как это сделать с помощью `map()` и `filter()`, а затем — с помощью спискового включения:

```
# even.squares.py
# использование map и filter
sq1 = list(
    map(lambda n: n**2, filter(lambda n: not n % 2, range(10)))
)
# эквивалентно, но с использованием списковых включений
sq2 = [n**2 for n in range(10) if not n % 2]
print(sq1, sq1 == sq2)  # выводит: [0, 4, 16, 36, 64] True
```

Теперь разница в читабельности становится очевидной. Списковое включение воспринимается куда легче. Такое выражение читается почти как фраза на естественном языке: «возьми квадрат числа `n`, если `n` четное, для всех `n` от 0 до 9».

Согласно официальной документации Python (https://docs.python.org/3.12/tutorial/datastructures.html#list-comprehensions), список, созданный с помощью включения, определяется так:

> «Списковое включение состоит из квадратных скобок, внутри которых находится выражение, за которым следует оператор `for`, а затем — ноль или более дополнительных `for` или `if`. В результате получится новый список, в котором выражение вычисляется в контексте последующих `for` и `if`».

Вложенные включения

Рассмотрим пример с вложенными циклами. Такие конструкции встречаются довольно часто, поскольку во многих алгоритмах приходится попарно перебирать элементы последовательности. Один цикл проходит по всей последовательности слева направо, а второй — тоже по ней, но начиная с позиции текущего элемента первого цикла. Таким образом, можно перебрать все уникальные пары, не имеющие повторений. Вот как это реализуется с помощью классического цикла `for`:

```
# pairs.for.loop.py
items = "ABCD"
pairs = []
for a in range(len(items)):
    for b in range(a, len(items)):
        pairs.append((items[a], items[b]))
```

Если вывести список `pairs`, то получится такой результат:

```
$ python pairs.for.loop.py
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'B'), ('B', 'C'),
('B', 'D'), ('C', 'C'), ('C', 'D'), ('D', 'D')]
```

Группы кортежей с одинаковой первой буквой — это все пары, начинающиеся с одной и той же позиции `a`. Теперь перепишем этот фрагмент как списковое включение:

```
# pairs.list.comprehension.py
items = "ABCD"
pairs = [
    (items[a], items[b])
    for a in range(len(items))
    for b in range(a, len(items))
]
```

Обратите внимание: цикл по `b` зависит от значения `a`, поэтому внутренний цикл (`for b in ...`) должен идти после внешнего (`for a in ...`). Если поменять их местами, то Python не поймет, что такое `a`, и возникнет ошибка `NameError`.

Добиться такого же результата можно и другим способом — с помощью функции combinations_with_replacement() из модуля itertools. Мы уже упоминали его в главе 3. Более подробная информация о нем доступна в официальной документации Python.

Фильтрация во включении

Списковые включения можно не только использовать для создания новых коллекций, но и дополнять фильтрацией. Начнем с примера на `filter()`: найдем все пифагоровы тройки, в которых катеты — целые числа меньше 10. *Пифагорова тройка* — это набор из трех целых чисел (a, b, c), удовлетворяющих уравнению $a^2 + b^2 = c^2$.

Разумеется, проверять каждую комбинацию по два раза мы не хотим, поэтому используем прием, аналогичный тому, который мы применили в предыдущем примере:

```
# pythagorean.triple.py
from math import sqrt

# так мы создаем все возможные пары
mx = 10
triples = [
    (a, b, sqrt(a**2 + b**2))
    for a in range(1, mx)
    for b in range(a, mx)
]
# так мы отфильтруем все непифагоровы тройки
triples = list(
    filter(lambda triple: triple[2].is_integer(), triples)
)
print(triples) # выводит: [(3, 4, 5.0), (6, 8, 10.0)]
```

В этом фрагменте мы создаем список *троек* (`triples`) — каждая состоит из двух целых чисел (катетов) и длины гипотенузы, вычисленной по теореме Пифагора. Например, если `a = 3`, `b = 4`, то `c = 5.0` и тройка будет `(3, 4, 5.0)`; а если `a = 5`, `b = 7`, то `c ≈ 8.60` и тройка будет `(5, 7, 8.602325267042627)`.

Далее мы фильтруем список, оставляя только те тройки, в которых гипотенуза — целое число. Это достигается проверкой: `c.is_integer()` должно возвращать `True`. Таким образом, тройка `(3, 4, 5.0)` останется, а `(5, 7, 8.6...)` будет отброшена.

Такой результат нас устраивает, но остается один нюанс: в итоговом кортеже два значения — целые, а одно — число с плавающей запятой. Нам бы хотелось, чтобы все три были целыми числами. Этого легко добиться с помощью `map()`:

```
# pythagorean.triple.int.py
from math import sqrt

mx = 10
triples = [
    (a, b, sqrt(a**2 + b**2))
    for a in range(1, mx)
    for b in range(a, mx)
]
triples = filter(lambda triple: triple[2].is_integer(), triples)
# так мы преобразуем третье число в кортежах в целое
triples = list(
    map(lambda triple: triple[:2] + (int(triple[2]),), triples)
)
print(triples) # выводит: [(3, 4, 5), (6, 8, 10)]
```

Обратите внимание на дополнительный шаг, который мы добавили. Для каждой тройки в списке `triples` мы берем только первые два элемента, а затем объединяем их с кортежем из одного элемента — целочисленным значением гипотенузы, которую ранее вычисляли как число с плавающей запятой. Этот подход работает, но код получается довольно громоздким. То же самое можно выразить значительно проще, используя списковое включение:

```
# pythagorean.triple.comprehension.py
from math import sqrt

# этот шаг такой же, как и раньше
mx = 10
triples = [
    (a, b, sqrt(a**2 + b**2))
    for a in range(1, mx)
    for b in range(a, mx)
]
# здесь мы объединяем filter и map в одном ЧИСТОМ списковом включении
triples = [
    (a, b, int(c)) for a, b, c in triples if c.is_integer()
]
print(triples) # выводит: [(3, 4, 5), (6, 8, 10)]
```

Такой вариант чище, короче и легче читается. Но и здесь остается простор для улучшения. Мы все еще тратим ресурсы памяти, создавая список с множеством троек, часть из которых в итоге отбрасывается. Эту проблему можно решить, объединив вычисление и фильтрацию в одном списковом включении:

```
# pythagorean.triple.walrus.py
from math import sqrt

# этот шаг такой же, как и раньше
mx = 10
# мы можем объединить генерацию и фильтрацию в одном включении
triples = [
    (a, b, int(c))
    for a in range(1, mx)
    for b in range(a, mx)
    if (c := sqrt(a**2 + b**2)).is_integer()
]
print(triples) # выводит: [(3, 4, 5), (6, 8, 10)]
```

Теперь результат выглядит действительно элегантно. Тройки создаются и фильтруются в одном включении, и память не расходуется на хранение неподходящих значений. Обратите внимание: здесь мы использовали выражение присваивания (`:=`), чтобы не пересчитывать значение `sqrt(a**2 + b**2)` дважды.

Словарные включения

Они устроены точно так же, как списковые, только результатом становится словарь, а не список. Различие лишь в синтаксисе: для создания словаря используется пара `ключ: значение` вместо одного выражения. Пример ниже даст полное представление о том, как это работает:

```
# dictionary.comprehensions.py
from string import ascii_lowercase

lettermap = {c: k for k, c in enumerate(ascii_lowercase, 1)}
```

Если вывести словарь `lettermap`, то можно увидеть такой результат:

```
$ python dictionary.comprehensions.py
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8, 'i': 9, 'j': 10,
'k': 11, 'l': 12, 'm': 13, 'n': 14, 'o': 15, 'p': 16, 'q': 17, 'r': 18, 's': 19,
't': 20, 'u': 21, 'v': 22, 'w': 23, 'x': 24, 'y': 25, 'z': 26}
```

В этом примере мы используем функцию `enumerate()` для перебора всех строчных букв английского алфавита (используется строка `string.ascii_lowercase`). Затем на основе пар `(индекс, буква)` строится словарь, в котором буквы становятся ключами, а номера — значениями. Обратите внимание: синтаксис очень похож на обычное создание словаря вручную, только записан в компактной форме.

Существует и другой способ достичь того же результата:

```
lettermap = dict((c, k) for k, c in enumerate(ascii_lowercase, 1))
```

В этом примере мы передаем генераторное выражение (подробнее о них поговорим чуть позже в данной главе) в конструктор `dict()`.

Словари в Python не допускают повторяющихся ключей, как видно из следующего примера:

```
# dictionary.comprehensions.duplicates.py
word = "Hello"
swaps = {c: c.swapcase() for c in word}
print(swaps) # выводит: {'H': 'h', 'e': 'E', 'l': 'L', 'o': 'O'}
```

Мы создаем словарь, в котором ключами становятся буквы из строки `"Hello"`, а значениями — те же буквы, но с противоположным регистром. Обратите внимание: в словаре будет только одна пара `"l": "L"`, несмотря на то что в исходной строке буква `"l"` встречается дважды. Конструктор `dict()` не выдает ошибку — просто перезаписывает значение для ключа, если тот уже есть. Остается последнее вхождение. В примере ниже вы можете увидеть еще более наглядный пример — теперь значением каждой буквы будет ее индекс в строке:

```
# dictionary.comprehensions.positions.py
word = "Hello"
positions = {c: k for k, c in enumerate(word)}
print(positions) # выводит: {'H': 0, 'e': 1, 'l': 3, 'o': 4}
```

Обратите внимание: значением для буквы `"l"` будет `3`, а не `2`. Пара `"l": 2` была перезаписана, и в словаре осталась только `"l": 3`.

Включения множеств

Они работают по тому же принципу, что и списковые и словарные включения. Приведем простой пример:

```
# set.comprehensions.py
word = "Hello"
letters1 = {c for c in word}
letters2 = set(c for c in word)
print(letters1)             # выводит[1]: {'H', 'o', 'e', 'l'}
print(letters1 == letters2) # выводит: True
```

Как и в случае со словарями, дубликаты в множествах не допускаются, поэтому в результате получается множество, содержащее только четыре уникальные буквы. Обратите внимание: выражения, присвоенные переменным `letters1` и `letters2`, создают эквивалентные множества.

---

[1] Порядок в выводе может отличаться, так как множество (set) — неупорядоченная последовательность. — *Примеч. науч. ред.*

Синтаксис включения для `letters1` похож на словарное включение. Разница лишь в том, что словари требуют пару «ключ: значение», тогда как множества — только выражение. В выражении `letters2` мы передаем генераторное выражение в конструктор `set()` — это альтернативный способ получить тот же результат.

Генераторы

Как мы уже говорили, генераторы основаны на концепции *итерации*. Они позволяют создавать элегантные конструкции, эффективные как по скорости, так и по использованию памяти.

В Python есть два типа генераторов:

- *генераторные функции* — выглядят как обычные функции, но вместо `return` используют оператор `yield`. Это позволяет приостановить выполнение функции и сохранять ее состояние между вызовами;
- *генераторные выражения* — очень похожи на списковые включения, с той разницей, что вместо списка возвращают объект-генератор, который выдает значения по одному — по мере запроса, а не сразу.

Генераторные функции

Эти функции во всем похожи на обычные, за одним исключением: вместо того чтобы собрать все результаты и вернуть их сразу, они автоматически превращаются в итераторы, которые выдают значения по одному — при каждом вызове.

Представьте, что вас попросили считать от 1 до 1 000 000. Вы начинаете, но вас прерывают. Спустя время просят продолжить. Если вы помните, на каком числе остановились, то легко продолжите — например, с 31 416. Главное, что вам не нужно запоминать все предыдущие числа и хранить их где-либо. Генераторы ведут себя точно так же.

Рассмотрим следующий код:

```
# first.n.squares.py
def get_squares(n):     # классический вариант с функцией
    return [x**2 for x in range(n)]

print(get_squares(10))

def get_squares_gen(n): # вариант с генератором
    for x in range(n):
        yield x**2      # используем yield, а не return

print(list(get_squares_gen(10)))
```

Обе функции при преобразовании в список возвращают одинаковый результат: `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`. Но работают они по-разному. Функция `get_squares()` — обычная: она собирает все квадраты чисел от 0 до `n` и возвращает

список. А вот `get_squares_gen()` — генераторная функция: она выдает значения по одному. Как только интерпретатор доходит до строки с `yield`, выполнение функции приостанавливается до следующего запроса. Результаты совпадают только потому, что мы передаем `get_squares_gen()` в конструктор `list()`, который полностью исчерпывает генератор, запрашивая элементы до тех пор, пока не возникнет исключение `StopIteration`. Пример ниже поможет вам увидеть, как все это выглядит в коде:

```
# first.n.squares.manual.py
def get_squares_gen(n):
    for x in range(n):
        yield x**2

squares = get_squares_gen(4)  # создает объект-генератор
print(squares)        # <generator object get_squares_gen at 0x10dd...>
print(next(squares))  # выводит: 0
print(next(squares))  # выводит: 1
print(next(squares))  # выводит: 4
print(next(squares))  # выводит: 9
# дальше возникает StopIteration, генератор исчерпан,
# любые последующие вызовы next будут выбрасывать StopIteration
print(next(squares))
```

При каждом вызове функции `next()` для генераторного объекта выполнение генератора либо запускается (в случае первого вызова), либо возобновляется с места последней приостановки (для всех последующих вызовов). При первом вызове `next()` мы получаем `0` — квадрат нуля, затем `1`, потом `4`, затем `9`, и на этом цикл `for` заканчивается (так как `n = 4`). Генератор завершает выполнение. Будь это обычная функция, она просто вернула бы `None`. Но, чтобы соответствовать протоколу итерации, генератор вместо этого вызывает исключение `StopIteration` — это сигнал, что итерация завершена.

Этот механизм объясняет, как устроен цикл `for` в Python. Когда вы пишете `for k in range(n)`, Python «за кадром» получает итератор с помощью `iter(range(n))`, а затем многократно вызывает `next()`, пока не получит `StopIteration`. Это поведение встроено в каждую итерационную конструкцию Python.

Именно благодаря этому встроенному механизму генераторы становятся особенно мощным инструментом: написав генератор один раз, можно подключать его к любому итерационному контексту — будь то цикл, списковое включение, функция `sum()` или любой другой механизм, работающий с итераторами.

На данном этапе вы, скорее всего, задаетесь вопросом: зачем вообще использовать генератор, если можно написать обычную функцию? Ответ прост: экономия времени и, что особенно важно, памяти.

О производительности мы поговорим подробнее позже, а сейчас сосредоточимся на одном ключевом моменте: иногда генераторы позволяют выполнить операцию, которая в принципе невозможна при работе с обычным списком. Представьте, что вам нужно перебрать все перестановки элементов в последовательности. Если длина этой последовательности — *N*, то количество возможных

перестановок равно $N!$. Например: при $N = 10$ получаем 3 628 800 вариантов, а при $N = 20$ — уже 2 432 902 008 176 640 000 перестановок. Такое количество растет факториально.

Теперь представьте обычную функцию, которая пытается сгенерировать все эти перестановки, сохранить их в списке и вернуть. При 10 элементах это, скорее всего, займет несколько секунд. А при 20 выполнить такую операцию просто невозможно: потребуются миллиарды гигабайт памяти и тысячи лет вычислений.

В свою очередь, генераторная функция может начать вычисление и сразу же вернуть первую перестановку, затем вторую и т. д. Конечно, вы не успеете обработать все — их слишком много, — но, по крайней мере, сможете работать с частью результата. Иногда объем данных, по которым нужно пройтись, настолько велик, что удерживать их все в памяти в виде списка невозможно. В таких случаях генераторы незаменимы — они позволяют делать то, что иначе было бы недостижимо.

Поэтому, чтобы сэкономить память (и время), используйте генераторы, когда это возможно.

Кстати, в генераторных функциях можно использовать оператор `return`. Он вызывает исключение `StopIteration`, тем самым корректно завершает итерацию. Если бы `return` пытался вернуть значение, это нарушило бы протокол итерации. Однако Python устроен последовательно: он просто завершает генератор, и это делает работу с итерациями удобной и безопасной. Рассмотрим короткий пример:

```
# gen.yield.return.py
def geometric_progression(a, q):
    k = 0
    while True:
        result = a * q**k
        if result <= 100000:
            yield result
        else:
            return
        k += 1

for n in geometric_progression(2, 5):
    print(n)
```

В этом примере генератор выдает все члены геометрической прогрессии: a, aq, aq^2, aq^3 и т. д. Как только очередной член становится больше 100 000, генератор завершает работу с помощью оператора `return`. При запуске такой код выведет следующий результат:

```
$ python gen.yield.return.py
2
10
50
250
1250
6250
31250
```

Следующий член прогрессии был бы равен 156 250 — это уже слишком много, поэтому он не попал в результат.

Другие инструменты, помимо функции next()

У генераторных объектов есть дополнительные методы, которые позволяют управлять их поведением: `send()`, `throw()` и `close()`. С помощью `send()` можно передавать значения обратно в генератор, а `throw()` и `close()` предназначены для вызова исключения внутри генератора и завершения его работы. Эти возможности относятся к более продвинутому уровню программирования, поэтому подробно мы их рассматривать не будем. Однако `send()` все же стоит уделить немного времени и показать его работу на простом примере:

```
# gen.send.preparation.py
def counter(start=0):
    n = start
    while True:
        yield n
        n += 1

c = counter()
print(next(c)) # выводит: 0
print(next(c)) # выводит: 1
print(next(c)) # выводит: 2
```

Такой генератор создает объект, который будет выполняться бесконечно. Вы можете вызывать у него `next()` сколько угодно раз — он никогда сам не завершится. А что если вы хотите остановить его в какой-то момент? Один из вариантов — использовать глобальную переменную для управления условием в цикле `while`:

```
# gen.send.preparation.stop.py
stop = False
def counter(start=0):
    n = start
    while not stop:
        yield n
        n += 1

c = counter()
print(next(c)) # выводит: 0
print(next(c)) # выводит: 1
stop = True
print(next(c)) # вызывает StopIteration
```

Изначально мы присваиваем переменной `stop` значение `False`, и до тех пор, пока не изменим его на `True`, генератор будет продолжать работать бесконечно. Как только `stop` присваивается значение `True`, цикл `while` завершается, и следующий вызов `next()` приведет к выбросу исключения `StopIteration`. Такой прием действительно работает, но он неидеален. Функция зависит от внешней переменной, что может привести к ошибкам. Например, генератор может быть случайно

остановлен, если другая часть программы изменит глобальное значение `stop`. Хорошей практикой считается писать функции так, чтобы они были независимы от внешнего состояния и не полагались на глобальные переменные.

Метод генератора `send()` принимает один аргумент, который передается обратно в генератор как значение выражения `yield`. Это позволяет использовать `send()` для передачи управляющего сигнала — например, флага остановки:

```
# gen.send.py
def counter(start=0):
    n = start
    while True:
        result = yield n            # A
        print(type(result), result) # Б
        if result == "Q":
            break
        n += 1

c = counter()
print(next(c))                      # В
print(c.send("Wow!"))               # Г
print(next(c))                      # Д
print(c.send("Q"))                  # Е
```

При запуске этот код выдаст следующий результат:

```
$ python gen.send.py
0
<class 'str'> Wow!
1
<class 'NoneType'> None
2
<class 'str'> Q
Traceback (most recent call last):
  File "gen.send.py", line 16 , in <module>
    print(c.send("Q")) # F
          ^^^^^^^^^^
StopIteration
```

Пройдемся по коду строка за строкой, чтобы вы могли точно понять, что происходит.

Сначала генератор запускается путем вызова `next()` (`# В`). Внутри генератора переменной `n` присваивается начальное значение. Генератор входит в цикл `while`, выполнение доходит до `yield` и приостанавливается (`# А`), возвращая значение `n` (в данном случае `0`) вызывающей стороне. Значение `0` выводится в консоль.

Затем вызывается метод `send()` (`# Г`). Выполнение генератора возобновляется, переменной `result` присваивается строка `"Wow!"` (все еще точка `# А`), и в консоли выводятся ее тип и значение (`# Б`). Поскольку `result` не равна `"Q"`, то переменная `n` увеличивается на 1 и выполнение возвращается в начало цикла. Условие

while по-прежнему истинно, начинается новая итерация. Генератор снова доходит до yield n (снова # A), приостанавливается и возвращает n, равное 1, вызывающему коду. В консоль выводится число 1.

На следующем шаге вызывается next() (# Д). Генератор возобновляется на том же месте (# A), но в send() ничего не передается, поэтому yield n возвращает None (поведение такое же, как у обычной функции без return). Переменная result становится равной None, и снова выводятся ее тип и значение (# Б). Далее все как прежде: result не "Q", n увеличивается на 1, цикл продолжается, и n = 2 снова выдается через yield. В консоль выводится 2.

Теперь снова вызывается send() (# E), на этот раз передается значение "Q". Генератор возобновляется, result становится "Q" (# A), и значение выводится на консоль (# Б). При проверке условия result == "Q" выражение возвращает True, и цикл while прерывается с помощью break. Генератор естественным образом завершает выполнение, что вызывает исключение StopIteration. В последних строках консоли можно увидеть трассировку этого исключения.

Этот механизм на первый взгляд может показаться непростым, так что если он пока не до конца ясен, то не переживайте: вы можете продолжать читать книгу и вернуться к примеру позже.

Метод send() позволяет реализовать интересные шаблоны работы с генераторами. Кстати, его можно использовать даже для запуска генератора с самого начала, если вызвать его с аргументом None.

Выражение yield from

Еще одна интересная конструкция — выражение yield from. С его помощью можно делегировать выдачу значений другому итератору. Это особенно полезно в продвинутых сценариях, где один генератор должен передавать значения от другого. Рассмотрим простой пример:

```
# gen.yield.for.py
def print_squares(start, end):
    for n in range(start, end):
        yield n**2

for n in print_squares(2, 5):
    print(n)
```

Код выведет числа 4, 9 и 16 — каждое на новой строке. К настоящему моменту вы, скорее всего, уже разбираетесь, как это работает, но все же напомним, что происходит. Внешний цикл for получает итератор от вызова print_squares(2, 5) и вызывает у него next(), пока итерация не завершится. Каждый раз, когда генератор вызывается, выполнение приостанавливается (и затем возобновляется) в строке yield n**2, возвращая квадрат текущего значения n. Теперь посмотрим, как можно записать тот же результат с помощью yield from:

```
# gen.yield.from.py
def print_squares(start, end):
    yield from (n**2 for n in range(start, end))

for n in print_squares(2, 5):
    print(n)
```

Результат остается тем же. Но теперь видно, что `yield from` делегирует итерацию *подгенератору* — в данном случае генераторному выражению `(n**2 for n in range(...))`. Все значения, которые выдает этот подгенератор, непосредственно передаются вызывающему коду. Такой вариант короче и к тому же проще читается.

Генераторные выражения

Помимо генераторных функций, генераторы можно создавать и с помощью *генераторных выражений*. Их синтаксис почти такой же, как у списковых включений — только вместо квадратных скобок используются круглые.

Генераторное выражение выдает ту же последовательность значений, что и соответствующее списковое включение. Но есть важное различие: вместо того чтобы создавать список в памяти, генератор выдает элементы по одному — по мере запроса. Важно помнить: пройтись по генератору можно только один раз. После этого он будет исчерпан — повторная итерация невозможна.

Рассмотрим пример вывода:

```
# generator.expressions.txt
>>> cubes = [k**3 for k in range(10)]      # обычный список
>>> cubes
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> type(cubes)
<class 'list'>
>>> cubes_gen = (k**3 for k in range(10))  # создаем генератор
>>> cubes_gen
<generator object <genexpr> at 0x7f08b2004860>
>>> type(cubes_gen)
<class 'generator'>
>>> list(cubes_gen)  # этот вызов исчерпывает генератор
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> list(cubes_gen)  # дальше генератор ничего не выдает
[]
```

Как видите, при попытке вывести `cubes_gen` мы получаем объект генератора. Чтобы увидеть значения, которые он выдает, можно использовать цикл `for`, вручную вызвать `next()` несколько раз или — как в примере — просто передать генератор в `list()`.

Важно понимать: после того как генератор исчерпан, получить из него те же элементы повторно невозможно. Если вы хотите начать сначала, то должны создать генератор заново.

В следующих примерах покажем, как можно воспроизвести поведение `map()` и `filter()` с помощью генераторных выражений. Сначала — эквивалент `map()`:

```
# gen.map.py
def adder(*n):
    return sum(n)

s1 = sum(map(adder, range(100), range(1, 101)))
s2 = sum(adder(*n) for n in zip(range(100), range(1, 101)))
```

Здесь переменные `s1` и `s2` содержат сумму значений `adder(0, 1)`, `adder(1, 2)`, `adder(2, 3)` и т. д., то есть фактически `sum(1, 3, 5, ...)`. На наш взгляд, читать синтаксис генераторного выражения намного проще.

Теперь посмотрим на `filter()`:

```
# gen.filter.py
cubes = [x**3 for x in range(10)]
odd_cubes1 = filter(lambda cube: cube % 2, cubes)
odd_cubes2 = (cube for cube in cubes if cube % 2)
```

В этом случае `odd_cubes1` и `odd_cubes2` эквивалентны — обе переменные содержат последовательность нечетных кубов. И генераторное выражение вновь выглядит проще и понятнее, особенно когда логика усложняется:

```
# gen.map.filter.py
N = 20
cubes1 = map(
    lambda n: (n, n**3),
    filter(lambda n: n % 3 == 0 or n % 5 == 0, range(N)),
)
cubes2 = ((n, n**3) for n in range(N) if n % 3 == 0 or n % 5 == 0)
```

В этом примере создаются два итератора — `cubes1` и `cubes2`. Оба возвращают одинаковую последовательность кортежей (n, n^3), где n — число, кратное 3 или 5. Если преобразовать любой из них в список и вывести, то результат будет таким: `[(0, 0), (3, 27), (5, 125), (6, 216), (9, 729), (10, 1000), (12, 1728), (15, 3375), (18, 5832)]`.

Обратите внимание: генераторное выражение читается значительно легче. В простых примерах этот момент может показаться несущественным, но стоит логике хоть немного усложниться — и преимущество генераторного синтаксиса становится очевидным: он компактнее, проще и элегантнее.

Теперь вопрос: в чем разница между этими строками кода?

```
# sum.example.py
s1 = sum([n**2 for n in range(10**6)])
s2 = sum((n**2 for n in range(10**6)))
s3 = sum(n**2 for n in range(10**6))
```

С формальной точки зрения все три строки возвращают одну и ту же сумму. Выражения для `s2` и `s3` эквивалентны, поскольку круглые скобки в `s2` избыточ-

ны — Python интерпретирует их как генераторное выражение, даже без внешних скобок. Оба случая — это генераторные выражения, передаваемые в `sum()`.

Выражение для вычисления `s1` устроено иначе. Здесь в `sum()` передается результат спискового включения. Это приводит к лишним затратам времени и памяти: сначала создается список из миллиона элементов, который хранится в памяти, затем он передается в `sum()`, а после выполнения вычисления — отбрасывается. Гораздо лучше использовать генераторное выражение: не нужно ждать, пока создастся весь список, и не нужно держать в памяти миллион значений — `sum()` просто перебирает их по одному.

Так что *будьте внимательны с лишними скобками в выражениях*. Такие мелочи легко пропустить, но они могут заметно повлиять на производительность. Рассмотрим пример:

```
# sum.example.2.py
s = sum([n**2 for n in range(10**10)])
# здесь программа завершается из-за большого потребления памяти
# s = sum(n**2 for n in range(10**10))
# а этот вариант с генератором успешно выполнится
print(s)  # выводит: 333333333283333333335000000000
```

Если запустить этот код, то результат будет таким:

```
$ python sum.example.2.py
Killed
```

Если закомментировать первую строку и раскомментировать вторую, то вывод будет выглядеть так:

```
$ python sum.example.2.py
333333333283333333335000000000
```

Разница между этими строками в том, что в первой Python пытается создать список квадратов десяти миллиардов чисел, чтобы передать его в `sum()`. Это огромный список, и памяти не хватает — процесс завершается операционной системой.

Во втором случае мы убираем квадратные скобки и `sum()` получает генератор. Он выдает значения по одному: `0`, `1`, `4`, `9` и т. д. — и суммирует их, не храня всю последовательность в памяти.

Несколько слов о производительности

Почти всегда есть несколько способов получить один и тот же результат. Можно использовать любые комбинации `map()`, `zip()` и `filter()`, выбрать списковое включение или генератор, а можно и вовсе обойтись циклом `for`. При выборе подхода важную роль играет читабельность. Включения и генераторные выражения зачастую воспринимаются легче, чем вложенные конструкции с `map()`

и `filter()`. А в более сложных случаях лучше подходят генераторные функции или обычные циклы.

Но, помимо читабельности, есть еще один важный фактор — производительность. Когда мы сравниваем разные подходы, нужно учитывать два аспекта: память и время.

Память — это объем оперативной памяти, который займут ваши структуры данных. Прежде чем выбрать тот или иной способ, спросите себя: действительно ли вам нужен список (или кортеж)? Может быть, достаточно генератора? Если он подойдет — выбирайте его. Он сэкономит значительный объем памяти. То же самое касается функций: если результат не обязательно должен быть списком или кортежем, то лучше преобразовать обычную функцию в генераторную.

Иногда, конечно, без списков не обойтись. Например, если алгоритм использует несколько указателей, проходящих по одной и той же последовательности, или когда требуется многократная итерация. По генератору — будь то функция или выражение — можно пройти только один раз, после чего он исчерпывается. В таких случаях он будет неудачным выбором.

С оценкой времени выполнения все немного сложнее, чем с объемом памяти. Время зависит от множества факторов, и нельзя с полной уверенностью утверждать, что *X всегда быстрее, чем Y*. Тем не менее по результатам тестов на современной версии Python можно сказать следующее: `map()`, списковые включения и генераторные выражения показывают сопоставимую производительность. Циклы `for` в среднем заметно медленнее.

Чтобы полностью понять, почему это так, нужно разобраться в устройстве интерпретатора Python, — а это узкая техническая тема, выходящая за рамки нашей книги. Достаточно сказать, что `map()` и включения выполняются на уровне кода C внутри интерпретатора, в то время как цикл `for` интерпретируется как байт-код Python внутри виртуальной машины. Это существенно влияет на скорость — в пользу включений и `map()`.

Существует несколько реализаций Python, но основная и самая распространенная — это CPython (https://github.com/python/cpython). Она написана на языке C — одном из самых мощных и до сих пор широко используемых языков программирования.

Далее мы проведем несколько простых экспериментов, чтобы подтвердить правильность вышеуказанных утверждений. Напишем короткий фрагмент кода, в котором будем собирать результаты вызова `divmod(a, b)` для набора целочисленных пар `(a, b)`. Измерять время будем с помощью функции `time()` из модуля `time`:

```
# performance.py
from time import time
mx = 5000
t = time()  # старт времени для цикла for
```

```
floop = []
for a in range(1, mx):
    for b in range(a, mx):
        floop.append(divmod(a, b))
print("for loop: {:.4f} s".format(time() - t))         # затраченное время

t = time()  # старт времени для спискового включения
compr = [divmod(a, b) for a in range(1, mx) for b in range(a, mx)]
print("list comprehension: {:.4f} s".format(time() - t))

t = time()  # старт времени для генераторного выражения
gener = list(
    divmod(a, b) for a in range(1, mx) for b in range(a, mx)
)
print("generator expression: {:.4f} s".format(time() - t))  # затраченное время
```

Как видите, мы создаем три списка: `floop`, `compr` и `gener`. При запуске кода мы получаем следующий результат:

```
$ python performance.py
for loop: 2.3832 s
list comprehension: 1.6882 s
generator expression: 1.6525 s
```

Списковое включение выполняется примерно за 71 % времени, которое требуется циклу `for`. Генераторное выражение немного быстрее — около 69 % от времени цикла. Разница между списковым включением и генератором незначительна, и если запустить пример несколько раз, то легко получить ситуацию, где включение работает быстрее, чем генератор.

Стоит отметить: внутри тела цикла `for` мы вызываем `append()` для добавления элементов в список. Это значит, что интерпретатор Python время от времени вынужден перераспределять память, чтобы вместить новые элементы. Мы предположили, что предварительное создание списка нужной длины (например, из нулей) и дополнение его по индексам может ускорить цикл, однако на практике это не дало преимущества. Можете попробовать сами — для этого потребуется заранее выделить `mx * (mx - 1) // 2` элементов.

Метод измерения времени, который мы использовали здесь, довольно упрощенный. В главе 11 мы обсудим более точные способы измерения производительности и профилирования кода.

Теперь рассмотрим похожий пример, в котором сравниваются цикл `for` и вызов `map()`:

```
# performance.map.py
from time import time

mx = 2 * 10**7
```

```
t = time()
absloop = []
for n in range(mx):
    absloop.append(abs(n))
print("for loop: {:.4f} s".format(time() - t))

t = time()
abslist = [abs(n) for n in range(mx)]
print("list comprehension: {:.4f} s".format(time() - t))

t = time()
absmap = list(map(abs, range(mx)))

print("map: {:.4f} s".format(time() - t))
```

Этот пример концептуально похож на предыдущий. Единственное различие в том, что теперь мы вызываем функцию `abs()` вместо `divmod()` и используем один цикл вместо вложенных. При выполнении код дает такой результат:

```
$ python performance.map.py
for loop: 1.9009 s
list comprehension: 1.0973 s
map: 0.5862 s
```

На этот раз метод `map()` оказался самым быстрым: он выполнился примерно за 53 % от времени, потребовавшегося списковому включению, и за 31 % от времени выполнения цикла `for`.

Эти простые эксперименты дают общее представление о сравнительной скорости циклов `for`, списковых включений, генераторных выражений и функции `map()`. Тем не менее не стоит слишком сильно полагаться на эти результаты: они получены в упрощенных условиях, а точное сравнение времени выполнения — задача далеко не тривиальная. На результат может повлиять множество факторов: от фоновых процессов на компьютере до характеристик оборудования, операционной системы и версии Python.

Тем не менее очевидно, что цикл `for` — самый медленный из всех рассмотренных вариантов. Далее мы обсудим, почему все равно часто используем его, несмотря на проигрыш в скорости.

Не увлекайтесь включениями и генераторами

Мы уже убедились, насколько мощными могут быть включения и генераторные выражения. Однако важно помнить: чем больше логики вы пытаетесь уместить в одну конструкцию, тем труднее ее читать, понимать и — в перспективе — сопровождать или изменять.

В «Дзен Python» есть несколько принципов, которые стоит учитывать при работе с оптимизированным кодом:

```
>>> import this
...
Явное лучше, чем неявное.
Простое лучше, чем сложное.
...
Читабельность имеет значение.
...
Если реализацию сложно объяснить – идея плоха.
...
```

Списковые включения и генераторные выражения часто менее явны, чем обычный код, и потому могут быть трудными для восприятия, объяснения и поддержки. Иногда приходится буквально «разворачивать» выражение изнутри наружу, чтобы разобраться, что происходит.

В качестве примера вернемся к пифагоровым тройкам — тройкам положительных целых чисел (a, b, c), для которых выполняется равенство $a^2 + b^2 = c^2$. Мы уже рассчитывали такие тройки в подразделе «Фильтрация во включении» (см. выше), но сделали это не самым эффективным способом: перебрали все пары чисел ниже определенного порога, вычислили гипотенузу и отфильтровали те, которые не образуют корректную пифагорову тройку.

Более эффективный способ генерации пифагоровых троек заключается в следующем. Вместо того чтобы перебирать все возможные пары чисел и фильтровать неподходящие, лучше сразу генерировать корректные пифагоровы тройки. Существует множество формул, позволяющих это сделать. Здесь мы воспользуемся *евклидовой формулой*. Согласно ей каждая тройка (a, b, c), где $a = m^2 - n^2$, $b = 2mn$, а $c = m^2 + n^2$, при положительных целых m и n таких, что $m > n$, является пифагоровой. Например, при $m = 2$, $n = 1$ получается наименьшая тройка (3, 4, 5).

Но тут есть один нюанс: тройка (6, 8, 10) — тоже пифагорова ($6^2 + 8^2 = 10^2$), однако ее можно получить из (3, 4, 5), умножив каждый элемент на 2. То же справедливо и для (9, 12, 15), (12, 16, 20) и вообще всех троек, записываемых как ($3k$, $4k$, $5k$) при положительном целом $k > 1$.

Тройку, которую нельзя получить умножением элементов другой тройки на какое-либо число k, называют *примитивной*. Иначе говоря, тройка примитивна, если ее три элемента — *взаимно простые*. Два числа называются взаимно простыми, если не имеют общих простых делителей, то есть их *наибольший общий делитель (НОД)* равен 1. Например, 3 и 5 — взаимно простые, а 3 и 6 — нет (общий делитель — 3).

Согласно евклидовой формуле если числа m и n — взаимно простые и разность $m - n$ — нечетная, то соответствующая им тройка будет *примитивной*.

В следующем примере мы напишем генераторное выражение, которое будет вычислять все примитивные пифагоровы тройки с гипотенузой $c \le N$. По формуле $c = m^2 + n^2$, поэтому нам нужно отобрать только те пары m и n, для которых $m^2 + n^2 \le N$. Если $n = 1$, то получается, что $m^2 \le N - 1$, а значит, верхнюю границу для m можно приближенно оценить как $m \le N^{1/2}$.

Подведем итог: m должно быть больше n, числа m и n — взаимно простые, а их разность $m - n$ — нечетная. Кроме того, чтобы избежать лишних вычислений, установим верхнюю границу для m равной floor(sqrt(N)) + 1.

Функция floor для вещественного числа x возвращает наибольшее целое число n такое, что $n \le x$. Например, floor(3.8) = 3, floor(13.1) = 13. Выражение floor(sqrt(N)) + 1 означает, что мы берем целую часть квадратного корня из N и добавляем небольшой запас, чтобы наверняка не пропустить возможные значения.

Теперь переведем все это в код — шаг за шагом. Начнем с простой функции `gcd()`, реализующей *алгоритм Евклида*:

```
# functions.py
def gcd(a, b):
    """Вычисляет наибольший общий делитель (НОД) чисел a и b."""
    while b != 0:
        a, b = b, a % b
    return a
```

Объяснение алгоритма Евклида легко найти в Интернете, поэтому мы не будем останавливаться на нем здесь — нам важно сосредоточиться на генераторном выражении. Следующий шаг — использовать полученные ранее знания, чтобы сгенерировать список примитивных пифагоровых троек:

```
# pythagorean.triple.generation.py
from functions import gcd
N = 50
triples = sorted(                                       # 1
    (
        (a, b, c) for a, b, c in (                      # 2
           ((m**2 - n**2), (2 * m * n), (m**2 + n**2))  # 3
            for m in range(1, int(N**.5) + 1)           # 4
            for n in range(1, m)                        # 5
            if (m - n) % 2 and gcd(m, n) == 1           # 6
        )
        if c <= N                                       # 7
    ),
    key=sum                                             # 8
)
```

Код кажется непростым для восприятия, поэтому разберем его построчно. В строке `#3` начинается генераторное выражение, которое создает тройки (a, b, c). Строки `#4` и `#5` задают вложенные циклы: `m` перебирается в диапазоне от 1 до M,

где M — это целая часть sqrt(N) плюс 1. Переменная `n` перебирается в диапазоне [1, m), чтобы выполнялось условие $m > n$. Обратите внимание: sqrt(N) мы вычислили как `N**0.5` — это просто другой способ записи, который здесь приведен для наглядности.

В строке `#6` указаны условия отбора примитивных троек: `(m - n) % 2` должно быть нечетным (то есть `m` и `n` разной четности), `gcd(m, n) == 1` — `m` и `n` взаимно простые. Эти фильтры гарантируют, что сгенерированные тройки будут примитивными. Внешнее генераторное выражение охватывает строки от `#2` до `#7`: оно перебирает тройки `(a, b, c)`, полученные из вложенного генератора, и отбирает только те, где `c` ≤ `N`.

Наконец, в строке `#1` мы сортируем результат, чтобы вывести тройки в определенном порядке. В строке `#8`, уже после завершения внешнего генераторного выражения, задается ключ сортировки — сумма `a + b + c`. Это просто наш субъективный выбор, а не математическое требование.

Такой код, безусловно, трудно читать, объяснять, отлаживать и сопровождать. В профессиональной среде подобная плотная запись нежелательна — она легко становится источником ошибок.

Теперь попробуем переписать код так, чтобы он стал более удобным для чтения и сопровождения:

```
# pythagorean.triple.generation.for.py
from functions import gcd
def gen_triples(N):
    for m in range(1, int(N**.5) + 1):          # 1
        for n in range(1, m):                   # 2
            if (m - n) % 2 and gcd(m, n) == 1:  # 3
                c = m**2 + n**2                 # 4
                if c <= N:                      # 5
                    a = m**2 - n**2             # 6
                    b = 2 * m * n               # 7
                    yield (a, b, c)             # 8
triples = sorted(gen_triples(50), key=sum)      # 9
```

Этот код читается намного легче. Пройдемся по нему построчно. Вы увидите, что и понять его значительно проще.

Циклы в строках `#1` и `#2` перебирают те же диапазоны значений `m` и `n`, что и в предыдущем примере. В строке `#3` фильтруются примитивные тройки. В строке `#4` мы немного отклоняемся от прежнего подхода: сначала вычисляем `c`, а в строке `#5` фильтруем случаи, где `c` ≤ `N`. Обратите внимание: мы не вычисляем `a` и `b` сразу, а откладываем это до тех пор, пока все условия корректности тройки не будут выполнены. Благодаря этому мы экономим время и ресурсы процессора: не тратим усилий на расчет `a` и `b`, если все равно будем отбрасывать результат из-за `c`. В самом конце применяется сортировка — по тому же ключу, что и раньше в генераторном выражении.

Надеемся, вы согласитесь, что этот вариант понятнее. Если потребуется изменить или доработать код, это будет сделать гораздо проще и с меньшим риском ошибок, чем в случае с плотной записью генераторного выражения.

Если вывести результаты обоих вариантов, то можно получить одинаковый список троек:

```
[(3, 4, 5), (5, 12, 13), (15, 8, 17), (7, 24, 25), (21, 20, 29), (35, 12, 37),
(9, 40, 41)]
```

Между производительностью и читабельностью часто приходится искать компромисс. И не всегда легко нащупать правильный баланс. Наш совет: используйте включения и генераторные выражения, когда они делают код понятнее и короче. Но если запись становится трудной для чтения, объяснения или изменения, не бойтесь рефакторинга. Перепишите код так, чтобы его было удобно поддерживать.

Локализация имен

Теперь, когда мы обсудили все виды включений и генераторных выражений, можно поговорить о том, как внутри них работают переменные цикла. В Python 3 эти переменные локализованы во всех четырех формах: списковых включениях, словарных включениях, включениях множеств и генераторных выражениях. Это поведение отличается от обычного цикла `for`. Чтобы вы могли увидеть это поведение, приведем несколько простых примеров:

```
# scopes.py
A = 100
ex1 = [A for A in range(5)]
print(A)  # выводит: 100

ex2 = list(A for A in range(5))
print(A)  # выводит: 100

ex3 = {A: 2 * A for A in range(5)}
print(A)  # выводит: 100

ex4 = {A for A in range(5)}
print(A)  # выводит: 100

s = 0
for A in range(5):
    s += A
print(A)  # выводит: 4
```

Здесь в глобальной области задано имя `A = 100`. Затем выполняются три вида включения: списковое, словарное и включение множества, а также генераторное выражение — все они используют имя `A` как переменную цикла. Тем не менее

ни одно из них не изменяет глобальное значение A. В свою очередь, цикл for, указанный после них, меняет глобальное имя — в результате последняя строка кода выводит 4.

Теперь посмотрим, что произойдет, если имя A изначально не было определено:

```
# scopes.noglobal.py
ex1 = [A for A in range(5)]
print(A)  # вызывает ошибку: NameError: name 'A' is not defined
```

Код будет работать точно так же и с любым другим видом включения или с генераторным выражением. После выполнения первой строки имя A не появляется в глобальной области имен. А вот цикл for снова ведет себя иначе:

```
# scopes.for.py
s = 0
for A in range(5):
    s += A
print(A)  # выводит: 4
print(globals())
```

После выполнения цикла имя A уже присутствует в глобальной области. Это можно проверить, изучив словарь, возвращаемый встроенной функцией globals():

```
$ python scopes.for.py
4
{'__name__': '__main__', '__doc__': None, ..., 's': 10, 'A': 4}
```

Среди прочих встроенных имен (которые здесь не выводятся) будет пара 'A': 4.

Поведение встроенных функций, напоминающее работу генераторов

Во встроенных функциях и типах Python часто встречается поведение, схожее с генераторным. Это одно из ключевых отличий Python 2 от Python 3. В Python 2 функции вроде map(), zip() и filter() возвращали списки. В Python 3 они возвращают итерируемые объекты, которые можно перебрать в цикле. Логика простая: если нужен список — всегда можно обернуть вызов в list(); если нужен только перебор, без лишней нагрузки на память, — достаточно самого итератора. То же относится к методу range(): в Python 2 он возвращал список, а его аналог назывался xrange(). В Python 3 range() теперь работает как xrange() — возвращает итерируемый объект, а не готовый список.

Идея возвращать из функций и методов итерируемые объекты получила широкое распространение. Такой прием можно встретить в функции open() (чтение строк из файла — подробнее см. в главе 8), в enumerate(), в методах словарей: .keys(), .values(), .items() и во многих других местах.

В этом есть смысл: Python стремится экономить память, особенно там, где функции и методы вызываются часто и повсеместно.

В начале главы мы говорили, что оптимизировать стоит те участки кода, которые работают с большими коллекциями, а не стремиться сэкономить миллисекунду на функции, вызываемой дважды в день. Именно такой стратегией руководствуется и сам Python.

Еще один пример напоследок

Прежде чем завершить главу, разберем одну простую задачу, которую Фабрицио раньше задавал на собеседованиях кандидатам на позицию Python-разработчика.

Условие такое: написать функцию, которая возвращает члены последовательности 0 1 1 2 3 5 8 13 21... вплоть до заданного предела *N*.

Возможно, эта последовательность показалась вам знакомой — это ряд Фибоначчи. Он определяется так: $F(0) = 0$, $F(1) = 1$, а для любого $n > 1$ $F(n) = F(n - 1) + F(n - 2)$. Эта задача хороша для проверки знания рекурсии, мемоизации и других технических деталей. Но в данном случае она была способом проверить, знаком ли кандидат с генераторами.

Начнем с простого варианта, а затем улучшим его:

```
# fibonacci.first.py
def fibonacci(N):
    """Возвращает все числа Фибоначчи до N включительно."""
    result = [0]
    next_n = 1
    while next_n <= N:
        result.append(next_n)
        next_n = sum(result[-2:])
    return result

print(fibonacci(0))   # [0]
print(fibonacci(1))   # [0, 1, 1]
print(fibonacci(50))  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Итак, сначала мы создаем список `result` с начальным значением `[0]`. Затем начинаем итерацию со следующего члена последовательности — `next_n = 1`. Пока `next_n` не превышает *N*, мы продолжаем добавлять его в список и вычислять следующее значение. Новое значение находится так: берется срез из двух последних элементов списка и передается в функцию `sum()`.

Если код кажется непонятным, то можно добавить вызовы print() — они помогут отследить, как меняются значения во время выполнения.

Как только условие в цикле становится ложным, выполнение выходит из цикла, и функция возвращает список `result`. Комментарии рядом с вызовами `print()` показывают промежуточные значения.

На данном этапе Фабрицио задавал кандидатам такой вопрос: *«А если мне нужно просто перебрать эти числа, не сохраняя в список?»* Толковый кандидат переписывал код вот так:

```
# fibonacci.second.py
def fibonacci(N):
    """Возвращает все числа Фибоначчи до N включительно."""
    yield 0
    if N == 0:
        return
    a = 0
    b = 1
    while b <= N:
        yield b
        a, b = b, a + b

print(list(fibonacci(0)))   # [0]
print(list(fibonacci(1)))   # [0, 1, 1]
print(list(fibonacci(50)))  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Это как раз одна из версий, которую Фабрицио получал от кандидатов. Теперь функция `fibonacci()` стала *генераторной*. Сначала она выдает `0`, и если `N` равен нулю — завершает выполнение (что приводит к выбрасыванию исключения `StopIteration`). Если нет, то начинается цикл: на каждой итерации мы выдаем значение `b`, а затем обновляем переменные `a` и `b`. Такой подход опирается на тот факт, что для генерации следующего числа достаточно помнить только два предыдущих значения: `a` и `b`.

Такой код эффективнее: он не накапливает данные в памяти, а выдает их по мере запроса. Если все-таки нужен список, то достаточно обернуть вызов в `list()`, как обычно. Но этот вариант можно сделать еще более элегантным:

```
# fibonacci.elegant.py
def fibonacci(N):
    """Возвращает все числа Фибоначчи до N включительно."""
    a, b = 0, 1
    while a <= N:
        yield a
        a, b = b, a + b
```

Теперь тело функции занимает всего четыре строки (пять, если учитывать строку документации). Особенно хорошо здесь работает множественное присваивание (`a, b = 0, 1` и затем `a, b = b, a + b`) — оно делает код и короче, и понятнее.

Резюме

В этой главе мы углубились в тему итераций и генерации. Мы подробно рассмотрели функции `map()`, `zip()` и `filter()`, вы научились использовать их как альтернативу классическому циклу `for`.

Затем мы обсудили включения — списковые, словарные и включения множеств. Рассмотрели их синтаксис и увидели, что они могут заменить как обычные циклы, так и вызовы `map()`, `zip()` и `filter()`.

После этого мы перешли к генераторам — как к генераторным функциям, так и к генераторным выражениям. Вы увидели, как генераторы помогают экономить время и память, а также позволяют выполнять те операции, которые было бы невозможно реализовать с помощью списков.

Мы обсудили производительность и пришли к выводу, что цикл `for` является самым медленным из всех, но при этом улучшает читабельность кода и обеспечивает наибольшую гибкость при изменении кода. С другой стороны, такие конструкции, как `map()`, `filter()` и включения, могут работать значительно быстрее.

Важно помнить: чем сложнее включение или генератор, тем труднее его читать и сопровождать. Поэтому в некоторых ситуациях ради простоты и понятности стоит предпочесть классический цикл `for`.

Вы также увидели, что переменные цикла во включениях и генераторных выражениях локализуются, в то время как в `for` они остаются доступными в общей области видимости, — это еще одно различие, о котором стоит помнить.

В следующей главе мы перейдем к объектам и классам. По структуре она будет похожа на эту: тем будет немного, но мы постараемся дать по ним много полезной информации.

6 ООП, декораторы и итераторы

Класс — это не пустяк. (La classe non è acqua.)

Итальянская пословица

Объектно-ориентированное программирование (ООП) — тема настолько обширная, что ей посвящены целые книги. В этой главе нам предстоит найти баланс между широтой и глубиной изложения материала. Здесь слишком много тем, и многие из них заслуживают отдельной главы. Поэтому мы постараемся дать панорамный обзор ключевых основ, а также рассмотреть несколько приемов, которые пригодятся в следующих главах. Все остальное вы сможете уточнить при необходимости — официальная документация Python поможет восполнить пробелы.

Декораторы

В главе 5, посвященной включениям и генераторам, мы замеряли время выполнения различных выражений. Нам приходилось вручную сохранять начальное время, а затем вычислять разницу с текущим временем после выполнения — чтобы определить, сколько прошло. И каждый раз мы выводили результат на экран. Такой подход совершенно непрактичен. Если мы постоянно повторяем один и тот же код, это тревожный сигнал: нельзя ли вынести эту часть в отдельную функцию? Обычно можно. Рассмотрим пример:

```
# decorators/time.measure.start.py
from time import sleep, time

def f():
    sleep(0.3)

def g():
    sleep(0.5)

t = time()
f()
print("f took:", time() - t) # f заняла: 0.3028988838195801
```

```
t = time()
g()
print("g took:", time() - t) # g заняла: 0.507941722869873
```

В этом коде мы определили две функции, `f()` и `g()`, которые просто «засыпают» на 0,3 и 0,5 секунды соответственно. Для имитации работы мы воспользовались функцией `sleep()` — и, как видите, измерение времени дает вполне точные результаты. Но как избавиться от повторяющегося кода и всех этих вычислений? Один из очевидных шагов — вынести всю логику измерения времени в отдельную функцию:

```
# decorators/time.measure.dry.py
from time import sleep, time

def f():
    sleep(0.3)

def g():
    sleep(0.5)

def measure(func):
    t = time()
    func()
    print(func.__name__, "took:", time() - t)

measure(f)  # f заняла: 0.3043971061706543
measure(g)  # g заняла: 0.5050859451293945
```

Так намного лучше: весь механизм замера времени инкапсулирован и мы не дублируем одни и те же строки. Мы даже выводим имя измеряемой функции динамически — все выглядит аккуратно. А если нужно передавать аргументы в функцию, которую мы замеряем? Код придется немного усложнить:

```
# decorators/time.measure.arguments.py
from time import sleep, time

def f(sleep_time=0.1):
    sleep(sleep_time)

def measure(func, *args, **kwargs):
    t = time()
    func(*args, **kwargs)
    print(func.__name__, "took:", time() - t)

measure(f, sleep_time=0.3)  # f заняла: 0.30092811584472656
measure(f, 0.2)             # f заняла: 0.20505475997924805
```

Теперь функция `f()` принимает аргумент `sleep_time` (по умолчанию — `0.1`), так что отдельная функция `g()` больше не нужна. Вдобавок мы изменили функцию

measure(), чтобы она принимала не только функцию, но и любое количество позиционных и именованных аргументов. Благодаря этому независимо от того, с чем мы вызовем measure(), все аргументы будут переданы внутрь вызова func().

Такой вариант уже более гибкий и универсальный. Но и его можно улучшить. Допустим, мы хотим, чтобы функция f() уже сама по себе измеряла время выполнения и чтобы мы просто вызывали ее, не заботясь о дополнительных обертках. Реализовать это можно так:

```
# decorators/time.measure.deco1.py
from time import sleep, time

def f(sleep_time=0.1):
    sleep(sleep_time)

def measure(func):
    def wrapper(*args, **kwargs):
        t = time()
        func(*args, **kwargs)
        print(func.__name__, "took:", time() - t)

    return wrapper

f = measure(f)  # точка декорации

f(0.2)  # f заняла: 0.20128178596496582
f(sleep_time=0.3)  # f заняла: 0.30509519577026367
print(f.__name__)  # wrapper <- увы!
```

Этот код уже не такой простой. Поговорим о том, как он устроен. Главное происходит в *момент декорирования*. Мы переопределяем функцию f(), присваивая ей результат вызова measure(f). А внутри measure() определяем функцию wrapper() и возвращаем ее. В итоге после декорации при вызове f() на самом деле вызывается wrapper(). Это хорошо видно в последней строке кода. Внутри wrapper() вызывается func() (а func в данном случае ссылается на оригинальную функцию f()), поэтому цепочка замыкается: мы вызываем обертку, которая вызывает саму функцию f() — с сохраненной логикой.

Функция wrapper(), как несложно догадаться, — это обертка. Она принимает любые позиционные и именованные аргументы, вызывает f() с ними и вокруг этого вызова производит измерение времени.

Такой прием называется *декорированием*, а функция measure() — это *декоратор*. Подход оказался настолько удобным и популярным, что начиная с версии Python 2.4 для него появился специальный синтаксис @, описанный в PEP 318 (https://peps.python.org/pep-0318/). Позже, в Python 3.0, синтаксис расширили для декораторов классов — см. PEP 3129 (https://peps.python.org/pep-3129/). А в Python 3.9

(см. PEP 614, https://peps.python.org/pep-0614/) сняли некоторые ограничения, чтобы грамматика стала более гибкой.

Теперь рассмотрим три варианта: один декоратор, два декоратора и декоратор с аргументами. Начнем с самого простого — одиночного декоратора:

```
# decorators/syntax.py
def func(arg1, arg2, ...):
    pass
func = decorator(func)

# эквивалентно следующему:

@decorator
def func(arg1, arg2, ...):
    pass
```

Вместо того чтобы вручную присваивать результат декоратора функции, мы можем использовать специальный синтаксис `@имя_декоратора` перед определением функции. Такой способ выглядит чище и читается проще.

Допускается применять несколько декораторов к одной и той же функции. Вот как это делается:

```
# decorators/syntax.py
def func(arg1, arg2, ...):
    pass
func = deco1(deco2(func))

# эквивалентно следующему:

@deco1
@deco2
def func(arg1, arg2, ...):
    pass
```

При использовании нескольких декораторов важен порядок: в данном примере `func()` сначала оборачивается в `deco2()`, а затем результат передается в `deco1()`. Полезное правило: *чем ближе декоратор расположен к определению функции, тем раньше он применяется*.

Прежде чем перейти к следующему примеру, исправим нюанс с именем функции. Взгляните на выделенный фрагмент:

```
# decorators/time.measure.deco1.py
def measure(func):
    def wrapper(*args, **kwargs):
        …
    return wrapper

f = measure(f) # точка декорирования
print(f.__name__) # wrapper <- увы!
```

Нам не хочется терять имя и строку документации исходной функции при ее декорировании. Но поскольку `f` — декорируемая функция — переопределяется и указывает на `wrapper`, то ее оригинальные атрибуты заменяются атрибутами функции `wrapper()`. Это легко исправить — с помощью функции `wraps()` в модуле `functools`. Сейчас мы устраним проблему и перепишем код, используя оператор `@`:

```
# decorators/time.measure.deco2.py
from time import sleep, time
from functools import wraps

def measure(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        t = time()
        func(*args, **kwargs)
        print(func.__name__, "took:", time() - t)
    return wrapper

@measure
def f(sleep_time=0.1):
    """I'm a cat. I love to sleep!"""
    sleep(sleep_time)

f(sleep_time=0.3) # f заняла: 0.30042004585266113
print(f.__name__) # f
print(f.__doc__ ) # I'm a cat. I love to sleep!
```

Выглядит прекрасно. Итак, достаточно сообщить Python, что `wrapper` оборачивает `func()` — для этого мы вызываем функцию `wraps()` в выделенном фрагменте кода. После этого имя функции и строка документации сохраняются.

Полный список атрибутов, которые копируются с помощью wraps(), можно найти в официальной документации к функции functools.update_wrapper() по адресу https://docs.python.org/3/library/functools.html?#functools.update_wrapper.

Теперь возьмем еще один пример. Мы хотим создать декоратор, который выводит сообщение об ошибке, если результат функции превышает заданный порог. И заодно посмотрим, как применить сразу два декоратора:

```
# decorators/two.decorators.py
from time import time
from functools import wraps

def measure(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        t = time()
```

```
        result = func(*args, **kwargs)
        print(func.__name__, "took:", time() - t)
        return result
    return wrapper

def max_result(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        if result > 100:
            print(
                f"Result is too big ({result}). "
                "Max allowed is 100."
            )
        return result
    return wrapper

@measure
@max_result
def cube(n):
    return n**3

print(cube(2))
print(cube(5))
```

Мы немного доработали декоратор `measure()`, чтобы теперь функция `wrapper()` возвращала результат вызова `func()`. Декоратор `max_result()` тоже возвращает результат, но перед этим проверяет, чтобы значение не превышало `100` — установленный максимум.

Функция `cube()` была декорирована обоими декораторами. Сначала применяется `max_result()`, затем `measure()`. Выполнение кода дает следующий результат:

```
$ python two.decorators.py
cube took: 9.5367431640625e-07
8

Result is too big (125). Max allowed is 100.
cube took: 3.0994415283203125e-06
125
```

Для удобства восприятия мы разделили результаты двух вызовов пустой строкой. В первом случае результат — `8`, он проходит проверку, затем измеряется и выводится время выполнения. В конце печатается сам результат (`8`).

Во втором случае результат — `125`. Он превышает порог, поэтому выводится сообщение об ошибке, после чего возвращается результат. Далее вызывается `measure()`, выводятся данные о том, сколько времени занял вызов, и снова выводится результат (`125`).

Если бы мы применили те же два декоратора, но в другом порядке, то порядок сообщений в выводе тоже изменился бы.

Фабрика декораторов

Некоторые декораторы могут принимать аргументы. Этот прием обычно используется для создания других декораторов, а получаемый объект называется *фабрикой декораторов*. Сначала разберемся с синтаксисом, а затем рассмотрим пример.

```
# decorators/syntax.py
def func(arg1, arg2, ...):
    pass

func = decoarg(arg_a, arg_b)(func)

# эквивалентно следующему:

@decoarg(arg_a, arg_b)
def func(arg1, arg2, ...):
    pass
```

Здесь все устроено чуть иначе. Сначала вызывается `decoarg()` с аргументами, а затем результат этого вызова — то есть сам декоратор — вызывается с `func()`.

Теперь немного доработаем пример. Вернемся к декоратору `max_result()`. Мы хотим, чтобы с его помощью можно было задавать разные пороги и применять один и тот же декоратор к разным функциям, не создавая отдельный вариант кода для каждого значения. Для этого изменим `max_result()` так, чтобы порог можно было передавать как аргумент при декорировании:

```
# decorators/decorators.factory.py
from functools import wraps

def max_result(threshold):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            result = func(*args, **kwargs)
            if result > threshold:
                print(
                    f"Result is too big ({result})."
                    f"Max allowed is {threshold}."
                )
            return result
        return wrapper
    return decorator

@max_result(75)
def cube(n):
    return n**3
```

Здесь показано, как написать фабрику декораторов. Как вы, вероятно, помните, дополнение функции декоратором с аргументами эквивалентно конструкции: `func = decorator(argA, argB)(func)`. Поэтому, когда мы пишем `@max_result(75)` перед определением `cube()`, это то же самое, что `cube = max_result(75)(cube)`.

Попробуем разобраться, что происходит, шаг за шагом. Когда мы вызываем `max_result(75)`, выполняется тело этой функции. Внутри нее определяется функция `decorator()`, которая принимает одну функцию как аргумент. Внутри `decorator()` реализуется обычный шаблон декорирования: мы определяем `wrapper()`, внутри которого проверяем результат вызова исходной функции. Главное достоинство такого подхода в том, что из глубины вложенности мы имеем доступ и к `func()`, и к значению `threshold`, которое передали снаружи. Благодаря этому можно гибко задавать порог при каждом декорировании.

Функция `wrapper()` возвращает `result`, функция `decorator()` возвращает `wrapper()`, а функция `max_result()` возвращает `decorator()`. Это означает, что инструкция `cube = max_result(75)(cube)` фактически превращается в `cube = decorator(cube)`. Однако в данном случае это не просто `decorator()`, а тот, для которого переменной `threshold` присвоено значение `75`. Это реализуется с помощью механизма, называемого *замыканием*.

Замыканиями называют функции, создаваемые динамически и возвращаемые другими функциями. Их основная особенность — полный доступ к переменным и именам, определенным в локальной области видимости на момент создания, даже если окружающая функция уже завершила выполнение.

При запуске последнего примера будет получен следующий результат:

```
$ python decorators.factory.py
Result is too big (125). Max allowed is 75.
125
```

Благодаря такому подходу можно применять декоратор `max_result()` с разными значениями порога, например:

```
# decorators/decorators.factory.py
@max_result(75)
def cube(n):
    return n**3

@max_result(100)
def square(n):
    return n**2

@max_result(1000)
def multiply(a, b):
    return a * b
```

Обратите внимание: при каждом применении декоратора используется свое значение порога.

Декораторы в Python очень популярны. Они используются довольно часто и делают код проще и элегантнее.

Объектно-ориентированное программирование

Теперь, когда мы разобрали основы шаблона декорирования, пора перейти к объектно-ориентированному программированию (ООП). Мы воспользуемся определением из статьи Е. Киндлера и И. Кривых (2011) *Object-oriented simulation of systems with sophisticated control* (журнал *International Journal of General Systems*) и адаптируем его под Python:

> «*Объектно-ориентированное программирование (ООП)* — это парадигма программирования, основанная на концепции объектов. Объекты — это структуры данных, содержащие как данные (в виде атрибутов), так и код (в виде функций, называемых методами). Отличительная особенность объектов состоит в том, что методы могут обращаться к атрибутам данных объекта и, как правило, изменять их. У объектов есть представление о самих себе — в Python это реализуется через ссылку self. В объектно-ориентированном подходе программы создаются как совокупность объектов, взаимодействующих друг с другом».

Python полностью поддерживает эту парадигму. Более того, как вы уже знаете, в этом языке все представлено объектами. Это значит, что объектно-ориентированное программирование не просто поддерживается — оно встроено в саму основу Python.

В ООП основными понятиями выступают *объекты* и *классы*. Классы используются для создания объектов, и такие объекты называют *экземплярами* класса.

Если вам трудно уловить разницу между классом и объектом, то представьте следующее. Когда вы слышите слово «ручка», то сразу понимаете, о каком типе предмета идет речь, — это и есть класс. А вот выражение «эта ручка» уже указывает не на класс, а на конкретный экземпляр — реальный объект.

Объект, созданный на основе класса, наследует его атрибуты и методы. Такие объекты представляют собой конкретные элементы предметной области программы.

Наипростейший класс в Python

Начнем с самого простого класса, который только можно написать на Python:

```
# oop/simplest.class.py
class Simplest:
    pass

print(type(Simplest))          # Какой тип у объекта Simplest?

simp = Simplest()              # Создаем экземпляр класса Simplest: simp
print(type(simp))              # Какой тип у объекта simp?
# Является ли simp экземпляром Simplest?
print(type(simp) is Simplest)  # Есть способ получше проверить это
```

Если мы запустим этот код, то получим такой результат:

```
$ python simplest.class.py
<class 'type'>
<class '__main__.Simplest'>
True
```

Теперь разберем код построчно. Класс `Simplest`, который мы определили, содержит в теле лишь инструкцию `pass`. Это означает, что в нем нет ни пользовательских атрибутов, ни методов. Мы выведем его тип (в данном случае `__main__` — это имя области видимости, в которой выполняется код верхнего уровня) и отметим, что в пояснительном комментарии выше мы написали *«объект»* вместо *«класс»*. Однако, как видно из результата выполнения инструкции `print`, *классы сами по себе — это тоже объекты*. Точнее говоря, они являются экземплярами класса `type`. Объяснение этого момента потребовало бы обсуждения *метаклассов* и *метапрограммирования* — более сложных тем, понимание которых требует прочного владения основами. Эти темы выходят за рамки главы. Мы упомянули о них, чтобы вы знали, какую информацию искать, если захотите изучить их более глубоко.

Вернемся к примеру: мы создали `simp` — экземпляр класса `Simplest`. Обратите внимание: синтаксис создания экземпляра такой же, как при вызове функции. Затем мы выводим тип `simp` и убеждаемся, что он действительно является экземпляром `Simplest`. Далее в главе мы покажем более надежный способ проверить это.

Пока все достаточно просто. Однако что происходит, когда вы пишете `class ClassName(): pass`? В таком случае Python создает объект класса и присваивает ему имя. Это очень похоже на то, что происходит при определении функции с помощью `def`.

Пространства имен классов и объектов

После создания объекта класса (что обычно происходит при первом импорте модуля) он представляет собой пространство имен. Этот класс можно вызывать для создания его экземпляров. Каждый экземпляр наследует атрибуты и методы класса и получает собственное пространство имен. Мы уже знаем, что для обращения к элементам пространства имен используется оператор точки (`.`).

Рассмотрим следующий пример:

```
# oop/class.namespaces.py
class Person:
    species = "Human"

print(Person.species)     # Human
Person.alive = True       # Динамически добавляем атрибут классу!
print(Person.alive)       # True
```

```
man = Person()
print(man.species)        # Human (унаследовано от класса)
print(man.alive)          # True (унаследовано от класса)

Person.alive = False
print(man.alive)          # False (унаследовано, значение изменилось в классе)

man.name = "Darth"
man.surname = "Vader"
print(man.name, man.surname)  # Darth Vader
```

В этом примере мы определили *атрибут класса* с именем `species`. Любое имя, определенное в теле класса, становится атрибутом данного класса. Кроме того, в коде определен `Person.alive` — еще один атрибут класса. Такие атрибуты доступны напрямую через имя класса, без ограничений.

Экземпляр `man`, созданный на основе класса `Person`, наследует оба атрибута и мгновенно отражает изменения их значений.

При этом у `man` есть и собственные атрибуты: `name` и `surname`. Они принадлежат пространству имен экземпляра и называются *атрибутами экземпляра*.

Атрибуты класса общие для всех экземпляров, тогда как атрибуты экземпляра уникальны для каждого объекта. Поэтому используйте атрибуты класса, если нужно задать состояние или поведение, общее для всех экземпляров, и задействуйте атрибуты экземпляра для хранения данных, уникальных для каждого объекта.

Затенение атрибутов

Когда Python ищет атрибут у объекта, но не находит его, поиск продолжается в классе этого объекта. Если атрибут не найден и там, то поиск движется дальше по цепочке наследования — до тех пор, пока атрибут не найдется или не будет достигнут конец цепочки. Мы поговорим о наследовании позже, а пока разберем интересное поведение, связанное с затенением атрибутов.

Рассмотрим пример:

```
# oop/class.attribute.shadowing.py
class Point:
    x = 10
    y = 7

p = Point()
print(p.x)      # 10 (из атрибута класса)
print(p.y)      # 7 (из атрибута класса)

p.x = 12        # у p появляется собственный атрибут x
print(p.x)      # 12 (теперь находится в экземпляре)
print(Point.x)  # 10 (атрибут класса не изменился)
```

```
del p.x         # удаляем атрибут экземпляра
print(p.x)      # 10 (поиск снова идет к атрибуту класса)

p.z = 3         # сделаем его трехмерной точкой
print(p.z)      # 3

print(Point.z)
# AttributeError: type object 'Point' has no attribute 'z'
```

В этом фрагменте показана любопытная ситуация. Мы определили класс `Point` с двумя атрибутами класса — `x` и `y`. После создания экземпляра `p` мы получаем доступ к `p.x` и `p.y`. При этом Python сначала ищет эти атрибуты в пространстве имен `p`. Не найдя их там, он переходит к классу `Point` — и успешно находит нужные значения.

Затем мы присваиваем значение `12` конкретно `p.x`, тем самым создавая атрибут экземпляра `x`, который затеняет одноименный атрибут класса. На первый взгляд это может показаться странным, но логика абсолютно та же, что и при работе с областями видимости: если внутри функции присвоить `x = 12`, а снаружи уже есть `x = 10` в глобальной области (см. раздел «Области видимости и разрешение имен» главы 4), то локальное значение `12` не влияет на глобальное. Аналогично атрибут экземпляра не влияет на одноименный атрибут класса.

После присваивания `p.x = 12` при выводе этого значения Python больше не обращается к атрибутам класса — ведь `x` уже найден в пространстве имен экземпляра. Поэтому выводится `12`. Кроме того, мы выводим `Point.x`, чтобы показать, что в пространстве имен класса `x` по-прежнему равен 10.

Затем мы удаляем атрибут `x` из пространства имен `p`, и на следующей строке, при повторном обращении к `p.x`, Python уже не находит его в экземпляре, поэтому снова обращается к атрибуту класса.

Последние три строки примера показывают: если присвоить значение атрибуту экземпляра, то оно не попадет в класс. Экземпляры получают значения из класса, но обратного направления нет.

Как вам идея хранить координаты `x` и `y` как атрибуты класса? Кажется ли она удачной? А что если создать второй экземпляр класса `Point`? Возможно, тогда станет понятнее, зачем нужны атрибуты экземпляра.

Аргумент self

Внутри метода класса можно обращаться к экземпляру с помощью специального аргумента, который по соглашению называется `self`. Он всегда указывается первым в списке параметров метода экземпляра. Рассмотрим, как он работает, и заодно увидим, что не только атрибуты, но и методы могут быть общими для всех экземпляров:

```
# oop/class.self.py
class Square:
    side = 8

    def area(self):     # self – это ссылка на экземпляр
        return self.side**2

sq = Square()
print(sq.area())        # 64 (side находится в классе)
print(Square.area(sq))  # 64 (эквивалентно sq.area())

sq.side = 10
print(sq.area())        # 100 (side находится в экземпляре)
```

Обратите внимание, как используется метод `area()` для объекта `sq`. Вызовы `Square.area(sq)` и `sq.area()` эквивалентны — они наглядно демонстрируют, как работает механизм вызова метода. В первом случае мы явно передаем экземпляр `sq` как аргумент, и внутри метода он получает имя `self`. Во втором используем более удобный синтаксис `sq.area()`, и Python сам неявно передает экземпляр `sq` как первый аргумент.

Рассмотрим более наглядный пример:

```
# oop/class.price.py
class Price:
    def final_price(self, vat, discount=0):
        """ Возвращает цену после применения НДС и фиксированной скидки."""
        return (self.net_price * (100 + vat) / 100) - discount

p1 = Price()
p1.net_price = 100
print(Price.final_price(p1, 20, 10))  # 110 (100 * 1.2 - 10)
print(p1.final_price(20, 10))         # эквивалентно
```

Этот код показывает, что ничто не мешает использовать дополнительные аргументы при объявлении методов. Синтаксис тот же, что и при определении обычных функций, но важно помнить: первым аргументом всегда будет сам экземпляр, с которым метод будет связан. Его не обязательно называть `self`, но таково принятое соглашение. И это как раз тот редкий случай, когда следовать соглашению особенно важно.

Инициализация экземпляра

Вы заметили, что перед вызовом `p1.final_price()` в предыдущем примере нам пришлось вручную задать `net_price` для `p1`? Есть более удобный способ. В других языках такую конструкцию назвали бы *конструктором*, но в Python это не совсем так. В нем используется *инициализатор*, поскольку он работает уже с созданным объектом. Этот метод называется `__init__()` — это *магический метод*, который

автоматически вызывается сразу после создания объекта. У объектов Python есть и метод `__new__()`, который представляет собой настоящий конструктор. Однако переопределять его на практике почти не требуется — он используется в основном при написании метаклассов.

Рассмотрим пример, в котором показано, как инициализируются объекты в Python:

```
# oop/class.init.py
class Rectangle:
    def __init__(self, side_a, side_b):
        self.side_a = side_a
        self.side_b = side_b

    def area(self):
        return self.side_a * self.side_b

r1 = Rectangle(10, 4)
print(r1.side_a, r1.side_b)  # 10 4
print(r1.area())             # 40

r2 = Rectangle(7, 3)
print(r2.area())             # 21
```

Теперь код начинает принимать должный вид. Когда создается объект, Python автоматически вызывает метод `__init__()`. В нашем случае мы написали инициализатор так, чтобы при создании объекта `Rectangle` (вызовом имени класса как функции) можно было сразу передать параметры, как при обычном вызове функции. Переданные значения соответствуют сигнатуре метода `__init__()`. В первом случае значения `10` и `4` становятся `side_a` и `side_b` для `r1`, во втором — `7` и `3` становятся `side_a` и `side_b` для `r2`. Вызовы `area()` для `r1` и `r2` подтверждают, что у объектов разные значения атрибутов экземпляра. Такой способ задания данных при создании объекта является гораздо более удобным.

ООП подразумевает повторное использование кода

На данном этапе идея уже должна быть понятна: *объектно-ориентированное программирование строится вокруг повторного использования кода*. Мы определяем класс, создаем экземпляры, и они могут вызывать методы, описанные в классе. Поведение методов зависит от того, как экземпляры были настроены в инициализаторе.

Наследование и композиция

Однако это лишь часть картины. В объектно-ориентированном программировании используются два ключевых приема проектирования: наследование и композиция.

Наследование подразумевает, что между двумя объектами существует отношение *«является» (Is-A)*. *Композиция* описывает отношение *«содержит» (Has-A)*. Объясним на примере: сначала мы объявим классы, представляющие типы двигателей:

```
# oop/class_inheritance.py
class Engine:
    def start(self):
        pass

    def stop(self):
        pass

class ElectricEngine(Engine):  # является Engine
    pass

class V8Engine(Engine):        # является Engine
    pass
```

Затем определим классы автомобилей, которые используют эти двигатели:

```
class Car:
    engine_cls = Engine

    def __init__(self):
        self.engine = self.engine_cls()  # содержит Engine

    def start(self):
        print(
            f"Starting {self.engine.__class__.__name__} for "
            f"{self.__class__.__name__}... Wroom, wroom!"
        )
        self.engine.start()

    def stop(self):
        self.engine.stop()

class RaceCar(Car):    # является Car
    engine_cls = V8Engine

class CityCar(Car):    # является Car
    engine_cls = ElectricEngine

class F1Car(RaceCar):  # является и RaceCar, и Car
    pass               # engine_cls такой же, как у родителя

car = Car()
racecar = RaceCar()
citycar = CityCar()
f1car = F1Car()
cars = [car, racecar, citycar, f1car]

for car in cars:
    car.start()
```

Этот код выдаст следующий результат:

```
$ python class_inheritance.py
Starting Engine for Car... Wroom, wroom!
Starting V8Engine for RaceCar... Wroom, wroom!
Starting ElectricEngine for CityCar... Wroom, wroom!
Starting V8Engine for F1Car... Wroom, wroom!
```

В примере показаны оба типа отношений — *Is-A* и *Has-A*. Начнем с класса `Engine`. Это простой класс с двумя методами: `start()` и `stop()`. Затем мы определяем `ElectricEngine` и `V8Engine` — оба наследуются от `Engine`. Это видно по их определению: имя `Engine` указано в круглых скобках после имени класса.

Это означает, что классы `ElectricEngine` и `V8Engine` наследуют атрибуты и методы от класса `Engine`, который называют *базовым* или *родительским классом*.

Аналогичная ситуация с автомобилями. Класс `Car` является базовым для классов `RaceCar` и `CityCar`. При этом `RaceCar` — базовый класс для `F1Car`. Другими словами, `F1Car` наследует от `RaceCar`, а тот — от `Car`. Значит, `F1Car` — это `RaceCar` (отношение *Is-A*), а `RaceCar` — это `Car`. Благодаря транзитивному свойству можно утверждать, что `F1Car` — это еще и `Car`. Точно так же `CityCar` — это `Car`.

Когда мы пишем `class A(B): pass`, это означает, что `A` — *потомок* (или *производный класс*) от `B`, а `B` — его *родитель*. Понятия «родительский класс» и «базовый класс» взаимозаменяемы, как и выражения «дочерний класс» и «производный класс». Мы также говорим, что один класс *наследует* от другого или *расширяет* его.

Это и есть механизм наследования.

Вернемся к нашему коду. В каждом классе автомобиля определен атрибут класса `engine_cls`, в котором хранится ссылка на класс двигателя, соответствующего типу автомобиля. У класса `Car` — базовый `Engine`, у двух гоночных автомобилей — `V8Engine`, а у городского — `ElectricEngine`.

Когда создается объект автомобиля, в методе инициализации `__init__()` создается экземпляр того класса двигателя, который указан в `engine_cls`, и он сохраняется в виде атрибута экземпляра `engine`.

Имеет смысл задавать `engine_cls` как атрибут класса, поскольку весьма вероятно, что все экземпляры одного и того же класса автомобиля будут использовать один и тот же тип двигателя. А вот хранить экземпляр двигателя в виде атрибута класса было бы ошибкой — в таком случае все автомобили использовали бы один и тот же двигатель, что явно противоречит здравому смыслу.

Связь между автомобилем и его двигателем — это отношение типа *Has-A*: автомобиль *содержит* двигатель. Такой подход называется *композицией* и отражает

идею, что одни объекты могут состоять из других. Автомобиль *содержит* двигатель, коробку передач, колеса, кузов, двери, сиденья и т. д.

При работе в парадигме ООП важно описывать объекты именно таким образом — это помогает правильно структурировать код.

Обратите внимание: мы не использовали точки в имени файла class_inheritance.py, поскольку точки в именах модулей затрудняют импорт. Большинство модулей, приведенных в книге, предназначены для запуска в качестве отдельных скриптов, и там мы иногда добавляли точки ради улучшения читабельности. Но в общем случае точек в названиях модулей лучше избегать.

И прежде, чем завершить этот подраздел, проверим правильность всего сказанного с помощью еще одного примера:

```
# oop/class.issubclass.isinstance.py
from class_inheritance import Car, RaceCar, F1Car

car = Car()
racecar = RaceCar()
f1car = F1Car()
cars = [(car, "car"), (racecar, "racecar"), (f1car, "f1car")]
car_classes = [Car, RaceCar, F1Car]

for car, car_name in cars:
    for class_ in car_classes:
        belongs = isinstance(car, class_)
        msg = "is a" if belongs else "is not a"
        print(car_name, msg, class_.__name__)

""" Выводит:
… (сообщения о запуске двигателя опущены)
car is a Car
car is not a RaceCar
car is not a F1Car
racecar is a Car
racecar is a RaceCar
racecar is not a F1Car
f1car is a Car
f1car is a RaceCar
f1car is a F1Car
"""
```

Как вы можете видеть, `car` — это просто экземпляр класса `Car`, а `racecar` — экземпляр `RaceCar`, который, в свою очередь, наследует от `Car`. Аналогично `f1car` — экземпляр `F1Car` и по цепочке также экземпляр `RaceCar` и `Car`. Та же логика работает и в других случаях: например, банан — это экземпляр `Banana`. Но он также является

`Fruit` и в более широком смысле — `Food`, верно? Это все одна и та же концепция наследования. Проверить, является ли объект экземпляром определенного класса, помогает функция `isinstance()`. Использовать ее предпочтительнее, чем напрямую сравнивать типы через `type(object) is Class`.

Обратите внимание: мы не стали повторно выводить сообщения о создании объектов, которые появлялись при создании машин, — они уже были показаны в предыдущем примере.

Теперь проверим наследование. Используем ту же структуру классов, но изменим логику в циклах `for`:

```
# oop/class.issubclass.isinstance.py
for class1 in car_classes:
    for class2 in car_classes:
        is_subclass = issubclass(class1, class2)
        msg = "{0} a subclass of".format(
            "is" if is_subclass else "is not"
        )
        print(class1.__name__, msg, class2.__name__)

""" Выводит:
Car is a subclass of Car
Car is not a subclass of RaceCar
Car is not a subclass of F1Car
RaceCar is a subclass of Car
RaceCar is a subclass of RaceCar
RaceCar is not a subclass of F1Car
F1Car is a subclass of Car
F1Car is a subclass of RaceCar
F1Car is a subclass of F1Car
"""
```

Интересно, что *класс считается подклассом самого себя*. Вы можете увидеть это в выводе предыдущего примера — он подтверждает сказанное выше.

Обратите внимание на соглашения об именовании. Имена классов пишутся в верблюжьем регистре, то есть вот так: ThisWayIsCorrect. В то же время имена функций и методов оформляются в змеином_стиле — например: this_way_is_correct. Если вы хотите использовать имя, которое совпадает с зарезервированным словом Python или названием встроенной функции/класса, то добавляйте подчеркивание в конце. Именно поэтому в первом примере с циклом for мы писали for class_ in ... — поскольку class является ключевым словом языка. Освежить знания о рекомендациях по стилю можно с помощью документа PEP 8.

Визуализировать различия между отношениями *Is-A* и *Has-A* поможет следующая схема (рис. 6.1).

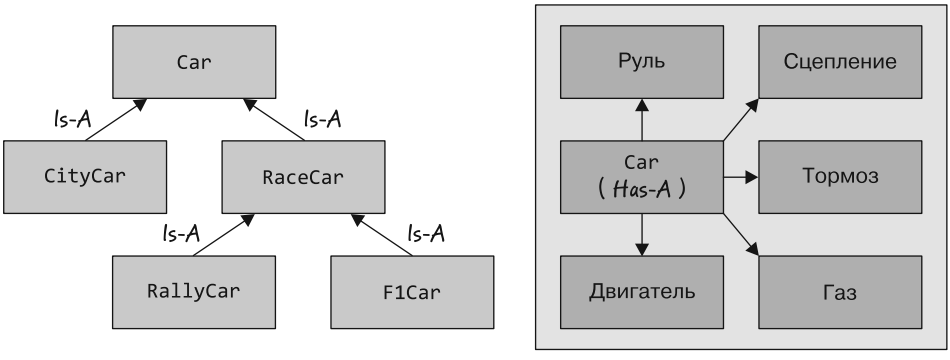


Рис. 6.1. Различия между отношениями Is-A и Has-A

Обращение к базовому классу

Ранее вы уже видели объявления классов наподобие `class ClassA: pass` и `class ClassB(BaseClassName): pass`. Если базовый класс не указан явно, то Python автоматически устанавливает в его качестве встроенный класс `object`. В конечном счете все классы в Python наследуются от `object`. Имейте в виду: если базовый класс не указывается, то круглые скобки писать не обязательно, и на практике их обычно опускают.

Таким образом, все три варианта — `class A: pass`, `class A(): pass` и `class A(object): pass` — эквивалентны. Класс `object` — особый: он содержит общие методы, унаследованные всеми классами в Python, и не позволяет задавать собственные атрибуты.

Рассмотрим пример, как можно обратиться к базовому классу изнутри дочернего:

```
# oop/super.duplication.py
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        self.title = title
        self.publisher = publisher
        self.pages = pages
        self.format_ = format_
```

Здесь три параметра, передаваемые в `Book`, повторяются в `Ebook`. Это неудачная практика, поскольку теперь один и тот же код дублируется в двух местах. Кроме того, если сигнатура метода `__init__()` в классе `Book` изменится, эти изменения не затронут `Ebook`. В идеале при изменении базового класса дочерние

должны автоматически наследовать эти изменения. Посмотрим, как можно исправить эту проблему:

```
# oop/super.explicit.py
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        Book.__init__(self, title, publisher, pages)
        self.format_ = format_

ebook = Ebook(
    "Learn Python Programming", "Packt Publishing", 500, "PDF"
)
print(ebook.title)      # Learn Python Programming
print(ebook.publisher)  # Packt Publishing
print(ebook.pages)      # 500
print(ebook.format_)    # PDF
```

Стало гораздо лучше — мы избавились от дублирования кода. В этом примере мы явно указываем Python вызвать метод `__init__()` базового класса `Book`, передавая ему `self`, чтобы связать вызов с текущим экземпляром. Теперь, если мы изменим логику в методе `__init__()` класса `Book`, то нам не придется вносить изменения в `Ebook` — они будут унаследованы автоматически.

Этот подход хорош, но у него все же есть небольшой недостаток. Представим, что мы решили переименовать класс `Book` в `Liber` (слово «книга» на латинском). Тогда нам придется изменить вызов `Book.__init__()` в классе `Ebook`, чтобы все продолжило работать. Этого можно избежать, воспользовавшись функцией `super()`:

```
# oop/super.implicit.py
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        super().__init__(title, publisher, pages)
        # Другой способ сделать то же самое:
        # super(Ebook, self).__init__(title, publisher, pages)
        self.format_ = format_

ebook = Ebook(
    "Learn Python Programming", "Packt Publishing", 500, "PDF"
)
print(ebook.title)      # Learn Python Programming
print(ebook.publisher)  # Packt Publishing
print(ebook.pages)      # 500
print(ebook.format_)    # PDF
```

Функция `super()` возвращает прокси-объект, который делегирует вызовы методов родительскому или родственному классу.

Родственные классы — это классы, которые наследуются от одного и того же базового класса.

В нашем случае `super()` делегирует вызов методу `Book.__init__()`. И самое удобное: теперь мы можем переименовать `Book` в `Liber` и не менять ничего в коде метода `__init__()` класса `Ebook`.

А теперь, когда мы обсудили обращение к базовому классу из дочернего, перейдем к следующей теме — множественному наследованию в Python.

Множественное наследование

В Python можно определять классы, которые наследуются от нескольких базовых классов. Это называется *множественным наследованием*. В таких случаях поиск атрибутов может идти по нескольким путям. Посмотрите на схему (рис. 6.2).

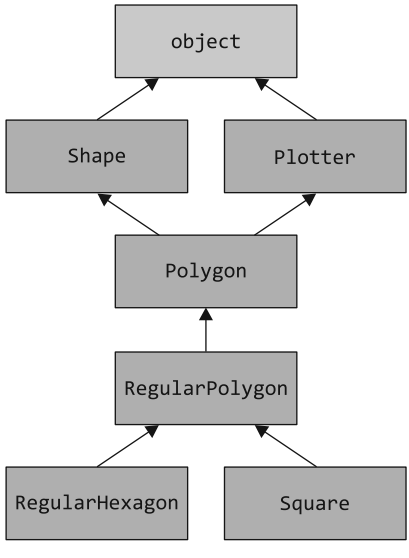


Рис. 6.2. Наследование классов

Как видите, `Shape` и `Plotter` выступают в роли базовых классов для остальных. `Polygon` наследуется напрямую от них, `RegularPolygon` — от `Polygon`, а классы `RegularHexagon` и `Square` — от `RegularPolygon`. Вдобавок обратите внимание вот на что: `Shape` и `Plotter` неявно наследуются от `object`, поэтому от `Polygon` до `object` проходит несколько путей наследования. Такая структура называется *ромбом*

(diamond). Проще говоря, существует несколько путей к базовому классу — и чуть позже мы увидим, почему это имеет значение. Теперь переведем схему наследования в код:

```
# oop/multiple.inheritance.py
class Shape:
    geometric_type = "Generic Shape"
    def area(self):                   # Это заглушка для интерфейса
        raise NotImplementedError

    def get_geometric_type(self):
        return self.geometric_type

class Plotter:
    def plot(self, ratio, topleft):
        # Представим, что здесь реализована красивая логика построения графика...
        print("Plotting at {}, ratio {}.".format(topleft, ratio))

class Polygon(Shape, Plotter):         # базовый класс для многоугольников
    geometric_type = "Polygon"

class RegularPolygon(Polygon):         # Является многоугольником
    geometric_type = "Regular Polygon"
    def __init__(self, side):
        self.side = side

class RegularHexagon(RegularPolygon):  # Является правильным многоугольником
    geometric_type = "RegularHexagon"
    def area(self):
        return 1.5 * (3**0.5 * self.side**2)

class Square(RegularPolygon):          # Является правильным многоугольником
    geometric_type = "Square"
    def area(self):
        return self.side * self.side

hexagon = RegularHexagon(10)
print(hexagon.area())                  # 259.8076211353316
print(hexagon.get_geometric_type())    # RegularHexagon
hexagon.plot(0.8, (75, 77))            # Построение в точке (75, 77), масштаб 0.8

square = Square(12)
print(square.area())                   # 144
print(square.get_geometric_type())     # Square
square.plot(0.93, (74, 75))            # Построение в точке (74, 75), масштаб 0.93
```

Итак, в данном коде класс `Shape` содержит один атрибут — `geometric_type`, а также два метода — `area()` и `get_geometric_type()`. Часто базовые классы (как `Shape` в этом примере) используются для определения *интерфейса* — набора методов, реализация которых должна быть предоставлена в дочерних классах. Существуют более надежные и формальные способы описания интерфейсов, но для простоты мы пока обходимся базовыми средствами.

Класс `Plotter` добавляет метод `plot()` — таким образом он предоставляет возможность визуализации для любого класса, который унаследует его. Конечно, в этом примере метод `plot()` реализован в виде простой заглушки — обычного вызова `print()`. Первый по-настоящему интересный класс — `Polygon`, который наследуется сразу от `Shape` и `Plotter`.

Существует множество типов многоугольников. Один из них — правильный многоугольник, у которого все углы равны (то есть он равноугольный) и все стороны равны (равносторонний). Для него мы создаем класс `RegularPolygon`, наследующий от `Polygon`. Так как все стороны равны, в классе `RegularPolygon` можно реализовать простой метод `__init__()`, принимающий только длину стороны. Далее мы создаем два класса — `RegularHexagon` и `Square`, которые наследуются от `RegularPolygon`.

Эта структура может показаться довольно громоздкой, но, надеемся, дает представление о том, как можно постепенно специализировать классификацию объектов.

Обратите внимание на последние восемь строк кода. Когда мы вызываем метод `area()` у `hexagon` и `square`, то получаем корректную площадь в обоих случаях. Это происходит потому, что в этих классах реализована собственная логика метода `area()`. Кроме того, мы вызываем `get_geometric_type()` у обоих объектов, хотя этот метод не определен в их собственных классах. Python начинает поиск и доходит до класса `Shape`, где и находит реализацию. При этом, хотя реализация находится в `Shape`, выражение `self.geometric_type` в методе возвращает значение, определенное в экземпляре, с которого был вызван метод.

Вызовы метода `plot()` тоже заслуживают внимания: они показывают, как можно добавить объектам дополнительные возможности, которых у них изначально не было. Такой прием широко используется, например, в веб-фреймворках, таких как Django. Там применяются специальные классы, называемые *миксинами* (mixins — «примеси»). Их функциональность можно просто «подмешать» в нужный класс — достаточно указать миксин среди его базовых классов.

Множественное наследование является мощным инструментом, но вместе с тем может привести к путанице. Поэтому крайне важно понимать, как именно оно работает.

Порядок разрешения методов

Теперь мы знаем: когда в Python выполняется обращение вида `someobject.attribute` и атрибут не найден в самом объекте, то Python начинает поиск в классе, на основе которого создан объект. Если атрибут не найден и там, то поиск продолжается вверх по цепочке наследования — до тех пор, пока атрибут не будет найден или пока не будет достигнут базовый класс `object`.

Такой порядок довольно легко понять, если речь идет об одиночном наследовании, где каждый класс имеет только одного родителя. Однако при множественном наследовании предсказать, в каком порядке Python будет искать атрибуты, может быть непросто.

Чтобы всегда иметь представление о порядке поиска, Python предоставляет механизм под названием *«порядок разрешения методов»* (method resolution order, MRO).

MRO определяет, в каком порядке базовые классы будут просматриваться при поиске метода или атрибута. Начиная с версии 2.3 Python использует алгоритм C3, который гарантирует монотонность, то есть стабильность и предсказуемость порядка.

Посмотрим, как выглядит MRO для класса `Square` из предыдущего примера:

```
# oop/multiple.inheritance.py
print(square.__class__.__mro__)
# выводит:
# (<class '__main__.Square'>, <class '__main__.RegularPolygon'>,
# <class '__main__.Polygon'>, <class '__main__.Shape'>,
# <class '__main__.Plotter'>, <class 'object'>)
```

Чтобы получить MRO класса, можно обратиться от экземпляра к его атрибуту `__class__`, а затем к атрибуту `__mro__`. Альтернативно можно использовать `Square.__mro__` или вызвать `Square.mro()` напрямую. Но если вы работаете с экземпляром и хотите получить MRO, то придется динамически определить его класс.

Единственная неоднозначность в примере с `Square` возникает после `Polygon`, поскольку оттуда идут две ветви наследования: одна — к `Shape`, другая — к `Plotter`. Просмотр MRO для `Square` показывает, что сначала просматривается `Shape`, а уже затем `Plotter`.

Почему это важно? Рассмотрим следующий пример:

```
# oop/mro.simple.py
class A:
    label = "a"

class B(A):
    label = "b"

class C(A):
    label = "c"

class D(B, C):
    pass

d = D()
print(d.label) # Гипотетически здесь может быть либо 'b', либо 'c'
```

Здесь классы `B` и `C` наследуются от `A`, а `D` — от `B` и `C`. Это означает, что поиск атрибута `label` может пройти к `A` либо через `B`, либо через `C`. Какой путь будет выбран — зависит от порядка в MRO, и результат может различаться.

В этом примере результат — `'b'`, что ожидаемо: класс `B` — первый в списке базовых классов `D`. А что произойдет, если удалить атрибут `label` из `B`? Python поднимется вверх к `A` или сначала обратится к `C`?

Проверим:

```
# oop/mro.py
class A:
    label = "a"

class B(A):
    pass # было: label = 'b'

class C(A):
    label = "c"

class D(B, C):
    pass

d = D()
print(d.label) # 'c'
print(d.__class__.mro()) # это еще один способ получить порядок разрешения методов (MRO)
# выводит:
# [<class '__main__.D'>, <class '__main__.B'>,
# <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

Мы видим, что MRO для `D` — это: `D → B → C → A → object`. Поэтому при обращении `d.label` получаем `'c'`.

В повседневной разработке нечасто приходится задумываться о MRO, но мы сочли важным упомянуть его в этом подразделе — на случай, если вы столкнетесь со сложной структурой с миксинами. Зная, как работает MRO, вы сможете разобраться в происходящем и выбрать подходящий вам вариант.

Статические методы и методы класса

До сих пор мы писали классы с атрибутами данных и методами экземпляра, но в определении класса можно встретить и два других типа методов: *статические методы* и *методы класса*.

Статические методы

Когда создается объект класса, Python присваивает ему имя, которое выступает в роли пространства имен. В ряде случаев удобно сгруппировать связанные функции именно в рамках такого пространства. Статические методы идеально подходят для таких задач. В отличие от методов экземпляра они не требуют передачи экземпляра (`self`) при вызове.

Рассмотрим пример:

```
# oop/static.methods.py
class StringUtil:
    @staticmethod
    def is_palindrome(s, case_insensitive=True):
        # разрешаем только буквы и цифры
        s = "".join(c for c in s if c.isalnum())  # Обратите внимание!
        # для нечувствительного к регистру сравнения переводим строку s
        # в нижний регистр
        if case_insensitive:
            s = s.lower()
        for c in range(len(s) // 2):
            if s[c] != s[-c - 1]:
                return False
        return True

    @staticmethod
    def get_unique_words(sentence):
        return set(sentence.split())

print(
    StringUtil.is_palindrome("Radar", case_insensitive=False)
)      # False: с учетом регистра
print(StringUtil.is_palindrome("A nut for a jar of tuna"))  # True
print(StringUtil.is_palindrome("Never Odd, Or Even!"))  # True
print(
    StringUtil.is_palindrome(
        "In Girum Imus Nocte Et Consumimur Igni"
    )  # Латинский палиндром
)  # True

print(
    StringUtil.get_unique_words(
        "I love palindromes. I really really love them!"
    )
)
# {'them!', 'palindromes.', 'I', 'really', 'love'}
```

Этот код заслуживает внимания. Для начала мы видим, что статические методы создаются с помощью декоратора `@staticmethod`. Такие методы не принимают дополнительных аргументов, поэтому внешне просто выглядят как обычные функции, только определенные внутри класса.

В примере есть класс `StringUtil`, который используется как контейнер для функций. Можно было бы оформить их в виде отдельного модуля, но это уже вопрос стиля и предпочтений.

Логика внутри метода `is_palindrome()` должна быть уже вам понятна, но все же кратко ее разберем. Сначала из строки `s` удаляются все символы, не являющиеся буквами или цифрами. Для этого вызывается метод `join()` у пустой строки, и ему передается генераторное выражение, которое последовательно возвращает все алфавитно-цифровые символы из `s`. Вызов `join()` у пустой строки означает, что

все элементы из переданного итерируемого объекта будут объединены в одну строку. Это стандартный прием при проверке палиндромов.

Если параметр `case_insensitive` установлен в `True`, то строка `s` переводится в нижний регистр. Далее происходит основная проверка: мы сравниваем первую и последнюю буквы, затем вторую и предпоследнюю и т. д. Если на каком-либо шаге находится несовпадение, это значит, что строка не является палиндромом и можно сразу вернуть `False`. Если же цикл завершается без выхода по условию, значит, различий не найдено и строка считается палиндромом.

Обратите внимание: этот код работает корректно при любой длине строки, независимо от количества символов в ней. Выражение `len(s) // 2` задает половину длины строки, и если количество символов нечетное, то средний символ не сравнивается (например, в слове `RaDaR` буква `D` не проверяется). Но это не имеет значения — символ все равно бы сравнивался сам с собой.

Метод `get_unique_words()` устроен гораздо проще: он возвращает множество, в которое мы передаем список слов из предложения. Класс `set` автоматически удаляет повторы, так что нам не нужно делать ничего дополнительно.

Класс `StringUtil` играет роль контейнера-пространства имен для методов, предназначенных для работы со строками. По аналогии можно было бы создать, например, `MathUtil` — с набором статических методов для работы с числами.

Методы класса

Эти методы немного отличаются от статических. Как и методы экземпляра, они получают специальный первый аргумент. Но в отличие от `self`, который ссылается на экземпляр, в случае метода класса этот аргумент — сам класс, и по соглашению он называется `cls`. Один из самых распространенных вариантов применения таких методов — создание фабрик (factories), то есть альтернативных способов создания экземпляров класса. Рассмотрим пример:

```
# oop/class.methods.factory.py
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def from_tuple(cls, coords):  # cls — это Point
        return cls(*coords)

    @classmethod
    def from_point(cls, point):  # cls — это Point
        return cls(point.x, point.y)

p = Point.from_tuple((3, 7))
print(p.x, p.y)  # 3 7
q = Point.from_point(p)
print(q.x, q.y)  # 3 7
```

Здесь показано, как с помощью метода класса реализовать фабрику для класса `Point`. В обычном случае объект создается с передачей координат напрямую (`p = Point(3, 7)`), но мы хотим также уметь создавать точку из кортежа (`Point.from_tuple()`) и копировать другую точку (`Point.from_point()`).

Во всех этих методах первым аргументом выступает `cls`, который указывает на сам класс `Point`. Это аналогично методу экземпляра, в котором первым аргументом передается `self`. Имена `self` и `cls` рекомендуются по соглашению. Вы не обязаны их использовать, но данное соглашение настоятельно рекомендуется соблюдать. Оно настолько строгое, что большинство профессиональных Python-разработчиков никогда не отклоняются от него — и многие инструменты (анализаторы, форматтеры, линтеры и др.) созданы с учетом использования именно этих имен.

Статические методы и методы класса хорошо сочетаются. Часто статические помогают разбить логику метода класса на части — чтобы код стал чище и понятнее.

Посмотрим, как можно переработать класс `StringUtil`, используя оба типа методов:

```
# oop/class.methods.split.py
class StringUtil:
    @classmethod
    def is_palindrome(cls, s, case_insensitive=True):
        s = cls._strip_string(s)
        # Для нечувствительного к регистру сравнения переводим
        # строку s в нижний регистр
        if case_insensitive:
            s = s.lower()
        return cls._is_palindrome(s)

    @staticmethod
    def _strip_string(s):
        return "".join(c for c in s if c.isalnum())

    @staticmethod
    def _is_palindrome(s):
        for c in range(len(s) // 2):
            if s[c] != s[-c - 1]:
                return False
        return True

    @staticmethod
    def get_unique_words(sentence):
        return set(sentence.split())

print(StringUtil.is_palindrome("radar"))             # True
print(StringUtil.is_palindrome("not a palindrome"))  # False
```

Сравните этот код с предыдущей версией. Прежде всего обратите внимание вот на что: `is_palindrome()` теперь стал методом класса, но вызов остался таким же, как и у статического метода.

Мы преобразовали его в метод класса, поскольку после выделения части логики во вспомогательные методы (`_strip_string()` и `_is_palindrome()`) нам нужна была ссылка на них. Если бы метод `is_palindrome()` оставался статическим, то единственным способом обращения к вспомогательным методам было бы использовать имя класса напрямую: `StringUtil._strip_string()` и `StringUtil._is_palindrome()`.

Но такой подход не рекомендуется, поскольку мы тем самым жестко привязываемся к имени класса. В случае его переименования пришлось бы вносить изменения в каждом месте, где оно использовалось. Вместо этого мы применяем `cls`, который играет роль имени текущего класса. Благодаря такому подходу код становится гибким и не требует доработки при изменении имени класса.

Как вы можете видеть, новая логика читается гораздо лучше, чем в прежней версии.

Обратите внимание еще вот на что: методы, в которые была *вынесена часть логики*, начинаются с подчеркивания (`_`). Это принятое соглашение, которое подсказывает, что такие методы не предназначены для вызова извне класса. Об этом мы поговорим подробнее в следующем подразделе.

Приватные методы и искажение имен

Если у вас есть опыт работы с такими языками, как Java, C# или C++, то вы знаете, что они позволяют задавать уровень доступа к атрибутам — как к данным, так и к методам. У каждого языка свои нюансы, но суть одна: публичные атрибуты доступны из любой части программы, а приватные — только в пределах определенной области видимости.

В Python такой системы нет. Здесь все считается публичным, поэтому для ограничения доступа используется соглашение об именовании и в случае необходимости специальный механизм, называемый *искажением имен*.

Соглашение таково: если имя атрибута не начинается с подчеркивания, то он считается публичным. Это означает, что к нему можно свободно обращаться и изменять его снаружи. Если имя начинается с одного подчеркивания, то атрибут считается приватным, то есть предназначенным только для внутреннего использования. Его не следует вызывать или изменять за пределами класса. На практике такое соглашение активно применяется для вспомогательных методов, используемых публичными методами (часто в составе цепочек вызовов); для внутренних данных, например коэффициентов масштабирования или любых других значений, которые в идеале следовало бы оформить как константы, то есть переменные, значение которых не должно изменяться. Однако в Python понятия константы нет и для подобных случаев тоже используется соглашение об именовании.

Мы знаем разработчиков, у которых отсутствие формальной системы доступа в Python вызывает дискомфорт. Но, по нашему опыту, отсутствие приватных атрибутов не становится источником ошибок. Все держится на дисциплине, здравом смысле и соблюдении принятых соглашений.

Свобода, которую Python предоставляет разработчику, — одна из причин, по которой его иногда называют *языком для взрослых*. У каждого проектного решения есть плюсы и минусы. В конечном счете кому-то ближе языки, дающие больше контроля (и требующие больше ответственности), а кто-то предпочитает более строгие языки. Здесь нет правильного или неправильного — все зависит от предпочтений.

Тем не менее требование к приватности вполне оправданно: без нее вы действительно рискуете внести ошибки в код. Покажем, о чем идет речь:

```
# oop/private.attrs.py
class A:
    def __init__(self, factor):
        self._factor = factor

    def op1(self):
        print("Op1 with factor {}...".format(self._factor))

class B(A):
    def op2(self, factor):
        self._factor = factor
        print("Op2 with factor {}...".format(self._factor))

obj = B(100)
obj.op1()   # Op1 with factor 100...
obj.op2(42) # Op2 with factor 42...
obj.op1()   # Op1 with factor 42... <- Это ПЛОХО
```

В этом фрагменте есть атрибут `_factor`. Допустим, он настолько важен, что не должен изменяться во время выполнения программы после создания объекта, поскольку от него зависит поведение метода `op1()`. Мы обозначили его как приватный с помощью символа подчеркивания, но проблема в том, что вызов `obj.op2(42)` изменяет значение `_factor` и это изменение влияет на дальнейшую работу `op1()`.

Чтобы избежать такого поведения, мы можем добавить второе подчеркивание в начале имени:

```
# oop/private.attrs.fixed.py
class A:
    def __init__(self, factor):
        self.__factor = factor

    def op1(self):
        print("Op1 with factor {}...".format(self.__factor))
```

```
class B(A):
    def op2(self, factor):
        self.__factor = factor
        print("Op2 with factor {}...".format(self.__factor))

obj = B(100)
obj.op1()   # Op1 with factor 100...
obj.op2(42) # Op2 with factor 42...
obj.op1()   # Op1 with factor 100... <- А вот это хорошо!
```

Теперь все работает так, как и было задумано. Python в этом плане немного «магический»: здесь срабатывает механизм, называемый искажением имен.

Искажение имен означает, что любой атрибут, имя которого начинается минимум с двух подчеркиваний и не заканчивается несколькими подчеркиваниями, например `__my_attr`, будет внутренне переименован. Его имя будет заменено на вид `_<ИмяКласса>__my_attr`.

Это позволяет избежать конфликтов имен: при наследовании класс-потомок получит собственную версию такого «приватного» атрибута. Таким образом, имена не пересекаются между базовым и дочерним классами. Каждый объект — и класс, и экземпляр — хранит ссылки на свои атрибуты во внутреннем атрибуте `__dict__`. Посмотрим, что содержится в `obj.__dict__`, чтобы увидеть механизм искажения имен в действии:

```
# oop/private.attrs.py
print(obj.__dict__.keys())
# dict_keys(['_factor'])
```

Это и есть атрибут `_factor`, который мы видели в проблемной версии примера. А теперь посмотрите на версию, где используется `__factor`:

```
# oop/private.attrs.fixed.py
print(obj.__dict__.keys())
# dict_keys(['_A__factor', '_B__factor'])
```

Объект `obj` теперь содержит два разных атрибута: `_A__factor` (искаженное имя в классе `A`) и `_B__factor` (искаженное имя в классе `B`). Именно так работает механизм искажения имен. Благодаря этому, выполняя `obj.factor = 42`, вы на самом деле создаете или изменяете `_B__factor`, а `_A__factor` остается неизменным. Таким образом, классы не конфликтуют между собой, даже если используют одинаковые имена «приватных» атрибутов.

Если вы разрабатываете библиотеку, в которой классы предполагается расширять и наследовать, то обязательно помните о механизме искажения имен. Это поможет избежать непреднамеренного переопределения важных атрибутов. Ошибки такого рода бывают коварными — их легко упустить из виду.

Декоратор property

Нельзя не упомянуть еще один важный инструмент — декоратор `property`. Представьте, что в классе `Person` у вас есть атрибут `age` и вы хотите удостовериться, что при изменении его значения выполняется проверка корректности — например, что возраст находится в пределах от 18 до 99.

Можно было бы написать методы `get_age()` и `set_age()` (так называемые *геттеры* и *сеттеры*) и поместить логику туда. Метод `get_age()`, скорее всего, просто вернет значение `age`, а `set_age()` — проверит входное значение перед присвоением. Проблема в том, что в коде может уже быть прямой доступ к атрибуту `age`, и в этом случае придется переписывать эти места или выполнять их рефакторинг. В таких языках, как Java, эта проблема решается с помощью *паттерна доступа* (accessor pattern), который обычно используется по умолчанию. Многие Java-ориентированные *интегрированные среды разработки* (Integrated Development Environments, IDEs) автоматически создают заготовки геттеров и сеттеров при объявлении атрибута.

Но мы изучаем не Java. В Python аналогичный результат достигается с помощью декоратора `property`. Если украсить метод этим декоратором, то его можно будет вызывать как обычный атрибут — без круглых скобок. Поэтому важно не помещать в такие методы долгую или ресурсоемкую логику: при доступе к атрибуту пользователь не ожидает задержек.

Рассмотрим пример:

```
# oop/property.py
class Person:
    def __init__(self, age):
        self.age = age  # любой может изменить это значение

class PersonWithAccessors:
    def __init__(self, age):
        self._age = age

    def get_age(self):
        return self._age

    def set_age(self, age):
        if 18 <= age <= 99:
            self._age = age
        else:
            raise ValueError("Age must be within [18, 99]")

class PersonPythonic:
    def __init__(self, age):
        self._age = age

    @property
    def age(self):
```

```
        return self._age

    @age.setter
    def age(self, age):
        if 18 <= age <= 99:
            self._age = age
        else:
            raise ValueError("Age must be within [18, 99]")

person = PersonPythonic(39)
print(person.age)  # 39 — обратите внимание, что обращаемся как к атрибуту данных
person.age = 42    # Обратите внимание, что обращаемся как к атрибуту данных
print(person.age)  # 42
person.age = 100   # ValueError: Age must be within [18, 99]
```

Класс `Person` может быть самой первой версией, которую мы пишем. Затем понадобится добавить проверку диапазона, и в других языках пришлось бы переписать `Person` как, скажем, `PersonWithAccessors` и выполнить рефакторинг всего кода, который обращается к `Person.age`. В Python все гораздо проще. Мы просто переписываем `Person` как `PersonPythonic` (хотя на практике имя класса не меняется — здесь это сделано для наглядности). В новом варианте возраст хранится во *внутренней* переменной `_age`, а доступ к нему осуществляется через декораторы `@property` и `@<имя>.setter`. Это позволяет продолжать использовать экземпляры класса так же, как и раньше, — без изменений в вызывающем коде. *Геттер* — это метод, вызываемый при чтении атрибута. *Сеттер* — метод, вызываемый при записи значения в атрибут.

В отличие от языков, где парадигма геттеров и сеттеров применяется с самого начала, Python позволяет начать с простого кода и добавить свойства только тогда, когда они действительно нужны. Нет необходимости заранее засорять код вспомогательными методами только потому, что они могут понадобиться в будущем.

Кроме того, декоратор `property` поддерживает только чтение (если не определен метод-сеттер) и обработку удаления атрибута — через `@<имя>.deleter`. Более подробная информация по этой теме содержится в официальной документации Python.

Декоратор cached_property

Один из удобных вариантов применения свойства (`property`) — это случаи, когда при обращении к атрибуту нужно выполнить некую инициализацию. Например, когда необходимо подключиться к базе данных или выйти на внешний API.

В таких случаях часто требуется создать объект клиента, который знает, как взаимодействовать с нужной системой. И здесь свойство (`property`) позволяет

спрятать сложную логику настройки клиента и предоставить простой интерфейс для доступа. Рассмотрим пример:

```
# oop/cached.property.py
class Client:
    def __init__(self):
        print("Setting up the client...")

    def query(self, **kwargs):
        print(f"Performing a query: {kwargs}")

class Manager:
    @property
    def client(self):
        return Client()

    def perform_query(self, **kwargs):
        return self.client.query(**kwargs)
```

В этом примере есть упрощенный класс `Client`, который при создании экземпляра выводит строку `"Setting up the client…"`. У него также есть фиктивный метод `query()`, который просто выводит строку. Затем используется класс `Manager`, у которого есть свойство `client`, возвращающее новый экземпляр `Client` каждый раз, когда к нему обращаются, — например, при вызове `perform_query()`.

Если запустить этот код, то можно заметить, что каждый раз при вызове `perform_query()` создается новый клиент и, соответственно, выводится сообщение `"Setting up the client…"`. В ситуациях, когда создание клиента — дорогостоящая операция, такой подход будет неэффективным и приведет к избыточным затратам ресурсов. В этом случае лучше кэшировать созданный объект клиента. Для этого используется `cached_property`:

```
# oop/cached.property.py
class ManualCacheManager:
    @property
    def client(self):
        if not hasattr(self, "_client"):
            self._client = Client()
        return self._client

    def perform_query(self, **kwargs):
        return self.client.query(**kwargs)
```

Класс `ManualCacheManager` устроен чуточку умнее: его свойство `client` сначала проверяет, существует ли атрибут `_client` у экземпляра, с помощью встроенной функции `hasattr()`. Если нет, то создается новый экземпляр `Client` и сохраняется в `_client`. Впоследствии свойство просто возвращает это значение. Таким образом, при повторных обращениях к `client` создается только один экземпляр `Client` — при первом вызове. Все последующие вызовы возвращают уже готовый объект, не создавая его повторно.

Поскольку такой подход встречается очень часто, в Python 3.8 в модуль `functools` был добавлен декоратор `cached_property`. Его главное преимущество, по сравнению с ручным кэшированием, — в возможности легко сбросить кэш: если клиент должен быть обновлен, то достаточно удалить атрибут `client` — и при следующем обращении он будет создан заново.

Рассмотрим пример:

```
# oop/cached.property.py
from functools import cached_property

class CachedPropertyManager:
    @cached_property
    def client(self):
        return Client()

    def perform_query(self, **kwargs):
        return self.client.query(**kwargs)

manager = CachedPropertyManager()
manager.perform_query(object_id=42)
manager.perform_query(name_ilike="%Python%")

del manager.client  # Это приведет к созданию нового экземпляра Client
                    # при следующем вызове
manager.perform_query(age_gte=18)
```

При запуске этого кода мы получаем следующий результат:

```
Setting up the client...                          # новый клиент
Performing a query: {'object_id': 42}             # первый запрос
Performing a query: {'name_ilike': '%Python%'}    # второй запрос
Setting up the client...                          # новый клиент
Performing a query: {'age_gte': 18}               # третий запрос
```

Как видите, новый экземпляр `Client` создается только после того, как мы вручную удалили `manager.client`, и при следующем вызове `manager.perform_query()` он создается заново.

В версии Python 3.9 появился также декоратор `cache`, который можно использовать в сочетании с `property` для тех случаев, где `cached_property` не подходит. Как всегда, мы рекомендуем ознакомиться с официальной документацией языка и поэкспериментировать — это поможет лучше понять, когда и как применять эти механизмы.

Перегрузка операторов

Подход Python к перегрузке операторов можно по праву назвать элегантным. *Перегрузка оператора* означает, что можно задать оператору смысл в зависимости от контекста. Например, если мы работаем с числами, то оператор `+` означает сложение, а если с последовательностями — то конкатенацию.

Когда мы используем операторы, Python внутренне вызывает специальные методы. Например, вызов `a[k]` для словаря примерно соответствует `type(a).__getitem__(a, k)`. Эти специальные методы мы можем переопределить под нужды своего класса.

Рассмотрим пример: создадим класс, который хранит строку и возвращает `True`, если в строке содержится `'42'`, и `False` — если нет. Кроме того, добавим свойство длины, которое будет отражать длину хранимой строки:

```
# oop/operator.overloading.py
class Weird:
    def __init__(self, s):
        self._s = s

    def __len__(self):
        return len(self._s)

    def __bool__(self):
        return "42" in self._s

weird = Weird("Hello! I am 9 years old!")
print(len(weird)) # 24
print(bool(weird)) # False
weird2 = Weird("Hello! I am 42 years old!")
print(len(weird2)) # 25
print(bool(weird2)) # True
```

Полный список магических методов, которые можно переопределить для реализации собственной логики работы операторов в классе, вы найдете в разделе *Data Model* официальной документации Python.

Полиморфизм (краткий обзор)

Слово *«полиморфизм»* происходит от греческих слов *polys* («много») и *morphē* («форма») и означает предоставление одного интерфейса для объектов разных типов.

Помните пример с автомобилями? Мы вызываем `engine.start()` — независимо от того, какой именно это двигатель. Главное, чтобы у объекта был метод `start()`. Это и есть полиморфизм в действии.

В других языках, таких как Java, для того чтобы функция могла принимать объекты разных типов и вызывать у них один и тот же метод, эти типы должны быть реализованы через общий интерфейс. Таким образом, компилятор может гарантировать, что нужный метод существует, при условии, что объект реализует или наследует соответствующий интерфейс.

В Python все устроено иначе. Полиморфизм реализован неявно, и ничто не мешает вам вызывать метод у объекта, если вы уверены, что он там есть. С техни-

ческой точки зрения нет необходимости использовать интерфейсы или другие структурные паттерны, как в Java.

Существует особый вид полиморфизма, называемый *ad hoc полиморфизмом*, — с ним мы уже сталкивались в предыдущем подразделе, посвященном перегрузке операторов. Это способность оператора менять поведение в зависимости от типа данных, к которому он применяется.

Полиморфизм в Python позволяет разработчику использовать интерфейс объекта (его методы и свойства) без проверки его конкретного класса. Благодаря этому код становится более компактным и выразительным.

Мы не будем углубляться в тему полиморфизма, но настоятельно рекомендуем разобраться с ней самостоятельно — это расширит ваше понимание объектно-ориентированного подхода.

Классы данных

Прежде чем завершить разговор об объектно-ориентированном программировании, стоит упомянуть еще один важный аспект — *классы данных*. Они были добавлены в Python 3.7 согласно PEP 557 (https://peps.python.org/pep-0557/) и представляют собой изменяемые именованные кортежи с поддержкой значений по умолчанию. Если вы хотите освежить знания об именованных кортежах, то вернитесь к главе 2. А сейчас сразу перейдем к примеру:

```
# oop/dataclass.py
from dataclasses import dataclass

@dataclass
class Body:
    """Класс, представляющий физическое тело."""

    name: str
    mass: float = 0.0   # кг
    speed: float = 1.0  # м/с

    def kinetic_energy(self) -> float:
        return (self.mass * self.speed**2) / 2

body = Body("Ball", 19, 3.1415)
print(body.kinetic_energy())  # 93,755711375 Джоуля
print(body)                   # Body(name='Ball', mass=19, speed=3.1415)
```

В этом примере мы создаем класс, представляющий физическое тело, с методом для вычисления кинетической энергии по формуле $E_k = 1/2mv^2$. Атрибут `name` должен быть строкой, а `mass` и `speed` — числами с плавающей запятой (float), причем для обоих заданы значения по умолчанию. Обратите внимание: мы не писали метод `__init__()` вручную — он был автоматически сгенерирован декоратором

`@dataclass`. Этот же декоратор создает методы сравнения и строковое представление объекта, которое и выводится на экран в последней строке через `print()`.

Другой важный нюанс — как заданы name, mass и speed. Этот прием называется аннотированием типов, и ему будет посвящена глава 12.

Если вам интересно, то можете ознакомиться с полными спецификациями в PEP 557. А пока просто запомните: классы данных — более удобная и гибкая альтернатива именованным кортежам, особенно если нужно добавлять поведение или задавать значения по умолчанию.

Создание собственного итератора

Теперь вы знаете все, что нужно для написания собственного итератора. Для начала уточним, что означают термины «итератор» и «итерируемый объект».

- *Итерируемый объект* — это объект, который может возвращать свои элементы по одному. Списки, кортежи, строки и словари являются итерируемыми объектами. Кроме того, итерируемыми считаются пользовательские объекты, в которых определен либо метод `__iter__()`, либо метод `__getitem__()`.
- *Итератор* — объект, представляющий поток данных. Чтобы быть итератором, объект должен: реализовать метод `__iter__()`, который возвращает сам объект; реализовать метод `__next__()`, который возвращает следующий элемент потока. Когда элементы заканчиваются, `__next__()` должен выбросить исключение `StopIteration`. Встроенные функции `iter()` и `next()` на самом деле вызывают `__iter__()` и `__next__()` внутри себя, обеспечивая единый интерфейс для работы с итераторами.

Исключения будут подробно рассмотрены в главе 7. Они могут обозначать ошибки выполнения, но также используются для управления потоком выполнения. В частности, итерационный протокол Python напрямую использует исключения.

Реализуем итератор, который сначала возвращает все символы строки с нечетными индексами, а затем — с четными:

```
# iterators/iterator.py
class OddEven:
    def __init__(self, data):
        self._data = data
        self.indexes = list(range(0, len(data), 2)) + list(
            range(1, len(data), 2)
        )

    def __iter__(self):
        return self
```

```
    def __next__(self):
        if self.indexes:
            return self._data[self.indexes.pop(0)]
        raise StopIteration

oddeven = OddEven("0123456789")
print("".join(c for c in oddeven))  # 0246813579

oddeven = OddEven("ABCD")           # или вручную...
it = iter(oddeven)                  # это вызывает oddeven.__iter__ внутренне
print(next(it))                     # A
print(next(it))                     # C
print(next(it))                     # B
print(next(it))                     # D
```

Итак, здесь мы реализуем метод `__iter__()`, возвращающий сам объект; а также метод `__next__()`. Логика такая: нам нужно возвращать символы в порядке `_data[0]`, `_data[2]`, `_data[4]`, ..., затем `_data[1]`, `_data[3]`, `_data[5]` и т. д. — до тех пор, пока все элементы не будут выданы. Для этого мы заранее формируем список индексов вида `[0, 2, 4, 6, ..., 1, 3, 5, ...]`, в каждом вызове `__next__()` удаляем первый индекс из этого списка (`pop(0)`), а затем возвращаем соответствующий символ из `_data`. Когда список индексов становится пустым, мы вызываем исключение `StopIteration`, как того требует протокол итератора.

Есть и другие способы добиться такого поведения. Попробуйте реализовать альтернативный вариант и проверьте его на крайних случаях, таких как пустые строки или строки длиной 1, 2 и т. д.

Резюме

В этой главе вы познакомились с декораторами, разобрались, зачем они нужны, и рассмотрели несколько примеров, в том числе комбинирование нескольких декораторов и декораторы с аргументами, которые часто используются как фабрики декораторов.

Кроме того, мы затронули основы объектно-ориентированного программирования (ООП) в Python. Вы узнали все базовые принципы и теперь сможете уверенно читать код, который будет появляться в следующих главах. Мы обсудили типы методов и атрибутов, которые можно описывать в классе, разницу между наследованием и композицией, переопределение методов, свойства, перегрузку операторов и полиморфизм.

В завершение мы кратко обсудили итераторы, что поможет вам лучше понять работу генераторов, к которым мы еще вернемся.

В следующей главе мы перейдем к исключениям и контекстным менеджерам.

7 Исключения и контекстные менеджеры

Мы счастья ждем, а на порог

Валит беда...

Роберт Бернс (в пер. С. Маршака)

Эта известная фраза Роберта Бернса должна быть запечатлена в памяти каждого программиста. Даже если ваш код написан корректно, ошибки все равно будут возникать. Если не обрабатывать их должным образом, то они могут свести на нет даже самые продуманные планы.

Необработанные ошибки могут привести к сбоям программы или некорректному поведению. В зависимости от характера программного обеспечения последствия могут быть весьма серьезными. Поэтому ошибки крайне важно научиться обнаруживать и обрабатывать.

Мы советуем выработать привычку: всегда думать о том, какие ошибки могут возникнуть и как ваш код должен на них реагировать.

Эта глава посвящена ошибкам и неожиданным ситуациям. Вы узнаете, как работают *исключения* — механизм Python для сигнализации об ошибках и других исключительных событиях; а также что такое *контекстные менеджеры*, с помощью которых можно инкапсулировать и многократно использовать код обработки ошибок.

Исключения

Мы еще не разбирали тему исключений подробно, но вы, вероятно, уже имеете общее представление о том, что это такое. В предыдущих главах вы видели, что:

- при завершении итератора вызов `next()` вызывает исключение `StopIteration`;
- при попытке обратиться к элементу списка за пределами допустимого диапазона возникает `IndexError`;

- при обращении к несуществующему атрибуту объекта появляется `AttributeError`;
- при попытке получить значение по несуществующему ключу словаря возникает `KeyError`.

В этой главе мы углубимся в тему исключений и разберемся, как с ними работать.

Даже если операция или фрагмент кода выполнены корректно, могут возникать ситуации, при которых что-то идет не так. Например, при преобразовании пользовательского ввода из `str` в `int` пользователь может случайно ввести букву вместо цифры и значение не получится преобразовать в число; при делении чисел заранее не всегда известно, не окажется ли знаменатель равным нулю; при открытии файла может выясниться, что он поврежден или вовсе отсутствует.

Когда во время выполнения возникает ошибка, это называется *исключением*. Исключения не обязательно фатальны. Так, `StopIteration` — неотъемлемая часть механизмов генераторов и итераторов в Python. Обычно если вы не предусмотрели обработку, то исключение приводит к прерыванию выполнения программы. В одних случаях это именно то, что нужно, но в других нужно предотвратить сбой и управлять ошибками. Например, если пользователь пытается открыть поврежденный файл, то программа может сообщить об этом и дать возможность исправить ситуацию. Рассмотрим пример с несколькими исключениями:

```
# exceptions/first.example.txt
>>> gen = (n for n in range(2))
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> print(undefined_name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'undefined_name' is not defined
>>> mylist = [1, 2, 3]
>>> mylist[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> mydict = {"a": "A", "b": "B"}
>>> mydict["c"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
>>> 1 / 0
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Как видите, интерактивная оболочка Python довольно спокойно реагирует на ошибки. При возникновении исключения мы видим *трассировку* (traceback) — подробную информацию об ошибке, — но сама оболочка продолжает работать. Это особенность интерактивного режима. Однако в обычной программе или скрипте все работает иначе: если исключение не обработано, то программа немедленно завершает выполнение. Ниже приведен пример:

```
# exceptions/unhandled.py
1 + "one"
print("This line will never be reached")
```

Если запустить этот код, то будет выведено следующее:

```
$ python exceptions/unhandled.py
Traceback (most recent call last):
  File "exceptions/unhandled.py", line 2, in <module>
    1 + "one"
    ~~^~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Мы не добавили обработку исключения, поэтому Python завершает выполнение сразу после ошибки, предварительно выведя информацию об исключении.

Генерация исключений

До этого момента мы видели исключения, которые создавались интерпретатором Python при обнаружении ошибки во время выполнения. Однако вы можете выбрасывать исключения и самостоятельно, если в вашем коде возникла ситуация, которую вы считаете ошибочной. Для этого используется оператор `raise`. Рассмотрим пример:

```
# exceptions/raising.txt
>>> raise NotImplementedError("I'm afraid I can't do that")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotImplementedError: I'm afraid I can't do that
```

В Python нет ограничений на типы исключений, которые можно выбрасывать. Это позволяет выбирать тот тип исключения, который лучше всего описывает конкретную ситуацию. Кроме того, вы можете создавать собственные типы исключений (как это делать, вы увидите чуть позже). Обратите внимание: аргумент, переданный в `Exception`, выводится как часть сообщения об ошибке.

Встроенных типов исключений в Python очень много, и перечислить их все в книге невозможно. Полный список можно найти в официальной документации по адресу https://docs.python.org/3.12/library/exceptions.html#bltin-exceptions.

Определение собственных исключений

Как мы упомянули ранее, вы можете определять собственные исключения. На практике это встречается довольно часто — например, в библиотеках почти всегда есть свои исключения.

Вам нужно просто создать класс, наследующийся от другого класса-исключения. Все исключения в Python происходят от `BaseException`, но наследоваться от него напрямую не рекомендуется. Ваши пользовательские исключения следует наследовать от `Exception`. Это общее правило: почти все встроенные исключения Python тоже наследуются от `Exception`. Исключения, не унаследованные от `Exception`, предназначены для внутреннего использования интерпретатором Python.

Трассировки

Трассировка исключения, которую Python выводит при возникновении необработанной ошибки, на первый взгляд может показаться пугающей. Однако на самом деле она очень полезна — с ее помощью можно разобраться, что именно вызвало исключение. В этом примере мы используем математическую формулу для решения квадратных уравнений. Ее знать не обязательно — на понимание кода она не влияет. Мы просто хотим посмотреть, что может рассказать трассировка исключения:

```
# exceptions/trace.back.py
def squareroot(number):
    if number < 0:
        raise ValueError("No negative numbers please")
    return number**0.5

def quadratic(a, b, c):
    d = b**2 - 4 * a * c
    return (
        (-b - squareroot(d)) / (2 * a),
        (-b + squareroot(d)) / (2 * a)
    )

quadratic(1, 0, 1) # x**2 + 1 == 0
```

Здесь мы определяем функцию `quadratic()`, в которой используется формула для нахождения корней уравнения; вместо `sqrt()` из модуля `math` мы реализуем свою версию — `squareroot()`, которая вызывает исключение, если число отрицательное. При вызове `quadratic(1, 0, 1)` мы решаем уравнение `x2 + 1 = 0`, у которого

нет действительных решений — дискриминант (`d`) оказывается отрицательным. В результате вызывается исключение `ValueError`, и Python выводит трассировку исключения:

```
$ python exceptions/trace.back.py
Traceback (most recent call last):
  File "exceptions/trace.back.py", line 16, in <module>
    quadratic(1, 0, 1) # x**2 + 1 == 0
    ^^^^^^^^^^^^^^^^^^
  File "exceptions/trace.back.py", line 11, in quadratic
    (-b - squareroot(d)) / (2 * a),
          ^^^^^^^^^^^^
  File "exceptions/trace.back.py", line 4, in squareroot
    raise ValueError("No negative numbers please")
ValueError: No negative numbers please
```

Часто бывает полезно читать трассировку с конца. В самой последней строке приводится сообщение об ошибке, объясняющее, что именно пошло не так: `ValueError: No negative numbers please`. Предыдущие строки показывают, где именно была вызвана эта ошибка — на 4-й строке файла `exceptions/trace.back.py` в функции `squareroot()`.

Вдобавок мы видим последовательность вызовов функций, которая привела к возникновению исключения: `squareroot()` была вызвана на 11-й строке функцией `quadratic()`, которая, в свою очередь, была вызвана на 16-й строке, на верхнем уровне модуля.

Таким образом, трассировка — это своего рода карта, показывающая путь, по которому выполнялся код до момента возникновения исключения. Чтобы понять, почему возникла ошибка, часто бывает полезно пройтись по этому пути и проанализировать код в каждой из функций.

Начиная с версий Python 3.10, 3.11 и 3.12 сообщения об ошибках значительно улучшились. Так, символы `^^^^`, подчеркивающие точное место в выражении или инструкции, в которых возникло исключение, появились в Python 3.11.

Обработка исключений

Обрабатывать исключения в Python можно с помощью оператора `try`. Когда выполнение переходит в блок `try`, интерпретатор начинает следить за указанными типами исключений (в зависимости от того, какие вы укажете), и если одно из них возникает, то вы получаете возможность отреагировать на него.

Оператор `try` состоит из следующих частей:

- блок `try`, открывающий конструкцию;
- один или несколько блоков `except`, описывающих, как реагировать на определенные типы исключений;

- необязательный блок `else`, который выполняется, если в блоке `try` не произошло ни одного исключения;
- необязательный блок `finally`, код которого выполняется всегда — независимо от того, возникло исключение или нет. Этот блок обычно используют для освобождения ресурсов (например, закрытия файлов или сетевых соединений).

Вы также можете написать `try` без блока `except`, если добавите `finally`. Такой вариант бывает полезен, если вы хотите, чтобы исключение обрабатывалось в другом месте, но при этом вам нужно гарантированно выполнить завершающий код.

Порядок блоков имеет значение: сначала `try`, затем `except`, потом `else` и в конце `finally`. Кроме того, `try` обязательно должен сопровождаться хотя бы одним блоком `except` или `finally`.

В примере ниже показано, как все это работает.

```
# exceptions/try.syntax.py
def try_syntax(numerator, denominator):
    try:
        print(f"In the try block: {numerator}/{denominator}")
        result = numerator / denominator
    except ZeroDivisionError as zde:
        print(zde)
    else:
        print("The result is:", result)
        return result
    finally:
        print("Exiting")

print(try_syntax(12, 4))
print(try_syntax(11, 0))
```

В примере определяется простая функция `try_syntax()`. В ней выполняется деление двух чисел. Мы готовы перехватить исключение `ZeroDivisionError`, которое возникает при делении на ноль. Сначала управление передается в блок `try`. Если знаменатель не равен нулю, то переменной `result` присваивается результат деления, и после выхода из блока `try` выполнение продолжается в блоке `else`, где мы выводим `result` и возвращаем его. Обратите внимание: прежде чем функция вернет результат (что соответствует ее завершению), Python выполняет блок `finally`. Это происходит всегда — независимо от того, произошло исключение или нет.

Когда знаменатель равен нулю, сценарий меняется. Попытка выполнить `numerator / denominator` вызывает исключение `ZeroDivisionError`. Поэтому управление передается в блок `except`, и мы выводим сообщение об ошибке.

Блок `else` не выполняется, поскольку исключение было вызвано в блоке `try`. Но прежде, чем функция неявно вернет `None`, блок `finally` все равно отрабатывает.

Результат поможет вам убедиться, что порядок выполнения действительно соответствует описанию выше:

```
$ python exceptions/try.syntax.py
In the try block: 12/4
The result is: 3.0
Exiting
3.0
In the try block: 11/0
division by zero
Exiting
None
```

Иногда при выполнении блока `try` может возникнуть несколько типов исключений. Например, при вызове функции `divmod()` может быть вызвано исключение `ZeroDivisionError` (если второй аргумент равен нулю) или `TypeError` (если хотя бы один аргумент не число). Если вы хотите одинаково обрабатывать оба случая, то код можно оформить так:

```
# exceptions/multiple.py
values = (1, 2)
try:
    q, r = divmod(*values)
except (ZeroDivisionError, TypeError) as e:
    print(type(e), e)
```

Здесь перехватываются сразу оба исключения. Попробуйте изменить `values = (1, 2)` на `values = (1, 0)` или `values = ('one', 2)` — и увидите, как программа будет вести себя иначе.

Если же вам нужно обрабатывать разные исключения по-разному, то используйте несколько отдельных блоков `except`, например:

```
# exceptions/multiple.py
try:
    q, r = divmod(*values)
except ZeroDivisionError:
    print("You tried to divide by zero!")
except TypeError as e:
    print(e)
```

Имейте в виду, что исключение обрабатывается в первом блоке `except`, класс которого совпадает с классом исключения или является его родителем. Поэтому если вы пишете несколько блоков `except` подряд, то сначала размещайте более конкретные исключения, а обобщенные оставляйте на конец. Иначе обработка может перехватить исключение раньше, чем вы ожидали. Если использовать терминологию классов, то сначала должны идти производные, затем базовые. Кроме того, когда происходит исключение, выполняется только один подходящий обработчик, а остальные игнорируются.

В Python можно использовать блок except, не указывая тип исключения. Такой подход равнозначен except BaseException, но его не рекомендуется применять. Причина в том, что в этом случае будут перехвачены и системные исключения, предназначенные для работы интерпретатора. К таким исключениям относятся, в частности, SystemExit (вызывается при завершении работы интерпретатора через exit()) и KeyboardInterrupt (возникает при прерывании выполнения программы пользователем, например, нажатием сочетания Ctrl+C или клавиши Del (на некоторых компьютерах)).

Кроме того, из блока `except` можно явно выбросить новое исключение с помощью инструкции `raise`. Это бывает полезно, когда вы хотите заменить встроенное исключение или исключение сторонней библиотеки собственным. Такой подход часто используется при разработке библиотек, чтобы скрыть детали реализации от пользователя и предоставить более удобные и понятные ошибки:

```
# exceptions/replace.txt
>>> class NotFoundError(Exception):
...     pass
...
>>> vowels = {"a": 1, "e": 5, "i": 9, "o": 15, "u": 21}
>>> try:
...     pos = vowels["y"]
... except KeyError as e:
...     raise NotFoundError(*e.args)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'y'
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NotFoundError: y
```

По умолчанию Python считает, что исключение, выброшенное внутри блока `except`, — это неожиданная ошибка, и поэтому выводит трассировки как для первоначального, так и для нового исключения. Чтобы явно указать, что новое исключение выбрасывается намеренно, можно воспользоваться конструкцией `raise ... from ...`:

```
# exceptions/replace.py
>>> try:
...     pos = vowels["y"]
... except KeyError as e:
...     raise NotFoundError(*e.args) from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'y'
```

```
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NotFoundError: y
```

В результате сообщение об ошибке изменится, но в трассировке по-прежнему будут показаны оба исключения. Это удобно при отладке, поскольку сохраняется полная картина произошедшего. Если же вы хотите полностью подавить информацию об исходном исключении, то можете использовать `from None` вместо `from e` (попробуйте сделать это самостоятельно).

Помимо описанных выше вариантов, вы можете использовать raise без аргументов, чтобы повторно выбросить текущее исключение. Такой подход применяют, например, для логирования: вы фиксируете факт возникновения исключения, но не перехватываете и не заменяете его — оно продолжает распространяться дальше.

Начиная с версии Python 3.11 к исключениям можно добавлять пояснения. Это позволяет дополнить трассировку дополнительной информацией, не подавляя и не подменяя само исключение. Чтобы вы могли увидеть, как это работает, немного изменим пример с квадратным уравнением из начала главы и добавим к исключению пояснение:

```
# exceptions/note.py
def squareroot(number):
    if number < 0:
        raise ValueError("No negative numbers please")
    return number**0.5

def quadratic(a, b, c):
    d = b**2 - 4 * a * c
    try:
        return (
            (-b - squareroot(d)) / (2 * a),
            (-b + squareroot(d)) / (2 * a),
        )
    except ValueError as e:
        e.add_note(f"Cannot solve {a}x**2 + {b}x + {c} == 0")
        raise

quadratic(1, 0, 1)
```

Мы выделили строки, где к исключению добавляется пояснение с помощью метода `add_note()` и где оно выбрасывается повторно:

```
$ python exceptions/note.py
Traceback (most recent call last):
  File "exceptions/note.py", line 20, in <module>
```

```
    quadratic(1, 0, 1)
  File "exceptions/note.py", line 12, in quadratic
    (-b - squareroot(d)) / (2 * a),
          ^^^^^^^^^^^^
  File "exceptions/note.py", line 4, in squareroot
    raise ValueError("No negative numbers please")
ValueError: No negative numbers please
Cannot solve 1x**2 + 0x + 1 == 0
```

При запуске такой программы пояснение выводится сразу после основного сообщения об ошибке. Можно добавить несколько пояснений, вызывая `add_note()` несколько раз, — каждое из них должно быть строкой.

Программирование с использованием исключений требует аккуратности. Вы можете случайно скрыть баги, перехватив исключения, которые могли бы указать на проблему. Чтобы избежать таких ситуаций, придерживайтесь нескольких простых рекомендаций.

- Старайтесь сделать блок `try` максимально коротким. В нем должно быть только то, что может вызвать те исключения, которые вы хотите обработать.
- Пишите блоки `except` максимально конкретными. Может возникнуть соблазн написать просто `except Exception`, но, скорее всего, так вы перехватите больше исключений, чем планировали.
- Проверяйте, как ваш код ведет себя при ожидаемых и неожиданных ошибках. О написании тестов мы подробнее поговорим в главе 10.

Если будете следовать этим советам, риск ошибиться будет минимальным.

Группы исключений

Работая с большими наборами данных, останавливать выполнение при первой же ошибке не всегда удобно. Часто лучше обработать все данные и только потом сообщить обо всех возникших ошибках. Так пользователю будет проще разобраться с ними сразу, а не запускать процесс заново каждый раз, исправляя их по одной.

Один из способов добиться этого — собирать ошибки в список и возвращать его в конце. Однако при таком подходе нельзя воспользоваться конструкцией `try/except`, чтобы перехватить ошибки привычным способом. Некоторые библиотеки обходят это ограничение, создавая специальный класс-обертку для исключений и помещая туда все собранные ошибки. Благодаря этому можно обрабатывать такую обертку в `except` и затем извлекать из нее отдельные исключения.

Начиная с версии Python 3.11 в язык добавлен встроенный класс `ExceptionGroup`, специально предназначенный для такой группировки ошибок. Его использование дает дополнительное преимущество: трассировка ошибок включает в себя отдельную трассировку для каждого вложенного исключения.

Представим, что нам нужно проверить список возрастов и убедиться, что все значения — положительные целые числа. Мы могли бы написать нечто подобное:

```
# exceptions/groups/util.py

def validate_age(age):
    if not isinstance(age, int):
        raise TypeError(f"Not an integer: {age}")
    if age < 0:
        raise ValueError(f"Negative age: {age}")

def validate_ages(ages):
    errors = []
    for age in ages:
        try:
            validate_age(age)
        except Exception as e:
            errors.append(e)

    if errors:
        raise ExceptionGroup("Validation errors", errors)
```

Функция `validate_ages()` вызывает `validate_age()` для каждого элемента в списке `ages`. Все исключения, которые при этом возникают, она перехватывает и добавляет в список `errors`. Если после завершения цикла этот список не пуст, то выбрасывается исключение `ExceptionGroup`, передавая в него сообщение об ошибке (`"Validation errors"`) и сам список исключений.

Если мы вызовем эту функцию в консоли Python с данными, содержащими некорректные значения, то получим следующую трассировку:

```
# exceptions/groups/exc.group.txt
>>> from util import validate_ages
>>> validate_ages([24, -5, "ninety", 30, None])
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |   File "exceptions/groups/util.py", line 20, in validate_ages
  |     raise ExceptionGroup("Validation errors", errors)
  | ExceptionGroup: Validation errors (3 sub-exceptions)
  +-+---------------- 1 ----------------
  | Traceback (most recent call last):
  |   File "exceptions/groups/util.py", line 15, in validate_ages
  |     validate_age(age)
  |   File "exceptions/groups/util.py", line 8, in validate_age
  |     raise ValueError(f"Negative age: {age}")
  | ValueError: Negative age: -5
  +---------------- 2 ----------------
  | Traceback (most recent call last):
  |   File "exceptions/groups/util.py", line 15, in validate_ages
  |     validate_age(age)
  |   File "exceptions/groups/util.py", line 6, in validate_age
  |     raise TypeError(f"Not an integer: {age}")
  | TypeError: Not an integer: ninety
  +---------------- 3 ----------------
```

```
  | Traceback (most recent call last):
  |   File "exceptions/groups/util.py", line 15, in validate_ages
  |     validate_age(age)
  |   File "exceptions/groups/util.py", line 6, in validate_age
  |     raise TypeError(f"Not an integer: {age}")
  | TypeError: Not an integer: None
  +------------------------------------
```

В этой трассировке отображается ошибка `ExceptionGroup`, а также переданное сообщение (`"Validation errors"`) и указание, что группа содержит три вложенных исключения. Ниже показана трассировка каждого из этих исключений. Чтобы их было легче читать, каждое исключение нумеруется и отделяется пунктирной линией.

Обрабатывать `ExceptionGroup` можно так же, как и любые другие исключения:

```
# exceptions/groups/handle.group.txt
>>> from util import validate_ages
>>> try:
...     validate_ages([24, -5, "ninety", 30, None])
... except ExceptionGroup as e:
...     print(e)
...     print(e.exceptions)
...
Validation errors (3 sub-exceptions)
(ValueError('Negative age: -5'),
TypeError('Not an integer: ninety'),
TypeError('Not an integer: None'))
```

Обратите внимание, что получить доступ к вложенным исключениям можно через свойство `exceptions` (только для чтения).

PEP 654 (https://peps.python.org/pep-0654/), в котором указывалось, что ошибка `ExceptionGroup` была добавлена в язык, вводит новую разновидность конструкции `try/except`, которая позволяет перехватывать только вложенные исключения нужного типа. Для этого используется ключевое слово `except*` вместо обычного `except`. В примере с валидацией возрастов это позволяет задать отдельные обработчики для ошибок типа (например, `TypeError`) и ошибок значений (например, `ValueError`), не перебирая исключения вручную и не проверяя их тип:

```
# exceptions/groups/handle.nested.txt
>>> from util import validate_ages
>>> try:
...     validate_ages([24, -5, "ninety", 30, None])
... except* TypeError as e:
...     print("Invalid types")
...     print(type(e), e)
...     print(e.exceptions)
... except* ValueError as e:
...     print("Invalid values")
...     print(type(e), e)
```

```
...     print(e.exceptions)
...
Invalid types
<class 'ExceptionGroup'> Validation errors (2 sub-exceptions)
(TypeError('Not an integer: ninety'),
TypeError('Not an integer: None'))
Invalid values
<class 'ExceptionGroup'> Validation errors (1 sub-exception)
(ValueError('Negative age: -5'),)
```

Вызов `validate_ages()` выбрасывает группу исключений, содержащую три исключения: два экземпляра `TypeError` и один `ValueError`. Интерпретатор сопоставляет каждую конструкцию `except*` с вложенными исключениями. Первая конструкция `except*` подходит для всех `TypeError`, поэтому интерпретатор создает новый экземпляр `ExceptionGroup`, содержащий все соответствующие исключения, и присваивает его переменной `e` в теле этой ветки. Мы выводим строку `"Invalid types"`, затем тип `e`, его значение и список `e.exceptions`. Оставшиеся исключения затем проверяются на соответствие следующей ветке `except*`.

Теперь срабатывает совпадение с `ValueError`, и переменной `e` присваивается новая группа исключений с этими экземплярами. Снова выводятся строка `"Invalid values"`, затем тип, объект и список вложенных исключений. На данном этапе все исключения обработаны, и выполнение продолжается как обычно.

Важно понимать, что поведение конструкции `try/except*` отличается от обычного `try/except`. В обычной конструкции выполняется только первая ветка `except`, которая соответствует исключению. В конструкции `try/except*` выполняются все подходящие ветки `except*`, пока в группе не останется необработанных исключений. Если какие-либо исключения останутся, то будут повторно выброшены в виде новой группы `ExceptionGroup`:

```
# exceptions/groups/handle.nested.txt
>>> try:
...     validate_ages([24, -5, "ninety", 30, None])
... except* ValueError as e:
...     print("Invalid values")
...
Invalid values
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |   File "exceptions/groups/util.py", line 20, in validate_ages
  |     raise ExceptionGroup("Validation errors", errors)
  | ExceptionGroup: Validation errors (2 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "exceptions/groups/util.py", line 15, in validate_ages
    |     validate_age(age)
    |   File "exceptions/groups/util.py", line 6, in validate_age
    |     raise TypeError(f"Not an integer: {age}")
    | TypeError: Not an integer: ninety
    +---------------- 2 ----------------
```

```
    | Traceback (most recent call last):
    |   File "exceptions/groups/util.py", line 15, in validate_ages
    |     validate_age(age)
    |   File "exceptions/groups/util.py", line 6, in validate_age
    |     raise TypeError(f"Not an integer: {age}")
    | TypeError: Not an integer: None
    +------------------------------------
```

Еще один важный момент: если в блоке `try/except*` возникает исключение, которое не является экземпляром `ExceptionGroup`, то интерпретатор все равно сопоставляет его с ветками `except*`. Если находится совпадение, то исключение оборачивается в `ExceptionGroup`, прежде чем будет передано в тело `except*`:

```
# exceptions/groups/handle.nested.txt
>>> try:
...     raise RuntimeError("Ungrouped")
... except* RuntimeError as e:
...     print(type(e), e)
...     print(e.exceptions)
...
<class 'ExceptionGroup'> (1 sub-exception)
(RuntimeError('Ungrouped'),)
```

Таким образом, в конструкции `except*` можно всегда считать, что обрабатываемый объект — это экземпляр `ExceptionGroup`.

Исключения как инструмент управления потоком выполнения

Прежде чем перейти к контекстным менеджерам, стоит отметить еще одно применение исключений. Они могут использоваться не только для обработки ошибок. В следующем примере показано, как с помощью исключения можно выйти из нескольких вложенных циклов:

```
# exceptions/for.loop.py
n = 100
found = False

for a in range(n):
    if found:
        break
    for b in range(n):
        if found:
            break
        for c in range(n):
            if 42 * a + 17 * b + c == 5096:
                found = True
                print(a, b, c)  # 79 99 95
                break
```

Здесь три вложенных цикла перебирают комбинации чисел `a`, `b` и `c`, чтобы найти те, которые удовлетворяют определенному уравнению. В начале каждого внешнего цикла проверяется значение флага `found`, которому присваивается значение `True`, как только находится решение. Благодаря этому можно завершить все три цикла сразу после успешного совпадения. Однако подобный подход затрудняет чтение кода — логика выхода с помощью флага мешает восприятию основного алгоритма. Поэтому мы решили попробовать другой, более лаконичный способ:

```
# exceptions/for.loop.py
class ExitLoopException(Exception):
    pass

try:
    n = 100
    for a in range(n):
        for b in range(n):
            for c in range(n):
                if 42 * a + 17 * b + c == 5096:
                    raise ExitLoopException(a, b, c)
except ExitLoopException as ele:
    print(ele.args)  # (79, 99, 95)
```

Надеемся, вы оценили, насколько этот способ проще и приятнее. В альтернативной версии для выхода из всех трех циклов используется пользовательское исключение `ExitLoopException`. Как только подходящее решение находится, вызывается исключение с нужными значениями и передается для обработки в блок `except`. К тому же с помощью атрибута `args` можно легко получить переданные значения: логика выхода сосредоточена в одном месте и не загромождает основную часть алгоритма.

Теперь у вас есть четкое представление о том, как исключения помогают управлять не только ошибками, но и потоком выполнения, а также особыми ситуациями. Пора перейти к следующей теме — *контекстным менеджерам*.

Контекстные менеджеры

При работе с внешними ресурсами часто требуется выполнить действия по ее завершению — например, после записи данных в файл его нужно закрыть. Если не выполнить такие шаги, то могут возникнуть самые разные ошибки. Поэтому важно быть уверенными, что завершающий код выполнится даже в случае исключения. Для этого можно воспользоваться конструкцией `try/finally`, но она не всегда удобна и может привести к дублированию кода, особенно если при работе с определенным типом ресурсов очистка выполняется по одной и той же схеме. *Контекстные менеджеры* помогают решить эту задачу: они создают кон-

текст выполнения, в рамках которого можно работать с ресурсом, а при выходе из этого контекста — автоматически выполнить нужные завершающие действия, даже если возникло исключение.

Контекстные менеджеры применяются и в других ситуациях — например, когда нужно временно изменить глобальное состояние программы. Один из таких случаев — управление точностью вычислений с десятичными числами. В задачах анализа данных иногда требуется провести вычисление с заданной точностью, но при этом сохранить стандартную точность для остального кода. Это можно реализовать, например, так:

```
# context/decimal.prec.py
from decimal import Context, Decimal, getcontext, setcontext

one = Decimal("1")
three = Decimal("3")

orig_ctx = getcontext()
ctx = Context(prec=5)
setcontext(ctx)
print(f"{ctx}\n")
print("Custom decimal context:", one / three)
setcontext(orig_ctx)
print("Original context restored:", one / three)
```

Обратите внимание: сначала мы сохраняем текущий контекст, затем устанавливаем новый (с измененной точностью), выполняем вычисление, а после этого возвращаем исходный контекст.

Если вы подзабыли, то напомним: класс Decimal позволяет выполнять вычисления с произвольной точностью в десятичной системе. При необходимости можно вернуться к соответствующему разделу главы 2.

Запуск кода приведет к следующему результату:

```
$ python context/decimal.prec.py
Context(prec=5, rounding=ROUND_HALF_EVEN, Emin=-999999,
        Emax=999999, capitals=1, clamp=0, flags=[],
        traps=[InvalidOperation, DivisionByZero, Overflow])

Custom decimal context: 0.33333
Original context restored: 0.33333333333333333333333333333
```

При запуске приведенного примера будет напечатан объект контекста, чтобы показать, какие параметры он содержит. Остальная часть кода вполне корректна, но если в блоке `try` произойдет исключение до того, как мы вернем исходный

контекст, то последующие вычисления окажутся некорректными. Это можно исправить с помощью конструкции `try/finally`:

```
# context/decimal.prec.try.py
from decimal import Context, Decimal, getcontext, setcontext

one = Decimal("1")
three = Decimal("3")

orig_ctx = getcontext()
ctx = Context(prec=5)
setcontext(ctx)
try:
    print("Custom decimal context:", one / three)
finally:
    setcontext(orig_ctx)
print("Original context restored:", one / three)
```

Такой подход надежнее: даже если произойдет исключение, то исходный контекст гарантированно восстановится. Однако сохранять и восстанавливать контекст вручную каждый раз неудобно, к тому же при этом нарушается принцип DRY (Don't Repeat Yourself — «Не повторяйся»). Вместо этого мы можем воспользоваться контекстным менеджером `localcontext` из модуля `decimal`. Он автоматически установит нужный контекст на время выполнения блока и восстановит его после выхода из него:

```
# context/decimal.prec.ctx.py
from decimal import Context, Decimal, localcontext

one = Decimal("1")
three = Decimal("3")

with localcontext(Context(prec=5)) as ctx:
    print("Custom decimal context:", one / three)
print("Original context restored:", one / three)
```

Оператор `with` используется для входа во временный контекст, определяемый контекстным менеджером `localcontext`. При выходе из блока кода, обозначенного этим оператором, автоматически выполняется операция очистки, заданная контекстным менеджером, — в данном случае восстанавливается исходный контекст модуля `decimal`.

Кроме того, можно объединить несколько контекстных менеджеров в одном операторе `with`. Это особенно удобно, когда нужно одновременно работать с несколькими ресурсами:

```
# context/multiple.py
from decimal import Context, Decimal, localcontext

one = Decimal("1")
three = Decimal("3")
```

```
with (
    localcontext(Context(prec=5)),
    open("output.txt", "w") as out_file
):
    out_file.write(f"{one} / {three} = {one / three}\n")
```

Здесь мы одновременно входим в локальный контекст `decimal` и открываем файл (он тоже является контекстным менеджером). Выполняется вычисление, результат записывается в файл. Когда выполнение блока завершается, файл автоматически закрывается, а контекст чисел с плавающей запятой восстанавливается. Сейчас мы не будем углубляться в работу с файлами — этой теме посвящена глава 8.

До Python 3.10 заключение нескольких контекстных менеджеров в круглые скобки, как показано выше, вызывало SyntaxError. В более старых версиях языка нужно было либо уместить все менеджеры в одну строку, либо ставить знак переноса строки внутри круглых скобок вызова localcontext() или open().

Помимо контекстов `decimal` и работы с файлами, множество других объектов из стандартной библиотеки Python также поддерживают протокол контекстных менеджеров. Ниже описаны два из них.

- Объекты `socket`, реализующие низкоуровневый сетевой интерфейс, могут использоваться как контекстные менеджеры для автоматического закрытия сетевых соединений.
- Классы блокировок, применяемые для синхронизации в многопоточных программах, автоматически освобождают блокировку при выходе из блока.

В оставшейся части главы мы разберем, как писать собственные контекстные менеджеры.

Контекстные менеджеры на базе классов

Контекстные менеджеры работают за счет двух специальных методов: `__enter__()` вызывается перед входом в тело оператора `with`, а `__exit__()` — при выходе из него. Это значит, что можно создать собственный контекстный менеджер, написав класс, в котором реализованы оба метода:

```
# context/manager.class.py
class MyContextManager:
    def __init__(self):
        print("MyContextManager init", id(self))
    def __enter__(self):
        print("Entering 'with' context")
        return self
```

```
    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"{exc_type=} {exc_val=} {exc_tb=}")
        print("Exiting 'with' context")
        return True
```

В примере определен класс контекстного менеджера `MyContextManager`. Рассмотрим несколько важных моментов, на которые стоит обратить внимание. Метод `__enter__()` возвращает `self`. Это распространенная практика, но вовсе не обязательная — метод может возвращать что угодно, даже `None`. Возвращаемое значение будет присвоено переменной, указанной после `as` в операторе `with`. Метод `__exit__()` принимает три параметра: `exc_type`, `exc_val` и `exc_tb`. Если внутри тела `with` возникло исключение, то интерпретатор передаст в эти параметры *тип* исключения, его *значение* и объект *трассировки*. Если исключения не произошло, то все три параметра будут иметь значение `None`.

Кроме того, обратите внимание, что метод `__exit__()` возвращает `True`. Благодаря этому любое исключение, возникшее внутри блока `with`, будет подавлено — как если бы оно было обработано внутри конструкции `try/except`. Если бы метод вернул `False`, то исключение продолжило бы распространяться после завершения работы `__exit__()`. Возможность подавлять исключения позволяет использовать контекстный менеджер как обработчик исключений. Это удобно: можно один раз реализовать логику обработки и использовать ее везде, где это необходимо. Рассмотрим пример:

```
# context/manager.class.py
ctx_mgr = MyContextManager()

print("About to enter 'with' context")

with ctx_mgr as mgr:
    print("Inside 'with' context")
    print(id(mgr))
    raise Exception("Exception inside 'with' context")
    print("This line will never be reached")
print("After 'with' context")
```

Здесь экземпляр контекстного менеджера создается отдельно, перед оператором `with` — так легче отследить происходящее. Однако на практике чаще встречается более компактная запись: `with MyContextManager() as mgr`. При запуске кода будет выведен результат, показывающий, как работает наш контекстный менеджер:

```
$ python context/manager.class.py
MyContextManager init 140340228792272
About to enter 'with' context
Entering 'with' context
Inside 'with' context
140340228792272
exc_type=<class 'Exception'> exc_val=Exception("Exception inside
'with' context") exc_tb=<traceback object at 0x7fa3817c5340>
Exiting 'with' context
After 'with' context
```

В выводе также отображаются идентификаторы объектов — они подтверждают, что переменная `mgr` действительно ссылается на тот же объект, который вернул метод `__enter__()`. Попробуйте изменить возвращаемые значения в методах `__enter__()` и `__exit__()` и посмотрите, как это влияет на поведение кода.

Контекстные менеджеры на базе генераторов

Если вы реализуете класс, представляющий некий ресурс, который необходимо захватывать и освобождать, то логично сделать этот класс контекстным менеджером. Однако бывают случаи, когда поведение менеджера нужно реализовать, но нет подходящего класса, к которому можно было бы привязать данную логику. Например, может понадобиться повторно использовать обработку ошибок в нескольких местах. В таких случаях создавать отдельный класс исключительно ради контекстного менеджера излишне трудоемко.

В подобных ситуациях помогает декоратор `contextmanager` из модуля `contextlib`. Он принимает *генераторную функцию* и превращает ее в контекстный менеджер (если вы подзабыли, как работают генераторы, то вернитесь в главу 5). Этот декоратор оборачивает генератор в объект — контекстный менеджер. Метод `__enter__()` такого объекта запускает генератор и возвращает значение, переданное через `yield`. Если внутри блока `with` произойдет исключение, то метод `__exit__()` передаст его в генератор с помощью метода `throw`. При отсутствии же исключения `__exit__()` просто вызывает `next`, чтобы завершить генератор. Важно помнить, что такой генератор должен вызывать `yield` только один раз. Если генератор попытается выдать значение повторно, то будет вызвано исключение `RuntimeError`. Преобразуем наш предыдущий пример в контекстный менеджер на основе генератора:

```
# context/generator.py
from contextlib import contextmanager

@contextmanager
def my_context_manager():
    print("Entering 'with' context")
    val = object()
    print(id(val))
    try:
        yield val
    except Exception as e:
        print(f"{type(e)=} {e=} {e.__traceback__=}")
    finally:
        print("Exiting 'with' context")

print("About to enter 'with' context")
with my_context_manager() as val:
    print("Inside 'with' context")
    print(id(val))
    raise Exception("Exception inside 'with' context")
    print("This line will never be reached")
print("After 'with' context")
```

Вывод при выполнении будет схож с тем, который мы получили в примере с классом:

```
$ python context/generator.py
About to enter 'with' context
Entering 'with' context
139768531985040
Inside 'with' context
139768531985040
type(e)=<class 'Exception'> e=Exception("Exception inside 'with'
context") e.__traceback__=<traceback object at 0x7f1e65a42800>
Exiting 'with' context
After 'with' context
```

Большинство генераторных контекстных менеджеров устроены примерно так же, как `my_context_manager()` в этом примере. Сначала выполняется некий код инициализации, затем вызывается `yield` внутри блока `try`. В этом примере мы возвращаем произвольный объект, чтобы показать, что именно он будет присвоен переменной после ключевого слова `as` в операторе `with`. Однако нередко используется просто `yield` без значения — в таком случае возвращается `None`, что эквивалентно возврату `None` из метода `__enter__()` в классе — контекстном менеджере. Тогда переменная после `as` в блоке `with` обычно опускается.

У контекстных менеджеров на основе генераторов есть еще одно удобное свойство: их можно применять как декораторы. Это полезно, если нужно, чтобы вся функция выполнялась в контексте `with`, но не хочется увеличивать уровень вложенности. В таком случае можно просто декорировать функцию и избавиться от лишнего отступа.

Помимо декоратора contextmanager, модуль contextlib предлагает множество готовых полезных контекстных менеджеров. Документация по нему (https://docs.python.org/3/library/contextlib.html) содержит как описания, так и наглядные примеры их использования и реализации. Обязательно ознакомьтесь с ней.

Примеры, приведенные в этом разделе, были намеренно упрощены — они в целом бесполезны, а служат только для демонстрации принципов работы контекстных менеджеров. Изучите эту тему более основательно, чтобы иметь полное понимание того, как работает механизм входа в контекст и выхода из него. Затем начните писать собственные контекстные менеджеры — как на основе классов, так и с помощью генераторов. Например, попробуйте превратить конструкцию `try/except`, с помощью которой мы раньше прерывали вложенные циклы, в контекстный менеджер. Кроме того, хорошим кандидатом для преобразования может быть декоратор `measure`, с которым вы познакомились в главе 6.

Резюме

В этой главе мы обсудили исключения и контекстные менеджеры.

Вы увидели, что исключения в Python используются для сигнализации об ошибках. Вы узнали, как обрабатывать исключения, чтобы программа не завершалась со сбоем при возникновении ошибок.

Кроме того, мы показали, как выбрасывать исключения вручную, когда ошибка обнаружена в вашем коде, и как определять собственные типы исключений. Вы познакомились с группами исключений и с новой конструкцией `except*`, которая расширяет привычный синтаксис обработки ошибок. В завершение темы исключений мы обсудили, как с их помощью можно не только обрабатывать ошибки, но и управлять потоком выполнения.

Затем мы кратко рассмотрели контекстные менеджеры. Вы узнали, как оператор `with` помогает входить в контекст, в котором автоматически выполняются операции очистки при выходе. Вдобавок мы показали способы, позволяющие создавать собственные контекстные менеджеры — как на основе классов, так и с помощью генераторных функций.

В следующей главе вы увидите примеры использования контекстных менеджеров при работе с файлами и постоянном хранении данных.

8 Работа с файлами и хранение данных

> Дело не в том, что я такой умный. Просто
> я дольше других не сдаюсь при решении задачи.
>
> *Альберт Эйнштейн*

В предыдущих главах мы рассмотрели разные возможности языка Python. Примеры, которые мы приводили, были учебными: мы запускали их в интерактивной оболочке или в виде простых модулей. Они выполнялись, иногда что-то выводили в консоль — и тут же завершались, не оставляя следов своего мимолетного существования.

Настоящие приложения устроены иначе. Конечно, они тоже выполняются в оперативной памяти, но при этом взаимодействуют с сетями, дисками и базами данных. Они обмениваются информацией с другими программами и устройствами, используя форматы, подходящие для конкретной задачи.

Как и прежде, мы постараемся сохранить баланс между широтой тем и глубиной объяснений, чтобы к концу главы вы имели хорошее понимание основ и знали, где искать дополнительную информацию.

Управление файлами и каталогами

Для работы с файлами и каталогами в Python есть множество полезных инструментов. В примерах этой главы мы будем использовать модули `os`, `pathlib` и `shutil`. Поскольку мы будем читать и записывать данные на диск, то в некоторых примерах будет использоваться файл `fear.txt`, содержащий отрывок из книги *Fear*[1] Тик Нат Хана. Этот файл послужит основой для ряда демонстраций.

---

[1] *Хан Т. Н.* Бесстрашие. Мудрость, которая позволит вам пережить бурю. — М., 2018.

Открытие файлов

Открыть файл в Python просто — достаточно воспользоваться удобной функцией `open()`. Приведем базовый пример:

```
# files/open_try.py
fh = open("fear.txt", "rt")  # r: чтение, t: текстовый режим

for line in fh.readlines():
    print(line.strip())      # удалить пробельные символы и вывести

fh.close()
```

Мы вызываем `open()`, передаем имя файла и указываем режим открытия — в данном случае `"rt"` (чтение в текстовом режиме). Поскольку путь к файлу не указан явно, то Python предполагает, что файл `fear.txt` находится в той же папке, откуда запускается сценарий. Это значит, что если вы выполните данный код из другой папки, то файл может быть не обнаружен.

После открытия файла возвращается файловый объект, в примере обозначенный как `fh`. Это высокоуровневая абстракция, оборачивающая файловый дескриптор (file handle). Далее используется метод `readlines()` для построчного прохода по файлу, после чего они выводятся на экран. Метод `strip()` применяется к каждой строке, чтобы удалить лишние пробельные символы с обоих концов строки, в том числе символ перевода строки, ведь функция `print()` сама добавляет перевод. Такой подход с использованием `strip()` — удобный, но грубый. Если в строках файла есть значимые пробелы, которые нужно сохранить, то лучше обрабатывать содержимое более аккуратно. В конце файла вызывается метод `close()`, чтобы освободить связанные ресурсы.

Закрывать файл важно, чтобы не возникло ситуации, когда программа продолжает удерживать к нему доступ. В противном случае могут возникать проблемы, например утечки памяти и появление уведомлений, что файл нельзя удалить, поскольку он еще используется.

Поэтому стоит быть осторожнее и обернуть логику работы с файлом в конструкцию `try/finally`. Благодаря этому, независимо от того, произойдет ли ошибка при открытии либо чтении файла, метод `close()` все равно будет вызван:

```
# files/open_try.py
fh = open("fear.txt", "rt")

try:
    for line in fh.readlines():
        print(line.strip())

finally:
    fh.close()
```

Такая структура делает код более надежным, хотя основная логика осталась прежней.

Если конструкция try/finally вам незнакома, то вернитесь к подразделу «Обработка исключений» главы 7 и внимательно изучите его содержимое.

Пример можно еще больше упростить:

```
# files/open_try.py
fh = open("fear.txt")  # rt – режим по умолчанию

try:
    for line in fh:    # можно итерироваться напрямую по fh
        print(line.strip())

finally:
    fh.close()
```

Во-первых, `"rt"` — это режим по умолчанию, его можно не указывать. Во-вторых, вместо вызова `readlines()` можно просто итерироваться по самому файловому объекту `fh`[1]. Python часто предлагает сокращения, которые делают код короче и улучшают его читабельность.

Все приведенные примеры в результате выводят содержимое файла в консоль (полный текст можно найти в исходном коде).

```
An excerpt from Fear - By Thich Nhat Hanh

The Present Is Free from Fear

When we are not fully present, we are not really living. We are not really
there, either for our loved ones or for ourselves. If we are not there, then
where are we? We are running, running, running, even during our sleep. We run
because we are trying to escape from our fear. […]
```

Использование контекстного менеджера для открытия файла

Чтобы не писать конструкции `try/finally` повсюду в коде, Python предлагает более удобный и не менее безопасный способ — использовать контекстный менеджер. Ниже приведен пример:

```
# files/open_with.py
with open("fear.txt") as fh:
    for line in fh:
        print(line.strip())
```

[1] В этом случае файл считывается так же, построчно, без предварительной загрузки в память, в отличие от использования метода read(). — *Примеч. науч. ред.*

Этот код делает то же самое, что и предыдущий, но читается гораздо проще. Функция `open()` возвращает файловый объект, который можно использовать как контекстный менеджер. Это значит, что метод `close()` будет вызван автоматически при выходе из блока `with`, даже если произойдет ошибка во время чтения.

Чтение из файла и запись в файл

Теперь, когда вы умеете открывать файлы, посмотрим, как можно читать из них и записывать в них:

```
# files/print_file.py
with open("print_example.txt", "w") as fw:
    print("Hey I am printing into a file!!!", file=fw)
```

Здесь используется уже знакомая вам функция `print()`. Получив файловый объект (на этот раз с режимом `"w"`, что означает запись), мы можем передать его в параметр `file` функции `print()`, и вывод пойдет не в *стандартный поток* (на экран), а в указанный файл.

В Python стандартные потоки ввода, вывода и ошибок представлены объектами sys.stdin, sys.stdout и sys.stderr. Обычно sys.stdin соответствует клавиатуре, а sys.stdout и sys.stderr — консоли, если только потоки не были перенаправлены.

Приведенный выше код создает файл `print_example.txt`, если тот не существует, либо очищает его, если он уже есть, и записывает в него строку `Hey I am printing into a file!!!`.

Обрезка файла (truncating) означает удаление его содержимого без удаления самого файла. После обрезки файл по-прежнему существует в файловой системе, но является пустым.

Предыдущий пример справляется с задачей, но не совсем типичен для записи в файл. Посмотрим на более распространенный подход:

```
# files/read_write.py
with open("fear.txt") as f:
    lines = [line.rstrip() for line in f]

with open("fear_copy.txt", "w") as fw:  # w – запись
    fw.write("\n".join(lines))
```

Здесь мы сначала открываем файл `fear.txt` и построчно считываем его содержимое в список. Обратите внимание, что на этот раз вызывается метод `rstrip()`, чтобы удалить пробелы только с правой стороны каждой строки — это просто пример, показывающий альтернативу методу `strip()`.

Во второй части кода мы создаем новый файл `fear_copy.txt` и записываем в него все строки из списка `lines`, объединенные символом новой строки (`\n`). Python по умолчанию использует *универсальные переводы строк*, то есть даже если в исходном файле строки заканчиваются на другие символы (например, `\r\n` в Windows), то при чтении они преобразуются в `\n`. Это поведение можно настраивать, но обычно его достаточно. К слову, раз уж речь зашла о переводах строк, — по-вашему, какой перевод строки может оказаться пропущенным в копии?

Чтение и запись в двоичном режиме

Когда вы открываете файл с флагом `t` (или не указываете флаг вовсе, так как `t` используется по умолчанию), файл открывается в текстовом режиме. Это значит, что содержимое файла интерпретируется как текст.

Если же вы хотите записывать в файл байты, то его нужно открывать в *двоичном режиме*. Это особенно важно, когда вы работаете с файлами, содержащими не только текст — например, с изображениями, аудио- и видеоданными, а также с любыми другими файлами, имеющими какой-то другой формат.

Чтобы открыть файл в двоичном режиме, добавьте флаг `b` при вызове функции `open()`, как в примере ниже:

```
# files/read_write_bin.py
with open("example.bin", "wb") as fw:
    fw.write(b"This is binary data...")

with open("example.bin", "rb") as f:
    print(f.read())  # выводит: b'This is binary data...'
```

Здесь, чтобы упростить пример, мы используем обычный текст, представленный в виде байтовой строки. Префикс `b` перед строкой указывает, что она интерпретируется как последовательность байтов.

Защита файла от перезаписи

Как мы уже говорили, Python позволяет открывать файлы для записи. При использовании флага `w` файл создается заново, а его содержимое полностью стирается — то есть он перезаписывается, и прежние данные безвозвратно теряются. Если вы хотите открыть файл для записи только в том случае, если он еще не существует, то используйте флаг `x`. Он сообщает Python, что файл должен быть создан заново и операция должна завершиться ошибкой, если файл уже есть. Рассмотрим такой пример:

```
# files/write_not_exists.py
with open("write_x.txt", "x") as fw:  # сработает
    fw.write("Writing line 1")

with open("write_x.txt", "x") as fw:  # вызовет ошибку
    fw.write("Writing line 2")
```

Если вы выполните этот код, то в каталоге появится файл `write_x.txt` с одной строкой текста. На самом деле вторая часть данного примера не выполняется. В консоли появится сообщение об ошибке (путь к файлу здесь представлен в укороченном виде в целях удобства):

```
$ python write_not_exists.py
Traceback (most recent call last):
  File "write_not_exists.py", line 6, in <module>
    with open("write_x.txt", "x") as fw: # вызовет ошибку
         ^^^^^^^^^^^^^^^^^^^^^^^
FileExistsError: [Errno 17] File exists: 'write_x.txt'
```

Итак, существуют разные режимы открытия файла. Полный список флагов, доступных для открытия файлов, приведен в официальной документации Python (https://docs.python.org/3/library/functions.html#open).

Проверка существования файлов и каталогов

Если нужно убедиться в том, что файл или каталог существует (или, наоборот, отсутствует), это можно сделать с помощью модуля `pathlib`:

```
# files/existence.py
from pathlib import Path

p = Path("fear.txt")
path = p.parent.absolute()

print(p.is_file())   # True
print(path)          # /Users/fab/code/lpp4ed/ch08/files
print(path.is_dir()) # True

q = Path("/Users/fab/code/lpp4ed/ch08/files")
print(q.is_dir())    # True
```

Здесь создается объект `Path`, которому передается имя проверяемого текстового файла. Использование метода `parent()` приводит к образованию пути к папке, в которой находится файл, а метод `absolute()` извлекает абсолютный путь к ней.

Далее происходит проверка: действительно ли `fear.txt` — это файл и существует ли каталог, в котором он находится.

Раньше такие операции выполнялись с помощью модуля `os.path` из стандартной библиотеки. Однако `os.path` работает со строками, тогда как `pathlib` предоставляет полноценные классы для представления путей в файловой системе, с учетом различий между операционными системами. Поэтому рекомендуется использовать `pathlib` по умолчанию и обращаться к `os.path` только в тех случаях, когда другого выхода нет.

Управление файлами и каталогами

Рассмотрим несколько простых примеров работы с содержимым файлов и каталогов. В первом примере выполняются операции с содержимым файла:

```
# files/manipulation.py
from collections import Counter
from string import ascii_letters

chars = ascii_letters + " "

def sanitize(s, chars):
    return "".join(c for c in s if c in chars)

def reverse(s):
    return s[::-1]

with open("fear.txt") as stream:
    lines = [line.rstrip() for line in stream]

# запишем отраженную (зеркальную) версию файла
with open("raef.txt", "w") as stream:
    stream.write("\n".join(reverse(line) for line in lines))

# теперь можно подсчитать статистику
lines = [sanitize(line, chars) for line in lines]
whole = " ".join(lines)

# сравнение будет проводиться по приведенной к нижнему регистру строке whole
cnt = Counter(whole.lower().split())

# выведем N самых часто встречающихся слов
print(cnt.most_common(3))  # [('we', 17), ('the', 13), ('were', 7)]
```

В коде определены две функции: `sanitize()` удаляет из строки все символы, кроме букв и пробелов, а `reverse()` создает строку, записанную в обратном порядке.

Далее открывается файл `fear.txt`, и его содержимое считывается построчно в список. Создается новый файл `raef.txt`, в который записывается «зеркальная» версия текста по горизонтали. Содержимое сохраняется в файл путем выполнения одной операции — объединения строк с помощью символа новой строки. Интереснее всего финальный фрагмент. Переменная `lines` переопределяется: теперь она содержит версии строк, очищенные от лишних символов. Затем строки объединяются в одну, переводятся в нижний регистр, разбиваются на отдельные слова и передаются объекту `Counter`. Благодаря этому каждое слово считается независимо от регистра, а разделение методом `split()` по пробелам избавляет от необходимости вручную очищать строку от лишних пробелов. В результате на экран выводятся три самых часто встречающихся слова. Первое среди них — `we`,

что точно отражает акцент, который Тик Нат Хан ставит на взаимосвязи и коллективной ответственности:

```
$ python manipulation.py
[('we', 17), ('the', 13), ('were', 7)]
```

А теперь посмотрим на пример обработки, который ближе к операциям с диском, — здесь как раз пригодится модуль `shutil`:

```
# files/ops_create.py
import shutil
from pathlib import Path

base_path = Path("ops_example")

# выполним начальную очистку (на всякий случай)
if base_path.exists() and base_path.is_dir():
    shutil.rmtree(base_path)

# теперь создаем каталог
base_path.mkdir()

path_b = base_path / "A" / "B"
path_c = base_path / "A" / "C"
path_d = base_path / "A" / "D"
path_b.mkdir(parents=True)
path_c.mkdir()  # параметр parents уже не нужен – 'A' создан

# добавим три файла в ops_example/A/B
for filename in ("ex1.txt", "ex2.txt", "ex3.txt"):
    with open(path_b / filename, "w") as stream:
        stream.write(f"Some content here in {filename}\n")

shutil.move(path_b, path_d)

# можно также переименовать файлы
ex1 = path_d / "ex1.txt"
ex1.rename(ex1.parent / "ex1.renamed.txt")
```

Здесь сначала задается базовый путь, где будут созданы все нужные папки и файлы. Затем создаются две папки: `ops_example/A/B` и `ops_example/A/C`. Обратите внимание: при создании папки `C` не требуется указывать `parents=True`, поскольку все родительские папки уже созданы предыдущим вызовом для `B`.

Для объединения имен папок используется оператор `/`, благодаря которому модуль `pathlib` автоматически подставляет нужный разделитель пути, в зависимости от операционной системы.

После создания папок в папке `B` создаются три файла. Затем она со всем содержимым переименовывается в `D`. Альтернативный способ — воспользоваться методом `rename()`, вызвав его у `path_b` с аргументом `path_d`.

Наконец, файл `ex1.txt` переименовывается в `ex1.renamed.txt`. Его содержимое не изменилось — оно было записано в процессе создания файла в цикле. Если после выполнения этих операций запустить команду `tree`, то можно увидеть структуру созданных файлов и папок.

```
$ tree ops_example
ops_example
└── A
    ├── C
    └── D
        ├── ex1.renamed.txt
        ├── ex2.txt
        └── ex3.txt
```

Пути к файлам и каталогам

Рассмотрим еще один пример, в котором продемонстрированы возможности модуля `pathlib`, на этот раз — по части работы с путями:

```
# files/paths.py
from pathlib import Path

p = Path("fear.txt")

print(p.absolute())
print(p.name)
print(p.parent.absolute())
print(p.suffix)

print(p.parts)
print(p.absolute().parts)

readme_path = p.parent / ".." / ".." / "README.rst"
print(readme_path.absolute())
print(readme_path.resolve())
```

Вывод для этого примера сам по себе является неплохим объяснением:

```
$ python paths.py
/Users/fab/code/lpp4ed/ch08/files/fear.txt
fear.txt
/Users/fab/code/lpp4ed/ch08/files
.txt
('fear.txt',)
(
    '/', 'Users', 'fab', 'code', 'lpp4ed',
    'ch08', 'files', 'fear.txt'
)
/Users/fab/code/lpp4ed/ch08/files/../../README.rst
/Users/fab/code/lpp4ed/README.rst
```

Особенно интересно происходящее в последних двух строках. Метод `absolute()` возвращает путь, содержащий две точки (`..`), которые обозначают переход на уровень выше в файловой иерархии. В данном случае это означает, что от папки `…/lpp4e/ch08/files/` путь поднимается дважды и в итоге приводит к `…/lpp4e/`. Это подтверждается выводом метода `resolve()`, который возвращает окончательный путь после того, как все переходы `..` и символические ссылки (если есть) были учтены.

Временные файлы и каталоги

Иногда необходимо создать временный каталог или файл. Например, при написании тестов, которые влияют на файловую систему, удобно использовать временные файлы и каталоги, чтобы выполнять нужную логику и проверять корректность результата. Вдобавок это гарантирует, что после завершения теста не останется лишних данных на диске. Посмотрим, как это можно сделать в Python:

```
# files/tmp.py
from tempfile import NamedTemporaryFile, TemporaryDirectory

with TemporaryDirectory(dir=".") as td:
    print("Temp directory:", td)
    with NamedTemporaryFile(dir=td) as t:
        name = t.name
        print(name)
```

Предыдущий пример довольно простой: мы создаем временный каталог в текущей папке и помещаем в него временный именованный файл. После этого выводим имя файла и его полный путь.

```
$ python tmp.py
Temp directory: /Users/fab/code/lpp4ed/ch08/files/tmpqq4quhbc
/Users/fab/code/lpp4ed/ch08/files/tmpqq4quhbc/tmpypwwhpwq
```

Следует помнить, что при каждом запуске такого скрипта будет получаться разный результат — временные имена создаются случайным образом.

Содержимое каталогов

В Python можно легко просматривать содержимое каталогов. Существует несколько способов сделать это. Один из них — воспользоваться методом `glob()` объекта `Path`:

```
# files/listing.py
from pathlib import Path

p = Path(".")

for entry in p.glob("*"):
    print("File:" if entry.is_file() else "Folder:", entry)
```

В примере рассматривается текущий каталог: мы перебираем все найденные элементы и проверяем, является ли каждый из них каталогом. Для каждого элемента выводится соответствующая информация. Результаты зависят от операционной системы, так как объекты `Path` будут экземплярами `PosixPath` или `WindowsPath` (показан фрагмент для краткости):

```
$ python listing.py
File: existence.py
File: manipulation.py
…
File: open_try.py
File: walking.pathlib.py
```

Альтернативный способ — использовать метод `walk()` объекта `Path`, чтобы обойти дерево каталогов:

```
# files/walking.pathlib.py
from pathlib import Path

p = Path(".")
for root, dirs, files in p.walk():
    print(f"{root=}")

    if dirs:
        print("Directories:")
        for dir_ in dirs:
            print(dir_)
        print()

    if files:
        print("Files:")
        for filename in files:
            print(filename)
        print()
```

Такой подход позволяет получить список всех файлов и подкаталогов, начиная с указанного каталога. В репозитории с кодом к книге также есть модуль `walking.py`, в котором используется функция `os.walk()` — она решает ту же задачу, но работает по-другому.

Сжатие файлов и каталогов

Прежде чем завершать раздел, рассмотрим пример создания сжатого файла. В папке `files/compression` с исходным кодом главы приведены два примера: один создает архив в формате ZIP, другой — в формате tar.gz. Python поддерживает создание архивов в нескольких форматах, и самым распространенным считается ZIP:

```
# files/compression/zip.py
from zipfile import ZipFile
```

```
with ZipFile("example.zip", "w") as zp:
    zp.write("content1.txt")
    zp.write("content2.txt")
    zp.write("subfolder/content3.txt")
    zp.write("subfolder/content4.txt")

with ZipFile("example.zip") as zp:
    zp.extract("content1.txt", "extract_zip")
    zp.extract("subfolder/content3.txt", "extract_zip")
```

Здесь используется класс `ZipFile`, с помощью которого создается ZIP-архив, содержащий четыре файла (в том числе из вложенной папки — так демонстрируется сохранение пути при упаковке). Затем показано, как открыть архив и извлечь отдельные файлы в папку `extract_zip`. Если вы хотите лучше разобраться в теме сжатия данных, то загляните в раздел *Data Compression and Archiving* официальной документации стандартной библиотеки (https://docs.python.org/3.9/library/archiving.html). Там вы найдете более подробную информацию и примеры.

Форматы обмена данными

Современные архитектуры программных систем часто предполагают разбиение приложения на отдельные компоненты. Даже если вы не используете принципы сервис-ориентированной архитектуры или микросервисы, все равно возникает необходимость обмена данными между этими компонентами. И даже при создании монолитного приложения, размещенного в одном проекте, скорее всего, придется взаимодействовать с внешними API, другими программами или просто организовывать обмен данными между фронтендом и бэкендом вашего сайта, а они, скорее всего, будут «говорить» на разных языках.

Выбор подходящего формата для обмена данными имеет ключевое значение. Формат, зависящий от конкретного языка программирования, удобен тем, что язык предоставляет все необходимое для *сериализации* и *десериализации* данных. Однако при этом теряется совместимость с другими компонентами, написанными на других версиях языка или вовсе на других языках. Поэтому использовать специфический для языка формат следует только в крайнем случае, когда других вариантов действительно нет.

Согласно Википедии (https://ru.wikipedia.org/wiki/Сериализация),

> «сериализация (в программировании) — процесс перевода структуры данных в оперативной/постоянной памяти в последовательность битов, байтов, слов, символов или других кодовых единиц. Сериализация используется для передачи объектов по сети и для сохранения их в файлы».

Более безопасный подход — выбирать формат, независимый от языка программирования. В программной инженерии со временем сформировались форматы, ставшие де-факто стандартами обмена данными. Наиболее известные из них — *XML*, *YAML* и *JSON*. В стандартную библиотеку Python входят

модули `xml` и `json`, а для работы с YAML можно найти несколько пакетов в каталоге PyPI (https://pypi.org).

В экосистеме Python наиболее распространен именно JSON. Он выигрывает у других форматов благодаря своей простоте и тому, что входит в стандартную библиотеку. XML обычно оказывается чересчур многословным и менее удобным для чтения.

Кроме того, при работе с базами данных, например PostgreSQL, наличие встроенных JSON-полей стимулирует использование JSON и в самом приложении.

Работа с JSON

JSON — это аббревиатура от *JavaScript Object Notation* (нотация объектов JavaScript). Формат представляет собой подмножество языка JavaScript, но при этом не зависит от конкретного языка программирования. Он используется уже почти два десятилетия и получил широкое распространение в различных языках. Подробнее о JSON можно прочитать на официальном сайте (https://www.json.org), а здесь мы кратко рассмотрим его основы.

JSON основан на двух структурах, таких как:

- набор пар «имя — значение»;
- упорядоченный список значений.

Неудивительно, что эти структуры соответствуют типам данных `dict` и `list` в Python. JSON поддерживает следующие типы значений: строки, числа, объекты, списки, а также логические значения `true`, `false` и значение `null`. Далее представлен простой пример, который поможет начать работать с JSON:

```
# json_examples/json_basic.py
import sys
import json

data = {
    "big_number": 2**3141,
    "max_float": sys.float_info.max,
    "a_list": [2, 3, 5, 7],
}

json_data = json.dumps(data)
data_out = json.loads(json_data)

assert data == data_out  # данные преобразованы в JSON и обратно — все совпадает
```

Мы начали с импорта модулей `sys` и `json`. Затем создали простой словарь, в котором есть числовые значения и список целых чисел. Мы специально добавили достаточно большое целое число (`23141`) и максимально возможное значение с плавающей запятой для текущей платформы, чтобы протестировать сериализацию и десериализацию больших чисел.

Для сериализации используется функция `json.dumps()`, которая преобразует структуру данных Python в строку в формате JSON. Полученная строка затем передается в функцию `json.loads()`, выполняющую обратную операцию — она восстанавливает структуру данных Python из строки JSON.

Следует отметить, что, помимо функций dumps и loads, модуль json также предоставляет функции dump и load, предназначенные для работы с файловыми объектами.

В последней строке примера с помощью выражения `assert` мы проверили, что данные до сериализации и после десериализации совпадают. Если условие после `assert` ложно, то будет выброшено исключение `AssertionError`. В главе 10, посвященной тестированию, мы подробнее рассмотрим утверждения.

Теперь посмотрим, как выглядит результат сериализации при выводе строки JSON.

```
# json_examples/json_basic.py
import json

info = {
    "full_name": "Sherlock Holmes",
    "address": {
        "street": "221B Baker St",
        "zip": "NW1 6XE",
        "city": "London",
        "country": "UK",
    },
}

print(json.dumps(info, indent=2, sort_keys=True))
```

Здесь создается словарь с данными о Шерлоке Холмсе. Если вы, как и мы, поклонник Шерлока и окажетесь в Лондоне, то рекомендуем заглянуть в его музей, расположенный на Бейкер-стрит, 221б, — он небольшой, но очень уютный.

Обратите внимание, как вызывается функция `json.dumps()`: мы явно задаем отступ в два пробела и просим отсортировать ключи по алфавиту.

```
$ python json_basic.py
{
  "address": {
    "city": "London",
    "country": "UK",
    "street": "221B Baker St",
    "zip": "NW1 6XE"
  },
  "full_name": "Sherlock Holmes"
}
```

Результат легко читается и по стилю напоминает структуру Python. Единственное отличие в синтаксисе — в JSON нельзя оставлять запятую после последнего элемента в словаре, как это часто делают в Python.

Теперь посмотрите на этот пример:

```
# json_examples/json_tuple.py
import json

data_in = {
    "a_tuple": (1, 2, 3, 4, 5),
}

json_data = json.dumps(data_in)
print(json_data) # {"a_tuple": [1, 2, 3, 4, 5]}
data_out = json.loads(json_data)
print(data_out) # {'a_tuple': [1, 2, 3, 4, 5]}
```

Он очень интересный. Здесь используется кортеж вместо списка. Кортеж в Python является неизменяемой структурой, но, по сути, тоже представляет собой упорядоченный набор элементов. Однако JSON не знает о типе `tuple`, и при сериализации кортеж преобразуется в обычный список. Это видно по выводу первой команды `print()` — в JSON кортеж становится списком. Следовательно, при десериализации мы получаем не кортеж, а список. Об этом важно помнить при работе с данными: использование форматов обмена данными, поддерживающих ограниченное множество структур (в данном случае JSON), может привести к потере информации, например, о типе структуры. Если вам важно сохранить тип, как в случае с кортежем, то стоит быть особенно внимательными.

Это действительно распространенная проблема. Не все объекты Python можно сериализовать в JSON, поскольку не всегда ясно, как JSON должен интерпретировать и восстанавливать такой объект. Возьмем, например, тип `datetime`. Этот объект является полноценным экземпляром класса Python, и JSON не знает, как его сериализовать. Мы можем вручную преобразовать дату и время в строку, например: `2018-03-04T12:00:30Z` — это стандарт ISO 8601, содержащий и дату, и время, и информацию о временной зоне. Но возникает вопрос: *должен ли JSON при десериализации интерпретировать такую строку как дату и время*? Или это просто строка и никакой дальнейшей интерпретации не нужно? Сложность возрастает, когда данные могут трактоваться по-разному.

Именно поэтому при обмене данными через JSON часто требуется предварительно упростить структуру объектов. Чем проще данные, тем легче представить их в формате JSON, с его ограниченным набором типов.

Тем не менее бывают ситуации, особенно во внутренней логике программ, когда нужно сериализовать нестандартные объекты. И, чтобы показать, как это можно сделать, в книге приведены два примера: с комплексными числами и с объектами `datetime`.

Пользовательская кодировка и декодировка в JSON

В контексте JSON термины «кодирование» и «декодирование» можно считать синонимами «сериализации» и «десериализации». Они обозначают преобразование данных в формат JSON и обратно.

В примере далее показано, как сериализовать комплексные числа — по умолчанию JSON не поддерживает их напрямую. Для этого создается собственный кодировщик.

```
# json_examples/json_cplx.py
import json

class ComplexEncoder(json.JSONEncoder):
    def default(self, obj):
        print(f"ComplexEncoder.default: {obj=}")
        if isinstance(obj, complex):
            return {
                "_meta": "complex",
                "num": [obj.real, obj.imag],
            }
        return super().default(obj)

data = {
    "an_int": 42,
    "a_float": 3.14159265,
    "a_complex": 3 + 4j,
}

json_data = json.dumps(data, cls=ComplexEncoder)
print(json_data)

def object_hook(obj):
    print(f"object_hook: {obj=}")
    try:
        if obj["_meta"] == "complex":
            return complex(*obj["num"])
    except KeyError:
        return obj

data_out = json.loads(json_data, object_hook=object_hook)
print(data_out)
```

Сначала определяется класс `ComplexEncoder`, производный от `JSONEncoder`. В нем переопределяется метод `default()`, который вызывается всякий раз, когда сериализатор сталкивается с объектом, не поддерживаемым по умолчанию. Этот метод должен вернуть представление объекта, которое JSON сможет сериализовать.

Если аргументом оказывается комплексное число, то метод возвращает словарь со специальной метаинформацией и списком, содержащим действительную и мнимую часть. Этого достаточно, чтобы сохранить всю информацию о числе.

Если переданный объект не комплексное число, то вызывается метод `default()` базового класса.

В завершение вызывается `json.dumps()` с указанием параметра `cls`, ссылающегося на пользовательский кодировщик. Результат сериализации выводится на экран:

```
$ python json_cplx.py
ComplexEncoder.default: obj=(3+4j)
{
    "an_int": 42, "a_float": 3.14159265,
    "a_complex": {"_meta": "complex", "num": [3.0, 4.0]}
}
```

Половина дела сделана — вы научились сериализовать комплексные числа. Теперь пора заняться десериализацией. Вместо того чтобы создавать отдельный класс, производный от `JSONDecoder`, можно использовать более простой и лаконичный способ — функцию `object_hook()`.

Функция используется при декодировании JSON и вызывается для каждого словаря, обнаруженного в процессе чтения. Внутри нее мы используем блок `try/except`. Это необходимо потому, что функция вызывается для всех словарей — и тех, которые представляют наши комплексные числа, и любых других. Если переданный словарь содержит ключ `_meta` со значением `"complex"`, то мы понимаем, что речь идет о ранее сериализованном комплексном числе, и используем функцию `complex()`, чтобы восстановить его из списка, содержащего действительную и мнимую часть. В противном случае возвращается исходный словарь без изменений.

```
object_hook:
  obj={'_meta': 'complex', 'num': [3.0, 4.0]}
object_hook:
  obj={'an_int': 42, 'a_float': 3.14159265, 'a_complex': (3+4j)}
{'an_int': 42, 'a_float': 3.14159265, 'a_complex': (3+4j)}
```

Результатом десериализации становится переменная `a_complex`, в которую корректно восстановлено исходное комплексное число. В качестве упражнения предлагаем написать собственные кодировщики и декодировщики для объектов `Fraction` и `Decimal`.

Теперь рассмотрим более сложный пример — работу с объектами `datetime`. Для наглядности разобьем код на две части — сначала сериализация, затем десериализация:

```
# json_examples/json_datetime.py
import json
from datetime import datetime, timedelta, timezone

now = datetime.now()
now_tz = datetime.now(tz=timezone(timedelta(hours=1)))
```

```
class DatetimeEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime):
            try:
                off = obj.utcoffset().seconds
            except AttributeError:
                off = None

            return {
                "_meta": "datetime",
                "data": obj.timetuple()[:6] + (obj.microsecond,),
                "utcoffset": off,
            }
        return super().default(obj)

data = {
    "an_int": 42,
    "a_float": 3.14159265,
    "a_datetime": now,
    "a_datetime_tz": now_tz,
}

json_data = json.dumps(data, cls=DatetimeEncoder)
print(json_data)
```

Этот пример более сложный из-за особенностей объектов `datetime` в Python: они могут содержать информацию о часовом поясе, а могут быть и без нее. Поэтому при сериализации важно учитывать оба варианта. Алгоритм остается тем же, что и в случае с комплексными числами, но теперь мы имеем дело с другим типом данных. Мы получаем текущее время в двух вариантах: без часового пояса (`now`) и с часовым поясом (`now_tz`). Затем определяем пользовательский кодировщик, как и раньше, переопределяя метод `default()`. В этом методе ключевой момент — извлечение смещения часового пояса в секундах (`off`). Возвращаемый словарь содержит *метаинформацию* (`_meta: "datetime"`) и список данных (`data`), в который входят шесть элементов из `timetuple()` (год, месяц, день, час, минута, секунда), а также `microsecond` и смещение. Обратите внимание: значение `"data"` представляет собой объединение двух кортежей.

После определения кодировщика мы создаем словарь с данными и сериализуем его. Результат выполнения команды `print()` выглядит следующим образом (мы слегка отформатировали его, чтобы он стал более удобным для чтения):

```
$ python json_datetime.py
{
    "an_int": 42,
    "a_float": 3.14159265,
    "a_datetime": {
        "_meta": "datetime",
        "data": [2024, 3, 29, 23, 24, 22, 232302],
        "utcoffset": null,
    },
```

```
    "a_datetime_tz": {
        "_meta": "datetime",
        "data": [2024, 3, 30, 0, 24, 22, 232316],
        "utcoffset": 3600,
    },
}
```

При выводе результата видно, что `None` преобразован в `null` — эквивалент в JavaScript/JSON, а сами объекты `datetime` закодированы по заданной структуре. Все выглядит корректно.

Теперь переходим ко второй части — десериализации:

```
# json_examples/json_datetime.py
def object_hook(obj):
    try:
        if obj["_meta"] == "datetime":
            if obj["utcoffset"] is None:
                tz = None
            else:
                tz = timezone(timedelta(seconds=obj["utcoffset"]))
            return datetime(*obj["data"], tzinfo=tz)
    except KeyError:
        return obj

data_out = json.loads(json_data, object_hook=object_hook)
print(data_out)
```

Сначала, как и ранее, мы проверяем, указывает ли метка `_meta` на объект `datetime`. Если да, то извлекаем информацию о часовом поясе. После этого с помощью распаковки (`*`) передаем значения из кортежа `data` в конструктор `datetime()`, добавляя информацию о смещении. В результате получаем исходный объект `datetime`. Чтобы убедиться в этом, выводим значение `data_out`:

```
{
    "an_int": 42,
    "a_float": 3.14159265,
    "a_datetime": datetime.datetime(
        2024, 3, 29, 23, 24, 22, 232302
    ),
    "a_datetime_tz": datetime.datetime(
        2024, 3, 30, 0, 24, 22, 232316,
        tzinfo=datetime.timezone(
            datetime.timedelta(seconds=3600)
        ),
    ),
}
```

И видим, что восстановленные объекты точно соответствуют исходным. В качестве упражнения можно реализовать аналогичную логику для объектов `date`. Это

будет сделать немного проще, поскольку такие объекты не содержат информации о времени и часовом поясе.

И напоследок — важное предостережение. Несмотря на кажущуюся простоту, работа с `datetime` может оказаться непростой. Мы подчеркиваем, что приведенный пример был проверен лишь поверхностно. Поэтому если вы собираетесь использовать такую реализацию, то обязательно проведите тщательное тестирование: с разными часовыми поясами, в условиях перехода на летнее/зимнее время, с датами до начала эпохи UNIX и т. п. Возможно, код придется немного адаптировать под ваш конкретный сценарий.

Ввод/вывод, потоки и запросы

I/O — аббревиатура от *input/output* («ввод/вывод»), и это понятие охватывает все виды взаимодействия между компьютером и внешним миром. Существует множество форм ввода-вывода, и в рамках этой главы не ставится задача рассмотреть их все. Тем не менее стоит разобрать пару примеров, чтобы вы могли получить общее представление. В первом примере вы познакомитесь с классом `io.StringIO`, представляющим собой поток ввода-вывода для текстовых данных, размещенный в памяти. Во втором, напротив, выйдете за пределы локального компьютера и увидите, как выполнить HTTP-запрос.

Управление объектами в памяти

Управление объектами в памяти может оказаться полезным во многих ситуациях. Операции с оперативной памятью выполняются значительно быстрее, чем с жестким диском; память всегда доступна, и для небольших объемов данных это может быть оптимальным решением. Рассмотрим пример:

```
# io_examples/string_io.py
import io

stream = io.StringIO()
stream.write("Learning Python Programming.\n")
print("Become a Python ninja!", file=stream)

contents = stream.getvalue()
print(contents)

stream.close()
```

Здесь мы импортируем модуль `io` из стандартной библиотеки. Он содержит множество инструментов, связанных с потоками и вводом-выводом. Один из них — класс `StringIO`, представляющий собой буфер в памяти. Мы записываем в него две строки, причем используем два разных способа, так же как и при работе с файлами в первых примерах текущей главы.

Класс `StringIO` особенно полезен, когда требуется:

- симулировать поведение файла для строк;
- тестировать код, работающий с файловыми объектами, не создавая реальные файлы;
- эффективно собирать или обрабатывать большие строки;
- перехватывать или подменять ввод/вывод в тестах — тесты при этом выполняются быстрее, поскольку избегают обращения к диску.

Можно вызывать метод `write()`, а можно использовать функцию `print()` и направлять ее вывод в поток.

Чтобы получить все содержимое буфера, нужно вызвать `getvalue()`. Затем можно вывести результат, а по завершении работы — закрыть поток, вызвав `close()`. При этом буфер в памяти будет уничтожен.

Есть более элегантный способ написать этот код — воспользоваться контекстным менеджером:

```
# io_examples/string_io.py
with io.StringIO() as stream:
    stream.write("Learning Python Programming.\n")
    print("Become a Python ninja!", file=stream)

    contents = stream.getvalue()
    print(contents)
```

`io.StringIO()`, как и встроенная функция `open()`, отлично работает в конструкции `with`, и при этом поток будет закрыт автоматически. Результат выполнения кода выглядит так:

```
$ python string_io.py
Learning Python Programming.
Become a Python ninja!
```

Теперь перейдем ко второму примеру.

Выполнение HTTP-запросов

Здесь мы рассмотрим два примера выполнения HTTP-запросов. Для этого используется библиотека `requests`, которую можно установить с помощью `pip`. Она также указана в файле зависимостей для этой главы.

В качестве сервера мы используем API сайта httpbin.org (https://httpbin.org) — это простой сервис, предназначенный для тестирования HTTP-запросов и ответов. Интересный факт: httpbin был создан Кеннетом Рейцем, автором самой библиотеки `requests`. Эта библиотека — одна из самых популярных и широко используемых в Python-сообществе:

```
# io_examples/reqs.py
import requests

urls = {
    "get": "https://httpbin.org/get?t=learn+python+programming",
    "headers": "https://httpbin.org/headers",
    "ip": "https://httpbin.org/ip",
    "user-agent": "https://httpbin.org/user-agent",
    "UUID": "https://httpbin.org/uuid",
    "JSON": "https://httpbin.org/json",
}

def get_content(title, url):
    resp = requests.get(url)
    print(f"Response for {title}")
    print(resp.json())

for title, url in urls.items():
    get_content(title, url)
    print("-" * 40)
```

В этом коде все устроено достаточно просто. Мы объявляем словарь с URL, к которым хотим выполнить HTTP-запросы. Логика выполнения запроса вынесена в отдельную функцию `get_content()`. Внутри нее выполняется запрос методом `GET` (с помощью `requests.get()`), после чего мы выводим заголовок и результат декодирования тела ответа из формата JSON. О последнем действии стоит сказать подробнее.

Отправляя запрос на сайт или к API, мы в ответ получаем объект `response`, который содержит данные, возвращенные сервером. Тело некоторых ответов от httpbin.org закодировано в формате JSON. Вместо того чтобы вручную читать его с помощью `resp.text` и потом отдельно вызывать `json.loads()`, мы можем воспользоваться методом `json()` объекта ответа. Он объединяет оба действия — извлечение тела и его декодирование — в один шаг. Такой удобный подход — одна из причин, по которым библиотека `requests` получила столь широкое распространение.

Разумеется, в реальных приложениях для обработки запросов потребуется более надежная стратегия: например, учет ошибок, тайм-аутов, повторных попыток. Однако для целей этой главы простого примера вполне достаточно. В главе 14, посвященной основам разработки API, мы рассмотрим больше примеров работы с запросами.

В завершение, как видно из кода, запускается цикл, в котором перебираются все URL. При выполнении скрипта вы увидите в консоли результат каждого запроса (в книге приведен отформатированный и сокращенный пример):

```
$ python reqs.py
Response for get
{
    "args": {"t": "learn python programming"},
```

```
    "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip, deflate",
        "Host": "httpbin.org",
        "User-Agent": "python-requests/2.31.0",
        "X-Amzn-Trace-Id": "Root=1-123abc-123abc"
    },
    "origin": "86.14.44.233",
    "url": "https://httpbin.org/get?t=learn+python+programming"
}
... остальная часть вывода опущена ...
```

Обратите внимание, что вывод может немного различаться — например, номера версий или IP-адреса будут другими, и это абсолютно нормально. Метод `GET` является лишь одним из множества HTTP-методов, хотя и используется наиболее часто. Теперь рассмотрим, как работает метод `POST`. Он применяется, когда необходимо отправить данные на сервер, например, чтобы создать новый ресурс. Каждый раз, отправляя форму на сайте, вы выполняете `POST`-запрос. Итак, попробуем сформировать такой запрос программно:

```
# io_examples/reqs_post.py
import requests

url = "https://httpbin.org/post"
data = dict(title="Learn Python Programming")

resp = requests.post(url, data=data)
print("Response for POST")
print(resp.json())
```

Пример очень похож на предыдущий: теперь мы вызываем функцию `post()`, а не `get()` и передаем данные в теле запроса. В теле ответа, в том числе в разделе `form`, можно увидеть отправленные данные в виде пар «ключ — значение». Библиотека `requests` предлагает гораздо больше возможностей — мы настоятельно рекомендуем внимательно изучить ее. Вероятность того, что вы будете работать с ней в реальных проектах, довольно высока.

Запуск приведенного выше кода дает следующий результат (здесь вывод слегка отформатирован ради удобства чтения):

```
$ python reqs_post.py
Response for POST
{
    "args": {},
    "data": "",
    "files": {},
    "form": {"title": "Learn Python Programming"},
    "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip, deflate",
        "Content-Length": "30",
```

```
        "Content-Type": "application/x-www-form-urlencoded",
        "Host": "httpbin.org",
        "User-Agent": "python-requests/2.31.0",
        "X-Amzn-Trace-Id": "Root=1-123abc-123abc"
    },
    "json": None,
    "origin": "86.14.44.233",
    "url": "https://httpbin.org/post"
}
```

Обратите внимание, что заголовки изменились, а отправленные нами данные видны в теле ответа, в виде пары «ключ — значение» в поле `form`.

Надеемся, что этих простых примеров достаточно, чтобы вы могли начать осваивать эту тему. Интернет постоянно меняется, и полезно время от времени освежать знания об основных принципах работы с HTTP-запросами.

Сохранение данных на диске

В этом разделе мы рассмотрим три способа сохранения данных на диске. Под сохранением подразумевается запись данных в энергонезависимую память — например, на жесткий диск — так, чтобы они не исчезли после завершения работы программы. Вы познакомитесь с модулями `pickle` и `shelve`, а также с кратким примером доступа к базе данных с помощью *SQLAlchemy* — одной из самых популярных ORM-библиотек в экосистеме Python.

Сериализация данных с помощью модуля pickle

Модуль `pickle` из стандартной библиотеки Python предоставляет инструменты для преобразования объектов Python в поток байтов и обратно. Интерфейсы `pickle` и `json` частично пересекаются, однако это два совершенно разных подхода. Как мы уже видели ранее в этой главе, JSON — текстовый формат, который легко читается человеком, не зависит от языка программирования и поддерживает лишь ограниченное множество типов данных. В отличие от него `pickle` создает двоичное представление, привязан к Python и благодаря встроенной в язык поддержке интроспекции способен сериализовать гораздо больше типов.

Помимо функциональных различий между `pickle` и `json`, важно учитывать и вопросы безопасности. *Распаковка* (`unpickle`) ошибочных или вредоносных данных из ненадежного источника может быть опасной. Поэтому если вы решите использовать `pickle` в приложении, то проявляйте особую осторожность.

Если вы все же выбрали pickle, то вам стоит подумать о криптографической подписи, позволяющей убедиться, что сериализованные данные не были подменены. Как создавать такие подписи, мы расскажем в главе 9.

А теперь взглянем на простой пример работы с `pickle`:

```
# persistence/pickler.py
import pickle
from dataclasses import dataclass

@dataclass
class Person:
    first_name: str
    last_name: str
    id: int

    def greet(self):
        print(
            f"Hi, I am {self.first_name} {self.last_name}"
            f" and my ID is {self.id}"
        )

people = [
    Person("Obi-Wan", "Kenobi", 123),
    Person("Anakin", "Skywalker", 456),
]

# сохраняем данные в двоичном формате в файл
with open("data.pickle", "wb") as stream:
    pickle.dump(people, stream)

# загружаем данные из файла
with open("data.pickle", "rb") as stream:
    peeps = pickle.load(stream)

for person in peeps:
    person.greet()
```

Здесь мы создаем класс `Person`, используя декоратор `@dataclass`, с которым вы познакомились в главе 6. Мы выбрали именно `dataclass`, чтобы показать, насколько легко `pickle` справляется с сериализацией таких объектов — не требуется никаких дополнительных действий по сравнению с простыми типами данных.

Класс `Person` содержит три атрибута: `first_name`, `last_name` и `id`. Кроме того, он предоставляет метод `greet()`, выводящий приветствие с данными конкретного экземпляра.

Мы создаем список экземпляров этого класса и сохраняем его в файл. Запись выполняется с помощью функции `pickle.dump()`, которой передается как содержимое, *подлежащее сериализации*, так и поток, в который нужно записать результат. Затем мы читаем данные из того же файла, используя `pickle.load()` для восстановления объектов из потока. Чтобы убедиться в корректном восстановлении объектов, мы вызываем метод `greet()` у каждого из них. Результат выполнения будет примерно таким:

```
$ python pickler.py
Hi, I am Obi-Wan Kenobi and my ID is 123
Hi, I am Anakin Skywalker and my ID is 456
```

Модуль `pickle` также позволяет сериализовать объекты Python в байтовое представление и обратно с помощью функций `dumps()` и `loads()` (обратите внимание на окончание *s* в их именах). В повседневной практике `pickle` обычно используют для сохранения данных, которые не предполагается передавать в другие приложения. Например, однажды нам встретился плагин к Flask, в котором объект сессии сериализовался `pickle` перед тем, как сохранялся в базу Redis. На практике вы вряд ли будете часто использовать этот модуль.

Еще один инструмент, который применяется даже реже, чем модуль `pickle`, но может быть полезен при ограниченных ресурсах, — это модуль `shelve`.

Сохранение данных с помощью модуля shelve

«Полка» (shelf) в Python — это объект, который ведет себя как словарь, но при этом сохраняет данные на диске. Преимущество модуля `shelve` в том, что значения, сохраняемые на такой «полке», могут быть любыми объектами, которые можно сериализовать с помощью `pickle`. Это дает большую гибкость по сравнению, например, с классическими базами, где структура данных должна быть заранее определена. Несмотря на это, модуль `shelve` на практике используется довольно редко. Тем не менее ради полноты картины разберем, как он работает:

```
# persistence/shelf.py

import shelve

class Person:
    def __init__(self, name, id):
        self.name = name
        self.id = id

with shelve.open("shelf1.shelve") as db:
    db["obi1"] = Person("Obi-Wan", 123)
    db["ani"] = Person("Anakin", 456)
    db["a_list"] = [2, 3, 5]
    db["delete_me"] = "we will have to delete this one..."
    print(
        list(db.keys())
    )                          # ['ani', 'delete_me', 'a_list', 'obi1']

    del db["delete_me"]        # удалена!
    print(list(db.keys()))     # ['ani', 'a_list', 'obi1']
    print("delete_me" in db)   # False
    print("ani" in db)         # True

    a_list = db["a_list"]
    a_list.append(7)
    db["a_list"] = a_list
    print(db["a_list"])        # [2, 3, 5, 7]
```

За вычетом технических деталей этот пример очень напоминает обычную работу со словарями. Мы создаем класс `Person`, затем открываем файл `shelve` внутри блока контекстного менеджера. Как вы могли заметить, с помощью синтаксиса словаря мы сохраняем четыре объекта: два экземпляра `Person`, список и строку. При выводе ключей отображаются четыре использованных имени. Сразу после этого мы удаляем пару «ключ — значение» с ключом `delete_me`, а при повторном выводе ключей видим, что элемент действительно удален. Затем проверяем наличие некоторых ключей и изменяем список, добавляя в него число 7. Обратите внимание, что для этого нужно сначала извлечь список из модуля, изменить и снова записать по тому же ключу — `shelve` не отслеживает изменения объектов по ссылке.

Существует и другой способ воспользоваться «полкой», немного ускоряющий работу:

```
# persistence/shelf.py
with shelve.open("shelf2.shelve", writeback=True) as db:
    db["a_list"] = [11, 13, 17]
    db["a_list"].append(19)  # добавление по месту!
    print(db["a_list"])      # [11, 13, 17, 19]
```

Если открыть «полку» с параметром `writeback=True`, то можно вносить изменения во вложенные объекты — например, добавлять элементы в список `a_list` — так, будто вы работаете с обычным словарем. Однако эта возможность не встроена по умолчанию: за нее приходится расплачиваться увеличенным потреблением памяти и более медленным закрытием «полки».

Теперь, когда мы отдали должное стандартным модулям Python для сохранения данных, пора взглянуть на один из самых популярных ORM-инструментов в экосистеме Python — SQLAlchemy.

Сохранение данных в базу данных

В этом примере мы будем использовать базу данных, хранящуюся в оперативной памяти. Такой подход удобен для демонстрационных целей, поскольку не требует создания файлов на диске. В сопроводительном коде к книге оставлены комментарии, показывающие, как работать с SQLite-файлом. Надеемся, вы попробуете и этот вариант.

Если вы решите поэкспериментировать с SQLite, то можете воспользоваться бесплатной программой DBeaver — она доступна по адресу https://dbeaver.io. DBeaver — это кросс-платформенная утилита для работы с базами данных, предназначенная для разработчиков, администраторов, аналитиков и всех, кто взаимодействует с подобными структурами. DBeaver поддерживает множество систем: MySQL, PostgreSQL, SQLite, Oracle, DB2, SQL Server, Sybase, MS Access, Teradata, Firebird, Apache Hive, Phoenix, Presto и др.

Прежде чем перейти к коду, кратко рассмотрим суть реляционной базы данных.

Реляционная база данных — это база, которая хранит данные согласно *реляционной модели*, предложенной в 1969 году Эдгаром Ф. Коддом. В этой модели данные хранятся в одной или нескольких таблицах. Каждая таблица состоит из строк (также называемых *записями* или *кортежами*), каждая из которых, в свою очередь, представляет отдельную сущность. Кроме того, таблицы имеют столбцы (или *атрибуты*), соответствующие характеристикам записей. Каждая запись идентифицируется с помощью уникального значения — *первичного ключа*, который содержит один или несколько столбцов.

В качестве примера представим таблицу со списком пользователей со столбцами: `id`, `username`, `password`, `name` и `surname`. Такая таблица может хранить данные о пользователях системы: каждая строка соответствует одному пользователю. Строка со значениями `3`, `fab`, `my_wonderful_pwd`, `Fabrizio` и `Romano` будет означать, что это учетная запись пользователя по имени Фабрицио.

Модель называется *реляционной*, поскольку таблицы в ней могут быть связаны между собой. Например, добавив таблицу `PhoneNumbers`, вы сможете хранить в ней номера телефонов и с помощью связи указать, какой номер принадлежит тому или иному пользователю.

Для работы с реляционными базами данных используется специальный язык запросов — *SQL* (Structured Query Language, язык структурированных запросов). Он основан на *реляционной алгебре* — формальной системе для описания операций над данными. С помощью SQL можно, в частности, фильтровать строки и столбцы, объединять таблицы, выполнять агрегацию по заданным критериям и многое другое.

Представим такой запрос на естественном языке: *«Найти всех пользователей (username, name, surname), имя пользователя которых начинается с буквы m и которые имеют не более одного телефонного номера»*. В этом примере мы выбираем подмножество строк таблицы пользователей, извлекая только три столбца, и фильтруем по условию: логин должен начинаться на `m`, а количество связанных телефонных номеров — не превышать одного.

Каждая система управления базами данных (СУБД) использует свою *разновидность* языка SQL. Все они в той или иной степени следуют стандарту, но ни одна не соответствует ему полностью и различается в деталях. Это создает определенные сложности при разработке современных приложений. Если в коде программы используются SQL-запросы напрямую, то при переходе на другой движок базы данных (или даже на другую версию того же движка) такой код часто требует корректировки.

Это может быть довольно утомительно, особенно если SQL-запросы сложные. Чтобы упростить работу, были разработаны инструменты, связывающие объекты языка программирования с таблицами реляционной базы данных. Такие

инструменты получили название *«объектно-реляционное отображение»* (Object-Relational Mapping, ORM).

В настоящее время создание приложений чаще всего начинается с работы через ORM. Если оказывается, что какую-то задачу невозможно выразить его средствами, тогда — и только тогда — прибегают к прямому написанию SQL-запросов. Такой подход обеспечивает разумный баланс между двумя крайностями: полным отказом от SQL и полным отказом от ORM, каждая из которых имеет свои недостатки.

В этом подразделе мы рассмотрим пример с использованием SQLAlchemy — одной из самых популярных ORM-библиотек на Python. Ее нужно установить в виртуальное окружение, созданное для текущей главы. Мы создадим две модели — `Person` и `Email`, каждая из которых будет соответствовать отдельной таблице. Затем добавим в базу данных информацию и выполним несколько запросов.

Начнем с объявления моделей:

```
# persistence/alchemy_models.py
from sqlalchemy import ForeignKey, String, Integer
from sqlalchemy.orm import (
    DeclarativeBase,
    mapped_column,
    relationship,
)
```

Сначала мы импортируем необходимые функции и типы. Затем определяем классы `Person` и `Email`, а также обязательный базовый класс для них. В коде это выглядит так:

```
# persistence/alchemy_models.py
class Base(DeclarativeBase):
    pass

class Person(Base):
    __tablename__ = "person"

    id = mapped_column(Integer, primary_key=True)
    name = mapped_column(String)
    age = mapped_column(Integer)
    emails = relationship(
        "Email",
        back_populates="person",
        order_by="Email.email",
        cascade="all, delete-orphan",
    )

    def __repr__(self):
        return f"{self.name}(id={self.id})"

class Email(Base):
    __tablename__ = "email"
```

```
    id = mapped_column(Integer, primary_key=True)
    email = mapped_column(String)
    person_id = mapped_column(ForeignKey("person.id"))
    person = relationship("Person", back_populates="emails")

    def __str__(self):
        return self.email
    __repr__ = __str__
```

Каждая модель наследует базу `Base`, которая в примере создается на основе класса `DeclarativeBase` из SQLAlchemy. Класс `Person` соответствует таблице с именем `"person"` и описывает три атрибута: `id`, `name` и `age`. Кроме того, в модели указывается связь с моделью `Email` — мы определяем атрибут `emails`, обращение к которому вернет все строки таблицы `Email`, связанные с конкретным экземпляром `Person`. Параметр `cascade` управляет поведением при создании и удалении связанных объектов, но это более сложная тема, и ее можно пока отложить — при желании вы всегда сможете изучить ее позже.

В завершение мы переопределяем метод `__repr__()`, который отвечает за строковое представление объекта. В идеале этот метод должен возвращать такую строку, по которой объект можно воссоздать. В нашем примере мы используем его скорее для того, чтобы получить осмысленный вывод в консоли. В Python вызов `repr(obj)` эквивалентен `obj.__repr__()`.

Кроме того, мы объявляем модель `Email`, которая соответствует таблице `"email"` и будет хранить адреса электронной почты, а также ссылку на человека, которому они принадлежат. Атрибуты `person_id` и `person` задают связь между классами `Email` и `Person`. Обратите внимание: в модели `Email` мы реализуем метод `__str__()`, а затем присваиваем его как `__repr__ = __str__`. Благодаря этому вызов `repr()` или `str()` для объектов `Email` приведет к выполнению одного и того же метода. В Python это распространенная практика — так можно избежать дублирования кода. Мы воспользовались случаем, чтобы продемонстрировать вам этот прием.

Более глубокий анализ приведенного кода потребовал бы уделить ему больше места, чем позволяет книга, поэтому мы рекомендуем изучить дополнительные материалы по системам управления базами данных (СУБД), языку SQL, реляционной алгебре и библиотеке SQLAlchemy.

Теперь, когда модели готовы, перейдем к тому, как с их помощью сохранять данные.

В следующем примере (пока не будет сказано иное, все фрагменты кода взяты из файла `alchemy.py` в папке `persistence`) показан этот процесс:

```
# persistence/alchemy.py

from sqlalchemy import create_engine, select, func
from sqlalchemy.orm import Session
from alchemy_models import Person, Email, Base
```

```
# если хотите работать с реальным файлом базы данных,
# раскомментируйте следующую строку
# engine = create_engine('sqlite:///example.db')
engine = create_engine("sqlite:///:memory:")  # а эту — закомментируйте
Base.metadata.create_all(engine)
```

Сначала мы импортируем необходимые функции и классы. Затем создаем объект движка (`engine`) для приложения и просим SQLAlchemy создать с его помощью все таблицы.

Функция create_engine() принимает параметр echo, который можно установить в значение True, False или строку "debug" — это определяет уровень вывода логов о выполняемых SQL-операциях и значениях переданных параметров. Подробности описаны в официальной документации SQLAlchemy.

В SQLAlchemy *движок* — это ключевой компонент, служащий основным интерфейсом между Python-приложением и базой данных. Он управляет двумя важными аспектами: соединениями с БД и выполнением SQL-запросов.

После импорта и создания движка и таблиц мы настраиваем объект сессии в контекстном менеджере, используя движок, который создали только что. Внутри этого блока создаем два объекта `Person`:

```
with Session(engine) as session:
    anakin = Person(name="Anakin Skywalker", age=32)
    obione = Person(name="Obi-Wan Kenobi", age=40)
```

Затем добавляем адреса электронной почты каждому человеку двумя способами: в одном случае — используя присваивание списка, в другом — метод `append()`:

```
    obione.emails = [
        Email(email="obi1@example.com"),
        Email(email="wanwan@example.com"),
    ]

    anakin.emails.append(Email(email="ani@example.com"))
    anakin.emails.append(Email(email="evil.dart@example.com"))
    anakin.emails.append(Email(email="vader@example.com"))
```

К этому моменту мы еще не взаимодействовали с базой данных. Она остается неизменной до тех пор, пока мы не используем объект `session`:

```
    session.add(anakin)
    session.add(obione)
    session.commit()
```

Добавление двух экземпляров `Person` автоматически приводит к добавлению связанных адресов электронной почты — благодаря параметру каскадирования. Вызов метода `commit()` приводит к фиксации транзакции и сохранению данных в базе.

Транзакция — это своего рода песочница в контексте базы данных. Пока транзакция не зафиксирована, можно откатить любые изменения и вернуться к предыдущему состоянию базы. В SQLAlchemy есть более сложные средства управления транзакциями — вы найдете их в официальной документации, поскольку эта тема требует отдельного изучения.

Затем выполняется запрос, позволяющий найти всех пользователей, имя которых начинается на `Obi`, с помощью метода `like()`, соответствующего оператору `LIKE` в SQL:

```
    obione = session.scalar(
        select(Person).where(Person.name.like("Obi%"))
    )
    print(obione, obione.emails)
```

Из результатов берется первый элемент (мы знаем, что это будет Obi-Wan) и выводится на экран. После этого мы извлекаем объект `anakin`, используя точное совпадение имени — просто чтобы продемонстрировать другой способ фильтрации:

```
    anakin = session.scalar(
        select(Person).where(Person.name == "Anakin Skywalker")
    )
    print(anakin, anakin.emails)
```

Далее сохраняем идентификатор Энакина в переменную и удаляем объект `anakin` из глобальной области видимости. Это не удаляет запись из базы данных — удаляется лишь объект в памяти:

```
    anakin_id = anakin.id
    del anakin
```

Мы удаляем объект `anakin`, чтобы показать, как получить объект по его идентификатору. Вывести содержимое базы данных призвана определенная в модуле функция `display_info()`. Она сначала извлекает адреса электронной почты, а затем объекты `Person`, связанные с ними через отношение. Кроме того, она выводит количество объектов каждой модели. Эта функция определена в модуле до входа в контекстный менеджер, предоставляющий сессию:

```
def display_info(session):
    # сначала получаем все адреса электронной почты
    emails = select(Email)

    # выводим результаты
    print("All emails:")
    for email in session.scalars(emails):
        print(f" - {email.person.name} <{email.email}>")

    # выводим общее количество объектов
    people = session.scalar(
        select(func.count()).select_from(Person)
    )
```

```
    emails = session.scalar(
        select(func.count()).select_from(Email)
    )

    print("Summary:")
    print(f" {people=}, {emails=}")
```

Мы вызываем эту функцию, затем получаем и удаляем `anakin`, а после этого снова вызываем `display_info()` — чтобы убедиться, что запись действительно удалена из базы:

```
    display_info(session)

    anakin = session.get(Person, anakin_id)
    session.delete(anakin)
    session.commit()

    display_info(session)
```

Результат выполнения всех фрагментов выглядит так (для удобства мы разделили его на четыре блока, которые соответствуют четырем частям кода):

```
$ python alchemy.py
Obi-Wan Kenobi(id=2) [obi1@example.com, wanwan@example.com]

Anakin Skywalker(id=1) [
    ani@example.com, evil.dart@example.com, vader@example.com
]

All emails:
 - Anakin Skywalker <ani@example.com>
 - Anakin Skywalker <evil.dart@example.com>
 - Anakin Skywalker <vader@example.com>
 - Obi-Wan Kenobi <obi1@example.com>
 - Obi-Wan Kenobi <wanwan@example.com>
Summary:
 people=2, emails=5

All emails:
 - Obi-Wan Kenobi <obi1@example.com>
 - Obi-Wan Kenobi <wanwan@example.com>
Summary:
 people=1, emails=2
```

Как видно из последних двух блоков, удаление `anakin` повлекло за собой удаление одного объекта `Person` и трех связанных адресов электронной почты. Это снова объясняется эффектом каскадного удаления.

Пока это вся информация о сохранении данных. Это обширная и порой непростая область, которую стоит изучать как можно глубже. Недостаток знаний о системах управления базами данных может привести к росту количества ошибок в приложении и снижению его производительности.

Файлы конфигурации

Файлы конфигурации играют важную роль во многих приложениях на Python. Благодаря им основной код приложения можно отделить от настроек и параметров. Такое разделение облегчает сопровождение, управление и распространение программ, особенно если одно и то же приложение должно запускаться в разных средах — например, в режиме разработки, тестирования и в готовой версии продукта — с различными конфигурациями.

Файлы конфигурации дают несколько ключевых преимуществ.

- *Гибкость* — пользователь может изменить поведение приложения, не затрагивая исходный код. Это особенно удобно, когда приложение развертывается в нескольких средах или требует подключения к базе данных, указания ключей API и других параметров.
- *Безопасность* — секретная информация, такая как учетные данные для аутентификации, ключи API или секретные токены, не должна храниться в коде и должна обрабатываться отдельно от основной кодовой базы.

Популярные форматы

Файлы конфигурации могут быть представлены в разных форматах, каждый из которых имеет собственный синтаксис и особенности. Наиболее распространенные — `INI`, `JSON`, `YAML`, `TOML` и `ENV`.

В этой короткой части мы кратко рассмотрим форматы `INI` и `TOML`. А в главе 14 будем использовать файл `ENV`.

Формат конфигурационных файлов INI

`INI` — простой текстовый файл, разделенный на секции. Каждая из них содержит свойства в виде пар «ключ — значение».

Чтобы узнать больше об этом формате, можно обратиться к статье в Википедии по адресу https://ru.wikipedia.org/wiki/INI.

Рассмотрим пример конфигурационного файла в формате `INI`:

```
# config_files/config.ini
[owner]
name = Fabrizio Romano
dob = 1975-12-29T11:50:00Z

[DEFAULT]
title = Config INI example
host = 192.168.1.1

[database]
host = 192.168.1.255
user = redis
password = redis-password
db_range = [0, 32]
```

```
[database.primary]
port = 6379
connection_max = 5000

[database.secondary]
port = 6380
connection_max = 4000
```

Здесь несколько секций посвящены настройкам подключения к базе данных. Общие свойства содержатся в секции `[database]`, а специфичные — в секциях `.primary` и `.secondary`, которые представляют конфигурации для подключения к основной базе данных (primary) и реплике (secondary) соответственно[1]. Кроме того, присутствуют секции `[owner]` и `[DEFAULT]`.

Читать такую конфигурацию в приложении удобно с помощью модуля `configparser` из стандартной библиотеки Python (https://docs.python.org/3/library/configparser.html). Он возвращает объект, похожий на словарь, предоставляя дополнительное преимущество: значения из секции `[DEFAULT]` автоматически доступны во всех других секциях.

Ниже показан пример интерактивной сессии в Python, демонстрирующий работу с этим модулем:

```
# config_files/config-ini.txt
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config.read("config.ini")
['config.ini']
>>> config.sections()
['owner', 'database', 'database.primary', 'database.secondary']
>>> config.items("database")
[
    ('title', 'Config INI example'), ('host', '192.168.1.255'),
    ('user', 'redis'), ('password', 'redis-password'),
    ('db_range', '[0, 32]')
]
>>> config["database"]
<Section: database>
>>> dict(config["database"])
{
    'host': '192.168.1.255', 'user': 'redis',
    'password': 'redis-password', 'db_range': '[0, 32]',
    'title': 'Config INI example'
}
>>> config["DEFAULT"]["host"]
'192.168.1.1'
>>> dict(config["database.secondary"])
```

[1] На практике часто, если база данных развернута в кластере с репликацией данных, запросы на запись отправляются в основную базу данных, в то время как запросы на чтение могут быть распределены между одной или несколькими репликами. Это разгружает основную базу данных от нагрузки на чтение. — *Примеч. науч. ред.*

```
{
    'port': '6380', 'connection_max': '4000',
    'title': 'Config INI example', 'host': '192.168.1.1'
}
>>> config.getint("database.primary", "port")
6379
```

Обратите внимание, что мы импортируем модуль `configparser` и создаем объект `config`. Этот объект предоставляет различные методы: можно получить список секций, а также любое значение из конфигурации.

Внутри `configparser` все значения хранятся в виде строк, поэтому при необходимости их следует приводить к нужным типам Python. Для этого у объекта `ConfigParser` есть специальные методы, такие как `getint()`, `getfloat()` и `getboolean()`, которые возвращают значения, преобразованные в указанный тип, но, как видите, их не слишком много.

Кроме того, стоит отметить, что свойства из секции `[DEFAULT]` автоматически добавляются во все остальные секции. При этом в случае, если в какой-то секции определен ключ с таким же именем, что и в `[DEFAULT]`, значение из конкретной секции не будет перезаписано значением из `[DEFAULT]`. В выделенном примере видно, что свойство `title` имеется в секции `[database]`, а `host`, которое есть в обеих секциях, корректно сохраняет значение `'192.168.1.255'` из секции `[database]`.

Формат конфигурационных файлов TOML

TOML — довольно популярный формат конфигурационных файлов в Python-приложениях. У него более богатый набор возможностей по сравнению с INI. Для изучения его синтаксиса можно обратиться к официальному сайту https://toml.io.

Рассмотрим короткий пример, схожий с предыдущим для INI:

```
# config_file/config.toml
title = "Config Example"

[owner]
name = "Fabrizio Romano"
dob = 1975-12-29T11:50:00Z

[database]
host = "192.168.1.255"
user = "redis"
password = "redis-password"
db_range = [0, 32]

[database.primary]
port = 6379
connection_max = 5000

[database.secondary]
port = 6380
connection_max = 4000
```

Здесь отсутствует секция DEFAULT. Свойства задаются немного иначе: строки заключены в кавычки, а числа — нет.

Для чтения такого файла мы используем модуль tomllib из стандартной библиотеки Python (https://docs.python.org/3/library/tomllib.html):

```
# config_files/config-toml.txt
>>> import tomllib
>>> with open("config.toml", "rb") as f:
...     config = tomllib.load(f)
...
>>> config
{
    'title': 'Config Example',
    'owner': {
        'name': 'Fabrizio Romano',
        'dob': datetime.datetime(
            1975, 12, 29, 11, 50, tzinfo=datetime.timezone.utc
        )
    },
    'database': {
        'host': '192.168.1.255',
        'user': 'redis',
        'password': 'redis-password',
        'db_range': [0, 32],
        'primary': {'port': 6379, 'connection_max': 5000},
        'secondary': {'port': 6380, 'connection_max': 4000}
    }
}
>>> config["title"]
'Config Example'
>>> config["owner"]
{
    'name': 'Fabrizio Romano',
    'dob': datetime.datetime(
        1975, 12, 29, 11, 50, tzinfo=datetime.timezone.utc
    )
}
>>> config["database"]["primary"]
{'port': 6379, 'connection_max': 5000}
>>> config["database"]["db_range"]
[0, 32]
```

Обратите внимание, что теперь объект config представлен в виде словаря. Благодаря тому, как мы задали секции database.primary и database.secondary, tomllib создал вложенную структуру, отражающую эту иерархию.

В формате TOML значения автоматически приводятся к соответствующим типам Python: строки, числа, списки, а также объект datetime, который создается из строки в формате ISO, в данном примере представляющей дату рождения Фабрицио. В документации по tomllib можно найти таблицу со всеми возможными преобразованиями типов.

Резюме

В этой главе мы говорили о работе с файлами и каталогами. Вы узнали, как читать и записывать файлы и как делать это элегантно — с помощью контекстных менеджеров. Кроме того, вы изучили способы вывода содержимого каталогов как с рекурсией, так и без нее. Отдельно рассмотрели пути — именно они являются шлюзами для доступа к файлам и каталогам.

Затем мы кратко разобрали, как создать ZIP-архив и распаковать его содержимое. В репозитории к книге вы также найдете пример с другим форматом сжатия — tar.gz.

Мы обсудили форматы обмена данными и подробнее остановились на JSON. Было весело поэкспериментировать с собственными кодировщиками и декодировщиками для специфичных типов данных Python.

Затем мы перешли к вводу-выводу: поработали с потоками в памяти и с HTTP-запросами.

После этого вы изучили способы сохранения данных с помощью модулей `pickle` и `shelve`, а также с использованием библиотеки ORM SQLAlchemy.

Наконец, мы рассмотрели два формата конфигурационных файлов — INI и TOML — и увидели, как работать с ними в Python.

Теперь у вас есть хорошее представление о том, как работать с файлами и сохранять данные, и мы надеемся, что вы самостоятельно продолжите изучать эти темы.

В следующей главе речь пойдет о криптографии и токенах.

9 Криптография и токены

> Трое могут сохранить тайну, только если двое из них мертвы.
>
> *Бенджамин Франклин. Poor Richard's Almanack*

В этой небольшой главе мы кратко рассмотрим криптографические возможности, предоставляемые стандартной библиотекой Python. Кроме того, мы затронем JSON Web Tokens (JWT) — стандарт, используемый для безопасного обмена утверждениями между двумя сторонами.

Начнем с криптографии и поговорим о том, почему она играет такую важную роль.

Зачем нужна криптография

По разным оценкам, в 2024 году количество пользователей Интернета в мире составляло от 5,35 до 5,44 миллиарда человек. С каждым годом все больше людей пользуются онлайн-банкингом, совершают покупки через Всемирную сеть или просто общаются с друзьями и семьей в социальных сетях. Все они ожидают, что их деньги будут в безопасности, транзакции — защищены, а переписка — конфиденциальна.

Поэтому если вы разрабатываете приложения, то должны очень серьезно относиться к вопросам безопасности. Не имеет значения, насколько ваше приложение маленькое или на первый взгляд незначительное, — безопасность всегда должна быть в поле вашего внимания.

В мире информационных технологий безопасность обеспечивается разными средствами, но самым важным все же остается криптография. С ней так или иначе связано почти все, что вы делаете на компьютере или телефоне. Криптография защищает онлайн-платежи, обеспечивает передачу сообщений по сети таким образом, чтобы перехвативший их человек не смог их прочитать, а также используется для шифрования файлов при их резервном копировании в облако.

Наша цель здесь не в том, чтобы научить вас всем тонкостям криптографии, — этой теме посвящены целые книги. Мы просто покажем, как с помощью средств, предлагаемых Python, можно создавать дайджесты, токены и в целом соблюдать правила безопасности в случаях, когда необходимо реализовать ту или иную криптографическую задачу. Важно понимать: под криптографией понимается не только шифрование и дешифровка. На самом деле в этой главе вы не найдете ни одного примера шифрования или расшифровки данных!

Полезные рекомендации

Запомните одно простое правило: никогда не пытайтесь разрабатывать собственные функции хеширования или шифрования. Просто не делайте этого. Всегда используйте уже существующие инструменты и функции. Разработка надежного, устойчивого и безопасного алгоритма хеширования или шифрования — крайне сложная задача, и ею должны заниматься профессиональные криптографы.

Тем не менее важно понимать основы криптографии, поэтому эту тему стоит изучать максимально тщательно. В Интернете можно найти много соответствующей информации, а в конце главы мы приведем ссылки на полезные источники.

А сейчас перейдем к первому модулю из стандартной библиотеки Python, о котором пойдет речь, — `hashlib`.

Модуль hashlib

Этот модуль предоставляет доступ к различным криптографическим хеш-алгоритмам. Такие алгоритмы являются математическими функциями, которые принимают сообщение произвольной длины и возвращают результат фиксированной длины, называемый *хешем* или *дайджестом*. Криптографические хеши применяются в самых разных задачах — от проверки целостности данных до безопасного хранения и проверки паролей.

Хеш-алгоритмы, пригодные для использования в криптографии, должны обладать такими свойствами, как:

- *детерминированность* — одно и то же сообщение всегда должно давать один и тот же хеш;
- *необратимость* — по хешу не должно быть возможно восстановить исходное сообщение;
- *устойчивость к коллизиям* — должно быть крайне сложно найти два разных сообщения, которые дают одинаковый хеш.

Эти свойства имеют решающее значение для безопасного использования хеш-функций. Например, крайне важно, чтобы пароли хранились только в виде хешей.

Свойство необратимости означает, что даже если произойдет утечка данных и злоумышленник получит доступ к базе паролей, то восстановить сами пароли из хешей будет невозможно. Кроме того, хранение паролей в виде хешей предполагает, что для проверки пароля при входе в систему достаточно вычислить хеш введенного значения и сравнить его с тем, который хранится в базе. Конечно, все это работает, только если хеш-функция детерминирована.

Свойство устойчивости к коллизиям тоже имеет большое значение. Оно обеспечивает целостность данных: если хеш используется как отпечаток данных, то важно, чтобы при любом их изменении менялся и сам отпечаток. Устойчивость к коллизиям не дает злоумышленнику подменить один документ другим, дающим тот же хеш. Кроме того, многие протоколы безопасности опираются на уникальность, гарантируемую хеш-функциями с защитой от коллизий.

Точный набор хеш-алгоритмов, доступных через модуль `hashlib`, зависит от библиотек, установленных в вашей системе. Однако некоторые алгоритмы гарантированно присутствуют на всех платформах. Посмотрим, какие из них доступны (ваши результаты могут различаться в зависимости от системы):

```
# hlib.txt
>>> import hashlib
>>> hashlib.algorithms_available
{'sha3_256', 'sha224', 'blake2b', 'sha512_224', 'ripemd160',
 'sha1', 'sha512_256', 'sha3_512', 'sha512', 'sha384', 'sha3_384',
 'sha3_224', 'shake_256', 'shake_128', 'sm3', 'md5-sha1', 'sha256',
 'md5', 'blake2s'}
>>> hashlib.algorithms_guaranteed
{'sha512', 'sha3_256', 'shake_128', 'sha224', 'blake2b',
 'shake_256', 'sha384', 'sha1', 'sha3_512', 'sha3_384', 'sha256',
 'sha3_224', 'md5', 'blake2s'}
```

Выполнив эти команды в оболочке Python, вы увидите множество доступных в системе алгоритмов. Если ваше приложение обменивается данными с другими программами, то лучше выбрать один из гарантированных алгоритмов — так вы будете уверены, что его поддерживают все платформы. Обратите внимание, что многие из них начинаются с `sha` — это сокращение от *Secure Hash Algorithm («надежный хеш-алгоритм»)*.

Продолжим пример в интерактивной оболочке — создадим хеш байтовой строки `b"Hash me now!"`:

```
>>> h = hashlib.blake2b()
>>> h.update(b"Hash me")
>>> h.update(b" now!")
>>> h.hexdigest()
'56441b566db9aafcf8cdad3a4729fa4b2bfaab0ada36155ece29f52ff70e1e9d'
'7f54cacfe44bc97c7e904cf79944357d023877929430bc58eb2dae168e73cedf'
>>> h.digest()
b'VD\x1bVm\xb9\xaa\xfc\xf8\xcd\xad:G)\xfaK+\xfa\xab\n\xda6\x15^'
b'\xce)\xf5/\xf7\x0e\x1e\x9d\x7fT\xca\xcf\xe4K\xc9|~\x90L\xf7'
```

```
b'\x99D5}\x028w\x92\x940\xbcX\xeb-\xae\x16\x8es\xce\xdf'
>>> h.block_size
128
>>> h.digest_size
64
>>> h.name
'blake2b'
```

Здесь используется криптографическая функция `blake2b()`, которая довольно сложна и появилась в Python начиная с версии 3.6. Мы создаем объект хеша `h` и обновляем его сообщение в два приема. Делать это не обязательно, но иногда приходится хешировать данные, которые поступают частями, и полезно знать, что обновление можно выполнять пошагово.

После того как сообщение полностью добавлено, мы получаем шестнадцатеричное представление дайджеста. На каждый байт приходятся два символа (каждый символ кодирует четыре бита, то есть половину байта). Мы также получаем байтовое представление дайджеста и изучаем свойства объекта хеша: внутренний размер блока (block size) у алгоритма составляет 128 байт, размер дайджеста (digest size) — 64 байта, и у него есть имя.

Теперь посмотрим, что получится, если вместо `blake2b()` использовать `sha512()`:

```
>>> hashlib.sha512(b"Hash me too!").hexdigest()
'a0d169ac9487fc6c78c7db64b54aefd01bd245bbd1b90b6fe5648c3c4eb0ea7d'
'93e1be50127164f21bc8ddb3dd45a6b4306dfe9209f2677518259502fed27686'
```

Хеш, полученный с помощью `sha512`, по длине совпадает с тем, что вернул `blake2b`. Обратите внимание: мы можем создать объект хеша, передав ему сообщение сразу при создании, и получить дайджест в одной строке.

Хеширование — интересная тема, и, конечно, простые примеры, которые мы рассмотрели, — это лишь начало. Функцию `blake2b()` можно адаптировать под разные задачи или с ее помощью защититься от определенных видов атак — и все это благодаря множеству настраиваемых параметров.

Здесь мы кратко упомянем только один из таких параметров. Подробную информацию вы найдете в официальной документации: https://docs.python.org/3/library/hashlib.html. Параметр `person` особенно интересен: он позволяет *персонализировать* хеш, заставляя его выдавать разные значения дайджеста для одного и того же сообщения. Это может повысить уровень безопасности в тех случаях, когда в одном приложении одна и та же хеш-функция используется для разных целей:

```
>>> import hashlib
>>> h1 = hashlib.blake2b(
...     b"Important data", digest_size=16, person=b"part-1")
>>> h2 = hashlib.blake2b(
...     b"Important data", digest_size=16, person=b"part-2")
```

```
>>> h3 = hashlib.blake2b(
...     b"Important data", digest_size=16)
>>> h1.hexdigest()
'c06b9af95d5aa6307e7e3fd025a15646'
>>> h2.hexdigest()
'9cb03be8f3114d0f06bddaedce2079c4'
>>> h3.hexdigest()
'7d35308ca3b042b5184728d2b1283d0d'
```

Здесь мы также воспользовались параметром `digest_size`, чтобы получить хеши длиной всего 16 байт.

Общие хеш-функции, такие как `blake2b()` или `sha512()`, не подходят для безопасного хранения паролей. Они слишком быстро выполняются на современных компьютерах, что делает возможными *атаки методом перебора (brute force)*: злоумышленник может попробовать миллионы вариантов в секунду, пока не найдет совпадение. Для таких целей применяются алгоритмы вывода ключа, например `pbkdf2_hmac()`. Они специально разработаны так, чтобы выполняться медленно и тем самым предотвращать атаки методом перебора. Алгоритм `pbkdf2_hmac()` достигает этого путем многократного применения обычной хеш-функции — количество итераций задается как параметр. По мере роста вычислительной мощности компьютеров важно со временем увеличивать количество итераций. Иначе вероятность успешной атаки методом перебора будет только возрастать.

Кроме того, надежные хеш-функции для паролей должны использовать *соль*. Это случайная последовательность байтов, которая используется для инициализации хеш-функции. Благодаря этому результат работы алгоритма становится уникальным, даже если входные данные совпадают. Это помогает защититься от атак с использованием предварительно вычисленных таблиц хешей. Функция `pbkdf2_hmac()` поддерживает добавление соли — для этого требуется обязательный параметр `salt`.

Вот как можно использовать `pbkdf2_hmac()` для хеширования пароля:

```
>>> import os
>>> dk = hashlib.pbkdf2_hmac("sha256", b"password123",
...     salt=os.urandom(16), iterations=200000)
>>> dk.hex()
'ac34579350cf6d05e01e745eb403fc50ac0e62fbeb553cbb895e834a77c37aed'
```

Обратите внимание, что в качестве соли мы использовали 16 случайных байт, полученных через `os.urandom()`, как рекомендовано в официальной документации.

Как правило, значение соли сохраняется вместе с хешем. Когда пользователь вводит пароль при входе, программа берет сохраненную соль, вычисляет хеш введенного пароля и сравнивает его с сохраненным значением. Использование той же самой соли обеспечивает совпадение хешей, если пароль верен.

Рекомендуем поэкспериментировать с модулем `hashlib` — рано или поздно вам точно придется им воспользоваться. Теперь перейдем к модулю `hmac`.

Алгоритм HMAC

Модуль `hmac` реализует алгоритм HMAC, описанный в RFC 2104 (https://datatracker.ietf.org/doc/html/rfc2104.html). Его название образовано от hash-based message authentication code — *код аутентификации (проверки подлинности) сообщений* или (в зависимости от источника) keyed-hash message authentication code — *код аутентификации сообщений, использующий хеш-функции с ключом*. HMAC — это широко применяемый механизм для проверки подлинности сообщений и защиты от их подмены.

Алгоритм работает следующим образом: сообщение объединяется с секретным ключом, после чего от этой комбинации вычисляется хеш. Его называют *кодом аутентификации сообщения* (message authentication code, MAC) или *подписью*. Подпись сохраняется или передается вместе с самим сообщением. Чтобы проверить, что сообщение не было подменено, необходимо повторно вычислить подпись с использованием того же секретного ключа и сравнить результат с исходной подписью. Секретный ключ должен храниться в безопасности: если злоумышленник получит к нему доступ, то сможет изменить сообщение и пересчитать подпись, тем самым обойдя механизм аутентификации.

Ниже показан небольшой пример вычисления MAC:

```
# hmc.py
import hmac
import hashlib
def calc_digest(key, message):
    key = bytes(key, "utf-8")
    message = bytes(message, "utf-8")
    dig = hmac.new(key, message, hashlib.sha256)
    return dig.hexdigest()

mac = calc_digest("secret-key", "Important Message")
```

Функция `hmac.new()` принимает три аргумента: секретный ключ, сообщение и хеш-алгоритм. Она возвращает объект HMAC, интерфейс которого похож на интерфейс объектов хешей из модуля `hashlib`. Ключ должен быть объектом типа `bytes` или `bytearray`, а сообщение — любым объектом, поддерживающим байтоподобный интерфейс. Поэтому перед созданием экземпляра HMAC (в примере он называется `dig`) ключ и сообщение предварительно преобразуются в байты. Полученный объект позволяет получить шестнадцатеричное представление хеша.

Позже в этой главе мы рассмотрим, как такие HMAC-подписи можно применять при работе с JWT. Но сначала кратко обсудим модуль `secrets`.

Модуль secrets

Этот небольшой модуль был добавлен в Python 3.6 и предназначен для генерации случайных чисел и токенов, а также безопасного сравнения дайджестов. Он использует наиболее безопасные генераторы случайных чисел, предоставляемые операционной системой, чтобы создавать токены и случайные значения, подходящие для криптографических задач. Ниже кратко описано, что именно он предлагает.

Случайные объекты

Для генерации случайных объектов модуль `secrets` предоставляет три функции:

```
# secrs/secr_rand.py
import secrets

print(secrets.choice("Choose one of these words".split()))
print(secrets.randbelow(10**6))
print(secrets.randbits(32))
```

Первая функция, `choice()`, возвращает случайный элемент из непустой последовательности. Вторая, `randbelow()`, генерирует случайное целое число от 0 до указанного значения (не включая его). А третья, `randbits()`, возвращает целое число, состоящее из заданного количества случайных битов. Запуск этих функций выдает разные результаты при каждом выполнении, что естественно при генерации случайных данных:

```
$ python secr_rand.py
one
133025
1509555468
```

Эти функции следует использовать вместо функций модуля `random` в любых криптографических сценариях, поскольку они специально разработаны для безопасного применения.

Теперь посмотрим, какие возможности модуль предлагает для работы с токенами.

Генерация токенов

Как и в предыдущем случае, модуль `secrets` предоставляет три функции, каждая из которых создает токен в разном формате. Ниже приведен пример:

```
# secrs/secr_rand.py
import secrets

print(secrets.token_bytes(16))
print(secrets.token_hex(32))
print(secrets.token_urlsafe(32))
```

Функция `token_bytes()` возвращает случайную последовательность байтов заданной длины (в примере — 16 байт). Функция `token_hex()` делает то же самое, но результат возвращается в шестнадцатеричном формате. Функция `token_urlsafe()` возвращает токен, содержащий только такие символы, которые можно безопасно добавлять в URL. Вывод этих функций представляет собой продолжение предыдущего запуска и каждый раз будет другим:

```
b'\x0f\x8b\x8f\x0f\xe3\xceJ\xbc\x18\xf2\x1e\xe0i\xee1\x99'
98e80cddf6c37181131804567239 9b0950b8e3207d18b50d99d724d31d17f0a7
63eNkRalj8dgZqmkezjbEYoGddVcutgvwJthSLf5kho
```

Теперь посмотрим, как с помощью этих инструментов можно написать собственный генератор случайных паролей:

```
# secrs/secr_gen.py
import secrets
from string import digits, ascii_letters

def generate_pwd(length=8):
    chars = digits + ascii_letters
    return "".join(secrets.choice(chars) for c in range(length))

def generate_secure_pwd(length=16, upper=3, digits=3):
    if length < upper + digits + 1:
        raise ValueError("Nice try!")
    while True:
        pwd = generate_pwd(length)
        if (
            any(c.islower() for c in pwd)
            and sum(c.isupper() for c in pwd) >= upper
            and sum(c.isdigit() for c in pwd) >= digits
        ):
            return pwd

print(generate_secure_pwd())
print(generate_secure_pwd(length=3, upper=1, digits=1))
```

Функция `generate_pwd()` создает случайную строку заданной длины, объединяя символы, случайным образом выбранные из строки, которая содержит все буквы латинского алфавита (прописные и строчные), а также десять десятичных цифр.

Затем мы определяем другую функцию — `generate_secure_pwd()`, которая продолжает вызывать `generate_pwd()`, пока случайная строка не будет соответствовать заданным минимальным требованиям. Пароль должен иметь заданную длину и содержать как минимум одну строчную букву, а также указанное количество прописных букв и цифр.

Если суммарное количество требуемых прописных и строчных букв, а также цифр превышает длину создаваемого пароля, то выполнить условия невозможно. В таком случае цикл будет бесконечно перебирать неподходящие варианты. Поэтому перед его запуском выполняется проверка, и при невозможности удовлетворить условия выбрасывается исключение `ValueError`.

Тело цикла `while` устроено просто: сначала генерируется случайный пароль, затем выполняется проверка условий с помощью функций `any()` и `sum()`. Функция `any()` возвращает `True`, если хотя бы один элемент переданного ей итерируемого объекта приводится к `True`. Применение `sum()` — более интересный процесс: в нем используется *полиморфизм*. Как вы, возможно, помните из главы 2, тип `bool` в Python является подклассом `int`. Поэтому при суммировании итерируемого объекта, содержащего значения `True` и `False`, функция `sum()` автоматически интерпретирует их как числа 1 и 0 соответственно. Это и есть пример полиморфизма, о котором мы коротко говорили в главе 6.

Результат запуска приведен ниже:

```
$ python secr_gen.py
mgQ3Hj57KjD1LI7M
b8G
```

Разумеется, использовать пароль длиной всего 3 символа не стоит.

Один из распространенных способов применения случайных токенов — генерация URL для сброса пароля на сайте. Вот пример того, как можно сгенерировать такой адрес:

```
# secrs/secr_reset.py
import secrets

def get_reset_pwd_url(token_length=16):
    token = secrets.token_urlsafe(token_length)
    return f"https://example.com/reset-pwd/{token}"

print(get_reset_pwd_url())
```

Запуск этого примера дает следующий результат:

```
$ python secr_reset.py
https://example.com/reset-pwd/ML_6_2wxDpXmDJLHrDnrRA
```

Сравнение дайджестов

Это может показаться неожиданным, но, помимо функций, описанных выше, модуль `secrets` предоставляет функцию `compare_digest(a, b)`, которая, казалось бы, делает то же самое, что и обычное сравнение `a == b`. Зачем тогда нужна отдельная функция? Дело в том, что `compare_digest()` специально разработана для защиты от атак по времени выполнения. Такие атаки позволяют извлекать информацию о том, с какого места два дайджеста начинают различаться по времени, за которое операция сравнения завершается с ошибкой. Функция `compare_digest()` предотвращает данную уязвимость, устраняя связь между временем выполнения и моментом отказа при сравнении. Это прекрасный пример того, насколько изощренными могут быть методы атак. Если вы слегка удивились при прочтении

этого абзаца, то, возможно, теперь лучше понимаете, почему мы советовали никогда не реализовывать криптографические функции самостоятельно.

На этом мы завершаем краткий экскурс в криптографические возможности стандартной библиотеки Python и переходим к другой разновидности токенов — JWT.

Токены JWT

JSON Web Token (JWT) — открытый стандарт на основе JSON для создания токенов, в которых содержится определенный набор *утверждений*. JWT часто используются в качестве токенов аутентификации. В этом контексте утверждения, как правило, представляют собой сведения о личности и правах аутентифицированного пользователя. Токены подписываются криптографически, что позволяет проверить, не были ли они изменены после выпуска. Более подробная информация о данной технологии доступна на сайте https://jwt.io.

Токен такого типа состоит из трех частей, соединенных точками, и имеет формат *A.B.C*. Здесь *A* — это *заголовок*, указывающий, что перед нами JWT, и сообщающий, какой алгоритм используется для вычисления подписи, *B* — *полезная нагрузка*, в которой находятся утверждения, а *C* — *подпись*, предназначенная для проверки подлинности токена. Все три части кодируются с помощью безопасного для использования в URL варианта кодировки Base64 (в дальнейшем будем называть ее Base64URL). Она позволяет безопасно использовать JWT в URL (обычно как параметры запроса), но они встречаются и в других местах, например в HTTP-заголовках.

Base64 — популярная схема кодирования, преобразующая бинарные данные в текст, пригодный для передачи в формате ASCII, с использованием представления Radix-64. В этом представлении используются символы латинского алфавита в верхнем и нижнем регистрах (A — Z, a — z), цифры от 0 до 9, а также символы + и / — всего 64 возможных значения. Base64, например, применяется для кодирования изображений, прикрепленных к электронным письмам. Этот процесс происходит незаметно, так что большинство пользователей даже не подозревают о его существовании.

Base64URL — вариант Base64, в котором символы + и /, имеющие специальное значение в URL, заменяются на - и _. Символ =, используемый в Base64 для выравнивания длины, тоже имеет особое значение в URL, поэтому в Base64URL опускается.

Принцип работы таких токенов немного отличается от всего, что мы рассматривали в главе до настоящего момента. В частности, содержимое токена всегда доступно для просмотра. Чтобы получить информацию об используемом алгоритме и полезной нагрузке, достаточно просто декодировать части *A* и *B* с помощью Base64URL. Надежность обеспечивается частью *C* — это HMAC-подпись, сформированная по заголовку и полезной нагрузке. Если кто-то попытается изменить

заголовок или полезную нагрузку (части *A* или *B*), а затем закодирует их обратно и подставит в токен, то подпись уже не будет совпадать. В этом случае токен считается недействительным.

Благодаря этому можно, например, добавить в полезную нагрузку утверждение о том, что пользователь *вошел в систему как администратор*. И если токен прошел аутентификацию, то мы можем быть уверены, что пользователь действительно авторизован с такими правами.

При работе с JWT крайне важно изучить и применять надежные практики безопасности. Например, никогда не следует принимать неподписанные токены, а также необходимо ограничивать список допустимых алгоритмов для кодирования и декодирования. Эти и другие меры безопасности играют ключевую роль, и стоит выделить время, чтобы тщательно разобраться в них.

Для следующего примера вам нужно будет установить пакеты PyJWT и cryptography. Как обычно, они указаны в списке зависимостей в исходном коде к главе.

Теперь рассмотрим простой пример:

```
# jwt/tok.py
import jwt

data = {"payload": "data", "id": 123456789}
algs = ["HS256", "HS512"]

token = jwt.encode(data, "secret-key")
data_out = jwt.decode(token, "secret-key", algorithms=algs)
print(token)
print(data_out)
```

Сначала определяется полезная нагрузка — словарь с идентификатором и некоторыми дополнительными данными. Затем создается токен с помощью функции `jwt.encode()`, в которую передается эта нагрузка и секретный ключ. Данный ключ используется для вычисления HMAC-подписи по заголовку и полезной нагрузке токена. Далее токен декодируется обратно с применением функции `jwt.decode()`, при этом указывается список допустимых алгоритмов подписи. По умолчанию для подписи используется алгоритм HS256, и в этом примере при проверке допускаются HS256 и HS512. Если токен был подписан другим алгоритмом, то произойдет исключение и токен будет отклонен.

Вот пример вывода:

```
$ python jwt/tok.py
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJwYXlsb2FkIjoiZGF0YSIsIm...
{'payload': 'data', 'id': 123456789}
```

Как видите, токен представляет собой бинарную строку, состоящую из частей, закодированных в формате Base64URL (для краткости в примере они сокращены до одной строки). Мы вызвали функцию `jwt.decode()`, передав в нее правильный секретный ключ. Если бы использовался другой ключ, то произошла бы ошибка, поскольку подпись можно проверить только с помощью того же ключа, который использовался при создании токена.

JWT часто применяются для передачи информации между двумя сторонами. Например, в протоколах аутентификации, которые позволяют сайтам использовать сторонние сервисы в качестве удостоверяющих центров, передача JWT-токенов — обычная практика. В таких случаях секретный ключ, которым подписываются токены, должен быть разделен между сторонами, и именно поэтому его часто называют общим секретом.

Важно обеспечить защиту этого ключа: любой, кто получит к нему доступ, сможет создавать действительные токены.

Иногда может понадобиться просто просмотреть содержимое токена, не проверяя подпись. Это можно сделать, вызвав `decode()` следующим образом:

```
# jwt/tok.py
jwt.decode(token, options={"verify_signature": False})
```

Такой способ декодирования бывает полезен, например, когда значения из полезной нагрузки токена нужны для восстановления секретного ключа. Однако такая техника довольно сложна, и в рамках текущей главы мы не будем в нее углубляться. Вместо этого посмотрим, как задать другой алгоритм для вычисления подписи:

```
# jwt/tok.py
token512 = jwt.encode(data, "secret-key", algorithm="HS512")
data_out = jwt.decode(
    token512, "secret-key", algorithms=["HS512"]
)
print(data_out)
```

Здесь токен генерируется с помощью алгоритма HS512, и при его проверке указано, что принимаются только токены, подписанные с использованием HS512. Результатом декодирования снова будет словарь с полезной нагрузкой, такой же, как при создании токена.

В полезную нагрузку можно поместить любые данные, но стоит знать, что существует набор стандартизированных утверждений. Они позволяют обеспечить безопасность, согласованность и совместимость между разными системами и приложениями.

Зарегистрированные утверждения

Стандарт JWT определяет следующий перечень официальных *зарегистрированных утверждений*:

- `iss` (issuer) — издатель токена;
- `sub` (subject) — субъект — информация о стороне, к которой относится токен;
- `aud` (audience) — аудитория, для которой предназначен токен;
- `exp` (expiration time) — срок действия токена — время, после которого он становится недействительным;
- `nbf` (not before) — «не ранее» — время, до наступления которого токен считается недействительным;
- `iat` (issued at) — время выпуска токена;
- `jti` (JWT ID) — уникальный идентификатор токена.

Утверждения, не входящие в этот список, делятся на два типа.

- *Публичные* (public claims) — утверждения, зарезервированные для общедоступных целей. Их можно зарегистрировать в реестре утверждений JSON Web Token Claims Registry IANA либо называть так, чтобы избежать пересечений с другими публичными или официальными именами. Один из подходов — добавлять к имени утверждения префикс с доменным именем.
- *Приватные* (private claims) — все прочие утверждения, не относящиеся к вышеуказанным категориям. Обычно они имеют смысл только в рамках конкретного приложения и не интерпретируются за его пределами. Чтобы избежать путаницы, важно следить за тем, чтобы имена таких утверждений не пересекались с зарегистрированными.

Больше информации об утверждениях можно найти на официальном сайте (https://datatracker.ietf.org/doc/html/rfc7519). А теперь рассмотрим пару примеров кода, в которых используется часть этих утверждений.

Утверждения, связанные с временем

Рассмотрим пример, как использовать утверждения, связанные с временем:

```
# jwt/claims_time.py
from datetime import datetime, timedelta, UTC
from time import sleep, time
import jwt

iat = datetime.now(tz=UTC)
nfb = iat + timedelta(seconds=1)
exp = iat + timedelta(seconds=3)

data = {"payload": "data", "nbf": nfb, "exp": exp, "iat": iat}
```

```
def decode(token, secret):
    print(f"{time():.2f}")
    try:
        print(jwt.decode(token, secret, algorithms=["HS256"]))
    except (
        jwt.ImmatureSignatureError,
        jwt.ExpiredSignatureError,
    ) as err:
        print(err)
        print(type(err))

secret = "secret-key"
token = jwt.encode(data, secret)

decode(token, secret)
sleep(2)
decode(token, secret)
sleep(2)
decode(token, secret)
```

Здесь мы задаем значение утверждения `iat` (время выпуска токена) как текущее время по *Всемирному координированному времени* (Coordinated Universal Time, UTC). Далее устанавливаем `nbf` (not before — «не ранее») на одну секунду вперед и `exp` (expiration time — «срок действия») на три секунды вперед от текущего времени. Мы определяем вспомогательную функцию `decode()`, которая обрабатывает ситуации, когда токен еще недействителен или уже истек, перехватывая соответствующие исключения. Функция вызывается трижды, с паузами между вызовами. Паузы задаются с помощью `sleep()`.

Таким образом, токен проверяется в трех состояниях: до наступления времени действия (до `nbf`), в момент, когда он действителен, и после истечения срока действия (`exp`). Кроме того, функция выводит текущую временную метку перед попыткой декодирования, что помогает отследить момент выполнения (пустые строки добавлены для удобства чтения):

```
$ python jwt/claims_time.py
1716674892.39
The token is not yet valid (nbf)
<class 'jwt.exceptions.ImmatureSignatureError'>

1716674894.39
{'payload': 'data', 'nbf': 1716674893, 'exp': 1716674895, 'iat': 1716674892}

1716674896.39
Signature has expired
<class 'jwt.exceptions.ExpiredSignatureError'>
```

Как видно из вывода, все сработало ожидаемо: в случае ошибки мы получаем информативные сообщения от исключений, а когда токен действителен — возвращается исходная полезная нагрузка.

Утверждения, связанные с аутентификацией

Теперь рассмотрим еще один краткий пример — на этот раз с утверждениями `iss` (issuer — «издатель») и `aud` (audience — «аудитория»). Логика примера во многом схожа с логикой предыдущего, и выполнять его мы будем аналогичным образом:

```
# jwt/claims_auth.py
import jwt

data = {"payload": "data", "iss": "hein", "aud": "learn-python"}
secret = "secret-key"
token = jwt.encode(data, secret)

def decode(token, secret, issuer=None, audience=None):
    try:
        print(
            jwt.decode(
                token,
                secret,
                issuer=issuer,
                audience=audience,
                algorithms=["HS256"],
            )
        )
    except (
        jwt.InvalidIssuerError,
        jwt.InvalidAudienceError,
    ) as err:
        print(err)
        print(type(err))

# Если не указать ни получателя (audience), ни издателя (issuer),
# то валидация не пройдет
decode(token, secret)

# Если не указать издателя — проверка пройдет
decode(token, secret, audience="learn-python")

# Если не указать получателя — проверка не пройдет
decode(token, secret, issuer="hein")

# Обе проверки не пройдут
decode(token, secret, issuer="wrong", audience="learn-python")
decode(token, secret, issuer="hein", audience="wrong")

# Эта проверка пройдет успешно
decode(token, secret, issuer="hein", audience="learn-python")
```

Как видите, при создании токена мы указали значения и для `iss`, и для `aud`. Декодирование такого токена пройдет успешно, даже если не указывать параметр `issuer` в функции `jwt.decode()`. Но если вы передаете этот параметр и он

не совпадает со значением поля `iss` в токене, то декодирование завершится с ошибкой. В свою очередь, если параметр `audience` опущен или не совпадает с полем `aud` в токене, то `jwt.decode()` тоже завершится с ошибкой.

Как и в предыдущем примере, мы реализовали собственную функцию `decode()`, которая обрабатывает соответствующие исключения. Обратите внимание на последовательность вызовов и сопутствующий вывод (для удобства чтения в него добавлены пустые строки):

```
$ python jwt/claims_time.py
Invalid audience
<class 'jwt.exceptions.InvalidAudienceError'>

{'payload': 'data', 'iss': 'hein', 'aud': 'learn-python'}

Invalid audience
<class 'jwt.exceptions.InvalidAudienceError'>

Invalid issuer
<class 'jwt.exceptions.InvalidIssuerError'>

Audience doesn't match
<class 'jwt.exceptions.InvalidAudienceError'>

{'payload': 'data', 'iss': 'hein', 'aud': 'learn-python'}
```

Заметьте, что в примере мы специально варьировали аргументы, передаваемые в `jwt.decode()`, чтобы показать поведение в разных сценариях. Однако в реальных приложениях, как правило, используются фиксированные значения для `audience` и `issuer` и все токены, не соответствующие этим условиям, отклоняются. Пропуск параметра `issuer` при декодировании означает, что вы принимаете токены от любого издателя. Пропуск параметра `audience` означает, что вы принимаете только те токены, в которых не указана аудитория.

Теперь рассмотрим последний пример — на этот раз с более сложным сценарием использования.

Применение асимметричных (публичных) алгоритмов

Иногда использование общего секрета (shared secret) может быть не лучшим решением. В таких случаях вместо HMAC можно применять асимметричную пару ключей для создания подписи JWT.

В этом примере мы создадим и проверим токен, используя пару *RSA*-ключей.

Криптография с открытым ключом, или асимметричная криптография, — это система, где используются пары ключей: публичный, который можно свободно

распространять, и закрытый, известный только владельцу. Если вы хотите лучше изучить эту тему, то в конце главы можете найти рекомендации по дополнительным материалам. Подпись может быть создана с помощью закрытого ключа, а для ее проверки используется публичный ключ. Таким образом, две стороны могут обмениваться JWT, не используя общий секрет.

Сначала создадим пару ключей RSA. Для этого воспользуемся утилитой `ssh-keygen` из комплекта OpenSSH (https://www.ssh.com/academy/ssh/keygen). В папке, содержащей сценарии этой главы, мы создаем подпапку `jwt/rsa`. Внутри нее выполняем следующую команду:

```
$ ssh-keygen -t rsa –m PEM
```

Когда программа `ssh-keygen` попросит ввести имя файла, укажите `key` (файлы будут сохранены в текущей папке). Когда появится запрос на парольную фразу — просто нажмите Enter.

После генерации ключей можно вернуться в папку `ch09` и выполнить следующий код:

```
# jwt/token_rsa.py
import jwt

data = {"payload": "data"}

def encode(data, priv_filename, algorithm="RS256"):
    with open(priv_filename, "rb") as key:
        private_key = key.read()
    return jwt.encode(data, private_key, algorithm=algorithm)

def decode(data, pub_filename, algorithm="RS256"):
    with open(pub_filename, "rb") as key:
        public_key = key.read()
    return jwt.decode(data, public_key, algorithms=[algorithm])

token = encode(data, "jwt/rsa/key")
data_out = decode(token, "jwt/rsa/key.pub")
print(data_out) # {'payload': 'data'}
```

В этом примере определены две вспомогательные функции: одна кодирует токен с помощью закрытого ключа, другая декодирует с помощью открытого. Обратите внимание: в функции `encode()` используется алгоритм RS256. При кодировании мы передаем закрытый ключ, который применяется для создания подписи JWT. При декодировании, наоборот, используется открытый ключ, с помощью которого проверяется подпись.

Логика довольно простая, но полезная: подумайте, в каких случаях такой подход — с использованием асимметричной криптографии — может быть более уместным, чем общий секретный ключ.

Полезные источники

Предлагаем список ресурсов, которые могут быть полезны, если вы захотите глубже погрузиться в увлекательный мир криптографии.

- Криптография: https://en.wikipedia.org/wiki/Cryptography (https://ru.wikipedia.org/wiki/Криптография).
- JSON Web Tokens: https://jwt.io.
- Стандарт RFC для JSON Web Token: https://datatracker.ietf.org/doc/html/rfc7519.
- Хеш-функции: https://en.wikipedia.org/wiki/Cryptographic_hash_function (https://ru.wikipedia.org/wiki/Хеш-функция).
- HMAC: https://en.wikipedia.org/wiki/HMAC (https://ru.wikipedia.org/wiki/HMAC).
- Криптографические службы (стандартная библиотека Python): https://docs.python.org/3/library/crypto.html.
- Реестр утверждений JWT IANA: https://www.iana.org/assignments/jwt/jwt.xhtml.
- Библиотека PyJWT: https://pyjwt.readthedocs.io.
- Библиотека cryptography: https://cryptography.io.

В Интернете и в печатной литературе можно найти множество подробностей, но мы рекомендуем сначала освоить базовые концепции и только потом постепенно углубляться в конкретные темы, которые вам особенно интересны.

Резюме

В этой короткой главе вы познакомились с криптографическими возможностями стандартной библиотеки Python. Вы узнали, как создавать хеши (или дайджесты) сообщений с помощью различных криптографических функций, а также как формировать токены и работать со случайными данными в контексте криптографии.

Затем вы вышли за пределы стандартной библиотеки и познакомились с JSON Web Tokens (JWT) — широко применяемым механизмом аутентификации и передачи утверждений в современных системах и приложениях.

Самое главное, что стоит уяснить: в криптографии ручная реализация — это всегда риск. Поэтому лучше довериться профессионалам и использовать надежные готовые инструменты, которые уже доступны.

В следующей главе мы займемся тестированием кода, чтобы убедиться, что он действительно работает так, как мы задумывали.

10 Тестирование

> Подобно тому как купец на базаре при покупке золота проверяет его: нагревает, плавит, режет — чтобы убедиться в его подлинности, так же проверяйте и мое учение и, только убедившись в его истинности, принимайте его!
>
> *Будда*

Нам, авторам, очень близка эта цитата Будды. В мире разработки программного обеспечения она как нельзя лучше иллюстрирует важную привычку: никогда не доверять коду лишь потому, что его написал умный человек, или потому, что он «всегда работал». Если код не протестирован — доверять ему нельзя.

Зачем нужны тесты? В первую очередь, они обеспечивают предсказуемость. Или, по крайней мере, помогают к ней приблизиться. Конечно, баги все равно случаются — они иногда появляются даже в самом аккуратном коде. Но мы стремимся к тому, чтобы поведение программы было как можно более стабильным и понятным. Непредсказуемость — это именно то, чего мы хотим избежать. Особенно если речь идет о программном обеспечении, которое взаимодействует с датчиками в самолете, поезде или ядерном реакторе. В таких случаях последствия могут быть катастрофическими.

Поэтому код нужно тестировать, чтобы проверить:

- правильно ли он себя ведет;
- работает ли должным образом даже в пограничных ситуациях;
- не зависает ли, если внешние компоненты недоступны или неисправны;
- остается ли его производительность в допустимых пределах;
- многие другие показатели.

Текущая глава посвящена именно этим вопросам — как подготовить ваш код к встрече с реальным миром, сделать его достаточно быстрым и устойчивым к неожиданным ситуациям.

Мы рассмотрим такие темы, как:

- общие рекомендации по тестированию;
- модульное тестирование;
- разработка через тестирование (кратко).

Начнем с основ: что такое тестирование и зачем оно нужно.

Тестирование приложения

Существует множество видов тестов — так много, что в некоторых компаниях создают специальный отдел под названием *«отдел обеспечения качества»* (quality assurance, QA), сотрудники которого занимаются исключительно тестированием программ, разработанных внутри компании.

Классификация тестов начинается с того, что их можно разделить на две обширные категории. *Тестирование методом «белого ящика»* — это процесс, при котором тесты проверяют внутреннюю реализацию кода, вплоть до мельчайших деталей. *Тестирование методом «черного ящика»*, наоборот, рассматривает приложение как закрытую систему, внутренняя реализация которой игнорируется. Даже язык программирования или используемые технологии не имеют значения. Все, что делают такие тесты, — подают входные данные на вход условной коробки и проверяют результат на выходе.

Существует и промежуточная категория — тестирование методом «серого ящика». Оно похоже на тестирование методом «черного ящика», но в нем имеет место частичное знание внутренней логики, включая используемые алгоритмы и структуры данных, а также ограниченный доступ к исходному коду.

В каждой из описанных ранее категорий существуют разные виды тестов, и каждый из них выполняет свою задачу. Ниже описано несколько примеров.

- *Тесты пользовательского интерфейса* — проверяют, корректно ли отображается информация на клиентской стороне приложения: ссылки, кнопки, рекламные блоки и все, что должно быть видно пользователю. Вдобавок такие тесты могут проверять, можно ли пройти по определенному сценарию, задействуя элементы *пользовательского интерфейса*.
- *Сценарные тесты* — используют истории или сценарии, которые помогают тестировщику воспроизвести сложную ситуацию или проверить конкретную часть системы.
- *Интеграционные тесты* — проверяют, как ведут себя разные компоненты приложения при совместной работе, обмениваясь данными через интерфейсы.

- *Проверочные (smoke) тесты* — особенно полезны при развертывании новой версии приложения. Они позволяют убедиться, что ключевые и наиболее критические части системы работают. Название пришло из инженерной практики: электрическая схема не дымится.
- *Приемочные тесты* или *пользовательское приемочное тестирование* (user acceptance testing, UAT) — выполняются совместно разработчиком и владельцем продукта (например, в среде SCRUM), чтобы подтвердить, что поставленная задача была выполнена правильно.
- *Функциональные тесты* — проверяют, реализованы ли функции программного обеспечения в соответствии с требованиями.
- *Разрушающие тесты* — намеренно «выводят из строя» части системы, моделируя сбои, чтобы оценить, насколько устойчиво работает остальная часть. Такие тесты активно применяются в компаниях, предоставляющих высоконадежные сервисы.
- *Тесты производительности* — измеряют, насколько хорошо система работает при определенной нагрузке (по данным или трафику). Это позволяет инженерам выявить узкие места, которые могут привести к сбоям при высокой нагрузке или помешать масштабированию.
- *Тесты удобства использования* и *тесты пользовательского опыта* (*UX-тесты*) — проверяют, насколько интерфейс понятен и удобен. Вдобавок они дают дизайнерам ценные данные, позволяющие сделать интерфейс более удобным для пользователей.
- *Тесты безопасности* и *тесты на проникновение* — оценивают степень защищенности системы от атак, несанкционированного доступа и других угроз.
- *Модульные тесты (unit-тесты)* — помогают разработчику писать надежный и согласованный код, предоставляя первую линию обратной связи и защиты от ошибок при написании и рефакторинге.
- *Регрессионные тесты* — информируют разработчика о возможных нарушениях в работе функций после обновлений. Поводом для такой проверки может быть возвращение старой ошибки, ухудшение существующей функции или появление нового сбоя.

Тема тестирования настолько обширна, что ей посвящено великое множество книг и статей. Если вы хотите досконально изучить разные типы тестов, то вам стоит обратиться к специализированным источникам. А в этой главе мы сосредоточимся на модульных тестах, поскольку они являются основой качественной разработки и составляют бóльшую часть тестов, которые пишет сам разработчик.

Тестирование — это *искусство*. И, к сожалению, не то, которому можно научиться только по книгам. Вы можете выучить все определения (и это стоит сделать), можете собрать максимум теоретических знаний о тестировании.

Но по-настоящему качественно тестировать свой код вы научитесь только по мере обретения опыта.

Когда вы сталкиваетесь с тем, что не можете нормально провести рефакторинг, поскольку любое, даже небольшое изменение вызывает сбой в тестах, вы начинаете понимать, как писать менее жесткие и ограничивающие тесты. Такие тесты все еще проверяют корректность кода, но при этом не мешают вам свободно его менять, пробовать, улучшать.

Когда вас слишком часто вызывают исправлять неожиданные баги, вы учитесь более продуманно покрывать код тестами, замечать пограничные случаи и заранее набрасывать варианты их обработки, прежде чем они выльются в реальные ошибки.

Когда вы тратите слишком много времени на чтение и переписывание тестов, чтобы внести небольшое изменение в поведение системы, вы учитесь писать более простые, короткие и точные тесты, фокусируясь на сути.

Можно продолжать в том же духе: *«Когда вы сталкиваетесь с... — вы учитесь...»* Но, думаем, суть уже ясна: тестированию нужно учиться на практике, на своем опыте. Что мы советуем? Изучайте теорию максимально тщательно — и обязательно экспериментируйте с разными подходами. И по возможности учитесь у более опытных разработчиков — это невероятно эффективно.

В идеале с опытом вы начнете ощущать, что исходный код и модульные тесты не две отдельные сущности. Тесты не дополнительный элемент. Они неотделимы от кода. Исходный код и модульные тесты влияют друг на друга и развиваются вместе.

Структура теста

Прежде чем перейти к модульному тестированию, разберем, что такое тест и в чем его цель.

Тест — это фрагмент кода, задача которого — проверить определенное поведение системы. Это может быть, например, вызов функции с двумя целыми числами, наличие у объекта свойства `donald_duck` или проверка того, что после оформления заказа через *API* (программный интерфейс) примерно через минуту в базе данных появляется корректно разложенная информация об этом заказе.

Структура типичного теста состоит из трех частей.

1. *Подготовка*. Здесь мы настраиваем среду. Создаем все необходимые данные, объекты и сервисы и размещаем их там, где они понадобятся. Все должно быть готово к использованию.
2. *Выполнение*. Здесь мы выполняем код, который тестируем. Проводим действие, используя те данные и интерфейсы, которые были подготовлены на предыдущем шаге.

3. *Проверка*. Здесь мы проверяем результат и убеждаемся, что он соответствует ожиданиям. Это может быть значение, возвращаемое функцией, запись в базе данных (появилась или, наоборот, отсутствует), факт изменения данных, наличие HTTP-запроса, вызов метода — все, что должно было произойти в результате выполнения.

Большинство тестов соответствует описанной ранее трехчастной структуре, но в полноценном наборе тестов обычно встречаются и другие элементы, участвующие в процессе тестирования.

- *Подготовка* (setup). Это этап, часто имеющийся в разных тестах. Он представляет собой логику инициализации, которую можно настроить для выполнения перед каждым тестом, для класса, модуля или даже всей сессии. На этом этапе разработчики, как правило, настраивают подключение к базам данных, заполняют их необходимыми сведениями и выполняют прочие действия, чтобы создать корректную тестовую среду.
- *Очистка* (teardown). Противоположный этап — выполняется после прохождения тестов. Как и подготовку, его можно настроить на любой уровень: отдельный тест, класс, модуль или сессию. На данном этапе обычно удаляются все временные артефакты, созданные в ходе тестов, и очищается среда. Это важно, поскольку позволяет избежать остаточных данных и обеспечить чистый старт для каждого теста.
- *Фикстуры* (fixtures). Это наборы данных, используемые в тестах. Благодаря фиксированным значениям и структурам тесты становятся предсказуемыми, и можно надежно сравнивать результаты с ожидаемыми.

В этой главе мы будем использовать библиотеку `pytest` — мощный инструмент, который, по сравнению с использованием только стандартной библиотеки, упрощает тестирование. Библиотека `pytest` предоставляет множество вспомогательных средств, благодаря которым можно сосредоточиться на самой логике теста, а не на подготовительных действиях или шаблонном коде. Когда мы перейдем к практическим примерам, вы увидите, что в `pytest` фикстуры, этапы подготовки и очистки зачастую органично переплетаются друг с другом.

Рекомендации по написанию тестов

Как и сам код, тесты могут быть хорошими или плохими — с множеством промежуточных оттенков. Чтобы писать качественные тесты, стоит придерживаться следующих рекомендаций.

- *Делайте тесты максимально простыми*. В тестах допустимо нарушать некоторые правила чистого кода — например, жестко прописывать значения или повторять код. Главное в тестах — их ясность и читабельность. Если тест тяжело читать и понимать, то вы не сможете быть уверены, что он действительно проверяет правильность поведения кода.

- *Один тест — одна проверка*. Очень важно, чтобы каждый тест проверял ровно одну вещь. Совершенно нормально написать несколько тестов для одной функции или объекта, но каждый из них должен иметь одну четкую цель и не выходить за ее рамки.
- *Тесты не должны делать лишних предположений*. Это может показаться неочевидным на первых порах, но принцип важный. Проверка, что результат вызова функции равен `[1, 2, 3]`, — не то же самое, что проверка, содержит ли список числа 1, 2 и 3. В первом случае мы предполагаем и порядок элементов, а во втором — только состав. Подобные различия могут быть тонкими, но они оказывают критическое влияние на корректность и устойчивость тестов.
- *Тесты должны проверять «что», а не «как»*. То есть важно фокусироваться на поведении функции, а не на внутренней реализации. Например, нужно проверять, что функция вычисляет квадратный корень *(что)*, а не то, что она внутри вызывает `math.sqrt()` *(как)*. Исключения составляют случаи, когда вы пишете тесты производительности или вам нужно точно проконтролировать, как выполняется операция. Во всех остальных случаях желательно избегать проверки реализации и концентрироваться на ожидаемом результате. Тесты, ориентированные на реализацию, ограничивают свободу рефакторинга, делают код тестов хрупким и ухудшают его качество при частых изменениях основного кода.
- *Тесты должны использовать минимально необходимое количество фикстур*. Это один из ключевых моментов. Фикстуры имеют тенденцию разрастаться со временем. Вдобавок они часто меняются, и если тесты используют слишком много фикстур или содержат избыточные зависимости, то рефакторинг замедляется, а поиск ошибок усложняется. Нужно стремиться к тому, чтобы набор фикстур был достаточным, но не избыточным — ровно столько, сколько необходимо для корректной работы теста.
- *Тесты должны потреблять как можно меньше ресурсов*. Причина проста: любой разработчик, взявший код из репозитория, должен иметь возможность запустить тесты, независимо от того, насколько мощная у него машина. Это может быть легкая виртуальная машина или сборка в CircleCI — тесты должны работать, не давая чрезмерную нагрузку на систему.
- *Тесты должны выполняться максимально быстро*. Часто база тестов в проекте оказывается больше самого исходного кода — особенно по мере роста проекта. Неважно, насколько велик проект, но в любом случае мы в итоге получаем сотни или тысячи тестов, и чем быстрее они выполняются, тем быстрее можно вернуться к работе над кодом. В *разработке через тестирование* тесты запускаются очень часто, поэтому скорость их выполнения критически важна.

CircleCI — одна из крупнейших платформ для непрерывной интеграции и доставки (CI/CD). Она легко интегрируется с сервисами наподобие GitHub. Чтобы начать использовать ее, достаточно добавить конфигурационный файл в исходный код проекта. После этого CircleCI автоматически запускает тесты при каждом предложении внести изменения в основную ветку кода.

Модульное тестирование

Теперь, когда вы имеете общее представление о том, что такое тестирование и зачем оно нужно, пришло время познакомить вас с лучшим другом разработчика — *модульным тестом*.

Прежде чем переходить к примерам, хотим вас предостеречь. В этой главе мы постараемся дать фундаментальные основы модульного тестирования, но не будем строго следовать какой-либо конкретной школе или методологии. За годы практики мы перепробовали множество подходов и в итоге выработали собственный стиль, который продолжает меняться и развиваться.

Или, как сказал бы Брюс Ли:

> «Впитай то, что полезно, отбрось то, что бесполезно, и добавь то, что по-настоящему твое».

Разработка модульного теста

Модульные тесты получили свое название потому, что применяются для проверки небольших фрагментов кода. Чтобы объяснить, как пишется такой тест, рассмотрим простой код:

```
# data.py
def get_clean_data(source):

    data = load_data(source)
    cleaned_data = clean_data(data)

    return cleaned_data
```

Функция `get_clean_data()` отвечает за получение данных из источника, их очистку и возврат результата вызывающей стороне. Как протестировать такую функцию?

Один из подходов состоит в следующем. Сначала нужно вызвать `get_clean_data()` и убедиться, что функция `load_data()` была вызвана один раз с аргументом `source`. Затем — что функция `clean_data()` тоже была вызвана один раз с результатом работы `load_data()` в качестве аргумента. И наконец, необходимо убедиться, что возвращаемое значение `get_clean_data()` совпадает с результатом `clean_data()`.

Чтобы все это реализовать, нам нужно подготовить исходный код и запустить его. И здесь возникает потенциальная проблема. Один из золотых принципов модульного тестирования заключается в следующем: *все, что выходит за границы приложения, должно быть заменено mock-объектом*. Нельзя взаимодействовать с реальными источниками данных и не следует вызывать реальные функции, если они обращаются к чему-либо, находящемуся вне нашей системы. Примеры таких внешних зависимостей — база данных, поисковый сервис, внешний API, файловая система.

Такие ограничения нужны, чтобы тесты можно было запускать, не рискуя навредить реальным данным, — это словно щит, защищающий как среду выполнения, так и саму систему.

Еще одна причина, по которой лучше не использовать реальные зависимости в модульных тестах, заключается в том, что воспроизвести полную архитектуру приложения на машине разработчика может быть крайне сложно. Это может потребовать настройки баз данных, API, сторонних сервисов, файлов и каталогов и т. п., а значит, быть трудоемким, отнимающим много времени или даже невозможным процессом.

Если говорить просто, то API — это набор инструментов для создания программных приложений. API описывает программный компонент через его операции, входные и выходные данные, а также используемые типы. Например, если вы пишете программу, которая должна взаимодействовать с поставщиком данных, то, скорее всего, придется обращаться к его API, чтобы получить доступ к нужной информации.

Поэтому в модульных тестах все сторонние подсистемы нужно как-то имитировать. Они должны запускаться на любой машине, не создавая необходимость развертывать полную инфраструктуру.

Однако есть альтернативный подход, который мы особенно ценим, когда его можно применить: вместо использования фиктивных объектов мы предпочитаем использовать специализированные тестовые. Например, если код обращается к базе данных, то вместо того, чтобы подменять все методы и функции, связанные с базой, и вручную программировать их поведение, лучше создать полноценную тестовую БД, настроить нужные таблицы и данные, а затем перенаправить соединение так, чтобы тесты запускались настоящим кодом, но с тестовой базой. Этот подход имеет важное преимущество: если используемая библиотека изменится и это приведет к сбою в вашем коде, то такой тест это обнаружит — он сломается. А вот тест с фиктивными объектами продолжит работать «успешно», ничего не заметив, ведь интерфейс-имитатор не знает, что в библиотеке под ним все изменилось. В таких случаях отлично подходят базы данных в памяти.

Одно из приложений, позволяющих создавать тестовую базу данных, — это Django. В пакете django.test есть множество инструментов, помогающих писать тесты, не симулируя вручную взаимодействие с базой данных. Кроме того, при таком подходе нужно проверять транзакции, кодировки и другие аспекты, связанные с работой с базой, что важно для надежной работы приложения. Еще одно преимущество такого подхода — возможность выявить различия между базами данных, которые могут по-разному обрабатывать одно и то же поведение (например, типы данных, правила транзакций и т. д.).

Однако не всегда можно использовать тестовые объекты. Например, если приложение взаимодействует с внешним API и у него нет тестовой версии, тогда нам приходится симулировать API с помощью фиктивных объектов. На практике чаще всего мы приходим к гибридному подходу: используем тестовые версии технологий, которые это позволяют (например, базы данных в памяти), и применяем фиктивные объекты для всего остального.

Теперь поговорим подробнее о фиктивных объектах.

Mock-объекты и подмена

Прежде всего, в Python такие подменные объекты называются *mock-объектами*. До версии 3.3 библиотека `mock` была сторонней и устанавливалась через `pip` практически в каждом проекте. Начиная с версии 3.3 она вошла в стандартную библиотеку как часть модуля `unittest` — и заслуженно, учитывая ее важность и широкое применение.

Процесс подмены реального объекта, функции или другой структуры данных на mock-объект называется *подменой*. Библиотека предоставляет инструмент `patch`, который можно использовать как декоратор функции или класса, а также как контекстный менеджер — для временной подмены зависимостей в рамках теста.

Утверждения

Фаза проверки результатов в тесте выполняется с помощью *утверждений*. Обычно утверждение — это функция или метод, который проверяет, выполнено ли то или иное условие (чаще всего — равенство объектов). Если условие не выполняется, то утверждение выбрасывает исключение и тест считается проваленным. Полный список утверждений можно найти в документации модуля `unittest`. Однако при использовании `pytest`, как правило, применяется универсальный оператор `assert`, который делает код проще и понятнее.

Тестирование генератора CSV

Теперь перейдем к практическому примеру. Мы покажем, как протестировать небольшой фрагмент кода, и по ходу разберем ключевые понятия модульного тестирования, используя этот пример как контекст.

Наша задача — написать функцию `export()`, которая получает список словарей, каждый из которых представляет пользователя. Она должна создать файл в формате *CSV* (comma-separated values — «значения, разделенные запятыми»), добавить заголовок, а затем внести всех пользователей, которые считаются допустимыми по определенным правилам. Функция будет принимать три параметра: список словарей пользователей; имя CSV-файла, который нужно создать; указание, можно ли перезаписать файл, если он уже существует.

Чтобы считаться допустимым и быть добавленным в итоговый файл, словарь пользователя должен соответствовать следующим условиям: в нем обязательно должны быть указаны электронная почта, имя и возраст; поле с ролью — необязательное; адрес электронной почты должен быть допустимым; имя не должно быть пустым; возраст должен быть целым числом от 18 до 65 включительно.

Такова наша задача. Сейчас мы покажем реализацию функции, а затем проанализируем тесты, которые написали для нее. Но сначала — важное уточнение: в приведенных ниже примерах кода используются две сторонние библиотеки — `marshmallow` и `pytest`. Обе они указаны в зависимостях для кода к текущей главе, поэтому убедитесь, что они установлены через `pip`.

Библиотека `marshmallow` (https://marshmallow.readthedocs.io) позволяет сериализовать (или, как это называется в терминологии marshmallow, *dump*) и десериализовать *(load)* объекты. Но что еще важнее — она позволяет определить схему, с помощью которой можно проверять словари пользователей на соответствие требованиям. Позднее, в главе 14, мы рассмотрим еще одну библиотеку для описания схем — `pydantic`.

В свою очередь, библиотека `pytest` (https://docs.pytest.org) — один из лучших инструментов для тестирования, которые мы когда-либо видели. Она используется повсеместно и фактически вытеснила другие библиотеки, такие как `nose`. В `pytest` доступны удобные инструменты, позволяющие писать тесты быстро, лаконично и эффективно.

Теперь перейдем к коду. Мы назвали файл `api.py` просто потому, что он содержит функцию, которую можно использовать для экспорта CSV. Мы покажем код по частям:

```
# api.py
from pathlib import Path
import csv
from copy import deepcopy

from marshmallow import Schema, fields, pre_load
from marshmallow.validate import Length, Range

class UserSchema(Schema):
    """Представляет *валидного* пользователя."""

    email = fields.Email(required=True)
    name = fields.Str(required=True, validate=Length(min=1))
    age = fields.Int(
        required=True, validate=Range(min=18, max=65)
    )
    role = fields.Str()

    @pre_load()
    def strip_name(self, data, **kwargs):
        data_copy = deepcopy(data)
```

```
        try:
            data_copy["name"] = data_copy["name"].strip()
        except (AttributeError, KeyError, TypeError):
            pass
        return data_copy

schema = UserSchema()
```

В первой части импортируются все необходимые компоненты (`Path`, `csv`, `deepcopy`, а также некоторые инструменты из `marshmallow`), а затем определяется схема для пользователей. Класс схемы наследуется от `marshmallow.Schema`, после чего задаются четыре поля. Здесь используются два строковых поля (`Str`), одно поле электронной почты (`Email`) и одно целочисленное (`Int`). Эти поля уже обеспечивают базовую валидацию, встроенную в `marshmallow`. Обратите внимание, что у поля `role` отсутствует параметр `required=True`.

Тем не менее требуется добавить немного пользовательского кода. Необходимо реализовать проверку для поля `age`, чтобы убедиться, что значение попадает в нужный диапазон. В противном случае `marshmallow` выбросит исключение `ValidationError`. Кроме того, библиотека сама по себе не допустит передачу значения, не являющегося целым числом.

Дополнительная проверка добавляется и для поля `name`, поскольку наличие ключа `name` в словаре еще не означает, что его значение непустое. Выполняется проверка, что длина строки составляет как минимум один символ. Обратите внимание, что для поля `email` дополнительных проверок не требуется — `marshmallow` выполняет валидацию самостоятельно.

После объявления полей добавляется еще один метод — `strip_name()`, помеченный декоратором `pre_load()` из библиотеки `marshmallow`. Он выполняется до того, как `marshmallow` десериализует (загружает) данные. Как вы можете видеть, сначала создается копия `data`, поскольку в таком контексте не рекомендуется работать напрямую с изменяемыми объектами. Затем из значения по ключу `name` удаляются пробелы в начале и конце строки. Этот ключ соответствует полю `name`, которое было объявлено выше. Обработка завершается внутри блока `try/except`, чтобы десериализация могла выполняться корректно даже в случае ошибок. Метод возвращает модифицированную копию данных, а остальную работу берет на себя `marshmallow`.

Затем создается экземпляр схемы, чтобы можно было использовать его для валидации данных.

Теперь напишем функцию `export()`:

```
# api.py
def export(filename, users, overwrite=True):
    """Экспортировать CSV-файл.

    Создает CSV-файл и заполняет его валидными пользователями. Если параметр
    overwrite равен False и файл уже существует, то вызывается ошибка IOError.
    """
```

```
    if not overwrite and Path(filename).is_file():
        raise IOError(f"'{filename}' already exists.")

    valid_users = get_valid_users(users)
    write_csv(filename, valid_users)
```

Как видите, логика функции довольно простая. Если параметр `overwrite` равен `False` и файл с указанным именем уже есть, то выбрасывается исключение `IOError` с сообщением о том, что файл уже существует. В противном случае, если можно продолжать, из исходного списка выбираются допустимые пользователи и передаются в функцию `write_csv()`, которая и выполняет основную работу. Теперь посмотрим, как определены все эти функции:

```
# api.py
def get_valid_users(users):
    """Возвращает по одному валидному пользователю из списка users."""
    yield from filter(is_valid, users)

def is_valid(user):
    """Проверяет, является ли пользователь валидным."""
    return not schema.validate(user)
```

Функция `get_valid_users()` реализована как генератор, поскольку нет необходимости заранее создавать потенциально большой список перед записью в файл. Пользователей можно проверять и записывать по одному. Функция `is_valid()` просто делегирует проверку методу `schema.validate()` из `marshmallow`. Он возвращает словарь: если данные соответствуют схеме — он пустой, если нет — содержит информацию об ошибках. В рамках данной задачи собирать информацию об ошибках не требуется, поэтому мы ее просто игнорируем. Функция `is_valid()` возвращает `True`, если `schema.validate()` вернул пустой словарь, и `False` — в противном случае.

Заключительный фрагмент кода в этом модуле выглядит так:

```
# api.py
def write_csv(filename, users):
    """Записать CSV-файл с заданным именем и списком пользователей.

    Пользователи считаются валидными для структуры этого CSV.
    """
    fieldnames = ["email", "name", "age", "role"]

    with open(filename, "w", newline="") as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(users)
```

Опять-таки логика здесь проста. Сначала определяется заголовок CSV-файла в `fieldnames`, затем файл `filename` открывается для записи. Параметр `newline=""` указан в соответствии с рекомендациями документации по работе с CSV-файлами.

После создания файла создается объект `writer` с помощью класса `csv.DictWriter`. Этот инструмент сопоставляет поля словаря пользователя с именами полей, поэтому нам не нужно заботиться о порядке колонок.

Сначала записывается заголовок, затем все пользователи добавляются по одному. Обратите внимание: функция предполагает, что ей передан список уже проверенных пользователей. Если это не так, то поведение может быть некорректным (например, при наличии лишних полей в словарях пользователей — с текущими настройками код сломается).

Этот код стоит держать в уме. Рекомендуем потратить немного времени и просмотреть его еще раз. Запоминать его не обязательно — небольшие вспомогательные функции с понятными именами облегчают отслеживание того, как организовано тестирование.

Теперь переходим к самой интересной части — тестированию функции `export()`. Как и раньше, мы будем показывать код по частям:

```
# tests/test_api.py
import re
from unittest.mock import patch, mock_open, call
import pytest
from api import is_valid, export, write_csv
```

Начнем с операторов импорта: сначала импортируется модуль `re` из стандартной библиотеки, так как он потребуется в одном из тестов. Затем подключаются некоторые инструменты из модуля `unittest.mock`, после этого — `pytest`, и, наконец, импортируются три функции, которые мы собираемся тестировать: `is_valid()`, `export()` и `write_csv()`.

Прежде чем писать тесты, нужно определить несколько фикстур. Как вы увидите, фикстура в `pytest` — это функция, помеченная декоратором `@pytest.fixture`. Фикстуры выполняются перед каждым тестом, к которому применяются. Обычно от фикстуры ожидается, что она вернет какое-либо значение, которое затем можно будет использовать в тесте. У нас есть определенные требования к словарю пользователя, поэтому напишем две фикстуры: одну с минимально необходимыми полями, другую — с полным набором данных. Обе фикстуры должны возвращать допустимые (валидные) словари пользователей. Вот код:

```
# tests/test_api.py
@pytest.fixture
def min_user():
    """Представляет валидного пользователя с минимальным набором данных."""
    return {
        "email": "minimal@example.com",
        "name": "Primus Minimus",
        "age": 18,
    }
```

```
@pytest.fixture
def full_user():
    """Представляет валидного пользователя с полным набором данных."""
    return {
        "email": "full@example.com",
        "name": "Maximus Plenus",
        "age": 65,
        "role": "emperor",
    }
```

В данном примере единственное различие между пользователями — наличие ключа `role`, но этого достаточно, чтобы продемонстрировать суть.

Обратите внимание: вместо того чтобы просто объявить словари на уровне модуля, мы написали две функции, возвращающие словарь, и пометили их декоратором `@pytest.fixture`. Причина в том, что если вы объявите словарь, предназначенный для использования в тестах, на уровне модуля, то вам нужно будет обязательно копировать его в начале каждого теста. Если этого не делать и какой-либо тест (или код, который он проверяет) изменит словарь, то все последующие тесты окажутся под угрозой: словарь уже не будет в исходном состоянии. Благодаря фикстурам `pytest` предоставляет новый словарь для каждого теста, поэтому необходимость в ручном копировании отпадает. Это помогает соблюдать принцип независимости, согласно которому каждый тест должен быть самодостаточным и изолированным.

Кроме того, фикстуры *компонуемы*, то есть одну фикстуру можно использовать внутри другой — это полезная особенность `pytest`. Чтобы вы могли это увидеть, напишем фикстуру для списка пользователей. В него войдут два уже созданных пользователя, а также третий, не прошедший валидацию, — у него будет отсутствовать поле `age`. Взгляните на следующий фрагмент кода:

```
# tests/test_api.py
@pytest.fixture
def users(min_user, full_user):
    """Список пользователей: два валидных и один невалидный."""
    bad_user = {
        "email": "invalid@example.com",
        "name": "Horribilis",
    }
    return [min_user, bad_user, full_user]
```

Теперь у нас есть два пользователя, которых можно использовать по отдельности, а также список из трех пользователей.

Первые несколько тестов будут посвящены проверке валидации пользователя. Мы сгруппируем все тесты этой категории в пределах одного класса. Так мы сможем выделить для них общее пространство имен — своего рода «дом». Кроме того, как мы увидим позже, это даст возможность определить фикстуры на уровне класса, доступные только для тестов внутри него. Одно из преимуществ фикстур

класса — возможность переопределения: если за пределами класса есть фикстура с тем же именем, то ее поведение можно заменить в пределах класса.

В данном случае мы организовали тесты в виде классов (так было удобнее), но вы можете просто размещать тестовые функции на уровне модуля — `pytest` предоставляет широкие возможности для выбора структуры.

Кроме того, по мере знакомства с примерами вы заметите, что имена всех тестовых функций начинаются с `test_`, а имена всех тестовых классов — с `Test`. Такое различие нужно для того, чтобы `pytest` смог обнаружить эти элементы и распознать их как тесты. Более подробную информацию об этом можно найти в официальной документации `pytest`.

А теперь вернемся к коду. Взгляните на следующий фрагмент:

```
# tests/test_api.py
class TestIsValid:
    """Проверка, как код определяет валидность пользователя."""

    def test_minimal(self, min_user):
        assert is_valid(min_user)

    def test_full(self, full_user):
        assert is_valid(full_user)
```

Начнем с самого простого: убедимся, что наши фикстуры действительно проходят валидацию. Такой шаг помогает проверить, корректно ли код валидирует пользователей, в том числе и тех, данные которых минимальны или максимально полные. Обратите внимание: каждой тестовой функции передается параметр, совпадающий с именем фикстуры. Благодаря этому `pytest` активирует соответствующую фикстуру и передает возвращаемое значение в качестве аргумента в функцию теста.

Прежде чем двигаться дальше, полезно будет запустить эти два теста — убедиться, что все настроено правильно и работает должным образом. Для запуска тестов выполните в терминале команду `pytest`, перейдя в папку `ch10`:

```
$ pytest tests -vv
===================== test session starts =====================
platform darwin -- Python 3.12.2, pytest-8.1.1, pluggy-1.4.0 --
  /Users/fab/.virtualenvs/lpp4ed-ch10/bin/python
cachedir: .pytest_cache
rootdir: /Users/fab/code/lpp4ed
configfile: pyproject.toml
collected 2 items

tests/test_api.py::TestIsValid::test_minimal PASSED      [ 50%]
tests/test_api.py::TestIsValid::test_full PASSED         [100%]
====================== 2 passed in 0.03s ======================
```

Здесь мы дали `pytest` указание искать тесты в папке `tests`. Чтобы увидеть подробный вывод, мы добавили флаг `-vv`.

После вывода служебной информации вы увидите строки, аналогичные тем, которые мы выделили. Они содержат полный путь к каждому выполненному тесту: сначала имя модуля, в котором находятся тесты, затем — имя класса, если тест находится в классе, и, наконец, имя самого теста.

Справа отображается процент выполнения набора тестов. У нас пока только два теста, поэтому после первого прогресс составит 50 %, а после второго — 100 %. Оба теста прошли успешно.

Если какой-либо тест завершится с ошибкой, то `pytest` выведет сообщение о ней, а также отладочную информацию — это поможет разобраться, что произошло, и внести исправления.

Теперь смоделируем сбой: удалим ключ `name` из фикстуры `min_user` и снова запустим тесты:

```
$ pytest tests -vv
===================== test session starts =====================
platform darwin -- Python 3.12.2, pytest-8.1.1, pluggy-1.4.0 --
/Users/fab/.virtualenvs/lpp4ed-ch10/bin/python
cachedir: .pytest_cache
rootdir: /Users/fab/code/lpp4ed
configfile: pyproject.toml
collected 2 items

tests/test_api.py::TestIsValid::test_minimal FAILED      [ 50%]
tests/test_api.py::TestIsValid::test_full PASSED         [100%]
=========================== FAILURES ===========================
_____________________ TestIsValid.test_minimal _____________________

self = <ch10.tests.test_api.TestIsValid object at 0x103603920>,
       min_user = {'age': 18, 'email': 'minimal@example.com'}

    def test_minimal(self, min_user):
>       assert is_valid(min_user)
E       AssertionError: assert False
E        + where False = is_valid(
              {'age': 18, 'email': 'minimal@example.com'}
          )

tests/test_api.py:45: AssertionError
=================== short test summary info ===================
FAILED tests/test_api.py::TestIsValid::test_minimal
       - AssertionError: assert False
================ 1 failed, 1 passed in 0.04s =================
```

Как видно в отмеченных строках, `pytest` сообщает, какие именно тесты завершились с ошибкой, а также показывает фрагмент кода, в котором возникла ошибка, чтобы вы могли проанализировать ситуацию и понять, в чем проблема. Слева от строки с ошибкой стоит символ `>` — он указывает на ту строку, где было выброшено исключение. Под ней вы найдете две строки с описанием ошибки. В нашем случае это сообщение о том, что `{'age': 18, 'email': 'minimal@example.com'}` не является допустимым пользователем.

Теперь, зная, как запускать тесты, вы можете делать это в любое время. Рекомендуется запускать тесты так, чтобы убедиться, что они действительно отлавливают ошибки. Поэтому можете поэкспериментировать с фикстурами и утверждениями, внося изменения, которые вызовут предсказуемый сбой.

А теперь вернемся к нашему тестовому набору. Следующая цель — проверить поле `age`. Для этого мы воспользуемся механизмом параметризации.

Параметризация — прием, который дает возможность запускать один и тот же тест несколько раз, подставляя в него разные входные данные. Это очень удобно, поскольку можно написать тест всего раз, не дублируя код, а `pytest` сам позаботится о том, чтобы запустить его с разными наборами данных и обработать каждый случай отдельно. Таким образом, вы получите отдельные, понятные сообщения об ошибках, если хотя бы один из параметров вызовет сбой.

В качестве альтернативы можно было бы написать один тест с циклом `for`, в котором вы перебираете все нужные значения вручную. Но такой подход значительно уступает параметризации, и вот почему: фреймворк не сможет точно указать, на каком наборе данных произошла ошибка; как только произойдет сбой, остальные итерации цикла не выполнятся — вы не узнаете, как вели бы себя другие данные; логика внутри теста усложнится и с точки зрения поддержки, и с точки зрения читабельности. Поэтому в таких ситуациях лучше выбирать параметризацию. Вдобавок благодаря ей можно не писать множество почти одинаковых тестов, чтобы охватить все возможные сценарии.

Теперь посмотрим, как мы тестируем поле `age`. Для вашего удобства мы повторим заголовок класса, но опустим уже рассмотренные тесты:

```
# tests/test_api.py
class TestIsValid:
    …
    @pytest.mark.parametrize("age", range(18))
    def test_invalid_age_too_young(self, age, min_user):
        min_user["age"] = age
        assert not is_valid(min_user)
```

Сначала мы напишем тест, проверяющий, что валидация не проходит, если пользователь слишком молод. Согласно нашим правилам «слишком молодой» — это значит младше 18 лет. Мы проверим все значения от 0 до 17, воспользовавшись функцией `range()`.

Если вы посмотрите, как работает параметризация, то увидите, что мы указываем имя параметра — `age`, а затем перечисляем значения, которые он должен принимать. Тест будет выполнен один раз для каждого из этих значений. В данном случае это все целые числа от 0 до 17 включительно, то есть результат `range(18)`. Вдобавок мы добавляем параметр `age` в саму функцию теста — `pytest` подставит туда нужное значение при каждом запуске.

Кроме того, в этом тесте мы используем фикстуру `min_user()`. Мы изменяем значение `age` в возвращаемом словаре `min_user`, а затем убеждаемся, что результат вызова `is_valid(min_user)` равен `False`. Для этого в `pytest` мы используем обычный `assert`, утверждая, что выражение ложно: `assert not is_valid(min_user)` — это означает, что `is_valid()` должен вернуть `False`. Если все работает правильно, то тест пройдет. Если нет — тест завершится с ошибкой.

Обратите внимание, что pytest будет повторно вызывать функцию фикстуры при каждом запуске теста, в котором она используется. Поэтому вы можете свободно изменять данные, возвращаемые фикстурой, внутри теста — это никак не повлияет на другие тесты.

Продолжим и добавим все оставшиеся тесты, которые выявляют, что значение `age` не проходит валидацию:

```
# tests/test_api.py
class TestIsValid:
    ...
    @pytest.mark.parametrize("age", range(66, 100))
    def test_invalid_age_too_old(self, age, min_user):
        min_user["age"] = age
        assert not is_valid(min_user)

    @pytest.mark.parametrize("age", ["NaN", 3.1415, None])
    def test_invalid_age_wrong_type(self, age, min_user):
        min_user["age"] = age
        assert not is_valid(min_user)
```

Здесь еще два теста. Один проверяет верхнюю границу допустимого диапазона — от 66 до 99 лет. Второй убеждается, что `age` недопустим, если это не целое число. Мы передаем туда строку, число с плавающей запятой и значение `None`, чтобы убедиться, что схема их отвергнет. Как видите, структура всех этих тестов одинаковая. Благодаря параметризации мы просто подставляем в них разные входные значения.

Мы завершили проверку недопустимых значений возраста, поэтому добавим тест, проверяющий, что валидация проходит для допустимых значений:

```
# tests/test_api.py
class TestIsValid:
    ...
    @pytest.mark.parametrize("age", range(18, 66))
    def test_valid_age(self, age, min_user):
        min_user["age"] = age
        assert is_valid(min_user)
```

Все так же просто. Мы передаем корректный диапазон значений — от 18 до 65 — и убираем `not` в выражении `assert`.

На этом проверку возраста можно считать завершенной. Теперь перейдем к тестам обязательных полей:

```
# tests/test_api.py
class TestIsValid:
    ...
    @pytest.mark.parametrize("field", ["email", "name", "age"])
    def test_mandatory_fields(self, field, min_user):
        del min_user[field]
        assert not is_valid(min_user)

    @pytest.mark.parametrize("field", ["email", "name", "age"])
    def test_mandatory_fields_empty(self, field, min_user):
        min_user[field] = ""
        assert not is_valid(min_user)

    def test_name_whitespace_only(self, min_user):
        min_user["name"] = " \n\t"
        assert not is_valid(min_user)
```

Эти три теста по-прежнему относятся к тому же классу. Первый проверяет, считается ли пользователь недопустимым, если отсутствует одно из обязательных полей. Напомним, что при каждом запуске теста фикстура `min_user` создается заново, поэтому мы удаляем только одно поле за раз — это корректный способ проверять обязательность полей. В теле теста просто удаляется ключ, имя которого передается как параметр. Как видно в первом тесте, параметризация охватывает все обязательные поля: `email`, `name` и `age`.

Второй тест устроен немного иначе. Вместо удаления ключей мы присваиваем им пустую строку — по одному полю за раз. В третьем тесте проверяется случай, когда имя (`name`) состоит только из пробельных символов.

Предыдущие тесты охватывают ситуации, когда обязательные поля либо отсутствуют, либо присутствуют, но пусты, а также касаются форматирования поля `name`. Теперь напишем последние два теста в этом классе. Они проверяют, во-первых, корректность адреса электронной почты, а во-вторых, типы данных для `email`, `name` и `role`:

```
# tests/test_api.py
class TestIsValid:
    ...

    @pytest.mark.parametrize(
        ("email", "outcome"),
        [
            ("missing_at.com", False),
            ("@missing_start.com", False),
            ("missing_end@", False),
            ("missing_dot@example", False),
            ("good.one@example.com", True),
            ("δοκιμή@παράδειγμα.δοκιμή", True),
            ("аджай@экзампл.рус", True),
        ],
    )
```

```
    def test_email(self, email, outcome, min_user):
        """Проверка различных форматов адресов электронной почты."""
        min_user["email"] = email
        assert is_valid(min_user) == outcome
```

На этот раз параметризация более сложная. Мы объявляем два параметра — `email` и `outcome` — и передаем в декоратор список кортежей, а не просто значений. При каждом запуске теста один из таких кортежей будет распакован: значение первого элемента попадет в `email`, второго — в `outcome`. Благодаря этому мы можем написать один тест сразу для допустимых и недопустимых адресов электронной почты, вместо того чтобы создавать два отдельных. В каждом случае мы указываем адрес электронной почты и ожидаемый результат валидации. Первые четыре — заведомо некорректные адреса, последние три — корректные. В том числе приведены примеры с символами, не относящимися к ASCII, — важный шаг, позволяющий не забывать о пользователях по всему миру.

Валидация проверяется просто: утверждается, что результат вызова `is_valid()` должен совпасть с ожидаемым значением `outcome`.

Теперь напишем еще один короткий тест, который проверяет, действительно ли валидация завершается с ошибкой, если в поля переданы значения неподходящего типа (напомним, что поле `age` мы уже тестировали отдельно):

```
# tests/test_api.py
class TestIsValid:
    ...
    @pytest.mark.parametrize(
        ("field", "value"),
        [
            ("email", None),
            ("email", 3.1415),
            ("email", {}),
            ("name", None),
            ("name", 3.1415),
            ("name", {}),
            ("role", None),
            ("role", 3.1415),
            ("role", {}),
        ],
    )
    def test_invalid_types(self, field, value, min_user):
        """Проверка недопустимых типов данных для полей."""
        min_user[field] = value
        assert not is_valid(min_user)
```

Как и в случае с полем `age`, мы передаем три различных значения, ни одно из которых не является строкой. Такой тест можно было бы дополнить и другими примерами, но, по правде говоря, в обычной практике делать это нет необходимости. Мы добавили его лишь затем, чтобы показать, какие возможности предоставляет `pytest`, хотя на деле чаще сосредотачиваются на проверке допустимых типов: достаточно убедиться, что поддерживаются все корректные варианты.

Прежде чем перейти к следующему классу тестов, подробнее остановимся на теме, которую мы затронули вскользь, когда писали тесты для возраста.

Границы и степень детализации

Тестируя поле возраста, мы написали три отдельных теста для трех диапазонов: от 0 до 17 (ожидается неудача), от 18 до 65 (успех) и от 66 до 99 (снова неудача). Почему именно так? Все дело в том, что в правилах проверки фигурируют две граничные точки: 18 и 65. Поэтому логично сосредоточить внимание на трех участках, которые эти границы задают: до 18, между 18 и 65 включительно и после 65. Каким способом вы это реализуете — не так важно, главное, чтобы граничные условия были протестированы. Это означает, что если кто-то изменит проверку в схеме, например, с `18 <= значение <= 65` на `18 <= значение < 65` (обратите внимание: второе `<=` стало `<`), то должен быть тест, который провалится при значении 65.

Такой подход называется *проверкой границ*, и очень важно уметь распознавать эти границы в своем коде, чтобы проверять его поведение именно в этих критических точках.

Еще один важный момент — это вопрос о степени детализации: насколько близко нужно подступиться к границе? Иными словами, какую единицу использовать при подходе к ней?

В случае с возрастом мы имели дело с целыми числами, поэтому единица измерения 1 оказалась идеальным выбором. Именно поэтому в тестах мы использовали такие значения, как 16, 17, 18, 19, 20 и т. д. Но что если бы вы проверяли значение временной метки? В таком случае подходящая степень детализации будет другой. Если код должен вести себя по-разному в зависимости от временной метки и она измеряется в секундах, то и тесты должны работать с точностью до секунд. Если временная метка выражает годы, то стоит использовать годы как единицу измерения. Надеемся, суть ясна. Такой подход характеризуется *степенью детализации* и обязательно должен сочетаться с понятием границ: правильно подобрав уровень детализации и протестировав поведение вблизи границ, вы удостоверитесь в надежности тестов и избежите случайных пропусков.

Теперь продолжим наш пример и перейдем к тестированию функции `export()`.

Тестирование функции export

В этом же тестовом модуле мы добавили еще один класс — он объединяет тесты для функции `export()`:

```
# tests/test_api.py
class TestExport:
    """Проверка поведения функции export."""
```

```
@pytest.fixture
def csv_file(self, tmp_path):
    """Подготовить имя файла во временной папке.

    Благодаря механизму фикстуры tmp_path в pytest
    сам файл еще не существует.
    """
    csv_path = tmp_path / "out.csv"
    yield csv_path
    csv_path.unlink(missing_ok=True)

@pytest.fixture
def existing_file(self, tmp_path):
    """Создать временный файл и записать в него содержимое."""
    existing = tmp_path / "existing.csv"
    existing.write_text("Please leave me alone...")
    return existing
```

Начнем с фикстур. На этот раз они определены на уровне класса, поскольку используются только внутри него. Выносить их на уровень модуля, как мы делали это с фикстурами пользователей, было бы избыточно.

Нам понадобятся два файла. В начале главы мы обсуждали, что при взаимодействии с базой данных, диском, сетью и подобными структурами лучше подменять все с помощью mock-объектов. Но по возможности лучше обойтись без них. Здесь мы используем временные каталоги, которые создаются и удаляются в рамках фикстуры. Это надежный и удобный способ, и он подходит лучше, чем mock-объекты. Для создания временных папок мы воспользуемся встроенной фикстурой `tmp_path` из `pytest`. Она возвращает объект `pathlib.Path`.

Первая фикстура — `csv_file()` — дает путь к временной папке. Все, что происходит до оператора `yield`, можно рассматривать как этап подготовки (setup). Данные, предоставляемые этой фикстурой, — это путь к файлу. Сам файл к текущему моменту еще не существует. Когда тест запускается, фикстура возвращает путь к нему, а после завершения теста выполняется все, что написано после `yield`, — это этап очистки (teardown).

Этап после `yield` в фикстуре можно считать завершающим — это очистка. В случае `csv_file()` это всего лишь вызов `csv_path.unlink()`, необходимый для удаления созданного файла `.csv` (если тот появился). В фикстурах можно закладывать гораздо больше логики на каждом этапе, и по мере обретения опыта вы начнете легко выстраивать как подготовку, так и завершение. Это быстро становится привычным.

На самом деле удалять файл после каждого теста не обязательно. Фикстура tmp_path создает новую временную папку для каждого теста, так что файлы, созданные в одном тесте, никак не повлияют на другие. Мы удаляем файл в этой фикстуре лишь для того, чтобы показать, как работает yield в pytest.

Вторая фикстура — `existing_file()` — почти такая же, как первая. Ее задача — подготовить файл с содержимым, чтобы можно было проверить, остается ли он при вызове `export()` с параметром `overwrite=False` нетронутым. Мы создаем файл во временной папке и записываем в него какой-нибудь текст, чтобы позже убедиться: в результате экспорта ничего не перезаписалось.

Теперь перейдем к самим тестам. Как и раньше, мы приведем определение класса, но опустим тесты, которые уже обсуждали выше:

```
# tests/test_api.py
class TestExport:
    ...
    def test_export(self, users, csv_file):
        export(csv_file, users)
        text = csv_file.read_text()

        assert (
            "email,name,age,role\n"
            "minimal@example.com,Primus Minimus,18,\n"
            "full@example.com,Maximus Plenus,65,emperor\n"
        ) == text
```

Один из тестов использует фикстуры `users()` и `csv_file()` и сразу вызывает `export()` с этими аргументами. Мы ожидаем, что файл будет создан и в него попадут два валидных пользователя (напомним: в списке три пользователя, но один из них невалидный).

Чтобы проверить результат, мы открываем файл и читаем его содержимое как строку. Затем сравниваем ее с ожидаемым содержимым — заголовком и данными двух корректных пользователей — строго в нужном порядке.

Теперь добавим еще один тест, чтобы убедиться вот в чем: если в одном из значений встречается запятая, то файл CSV все равно формируется корректно. Поскольку речь идет о CSV, важно проверить, что запятая внутри данных не нарушает структуру файла:

```
# tests/test_api.py
class TestExport:
    ...
    def test_export_quoting(self, min_user, csv_file):
        min_user["name"] = "A name, with a comma"
        export(csv_file, [min_user])
        text = csv_file.read_text()

        assert (
            "email,name,age,role\n"
            'minimal@example.com,"A name, with a comma",18,\n'
        ) == text
```

Для этого не нужен весь список пользователей — достаточно одного. Мы уже проверили, как работает экспорт при полном списке, а сейчас концентрируемся

на конкретной ситуации. Всегда старайтесь минимизировать объем действий внутри теста.

Итак, мы берем словарь `min_user()` и добавляем запятую в поле `name`. Далее повторяем ту же процедуру, что и в предыдущем тесте: экспортируем данные, читаем файл и сравниваем содержимое. Мы хотим убедиться, что имя, содержащее запятую, оказалось обернуто в двойные кавычки — это стандартный способ экранирования значений в CSV, и любой корректный парсер должен интерпретировать их правильно, не разбивая строку на части по запятой.

А теперь напишем еще один тест, позволяющий убедиться, что при наличии файла и параметре `overwrite=False` экспорт не перезапишет его:

```
# tests/test_api.py
class TestExport:
    ...
    def test_does_not_overwrite(self, users, existing_file):
        with pytest.raises(IOError) as err:
            export(existing_file, users, overwrite=False)

        err.match(
            r"'{}' already exists\.".format(
                re.escape(str(existing_file))
            )
        )

        # Кроме того, убедимся, что содержимое файла осталось без изменений
        assert existing_file.read_text() == (
            "Please leave me alone..."
        )
```

Этот тест весьма интересен, поскольку на его примере можно показать, как в `pytest` задать ожидание, что функция должна сгенерировать исключение. Мы используем контекстный менеджер `raises()` — он входит в состав `pytest`. В него передается тип ожидаемого исключения, а в теле блока выполняется вызов, который должен его сгенерировать. Если исключение не возникнет, то тест завершится с ошибкой.

Однако на этом мы не останавливаемся. Мы проверяем еще и текст исключения с помощью удобного помощника `err.match()`. Обратите внимание, что использовать `assert` в этом случае не нужно: если переданная строка не совпадает с сообщением об ошибке, то `match()` сам вызовет `AssertionError` и тест провалится. Еще один важный момент: путь к файлу `existing_file` приходится экранировать, поскольку в Windows он содержит обратные косые черты (`\`), которые могут быть интерпретированы как спецсимволы в регулярном выражении, передаваемом в `err.match()`[1]. Мы обрабатываем это соответствующим образом, чтобы проверка прошла как нужно.

[1] Обратите внимание, что match() принимает именно регулярное выражение, а не строку для сравнения. — *Примеч. науч. ред.*

В завершение мы читаем содержимое файла и сверяем его с исходной строкой, которую поместили туда при создании фикстуры. Благодаря этому мы можем быть уверены, что файл не был изменен, как и предполагалось.

Финальные штрихи

Прежде чем переходить к следующей теме, стоит подвести некоторые итоги.

Возможно, вы заметили, что мы не стали писать тесты для всех функций, которые реализовали. В частности, мы не тестировали `get_valid_users()`, `validate()` и `write_csv()`. Причина в том, что эти функции уже косвенно охвачены нашей тестовой системой. Мы тщательно протестировали `is_valid()` и `export()`, а этого вполне достаточно, чтобы убедиться: схема правильно проверяет пользователей, а функция экспорта отфильтровывает недопустимых пользователей, не перезаписывает файлы без разрешения и сохраняет данные в корректном формате CSV. Непротестированные функции — это внутренние части системы. Они лишь участвуют в логике, которая уже была протестирована на более высоком уровне.

Так хорошо ли было бы написать отдельные тесты и для этих функций? На этот вопрос сложно дать однозначный ответ.

С одной стороны, чем больше у нас тестов, тем меньше возможностей для рефакторинга. В текущем виде мы, например, можем безболезненно переименовать `validate()`, и ни один тест не потребует изменений. И это логично: если эта функция корректно выполняет проверку данных в рамках `get_valid_users()`, то нам неважно, как она называется и как реализована. Тесты проверяют не реализацию, а поведение всей системы. Таким образом, важно различать тестирование внутренней реализации и тестирование поведения системы снаружи. В нашем случае приоритет был отдан второму подходу, поскольку он открывает больше возможностей для изменения кода.

Если бы мы, наоборот, написали тесты для функции `validate()`, то нам бы пришлось их переделывать при любом изменении — например, если бы мы переименовали эту функцию или поменяли ее сигнатуру.

Как же поступать правильно? Писать тесты или нет? Единственно верного ответа не существует — это решение всегда остается за вами. Вам нужно найти баланс. Мы, со своей стороны, придерживаемся следующего подхода: все должно быть покрыто тестами либо напрямую, либо косвенно. Мы стараемся составить настолько минимальную тестовую систему, насколько это возможно, при этом обеспечивая полное покрытие. Это позволяет поддерживать тесты, не создавая лишнюю нагрузку.

Надеемся, пример был вам понятен — он дал нам повод обсудить несколько важных тем.

Если вы заглянете в исходный код к этой книге, в модуль `test_api.py`, то увидите еще несколько тестовых классов. Они демонстрируют, как могли бы выглядеть тесты, если бы мы решили использовать только mock-объекты.

Обязательно изучите этот код — он простой и даст хорошее представление о различиях между подходами.

А теперь запустим всю тестовую систему.

```
$ pytest tests
====================== test session starts ======================
 platform darwin -- Python 3.12.2, pytest-8.1.1, pluggy-1.4.0
 rootdir: /Users/fab/code/lpp4ed
 configfile: pyproject.toml
 collected 132 items

 tests/test_api.py ..........................................
 ............................................................
 ......................                                   [100%]

====================== 132 passed in 0.14s ======================
```

Как уже упоминалось, запускать команду `pytest test` нужно из папки `ch10` (если добавить флаг `-vv`, то будет выведен подробный отчет с демонстрацией того, как параметризация влияет на имена тестов). Библиотека `pytest` сканирует файлы и каталоги, ища модули с именами, начинающимися или заканчивающимися на `test_`, например: `test_*.py` или `*_test.py`. В них она определяет функции, начинающиеся на `test_`, или методы с таким префиксом внутри классов, начинающихся на `Test`. (Полные правила описаны в документации библиотеки.) Как видно из вывода, было выполнено 132 теста за 140 миллисекунд и все они прошли успешно. Очень рекомендуем поработать с этим кодом самостоятельно: внесите изменения и проверьте, сломаются ли тесты. Поймите, почему они ломаются (или почему нет).

Проходят ли тесты даже при наличии ошибки в коде? Слишком ли строгие тесты — ломаются ли они при изменениях, которые не влияют на правильность результата? Размышления над такими ситуациями помогут вам глубже понять искусство тестирования.

Кроме того, мы советуем хорошо изучить модуль `unittest` и библиотеку `pytest` — вы будете часто с ними работать.

Разработка через тестирование

Разработка через тестирование (Test-Driven Development, TDD) — методология, которую заново открыл Кент Бек и описал в своей книге *Test-Driven Development by Example*[1]. Обязательно прочтите ее, если вы хотите разобраться в основах этого подхода.

TDD — методика разработки программного обеспечения, основанная на непрерывном повторении очень короткого цикла разработки.

---

[1] *Бек К.* Экстремальное программирование. Разработка через тестирование. — СПб.: Питер, 2020.

Сначала разработчик пишет тест, который проверяет еще не реализованную функцию. Это может быть новая функциональность или некий элемент в поведении программы, который нужно изменить или удалить. Запуск этого теста обязательно завершится сбоем, поскольку нужной логики еще нет. Поэтому данный этап называют *красной фазой*.

Затем пишется минимальный код, чтобы пройти этот тест. Как только тест начинает выполняться успешно, наступает *зеленая фаза*. На данном этапе допускается даже чит-код — важно, чтобы он прошел тест. Такой прием известен как *«притворяйся, пока не получится»*.

На следующем шаге в цикл добавляют новые тесты, особенно на пограничные случаи. Если код был «читерским», то, скорее всего, не пройдет все тесты одновременно, и разработчику придется написать настоящую логику, чтобы они проходили. Этот процесс уточнения поведения за счет добавления новых кейсов часто называют *триангуляцией*.

Последний этап цикла заключается в том, что разработчик приводит в порядок как код, так и тесты, добиваясь желаемого уровня чистоты и читабельности. Этот этап называется *рефакторингом*.

Таким образом, мантра TDD звучит как *«красный — зеленый — рефакторинг»*.

Сначала может показаться странным писать тесты до кода. Признаемся, нам самим потребовалось время, чтобы привыкнуть к такому подходу. Но если вы будете настойчиво придерживаться его, то постепенно произойдет почти волшебное изменение: вы начнете замечать, как качество вашего кода растет — и так, как в другом случае вряд ли бы получилось.

Когда мы пишем код до тестов, нам приходится одновременно думать и о том, *что* код должен делать, и о том, *как* он должен это делать. А вот если сначала пишутся тесты, то можно сосредоточиться почти исключительно на том, *что* требуется реализовать.

Позже, при написании самого кода, внимание будет сосредоточено уже на том, *как* реализовать ожидаемое поведение, *что* описано в тестах. Благодаря этому разделению мышление становится чище, фокус — точнее, а результат — мощнее. Такое ощущение повышения ментальной эффективности часто бывает неожиданным, но очень заметным. И это лишь часть выгод, которые дает применение TDD.

Ниже описаны еще несколько преимуществ использования данной методологии.

- *Повышение качества кода*. Предварительное написание тестов помогает обеспечить хорошее покрытие и уменьшает вероятность появления ошибок и багов в рабочем коде. Такой подход побуждает разрабатывать лишь тот функционал, который действительно нужен для прохождения тестов, — это часто приводит к более чистому и простому коду.
- *Лучшие архитектурные решения*. TDD заставляет заранее задуматься о структуре и организации кода. Это раннее проектирование нередко приводит к более продуманной архитектуре и понятной структуре программы.

- *Упрощение рефакторинга*. Если у вас есть надежный набор тестов, то можно спокойно переписывать и улучшать код, не опасаясь, что изменения приведут к регрессиям, — тесты сразу покажут возможные ошибки.
- *Тесты как документация*. Хорошо написанные тесты наглядно демонстрируют, что должен делать код. Это особенно полезно для новых членов команды или при возвращении к старым участкам проекта.
- *Меньше времени на отладку*. Ошибки выявляются еще на стадии кодинга, и в результате тратить время на их диагностику и устранение приходится значительно реже.
- *Лучшее понимание требований*. Если тесты проходят, это означает, что поведение кода соответствует требованиям и разработчики могут быть уверены, что программа работает так, как задумано.

Тем не менее TDD не лишена недостатков.

- *Замедление старта*. На начальных этапах написание тестов может замедлить процесс разработки. В особенно сжатые сроки или при высокой динамике проекта это может стать проблемой.
- *Порог входа*. TDD требует смены мышления и иной логики построения работы с кодом. Многим разработчикам сложно сразу привыкнуть к написанию тестов «наперед» и делать это эффективно.
- *Нагрузка при мелких изменениях*. В случае простых правок необходимость писать отдельный тест может восприниматься как лишняя трата времени.
- *Сложности при работе с интерфейсами и внешними системами*. Тестирование пользовательского интерфейса, работы с базами данных или API — непростая задача. Mock-объекты и заглушки, которые часто применяются в таких случаях, сами по себе усложняют код и требуют поддержки.

Мы весьма положительно относимся к TDD. Однако с годами практики мы столкнулись с ситуациями, когда применять эту медологию становится затруднительно. Один из ярких примеров — проекты с тестовыми наборами, содержащими сотни или даже тысячи тестов. В таких случаях бывает практически невозможно определить заранее, какие именно тесты нужно изменить, чтобы добиться желаемого изменения в исходном коде. Иногда разумнее сначала изменить код, а затем посмотреть, какие тесты перестали проходить.

Тем не менее мы по-прежнему уверены в том, что овладеть TDD крайне важно. Главные достоинства этой методологии могут проявляться скорее в образовательном, чем в прикладном аспекте, однаки навыки и мышление, которые формируются благодаря ей, по-настоящему важны. Если вы освоите TDD настолько, что сможете применять ее эффективно, это неизбежно повлияет на ваш стиль разработки, даже в тех проектах, где сама по себе эта методология не используется.

Поэтому стоит запомнить главное правило: всегда тщательно тестируйте свой код. Какую бы методологию вы ни использовали, практика тестирования — основа надежности и устойчивости программного обеспечения.

Резюме

В этой главе вы разобрались в основах тестирования.

Мы постарались дать достаточно полное описание этого процесса, особенно модульных тестов — именно с ними разработчик имеет дело чаще всего. Надеемся, нам удалось донести важную мысль: тестирование не строго формализованный процесс, которому можно научиться, просто читая книгу. Чтобы разобраться в нем досконально, нужно много практиковаться и экспериментировать. Из всех усилий, которые программист прилагает для освоения профессии, наработка навыков тестирования — одно из самых значимых действий.

В следующей главе мы займемся отладкой и профилированием — приемами, тесно связанными с тестированием. Обязательно изучите их.

11 Отладка и профилирование

> Если отладка — процесс удаления ошибок,
> то программирование должно быть процессом
> их внесения.
>
> *Эдсгер Дейкстра*

В жизни профессионального разработчика отладка и устранение неполадок занимают значительную часть времени. Любая, даже относительно простая программа почти наверняка будет содержать ошибки. Люди неидеальны — мы совершаем ошибки, а значит, и наш код не может быть безупречным. Кроме того, много времени мы тратим на чтение чужого кода. По нашему мнению, хороший разработчик — это тот, кто внимательно отслеживает возможные ошибки, даже если работает с кодом, о котором не сообщалось, что в нем есть проблемы.

Умение быстро и эффективно отлаживать код — навык, который нужно постоянно развивать. Как и тестирование, отладка — это область, в которой лучше всего учиться на практике. Существуют полезные приемы и советы, но нет ни одной книги, способной научить всему, что нужно для уверенного выполнения отладки.

По нашему опыту, научиться этому в полной мере больше всего помогает работа с коллегами. Мы до сих пор с восхищением наблюдаем за тем, как опытные разработчики устраняют проблемы. Особенно интересно смотреть, какие шаги они предпринимают, какие причины исключают и как выбирают направление, которое в итоге приводит их к решению.

У каждого коллеги, с которым мы работаем, есть чему поучиться. Иногда они удивляют блестящей догадкой, которая оказывается правильной. В такие моменты не стоит просто восхищаться (а тем более завидовать) — лучше сразу спросить, как удалось прийти к этой догадке и почему была выбрана именно она. В ответах можно найти полезную информацию и идеи для самостоятельного изучения. И тогда в следующий раз уже вы будете тем, кто найдет баг.

Некоторые ошибки легко заметить: они связаны с очевидными промахами, и как только проявляется их эффект, решение становится очевидным. Но бывают и другие — коварные баги, обнаружение которых требует опыта, нестандартного

мышления и настоящей смекалки разработчика. Самый сложный вид ошибок — недетерминированные. Они возникают не всегда и могут проявляться только в определенной среде, но не воспроизводиться в другой, казалось бы, идентичной.

В профессиональной среде отладка часто происходит в стрессовых условиях. Например, если перестал работать сайт или клиенты недовольны, бизнес несет убытки. Разработчики при этом оказываются под давлением: нужно срочно найти и устранить проблему. В таких ситуациях важнее всего — сохранять спокойствие. Без этого невозможно эффективно бороться с багами. Стресс ухудшает креативное мышление и способность находить решения. Поэтому сделайте глубокий вдох, сядьте удобно и сосредоточьтесь.

В этой главе мы разберем несколько полезных приемов, которые можно выбирать в зависимости от серьезности проблемы, а также дадим советы, которые, надеемся, помогут вам пополнить арсенал средств для борьбы с багами и неполадками.

Приемы отладки

В этом разделе мы познакомим вас с рядом приемов, которые чаще всего используем сами. Это далеко не полный список, но он поможет понять, с чего стоит начать, если нужно отлаживать собственный код Python.

Отладка с помощью функции print()

Чтобы понять, в чем заключается баг, необходимо разобраться, что именно делает код в момент ошибки. Поэтому мы рассмотрим несколько способов, с помощью которых можно исследовать состояние программы во время ее выполнения.

Самый простой способ — расставить в нужных местах вызовы функции `print()`. Так вы увидите, какие участки кода выполняются и какие значения принимают важные переменные на разных этапах выполнения. Например, если вы разрабатываете сайт на Django и поведение страницы не соответствует ожиданиям, то можно просто добавить вызовы `print()` и следить за выводом в консоли при каждом обновлении страницы.

У этого способа есть ограничения и недостатки. Его использование требует доступа к исходному коду и возможности запускать его в терминале, где видно результат вызовов `print()`. Это удобно в локальной среде, но бесполезно в других случаях — например, на сервере или в готовой версии продукта.

Кроме того, повсеместное добавление вызовов `print()` может привести к возникновению повторяющегося отладочного кода. Иногда хочется выводить, скажем, отметки времени (как мы делали при замерах скорости генераторов и списковых включений) или собирать строку из разных значений. Все это может усложнить отладку. К тому же можно забыть удалить `print()` из кода, и такие вызовы потом окажутся в готовой версии программы.

По этим причинам иногда удобнее воспользоваться собственной отладочной функцией, а не просто вызывать `print()` напрямую. Сейчас разберемся, как это сделать.

Отладка с помощью специальной функции

Иметь собственную отладочную функцию, сохраненную в отдельном файле, откуда ее можно быстро скопировать и вставить в нужное место, — очень удобно. Если у вас уже есть навык, то можете написать такую функцию прямо на месте. *Главное, чтобы она была самодостаточной и не оставляла после себя мусор, когда вы будете ее удалять.* Кроме того, наличие этой функции помогает избежать конфликтов имен с другими частями кода.

Рассмотрим пример такой функции:

```
# custom.py
def debug(*msg, print_separator=True):
    print(*msg)
    if print_separator:
        print("-" * 40)

debug("Data is ...")
debug("Different", "Strings", "Are not a problem")
debug("After while loop", print_separator=False)
```

Здесь используется именованный аргумент `print_separator`, который позволяет при необходимости добавить строку из 40 дефисов в качестве разделителя.

Функция просто выводит переданное сообщение, и если флаг `print_separator` установлен в значение `True`, то выводит строку-разделитель. При запуске такого кода получится вывод наподобие следующего:

```
$ python custom.py
Data is ...
----------------------------------------
Different Strings Are not a problem
----------------------------------------
After while loop
```

Как видите, после последней строки разделитель не выводится — все работает должным образом.

Это простой способ немного расширить функциональность `print()`.

Теперь посмотрим, как можно измерить разницу во времени между вызовами, воспользовавшись одной из интересных возможностей Python:

```
# custom_timestamp.py
from time import sleep
def debug(*msg, timestamp=[None]):
    from time import time     # локальный импорт
```

```
    print(*msg)
    if timestamp[0] is None:
        timestamp[0] = time() # 1
    else:
        now = time()
        print(f" Time elapsed: {now - timestamp[0]:.3f}s")
        timestamp[0] = now    # 2

debug("Entering buggy piece of code...")
sleep(0.3)
debug("First step done.")
sleep(0.5)
debug("Second step done.")
```

Это код посложнее. Прежде всего обратите внимание вот на что: импорт `time()` из модуля `time` выполняется *внутри* функции `debug()`. Благодаря этому не требуется явно добавлять оператор `import` в начало файла и затем помнить о его удалении. Такой подход делает функцию полностью автономной.

Заметьте, как мы определили параметр `timestamp`. Это параметр функции, в качестве значения по умолчанию которой выступает список. В главе 4 мы уже предупреждали: не стоит использовать изменяемые значения по умолчанию в параметрах, поскольку они инициализируются при первом разборе функции и затем этот объект сохраняется между вызовами функции. Чаще всего такое поведение нежелательно. Однако в данном случае мы сознательно используем эту особенность, чтобы сохранить метку времени предыдущего вызова функции, не используя внешнюю глобальную переменную. Этот прием мы позаимствовали из техники *замыканий*, с которой советуем познакомиться и вам.

После вывода сообщения мы проверяем содержимое единственного элемента списка `timestamp`. Если в нем `None`, значит, это первый вызов и у нас нет предыдущей метки времени — тогда мы присваиваем текущую метку времени этому элементу (`#1`). Если значение уже есть, то мы можем вычислить разницу между текущим временем и предыдущим (мы форматируем ее до трех знаков после запятой) и после этого обновляем значение в списке, записывая туда текущую метку времени (`#2`).

Запуск такого кода приведет к следующему выводу:

```
$ python custom_timestamp.py
Entering buggy piece of code...
First step done.
 Time elapsed: 0.300s
Second step done.
 Time elapsed: 0.500s
```

Применение собственной функции `debug()` помогает устранить некоторые недостатки `print()`. Она снижает дублирование отладочного кода и облегчает удаление всех таких вызовов, когда они больше не нужны. Тем не менее вам все еще приходится изменять код и запускать его в терминале, где можно

видеть вывод. Позже в текущей главе мы разберемся, как преодолеть эти ограничения с помощью логирования.

Отладка с помощью отладчика Python

Еще один эффективный способ отладки кода Python — использовать интерактивный отладчик. Он доступен в модуле `pdb` стандартной библиотеки. Тем не менее мы предпочитаем сторонний пакет `pdbpp`, который можно использовать как полноценную замену `pdb`. Он имеет чуть более удобный интерфейс и полезные дополнительные функции. Наша любимая функция — *sticky-режим*, в котором при пошаговом выполнении можно видеть весь текст функции целиком.

Существует несколько способов активировать отладчик. Если установлен пакет `pdbpp`, то именно он будет запущен вместо стандартного `pdb`. Самый распространенный способ — добавить в код команду, вызывающую отладчик. Это называется установкой *точки останова*.

Дойдя до этой точки, программа приостанавливает свое выполнение, и вы получаете доступ к интерактивной сессии отладчика в консоли. В ней можно просматривать все переменные в текущей области видимости и пошагово выполнять строки программы. Кроме того, можно динамически изменять данные, чтобы повлиять на дальнейшее поведение кода.

Представьте простой пример. Программа получает словарь и кортеж ключей. Затем она обрабатывает элементы словаря, соответствующие этим ключам. При выполнении возникает исключение `KeyError`, поскольку одного из ключей нет в словаре. Допустим, мы не можем повлиять на входные данные (например, они поступают из API стороннего сервиса), но хотим обойти эту ошибку, чтобы проверить, корректно ли программа работает при валидных данных. Ниже показан пример того, как с помощью отладчика остановить выполнение, проанализировать и изменить данные, а затем продолжить выполнение:

```
# pdebugger.py
# d приходит из внешнего источника, на который у нас нет влияния
d = {"first": "v1", "second": "v2", "fourth": "v4"}
# keys тоже задаются извне и находятся вне нашего контроля
keys = ("first", "second", "third", "fourth")

def do_something_with_value(value):
    print(value)

for key in keys:
    do_something_with_value(d[key])

print("Validation done.")
```

Как видно из примера, программа завершится с ошибкой, когда переменной `key` будет присвоено значение `"third"`, которого нет в словаре. Напомним: в этом примере мы предполагаем, что переменные `d` и `keys` поступают из внешнего

источника, на который мы не можем повлиять. Если просто запустить код, то вывод будет следующим:

```
$ python pdebugger.py
v1
v2
Traceback (most recent call last):
  File ".../ch11/pdebugger.py", line 13, in <module>
    do_something_with_value(d[key])
                            ~^^^^^
KeyError: 'third'
```

Мы увидим, что ключа нет в словаре, но поскольку при каждом запуске мы можем получить другой словарь или кортеж ключей, то данная информация особой пользы не несет. Нам нужно иметь возможность исследовать и изменять данные во время выполнения программы, поэтому добавим точку останова прямо перед циклом `for`. В современных версиях Python это делается путем вызова встроенной функции `breakpoint()`:

```
breakpoint()
```

В версиях Python до 3.7 вместо этого приходилось импортировать модуль `pdb` и вызывать функцию `pdb.set_trace()`:

```
import pdb; pdb.set_trace()
```

Мы использовали точку с запятой, чтобы объединить несколько выражений в одной строке. Согласно стандарту оформления кода PEP 8 такой стиль не одобряется, но это достаточно распространенная практика для установки точки останова — благодаря ей впоследствии придется удалять меньше строк.

Функция breakpoint() вызывает sys.breakpointhook(), которая, в свою очередь, вызывает pdb.set_trace(). При необходимости можно переопределить поведение sys.breakpointhook(), задав переменную среды PYTHONBREAKPOINT. Ее можно направить на другую функцию, которую Python импортирует и вызовет вместо pdb.set_trace().

Пример с этим кодом находится в модуле `pdebugger_pdb.py`. Если теперь запустить его, то начнется самое интересное (ваш вывод может немного отличаться; в наш мы добавили комментарии):

```
$ python pdebugger_pdb.py
[0] > .../ch11/pdebugger_pdb.py(17)<module>()
-> for key in keys:
(Pdb++) l                 # показать исходный код вокруг текущей строки
 16
 17 -> for key in keys:   # здесь установлена точка останова
 18      do_something_with_value(d[key])
 19
```

```
(Pdb++) keys              # посмотреть содержимое кортежа keys
('first', 'second', 'third', 'fourth')
(Pdb++) d.keys()          # посмотреть ключи словаря d
dict_keys(['first', 'second', 'fourth'])
(Pdb++) d['third'] = 'placeholder'   # вручную добавить недостающий ключ
(Pdb++) c                 # продолжить выполнение
v1
v2
placeholder
v4
Validation done.
```

Прежде всего обратите внимание на то, что при достижении точки останова откроется консоль, в которой будет указано, где именно находится интерпретатор (модуль Python) и какая строка будет выполнена следующей. На данном этапе можно выполнить несколько исследовательских действий — например, просмотреть код до и после текущей строки, вывести трассировку стека, поработать с объектами. В нашем примере мы сначала исследуем кортеж `keys`, а затем ключи словаря `d`. Видим, что `'third'` отсутствует, и вручную добавляем его в словарь. (Опасно ли такое действие? Подумайте об этом.) Затем, когда все ключи имеются, мы вводим `c`, чтобы продолжить выполнение программы.

Кроме того, в отладчике можно выполнять программу построчно с помощью команды `n` (next). Команда `s` (step) позволяет шагнуть внутрь вызываемой функции, чтобы выполнить более детальный анализ. Дополнительные точки останова можно установить с помощью команды `b`. Полный список команд приведен в официальной документации (https://docs.python.org/3.12/library/pdb.html), а также доступен прямо в консоли отладчика через команду `h` (help).

Из вывода предыдущего запуска видно, что мы успешно достигли завершения проверки.

`pdb` (или `pdbpp`) — незаменимый инструмент, который мы используем ежедневно. Пожалуйста, поэкспериментируйте с ним. Добавьте точку останова, посмотрите, как ее исследовать, изучите официальную документацию и попробуйте использовать в своем коде различные команды, чтобы понять, как они работают.

Обратите внимание: в примере предполагается, что установлен pdbpp. Если это не так и используется обычный pdb, то некоторые команды могут вести себя иначе. Например, буква d в pdb означает команду down. Чтобы интерпретировать ее как обычное выражение, а не как команду, перед ней нужно поставить !.

Анализ логов

Другой способ отладки неправильно работающего приложения — изучение его логов. *Лог* — это упорядоченный список событий, произошедших во время работы приложения, или действий, которые были выполнены. Если лог сохраняется на диск в виде файла, то его называют *файловым*.

Отладка с помощью логов во многом напоминает добавление вызовов `print()` или пользовательской отладочной функции. Главное различие заключается в том, что логирование обычно добавляется в код заранее — чтобы упростить будущую отладку, а не в момент, когда проблема уже возникла. Кроме того, систему логирования легко настроить так, чтобы она выводила сообщения в файл или отправляла их по сети. Благодаря этому логирование становится особенно удобным для отладки кода, работающего на удаленной машине, к которой у вас нет прямого доступа.

Однако из-за того, что логирование добавляется до возникновения ошибки, возникает вопрос: что именно следует логировать? Обычно в логе должны быть записи, отражающие начало, завершение (а возможно, и промежуточные этапы) выполнения важных процессов в приложении. В эти записи стоит добавлять значения ключевых переменных. Кроме того, необходимо записывать ошибки — так при возникновении проблемы можно будет проследить, что именно пошло не так.

Практически все параметры логирования в Python можно настраивать. Это дает большую свободу действий: можно изменить место вывода лога, выбрать, какие сообщения записывать, и задать их формат. И все это можно делать, не затрагивая остальной код, — достаточно поменять конфигурацию логирования.

В системе логирования Python задействованы четыре основных типа объектов.

- *Логгеры* (loggers) предоставляют интерфейс, который приложение использует для записи сообщений в лог.
- *Обработчики* (handlers) направляют записи лога, созданные логгерами, в нужное место (например, в файл или консоль).
- *Фильтры* (filters) позволяют более точно настроить параметры того, какие записи лога будут выведены.
- *Форматтеры* (formatters) задают, как именно будет выглядеть каждая запись лога в итоговом выводе.

Запись в лог осуществляется с помощью методов экземпляра класса `Logger`. Каждое сообщение имеет уровень серьезности. Чаще всего используются уровни `DEBUG`, `INFO`, `WARNING`, `ERROR` и `CRITICAL`. Логгеры используют эти уровни, чтобы определить, какие сообщения нужно выводить. Все, что ниже заданного уровня логгера, будет проигнорировано. Поэтому важно правильно выбирать уровень для каждой записи. Если вы записываете все на уровне `DEBUG`, то для отображения этих сообщений логгер должен быть настроен на уровень `DEBUG` или ниже. Это может быстро привести к тому, что логи станут чересчур объемными. С другой стороны, если вы записываете все на уровне `CRITICAL`, то можно упустить важную отладочную информацию.

В Python можно выбирать, куда отправлять лог-сообщения: в файл, сетевое хранилище, очередь, консоль, системный лог операционной системы и т. д. Конкретный выбор зависит от контекста. Например, при разработке вы чаще всего будете выводить сообщения в терминал. Если приложение работает на одном компьютере, то можно писать в файл или передавать сообщения в системный лог.

Но если архитектура приложения является распределенной и охватывает несколько машин (как в случае сервис-ориентированных или микросервисных архитектур), то для логирования разумнее использовать централизованное решение. Так все сообщения от разных сервисов будут собраны в одном месте, и их можно будет анализировать совокупно. Это значительно упрощает отладку: сопоставлять логи из множества отдельных файлов бывает очень сложно.

Сервис-ориентированная архитектура (SOA) — архитектурный подход, при котором компоненты приложения предоставляют друг другу функциональность, используя сетевые протоколы. Преимущество такого подхода в том, что каждый сервис можно реализовать на наиболее подходящем языке. Главное — обеспечить их взаимодействие с помощью общего формата обмена данными.

Микросервисная архитектура является результатом развития идеи SOA, но имеет свои архитектурные принципы.

Обратная сторона гибкости системы логирования Python — ее относительная сложность. Но чаще всего достаточно стандартных настроек, и менять конфигурацию вручную приходится только при особых требованиях. Посмотрим, как выглядит простой пример записи сообщений в файл:

```
# log.py
import logging

logging.basicConfig(
    filename="ch11.log",
    level=logging.DEBUG,
    format="[%(asctime)s] %(levelname)s: %(message)s",
    datefmt="%m/%d/%Y %I:%M:%S %p")

mylist = [1, 2, 3]
logging.info("Starting to process 'mylist'...")

for position in range(4):
    try:
        logging.debug(
            "Value at position %s is %s",
            position,
            mylist[position]
        )
    except IndexError:
        logging.exception("Faulty position: %s", position)

logging.info("Done processing 'mylist'.")
```

Сначала импортируется модуль `logging`, а затем задается базовая конфигурация. Указывается имя лог-файла, настраивается уровень лог-сообщений (все сообщения уровня `DEBUG` и выше), а также формат вывода. Мы хотим видеть в сообщениях дату и время, уровень серьезности и сам текст.

Выполнив настройку, мы можем начинать логирование. Сначала записываем информационное сообщение (уровень `INFO`) о том, что начинается обработка списка. Внутри цикла мы будем записывать текущее значение (через вызов `debug()` — это логирование уровня `DEBUG`). Мы используем `debug()` именно для того, чтобы в будущем при необходимости можно было исключить эти записи, просто повысив уровень логирования до `logging.INFO` или выше. Это особенно важно при работе с большими списками — мы не хотим, чтобы в лог всегда попадали все значения.

Если возникает `IndexError` (а она возникнет, поскольку цикл идет по `range(4)`), то вызывается `logging.exception()`. Это специальная функция, которая регистрирует сообщение уровня `ERROR` и автоматически добавляет трассировку стека.

В конце программы записывается еще одно информационное сообщение о завершении обработки. После запуска этого кода появится новый файл `ch11.log` с примерно следующим содержимым:

```
# ch11.log
[10/06/2024 10:08:04 PM] INFO: Starting to process 'mylist'...
[10/06/2024 10:08:04 PM] DEBUG: Value at position 0 is 1
[10/06/2024 10:08:04 PM] DEBUG: Value at position 1 is 2
[10/06/2024 10:08:04 PM] DEBUG: Value at position 2 is 3
[10/06/2024 10:08:04 PM] ERROR: Faulty position: 3
Traceback (most recent call last):
  File ".../ch11/log.py", line 20, in <module>
    mylist[position],
    ~~~~~~^^^^^^^^^^
IndexError: list index out of range
[10/06/2024 10:08:04 PM] INFO: Done processing 'mylist'.
```

Именно эти данные и нужны для отладки приложения, запущенного не на локальной машине, а где-то удаленно. Мы видим, какие действия выполнил код, а также получаем трассировку при возникновении исключений.

Вы можете поэкспериментировать с уровнями логирования в предыдущем примере — изменить как код, так и конфигурацию. Это поможет вам увидеть, как меняется содержимое лога в зависимости от настроек.

Приведенный пример — лишь первый шаг в освоении логирования. Более подробную информацию можно найти в официальной документации Python, в разделах HOWTO: Logging HOWTO и Logging Cookbook.

Логирование — своего рода искусство. Важно найти баланс между избыточностью и отсутствием информации. Идеальный подход — фиксировать все, что помогает убедиться в корректной работе приложения, и (в обязательном порядке) все ошибки и исключения.

Другие приемы

Завершая раздел, кратко обсудим еще пару полезных приемов.

Анализ трассировок

Ошибки часто проявляются в виде необработанных исключений. Умение читать трассировку стека — навык, крайне важный при отладке. Обязательно вернитесь к подразделу о трассировках в главе 7 и убедитесь, что вы поняли его содержимое. Когда вы пытаетесь выяснить причину исключения, полезно исследовать состояние программы (с помощью описанных выше приемов), опираясь на строки, указанные в трассировке.

Утверждения

Ошибки часто возникают из-за неверных предположений в коде. Утверждения помогают проверить, верны ли эти предположения. Если да, то выполнение продолжается как обычно. Если нет — выбрасывается исключение, указывающее, что предположение оказалось неверным.

Иногда быстрее добавить несколько утверждений, чтобы исключить возможные причины ошибки, чем разбираться с отладчиком или вставлять вызовы `print()`. Ниже приведен пример:

```
# assertions.py
mylist = [1, 2, 3]        # предположим, это данные из внешнего источника
assert 4 == len(mylist)   # вызовет ошибку
for position in range(4):
    print(mylist[position])
```

Здесь мы предполагаем, что список `mylist` поступает из внешнего источника (например, от пользователя) и содержит четыре элемента. Мы добавили утверждение, чтобы проверить это предположение. При запуске программы получим сообщение об ошибке:

```
$ python assertions.py
Traceback (most recent call last):
  File ".../ch11/assertions.py", line 4, in <module>
    assert 4 == len(mylist) # this will break
           ^^^^^^^^^^^^^^^^
AssertionError
```

Благодаря этому сразу становится понятно, где возникла проблема.

Следует помнить, что при запуске программы с флагом -O Python игнорирует все утверждения. Если работоспособность кода зависит от них, такое игнорирование может привести к неожиданным последствиям.

Можно использовать расширенную форму утверждения со вторым выражением, в котором указывается сообщение об ошибке:

```
assert выражение1, выражение2
```

Оно передается в исключение `AssertionError`, которое выбрасывается при срабатывании утверждения. Обычно это строка с сообщением об ошибке. Например, если изменить утверждение из предыдущего примера так:

```
assert 4 == len(mylist), f"Mylist has {len(mylist)} elements"
```

то результат будет таким:

```
$ python assertions.py
Traceback (most recent call last):
  File ".../ch11/assertions.py", line 19, in <module>
    assert 4 == len(mylist), f"Mylist has {len(mylist)} elements"
           ^^^^^^^^^^^^^^^^
AssertionError: Mylist has 3 elements
```

Поиск справочной информации

В официальной документации Python есть раздел, посвященный отладке и профилированию. В нем вы найдете сведения о фреймворке отладки `bdb`, а также о модулях `faulthandler`, `timeit`, `trace`, `tracemalloc` и `pdb`.

Далее рассмотрим несколько практических советов по поиску и устранению проблем.

Рекомендации по устранению неполадок

В этом небольшом разделе мы поделимся советами, основанными на собственном опыте отладки и диагностики проблем.

Где искать

Первый совет касается того, куда именно ставить точки останова при отладке. Независимо от того, что вы используете: `print()`, пользовательскую функцию, `pdb` или логирование, — вам все равно нужно выбрать правильные места получения информации. Одни участки кода подходят для этого лучше других, и есть более эффективные стратегии отладки.

Обычно мы избегаем размещения точки останова внутри условного оператора `if`. Если выполнение не зайдет в нужную ветку, то вы просто не получите необходимую информацию. Иногда ошибку трудно воспроизвести, или программа доходит до нужного участка только после долгого выполнения. Поэтому стоит хорошо обдумать, куда ставить такие точки.

Еще один важный вопрос — с чего начинать отладку. Представьте, что у вас есть сто строк кода, обрабатывающих данные. Данные поступают на первой строке, а к сотой уже становятся некорректными. Вы не знаете, на каком этапе возникает ошибка. Что делать? Один из вариантов — поставить точку останова на строке 1 и пошагово проверять данные на каждой из ста строк. В худшем случае вы потратите много времени (и, возможно, не одну чашку кофе), прежде чем доберетесь до источника ошибки. Но можно подойти к делу иначе.

Начните с середины — со строки 50. Если данные там корректны, значит, ошибка находится дальше и следующую точку останова можно поставить на строке 75. Если на строке 50 данные уже испорчены, то поставьте следующую точку на строке 25. И так далее — каждый раз сужайте диапазон вдвое.

Во «вредном» линейном подходе вы бы пошли от строки 1 к 2, затем к 3 — и так до 99. А вот при использовании описанного метода последовательность точек останова будет выглядеть примерно так: 50, 75, 87, 93, 96... 99. Этот вариант намного быстрее, поскольку выполняется по логарифмическому принципу. Такой метод называется *двоичным поиском*. Он основан на стратегии «разделяй и властвуй» и отлично подходит для поиска ошибок. Освойте его — и он не раз поможет вам.

Использование тестов для отладки

В главе 10 мы кратко познакомили вас с *разработкой через тестирование* (TDD). Даже если вы не планируете полностью придерживаться этой методики, один из ее приемов точно стоит взять на вооружение: перед исправлением бага напишите воспроизводящий его тест. Причин для этого несколько. Если у вас есть баг, но все тесты проходят успешно, значит, в тестовой базе не хватает нужного покрытия — она не проверяет поведение, при котором проявляется баг.

Написание теста перед исправлением помогает убедиться, что баг действительно устранен, — тест должен начать проходить только после внесения корректного исправления. Такой тест станет частью вашей постоянной базы и защитит от повторного появления той же ошибки в будущем.

Мониторинг

Мониторинг — еще один важный этап. Программное обеспечение порой ведет себя неожиданно в граничных ситуациях: когда сеть недоступна, очередь переполнена или внешний компонент не отвечает. В такие моменты важно иметь целостную картину происходящего, чтобы можно было связать конкретную проблему с внешними условиями, даже если эта связь на первый взгляд неочевидна.

Мониторить можно все, что поддается автоматической проверке: API, процессы, доступность веб-страниц, время отклика и многое другое. В идеале о мониторинге

стоит думать уже на стадии проектирования — это поможет заранее предусмотреть нужные точки отслеживания.

Теперь перейдем к теме профилирования кода на Python.

Профилирование кода на Python

Профилирование — это запуск приложения с одновременным сбором информации о его поведении: сколько раз вызывается каждая функция, сколько времени тратится внутри нее и т. д.

Профилирование тесно связано с отладкой. Инструменты и подходы у этих процессов разные, однако цель одна — понять, в чем причина проблемы, и устранить ее. Разница в том, что при отладке мы ищем причины ошибок и сбоев, а при профилировании — источники проблем с производительностью.

Иногда профилирование указывает, где именно возникает узкое место. Тогда понадобится применить отладочные приемы, чтобы разобраться, почему этот участок кода работает медленно. Например, ошибочная логика SQL-запроса может привести к тому, что из таблицы загружаются тысячи строк вместо сотен. Или функция вызывается гораздо чаще, чем предполагалось, и тогда нужно проанализировать, почему так происходит, и найти способ это исправить.

Существует несколько способов профилирования программ, написанных на Python. В официальной документации в разделе *Profiling* вы найдете описание двух стандартных реализаций одного интерфейса — `profile` и `cProfile`:

- `cProfile` написан на C и предполагает минимальные накладные расходы, благодаря чему подходит для профилирования долгоживущих программ;
- `profile` реализован на чистом Python и, как следствие, сильно тормозит выполнение профилируемого кода.

Обе реализации поддерживают *детерминированное профилирование*, при котором отслеживаются все вызовы функций, возвраты и исключения. Время между событиями измеряется с высокой точностью. Существует и другой подход — *статистическое профилирование*. В этом случае программа периодически «опрашивается» — случайным образом считывается стек вызовов, и по накопленной статистике делается вывод, в каких местах тратится больше всего времени.

Такой метод меньше влияет на производительность, но дает только приближенные результаты. Интерпретатор Python работает так, что даже детерминированное профилирование не создает слишком большого накладного эффекта, поэтому мы покажем простой пример с использованием `cProfile` из командной строки.

Бывают случаи, когда даже небольшие накладные расходы от cProfile недопустимы — например, если вы хотите профилировать код на рабочем сервере, а в среде разработки не удается воспроизвести проблему с производительностью. В таких ситуациях действительно стоит использовать статистический профилировщик. Мы рекомендуем обратить внимание на py-spy (см. на GitHub: https://github.com/benfred/py-spy).

В следующем примере мы снова вычислим пифагоровы тройки:

```
# profiling/triples.py
def calc_triples(mx):
    triples = []
    for a in range(1, mx + 1):
        for b in range(a, mx + 1):
            hypotenuse = calc_hypotenuse(a, b)
            if is_int(hypotenuse):
                triples.append((a, b, int(hypotenuse)))
    return triples

def calc_hypotenuse(a, b):
    return (a**2 + b**2) ** 0.5

def is_int(n):
    return n.is_integer()

triples = calc_triples(1000)
```

Сценарий достаточно простой: мы перебираем значения `a` и `b` в диапазоне от 1 до `mx`, при этом следим, чтобы $b \geq a$ — во избежание повторения пар. Для каждой пары с помощью функции `calc_hypotenuse()` вычисляется гипотенуза, затем с помощью `is_int()` проверяется, является ли она целым числом. Если да, значит, перед нами пифагорова тройка — мы добавляем ее в список.

Когда мы запускаем профилирование этого сценария, вывод отображается в виде таблицы. В колонках указаны: `ncalls` — сколько раз была вызвана функция, `tottime` — общее время, затраченное в теле функции (без учета вызванных ею других функций), `cumtime` — суммарное время, вместе с вызовами других функций, `filename:lineno(function)` — имя файла, строка и название функции. Результат выглядит так (для краткости мы опустили два столбца `percall`):

```
$ python -m cProfile profiling/triples.py
1502538 function calls in 0.393 seconds
Ordered by: cumulative time

ncalls tottime cumtime filename:lineno(function)
     1 0.000     0.393 {built-in method builtins.exec}
     1 0.000     0.393 triples.py:1(<module>)
     1 0.143     0.393 triples.py:1(calc_triples)
```

```
500500 0.087     0.147 triples.py:15(is_int)
500500 0.102     0.102 triples.py:11(calc_hypotenuse)
500500 0.060     0.060 {method 'is_integer' of 'float' objects}
  1034 0.000     0.000 {method 'append' of 'list' objects}
     1 0.000     0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Даже при такой ограниченной информации можно сделать полезные выводы. Например, видно, что временная сложность алгоритма растет квадратично по отношению к размеру входных данных. Функция `calc_hypotenuse()` вызывается ровно `mx` × (`mx` + 1) / 2 раз. В нашем случае мы задали `mx` = 1000 и получили ровно 500 500 вызовов. Внутри основного цикла происходит три ключевых действия: вызов `calc_hypotenuse()`, вызов `is_int()`, добавление тройки в список, если условие выполнено.

Если взглянуть на значения в колонке `cumtime` в отчете профилирования, то можно увидеть, что программа провела 0,147 секунды внутри функции `is_int()`, по сравнению с 0,102 секунды в `calc_hypotenuse()`. Обе функции были вызваны одинаковое количество раз, поэтому логично начать оптимизацию с более затратной — `is_int()`.

Если перейти к колонке `tottime`, то окажется, что внутри `is_int()` программа провела 0,087 секунды. В данное значение не входят те 0,060 секунды, которые ушли на вызовы метода `is_integer()` у объектов типа `float`, изнутри `is_int()`. А ведь `is_int()` только вызывает этот метод у своего параметра `n`. Значит, сама по себе обертка `is_int()` добавляет накладные расходы в 87 миллисекунд. В данном случае от этой функции нет практической пользы — можно напрямую вызывать `hypotenuse.is_integer()`, и таким образом мы выигрываем 87 миллисекунд.

После повторного запуска профилирования оказывается, что теперь программа тратит больше времени в `calc_hypotenuse()`, чем в `is_integer()`. Можно попробовать улучшить и ее. Как мы обсуждали в главе 5, оператор возведения в степень ** используется дольше, чем умножение на само себя. Это позволяет немного ускорить вычисления, если изменить `calc_hypotenuse()` следующим образом:

```
def calc_hypotenuse(a, b):
    return (a * a + b * b) ** 0.5
```

После следующего запуска профилирования видно, что теперь выполнение `calc_hypotenuse()` занимает 0,084 секунды. Мы выиграли всего 18 миллисекунд. Вероятно, этот показатель можно было увеличить еще больше, если отказаться от самой функции `calc_hypotenuse()` и вычислять гипотенузу прямо в теле цикла:

```
            hypotenuse = (a * a + b * b) ** 0.5
```

Профилирование этой версии показывает, что так можно сэкономить до 100 миллисекунд. Однако, на наш взгляд, преимущества функции `calc_hypotenuse()` — понятность, удобство поддержки и легкость тестирования — более весомы, чем незначительный прирост производительности при ее удалении.

Все четыре версии этой программы добавлены в исходный код, прилагаемый к книге. Мы рекомендуем вам самостоятельно запустить профилирование и поэкспериментировать с другими изменениями, чтобы посмотреть, как они влияют на производительность. Например, можно попробовать превратить `calc_triples()` в генераторную функцию.

Пример, конечно, простой, но позволяет понять, как профилировать приложение. Количество вызовов функции помогает лучше представить временную сложность алгоритма. Например, многие разработчики не замечают, что две вложенные инструкции `for` работают пропорционально квадрату размера входных данных.

Мы рассмотрели профилирование функций, но при необходимости можно добиться еще большей точности и профилировать каждую строку кода отдельно. В среднем Python-разработчику редко приходится заниматься профилированием, но иногда это необходимо, и полезно знать, что такая возможность есть.

И еще один важный момент: результаты профилирования могут отличаться в зависимости от системы, в которой вы его выполняете. Поэтому имеет смысл профилировать программу в системе, максимально похожей на целевую, — а в идеале в той же самой.

В этом разделе мы сосредоточились на профилировании и оптимизации времени выполнения программы. Однако с помощью профилирования можно еще и анализировать использование памяти. Один из самых популярных инструментов для профилирования памяти в Python — memray. Подробнее о нем можно прочитать на сайте https://bloomberg.github.io/memray/.

Когда стоит профилировать

Важно понимать, в каких случаях действительно нужно профилировать код и что делать с результатами профилирования. Дональд Кнут как-то сказал: «Преждевременная оптимизация — корень всех зол». Мы бы не стали выражаться столь категорично, но суть верна: не стоит жертвовать читабельностью и поддерживаемостью кода ради экономии нескольких миллисекунд.

Во-первых, ваша главная забота — *корректность* кода. Он должен выдавать правильный результат. Поэтому пишите тесты, проверяйте крайние случаи, нагружайте код по максимуму — всеми способами, которые кажутся вам разумными. Не старайтесь отложить на потом то, что может пойти не так. Лучше сразу проверить — и быть уверенными.

Во-вторых, следуйте *лучшим практикам* программирования. Думайте о читабельности, расширяемости, слабой связанности, модульности и хорошем проектировании. Используйте принципы ООП: инкапсуляцию, абстракцию, принцип единственной ответственности, открытость/закрытость и др. Эти концепции помогут вам развить мышление, выйти за рамки механического кодинга и по-новому взглянуть на архитектуру.

В-третьих, выполняйте *рефакторинг* кода. Придерживайтесь правила скаутов:

> Всегда оставляйте место более чистым, чем оно было до вас.

Применяйте это правило к своему коду.

И только после того как вы убедились в корректности кода, навели порядок в архитектуре и выполнили рефакторинг, можете переходить к оптимизации и профилированию.

Запустите профилировщик и выясните, где находятся узкие места. Анализируя результаты, обращайте внимание на функции, которые вызываются чаще всего. Как мы уже говорили в главе 5 (посвященной включениям и генераторам), небольшая оптимизация в функции, вызываемой миллион раз, даст гораздо больший эффект, чем серьезная оптимизация в коде, который вызывается один-два раза.

Когда вы определите, какие участки кода тормозят выполнение, начинайте с самого тяжелого. Порой устранение одного узкого места вызывает цепную реакцию, меняющую работу остальной части программы. В одних случаях такие изменения минимальны, в других — более значительны. Все зависит от структуры и реализации вашего кода. Именно поэтому стоит начинать с самой серьезной проблемы.

Одна из причин популярности Python — возможность расширять его за счет модулей, написанных на более быстрых компилируемых языках, таких как C или C++. Поэтому, если в вашем проекте есть критически важный участок, а добиться нужной производительности на Python не получается, вы всегда можете переписать эту часть на C.

Измерение времени выполнения

Прежде чем завершить эту главу, хочется кратко поговорить об измерении времени выполнения кода. Иногда бывает полезно измерить производительность небольших фрагментов, чтобы сравнить разные варианты реализации. Например, если вы рассматриваете несколько способов выполнения одной операции и хотите выбрать наиболее быстрый, то имеет смысл сравнить их по времени, не прибегая к профилированию всей программы.

Некоторые примеры измерения и сравнения времени выполнения мы уже приводили — например, в главе 5 (посвященной включениям и генераторам), когда сравнивали производительность циклов `for`, включений списков и функции `map()`. Сейчас мы хотим показать более надежный подход, в котором используется модуль `timeit`. В нем применяется повторный запуск кода и другие приемы для получения более точных измерений времени.

Освоить использование модуля `timeit` бывает непросто, поэтому мы рекомендуем прочитать документацию на официальном сайте Python и попробовать несколько примеров, прежде чем активно его применять. Ниже представлен краткий образец того, как запустить `timeit` из командной строки и измерить время выполнения двух разных версий функции `calc_hypotenuse()` из предыдущего примера:

```
$ python -m timeit -s 'a=2; b=3' '(a**2 + b**2) ** .5'
5000000 loops, best of 5: 91 nsec per loop
```

Здесь мы запускаем модуль `timeit`, предварительно инициализируя переменные `a = 2` и `b = 3`, а затем измеряем время выполнения выражения `(a**2 + b**2) ** 0.5`. В выводе видно, что `timeit` выполнил пять прогонов, каждый из которых содержал 5 000 000 итераций вычисления. Лучшая средняя продолжительность одной итерации составила 91 наносекунду. Теперь посмотрим, сколько времени занимает альтернативный вариант вычисления — `(a * a + b * b) ** 0.5`:

```
$ python -m timeit -s 'a=2; b=3' '(a*a + b*b) ** .5'
5000000 loops, best of 5: 72.8 nsec per loop
```

На этот раз среднее время выполнения составило 72,8 наносекунды на итерацию. Это еще раз подтверждает, что второй способ — более быстрый.

Модуль `timeit` автоматически подбирает количество итераций так, чтобы общее время выполнения составляло не менее 0,2 секунды. Благодаря этому удается повысить точность измерений, уменьшив влияние накладных расходов на замеры.

Если вас интересует более глубокий анализ производительности Python-кода, то изучите информацию об инструментах pyperf (https://github.com/psf/pyperf) и pyperformance (https://github.com/python/pyperformance).

Резюме

В этой короткой главе мы рассмотрели различные методы и рекомендации по отладке, устранению неполадок и профилированию кода. Отладкой приходится заниматься любому разработчику, и важно уметь выполнять ее эффективно. При правильном подходе она может быть не только полезной, но и увлекательной.

Вы изучили способы анализа кода: использование пользовательских функций, логирование, отладчики, расшифровку трассировок, профилирование и утверждения. Большинство приемов были показаны на простых примерах. Вдобавок мы обсудили полезные рекомендации, которые пригодятся в сложных ситуациях.

Всегда *сохраняйте спокойствие и сосредоточенность* — это позволит облегчить выполнение отладки. Это тоже навык, который необходимо развивать, и он, пожалуй, самый важный. Взбудораженное и утомленное мышление не способно работать логично и творчески. Поэтому если чувствуете, что зашли в тупик, то совершите короткую прогулку или вздремните — просто переключитесь. Нередко решение приходит именно после перерыва.

В следующей главе мы поговорим об аннотациях типов и статических проверках типов — инструментах, которые могут помочь снизить вероятность возникновения некоторых видов ошибок.

12 Аннотации типов

Познание себя — начало всякой мудрости.

Аристотель

Аннотации типов, пожалуй, самое значительное изменение в Python со времен версии 2.2, когда были объединены типы и классы.

Система типов в Python

Python — язык с *динамической* и *строгой типизацией*.

Строгая типизация означает, что Python не выполняет неявные преобразования типов, которые могли бы привести к неожиданному поведению. Рассмотрим следующий пример на PHP:

```
<?php
$a = 2;
$b = "2";
echo $a + $b; // выводит: 4
?>
```

В PHP переменные обозначаются с префиксом `$`. В этом примере переменной `$a` присвоено целое число 2, а переменной `$b` — строка `"2"`. При попытке сложить их PHP автоматически приводит строку к числу. Такой механизм называется *жонглированием типами*. На первый взгляд это удобно, но слабая типизация PHP может легко привести к ошибкам. Если попытаться сделать то же самое в Python:

```
# example.strongly.typed.py
a = 2
b = "2"
print(a + b)
```

то результат будет совсем другим:

```
$ python ch12/example.strongly.typed.py
Traceback (most recent call last):
```

```
  File "ch12/example.strongly.typed.py", line 3, in <module>
    print(a + b)
          ~~^~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Выполнение этого кода вызовет исключение `TypeError`, так как Python не допускает неявного сложения строки и числа.

Одновременно с этим Python — язык с *динамической типизацией*. Это означает, что тип переменной определяется во время выполнения и в коде его явно указывать не нужно.

В противоположность этому языки наподобие C++, Java, C# и Swift — *статически типизированные*. В них при объявлении переменной требуется указывать ее тип. Например, в Java часто можно увидеть такие объявления переменных:

```
String name = "John Doe";
int age = 60;
```

У каждого подхода есть плюсы и минусы, и вопрос не в том, какой язык лучше. Python создавался как лаконичный, чистый и элегантный язык. Одной из особенностей такого дизайна стала поддержка *утиной типизации*.

Утиная типизация

Одним из понятий, популяризированных Python, можно считать *утиную типизацию*. По сути, она означает следующее: не так важно, к какому типу или классу принадлежит объект, — главное, какие методы он реализует и какие операции поддерживает. Как говорится, «если что-то выглядит как утка, плавает как утка и крякает как утка — скорее всего, это утка».

Утиная типизация в Python достаточно широко применяется, поскольку ее поддерживает динамическая система типов. Такой подход делает код более гибким и упрощает его повторное использование. Рассмотрим пример:

```
# duck.typing.py
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * (self.radius**2)

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

```
def print_shape_info(shape):
    print(f"{shape.__class__.__name__} area: {shape.area()}")

circle = Circle(5)
rectangle = Rectangle(4, 6)

print_shape_info(circle)     # Площадь Circle: 78.53975
print_shape_info(rectangle)  # Площадь Rectangle: 24
```

Функция `print_shape_info()` не интересуется конкретным типом объекта `shape`. Ей важно лишь то, чтобы у объекта был метод `area()`.

История появления аннотаций типов

Гибкий подход к типам — одна из особенностей, благодаря которой Python получил широкое распространение. Однако начиная с версии 3 языка началась его постепенная и продуманная эволюция в сторону добавления средств статической типизации — при сохранении его динамической природы.

Первым шагом в этом направлении стало появление аннотаций функций в Python 3.0 — благодаря PEP 3107 (https://peps.python.org/pep-3107/). Это нововведение позволило добавлять к параметрам и возвращаемым значениям произвольные метаданные. Изначально аннотации использовались только как средство документации и не имели никакого смысла для самого интерпретатора. Тем не менее именно они заложили основу для появления явных аннотаций типов.

Настоящим стартом аннотаций типов стал PEP 484 (https://peps.python.org/pep-0484), который появился в версии Python 3.5. Он формализовал использование аннотаций типов, развивая синтаксис, предложенный в PEP 3107. В этом стандарте было описано, как указывать типы параметров функций, возвращаемых значений и переменных.

Благодаря этому стало возможно выполнять необязательную статическую проверку типов. Разработчики получили такие инструменты, как `mypy`, позволяющие находить ошибки, связанные с типами, еще до запуска программы.

Начиная с версии Python 3.6 аннотации типов можно добавлять к переменным не только внутри функций. Это стало реальным после выхода PEP 526 (https://peps.python.org/pep-0526/). Благодаря этому нововведению типы стало можно указывать явно во всем коде — для атрибутов классов, переменных на уровне модуля и других случаев. Такой шаг значительно расширил возможности системы аннотаций типов и упростил статический анализ кода.

Позднее аннотации типов продолжили развиваться — новые версии Python и соответствующие PEP улучшали и уточняли поведение типовой системы.

- PEP 544 (https://peps.python.org/pep-0544/), появившийся в Python 3.8, ввел понятие протоколов. Они позволили использовать утиную типизацию при статической проверке кода.

- PEP 585 (https://peps.python.org/pep-0585/), добавленный в Python 3.9, стал важным шагом. Он интегрировал аннотации типов прямо в стандартные коллекции Python. Это позволило отказаться от импорта типов из модуля `typing` при работе с такими структурами, как `list`, `dict` и др.
- PEP 586 (https://peps.python.org/pep-0586/), реализованный в Python 3.8, добавил литеральные типы, которые позволяют указывать в аннотациях конкретные значения, а не только типы.
- PEP 589 (https://peps.python.org/pep-0589/), также в Python 3.8, представил `TypedDict` — способ точно описывать словари с фиксированными ключами и их типами.
- PEP 604 (https://peps.python.org/pep-0604/), добавленный в Python 3.10, упростил синтаксис объединений типов. Благодаря этому аннотации стали более компактными и читабельными.

Среди других важных документов PEP, связанных с системой аннотаций типов, можно выделить следующие.

- PEP 561 (https://peps.python.org/pep-0561/) описывает, как распространять пакеты, поддерживающие проверку типов.
- PEP 563 (https://peps.python.org/pep-0563/) изменил порядок обработки аннотаций: они больше не вычисляются в момент определения функции. Такой отложенный режим стал поведением по умолчанию начиная с Python 3.10.
- PEP 593 (https://peps.python.org/pep-0593/) позволил дополнять аннотации типов произвольными метаданными. Эти данные могут использоваться сторонними инструментами.
- PEP 612 (https://peps.python.org/pep-0612/) ввел спецификации параметров, которые позволяют описывать более сложные случаи аннотаций типов, особенно полезные при создании декораторов, изменяющих сигнатуру функций.
- PEP 647 (https://peps.python.org/pep-0647/) представил сужения типов (TypeGuard), функции, которые помогают уточнять типы внутри условных блоков кода.
- PEP 673 (https://peps.python.org/pep-0673/) добавил специальный тип `Self`, позволяющий ссылаться на текущий тип экземпляра внутри тела класса и возвращаемых значений методов. Это сделало аннотации внутри классов более выразительными и точными.

Развитие системы аннотаций типов в Python продиктовано стремлением повысить надежность и масштабируемость кода, сохранив при этом гибкость и динамичную природу языка.

Несмотря на растущую популярность аннотаций типов, важно помнить слова авторов PEP 484 (Гвидо ван Россума, Юкки Лехтосало и Лукаша Ланги):

> «Python останется динамически типизированным языком, и у авторов нет желания когда-либо делать аннотации типов обязательными, даже по соглашению».

Философия аннотаций типов в Python в том, что выбор всегда остается за разработчиком. Использовать типы или нет — зависит от целей, предпочтений и контекста проекта.

Теперь рассмотрим основные выгоды, которые можно получить благодаря использованию аннотаций типов.

Преимущества аннотаций типов

Использование аннотаций типов приносит сразу несколько ощутимых преимуществ: повышается качество кода, удобство его сопровождения и эффективность работы разработчиков.

- *Лучшая читабельность и документирование кода.* Аннотации типов выступают как встроенная документация: они сразу показывают, какие аргументы ожидает функция и что она возвращает. Это помогает быстрее разобраться в коде, не читая длинные комментарии или не вникая в реализацию.
- *Более раннее выявление ошибок.* Инструменты статической проверки типов, такие как `mypy`, позволяют находить ошибки до запуска программы. Благодаря этому некоторые баги можно устранить заранее, не дожидаясь, пока они проявятся в работе.
- *Повышенное удобство работы в IDE и автодополнение.* Современные среды разработки используют аннотации типов для улучшения автодополнения, навигации по коду и более надежного рефакторинга. Кроме того, IDE могут подсказывать, какие методы и атрибуты доступны у объекта, исходя из его типа.
- *Упрощение совместной работы и ревью кода.* Аннотации повышают наглядность, поэтому читать чужой код или анализировать запросы на изменения становится проще — особенно если время ограничено.
- *Гибкость и повторное использование кода.* Система аннотаций в Python поддерживает обобщенные типы, пользовательские типы и протоколы. Это помогает проектировать более универсальные и структурированные решения.

Еще одно важное достоинство аннотаций — их можно внедрять постепенно. На практике это и делается: сначала типизируют только самые критические участки кода, а затем по мере необходимости расширяют охват.

Использование аннотаций типов

Теперь, когда у вас есть общее представление о динамической природе Python и о подходе к аннотациям типов, мы можем перейти к примерам и более подробному разбору того, как аннотации применяются на практике.

В процессе рассмотрения базовых типов, доступных для аннотирования кода, вы можете заметить некоторые отличия от того, к чему привыкли, особенно если вы уже работали с аннотациями в более ранних версиях Python. Это нормально: система аннотаций в языке продолжает развиваться и синтаксис может отличаться в зависимости от версии. В этой главе мы будем придерживаться правил, актуальных для Python 3.12.

Аннотирование функций

В Python можно аннотировать как параметры функций, так и возвращаемые значения. Начнем с простого примера — функции приветствия:

```
# annotations/basic.py
def greeter(name):
    return f"Hello, {name}!"
```

Эта функция принимает имя и возвращает приветствие. Мы можем аннотировать ее, указав, что ожидаем строку в качестве имени и возвращаем тоже строку:

```
# annotations/basic.py
def greeter_annotated(name: str) -> str:
    return f"Hello, {name}!"
```

Как видно по выделенному фрагменту, синтаксис аннотаций довольно простой. Тип параметра указывается после имени, через двоеточие. Возвращаемое значение аннотируется с помощью стрелки (`->`), за которой следует тип возвращаемого объекта.

Теперь добавим еще один параметр — возраст, который должен быть целым числом:

```
# annotations/basic.py
def greeter_annotated_age(name: str, age: int) -> str:
    return f"Hello, {name}! You are {age} years old."
```

Как вы, вероятно, и ожидали, мы использовали ту же самую форму записи, только указали `int` вместо `str` для параметра `age`.

Если вы используете современную IDE, то попробуйте ввести имя переменной `name` или `age`, а затем поставить точку. Редактор предложит только те методы и свойства, которые соответствуют указанному типу.

Это был базовый пример, чтобы показать синтаксис аннотаций. Не обращайте внимания на такие названия, как `greeter_annotated()` или `greeter_annotated_age()`, — они не особо удачны, но помогают более явно показать различия между примерами.

Далее мы расширим эту тему и покажем реальные возможности аннотаций типов в Python.

Тип Any

`Any` — особый тип. Если у функции не указаны аннотации типов для параметров или возвращаемого значения, то по умолчанию используется `Any`. Поэтому следующие объявления функций полностью эквивалентны:

```
# annotations/any.py
from typing import Any

def square(n):
    return n**2

def square_annotated(n: Any) -> Any:
    return n**2
```

Тип `Any` бывает полезен в некоторых ситуациях — например, при аннотировании контейнеров данных, декораторов функций или когда функция должна принимать значения разных типов. Вот простой пример:

```
# annotations/any.py
from typing import Any
def reverse(items: list) -> list:
    return list(reversed(items))

def reverse_any(items: list[Any]) -> list[Any]:
    return list(reversed(items))
```

И здесь оба варианта полностью эквивалентны. Единственное различие: во втором случае, используя `Any`, мы явно указываем, что элементы списка могут быть объектами любого типа[1].

Псевдонимы типов

В аннотациях можно использовать практически любой тип из Python. Кроме того, модуль `typing` предлагает ряд конструкций, которые позволяют расширить выразительность типов.

Одна из таких конструкций — *псевдонимы типов*, которые создаются с помощью инструкции `type`. В результате получается объект типа `TypeAliasType`. Подобные псевдонимы помогают упростить чтение кода. Рассмотрим пример:

---

[1] Использовать Any нужно с осторожностью, потому что это лишает средства проверки типов возможности до запуска программы обнаруживать потенциально ошибочные операции, связанные с типизацией, и может привести к аварийному завершению программы. — *Примеч. науч. ред.*

```
# annotations/type.aliases.py
type DatabasePort = int

def connect_to_database(host: str, port: DatabasePort):
    print(f"Connecting to {host} on port {port}...")
```

Как видите, мы можем определить собственные псевдонимы типов. С точки зрения статического анализатора кода `DatabasePort` и `int` будут интерпретироваться одинаково.

На таком искусственном примере это, может быть, неочевидно, однако в реальном проекте псевдонимы типов улучшают читабельность кода и упрощают его рефакторинг. Представьте, что у вас есть несколько функций, принимающих параметр типа `DatabasePort`. Если в дальнейшем вы решите представить порт как строку, то достаточно будет изменить всего одну строку — ту, в которой определен `DatabasePort`. Если бы вы использовали `int` напрямую, то пришлось бы переписывать все объявления функций.

Специальные формы

Специальные формы можно использовать как типы в аннотациях. Все они поддерживают подписку с помощью квадратных скобок `[]`, но каждая имеет собственную семантику. Здесь мы рассмотрим только типы `Optional` и `Union`. Полный список специальных форм вы найдете в официальной документации по адресу https://docs.python.org/3/library/typing.html#special-forms.

Тип Optional

В Python аргументы функции могут быть необязательными, если для них указано значение по умолчанию. Здесь стоит сделать важное уточнение. Давайте снова рассмотрим функцию `greeter` и добавим ей значение по умолчанию:

```
# annotations/optional.py
def greeter(name: str = "stranger") -> str:
    return f"Hello, {name}!"
```

Теперь у функции `greeter()` есть значение по умолчанию. Это значит, что если вызвать ее без аргументов, то она вернет строку `"Hello, stranger!"`.

В такой форме предполагается, что если аргумент `name` все-таки передается, то будет строкой. Однако иногда нам нужно, чтобы параметр явно принимал значение `None`, если аргумент не указан. Для таких случаев модуль `typing` предлагает специальную форму `Optional`:

```
# annotations/optional.py
def greeter_optional(name: Optional[str] = None) -> str:
    if name is not None:
        return f"Hello, {name}!"
    return "No-one to greet!"
```

В функции `greeter_optional()` мы не хотим возвращать приветствие, если имя не указано. Поскольку `None` не является строкой, то мы указываем, что параметр `name` может быть неопределенным, и задаем для него значение по умолчанию `None`.

Тип Union

Иногда параметр может принимать значения разных типов. Например, при подключении к базе данных порт может быть указан как целое число или как строка. В таких случаях удобно использовать специальную форму `Union`:

```
# annotations/union.py
from typing import Union

def connect_to_database(host: str, port: Union[int, str]):
    print(f"Connecting to {host} on port {port}...")
```

Здесь для параметра `port` мы хотим допустить как `int`, так и `str`. Специальная форма `Union` позволяет нам сделать это.

Начиная с Python 3.10 модуль `typing.Union` импортировать больше не нужно — вместо него можно использовать вертикальную черту (`|`) для указания объединенного типа:

```
# annotations/union.py
def connect_to_db(host: str, port: int | str):
    print(f"Connecting to {host} on port {port}...")
```

Такой синтаксис выглядит лаконичнее.

Кроме того, `Union` или его эквивалент с `|` избавляет от необходимости импортировать `Optional`, ведь `Optional[str]` — это то же самое, что `Union[str, None]`, а значит, можно просто написать `str | None`. Ниже приведен пример такого использования:

```
# annotations/optional.py
def another_greeter(name: str | None = None) -> str:
    if name is not None:
        return f"Hello, {name}!"
    return "No-one to greet!"
```

Обратите внимание, что у некоторых функций выше указаны типы параметров, но не указан тип возвращаемого значения. Это сделано для того, чтобы показать: аннотации — элемент совершенно необязательный. Мы даже можем аннотировать только часть параметров.

Обобщенные типы

Продолжим изучение возможностей, которые предоставляют аннотации типов, и разберемся с понятием *обобщенных типов* (дженериков).

Представьте, что нужно написать функцию `last()`, которая принимает список элементов любого типа и возвращает последний элемент, если он есть, или `None`.

Мы знаем, что все элементы списка имеют один и тот же тип, но заранее не знаем, какой именно. Как корректно аннотировать такую функцию? Здесь может помочь концепция обобщенных типов. Синтаксис немного сложнее, чем тот, который мы видели прежде, но разобраться в нем несложно. Вот как может выглядеть такая функция:

```
# annotations/generics.py
def last[T](items: list[T]) -> T | None:
    return items[-1] if items else None
```

Обратите особое внимание на выделенные части кода. Сначала мы указываем `[T]` после имени функции — это определяет параметр типа `T`. Затем можно задать тип параметра `items` как список объектов типа `T`, каким бы он ни был. Наконец, для возвращаемого значения также можно использовать `T`, хотя в данном случае возвращаемый тип описан как `T | None`, чтобы учесть ситуацию, когда список пуст.

В сигнатуре функции используется синтаксис обобщенных типов, однако вызов остается обычным: `last([1, 2, 3])`.

Поддержка обобщенных типов на синтаксическом уровне появилась в Python 3.12. До этого для достижения того же результата приходилось использовать фабрику `TypeVar`, как показано ниже:

```
# annotations/generics.py
from typing import TypeVar
U = TypeVar("U")

def first(items: list[U]) -> U | None:
    return items[0] if items else None
```

Обратите внимание, что в этом случае функция `first()` не объявляется как `first[U](…)`.

Благодаря новому синтаксису в Python 3.12 использовать обобщенные типы стало гораздо проще.

Аннотирование переменных

Сделаем небольшое отступление от функций и поговорим об аннотациях переменных. Пример будет довольно простым и не потребует длительных объяснений:

```
# annotations/variables.py
x: int = 10
x: float = 3.14
x: bool = True
x: str = "Hello!"
x: bytes = b"Hello!"

# Python 3.9+
x: list[int] = [7, 14, 21]
x: set[int] = {1, 2, 3}
x: dict[str, float] = {"planck": 6.62607015e-34}
```

```
# Python 3.8 и ранние версии
from typing import List, Set, Dict
x: List[int] = [7, 14, 21]
x: Set[int] = {1, 2, 3}
x: Dict[str, float] = {"planck": 6.62607015e-34}

# Python 3.10+
x: list[int | str] = [0, 1, 1, 2, "fibonacci", "rules"]

# Python 3.9 и ранние версии
from typing import Union
x: list[Union[int, str]] = [0, 1, 1, 2, "fibonacci", "rules"]
```

Здесь переменной `x` мы присваиваем разные значения — просто чтобы показать, как работает аннотирование. Как видите, синтаксис очень простой: сначала указывается имя переменной, затем ее тип (после двоеточия), а затем — значение (после знака равенства). Кроме того, приведены примеры того, как раньше приходилось аннотировать переменные в предыдущих версиях Python. Удобно, что начиная с Python 3.12 можно использовать встроенные типы напрямую, почти ничего не импортируя из модуля `typing`.

Аннотирование контейнеров

Система типов Python предполагает, что все элементы в контейнерах будут одного типа. В большинстве случаев это действительно так. Например, рассмотрим следующий код:

```
# annotations/containers.py

# Проверка типов предполагает, что все элементы списка — целые числа
a: list[int] = [1, 2, 3]

# Нельзя указать два типа для элементов списка —
# допускается только один аргумент типа
b: list[int, str] = [1, 2, 3, "four"]  # Неверно!

# Проверка типов определит, что все ключи в c — строки,
# а значения — либо целые числа, либо строки
c: dict[str, int | str] = {"one": 1, "two": "2"}
```

Как видно из примера, `list` принимает один параметр типа. В этом контексте объединенный тип, такой как `int | str` в аннотации переменной `c`, все еще считается одним типом. Однако для переменной `b` средство проверки типов выдаст предупреждение. Это связано с тем, что списки в Python обычно используют для хранения произвольного количества элементов одного типа.

Контейнеры, все элементы которых одного типа, называются однородными.

Обратите внимание: синтаксис схож, но `dict` требует указания двух типов — одного для ключей и одного для значений.

Аннотирование кортежей

В отличие от большинства других контейнеров кортежи часто содержат фиксированное количество элементов, причем для каждого элемента предполагается определенный тип. Такие кортежи называются *неоднородными*. Именно поэтому система типов обрабатывает кортежи особым образом.

Существует несколько основных способов аннотировать кортежи:

- кортежи *фиксированной* длины, которые делятся на:
  - кортежи без именованных полей;
  - кортежи с именованными полями;
- кортежи *произвольной* длины.

Кортежи фиксированной длины без именованных полей

Рассмотрим пример:

```
# annotations/tuples.fixed.py
# Кортеж a содержит один строковый элемент
a: tuple[str] = ("hello",)

# Кортеж b состоит из двух элементов: целого числа и строки
b: tuple[int, str] = (1, "one")

# Ошибка проверки типов: указано, что кортеж должен содержать
# один элемент, но фактически в нем три значения
c: tuple[float] = (3.14, 1.0, -1.0)  # Неверно!
```

Здесь переменные `a` и `b` аннотированы правильно. Однако `c` аннотирована некорректно, поскольку указана длина `1`, а в самом кортеже — три элемента.

Кортежи фиксированной длины с именованными полями

Если в кортеже больше одного-двух элементов или если он используется в нескольких частях кода, то удобно аннотировать его с помощью `typing.NamedTuple`. Вот простой пример:

```
# annotations/tuples.named.py
from typing import NamedTuple

class Person(NamedTuple):
    name: str
    age: int
```

```
fab = Person("Fab", 48)
print(fab)      # Person(name='Fab', age=48)
print(fab.name) # Fab
print(fab.age)  # 48
print(fab[0])   # Fab
print(fab[1])   # 48
```

Как видно по результатам вызовов `print()`, это эквивалентно объявлению кортежа, с которым вы уже знакомы по главе 2:

```
# annotations/tuples.named.py
import collections
Person = collections.namedtuple("Person", ["name", "age"])
```

Использование `typing.NamedTuple` позволяет не только задать аннотацию типа для кортежа, но и при необходимости указать значения по умолчанию:

```
# annotations/tuples.named.py
class Point(NamedTuple):
    x: int
    y: int
    z: int = 0

p = Point(1, 2)
print(p)   # Point(x=1, y=2, z=0)
```

Обратите внимание: в примере при создании переменной `p` мы не передали третий аргумент, но переменной `z` все равно было присвоено значение `0`, как и ожидалось.

Кортежи произвольной длины

Если нужно аннотировать кортеж произвольной длины, в котором все элементы одного типа, то можно воспользоваться специальным синтаксисом. Это бывает полезно, например, когда кортеж используется как неизменяемая последовательность. Вот несколько примеров:

```
# annotations/tuples.any.length.py
from typing import Any

# Многоточие указывает, что кортеж может содержать любое количество элементов
a: tuple[int, ...] = (1, 2, 3)

# Все следующие варианты допустимы, поскольку кортеж может иметь
# любое количество элементов
a = ()
a = (7,)
a = (7, 8, 9, 10, 11)

# А вот это ошибка: кортеж должен содержать только целые числа
a = ("hello", "world")
```

```
# Можно явно указать, что кортеж должен быть пустым
b: tuple[()] = ()

# Наконец, если аннотировать кортеж так:
c: tuple = (1, 2, 3)
# Проверяющий типы будет считать, что написано вот так:
c: tuple[Any, ...] = (1, 2, 3)
# Поэтому все следующие варианты считаются допустимыми:
c = ()
c = ("hello", "my", "friend")
```

Как видите, существует множество способов аннотировать кортеж. Насколько строго это делать — зависит от вас. Главное — придерживаться единого подхода в рамках всей кодовой базы. Помните, что аннотации должны приносить пользу. Если вы пишете библиотеку, которую будут использовать другие разработчики, то разумно делать аннотации максимально точными. Но чрезмерная строгость без явной причины может только мешать — особенно в тех случаях, когда такая точность не требуется.

Абстрактные базовые классы

В более ранних версиях Python модуль `typing` предоставлял обобщенные версии коллекций. Например, списки, кортежи, множества и словари можно было аннотировать с помощью обобщенных конкретных типов `List`, `Tuple`, `Set`, `Dict` и т. д.

Начиная с Python 3.9 использование этих обобщенных коллекций признано устаревшим. Теперь вместо них можно использовать соответствующие встроенные типы. То есть, например, список можно аннотировать просто как `list`, без `typing.List`.

Кроме того, в документации подчеркивается, что такие обобщенные коллекции следует использовать для аннотаций возвращаемых значений, тогда как для параметров предпочтительно применять абстрактные коллекции. Например, для аннотации последовательностей, которые могут быть доступны только для чтения и являются изменяемыми, таких как `list`, `tuple`, `str`, `bytes` и др., необходимо использовать `collections.abc.Sequence`.

Такой подход согласуется с *законом Постела*, также известным как *принцип надежности*, суть которого можно выразить так:

> «Будь либерален в том, что принимаешь, и консервативен в том, что отправляешь».

Формулировка «будь либерален в том, что принимаешь» означает, что аннотации параметров не стоит делать слишком жесткими. Если функция принимает параметр `items` и все, что она с ним делает, — перебирает элементы или обращается к ним по индексу, то не имеет значения, список это или кортеж.

В таких случаях не стоит аннотировать параметр как `tuple` или `list`. Вместо этого следует использовать `collections.abc.Sequence`, чтобы допускались и кортежи, и списки.

Представьте, что вы аннотировали `items` как `tuple`, но затем в процессе рефакторинга начали передавать список. Аннотация окажется некорректной, поскольку теперь `items` — уже не кортеж. Если бы с самого начала использовалась аннотация `collections.abc.Sequence`, то никаких изменений в функции делать бы не пришлось — подходили бы оба варианта.

Вот пример, который проясняет эту идею:

```
# annotations/collections.abcs.py
from collections.abc import Mapping, Sequence

def average_bad(v: list[float]) -> float:
    return sum(v) / len(v)

def average(v: Sequence[float]) -> float:
    return sum(v) / len(v)

def greet_user_bad(user: dict[str, str]) -> str:
    return f"Hello, {user['name']}!"

def greet_user(user: Mapping[str, str]) -> str:
    return f"Hello, {user['name']}!"
```

Функции из примера хорошо иллюстрируют разницу. В `average_bad()` при передаче значения `v` в виде кортежа произойдет несоответствие с аннотацией, так как указано, что параметр должен быть списком. А вот `average()` не имеет этой проблемы, поскольку использует более общий тип.

Тот же подход применим и к функциям `greet_user_bad()` и `greet_user()` — вторая функция использует более гибкую аннотацию и не ограничивает вызывающего в типе передаваемого аргумента.

Возвращаясь к закону Постела: когда речь идет о возвращаемых значениях, наоборот, стоит быть консервативными, то есть точными. Тип возвращаемого значения должен максимально ясно указывать, что именно возвращает функция.

Это особенно важно для вызывающего кода, которому нужно точно понимать, с каким типом объекта он будет работать после вызова функции.

Вот еще один простой пример из того же файла, который должен помочь лучше понять идею:

```
# annotations/collections.abcs.py
def add_defaults_bad(
    data: Mapping[str, str]
) -> Mapping[str, str]:
    defaults = {"host": "localhost", "port": "5432"}
    return {**defaults, **data}
```

```
def add_defaults(data: Mapping[str, str]) -> dict[str, str]:
    defaults = {"host": "localhost", "port": "5432"}
    return {**defaults, **data}
```

В этих двух функциях мы добавляем фиктивные значения по умолчанию к словарю `data`, если в нем отсутствуют ключи `"host"` и `"port"`. Функция `add_defaults_bad()` аннотирована с возвращаемым типом `Mapping`. Проблема здесь в том, что такой тип слишком обобщен. Объекты наподобие `dict`, а также родственные ему типы из модуля `collections` (`defaultdict`, `Counter`, `OrderedDict`, `ChainMap`, `UserDict` и др.) реализуют интерфейс `Mapping`. В результате для вызывающего кода становится неясно, что именно вернет функция.

А вот функция `add_defaults()` работает лучше — в аннотации явно указано, что возвращается объект типа `dict`.

К часто используемым абстрактным базовым классам относятся `Iterable`, `Iterator`, `Collection`, `Sequence`, `Mapping`, `Set`, изменяемые версии — `MutableSequence`, `MutableMapping`, `MutableSet`, а также функциональный тип `Callable`.

Вот пример с `Iterable`:

```
# annotations/collections.abc.iterable.py
from collections.abc import Iterable

def process_items(items: Iterable) -> None:
    for item in items:
        print(item)
```

Функция `process_items()` просто перебирает элементы — значит, разумно аннотировать параметр как `Iterable`.

Более интересный пример касается типа `Callable`:

```
# annotations/collections.abc.iterable.py
from collections.abc import Callable

def process_callback(
    arg: str, callback: Callable[[str], str]
) -> str:
    return callback(arg)

def greeter(name: str) -> str:
    return f"Hello, {name}!"

def reverse(name: str) -> str:
    return name[::-1]
```

Здесь определена функция `process_callback()`, которая принимает строку `arg` и аргумент `callback` — вызываемый объект. Затем показаны две функции, каждая из которых принимает строку на вход и возвращает также строку. Аннотация `Callable[[str], str]` означает, что `callback` должен быть вызываемым объектом, принимающим и возвращающим строку.

Когда мы вызываем `process_callback()` с этими функциями, то результат соответствует комментариям в коде:

```
# annotations/collections.abc.iterable.py
print(process_callback("Alice", greeter)) # Hello, Alice!
print(process_callback("Alice", reverse)) # ecilA
```

На этом обзор абстрактных базовых классов завершен.

Особые формы типизации

В модуле `typing` существует категория объектов, называемых *особыми формами типизации*. Это интересные и полезные конструкции, и хотя бы с наиболее распространенными из них стоит быть знакомыми. Один из таких примеров вы уже встречали — `Any`.

Перечислим другие интересные примеры.

- `AnyStr` — используется для аннотации функций, которые могут принимать аргументы типа `str` или `bytes`, но не допускают их совместного использования. Это так называемая ограниченная переменная типа: допустим только один из указанных вариантов — либо `str`, либо `bytes`.
- `LiteralString` — специальный тип, охватывающий только строковые литералы, заданные напрямую в коде.
- `Never`/`NoReturn` — обозначает, что функция никогда не возвращает значение — например, если всегда выбрасывает исключение.
- `TypeAlias` — ранее служил для задания псевдонимов типов, но считается устаревшим. Вместо него теперь следует использовать оператор `type`.

Наконец, отдельного внимания заслуживает тип `Self`, о нем и поговорим далее.

Тип Self

Тип был добавлен в версии Python 3.11. Это специальная форма типизации, обозначающая текущий охватывающий класс. Рассмотрим пример:

```
# annotations/self.py
from typing import Self
from collections.abc import Iterable
from dataclasses import dataclass

@dataclass
class Point:
    x: float = 0.0
    y: float = 0.0
    z: float = 0.0

    def magnitude(self) -> float:
        return (self.x**2 + self.y**2 + self.z**2) ** 0.5
```

```
    @classmethod
    def sum_points(cls, points: Iterable[Self]) -> Self:
        return cls(
            sum(p.x for p in points),
            sum(p.y for p in points),
            sum(p.z for p in points),
        )
```

Здесь создается простой класс `Point`, представляющий трехмерную точку в пространстве. Чтобы продемонстрировать использование типа `Self`, мы определили в классе метод класса `sum_points()`, который принимает итерируемый объект `points` и возвращает новый объект `Point`, координаты которого — суммы соответствующих координат всех точек из `points`.

Параметр `points` аннотирован как `Iterable[Self]`, и то же самое указано для возвращаемого значения метода. До появления типа `Self` приходилось явно создавать уникальную переменную типа (`TypeVar`) для каждой отдельной реализации. Пример такого подхода можно найти в официальной документации (https://docs.python.org/3/library/typing.html#typing.Self).

Обратите внимание: в соответствии с соглашением параметры self и cls не аннотируются.

Аннотация переменных параметров

Теперь посмотрим, как аннотировать переменные параметры — те, которые передаются через `*args` и `**kwargs`. Для этого используется тот же синтаксис, что и раньше. Рассмотрим такой пример:

```
# annotations/variable.parameters.py
def add_query_params(
    *urls: str, **query_params: str
) -> list[str]:
    params = "&".join(f"{k}={v}" for k, v in query_params.items())
    return [f"{url}?{params}" for url in urls]

urls = add_query_params(
    "https://example1.com",
    "https://example2.com",
    "https://example3.com",
    limit="10",
    offset="20",
    sort="desc",
)
print(urls)
# ['https://example1.com?limit=10&offset=20&sort=desc',
# 'https://example2.com?limit=10&offset=20&sort=desc',
# 'https://example3.com?limit=10&offset=20&sort=desc']
```

Здесь определена простая функция `add_query_params()`, которая добавляет параметры запроса к набору URL. Обратите внимание: в определении функции мы указали только тип элементов в кортеже `urls` и тип значений в словаре `query_params`. Аннотация `*urls: str` означает, что `urls` — это кортеж строк (`tuple[str, ...]`), а `**query_params: str` указывает на словарь `dict[str, str]`.

Протоколы

Наше путешествие по аннотациям типов мы завершим разговором о *протоколах*. В объектно-ориентированном программировании протоколы описывают набор методов, которые должен реализовать класс, не наследуя от какого-либо конкретного базового класса. Это похоже на интерфейсы в других языках, но протоколы в Python — более простые и гибкие элементы.

Идея в том, чтобы поощрять полиморфизм: если два класса реализуют один и тот же протокол, то их можно использовать взаимозаменяемо, даже если они вообще не связаны по иерархии.

Такой подход существует в Python с самого начала. Его называют *динамическим протоколом*, и он описан в главе *Data Model* документации Python (https://docs.python.org/3/reference/datamodel.html).

В контексте аннотаций типов протокол является подклассом `typing.Protocol`, который описывает интерфейс, понятный статическому анализатору кода.

Протоколы появились в Python благодаря PEP 544 (https://peps.python.org/pep-0544/) и обеспечивают структурную подстановку типов (ее еще называют *статической утиной типизацией*). Мы затрагивали эту идею в начале главы.

Совместимость объекта с протоколом определяется наличием нужных методов или атрибутов, а не фактом наследования от какого-либо класса.

Протоколы особенно полезны в тех случаях, когда сложно определить тип явно и удобнее просто указать: *«Объект должен поддерживать определенные методы или иметь нужные атрибуты»*.

Протоколы, описанные в PEP 544, обычно называют *статическими протоколами*.

Динамические и статические протоколы имеют два ключевых различия.

- Динамические допускают частичную реализацию. Это значит, что объект может реализовать лишь часть методов и все равно быть полезным. Статические протоколы требуют полной реализации. Объект должен поддерживать все методы, указанные в протоколе, даже если реально используется только один или два.
- Статические проверяются средствами статической типизации, а динамические — нет.

Список встроенных протоколов из модуля `typing` можно найти в официальной документации по адресу https://docs.python.org/3/library/typing.html#protocols. Имена всех этих протоколов начинаются с `Supports`, а далее с заглавной буквы идет название метода с двойным подчеркиванием, который они поддерживают. Вот несколько примеров:

- `SupportsAbs` — абстрактный базовый класс, содержащий метод `__abs__()`;
- `SupportsBytes` — содержит метод `__bytes__()`;
- `SupportsComplex` — содержит метод `__complex__()`.

Другие протоколы, которые раньше находились в модуле `typing`, но были перенесены в `collections.abc`, — `Iterable`, `Iterator`, `Sized`, `Container`, `Collection`, `Reversible`, `ContextManager` и др.

Полный список можно найти в документации `mypy` по адресу https://mypy.readthedocs.io/en/stable/protocols.html#predefined-protocols-reference.

Теперь, когда вы понимаете, что такое протокол в контексте аннотаций типов, рассмотрим пример создания простого собственного протокола:

```
# annotations/protocols.py
from typing import Iterable, Protocol

class SupportsStart(Protocol):
    def start(self) -> None: ...

class Worker:  # Класс не наследуется от SupportsStart
    def __init__(self, name: str) -> None:
        self.name = name

    def start(self) -> None:
        print(f"Starting worker {self.name}")

def start_workers(workers: Iterable[SupportsStart]) -> None:
    for worker in workers:
        worker.start()

workers = [Worker("Alice"), Worker("Bob")]
start_workers(workers)
```

В этом коде мы определяем протокол `SupportsStart` с одним методом — `start()`. Чтобы сделать его статическим протоколом, `SupportsStart` наследуется от `Protocol`. Самое интересное начинается позже — когда создается класс `Worker`. Обратите внимание, что ему не нужно наследоваться от `SupportsStart`. Класс `Worker` просто должен соответствовать протоколу, то есть иметь метод `start()`.

Кроме того, есть функция `start_workers()`, принимающая параметр `workers`, аннотированный как `Iterable[SupportsStart]`. Этого достаточно, чтобы использовать протокол. В примере определены два объекта — `Alice` и `Bob`, которые передаются в функцию `start_workers()`.

При запуске этого кода вывод будет таким:

```
Starting worker Alice
Starting worker Bob
```

Теперь представим, что мы хотим не только запускать объект, но и останавливать его. Это уже более интересный случай — и отличная возможность поговорить о наследовании протоколов. Рассмотрим пример:

```
# annotations/protocols.subclassing.py
from typing import Iterable, Protocol

class SupportsStart(Protocol):
    def start(self) -> None: ...

class SupportsStop(Protocol):
    def stop(self) -> None: ...

class SupportsWorkCycle(SupportsStart, SupportsStop, Protocol):
    pass

class Worker:
    def __init__(self, name: str) -> None:
        self.name = name

    def start(self) -> None:
        print(f"Starting worker {self.name}")

    def stop(self) -> None:
        print(f"Stopping worker {self.name}")

def start_workers(workers: Iterable[SupportsWorkCycle]) -> None:
    for worker in workers:
        worker.start()
        worker.stop()

workers = [Worker("Alice"), Worker("Bob")]
start_workers(workers)
```

Здесь видно, что протоколы можно комбинировать так же, как миксины. Но есть одно важное различие: если мы наследуемся от другого протокола, как в случае `SupportsWorkCycle`, то все равно должны явно указать `Protocol` в списке базовых классов. Если этого не сделать, то статический анализатор выдаст ошибку. Это связано с тем, что наследование от существующего протокола еще не делает новый класс протоколом автоматически — он просто становится обычным классом или абстрактным базовым, реализующим указанные методы.

Больше информации о протоколах можно найти в документации `mypy`: https://mypy.readthedocs.io/en/stable/protocols.html. Это самый популярный в сообществе Python статический анализатор типов. О нем и поговорим далее.

Статический анализатор типов mypy

Существует несколько статических анализаторов типов для Python. Самые популярные на сегодняшний день описаны ниже.

- `mypy` — разработан для тесной интеграции с аннотациями типов, описанными в PEP 484. Поддерживает постепенную типизацию, хорошо работает с существующим кодом и имеет подробную документацию. Больше информации доступно на сайте https://mypy.readthedocs.io.
- `Pyright` — быстрый анализатор от Microsoft, оптимизированный для работы в Visual Studio Code. Проводит инкрементный анализ, благодаря чему типы проверяются очень быстро. Поддерживает как TypeScript, так и Python. Подробнее см. на https://github.com/microsoft/pyright.
- `Pylint` — универсальный инструмент для статического анализа, позволяющий выполнять проверку типов, линтинг и оценку качества кода. Предоставляет высокую степень настройки, поддержку пользовательских плагинов и подробные отчеты. Больше информации — на https://pylint.org.
- `Pyre` — разработан в Facebook. Быстрый и масштабируемый инструмент, рассчитанный на крупные проекты. Поддерживает постепенную типизацию, умеет сам выводить типы, хорошо встраивается в процессы CI. Подробнее см. на https://pyre-check.org.
- `Pytype` — разработан Google. Автоматически выводит типы и снижает потребность в явных аннотациях, которые сам же может и сгенерировать. Хорошо интегрируется с открытыми инструментами Google. Репозиторий: https://github.com/google/pytype.

В этом разделе мы решили использовать `mypy`, поскольку, как мы уже сказали, на данный момент он считается самым популярным инструментом для проверки типов.

Чтобы установить его в виртуальное окружение, выполните команду:

```
$ pip install mypy
```

Кроме того, `mypy` уже добавлен в файл с кодом для этой главы. После установки вы можете запускать его для проверки любых файлов или папок. Анализатор будет рекурсивно обходить указанные каталоги и проверять все модули Python (`*.py`). Например:

```
$ mypy program.py некая_папка другая_папка
```

У этой команды довольно много параметров. Советуем изучить их, выполнив такую команду:

```
$ mypy --help
```

Начнем с простого примера — функции без аннотаций и посмотрим, что скажет mypy, если запустить проверку на модуль, содержащий такой код:

```
# mypy_src/simple_function.py
def hypothenuse(a, b):
    return (a**2 + b**2) ** 0.5
```

Вот как может выглядеть вывод:

```
$ mypy simple_function.py
Success: no issues found in 1 source file
```

Вероятно, такой результат может показаться неожиданным, но это нормально: mypy специально сделан так, чтобы позволять постепенно добавлять аннотации в уже существующий код. Если бы он сразу выдавал предупреждение обо всех неаннотированных элементах, это бы только отпугнуло разработчиков. Поэтому по умолчанию анализатор игнорирует функции без аннотаций. Если же вы все-таки хотите, чтобы он проверил функцию hypothenuse(), то можете запустить его следующим образом (вывод отформатирован под ширину страницы книги):

```
$ mypy --strict mypy_src/simple_function.py
mypy_src/simple_function.py:4:
    error: Function is missing a type annotation [no-untyped-def]
Found 1 error in 1 file (checked 1 source file)
```

Теперь mypy сообщает, что функции не хватает аннотаций. Исправим это:

```
# mypy_src/simple_function_annotated.py
def hypothenuse(a: float, b: float) -> float:
    return (a**2 + b**2) ** 0.5
```

Запускаем анализатор еще раз:

```
$ mypy simple_function_annotated.py
Success: no issues found in 1 source file
```

Отлично — теперь функция аннотирована и проверка прошла успешно. Попробуем теперь вызвать эту функцию с разными аргументами:

```
print(hypothenuse(3, 4))                # Это допустимо
print(hypothenuse(3.5, 4.9))            # Это тоже допустимо
print(hypothenuse(complex(1, 2), 10))   # Ошибка статической проверки типов
```

Первые два вызова проходят без ошибок, но последний вызывает сообщение об ошибке:

```
$ mypy mypy_src/simple_function_annotated.py
mypy_src/simple_function_annotated.py:10:
    error: Argument 1 to "hypothenuse" has incompatible
    type "complex"; expected "float" [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

mypy сообщает, что передавать complex, где ожидается float, нельзя. Эти типы несовместимы.

Попробуем немного более сложный пример:

```
# mypy_src/case.py
from collections.abc import Iterable

def title(names: Iterable[str]) -> list[str]:
    return [name.title() for name in names]

print(title(["ALICE", "bob"]))    # ['Alice', 'Bob'] — все хорошо,
                                  # mypy не выдает предупреждение
print(title([b"ALICE", b"bob"]))  # [b'Alice', b'Bob'] — ошибка mypy
```

Функция здесь приводит каждую строку из names к заглавному формату (title() применяется ко всем элементам). Сначала мы вызываем ее со строками "ALICE" и "bob", а потом — с байтовыми строками b"ALICE" и b"bob". Оба вызова работают, поскольку и str, и bytes имеют метод title(). Однако при запуске mypy мы получаем такой результат:

```
$ mypy mypy_src/case.py
mypy_src/case.py:10:
    error: List item 0 has incompatible type "bytes";
    expected "str" [list-item]
mypy_src/case.py:10:
    error: List item 1 has incompatible type "bytes";
    expected "str" [list-item]
Found 2 errors in 1 file (checked 1 source file)
```

И снова анализатор указывает на несовместимость типов — теперь между str и bytes. Исправить это просто: можно либо изменить второй вызов функции, либо скорректировать аннотацию типа. Сделаем второе:

```
# mypy_src/case.fixed.py
from collections.abc import Iterable

def title(names: Iterable[str | bytes]) -> list[str | bytes]:
    return [name.title() for name in names]

print(title(["ALICE", "bob"]))    # ['Alice', 'Bob'] — все хорошо,
                                  # mypy не выдает предупреждение
print(title([b"ALICE", b"bob"]))  # [b'Alice', b'Bob'] — все хорошо,
                                  # mypy не выдает предупреждение
```

Теперь мы используем объединение типов str и bytes в аннотации, и mypy не выдает ошибок.

Наш совет: установите этот анализатор и попробуйте запустить его в существующем проекте. Начинайте постепенно добавлять аннотации типов и проверяйте их с помощью mypy. Так вы быстрее освоите систему аннотаций типов и сможете улучшить ваш код.

Полезные ресурсы

Мы рекомендуем хотя бы бегло просмотреть все PEP, упомянутые в начале этой главы, а также изучить полезные материалы, на которые мы ссылались по ходу повествования. Ссылки на некоторые из них приведены ниже.

- Документация по типизации в Python: https://typing.readthedocs.io/en/latest/.
- Статическая типизация в Python (официальная документация): https://docs.python.org/3/library/typing.html.
- Абстрактные базовые классы: https://docs.python.org/3/library/abc.html.
- Абстрактные базовые классы для контейнеров: https://docs.python.org/3/library/collections.abc.html.
- Документация по `mypy`: https://mypy.readthedocs.io/en/stable/.

Кроме того, краткий вводный раздел *Python Types Intro* доступен в документации к FastAPI: https://fastapi.tiangolo.com/python-types/.

FastAPI — современный фреймворк для создания API на Python. Ему посвящена глава 14, и до того, как вы приступите к ней, мы рекомендуем хотя бы бегло ознакомиться с типами.

Резюме

В этой главе мы подробно разобрали систему аннотаций типов в Python. Начав со встроенного подхода Python к типам, мы проследили, как аннотации появились начиная с Python 3 и продолжают развиваться до сих пор.

Вы узнали, какую пользу приносят аннотации типов, и научились их использовать для функций, классов и переменных. Сначала мы обсудили базовые примеры со встроенными типами, а затем перешли к более сложным темам: обобщенным типам, абстрактным базовым классам и протоколам.

В завершение мы привели несколько примеров того, как использовать анализатор `mypy`, чтобы постепенно вводить типизацию в существующий код. В конце главы были представлены полезные ссылки на ресурсы, которые при необходимости помогут вам углубить знания.

На этом теоретическая часть книги завершается. Знаний, полученных при ее прочтении, должно быть достаточно для того, чтобы вы могли уверенно двигаться дальше и решать прикладные задачи.

13 Введение в Data Science

> Если у нас есть данные, давайте посмотрим на них. Если все, что у нас есть, — это мнения, давайте остановимся на моем.
>
> *Джим Барксдейл, бывший CEO Netscape*

Data Science — широкое понятие. В разных контекстах оно может означать разные вещи — многое зависит от задач, инструментов и понимания самого этого термина. Чтобы всерьез заниматься анализом данных, нужно как минимум разбираться в математике и статистике. А далее можно изучать другие темы, такие как распознавание шаблонов, машинное обучение, и, конечно же, осваивать разнообразные языки программирования и инструменты.

В одной главе мы не сможем охватить все эти аспекты. Поэтому, чтобы сделать ее действительно полезной, мы поработаем над совместным проектом.

Примерно в 2012–2013 годах Фабрицио работал в ведущей социальной медиакомпании в Лондоне. Он провел там два года и имел честь работать с по-настоящему выдающимися людьми. Эта компания первой в мире получила доступ к Twitter Ads API, а также сотрудничала с Facebook. А это значит — обладала огромными объемами данных.

Ее аналитики вели огромное количество рекламных кампаний и буквально утопали в объемах работы. Команда, в которую входил Фабрицио, решила помочь. Разработчики начали знакомить аналитиков с Python и с теми инструментами, которые язык предлагает для работы с данными. Это было действительно интересное путешествие: Фабрицио стал наставником для нескольких сотрудников, а позже отправился в Манилу, где провел двухнедельный интенсив по Python и анализу данных для команды аналитиков, работающих в филиале компании.

Проект, представленный в этой главе, — облегченная версия финального примера, который Фабрицио показывал своим студентам в Маниле. Мы немного сократили его, чтобы он поместился в главу, и слегка адаптировали структуру

под учебные цели — но все ключевые идеи остались неизменными. Так что далее будет и интересно, и полезно.

А начнем мы... с римских богов.

IPython и Jupyter Notebook

В 2001 году Фернандо Перес, аспирант физического факультета Университета Колорадо в Боулдере, захотел улучшить стандартную оболочку Python. Ему не хватало возможностей, к которым он привык, работая с такими инструментами, как Mathematica и Maple. Результатом этих попыток стал проект *IPython*.

Все началось с небольшого сценария — улучшенной версии оболочки Python. Со временем, благодаря усилиям других разработчиков и финансовой поддержке со стороны ряда компаний, проект превратился в полноценный инструмент. Примерно через десять лет после появления IPython на его основе была создана среда Notebook, основанная на таких технологиях, как WebSockets, веб-сервер Tornado, jQuery, CodeMirror и MathJax. Для передачи сообщений между интерфейсом Notebook и ядром Python использовалась библиотека ZeroMQ.

IPython Notebook быстро стал популярным и получил множество новых возможностей: он поддерживает виджеты, параллельные вычисления, разные медиаформаты и многое другое. Более того, в какой-то момент блокнот предоставил возможность программировать на языках, отличных от Python.

Со временем проект был разделен на два: IPython сосредоточился исключительно на ядре и интерактивной оболочке, а Notebook выделился в отдельный проект — *Jupyter*. В настоящее время Jupyter позволяет выполнять интерактивные научные вычисления более чем на 40 языках. А еще чуть позже появился *JupyterLab* — полноценная веб-IDE, в которую входят блокноты Jupyter, интерактивная консоль, редактор кода и другие инструменты.

Проект из этой главы мы будем писать и запускать в Jupyter Notebook, поэтому сначала коротко объясним, что собой представляет такой блокнот. Это веб-страница с простым меню и ячейками, в которые можно вставлять и запускать код на Python. Ячейки независимы и могут запускаться по отдельности, но все они работают с одним и тем же ядром Python. Это означает, что все переменные, функции и другие объекты, которые вы определяете в одной ячейке, будут доступны во всех остальных.

Если проще, то ядро Python — это процесс, в котором запущен Python. А страница блокнота — это интерфейс, с помощью которого пользователь взаимодействует с данным процессом. Страница и ядро обмениваются сообщениями, используя высокоскоростной протокол.

Помимо визуального удобства, главная прелесть блокнотов в том, что можно выполнять скрипт по частям, и это дает огромное преимущество. Представьте

обычный скрипт Python, который подключается к базе данных, загружает их и потом работает с ними. В привычной среде вам пришлось бы каждый раз заново загружать данные при каждой новой попытке что-то изменить. А вот в блокноте вы можете загрузить данные в одной ячейке, а потом изменять и исследовать их в других ячейках, не повторяя загрузку.

Такой пошаговый подход особенно полезен в анализе данных: вы выполняете один фрагмент работы и изучаете результат, добавляете следующий фрагмент — снова проверяете и т. д.

А при прототипировании блокноты вообще незаменимы — результаты видны сразу, никаких дополнительных действий не нужно.

Если вы хотите узнать больше об этих инструментах, то загляните на сайты ipython.org и jupyter.org.

Чтобы начать работу, мы подготовили простой блокнот с функцией `fibonacci()`, возвращающей список всех чисел Фибоначчи, которые меньше заданного *N*. Выглядит он следующим образом (рис. 13.1).

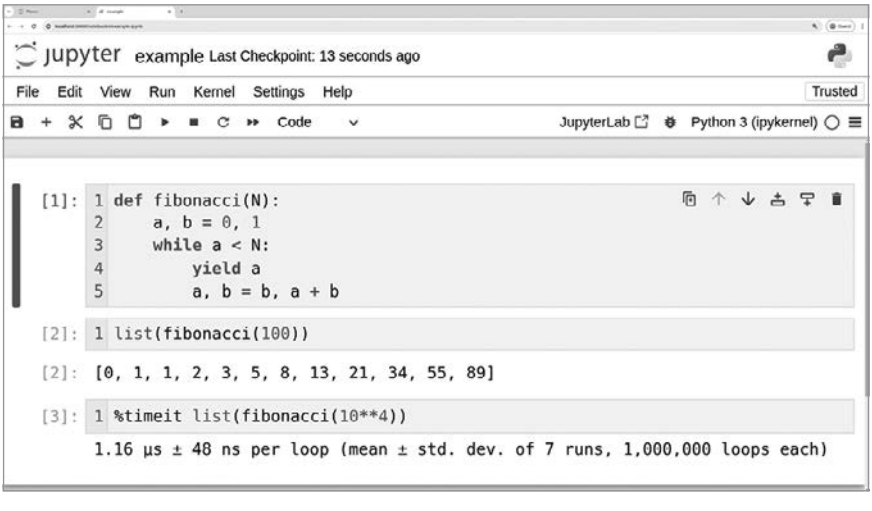


Рис. 13.1. Страница блокнота Jupyter

Каждая ячейка в блокноте имеет метку в квадратных скобках, например [1]. Если скобки пустые, значит, ячейка еще не запускалась. Если указан номер, значит, ячейка уже была выполнена, и число показывает, в каком порядке это произошло. Если вместо номера указана звездочка ([*]), значит, ячейка выполняется прямо сейчас.

На снимке экрана видно, что в первой ячейке мы определили функцию `fibonacci()` и выполнили ее. Это привело к тому, что имя `fibonacci` стало доступно в глобальной области видимости блокнота и теперь эту функцию можно вызывать в любой другой ячейке. Во второй ячейке мы вызываем `list(fibonacci(100))`,

и результат сразу появляется под ячейкой [2] — выходные данные каждой ячейки помечаются тем же номером, что и она сама. В третьей ячейке показана одна из так называемых магических команд — `%timeit`. Она запускает код несколько раз и выводит среднее время выполнения. Эта команда реализована с помощью модуля `timeit`, о котором мы уже кратко говорили в главе 11.

Вы можете запускать любую ячейку сколько угодно раз и в любом порядке. Ячейки — очень гибкий инструмент. Бывают также raw-ячейки, содержащие обычный текст без форматирования. А в markdown-ячейках удобно писать пояснения, заголовки и другие форматированные тексты прямо в блокноте.

Markdown — легковесный язык разметки с простым текстовым синтаксисом. Он может быть преобразован в HTML или другие форматы.

У ячеек есть еще одна удобная особенность: если в ячейке на последней строке что-то возвращается, эта информация выводится автоматически. То есть вам не нужно писать `print(...)`, чтобы посмотреть результат, — он и так отобразится.

Использование Anaconda

Как обычно, все библиотеки, необходимые для выполнения примеров главы, можно установить с помощью файла `requirements.txt`, который находится в коде к ней. Однако стоит признать: иногда установка библиотек для анализа данных — настоящее испытание. Если в вашем виртуальном окружении возникают сложности с установкой нужных пакетов, то можно воспользоваться альтернативой — установить Anaconda. Это бесплатный дистрибутив Python и R, имеющий открытый исходный код и созданный специально для задач анализа данных и машинного обучения. Цель Anaconda — упростить управление пакетами и настройку сред. Скачать ее можно на сайте anaconda.org. После установки вы можете, используя интерфейс Anaconda, создать виртуальное окружение и установить в него все пакеты из файла `requirements.in`, который также входит в исходный код к главе.

Запуск Jupyter Notebook

Когда все необходимые библиотеки установлены, можно запустить Jupyter Notebook с помощью команды:

```
$ jupyter notebook
```

Если вы устанавливали зависимости с помощью Anaconda, то запуск блокнота можно выполнить и через интерфейс данного дистрибутива. В любом случае в браузере откроется страница http://localhost:8888/ (порт может быть другим).

Кроме того, можно запустить JupyterLab — либо через интерфейс Anaconda, либо вот так:

```
$ jupyter lab
```

Откроется новая вкладка браузера с интерфейсом JupyterLab.

Обязательно изучите оба интерфейса. Создайте новый блокнот или откройте пример `example1.ipynb` из исходников к этой главе. Посмотрите, какой интерфейс вам подходит больше, и немного освойтесь в нем, прежде чем читать дальше. В исходном коде к главе есть сохраненная рабочая среда JupyterLab — файл `ch13.jupyterlab-workspace`. Там собраны все блокноты, которые мы будем использовать дальше. Вы можете работать в JupyterLab или использовать классический интерфейс Notebook — выбирайте вариант, удобный для вас.

Если вы работаете в современной IDE (например, VS Code), то, скорее всего, сможете установить плагин и запускать блокнот прямо из нее.

Чтобы вам было проще ориентироваться, для всех примеров кода в этой главе будет указан номер ячейки, к которой они относятся.

И еще один совет: если освоите горячие клавиши (в классическом блокноте — в меню Help, в JupyterLab — в разделе Settings ▸ Keyboard Shortcuts), то сможете перемещаться между ячейками и работать с их содержимым, не используя мышь. Это серьезно ускорит вашу работу.

Теперь перейдем к самой интересной части этой главы — к данным.

Работа с данными

Обычно такая работа выполняется следующим образом: вы получаете данные, очищаете и преобразуете их, а затем анализируете и представляете результат — в виде значений, таблиц, графиков и т. п. Мы хотим, чтобы вы могли пройти все эти этапы, не впадая во внешнюю зависимость от поставщика данных, поэтому будем выполнять следующие действия.

1. Создадим данные, симулируя ситуацию, в которой они поступают в неидеальном виде и не готовы к немедленной обработке.
2. Очистим данные и передадим их в основной инструмент, с которым будем работать в этом проекте, — это объект `DataFrame` из библиотеки `pandas`.
3. Преобразуем и обработаем данные в `DataFrame`.
4. Сохраним `DataFrame` в файл в нескольких форматах.
5. Проанализируем данные и получим из них осмысленные результаты.

Подготовка блокнота

Начнем с генерации данных. Для этого откроем блокнот `ch13-dataprep`. Ячейка № 1 отвечает за все нужные импорты:

```
#1
import json
import random
from datetime import date, timedelta

import faker
```

Из всего этого набора вы еще не встречали только два модуля: `random` и `faker`. `random` — стандартный модуль Python для генерации псевдослучайных чисел. `faker` — сторонняя библиотека, позволяющая генерировать фиктивные данные. Она особенно полезна в тестах и при создании фикстур: можно сгенерировать имена, адреса электронной почты, номера телефонов, данные кредитных карт и многое другое.

Подготовка данных

Мы хотим получить следующую структуру данных: список объектов-пользователей, при этом каждый пользователь связан с несколькими объектами-кампаниями. Напоминаем, что в Python все является объектом, поэтому в данном случае это слово используется в самом широком смысле: пользователь может быть представлен строкой, словарем или иным типом данных.

Кампания в мире соцсетей — это рекламное мероприятие, которое медиаагентство проводит от имени клиента на разных платформах. И помните: мы специально подготовим данные так, чтобы они имели неидеальный вид, — это даст возможность попрактиковаться в их очистке и обработке. Сначала создадим объект `Faker`, с помощью которого будем генерировать фиктивные данные:

```
#2
fake = faker.Faker()
```

Объект ведет учет уже созданных значений, чтобы гарантировать уникальность. С помощью генератора списка мы создадим 1000 уникальных имен пользователей (`usernames`).

Далее сформируем список пользователей. Мы сгенерируем 1000 словарей, по одному на каждого пользователя. В каждом словаре будут, в частности, следующие поля: `username`, `name`, `gender` и `email`. Каждый словарь затем будет преобразован в строку в формате JSON и добавлен в список. Конечно, эта структура далека от оптимальной, но в этом и суть — мы имитируем ситуацию, когда данные приходят к нам в сыром виде и их еще предстоит обработать:

```
#3
def get_users(no_of_users):
    usernames = (
        fake.unique.user_name() for i in range(no_of_users)
    )
    genders = random.choices(
        ["M", "F"], weights=[0.43, 0.57], k=no_of_users
    )
    for username, gender in zip(usernames, genders):
        name = get_random_name(gender)
        user = {
            "username": username,
            "name": name,
            "gender": gender,
            "email": fake.email(),
            "age": fake.random_int(min=18, max=90),
            "address": fake.address(),
        }
        yield json.dumps(user)

def get_random_name(gender):
    match gender:
        case "F":
            name = fake.name_female()
        case "M":
            name = fake.name_male()
    return name

users = get_users(1000)
users[:3]
```

Функция-генератор `get_users()` принимает на вход количество пользователей, которых нужно создать. Мы используем `fake.unique.user_name()` в генераторном выражении, чтобы получить уникальные имена пользователей. Свойство `fake.unique` отслеживает уже сгенерированные значения и выдает только те, которые не повторяются. Далее с помощью `random.choices()` мы создаем список из `no_of_users` случайных значений из набора `["M", "F"]` — это будут обозначения пола: Male (мужчина) и Female (женщина). Весовые коэффициенты `0.43` и `0.57` обеспечивают такое распределение: примерно 43 % пользователей будут мужчинами, а 57 % — женщинами. Мы используем `zip()`, чтобы пройтись сразу по именам пользователей и полу. Для каждого пользователя вызываем функцию `get_random_name()`, которая с помощью оператора `match` генерирует имя, соответствующее полу, создаем фиктивный адрес электронной почты, генерируем возраст и адрес, затем сериализуем все это в строку JSON и возвращаем через `yield`.

Обратите внимание на последнюю строку в ячейке. Как вы уже знаете, в Jupyter Notebook все, что написано в этой строке, выводится автоматически. Поэтому ячейка № 3 возвращает список первых трех пользователей:

```
['{"username": "epennington", "name": "Stephanie Gonzalez", ...}',
'{"username": "joshua61", "name": "Diana Richards", ...}',
'{"username": "dmoore", "name": "Erin Rose", "gender": "F",...}']
```

Если вы выполняете примеры в своем блокноте — отлично! Но помните: все данные генерируются псевдослучайно, поэтому ваши результаты будут отличаться. И каждый раз при запуске блокнота они будут другими. Кроме того, чтобы текст поместился на страницу, мы обрезали значительную часть вывода в этой главе — в реальном блокноте вы увидите гораздо больше данных.

В своей повседневной работе аналитики часто используют таблицы и создают разные схемы кодирования, чтобы уместить максимум информации прямо в названия кампаний. Формат, который мы выбрали, — упрощенный пример такой схемы. Каждое имя кампании будет содержать код ее типа, дату начала и окончания, целевую аудиторию по возрасту и полу (`M`, `F` или `A` — для сотрудников любого пола), а также валюту.

Все значения разделены знаком подчеркивания. Код, который генерирует такие имена кампаний, находится в ячейке № 4:

```
#4
# формат имени кампании:
# InternalType_StartDate_EndDate_TargetAge_TargetGender_Currency

from datetime import date, timedelta
import random

def get_type():
    # просто несколько случайных примеров кодов
    types = ["AKX", "BYU", "GRZ", "KTR"]
    return random.choice(types)

def get_start_end_dates():
    duration = random.randint(1, 2 * 365)
    offset = random.randint(-365, 365)
    start = date.today() - timedelta(days=offset)
    end = start + timedelta(days=duration)

    def _format_date(date_):
        return date_.strftime("%Y%m%d")
    return _format_date(start), _format_date(end)

def get_age_range():
    age = random.randrange(20, 46, 5)
    diff = random.randrange(5, 26, 5)
    return f"{age}-{age + diff}"

def get_gender():
    return random.choice(("M", "F", "A"))

def get_currency():
    return random.choice(("GBP", "EUR", "USD"))

def get_campaign_name():
    separator = "_"
    type_ = get_type()
    start, end = get_start_end_dates()
    age_range = get_age_range()
```

```
    gender = get_gender()
    currency = get_currency()
    return separator.join((type_, start, end, age_range, gender, currency))
```

В функции `get_type()` мы вызываем метод `random.choice()`, чтобы случайным образом выбрать один тип кампании из набора возможных значений. А вот `get_start_end_dates()` — более интересна. Сначала мы генерируем два случайных целых числа: продолжительность кампании в днях (от 1 дня до 2 лет) и смещение — количество дней от сегодняшнего дня (в диапазоне от –365 до +365). Мы вычитаем `offset` (в виде `timedelta`) из текущей даты, чтобы получить дату начала, и прибавляем `duration`, чтобы получить дату окончания. В результате возвращаются строки с этими датами.

Функция `get_age_range()` создает случайный возрастной диапазон для целевой аудитории — так, чтобы обе границы были кратны 5. Для этого мы используем функцию `random.randrange()` — она работает по тому же принципу, что и `range` с параметрами `start`, `stop`, `step` (эти параметры имеют то же значение, что и для объекта диапазона, с которым мы впервые столкнулись в главе 3). Мы случайным образом выбираем два числа: `age` (кратное 5 число от 20 до 45 включительно) и `diff` (также кратное 5 число от 5 до 25). Добавляем `diff` к `age`, чтобы получить верхнюю границу диапазона, и возвращаем строку в формате `"age-age+diff"`.

Остальные функции просто комбинируют `random.choice()` для разных параметров. А последняя, `get_campaign_name()`, собирает все воедино — и возвращает название кампании.

В ячейке № 5 мы пишем функцию, которая создает полноценный объект кампании:

```
#5
# данные кампании:
# название, бюджет, потрачено, клики, показы
def get_campaign_data():
    name = get_campaign_name()
    budget = random.randint(10**3, 10**6)
    spent = random.randint(10**2, budget)
    clicks = int(random.triangular(10**2, 10**5, 0.2 * 10**5))
    impressions = int(random.gauss(0.5 * 10**6, 2))
    return {
        "cmp_name": name,
        "cmp_bgt": budget,
        "cmp_spent": spent,
        "cmp_clicks": clicks,
        "cmp_impr": impressions,
    }
```

В этом фрагменте мы использовали несколько функций из модуля `random`. Функция `random.randint()` возвращает целое число в заданном диапазоне. Она работает по равномерному распределению вероятностей, то есть все числа в интервале имеют одинаковый шанс быть выбранными.

Чтобы данные не выглядели слишком однообразно, для значений переменных `clicks` и `impressions` мы использовали функции `triangular()` и `gauss()`. Они

работают на основе других распределений — так результат получится более естественным и разнообразным.

И прежде, чем двигаться дальше, убедимся, что одинаково понимаем названия используемых переменных: `clicks` — количество кликов по рекламному объявлению кампании; `budget` — общий бюджет, выделенный на кампанию; `spent` — сколько из этого бюджета уже израсходовано; `impressions` — количество показов кампании (вне зависимости от того, были клики или нет).

Обычно показов больше, чем кликов, так как пользователь может увидеть рекламу, но не щелкнуть на ней.

Теперь, имея все необходимые данные, собираем их в общую структуру:

```
#6
def get_data(users):
    data = []
    for user in users:
        campaigns = [
            get_campaign_data()
            for _ in range(random.randint(2, 8))
        ]
        data.append({"user": user, "campaigns": campaigns})
    return data
```

Как видите, каждый элемент в `data` — это словарь, содержащий один объект `user` и список `campaigns`, связанные с этим пользователем.

Очистка данных

Теперь можно переходить к очистке данных:

```
#7
rough_data = get_data(users)
rough_data[:2]  # взглянем на первые два элемента
```

Мы имитируем процесс загрузки данных из внешнего источника, а затем изучаем их содержимое. Jupyter Notebook идеально подходит для таких задач — можно шаг за шагом анализировать, что происходит.

Вы можете выбирать ту степень детализации, которая вам удобна. Вот как выглядит первый элемент в списке `rough_data`:

```
{'user': '{"username": "epennington", "name": ...}',
 'campaigns': [{'cmp_name': 'KTR_20250404_20250916_35-50_A_EUR',
 'cmp_bgt': 964496,
 'cmp_spent': 29586,
 'cmp_clicks': 36632,
 'cmp_impr': 500001},
{'cmp_name': 'AKX_20240130_20241017_20-25_M_GBP',
 'cmp_bgt': 344739,
 'cmp_spent': 166010,
 'cmp_clicks': 67325,
 'cmp_impr': 499999}]}
```

Теперь можно поработать с самими данными. Первое, что нужно сделать, чтобы продолжить анализ, — это денормализовать структуру. *Денормализация* — процесс, при котором данные приводятся к одной плоской таблице. Обычно это означает объединение информации из разных таблиц или сведение вложенных структур. Да, при этом возможны повторы одних и тех же данных, но зато упрощается анализ, поскольку не нужно разбираться с вложенными уровнями и не нужно «связывать» таблицы между собой вручную. В нашем случае денормализация означает, что мы получим список словарей-кампаний, в каждый из которых добавлен связанный с ним словарь-пользователь. Да, один и тот же пользователь будет дублироваться в каждой кампании, с которой он связан:

```
#8
data = []
for datum in rough_data:
    for campaign in datum["campaigns"]:
        campaign.update({"user": datum["user"]})
        data.append(campaign)
data[:2]  # взглянем на первые два элемента
```

Теперь первый элемент в `data` выглядит вот так:

```
{'cmp_name': 'KTR_20250404_20250916_35-50_A_EUR',
 'cmp_bgt': 964496,
 'cmp_spent': 29586,
 'cmp_clicks': 36632,
 'cmp_impr': 500001,
 'user': '{"username": "epennington", ...}'},
```

Чтобы вторая часть главы была определенной и воспроизводимой, мы сохраним сгенерированные данные. Так мы сможем загрузить этот файл в следующем блокноте и продолжить работу с теми же самыми результатами:

```
#9
with open("data.json", "w") as stream:
    stream.write(json.dumps(data))
```

Итак, в папке с исходным кодом главы должен появиться файл `data.json`. На этом работа в блокноте `ch13-dataprep` завершена. Можно закрыть его и открыть следующий: `ch13`.

Создание DataFrame

Теперь, когда мы подготовили данные, можно переходить к их анализу. Начнем еще с одной порции импортов:

```
#1
import json

import arrow
import pandas as pd
from pandas import DataFrame
```

С модулем `json` вы уже сталкивались в главе 8. А `arrow` мы коротко упоминали в главе 2 — это сторонняя библиотека, которая упрощает работу с датами и временем. Центральным элементом текущей главы (и проекта) будет библиотека `pandas` (название происходит от эконометрического термина «панельные данные» — *panel data*). Она предоставляет мощную структуру данных — `DataFrame` — таблицу, похожую на матрицу, с широкими возможностями обработки. Обычно `pandas` импортируют как `pd` (`import pandas as pd`), а сам класс `DataFrame` импортируют дополнительно.

После этого мы загружаем данные из файла `data.json` в `DataFrame` с помощью функции `pandas.read_json()`:

```
#2
df = pd.read_json("data.json")
df.head()
```

Теперь можно взглянуть на первые пять строк таблицы с помощью метода `head()` объекта `DataFrame`. Вы увидите примерно такую таблицу (рис. 13.2).

	cmp_name	cmp_bgt	cmp_spent	cmp_clicks	cmp_impr	user
0	KTR_20250404_20250916_35-50_A_EUR	964496	29586	36632	500001	{"username": "epennington", "name": "Stephanie...
1	AKX_20240130_20241017_20-25_M_GBP	344739	166010	67325	499999	{"username": "epennington", "name": "Stephanie...
2	BYU_20230828_20250115_25-45_M_GBP	177403	125738	29989	499997	{"username": "joshua61", "name": "Diana Richar...
3	AKX_20250216_20261129_45-60_F_USD	618256	75017	76301	500000	{"username": "joshua61", "name": "Diana Richar...
4	AKX_20231229_20250721_20-40_F_GBP	113805	12583	48915	500001	{"username": "joshua61", "name": "Diana Richar...

Рис. 13.2. Первые строки объекта DataFrame

В Jupyter результат вызова `df.head()` автоматически отображается в виде HTML-таблицы. Если вам нужен текстовый вывод, то можете просто обернуть `df.head()` в `print()`.

Структура `DataFrame` дает множество возможностей: можно фильтровать строки и столбцы, агрегировать данные и многое другое. К тому же можно выполнять операции сразу над целыми столбцами или строками и при этом не терять в производительности, как это случилось бы при работе с данными на чистом Python. Это достигается благодаря тому, что в `pandas` используется библиотека `NumPy`, которая, в свою очередь, получает высокую скорость за счет низкоуровневой реализации ядра.

NumPy расшифровывается как Numeric Python. Это одна из самых востребованных библиотек в мире анализа данных.

Работа с `DataFrame` позволяет объединить вычислительную мощь `NumPy` с возможностями, напоминающими работу с таблицами. По сути, мы получаем тот же подход, который аналитик применяет в Excel, но делаем это с помощью кода.

Теперь рассмотрим два способа быстро получить общее представление о данных:

```
#3
df.count()
```

Метод `count()` возвращает количество непустых ячеек в каждом столбце. Это помогает понять, насколько полно представлены данные. В нашем случае пропусков нет и вывод выглядит так.

`cmp_name`	5065
`cmp_bgt`	5065
`cmp_spent`	5065
`cmp_clicks`	5065
`cmp_impr`	5065
`user`	5065
`dtype: int64`	

У нас получилось 5065 строк. Учитывая, что в наборе данных 1000 пользователей и у каждого от двух до восьми кампаний, такое количество строк вполне логично и соответствует ожиданиям.

Строка dtype: int64 в конце означает, что значения, возвращенные df.count(), — это объекты NumPy типа int64. Здесь dtype означает data type (тип данных), а int64 — целое 64-битное число.

NumPy, по большому счету, реализована на языке C и вместо встроенных типов Python использует собственные типы, максимально приближенные к типам C. Благодаря этому числовые операции в NumPy выполняются гораздо быстрее, чем на чистом Python.

Метод `describe()` позволяет быстро получить статистическую сводку по данным:

```
#4
df.describe()
```

Как видно из приведенного ниже вывода, метод `describe()` предоставляет целый ряд статистических характеристик: количество значений (`count`), среднее значение (`mean`), среднеквадратическое отклонение (`std`), минимальное и максимальное

значения (min, max), а также распределение данных по квартилям. Благодаря этому методу мы уже получаем общее представление о структуре данных.

	cmp_bgt	cmp_spent	cmp_clicks	cmp_impr
count	5065.000000	5065.000000	5065.000000	5065.000000
mean	502965.054097	253389.854689	40265.781639	499999.474630
std	290468.998656	222774.897138	21840.783154	2.023801
min	1764.000000	107.000000	899.000000	499992.000000
25 %	251171.000000	67071.000000	22575.000000	499998.000000
50 %	500694.000000	187743.000000	36746.000000	499999.000000
75 %	756850.000000	391790.000000	55817.000000	500001.000000
max	999565.000000	984705.000000	98379.000000	500007.000000

Чтобы отобразить кампании с наибольшим бюджетом, можно воспользоваться методами sort_values() и head():

```
#5
df.sort_values(by=["cmp_bgt"], ascending=False).head(3)
```

Вывод будет примерно таким (часть столбцов опущена для компактности).

	cmp_name	cmp_bgt	cmp_clicks	cmp_impr
3186	GRZ_20230914_20230929_40-60_A_EUR	999565	63869	499998
3168	KTR_20250315_20260507_25-40_M_USD	999487	21097	500000
3624	GRZ_20250227_20250617_30-45_F_USD	999482	3435	499998

Вызов метода tail() вместо head() выводит кампании с наименьшими бюджетами:

```
#6
df.sort_values(by=["cmp_bgt"], ascending=False).tail(3)
```

Далее мы перейдем к более сложным задачам.

Распаковка названия кампании

Первым делом мы хотим избавиться от столбца cmp_name, так как он содержит слишком много информации в одном поле. Нам нужно разбить его на части и поместить каждую из них в отдельный столбец. Для этого мы воспользуемся методом apply(), предоставляемым объектом Series.

Класс pandas.core.series.Series — это мощная обертка над массивом (можно думать о нем как о списке, только с расширенными возможностями). Из объекта

DataFrame можно извлечь Series, обратившись к нужному столбцу как к элементу словаря. После этого мы вызываем метод apply(), который позволяет применить функцию к каждому элементу Series. В результате получается новый объект Series с обработанными значениями. Функцию мы затем преобразуем в новый DataFrame, а потом объединяем с исходным DataFrame с помощью метода join().

Мы начнем с написания функции, которая будет разбивать строку с названием кампании на кортеж из следующих частей: тип кампании, дата ее начала, дата окончания, целевой возраст, целевой пол и валюта. Для преобразования строковых представлений дат в объекты даты мы используем функцию arrow.get():

```
#7
def unpack_campaign_name(name):
    # очень оптимистичный метод: предполагает,
    # что формат названий кампаний всегда корректный
    type_, start, end, age, gender, currency = name.split("_")
    start = arrow.get(start, "YYYYMMDD").date()
    end = arrow.get(end, "YYYYMMDD").date()
    return type_, start, end, age, gender, currency
```

Затем мы извлекаем столбец с названиями кампаний из df и применяем к каждому значению функцию unpack_campaign_name():

```
#8
campaign_data = df["cmp_name"].apply(unpack_campaign_name)
```

После этого создаем новый DataFrame из собранных данных, которые хранятся в campaign_data:

```
#9
campaign_cols = [
    "Type",
    "Start",
    "End",
    "Target Age",
    "Target Gender",
    "Currency",
]
campaign_df = DataFrame.from_records(
    campaign_data, columns=campaign_cols, index=df.index
)
campaign_df.head(3)
```

Первые три строки дают общее представление о данных.

	Type	Start	End	Target Age	Target Gender	Currency
0	KTR	2025-04-04	2025-09-16	35-50	A	EUR
1	AKX	2024-01-30	2024-10-17	20-25	M	GBP
2	BYU	2023-08-28	2025-01-15	25-45	M	GBP

Так гораздо удобнее. Теперь данные из столбцов легко обрабатывать по отдельности. Важно помнить: даты отображаются как строки, однако на самом деле в `DataFrame` хранятся настоящие date-объекты.

Наконец, мы можем объединить исходный `DataFrame` (`df`) и `campaign_df`. При объединении двух `DataFrame` необходимо, чтобы у них был одинаковый индекс, в противном случае `pandas` не сможет правильно сопоставить строки. Мы заранее позаботились об этом, явно указав индекс из `df` при создании `campaign_df`:

```
#10
df = df.join(campaign_df)
```

Проверим данные, чтобы убедиться, что все корректно:

```
#11
df[["cmp_name"] + campaign_cols].head(3)
```

Первые несколько столбцов вывода выглядят так.

	cmp_name	Type	Start	End
0	KTR_20250404_20250916_35-50_A_EUR	KTR	2025-04-04	2025-09-16
1	AKX_20240130_20241017_20-25_M_GBP	AKX	2024-01-30	2024-10-17
2	BYU_20230828_20250115_25-45_M_GBP	BYU	2023-08-28	2025-01-15

Итак, объединение сработало: названия кампаний и разобранные по столбцам данные совпадают.

Обратите внимание, что мы обратились к `DataFrame` с помощью квадратных скобок, передав в них список названий столбцов. Это вернуло новый `DataFrame`, содержащий указанные столбцы (в заданном порядке), к которому затем применен метод `head()`.

Распаковка данных о пользователях

Аналогично предыдущему шагу теперь обработаем данные каждого пользователя, представленные в формате JSON. Мы вызываем метод `apply()` для столбца `user`, передавая функцию `unpack_user_json()`, которая принимает JSON-объект и преобразует его в список отдельных полей. На основе полученных данных создаем новый `DataFrame` под названием `user_df`:

```
#12
def unpack_user_json(user):
    # эта функция тоже весьма оптимистична — предполагает, что у каждого
    # объекта пользователя есть все необходимые атрибуты
    user = json.loads(user.strip())
    return [
        user["username"],
        user["email"],
        user["name"],
```

```
        user["gender"],
        user["age"],
        user["address"],
    ]

user_data = df["user"].apply(unpack_user_json)
user_cols = [
    "username",
    "email",
    "name",
    "gender",
    "age",
    "address",
]

user_df = DataFrame.from_records(
    user_data, columns=user_cols, index=df.index
)
```

Затем объединяем `user_df` с исходным `DataFrame` `df` (аналогично объединению с `campaign_df`):

```
#13
df = df.join(user_df)
df[["user"] + user_cols].head(2)
```

Проверяем результат. Вывод показывает, что объединение прошло успешно.

Переименование столбцов

Если вы введете `df.columns` в ячейке, то увидите, что названия некоторых столбцов по-прежнему неудачны. Исправим это с помощью метода `rename()`:

```
#14
new_column_names = {
    "cmp_bgt": "Budget",
    "cmp_spent": "Spent",
    "cmp_clicks": "Clicks",
    "cmp_impr": "Impressions",
}
df.rename(columns=new_column_names, inplace=True)
```

Метод `rename()` позволяет переименовывать столбцы (или строки), принимая словарь, в котором указано соответствие между старым и новым именем. Все столбцы, не указанные в словаре, останутся неизменными.

Вычисление метрик

Следующий шаг — добавить в таблицу дополнительные столбцы. Для каждой кампании у нас есть количество кликов и показов, а также сумма потраченных средств. Этого достаточно, чтобы ввести три метрики: `CTR`, `CPC` и `CPI` — показатели, часто применяемые в цифровом маркетинге.

Они расшифровываются следующим образом. CTR (Click Through Rate) — отношение количества кликов к количеству показов. Показывает, насколько эффективно объявление привлекает внимание пользователей. Чем выше CTR, тем лучше. CPC (Cost Per Click) — стоимость одного клика. CPI (Cost Per Impression) — стоимость одного показа. Напишем функцию, которая рассчитывает эти показатели и добавляет их в `DataFrame`:

```
#15
def calculate_metrics(df):
    # отношение количества кликов к количеству показов
    df["CTR"] = df["Clicks"] / df["Impressions"]
    # стоимость одного клика
    df["CPC"] = df["Spent"] / df["Clicks"]
    # стоимость одного показа
    df["CPI"] = df["Spent"] / df["Impressions"]

calculate_metrics(df)
```

Каждая строка добавляет новый столбец, а `pandas` автоматически применяет операцию деления ко всем строкам таблицы. Несмотря на то что в коде всего три деления, на самом деле их выполняется по одному на каждую строку, то есть 5140×3 операций. `pandas` скрывает от нас эти вычисления, обрабатывая таблицу быстро и эффективно.

Функция `calculate_metrics()` принимает `DataFrame` и изменяет его напрямую — это называется изменением *по месту* (in-place). Аналогичным образом работает, например, метод `list.sort()`. Такая функция считается нечистой, поскольку вносит изменения в переданный ей изменяемый объект.

Чтобы посмотреть результат, можно отфильтровать нужные столбцы и вызвать метод `head()`:

```
#16
df[["Spent", "Clicks", "Impressions", "CTR", "CPC", "CPI"]].head(
    3
)
```

Мы видим, что вычисления выполнены корректно для каждой строки:

	Spent	Clicks	Impressions	CTR	CPC	CPI
0	29586	36632	500001	0.073264	0.807655	0.059172
1	166010	67325	499999	0.134650	2.465800	0.332021
2	125738	29989	499997	0.059978	4.192804	0.251478

Кроме того, можно вручную проверить точность вычислений в первой строке:

```
#17
clicks = df["Clicks"][0]
impressions = df["Impressions"][0]
spent = df["Spent"][0]
```

```
CTR = df["CTR"][0]
CPC = df["CPC"][0]
CPI = df["CPI"][0]

print("CTR:", CTR, clicks / impressions)
print("CPC:", CPC, spent / clicks)
print("CPI:", CPI, spent / impressions)
```

Вывод будет следующим:

```
CTR: 0.07326385347229306 0.07326385347229306
CPC: 0.8076545097182791 0.8076545097182791
CPI: 0.059171881656236686 0.059171881656236686
```

Значения совпадают — это подтверждает правильность наших расчетов. Конечно, в реальной работе такую проверку выполнять не нужно, но она помогает понять, как можно вручную получить доступ к значениям и произвести вычисления. Чтобы обратиться к столбцу, можно использовать квадратные скобки с именем столбца, как в словаре. Далее по аналогии со списками/кортежами можно обратиться к конкретной строке, используя ее индекс.

Наш `DataFrame` почти готов. Осталось добавить два заключительных столбца: продолжительность кампании, чтобы можно было сопоставлять такие значения, как бюджет и количество показов, с длительностью; день недели начала кампании, поскольку для некоторых кампаний день может иметь значение — например, рекламные акции, приуроченные к спортивным событиям, проходящим на выходных.

```
#18
def get_day_of_the_week(day):
    return day.strftime("%A")

def get_duration(row):
    return (row["End"] - row["Start"]).days

df["Day of Week"] = df["Start"].apply(get_day_of_the_week)
df["Duration"] = df.apply(get_duration, axis="columns")
```

Функция `get_day_of_the_week()` получает объект даты и форматирует его в строку с названием соответствующего дня недели. Функция `get_duration()` устроена интереснее. Она принимает на вход целую строку таблицы, а не отдельное значение. Внутри происходит вычитание даты начала кампании из даты окончания. При вычитании объектов типа `date` получается объект `timedelta`, который представляет собой промежуток времени. Из него мы берем значение свойства `.days`, чтобы получить продолжительность в днях.

Чтобы определить день недели начала кампании, мы применяем `get_day_of_the_week()` к столбцу `Start` (это объект `Series`). Это тот же прием, который мы использовали ранее для столбцов `user` и `cmp_name`. Затем применяем `get_duration()` ко всей таблице, указав `axis="columns"`, — это нужно, чтобы функция получала на вход не одно значение, а всю строку. На первый взгляд параметр может

показаться неочевидным, это стоит воспринимать как передачу всех столбцов в каждую итерацию вызова `get_duration()`.

Проверим результаты следующим образом:

```
#19
df[["Start", "End", "Duration", "Day of Week"]].head(3)
```

Результат выглядит так.

	Start	End	Duration	Day of Week
0	2025-04-04	2025-09-16	165	Friday
1	2024-01-30	2024-10-17	261	Tuesday
2	2023-08-28	2025-01-15	506	Monday

Теперь мы знаем, что между 4 апреля 2025 года и 16 сентября 2025 года проходит 165 дней, а 28 августа 2023 года — это понедельник.

Финальная очистка

Итак, все нужные данные собраны и пора провести финальную очистку. Напомним: у нас еще остались столбцы `cmp_name` и `user` — они уже не нужны, поэтому мы их удалим. Кроме того, имеет смысл переупорядочить столбцы в таблице так, чтобы они отражали актуальную структуру данных. Для этого мы просто отберем нужные столбцы в нужном порядке, передав их в `df[...]`. Результатом будет новый `DataFrame`, который можно снова присвоить имени `df`:

```
#19
final_columns = [
    "Type",
    "Start",
    "End",
    "Duration",
    "Day of Week",
    "Budget",
    "Currency",
    "Clicks",
    "Impressions",
    "Spent",
    "CTR",
    "CPC",
    "CPI",
    "Target Age",
    "Target Gender",
    "Username",
    "Email",
    "Name",
    "Gender",
    "Age",
]
df = df[final_columns]
```

При этом мы сгруппируем сначала информацию о кампании, затем метрики, а в конце — данные пользователя. Так структура таблицы станет более логичной и удобной для анализа.

Теперь наш `DataFrame` очищен и готов к изучению. Прежде чем перейти к построению графиков, стоит сохранить снимок текущего состояния таблицы. Это позволит быстро восстановить ее из файла, не выполняя заново все предыдущие шаги. Некоторым аналитикам может быть удобно работать с электронной таблицей, поэтому посмотрим, как сохранить `DataFrame` в файл.

Сохранение DataFrame в файл

`DataFrame` можно сохранить в нескольких форматах. Если ввести `df.to_` и нажать Tab, то появится список автодополнения со всеми доступными вариантами.

В этом примере мы сохраним таблицу в трех форматах. Сначала — в CSV:

```
#20
df.to_csv("df.csv")
```

Затем — в JSON:

```
#21
df.to_json("df.json")
```

И наконец, в виде таблицы Excel:

```
#22
df.to_excel("df.xlsx")
```

Метод to_excel() требует наличия установленного пакета openpyxl. Этот пакет уже добавлен в файл requirements.txt для данной главы, так что если вы использовали его для установки зависимостей, то все уже готово к работе.

Как видите, сохранить `DataFrame` в любом из распространенных форматов довольно просто. И хорошая новость: обратный процесс столь же удобен — чтобы загрузить таблицу обратно, достаточно использовать функции `pandas.read_csv()` или `pandas.read_excel()`.

Визуализация результатов

В этом подразделе мы займемся визуализацией данных. С точки зрения Data Science мы не будем пытаться провести глубокий анализ или делать выводы — это не имело бы смысла, поскольку данные полностью случайны. Тем не менее этот пример вполне подойдет, чтобы начать работать с графиками/диаграммами и освоить некоторые возможности.

Один важный урок, который мы усвоили на практике: внешний вид действительно имеет значение. То, как вы оформляете свои данные, сильно влияет на то, как

воспринимают вашу работу. Если вы хотите, чтобы к вам относились серьезно, то позаботьтесь о визуальном представлении ваших графиков и таблиц: старайтесь делать их аккуратными и наглядными.

Библиотека `pandas` использует для построения графиков инструмент `Matplotlib`. Мы не будем работать с ней напрямую, разве что настроим стиль графиков и сохраним их на диск. Узнать больше информации об этой универсальной библиотеке можно на сайте https://matplotlib.org.

Для начала настроим Jupyter Notebook так, чтобы графики `Matplotlib` отображались как интерактивные виджеты прямо в ячейках. Это делается с помощью следующей команды:

```
#23
%matplotlib widget
```

Так вы сможете перемещать и масштабировать изображение, а также сохранить снимок (в низком разрешении) на диск.

Если не использовать команду %matplotlib widget, то графики будут отображаться как статические изображения. Для работы в интерактивном режиме с виджетами необходимо, чтобы в виртуальном окружении был установлен пакет ipympl. Он входит в файл requirements.txt для этой главы, так что если вы использовали его для установки зависимостей, то все должно быть готово.

Затем мы настраиваем стиль отображения графиков:

```
#24
import matplotlib.pyplot as plt
plt.style.use(["classic", "ggplot"])
plt.rc("font", family="serif"})
plt.rc("savefig", dpi=300)
```

Мы используем интерфейс `matplotlib.pyplot`, чтобы задать стиль. Выбрана комбинация стилей `classic` и `ggplot`. Стиль применяется слева направо, так что `ggplot` переопределяет элементы, которые заданы и в `classic`, и в `ggplot`. Кроме того, мы задаем семейство шрифтов Serif для текста на графиках. Вызов `plt.rc("savefig", dpi=300)` настраивает функцию `savefig()` на сохранение изображений с высоким разрешением, пригодных для печати.

Прежде чем строить графики, снова вызовем метод `df.describe()` (ячейка № 26). Результаты должны выглядеть следующим образом (рис. 13.3).

Такой краткий результат хорошо представлять занятым руководителям, которые хотят получить лишь примерное представление в цифрах.

Еще раз подчеркнем: в наших кампаниях указаны разные валюты, так что приведенные значения не имеют практического смысла. Цель здесь — продемонстрировать возможности DataFrame, а не провести корректный анализ реальных данных.

	Duration	Budget	Clicks	Impressions	Spent	CTR	CPC	CPI	Age
count	5065.000000	5065.000000	5065.000000	5065.000000	5065.000000	5065.000000	5065.000000	5065.000000	5065.000000
mean	369.488253	502965.054097	40265.781639	499999.474630	253389.854689	0.080532	10.131895	0.506780	53.882330
std	209.852969	290468.998658	21840.783154	2.023801	222774.897138	0.043682	19.527218	0.445550	21.170077
min	1.000000	1764.000000	899.000000	499992.000000	107.000000	0.001798	0.001493	0.000214	18.000000
25%	192.000000	251171.000000	22575.000000	499998.000000	67071.000000	0.045150	1.756140	0.134143	35.000000
50%	371.000000	500694.000000	36746.000000	499999.000000	187743.000000	0.073493	5.207316	0.375486	54.000000
75%	554.000000	756850.000000	55817.000000	500001.000000	391790.000000	0.111634	11.630131	0.783572	72.000000
max	730.000000	999565.000000	98379.000000	500007.000000	984705.000000	0.196758	462.814233	1.969410	90.000000

Рис. 13.3. Некоторые статистические показатели для очищенных данных

Тем не менее график обычно воспринимается гораздо легче, чем таблица чисел: его проще читать и он мгновенно дает наглядную картину. Построим график по четырем показателям, которые есть для каждой кампании: `Budget` (бюджет), `Spent` (расходы), `Clicks` (клики) и `Impressions` (показы):

```
#27
df[["Budget", "Spent", "Clicks", "Impressions"]].hist(
    bins=16, figsize=(16, 6)
)
plt.savefig("Figure13.4.png")
```

Мы извлекаем эти четыре столбца (в результате получаем новый `DataFrame`, содержащий только их) и вызываем метод `hist()`, чтобы построить гистограммы. Мы задаем параметры: количество интервалов и размер изображения, а остальное делает библиотека автоматически. Результат будет выглядеть примерно так (рис. 13.4).

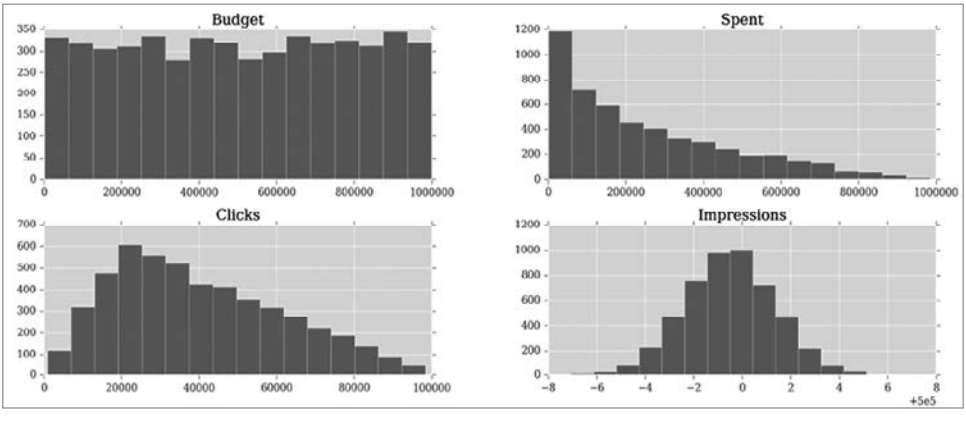


Рис. 13.4. Гистограммы по данным рекламных кампаний

Кроме того, мы вызываем `plt.savefig("Figure13.4.png")`, чтобы сохранить изображение в файл `Figure13.4.png`. При сохранении будет использоваться ранее заданное разрешение 300 dpi, обеспечивающее высокое качество изображения.

Обратите внимание: функция `plt.savefig()` сохраняет последнее изображение, созданное с помощью `Matplotlib`. Если вы вызовете ее в той же ячейке, в которой строится график, то гарантированно сохраните нужную картинку под нужным именем.

В текущем примере анализировать графики бессмысленно (данные случайные), но мы можем хотя бы убедиться, что форма графиков соответствует логике генерации данных.

- *Бюджет* выбирался случайным образом в заданном диапазоне, так что ожидаем равномерное распределение. На графике как раз это и видно.
- *Расходы* тоже распределены равномерно, но их верхний предел зависит от значения бюджета, которое различается. Поэтому можно ожидать убывающую кривую, напоминающую логарифмическую. Именно такую форму мы и наблюдаем.
- *Клики* генерировались по треугольному распределению, с пиком примерно на 20 % от максимального значения. На графике пик действительно находится в левой части, около 20 %.
- *Показы* распределены по нормальному закону — это распределение Гаусса, которое известно своей колоколообразной формой. Среднее значение — точно по центру, а стандартное отклонение равно 2. График подтверждает это.

Теперь построим графики по рассчитанным метрикам:

```
#28
df[["CTR", "CPC", "CPI"]].hist(bins=20, figsize=(16, 6))
plt.savefig("Figure13.5.png")
```

Результат показан на рис. 13.5.

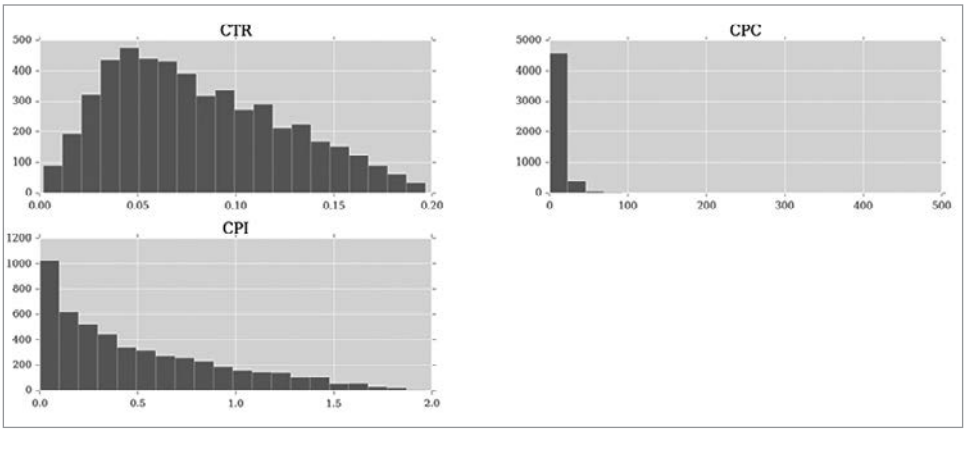


Рис. 13.5. Гистограммы по рассчитанным метрикам

На графике видно, что CPC (стоимость за клик) сильно смещена влево, то есть бóльшая часть значений — низкие. CPI (стоимость за показ) имеет похожее распределение, но выражено слабее.

Теперь предположим, что вы хотите проанализировать только определенный сегмент данных. Это можно сделать с помощью маски — логического условия, которое отфильтрует строки по заданному критерию. Процесс похож на применение условного оператора `if`, но ко всем строкам сразу:

```
#29
selector = df.Spent > df.Budget * 0.75
df[selector][["Budget", "Spent", "Clicks", "Impressions"]].hist(
    bins=15, figsize=(16, 6), color="green"
)
plt.savefig("Figure13.6.png")
```

В данном случае мы создаем маску `selector`, исключающую все строки, в которых потрачено менее 75 % бюджета. То есть мы отбираем лишь те кампании, в которых израсходовано не менее трех четвертей от бюджета. Обратите внимание: здесь показан альтернативный способ обращения к столбцам `DataFrame` — через точку (`object.имя_свойства`) вместо привычного использования квадратных скобок (`object['имя_свойства']`). Если имя столбца допустимо как имя переменной Python, то оба способа равнозначны.

Маска `selector` применяется так же, как если бы мы обращались к словарю по ключу. После применения маски `selector` к `df` мы получаем новый `DataFrame`. Затем выбираем нужные столбцы и снова вызываем `hist()`. В этот раз задаем параметр `color="green"`, просто чтобы показать, как можно менять цвет графика (рис. 13.6).

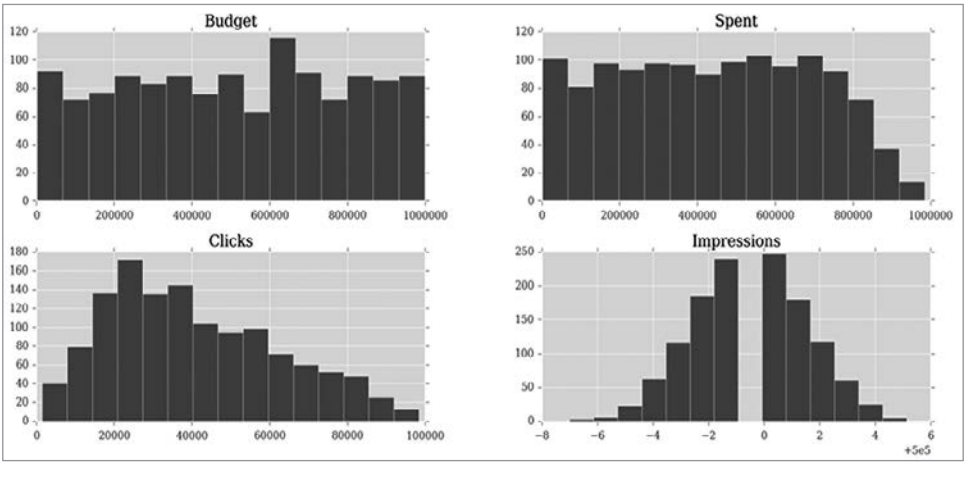


Рис. 13.6. Гистограммы по кампаниям, где потрачено не менее 75 % бюджета

Обратите внимание: форма большинства графиков практически не изменилась, за исключением графика *Spent* — он заметно отличается. Причина в том, что мы отобрали только строки, в которых сумма расходов составляет как минимум 75 % от бюджета. То есть мы рассматриваем лишь те кампании, в которых потрачены почти все суммы. А поскольку сами значения бюджета были выбраны из равномерного распределения, то график *Spent* теперь начинает принимать такую же форму. Если ужесточить границу — например, задать 85 %, — то график *Spent* будет становиться все более похожим на график *Budget*.

Теперь посмотрим на другой тип графика. Мы построим сумму значений `Spent`, `Clicks` и `Impressions` для каждого дня недели:

```
#30
df_weekday = df.groupby(["Day of Week"]).sum(numeric_only=True)
df_weekday[["Impressions", "Spent", "Clicks"]].plot(
    figsize=(16, 6), subplots=True
)
plt.savefig("Figure13.7.png")
```

Первая строка создает новый `DataFrame` `df_weekday`, сгруппировав данные по столбцу `Day of Week`. Затем вызывается агрегатная функция `sum()`, чтобы посчитать сумму по каждой группе. Параметр `numeric_only=True` необходим, чтобы избежать ошибок при попытке просуммировать нечисловые столбцы. Альтернативный вариант — выбрать только нужные столбцы (`Day of Week`, `Impressions`, `Spent`, `Clicks`) еще до группировки и суммирования.

Обратите внимание, что теперь мы вызываем метод `plot()` вместо `hist()`. Он строит линейный график. Параметр `subplots=True` заставляет `plot()` нарисовать отдельный график для каждого столбца (рис. 13.7).

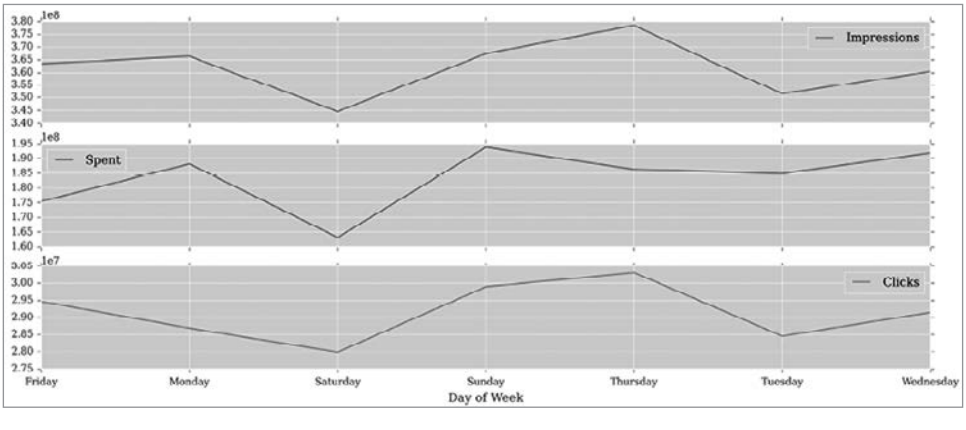


Рис. 13.7. Графики, агрегированные по дням недели

На этих графиках видно, что кампании, начавшиеся в четверг, получили наибольшее количество кликов и показов, а кампании, начавшиеся в субботу, потратили

меньше всего. Если бы мы имели дело с реальными данными, то такая информация могла бы быть важной и ее стоило бы предоставить клиентам.

Следует отметить, что дни недели отсортированы в алфавитном порядке, из-за чего графики не слишком удобно читать. В качестве упражнения вы можете поискать способ исправить это.

Завершим подраздел, посвященный визуализации, еще одной задачей. Мы хотим провести простую агрегацию: сгруппировать данные по целевому полу (`Target Gender`) и целевому возрасту (`Target Age`), а затем посчитать среднее значение и стандартное отклонение (`std`) для показателей `Impressions` и `Spent` внутри каждой группы:

```
#31
agg_config = {
    "Impressions": ["mean", "std"],
    "Spent": ["mean", "std"],
}
df.groupby(["Target Gender", "Target Age"]).agg(agg_config)
```

Мы подготовим словарь с конфигурацией, в котором укажем, какие агрегаты (среднее значение и стандартное отклонение) нас интересуют для каждого показателя. Затем выполним группировку по столбцам `Target Gender` и `Target Age`, передав этот словарь в метод `agg()`.

Результат выглядит так.

		Impressions		**Spent**	
		mean	**std**	**mean**	**std**
Target Gender	**Target Age**				
A	20–25	499999.245614	2.189918	217330.771930	204518.652595
	20–30	499999.465517	2.210148	252261.637931	228932.088945
	20–35	499998.564103	1.774006	218726.410256	215060.976707
	20–40	499999.459016	1.971241	255598.213115	222697.755231
	20–45	499999.574074	2.245346	216527.666667	190345.252888
...	...	...	...	...	...
M	45–50	499999.480769	2.128153	276112.557692	226975.008137
	45–55	499999.306122	2.053494	267137.938776	239249.474145
	45–60	499999.500000	1.984063	236623.312500	223464.578371
	45–65	499999.679245	1.503503	215634.528302	223308.046968
	45–70	499998.870370	1.822773	310267.944444	242353.980346

Это очень полезное агрегирование, поскольку дает возможность понять, как различаются показатели по полу и возрасту целевой аудитории. Теперь добавим еще один инструмент: *сводную таблицу*. Это мощный способ агрегировать данные по категориям и представлять их в табличной форме, удобной для проведения анализа. Рассмотрим простой пример такой таблицы:

```
#31
df.pivot_table(
    values=["Impressions", "Clicks", "Spent"],
    index=["Target Age"],
    columns=["Target Gender"],
    aggfunc="sum"
)
```

Мы создаем сводную таблицу, чтобы увидеть, как связаны возраст целевой аудитории (`Target Age`) и три ключевых показателя: `Impressions`, `Clicks` и `Spent`. В таблице эти показатели разделяются по полу целевой аудитории (`Target Gender`). Параметр `aggfunc` определяет функцию агрегации, которая будет применяться к каждой группе. Можно указать имя функции, объект функции, список функций или словарь, где каждому столбцу соответствует функция. В нашем случае используется `"sum"`, то есть мы смотрим сумму показателей по каждой группе. Если бы мы не указали `aggfunc`, то по умолчанию считалось бы среднее значение. Результат — новый объект `DataFrame` со сводной таблицей (рис. 13.8).

	Clicks			Impressions			Spent		
Target Gender	**A**	**F**	**M**	**A**	**F**	**M**	**A**	**F**	**M**
Target Age									
20-25	2460345	2355790	2954169	28499957	29999964	34999963	12387854	16271204	14605751
20-30	2254458	1742729	2133740	28999969	21999963	25999959	14631175	11435369	13184984
20-35	1323341	1735301	2926626	19499944	22499974	34999942	8530330	10987452	18383305
20-40	2304325	1987013	1975200	30499967	28499973	26999950	15591491	15490069	13806347
20-45	2402785	1667405	1790037	26999977	22499993	21999968	11692494	9064229	9623006

Рис. 13.8. Сводная таблица с показателями Impressions, Clicks и Spent по полу и возрасту целевой аудитории

На этом наш проект по анализу данных завершен. Вы познакомились с полным циклом: от загрузки и очистки данных до визуализации и создания сводных таблиц. Мы подчеркиваем важность освоения среды Jupyter Notebook — она гораздо лучше обычной консоли, практична и удобна в использовании, а также дает возможность создавать слайды, отчеты и интерактивные презентации. Удачи в исследовании мира Python, Jupyter и Data Science!

Что дальше

Data Science действительно увлекательная область. Как говорилось в начале главы, тем, кто хочет основательно изучить эту область, потребуется получить прочные базовые знания математики и статистики. Работа с некорректно интерполированными данными делает любые выводы бесполезными. То же касается данных, полученных неправильным способом экстраполяции или при неверной частоте выборки.

Вот простой пример: представьте, что у нас есть очередь людей и их пол чередуется — женщина, мужчина, женщина, мужчина и т. д. (F-M-F-M-F-M-F...).

Если мы возьмем в выборку только элементы с четными индексами, то окажется, что в очереди одни мужчины. Если с нечетными — что только женщины.

Это, конечно, утрированный случай, но он показывает, насколько легко ошибиться, особенно при работе с большими объемами данных, где выборка неизбежна. А значит, качество анализа напрямую зависит от качества самой выборки.

Основные инструменты, с которых стоит начать работу с Data Science и Python, описаны ниже.

- NumPy (https://www.numpy.org) — базовый пакет для научных вычислений на Python. Включает в себя мощный объект многомерного массива, функции для широковещательных операций, средства интеграции с кодом на C/C++ и Fortran, линейную алгебру, преобразование Фурье, генерацию случайных чисел и многое другое.
- scikit-learn (https://scikit-learn.org) — одна из самых популярных библиотек машинного обучения для Python. Предлагает простые и эффективные инструменты для интеллектуального анализа данных и аналитики. Подходит для самых разных задач и может использоваться повторно в различных контекстах. Построена на базе NumPy, SciPy и Matplotlib.
- pandas (https://pandas.pydata.org) — библиотека с открытым исходным кодом и лицензией BSD. Предоставляет высокопроизводительные и удобные в использовании структуры данных и инструменты анализа. Мы активно пользовались ею в этой главе.
- IPython (https://ipython.org) и Jupyter (https://jupyter.org) — архитектура для интерактивных вычислений. Jupyter позволяет работать с кодом, текстом, графиками и уравнениями в одном интерфейсе, в частности имеющем вид блокнотов.
- Matplotlib (https://matplotlib.org) — библиотека для построения двумерных графиков. Позволяет создавать графики типографского качества в самых разных форматах: как для печати, так и для интерактивной работы. Работает в скриптах Python, оболочках IPython, блокнотах Jupyter, веб-приложениях и графических интерфейсах.

- Seaborn (https://seaborn.pydata.org) — библиотека визуализации данных на базе Matplotlib. Предлагает удобный высокоуровневый интерфейс для построения красивых и информативных статистических графиков.
- Numba (https://numba.pydata.org) — позволяет ускорить выполнение программ благодаря высокопроизводительным функциям, написанным непосредственно на Python. С помощью нескольких аннотаций можно динамически компилировать численно нагруженный и ориентированный на массивы код Python в машинные инструкции (JIT-компиляция), получая производительность, сравнимую с C, C++ или Fortran, и при этом не переходя на другой язык или интерпретатор.
- Bokeh (https://bokeh.pydata.org) — интерактивная библиотека визуализации для Python, ориентированная на современные браузеры. Ее цель — предложить элегантные и лаконичные средства построения оригинальных графиков в духе D3.js, но при этом поддерживать высокую интерактивность даже для больших или потоковых наборов данных.

Помимо отдельных библиотек, существуют и целые экосистемы, такие как SciPy (https://scipy.org) и уже упомянутая Anaconda (https://anaconda.org), которые содержат множество пакетов и предоставляют рабочую среду по умолчанию.

Возможно, вам будет удобнее работать именно с этими экосистемами, поскольку установка всех вышеописанных инструментов и их зависимостей может быть затруднительной в некоторых системах.

Резюме

В этой главе мы говорили об анализе данных. Вместо того чтобы погружаться в теоретические основы этой широкой области, мы сосредоточились на практическом проекте. Вы познакомились с Jupyter Notebook и различными библиотеками, такими как pandas, Matplotlib и NumPy.

Разумеется, описать все эти темы в одной главе — задача не из простых, поэтому многие из них мы затронули лишь поверхностно. Надеемся, что проект, с которым мы поработали совместно, дал вам общее представление о том, как выглядит рабочий процесс в этой сфере.

Следующая глава посвящена разработке API.

14 Введение в разработку API

> Единственная цель сострадательного общения — облегчить страдания других людей.
>
> *Тик Нат Хан*

В этой главе мы поговорим о том, что такое *программный интерфейс приложения* (Application Programming Interface, API).

Мы кратко затронем протокол HTTP, поскольку именно с его помощью будем создавать наш API. Кроме того, выбранный нами фреймворк FastAPI активно использует аннотации типов, так что рекомендуем освежить в памяти материал главы 12, посвященный этой теме.

После короткого общего введения в API мы покажем вам проект, связанный с железнодорожной системой. Его завершенная версия добавлена в исходный код для этой главы — вместе с файлом зависимостей и README, в котором объясняется, как запустить API и обращаться к нему.

Для этого проекта FastAPI оказался идеальным вариантом. Благодаря его возможностям нам удалось создать API с чистым, лаконичным и выразительным кодом. Мы считаем его отличной отправной точкой, которую вы сможете исследовать и расширять.

Скорее всего, вам, как разработчику, рано или поздно придется иметь дело с API. Технологии и фреймворки быстро меняются, поэтому важно обратить внимание на теоретические основы, которые мы изложим в данной главе. Они помогут вам не зависеть от конкретных инструментов или библиотек.

Начнем с HTTP.

Протокол передачи гипертекста (HTTP)

Всемирная паутина (World Wide Web), или просто *Веб* — это способ доступа к информации через Интернет. Сама Всемирная сеть представляет собой огромную сетевую инфраструктуру, объединяющую миллиарды устройств по всему миру, чтобы они могли обмениваться данными. Информация передается через

Интернет с помощью множества языков общения — протоколов, которые позволяют самым разным устройствам делиться содержимым.

Web — это модель обмена информацией, построенная поверх Интернета и использующая в качестве основы *протокол передачи гипертекста* (Hypertext Transfer Protocol, HTTP). Таким образом, Web — всего лишь один из способов передачи данных по Интернету. Электронная почта, обмен мгновенными сообщениями, новостные группы и другие службы также используют собственные протоколы.

Как работает HTTP

HTTP — это асимметричный *клиент-серверный* протокол *запроса и ответа*. HTTP-клиент — например, ваш браузер — отправляет сообщение-запрос на HTTP-сервер. В ответ тот возвращает сообщение-ответ. В своей основе HTTP — *протокол с моделью запроса*: инициатором взаимодействия всегда выступает клиент, который запрашивает данные с сервера, а не наоборот. Существуют приемы, имитирующие поведение с отправкой данных от сервера, такие как long polling, WebSockets или HTTP/2 Server Push, однако основа HTTP неизменна: клиент делает запрос, сервер отвечает. На рис. 14.1 приведена схема этого процесса.

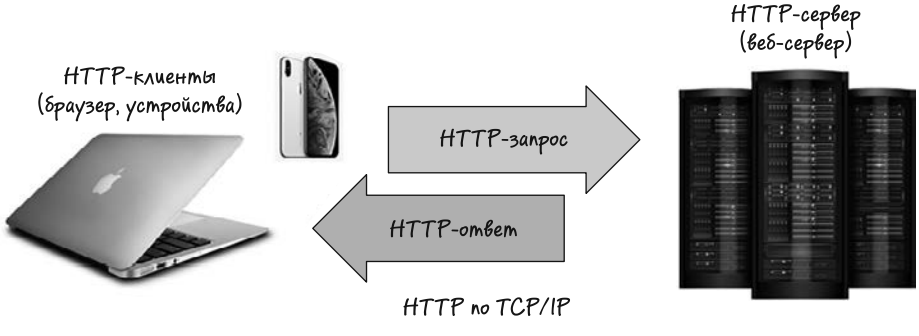


Рис. 14.1. Упрощенная схема работы протокола HTTP

HTTP передается с использованием протокола *TCP/IP* (*Transmission Control Protocol* — протокол управления передачей; *Internet Protocol* — протокол Интернета), который обеспечивает надежный обмен данными через Интернет.

Одна из ключевых особенностей HTTP — *отсутствие сохранения состояния* (stateless). Это означает, что текущий запрос не содержит информации о предыдущих. Такая техническая особенность существует не случайно, и ее вполне можно обойти. На практике большинство сайтов предлагают возможность входа в систему, создавая иллюзию непрерывного состояния между страницами. При входе на сайт пользователя его информация сохраняется (чаще всего на стороне клиента — в специальных файлах, называемых *cookie*). Благодаря этому каждый

новый запрос содержит данные, позволяющие серверу «узнать» пользователя, показать персонализированный интерфейс, отобразить его имя, сохранить содержимое корзины и т. д.

HTTP определяет набор методов (их также называют *глаголами*), которые указывают, какое действие должно быть выполнено над конкретным ресурсом. Каждый из методов имеет особенности, но некоторые из них обладают схожими свойствами. В API, с которым мы будем работать, используются следующие методы:

- `GET` — используется для запроса представления указанного ресурса. Запросы с этим методом должны только получать данные, не изменяя состояние сервера;
- `POST` — применяется для отправки сущности на указанный ресурс. Обычно приводит к изменению состояния или вызывает побочные эффекты на стороне сервера;
- `PUT` — запрашивает создание или обновление ресурса: состояние ресурса должно соответствовать представлению, переданному в теле запроса;
- `DELETE` — используется для удаления состояния указанного ресурса.

Существуют и другие методы, такие как `HEAD`, `CONNECT`, `OPTIONS`, `TRACE` и `PATCH`. Полное описание всех методов можно найти на сайте https://developer.mozilla.org/en-US/docs/Web/HTTP.

API, который мы напишем, будет работать по протоколу HTTP. Это значит, что нам предстоит писать код для отправки и обработки запросов и ответов. Далее мы будем опускать приставку HTTP перед словами «запрос» и «ответ», так как в этом контексте путаницы не возникнет.

Коды состояния

Один из ключевых аспектов HTTP-ответов — наличие *кода состояния*, который кратко сообщает о результате запроса. Код состоит из числа и краткого описания, например: `404 Not Found`. Полный список кодов состояния HTTP можно найти по адресу https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

Коды делятся на следующие категории:

- `1xx` — информационные ответы: запрос получен, обработка продолжается;
- `2xx` — успешные ответы: запрос успешно получен, понят и принят;
- `3xx` — перенаправление: для завершения запроса требуется дополнительное действие;
- `4xx` — ошибка на стороне клиента: в запросе синтаксическая ошибка или он не может быть выполнен;
- `5xx` — ошибка на стороне сервера: сервер не смог выполнить корректный запрос.

При работе с API мы будем получать коды состояния в каждом ответе, поэтому важно понимать, что они означают, — это помогает быстро определять, все ли прошло как надо или возникла ошибка.

Знакомство с API

Прежде чем углубляться в детали проекта этой главы, кратко обсудим, что такое API в целом.

Что такое API

Как уже упоминалось в начале главы, API расшифровывается как Application Programming Interface — *программный интерфейс приложения*. Это набор правил, протоколов и инструментов, предназначенных для разработки программного обеспечения и приложений. По сути, API служит связующим звеном между компьютерами или программами, тогда как пользовательский интерфейс соединяет компьютер с человеком.

Обычно API сопровождается *спецификацией (стандартом)* — своего рода чертежом, содержащим описание того, как должны взаимодействовать компоненты программного обеспечения. Система, соответствующая спецификации, считается реализующей или предоставляющей этот API. Термин API может обозначать и саму реализацию, и ее описание.

Проще говоря, API определяет методы и форматы данных, которые разработчики могут использовать для взаимодействия с программой, веб-службой или другим программным компонентом.

В широком смысле различают два типа API.

- *Веб-API* — работают через Интернет и позволяют веб-приложениям взаимодействовать между собой или с серверной частью. Такие API — основа современной веб-разработки: они позволяют, например, получать данные с сервера, отправлять информацию или подключаться к сторонним сервисам, таким как платежные шлюзы или медиаплатформы.
- *Фреймворки* и *библиотеки* — предоставляют готовые функции и процедуры для выполнения конкретных задач и ускоряют разработку приложений, выступая как строительные блоки.

В этой главе мы сосредоточимся на веб-API.

Обычно API состоит из нескольких частей. Они могут называться по-разному: *методы*, *подпрограммы*, но чаще всего в контексте веб-API используется выражение *«конечные точки»* (чаще говорят просто «эндпоинты» — от endpoints),

и именно его мы будем использовать в данной главе. Когда вы обращаетесь к этим частям интерфейса, это называется *вызовом* конечной точки.

В спецификации API объясняется, как вызывать каждую конечную точку, какие типы запросов использовать, какие параметры и заголовки нужно передать, по каким адресам обращаться и т. д.

Зачем нужны API

Существует несколько причин, по которым в системе вводится API. Одна из них уже была упомянута — обеспечение взаимодействия между разными приложениями.

Есть и другая важная цель — предоставление доступа к системе через специальный слой, через который внешний мир может взаимодействовать с данной системой.

Этот API-слой берет на себя задачи безопасности, такие как *аутентификация и авторизация* пользователей, а также валидация данных, которые передаются в ходе взаимодействия.

Аутентификация — процесс подтверждения личности пользователя по его учетным данным. Авторизация — проверка того, какие действия или ресурсы доступны конкретному пользователю.

Таким образом, пользователи, системы и данные проходят проверку на границе и только после этого получают доступ к остальной части системы — через API.

Этот процесс можно сравнить с прохождением паспортного контроля в аэропорту: чтобы попасть в страну (то есть в систему), необходимо предъявить документы и пройти проверку.

Дополнительное преимущество API заключается в том, что он скрывает внутреннюю реализацию системы от внешнего мира. Если в системе меняется внутренняя структура, язык программирования или логика работы, то API можно адаптировать, сохранив при этом единый стабильный интерфейс для внешнего использования. Представьте, что опускаете письмо в почтовый ящик. Вам не нужно знать, как именно работает почтовая служба, — главное, чтобы письмо дошло до адресата. Интерфейс — «почтовый ящик» — остается неизменным, тогда как все, что «за кулисами» (инструменты, технологии, рабочие процессы и т. д.), может меняться.

Неудивительно, что практически любое современное устройство, подключенное к Интернету, постоянно взаимодействует с множеством API, выполняя свои функции.

Протоколы API

Существует несколько разновидностей API. Они могут быть открытыми для всех или ограниченными (частными), предоставлять доступ к данным, к сервисам или и к тому и другому сразу. API могут быть написаны и спроектированы с использованием разных подходов и стандартов, а также работать на различных протоколах.

- *HTTP/HTTPS* (*Hypertext Transfer Protocol/Secure Hypertext Transfer Protocol*, протокол передачи гипертекста/протокол защищенной передачи гипертекста) — основа веб-коммуникации и передачи данных в Интернете.
- *REST* (*Representational State Transfer*, передача состояния представления) — строго говоря, не протокол, а архитектурный стиль, построенный поверх HTTP. API, созданные в этом стиле, называются RESTful. Они не сохраняют состояние (не хранят информацию о предыдущих запросах) и позволяют эффективно кэшировать данные.
- *SOAP* (*Simple Object Access Protocol*, простой протокол доступа к объектам) — зарекомендовавший себя как надежный протокол для построения веб-сервисов. Сообщения формируются в формате XML, а спецификация довольно строгая, что позволяет использовать SOAP в системах с повышенными требованиями к безопасности и надежности транзакций.
- *GraphQL* — язык запросов к API, использующий систему типов для определения структуры данных. В отличие от REST GraphQL использует одну общую конечную точку и позволяет клиентам запрашивать только нужные данные.
- *WebSocket* — подходит для приложений, где требуется двусторонняя связь и обновление данных в реальном времени. Обеспечивает полнодуплексное соединение через один TCP-канал.
- *RPC* (*Remote Procedure Call*, удаленный вызов процедуры) — позволяет выполнять код на сервере, удаленно вызывая процедуры (по имени). Такие API обычно тесно связаны с реализацией на серверной стороне и редко публикуются в открытом доступе.

Форматы обмена данными в API

Ранее мы говорили, что API служит интерфейсом между как минимум двумя компьютерными системами. Было бы крайне неудобно, если бы при их интеграции приходилось адаптировать данные под произвольный формат каждой из сторон. Поэтому API, выступающий как слой связи между системами, определяет не только используемые протоколы, но и форматы, в которых происходит обмен данными.

Наиболее распространенные форматы сегодня — *JSON*, *XML* и *YAML*. С JSON вы познакомились в главе 8, посвященной файлам и хранению данных. Именно JSON будет использоваться и в API, над которым мы будем работать в этой главе. Этот формат стал широко использоваться в API благодаря своей простоте, чи-

табельности и универсальности. Более того, многие фреймворки поддерживают автоматическое преобразование данных в JSON и обратно по умолчанию, без дополнительной настройки.

API для железнодорожной системы

Теперь, когда у вас есть общее представление о том, что такое API, перейдем к практическому примеру.

Прежде чем мы покажем код, хотим уточнить: он не предназначен для использования в условиях производственной эксплуатации. Мы сознательно не стали усложнять главу избыточными деталями. Тем не менее код полностью рабочий и может служить хорошей отправной точкой для дальнейшего изучения темы API. В конце главы мы оставим рекомендации, как с ним можно поэкспериментировать.

У нас есть база данных с несколькими сущностями, описывающими железнодорожную систему. Мы хотим предоставить внешним приложениям возможность выполнять операции *CRUD* с этой базой и для этого напишем API, который станет интерфейсом к данным.

CRUD — это аббревиатура, означающая четыре базовые операции с данными: создание (Create), чтение (Read), обновление (Update) и удаление (Delete). Большинство HTTP-сервисов реализуют именно такой подход, особенно в REST-ориентированных или REST-подобных API.

Начнем с обзора файлов проекта, чтобы вы представляли его структуру. Вы найдете их в папке этой главы в составе исходного кода:

```
$ tree -a api_code
api_code
├── .env.example
├── api
│   ├── __init__.py
│   ├── admin.py
│   ├── config.py
│   ├── crud.py
│   ├── database.py
│   ├── deps.py
│   ├── models.py
│   ├── schemas.py
│   ├── stations.py
│   ├── tickets.py
│   ├── trains.py
│   ├── users.py
│   └── util.py
├── dummy_data.py
├── main.py
├── queries.md
└── train.db
```

В папке `api_code` находятся все файлы, относящиеся к проекту на FastAPI. Основной модуль приложения — `main.py`. Вдобавок в проекте оставлен вспомогательный сценарий `dummy_data.py`, с помощью которого можно сгенерировать новую базу данных `train.db`. Перед его запуском стоит прочитать файл `README.md`, расположенный в папке этой главы: в нем описано, как использовать сценарий. Кроме того, в файле `queries.md` собраны примеры запросов к API, которые можно скопировать и опробовать.

Пакет `api` содержит модули самого приложения: `models.py` — модели базы данных; `schemas.py` — схемы, описывающие структуру данных, передаваемых через API; `users.py`, `stations.py`, `tickets.py`, `trains.py`, `admin.py` — модули с определениями соответствующих конечных точек API; `util.py` — вспомогательные функции; `deps.py` — определение поставщиков зависимостей; `config.py` — настройки конфигурации; `crud.py` — функции для выполнения операций CRUD; `.env.example` — шаблон для создания собственного файла `.env` с настройками приложения.

В инженерии программного обеспечения внедрение зависимостей (dependency injection, DI) — это паттерн проектирования, при котором объект получает другие объекты, от которых он зависит. Эти объекты называются зависимостями. Компонент, отвечающий за создание и предоставление таких зависимостей, называется инжектором или поставщиком зависимостей. Благодаря этому подходу остальные части системы могут использовать нужные зависимости, не заботясь об их создании, настройке или удалении. Дополнительную информацию о паттерне внедрения зависимостей можно найти в Википедии по адресу https://en.wikipedia.org/wiki/Dependency_injection.

Проектирование базы данных

При проектировании схемы *«сущность — связь»* (entity-relationship schema, ER-схема) для этого проекта мы постарались создать структуру, которая будет одновременно интересной, но при этом простой и компактной. В приложении рассматриваются четыре сущности: `Station` (станция), `Train` (поезд), `Ticket` (билет) и `User` (пользователь). `Train` представляет собой поездку от одной станции до другой. `Ticket` связывает `Train` с `User`. Пользователи могут быть как пассажирами, так и администраторами, в зависимости от того, какие действия им разрешено выполнять через API.

На рис. 14.2 показана ER-модель базы данных. В ней описаны четыре сущности и связи между ними.

Мы определили модели базы данных с помощью SQLAlchemy, а в качестве СУБД выбрали SQLite — для простоты.

Если вы пропустили главу 8, то самое время прочесть ее — в ней вы найдете информацию, которая поможет понять модели, используемые в проекте текущей главы.

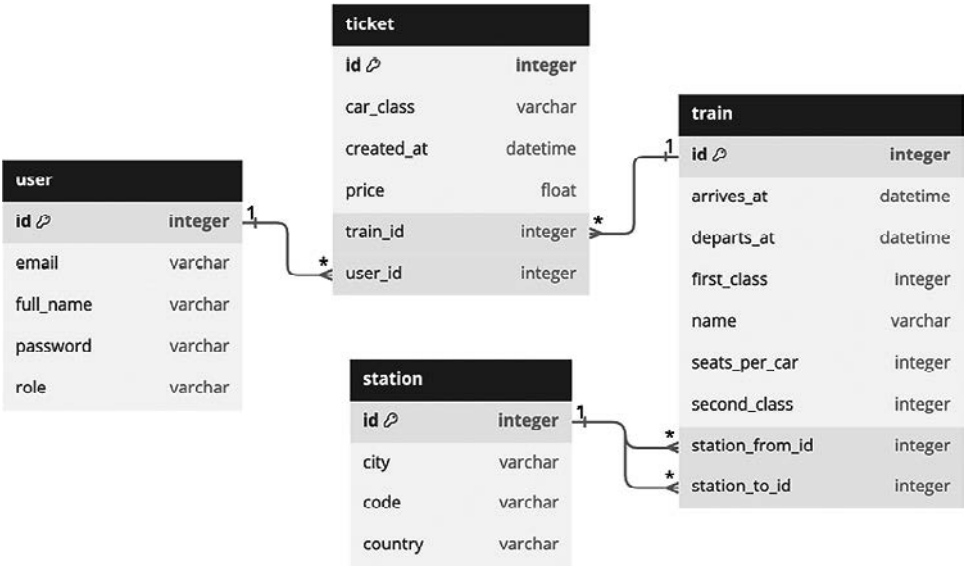


Рис. 14.2. ER-модель базы данных

Модуль `models` выглядит так:

```
# api_code/api/models.py
import hashlib
import os
import secrets
from enum import StrEnum, auto

from sqlalchemy import (
    DateTime,
    Enum,
    ForeignKey,
    Unicode,
)
from sqlalchemy.orm import mapped_column, relationship, Mapped

from .database import Base

UNICODE_LEN = 128
SALT_LEN = 64

# Перечисления
class Classes(StrEnum):
    first = auto()
    second = auto()

class Roles(StrEnum):
    admin = auto()
    passenger = auto()
```

Как обычно, в начале модуля импортируются все необходимые библиотеки. Затем определяются две переменные: `UNICODE_LEN` — стандартная длина для текстовых полей с поддержкой Юникода, и `SALT_LEN` — длина соли, используемой при хешировании паролей.

Информация о том, что такое соль, была представлена в главе 9.

Кроме того, в этом модуле определены два перечисления: `Classes` и `Roles`. Они используются при описании моделей. В качестве базового класса выбран `StrEnum`, появившийся в Python 3.11. Он удобен тем, что члены перечисления можно напрямую сравнивать со строками. Функция `auto()` автоматически задает значения для элементов перечисления. В случае с `StrEnum` это будет строка с именем элемента, приведенным к нижнему регистру.

Теперь рассмотрим определение модели `Station`:

```
# api_code/api/models.py
class Station(Base):
    __tablename__ = "station"

    id: Mapped[int] = mapped_column(primary_key=True)
    code: Mapped[str] = mapped_column(
        Unicode(UNICODE_LEN), unique=True
    )
    country: Mapped[str] = mapped_column(Unicode(UNICODE_LEN))
    city: Mapped[str] = mapped_column(Unicode(UNICODE_LEN))

    departures: Mapped[list["Train"]] = relationship(
        foreign_keys="[Train.station_from_id]",
        back_populates="station_from",
    )
    arrivals: Mapped[list["Train"]] = relationship(
        foreign_keys="[Train.station_to_id]",
        back_populates="station_to",
    )

    def __repr__(self):
        return f"<{self.code}: id={self.id} city={self.city}>"

    __str__ = __repr__
```

Модель `Station` достаточно проста. В ней определены несколько атрибутов: `id` выступает в роли первичного ключа, а `code`, `country` и `city` в совокупности позволяют однозначно идентифицировать станцию. Кроме того, заданы две связи: одна соединяет станцию со всеми отправляющимися с нее поездами, другая — с прибывающими. В завершение определен метод `__repr__()`, который возвращает строковое представление экземпляра станции. Эта же реализация назначена и методу `__str__()`, так что результат будет одинаковым при вызове

`str(station_instance)` и `repr(station_instance)`. Такой подход часто используют, чтобы избежать дублирования кода.

На поле `code` наложено ограничение уникальности — это исключает возможность существования двух станций с одинаковым кодом в базе данных. В крупных городах, таких как Рим, Лондон или Париж, может быть несколько станций, поэтому поля `city` и `country` могут совпадать. Однако каждая станция должна иметь уникальный код.

Далее идет определение модели `Train`:

```
# api_code/api/models.py
class Train(Base):
    __tablename__ = "train"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(Unicode(UNICODE_LEN))

    station_from_id: Mapped[int] = mapped_column(
        ForeignKey("station.id")
    )
    station_from: Mapped["Station"] = relationship(
        foreign_keys=[station_from_id],
        back_populates="departures",
    )

    station_to_id: Mapped[int] = mapped_column(
        ForeignKey("station.id")
    )
    station_to: Mapped["Station"] = relationship(
        foreign_keys=[station_to_id],
        back_populates="arrivals",
    )

    departs_at: Mapped[DateTime] = mapped_column(
        DateTime(timezone=True)
    )
    arrives_at: Mapped[DateTime] = mapped_column(
        DateTime(timezone=True)
    )

    first_class: Mapped[int] = mapped_column(default=0)
    second_class: Mapped[int] = mapped_column(default=0)
    seats_per_car: Mapped[int] = mapped_column(default=0)

    tickets: Mapped[list["Ticket"]] = relationship(
        back_populates="train"
    )

    def __repr__(self):
        return f"<{self.name}: id={self.id}>"

    __str__ = __repr__
```

В модели `Train` перечислены все атрибуты, необходимые для описания экземпляра поезда, а также задана удобная связь `tickets`, которая позволяет получить все билеты, оформленные на этот поезд. Поля `first_class` и `second_class` хранят количество вагонов первого и второго классов соответственно.

Кроме того, добавлены связи `station_from` и `station_to`, которые указывают на станции отправления и прибытия. Благодаря этим связям можно получить доступ к объектам станций напрямую, а не только к их идентификаторам.

Далее следует модель `Ticket`:

```
# api_code/api/models.py
class Ticket(Base):
    __tablename__ = "ticket"

    id: Mapped[int] = mapped_column(primary_key=True)
    created_at: Mapped[DateTime] = mapped_column(
        DateTime(timezone=True)
    )

    user_id: Mapped[int] = mapped_column(ForeignKey("user.id"))
    user: Mapped["User"] = relationship(
        foreign_keys=[user_id],
        back_populates="tickets"
    )

    train_id: Mapped[int] = mapped_column(ForeignKey("train.id"))
    train: Mapped["Train"] = relationship(
        foreign_keys=[train_id],
        back_populates="tickets"
    )

    price: Mapped[float] = mapped_column(default=0)
    car_class: Mapped[Enum] = mapped_column(Enum(Classes))

    def __repr__(self):
        return f"<id={self.id} user={self.user} train={self.train}>"

    __str__ = __repr__
```

У билета тоже есть свои свойства, в том числе две связи: `user` и `train`. Первая указывает на пользователя, оформившего билет, вторая — на поезд, к которому относится билет.

Обратите внимание, что в определении поля `car_class` используется перечисление `Classes`. Это приводит к созданию соответствующего перечисляемого поля в схеме базы данных.

Наконец, модель `User`:

```
# api_code/api/models.py
class User(Base):
    __tablename__ = "user"
```

```
    pwd_separator = "#"

    id: Mapped[int] = mapped_column(primary_key=True)
    full_name: Mapped[str] = mapped_column(
        Unicode(UNICODE_LEN), nullable=False
    )
    email: Mapped[str] = mapped_column(
        Unicode(2 * UNICODE_LEN), unique=True
    )
    password: Mapped[str] = mapped_column(
        Unicode(2 * UNICODE_LEN)
    )
    role: Mapped[Enum] = mapped_column(Enum(Roles))

    tickets: Mapped[list["Ticket"]] = relationship(
        back_populates="user"
    )

    def is_valid_password(self, password: str):
        """Проверяет, совпадает ли переданный пароль с тем, который хранится
           в базе данных."""
        salt, stored_hash = self.password.split(self.pwd_separator)
        _, computed_hash = _hash(
            password=password, salt=bytes.fromhex(salt)
        )
        return secrets.compare_digest(stored_hash, computed_hash)

    @classmethod
    def hash_password(cls, password: str, salt: bytes = None):
        salt, hashed = _hash(password=password, salt=salt)
        return f"{salt}{cls.pwd_separator}{hashed}"

    def __repr__(self):
        return (
            f"<{self.full_name}: id={self.id} "
            f"role={self.role.name}>"
        )

    __str__ = __repr__
```

Модель `User` определяет свойства каждого пользователя. Обратите внимание: для роли пользователя снова используется перечисление. Пользователь может быть либо пассажиром, либо администратором. В следующем разделе мы покажем простой пример конечной точки, доступ к которой разрешен лишь авторизованным пользователям.

В модели определены и два метода, связанные с паролями: один выполняет хеширование, другой — проверку. Напомним: в главе 9 мы обсуждали, что хранить пароли в базе данных в открытом виде (как есть) недопустимо. В нашем API при сохранении пароля мы создаем его хеш и сохраняем его вместе с солью, использованной при хешировании. В исходном коде вы найдете реализацию функции `_hash()` в конце данного модуля (мы не добавили ее в текущую главу, чтобы не увеличивать объем книги).

Основная настройка и конфигурация

Теперь, разобравшись с моделями базы данных, взглянем на главный модуль приложения:

```
# api_code/main.py
from api import admin, config, stations, tickets, trains, users
from fastapi import FastAPI

settings = config.Settings()

app = FastAPI()
app.include_router(admin.router)
app.include_router(stations.router)
app.include_router(trains.router)
app.include_router(users.router)
app.include_router(tickets.router)

@app.get("/")
def root():
    return {
        "message": (
            f"Welcome to version {settings.api_version} "
            f"of our API"
        )
    }
```

Весь код содержится в файле `main.py`. Он импортирует модули с конечными точками и подключает их маршрутизаторы к основному приложению. Подключая маршрутизатор, мы позволяем приложению обрабатывать все конечные точки, описанные с его помощью. О том, что такое маршрутизаторы, мы расскажем чуть позже в этой главе.

В основном модуле определена только одна конечная точка — приветственное сообщение. Это обычная функция — в данном случае `root()` — с кодом, который выполняется при обращении к этой точке. Когда и как будет вызвана функция, зависит от примененных к ней декораторов. В данном случае используется декоратор `app.get()`, который указывает API, что эту точку нужно обрабатывать при запросах типа `GET`. Аргумент декоратора определяет URL, по которому будет доступна конечная точка. Здесь это `"/"`, то есть корневой путь — базовый адрес, по которому запущено приложение.

Если бы API работал по адресу http://localhost:8000, то данная конечная точка вызывалась бы при обращении к http://localhost:8000 или http://localhost:8000/ (обратите внимание на наличие или отсутствие косой черты в конце пути).

Параметры приложения

В приветственном сообщении, показанном в предыдущем фрагменте кода, используется переменная `api_version`, полученная из объекта настроек `settings`. Все современные фреймворки позволяют передавать в приложение набор параметров, которые определяют его поведение. Этот проект не требовал обязательного

использования настроек (нужные значения можно было бы просто прописать напрямую в главном модуле), однако мы считаем полезным показать, как устроен этот механизм:

```
# api_code/api/config.py
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    model_config = SettingsConfigDict(env_file=".env")

    secret_key: str
    debug: bool
    api_version: str
```

Настройки описываются с помощью модели Pydantic (https://github.com/pydantic/pydantic). `Pydantic` — это библиотека, предоставляющая валидацию данных на основе аннотаций типов в Python. В более ранних версиях в Pydantic была доступна поддержка конфигурации через переменные окружения, но сейчас эта возможность выделена в отдельную библиотеку `pydantic-settings` (https://github.com/pydantic/pydantic-settings), которая дополнительно расширяет возможности управления настройками[1]. В нашем примере мы определили три параметра:

- `secret_key` — используется для подписи и проверки токенов JWT;
- `debug` — если установить значение `True`, то движок SQLAlchemy будет подробно логировать SQL-запросы. Это удобно для отладки;
- `api_version` — версия API. Мы никак ее не используем, кроме как отображаем в приветственном сообщении, но в реальных приложениях версия имеет значение, так как связана с определенной спецификацией API.

FastAPI считывает эти параметры из файла `.env`, как указано при создании объекта `SettingsConfigDict`. Данный файл выглядит следующим образом:

```
# api_code/.env
SECRET_KEY="018ea65f62337ed59567a794b19dcaf8"
DEBUG=false
API_VERSION=2.0.0
```

Для работы FastAPI требуется сторонняя библиотека python-dotenv. Она входит в список зависимостей этой главы, так что если вы уже установили их в виртуальное окружение, то дополнительные действия не потребуются.

Конечные точки Station

Здесь мы рассмотрим некоторые конечные точки FastAPI. Поскольку он построен на операциях CRUD, то в коде встречаются повторы. Поэтому мы приведем по одному примеру для каждой из операций CRUD, используя конечные точки

[1] Была вынесена в отдельную библиотеку начиная с Pydantic 2.0. — *Примеч. науч. ред.*

Station. Исходный код к книге содержит конечные точки и для других моделей — вы найдете там такие же шаблоны и соглашения. Основное различие в том, что они связаны с другими моделями базы данных.

Чтение данных

Начнем с операции чтения — с запроса GET. В нашем случае извлечем все станции из базы данных:

```
# api_code/api/stations.py
from typing import Optional

from fastapi import (
    APIRouter,
    Depends,
    HTTPException,
    Response,
    status,
)
from sqlalchemy.orm import Session

from . import crud
from .deps import get_db
from .schemas import Station, StationCreate, StationUpdate, Train

router = APIRouter(prefix="/stations")

@router.get("", tags=["Stations"])
def get_stations(
    db: Session = Depends(get_db), code: Optional[str] = None
) -> list[Station]:
    return crud.get_stations(db=db, code=code)
```

В модуле stations.py сначала импортируются необходимые объекты из модулей typing и fastapi. Кроме того, импортируется Session из SQLAlchemy и несколько других компонентов из локального проекта.

Конечная точка get_stations() обернута в декоратор router, а не app, как в основном файле. Объект APIRouter можно рассматривать как мини-класс FastAPI: он принимает те же аргументы. Мы объявляем router и присваиваем ему префикс (в нашем случае "/stations"); это означает, что все функции, помеченные этим декоратором, становятся конечными точками по адресам, начинающимся с http://localhost:8000/stations. В данном случае пустая строка, переданная методу router.get(), дает приложению указание разместить эту конечную точку по корневому URL маршрутизатора — он формируется как сочетание базового адреса и префикса маршрутизатора, как описано выше.

FastAPI предоставляет несколько способов указания типа данных, возвращаемых конечной точкой. Один из них — передать аргумент response_model

в декоратор. Однако в нашем случае достаточно указать возвращаемое значение функции с помощью аннотации типа. Для этой конечной точки возвращается список объектов `Station`. Их реализация будет рассмотрена позже.

Аргумент `tags` используется для генерации документации.

Сама функция принимает несколько аргументов: сессию базы данных `db` и необязательную строку `code`. Если передано значение `code`, то конечная точка возвращает только те станции, у которых поле `code` совпадает с указанным значением.

На какие нюансы стоит обратить внимание?

- Данные, передаваемые в запросе (например, параметры запроса), указываются непосредственно в объявлении конечной точки. Если функция требует данные из тела запроса, то используется модель Pydantic. В этом проекте такие модели определены в модуле `schemas.py`.
- Возвращаемое значение функции становится телом ответа. FastAPI попытается сериализовать эти данные в формат JSON. Но если указан аргумент `response_model`, то сериализация сначала проходит через соответствующую модель Pydantic, а затем преобразуется в JSON.
- Сессия базы данных используется в теле функции конечной точки с помощью механизма зависимостей. В этом случае применяется класс `Depends`, которому передается функция `get_db()`. Эта функция предоставляет локальную сессию базы данных и закрывает ее после завершения вызова конечной точки.
- С помощью класса `Optional` из модуля `typing` указывается, что параметр запроса является необязательным.

Тело функции `get_stations()` просто вызывает одноименную функцию из модуля `crud` и возвращает результат. Все функции, отвечающие за взаимодействие с базой данных, размещены в модуле `crud.py`.

Такое решение принято намеренно — оно облегчает повторное использование кода и написание тестов. Кроме того, упрощается чтение кода самой конечной точки.

Теперь рассмотрим тело функции `get_stations()` из модуля `crud.py`:

```
# api_code/api/crud.py
from datetime import UTC, datetime
from sqlalchemy import delete, select, update
from sqlalchemy.orm import Session, aliased
from . import models, schemas

def get_stations(db: Session, code: str | None = None):
    stm = select(models.Station)
    if code is not None:
        stm = stm.where(models.Station.code.ilike(code))
    return db.scalars(stm).all()
```

Функция `get_stations()` очень похожа по сигнатуре на вызывающую ее конечную точку. Она выбирает и возвращает все записи модели `Station`, при необходимости отфильтрованные по значению `code` (если оно указано и не равно `None`).

Чтобы запустить API, активируйте виртуальное окружение и в папке `api_code` выполните следующую команду:

```
$ uvicorn main:app --reload
```

Uvicorn — это сверхбыстрый *ASGI-сервер*, построенный на базе `uvloop` и `httptools`. Он отлично работает как с обычными, так и с асинхронными функциями.

На странице документации ASGI (https://asgi.readthedocs.io) сказано:

> «*ASGI (Asynchronous Server Gateway Interface)* — идейный преемник WSGI *(Web Server Gateway Interface)*, призванный стать стандартом взаимодействия между асинхронными Python-серверами, фреймворками и приложениями.
>
> Если WSGI задавал стандарт для синхронных приложений на Python, то ASGI поддерживает как асинхронные, так и синхронные приложения, в том числе совместимость с WSGI и множество серверов и фреймворков».

В этом проекте мы выбрали синхронный стиль кодинга, поскольку асинхронная версия лишь усложнила бы его восприятие.

Если вы уверенно работаете с асинхронным кодом, то обратитесь к документации FastAPI (https://fastapi.tiangolo.com), чтобы узнать, как реализовать асинхронные конечные точки.

Флаг --reload в команде запуска uvicorn, приведенной выше, активирует автоматическую перезагрузку сервера при сохранении любого файла. Он необязателен, но значительно ускоряет разработку API, особенно в процессе внесения правок в исходный код.

Если вызвать конечную точку `get_stations()`, то результат будет таким:

```
$ http http://localhost:8000/stations
HTTP/1.1 200 OK
content-length: 702
content-type: application/json
date: Thu, 04 Apr 2024 09:46:29 GMT
server: uvicorn

[
    {
        "city":    "Rome",
        "code":    "ROM",
        "country": "Italy",
        "id":      0
    },
```

```
    {
        "city":    "Paris",
        "code":    "PAR",
        "country": "France",
        "id":      1
    },
    ... некоторые станции пропущены ...
    {
        "city":    "Sofia",
        "code":    "SFA",
        "country": "Bulgaria",
        "id":      11
    }
]
```

Обратите внимание на команду, с помощью которой мы вызываем API: `http`. Это команда, входящая в состав утилиты `Httpie`.

Сайт проекта Httpie — https://httpie.io. Это удобный клиент HTTP для командной строки, ориентированный на работу с API. Он поддерживает формат JSON, подсветку синтаксиса, постоянные сессии, загрузки в стиле wget, подключаемые расширения и многое другое. Существуют и другие инструменты для выполнения HTTP-запросов, например curl. Выбор зависит от предпочтений — с точки зрения API нет разницы, чем именно вы вызываете конечную точку из командной строки.

По умолчанию API доступен по адресу http://localhost:8000. При необходимости адрес можно изменить, передав дополнительные аргументы команде `uvicorn`.

Первые строки ответа содержат техническую информацию от API: используется протокол HTTP/1.1, запрос выполнен успешно (код состояния `200 OK`), указаны длина содержимого и его тип — JSON. Кроме того, есть временная метка и тип сервера. Далее мы будем опускать ту часть информации, которая просто повторяется.

Основное тело ответа — это список объектов `Station` в формате JSON. Такой результат достигается благодаря аннотации `list[Station]` в сигнатуре функции.

Если вы хотите выполнить поиск по коду, например по станции London, то можете использовать следующую команду:

```
$ http http://localhost:8000/stations?code=LDN
```

В ней используется тот же URL, что и ранее, но добавляется параметр запроса `code` (отделен от пути символом `?`). Результат будет следующим:

```
$ http http://localhost:8000/stations?code=LDN
HTTP/1.1 200 OK
...
[
```

```
    {
        "city": "London",
        "code": "LDN",
        "country": "UK",
        "id": 2
    }
]
```

Мы получили одну запись — станцию `London`, и она возвращается в составе списка, как и предполагается аннотацией типа для этой конечной точки (`list[Station]`).

Теперь перейдем к конечной точке, предназначенной для получения одной станции по ID:

```
# api_code/api/stations.py
@router.get("/{station_id}", tags=["Stations"])
def get_station(
    station_id: int, db: Session = Depends(get_db)
) -> Station:
    db_station = crud.get_station(db=db, station_id=station_id)
    if db_station is None:
        raise HTTPException(
            status_code=404,
            detail=f"Station {station_id} not found.",
        )
    return db_station
```

В этой конечной точке маршрутизатор настроен на обработку запросов `GET` по адресу http://localhost:8000/stations/{station_id}, где `station_id` — целое число. Надеемся, теперь структура URL становится более понятной. У нас есть базовая часть `http://localhost:8000`, затем префикс маршрутизатора `/stations` и, наконец, часть, передаваемая конкретной конечной точке, — в данном случае `/{station_id}`.

Теперь запросим станцию `Moscow`, у которой `ID = 3`:

```
$ http http://localhost:8000/stations/3
HTTP/1.1 200 OK
...
{
    "city": "Moscow",
    "code": "MSK",
    "country": "Russia",
    "id": 3
}
```

На сей раз мы получили объект напрямую, а не внутри списка — в отличие от предыдущей конечной точки `get_stations()`. Это соответствует аннотации типа

для данной точки, где указан `Station`. И это логично, ведь мы запрашиваем конкретный объект по идентификатору.

Функция `get_station()` принимает два аргумента: `station_id` (с аннотацией типа `int`) и объект сессии базы данных `db`. Благодаря аннотациям типов FastAPI выполняет проверку аргументов при вызове конечной точки.

Если передать значение, не являющееся целым числом, в качестве `station_id`, то произойдет следующее:

```
$ http http://localhost:8000/stations/moscow
HTTP/1.1 422 Unprocessable Entity
...
{
    "detail": [
        {
            "input": "moscow",
            "loc": [
                "path",
                "station_id"
            ],
            "msg": "Input should be a valid integer, …",
            "type": "int_parsing",
            "url": "https://errors.pydantic.dev/2.6/v/int_parsing"
        }
    ]
}
```

Сообщение об ошибке мы сократили, так как оно слишком длинное. FastAPI вернул полезную информацию: значение `station_id`, извлеченное из URL, не является допустимым целым числом. Кроме того, обратите внимание: вместо кода состояния `200 OK` мы получили `422 Unprocessable Entity`. В целом коды *4xx* указывают на ошибки на стороне клиента, а коды *5xx* — на ошибки сервера. В данном случае мы использовали некорректный URL (указали нецелое число), так что это клиентская ошибка. Другие фреймворки API в такой ситуации чаще всего вернули бы `400 Bad Request`, но FastAPI возвращает `422 Unprocessable Entity` — довольно специфичный код. Тем не менее в FastAPI легко настроить, какой код должен возвращаться при ошибке запроса. В официальной документации приведены примеры того, как это сделать.

Теперь посмотрим, что произойдет, если указать `station_id`, которого не существует:

```
$ http http://localhost:8000/stations/100
HTTP/1.1 404 Not Found
...
{
    "detail": "Station 100 not found."
}
```

На этот раз URL корректен — `station_id` действительно является целым числом. Однако станции с ID 100 не существует. В данном случае API возвращает статус `404 Not Found`, о чем сообщается в теле ответа.

Если вернуться к коду этой конечной точки, то можно увидеть, насколько прост ее механизм. При условии, что переданные аргументы корректны (соответствуют аннотациям типов), функция пытается получить станцию из базы данных, используя функцию из модуля `crud`. Если станция не найдена — вызывается `HTTPException` со статусом `404` и сообщением, которое должно помочь понять причину ошибки. Если станция найдена, то возвращается. Преобразование объекта в JSON-формат происходит автоматически. Объект, полученный из базы данных, — это экземпляр SQLAlchemy-класса `Station` (из модуля `models`). Он передается в класс `Station` из Pydantic (из модуля `schemas`), который используется для создания JSON-представления, возвращаемого конечной точкой.

На первый взгляд все это может показаться сложным, однако на самом деле это отличный пример разделения ответственности. FastAPI берет на себя всю организацию обработки, а разработчику нужно только настроить взаимосвязи: параметры запроса, модели ответа, зависимости и т. д.

Создание данных

Теперь рассмотрим нечто более интересное — как создать новую станцию. Начнем с конечной точки:

```
# api_code/api/stations.py
@router.post(
    "",
    status_code=status.HTTP_201_CREATED,
    tags=["Stations"],
)
def create_station(
    station: StationCreate, db: Session = Depends(get_db)
) -> Station:
    db_station = crud.get_station_by_code(
        db=db, code=station.code
    )
    if db_station:
        raise HTTPException(
            status_code=400,
            detail=f"Station {station.code} already exists.",
        )
    return crud.create_station(db=db, station=station)
```

В этот раз мы указываем маршрутизатору, что хотим принимать запросы `POST` на корневой адрес (напомним: базовая часть адреса + префикс маршрутизатора). Возвращаемое значение аннотировано как `Station`, поскольку конечная точка будет возвращать только что созданный объект. Кроме того, указан код состояния по умолчанию — `201 Created`.

Функция `create_station()` принимает обычную сессию базы данных `db` и объект `station`. Он создается автоматически: FastAPI извлекает данные из тела запроса и передает их в схему Pydantic `StationCreate`. Она определяет, какие именно данные нужно получить от клиента, и на ее основе формируется объект `station`.

Логика внутри функции следующая: сначала пробуем найти станцию по переданному коду. Если такая станция уже существует, то создать еще одну с тем же кодом нельзя — поле `code` должно быть уникальным. В этом случае возвращается статус `400 Bad Request` с пояснением, что такая станция уже существует. Если станция с указанным кодом не найдена, то можно создать новую и вернуть ее в ответе. Теперь посмотрим на определение схем Pydantic, которые участвуют в данном процессе:

```
# api_code/api/schemas.py
from pydantic import BaseModel, ConfigDict

class StationBase(BaseModel):
    code: str
    country: str
    city: str

class Station(StationBase):
    model_config = ConfigDict(from_attributes=True)
    id: int

class StationCreate(StationBase):
    pass
```

Обратите внимание, как в определении схем используется наследование. Это обычная практика — создавать базовую схему, которая содержит общие поля и функциональность, а затем в дочерних схемах описывать конкретные особенности. В данном случае базовая схема содержит поля `code`, `country` и `city`. Когда мы получаем данные о станциях, нам также нужен `id`, поэтому он добавляется в класс `Station`. Кроме того, данный класс используется для преобразования объектов SQLAlchemy, и об этом необходимо явно сообщить. Для этого указывается атрибут `model_config`. SQLAlchemy — это система *объектно-реляционного отображения* (object-relational mapping, ORM), поэтому необходимо указать модели, что нужно считывать атрибуты объекта, установив `from_attributes=True`.

Модель `StationCreate` не требует дополнительных полей, поэтому ее тело можно оставить пустым, использовав команду `pass`.

Теперь рассмотрим функции из модуля `crud`, которые связаны с этой конечной точкой:

```
# api_code/api/crud.py
def get_station_by_code(db: Session, code: str):
    return db.scalar(
        select(models.Station).where(
            models.Station.code.ilike(code)
        )
    )
```

```
def create_station(
    db: Session,
    station: schemas.StationCreate,
):

    db_station = models.Station(**station.model_dump())
    db.add(db_station)
    db.commit()
    return db_station
```

Функция `get_station_by_code()` довольно проста. Она выполняет выборку объекта `Station`, сравнивая значение поля `code` без учета регистра (префикс `i` в `ilike()` означает сравнение, не зависящее от регистра).

Существуют и другие способы выполнения сравнения, не зависящего от регистра, в которых не используется ilike. Они могут быть эффективными с точки зрения производительности, но для целей этой главы нам подошел именно такой простой вариант.

Функция `create_station()` принимает сессию базы данных `db` и экземпляр `StationCreate`. Сначала мы преобразуем объект `station` в словарь Python, вызвав метод `model_dump()`. Мы уверены, что все необходимые данные имеются — иначе бы валидация схемы Pydantic уже прервала выполнение запроса.

Используя данные из `station.model_dump()`, мы создаем экземпляр модели SQLAlchemy `Station`, добавляем его в базу данных, фиксируем изменения (вызов `commit()`) и возвращаем объект. Обратите внимание: при создании объекта `db_station` поле `id` еще отсутствует. Оно автоматически присваивается движком базы данных при вставке строки в таблицу `stations`, то есть в момент вызова `db.commit()`. После фиксации транзакции SQLAlchemy автоматически заполняет атрибут `id`.

Теперь посмотрим, как эта конечная точка работает на практике. Обратите внимание, что при вызове команды `http` нужно явно указать метод `POST`, чтобы можно было передать данные в теле запроса в формате JSON. Предыдущие запросы относились к виду `GET` — именно этот метод используется по умолчанию в `http`. Из-за ограничений на длину строки в книге команда разбита на две строки:

```
$ http POST http://localhost:8000/stations \
code=TMP country=Temporary-Country city=tmp-city
HTTP/1.1 201 Created
...
{
    "city": "tmp-city",
    "code": "TMP",
    "country": "Temporary-Country",
    "id": 12
}
```

Станция успешно создана. Теперь повторим попытку, но на этот раз не укажем обязательное поле `code`:

```
$ http POST http://localhost:8000/stations \
country=Another-Country city=another-city
HTTP/1.1 422 Unprocessable Entity
...
{
    "detail": [
        {
            "input": {
                "city": "another-city",
                "country": "Another-Country"
            },
            "loc": [
                "body",
                "code"
            ],
            "msg": "Field required",
            "type": "missing",
            "url": "https://errors.pydantic.dev/2.6/v/missing"
        }
    ]
}
```

Как и ожидалось, мы снова получили статус `422 Unprocessable Entity`. Это связано с тем, что не прошла валидация по схеме Pydantic `StationCreate`: в теле запроса отсутствует поле `code`. В теле ответа содержится пояснение и полезная ссылка, которая позволяет уточнить суть ошибки.

Обновление данных

Логика обновления станции чуть более сложная. Начнем с конечной точки:

```
# api_code/api/stations.py
@router.put("/{station_id}", tags=["Stations"])
def update_station(
    station_id: int,
    station: StationUpdate,
    db: Session = Depends(get_db),
):
    db_station = crud.get_station(db=db, station_id=station_id)

    if db_station is None:
        raise HTTPException(
            status_code=404,
            detail=f"Station {station_id} not found.",
        )
    else:
        crud.update_station(
            db=db, station=station, station_id=station_id
        )
        return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Маршрутизатор настроен на прием запросов PUT — именно такой метод обычно используется для изменения ресурсов. URL заканчивается параметром station_id, который указывает, какую станцию нужно обновить. Функция принимает station_id, экземпляр схемы Pydantic StationUpdate и стандартную сессию базы данных db.

Сначала мы пробуем получить станцию из базы. Если она не найдена, то возвращается статус 404 Not Found — обновлять нечего. В случае же успеха мы выполняем обновление и возвращаем статус 204 No Content, что считается стандартным способом ответа на запрос PUT. Мы могли бы вернуть 200 OK, но в этом случае нужно было бы добавить обновленный объект в тело ответа.

Схема Pydantic для обновления станции выглядит так:

```
# api_code/api/schemas.py
from typing import Optional

class StationUpdate(StationBase):
    code: Optional[str] = None
    country: Optional[str] = None
    city: Optional[str] = None
```

Все свойства объявлены как Optional, поскольку мы хотим допустить частичное обновление — вызывающая сторона может передать только те поля, которые хочет изменить.

Теперь взглянем на код функции из модуля crud, которая отвечает за обновление станции:

```
# api_code/api/crud.py
def update_station(
    db: Session, station: schemas.StationUpdate, station_id: int
):
    stm = (
        update(models.Station)
        .where(models.Station.id == station_id)
        .values(station.model_dump(exclude_unset=True))
    )
    result = db.execute(stm)
    db.commit()
    return result.rowcount
```

Функция update_station() принимает аргументы, необходимые для идентификации станции, которую нужно обновить, а также данные станции, которые будут использованы для обновления записи в базе, и стандартную сессию базы db.

Мы формируем выражение с помощью вспомогательной функции update() из SQLAlchemy. С помощью where() фильтруем запись по id, а новые значения передаем в виде словаря, полученного из объекта Pydantic station, при этом ис-

ключаем все поля, которые не были явно переданы в запросе. Это позволяет выполнять частичное обновление. Если не указать `exclude_unset=True`, то в словарь попадут все поля, не переданные в запросе, — они будут установлены в значения по умолчанию (например, `None`), чего мы не хотим.

Строго говоря, для частичного обновления корректнее использовать метод `PATCH`, но на практике часто используют `PUT` как для полного, так и для частичного обновления.

Мы выполняем выражение и возвращаем количество строк, затронутых данной операцией. В теле конечной точки эта информация не используется, но вы можете попробовать добавить ее сами, это хорошее упражнение. Чуть позже вы увидите, как использовать это значение при удалении станции.

Теперь вызовем эту конечную точку для станции с `ID = 12`, которую мы создали ранее:

```
$ http PUT http://localhost:8000/stations/12 \
code=SMC country=Some-Country city=Some-city
HTTP/1.1 204 No Content
...
```

Результат ожидаемый. Проверим, что обновление действительно произошло:

```
$ http http://localhost:8000/stations/12
HTTP/1.1 200 OK
...
{
    "city": "Some-city",
    "code": "SMC",
    "country": "Some-Country",
    "id": 12
}
```

Все три поля объекта с `ID = 12` были изменены. Теперь выполним частичное обновление:

```
$ http PUT http://localhost:8000/stations/12 code=xxx
HTTP/1.1 204 No Content
...
```

На этот раз мы обновили только поле `code`. Проверим снова:

```
$ http http://localhost:8000/stations/12
HTTP/1.1 200 OK
...
{
    "city": "Some-city",
```

```
    "code": "xxx",
    "country": "Some-Country",
    "id": 12
}
```

Как и ожидалось — изменено только поле `code`[1].

Удаление данных

Наконец, рассмотрим способы удаления станции. Как обычно, начнем с конечной точки:

```
# api_code/api/stations.py
@router.delete("/{station_id}", tags=["Stations"])
def delete_station(
    station_id: int, db: Session = Depends(get_db)
):
    row_count = crud.delete_station(db=db, station_id=station_id)

    if row_count:
        return Response(status_code=status.HTTP_204_NO_CONTENT)
    return Response(status_code=status.HTTP_404_NOT_FOUND)
```

Чтобы удалить станции, необходимо настроить маршрутизатор на прием запросов `DELETE`. URL тот же, который использовался для получения и обновления станции. Здесь поведение определяется не адресом, а HTTP-методом, указанным в запросе. Функция `delete_station()` принимает `station_id` и стандартную сессию базы данных `db`.

В теле функции мы получаем количество строк, затронутых операцией удаления. Если была удалена хотя бы одна строка, то возвращается статус `204 No Content`, сигнализирующий, что удаление прошло успешно. Если не была затронута ни одна строка, то возвращается статус `404 Not Found`. Обратите внимание: метод обновления тоже можно было бы реализовать аналогично — с проверкой количества затронутых строк. Но мы выбрали другой подход, чтобы показать вам разные варианты реализации.

Теперь рассмотрим функцию из модуля `crud`, которая выполняет удаление:

```
# api_code/api/crud.py
def delete_station(db: Session, station_id: int):
    stm = delete(models.Station).where(
        models.Station.id == station_id
    )
    result = db.execute(stm)
    db.commit()
    return result.rowcount
```

---

[1] На практике часто в ответе на PUT-запрос отдают тело измененного ресурса с кодом ответа 200 OK, чтобы не приходилось на клиентской части перезапрашивать данные. — *Примеч. науч. пер.*

В этой функции используется вспомогательный метод `delete()` из SQLAlchemy. Аналогично тому, как мы поступали при обновлении, формируем выражение, которое находит станцию по ее идентификатору и указывает на ее удаление. Мы выполняем данное выражение и возвращаем количество затронутых строк.

Теперь посмотрим, как эта конечная точка работает на практике. Сначала — успешный сценарий:

```
$ http DELETE http://localhost:8000/stations/12
HTTP/1.1 204 No Content
...
```

Получен статус `204 No Content`, что подтверждает успешное удаление. Проверим это косвенно: попробуем удалить станцию с `ID = 12` еще раз. На этот раз ожидаем, что станции уже нет, и в ответ вернется статус `404 Not Found`:

```
$ http DELETE http://localhost:8000/stations/12
HTTP/1.1 404 Not Found
...
```

Как и ожидалось, получен статус `404 Not Found`. Он означает следующее: станция с `ID = 12` не найдена. Это подтверждает, что первое удаление прошло успешно. В модуле `stations.py` есть еще несколько конечных точек, с которыми вам стоит ознакомиться.

Кроме того, мы реализовали конечные точки, позволяющие создавать, читать, обновлять и удалять данные о пользователях, поездах и билетах. Единственное их различие заключается в том, с какими моделями базы данных и Pydantic они работают. С точки зрения логики и структуры они полностью аналогичны рассмотренным выше и не добавили бы ничего нового к нашему обзору. Поэтому теперь рассмотрим пример аутентификации пользователя.

Аутентификация пользователя

В этом проекте аутентификация осуществляется с помощью JWT. О том, что такое JWT, можно прочитать в главе 9.

Начнем с конечной точки аутентификации, определенной в модуле `users.py`:

```
# api_code/api/users.py
from .util import InvalidToken, create_token, extract_payload

@router.post("/authenticate", tags=["Auth"])
def authenticate(
    auth: Auth,
    db: Session = Depends(get_db),
    settings: Settings = Depends(get_settings),
):
    db_user = crud.get_user_by_email(db=db, email=auth.email)
    if db_user is None:
```

```
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"User {auth.email} not found.",
        )

    if not db_user.is_valid_password(auth.password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Wrong username/password.",
        )

    payload = {
        "email": auth.email,
        "role": db_user.role.value,
    }
    return create_token(payload, settings.secret_key)
```

Маршрутизатор использует префикс `"/users"`. Аутентификация требует отправки запроса `POST` в соответствующую конечную точку. Функция принимает объект схемы Pydantic `Auth`, стандартную сессию базы данных `db` и объект `settings`, который нужен для доступа к секретному ключу, используемому при создании токена.

Если пользователь не найден, то возвращается статус `404 Not Found`. Если найден, но введенный пароль не совпадает с хранящимся в базе, то возвращается статус `401 Unauthorized`. И наконец, если пользователь найден и пароль верный — создается токен с двумя утверждениями: `email` и `role`. Роль (`role`) будет использоваться при проверке прав доступа (авторизации).

Функция `create_token()` представляет собой обертку над `jwt.encode()` и добавляет в полезную нагрузку токена пару временных меток. Показывать ее код здесь не имеет смысла. Вместо этого рассмотрим модель `Auth`:

```
# api_code/api/schemas.py
class Auth(BaseModel):
    email: str
    password: str
```

Аутентификация пользователей выполняется по адресу электронной почты (она выступает в роли имени пользователя) и паролю. Именно поэтому в модели SQLAlchemy `User` для поля `email` задано ограничение уникальности. Нам нужно, чтобы у каждого пользователя был уникальный логин, а `email` — одно из самых распространенных решений, позволяющих добиться этого.

Выполним вызов этой конечной точки:

```
$ http POST http://localhost:8000/users/authenticate \
email="fabrizio.romano@example.com" password="f4bPassword"
HTTP/1.1 200 OK
...

"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....01GK4QyzZje8NKMzBBVckc"
```

В ответе мы получили токен (в приведенном фрагменте он обрезан) и теперь можем использовать его. Пользователь, которого мы аутентифицировали, — это администратор. Поэтому покажем, как могла бы выглядеть конечная точка удаления станции, если бы мы захотели разрешить выполнение такой операции только администраторам.

Взгляните на код:

```
# api_code/api/admin.py
from .util import is_admin

router = APIRouter(prefix="/admin")

def ensure_admin(settings: Settings, authorization: str):
    if not is_admin(
        settings=settings, authorization=authorization
    ):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="You must be admin to access this endpoint.",
        )

@router.delete("/stations/{station_id}", tags=["Admin"])
def admin_delete_station(
    station_id: int,
    authorization: Optional[str] = Header(None),
    settings: Settings = Depends(get_settings),
    db: Session = Depends(get_db),
):
    ensure_admin(settings, authorization)
    row_count = crud.delete_station(db=db, station_id=station_id)
    if row_count:
        return Response(status_code=status.HTTP_204_NO_CONTENT)
    return Response(status_code=status.HTTP_404_NOT_FOUND)
```

Как видите, объявление и логика этой конечной точки почти полностью совпадают с обычной версией. Единственное важное различие — перед попыткой удалить станцию вызывается функция `ensure_admin()`. Внутри конечной точки мы получаем заголовок `Authorization` из запроса — именно в нем содержится токен. Чтобы передать его в `ensure_admin()`, мы добавляем в сигнатуру параметр с типом `Optional[str]`, который извлекается из заголовка с помощью объекта `Header`.

Функция `ensure_admin()` делегирует проверку функции `util.is_admin()`, которая распаковывает токен, проверяет его корректность, извлекает поле `role` из полезной нагрузки и проверяет, соответствует ли оно значению `admin`. Если все проверки пройдены, то `is_admin()` возвращает `True`, в противном случае — `False`. Функция `ensure_admin()` не делает ничего, если пользователь — администратор, но при других условиях она вызывает `HTTPException` со статусом `403 Forbidden`. Это означает, что если у пользователя нет прав на выполнение операции, то выполнение тела конечной точки прерывается на самой первой строке.

Существуют более сложные способы реализации аутентификации и авторизации, но описывать их в рамках текущей главы было бы неудобно. Зато этот простой пример вполне подойдет для усвоения базовых принципов реализации таких механизмов в API.

Документирование API

Составление документации к API — утомительное занятие. Одно из преимуществ использования FastAPI заключается в том, что проект можно не документировать вручную: документация автоматически генерируется самим фреймворком. Это становится возможным благодаря аннотациям типов и использованию моделей Pydantic.

Убедитесь, что ваш API запущен, затем откройте браузер и перейдите по адресу http://localhost:8000/docs. Откроется страница, которая будет выглядеть примерно так (рис. 14.3).

Рис. 14.3. Фрагмент автоматически сгенерированной документации FastAPI

На рис. 14.3 показан список конечных точек. Они сгруппированы по категориям с помощью параметра `tags`, который мы указывали при описании каждой точки. Эта документация позволяет не только подробно изучить каждую точку, но и взаимодействовать с ней: можно отправлять запросы прямо со страницы и тут же получать ответы.

Что дальше

Теперь вы должны иметь базовое представление о проектировании API и ключевых концепциях. Конечно, подробное изучение кода из этой главы поможет лучше разобраться в материале и, вероятно, вызовет новые вопросы. Если вы захотите углубиться в тему, то вот вам несколько рекомендаций.

- Детально изучите FastAPI. На официальном сайте есть учебные материалы как для начинающих, так и для опытных разработчиков. В них гораздо больше информации, чем мы могли бы уместить в одной главе.
- На основе кода из этой главы усовершенствуйте API, добавив расширенные возможности поиска и фильтрации. Попробуйте реализовать более сложную систему аутентификации, а также изучите использование фоновых задач, сортировки и постраничной навигации. Раздел администратора можно дополнить другими конечными точками, доступными только администраторам.
- Доработайте конечную точку бронирования билета так, чтобы она проверяла наличие свободных мест в поезде. Для каждого поезда указано количество вагонов первого и второго классов, а также количество мест в каждом вагоне. Мы специально спроектировали модель поезда таким образом, чтобы вы могли потренироваться на этом упражнении.
- Добавьте тесты для существующих конечных точек, а также для всего, чем вы дополните код.
- Познакомьтесь с WSGI (https://wsgi.readthedocs.io) — стандартом для синхронных Python-приложений, а если вы знакомы с асинхронным программированием, то изучите его асинхронный аналог ASGI (https://asgi.readthedocs.io).
- Изучите, как работает middleware в FastAPI, а также такие важные понятия, как *совместное использование ресурсов между разными источниками* (Cross-Origin Resource Sharing, CORS), которые особенно актуальны при развертывании API в реальных условиях.
- Познакомьтесь с другими фреймворками для разработки API, например Falcon (https://falcon.readthedocs.io) или Django Rest Framework (https://www.django-rest-framework.org).
- Изучите подробнее концепцию передачи состояния представления (Representational State Transfer, REST). Она используется повсеместно, но подходы к созданию API на основе REST могут сильно различаться.
- Погрузитесь в более сложные темы проектирования API, такие как управление версиями, форматы данных, протоколы и другие технические детали. Узнайте, как работают заголовки и какую роль они могут играть.
- Если вы знакомы с асинхронным программированием, то попробуйте переписать код из этой главы так, чтобы он стал асинхронным.

Не забудьте добавить в файл .env строку DEBUG=true, когда работаете с API. Это действие активирует автоматический вывод SQL-запросов в терминал, что помогает убедиться в их соответствии ожиданиям. Такой способ отладки особенно полезен при работе со сложной логикой в SQLAlchemy.

Проектирование API — очень важный навык. Мы не устанем повторять: освоить его необходимо каждому разработчику.

Резюме

В этой главе вы познакомились с миром API. Мы начали с краткого обзора работы Интернета, а затем перешли к FastAPI — фреймворку, который активно использует аннотации типов. Напомним, они были впервые представлены в главе 12.

Мы поговорили об API в целом: о разных способах их классификации, целях использования и преимуществах. Кроме того, рассмотрели протоколы и форматы обмена данными.

Затем мы перешли к исходному коду, проанализировав небольшую часть проекта на FastAPI, написанного специально для этой главы.

В завершение мы предложили список направлений, в которых вы можете развивать свои навыки.

В следующей главе речь пойдет о разработке CLI-приложений на Python.

15 Консольные приложения (CLI)

> Пользовательский интерфейс как шутка. Если вам приходится объяснять его, значит, он так себе.
>
> *Мартин ЛеБлан*

В этой главе вы научитесь создавать на языке Python приложения с *интерфейсом командной строки* (Command-Line Interface, CLI), также называемые *консольными*. CLI — это пользовательский интерфейс, в котором команды вводятся вручную в консоли или терминале. Из известных примеров можно назвать оболочки *Bash* и *Zsh* в macOS, Linux и других UNIX-подобных системах, а также *командную строку Windows* (Command Prompt) и *PowerShell*. CLI-приложение — это программа, предназначенная в первую очередь для работы в такой среде. Запустить ее очень просто: достаточно ввести команду в терминале, возможно добавив к ней аргументы.

Хотя *графические пользовательские интерфейсы* (Graphical User Interfaces, GUI) и веб-приложения сейчас гораздо популярнее, CLI-приложения по-прежнему занимают свою нишу. Они особенно востребованы у разработчиков, системных и сетевых администраторов и других технических специалистов. Причин популярности несколько. Если вы знаете нужные команды, то с помощью CLI сможете выполнить задачу быстрее, чем в GUI, — вам не придется нажимать кнопки и переходить по меню. Кроме того, большинство оболочек позволяют передавать вывод одной команды на ввод другой — это называется *конвейером*. Такой подход позволяет объединять простые команды в цепочки для выполнения более сложных задач. Команды можно сохранять в сценариях, что делает их воспроизводимыми и удобными для автоматизации. Кроме того, гораздо проще задокументировать процесс, указав точные команды, чем объяснять, где именно щелкать кнопкой мыши в графическом интерфейсе или веб-программе.

CLI-приложения гораздо быстрее и проще разрабатывать и сопровождать, чем графические интерфейсы или веб-приложения. Именно поэтому команды разработчиков нередко создают внутренние инструменты в виде CLI-программ. Такой подход позволяет сократить время и трудозатраты на создание служебных утилит и сосредоточиться на функциях, ориентированных на конечного пользователя.

Освоение принципов создания CLI-приложений — отличный первый шаг на пути к разработке более сложных программ, таких как графические интерфейсы или распределенные системы.

В этой главе мы создадим командное приложение для работы с API железнодорожной системы, который вы изучали в предыдущей главе. В конце будет представлена подборка дополнительных ресурсов, с помощью которых можно углубить знания по разработке CLI-программ.

Аргументы командной строки

Главный интерфейс взаимодействия с CLI-приложением — это аргументы, которые можно передать при запуске программы через терминал. Прежде чем переходить к CLI-проекту для работы с API железнодорожной системы, кратко рассмотрим аргументы командной строки и средства, которые Python предоставляет для их обработки.

Большинство CLI-приложений поддерживают различные *параметры (флаги)*, а также *позиционные аргументы*. Некоторые программы состоят из нескольких *подкоманд*, каждая из которых имеет собственный набор параметров и аргументов.

Позиционные аргументы

Такие аргументы представляют собой основные данные или объекты, с которыми должно работать приложение. Их нужно указывать в определенном порядке, и, как правило, они обязательны.

Рассмотрим, например, такую команду:

```
$ cp original.txt copy.txt
```

Она создает копию файла `original.txt` с именем `copy.txt`. Оба аргумента — `original.txt` и `copy.txt` — являются обязательными. Если поменять их местами, то смысл команды изменится.

Параметры

Параметры позволяют изменить поведение приложения. Обычно они необязательны и задаются либо одной буквой с префиксом `-`, либо словом с префиксом `--`. Параметры можно указывать в любом порядке и в любой позиции командной строки, даже после позиционных аргументов или между ними.

Например, многие программы принимают параметр `-v` или `--verbose` для включения расширенного вывода. Некоторые параметры ведут себя как переключатели:

их наличие (или отсутствие) включает или отключает определенную функцию. Другие параметры требуют дополнительного значения, в роли которого может выступать, например, имя файла, шаблон или путь.

Рассмотрим такую команду:

```
$ grep -r --exclude '*.txt' hello .
```

Она рекурсивно обходит текущий каталог и ищет строку `"hello"` во всех файлах, кроме тех, которые заканчиваются на `.txt`. Параметр `-r` заставляет `grep` выполнять поиск рекурсивно — без него программа завершится с ошибкой при попытке обработать каталог вместо обычного файла. Параметр `--exclude` требует аргумент — шаблон имени файла (`'*.txt'`) — и исключает из поиска все файлы, соответствующие этому шаблону.

В операционной системе Windows параметры традиционно начинаются с косой черты (/), а не с дефиса. Однако многие современные кросс-платформенные приложения используют дефисы, чтобы быть совместимыми с UNIX-подобными системами.

Подкоманды

Сложные приложения часто разделяются на несколько подкоманд. Отличный пример — система контроля версий *Git*. Рассмотрим, например, такие команды:

```
$ git commit -m "Fix some bugs"
```

и

```
$ git ls-files -m
```

Здесь `commit` и `ls-files` — это подкоманды приложения `git`. Подкоманда `commit` создает новый коммит, используя текст, переданный через параметр `-m` (`"Fix some bugs"`), в качестве сообщения коммита. Команда `ls-files` показывает информацию о файлах в репозитории. Параметр `-m` для этой команды означает: показать только те файлы, в которых есть изменения, но они еще не добавлены в индекс (не подготовлены к коммиту).

Использование подкоманд помогает структурировать и упрощать интерфейс приложения, позволяя пользователям быстрее находить нужные функции. Каждая подкоманда может иметь собственную справочную информацию, что облегчает изучение отдельных функций, избавляя от необходимости читать полную документацию ко всему приложению. Кроме того, подкоманды способствуют модульности кода, упрощают сопровождение и позволяют добавлять новые команды, не затрагивая уже существующие.

Анализ аргументов

Python-приложения могут получить доступ к аргументам командной строки через список `sys.argv`. Напишем простой сценарий, который выводит значение `sys.argv`:

```
# argument_parsing/argv.py
import sys

print(sys.argv)
```

Если запустить его без аргументов, то вывод будет примерно таким:

```
$ python argument_parsing/argv.py
['argument_parsing/argv.py']
```

Если передать аргументы, то результат изменится:

```
$ python argument_parsing/argv.py your lucky number is 13
['argument_parsing/argv.py', 'your', 'lucky', 'number', 'is', '13']
```

Как видите, `sys.argv` представляет собой список строк. Первый элемент — это команда, с помощью которой запускается приложение. Остальные элементы являются переданными аргументами командной строки.

Для простых приложений, не использующих параметры, можно извлекать позиционные аргументы напрямую из `sys.argv`. Но если приложение принимает флаги и параметры, то логика разбора может быстро усложниться. На этот случай в стандартной библиотеке Python есть модуль `argparse`, который предоставляет удобный механизм для разбора аргументов командной строки, использование которого избавляет от необходимости вручную писать сложный код.

Существуют и сторонние библиотеки, являющиеся альтернативами argparse. В этой главе мы их рассматривать не будем, но в конце приведем ссылки на самые популярные.

Для примера мы написали скрипт, который принимает имя и возраст в качестве позиционных аргументов и выводит приветствие. Если указать имя `Heinrich` и возраст `42`, то программа должна вывести `Hi Heinrich. You are 42 years old.`. Кроме того, можно задать пользовательское приветствие вместо `"Hi"`, используя параметр `-g`. Добавление параметра `-r` или `--reverse` приведет к обратному выводу сообщения (текст будет напечатан задом наперед).

```
# argument_parsing/greet.argparse.py
import argparse
```

```
def main():
    args = parse_arguments()
    print(args)

    msg = "{greet} {name}. You are {age} years old.".format(
        **vars(args)
    )
    if args.reverse:
        msg = msg[::-1]

    print(msg)

def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument("name", help="Your name")
    parser.add_argument("age", type=int, help="Age")
    parser.add_argument("-r", "--reverse", action="store_true")
    parser.add_argument(
        "-g", default="Hi", help="Custom greeting", dest="greet"
    )
    return parser.parse_args()

if __name__ == "__main__":
    main()
```

Теперь подробнее рассмотрим функцию `parse_arguments()`. Сначала мы создаем экземпляр класса `ArgumentParser`. Затем поочередно добавляем поддерживаемые аргументы, вызывая метод `add_argument()`. Мы начинаем с позиционных аргументов `name` и `age`, указываем справочную строку (`help`) для каждого, а для аргумента `age` задаем тип `int`. Если не указать тип, то аргумент будет воспринят как строка. Затем мы добавляем необязательный аргумент `-r` или `--reverse`. Еще один параметр — `-g`, задает приветствие и имеет значение по умолчанию `"Hi"`. В конце мы вызываем метод `parse_args()` — он анализирует аргументы командной строки и возвращает объект `Namespace`, содержащий разобранные значения.

Ключевое слово `action`, передаваемое в `add_argument()`, определяет, как парсер должен обработать аргумент. Если `action` не указан, то используется значение по умолчанию `"store"`, которое просто сохраняется в соответствующем атрибуте объекта `Namespace`. Если задано `action="store_true"`, то параметр воспринимается как переключатель: если он есть в командной строке — сохраняется `True`, если отсутствует — `False`. Аргумент `dest` позволяет задать имя атрибута, в который будет сохранено значение. Если `dest` не указан, то для позиционных аргументов используется имя аргумента, а для параметров — длинное имя без `--`. Если указано только короткое имя, то используется оно (без `-`).

Теперь посмотрим, как этот скрипт работает при запуске:

```
$ python argument_parsing/greet.argparse.py Heinrich -r 42
Namespace(name='Heinrich', age=42, reverse=True, greet='Hi')
.dlo sraey 24 era uoY .hcirnieH iH
```

Если мы передаем некорректные аргументы, то скрипт выводит сообщение об использовании, в котором указано, какие аргументы ожидаются, а также сообщение об ошибке, поясняющее, что именно было сделано неправильно:

```
$ python argument_parsing/greet.argparse.py -g -r Heinrich 42
usage: greet.argparse.py [-h] [-r] [-g GREET] name age
greet.argparse.py: error: argument -g: expected one argument
```

В этом сообщении об использовании упоминается параметр `-h`, хотя мы его не добавляли. Посмотрим, что он делает:

```
$ python argument_parsing/greet.argparse.py -h
usage: greet.argparse.py [-h] [-r] [-g GREET] name age

positional arguments:
  name           Your name
  age            Age

options:
  -h, --help     show this help message and exit
  -r, --reverse
  -g GREET       custom greeting
```

Парсер автоматически добавляет параметр справки (`-h` или `--help`), который выводит подробную информацию об использовании, в том числе описания `help`, указанные в методах `add_argument()`.

Чтобы вы могли оценить, насколько argparse упрощает написание CLI-приложений, в исходном коде к этой главе мы добавили версию скрипта приветствия без использования argparse. Вы найдете ее в файле argument_parsing/greet.argv.py.

В этом разделе мы лишь слегка коснулись возможностей `argparse`. Некоторые более сложные функции мы рассмотрим далее — когда начнем работу с CLI-приложением для работы с API железнодорожной системы.

Создание CLI-клиента для работы с API железнодорожной системы

Теперь, когда мы обсудили основы разбора аргументов командной строки, можно перейти к созданию более сложного CLI-приложения. Исходный код проекта находится в папке с материалами к этой главе. Начнем с обзора содержимого каталога `project`:

```
$ tree -a project
project
├── .env.example
```

```
├── railway_cli
│   ├── __init__.py
│   ├── __main__.py
│   ├── api
│   │   ├── __init__.py
│   │   ├── client.py
│   │   └── schemas.py
│   ├── cli.py
│   ├── commands
│   │   ├── __init__.py
│   │   ├── admin.py
│   │   ├── base.py
│   │   └── stations.py
│   ├── config.py
│   └── exceptions.py
└── secrets
    ├── railway_api_email
    └── railway_api_password
```

Файл `.env.example` представляет собой шаблон для создания конфигурационного файла `.env`, используемого в приложении. Файлы в папке `secrets` содержат учетные данные, необходимые для аутентификации в API железнодорожной системы от имени администратора.

Чтобы примеры из этого раздела работали корректно, запустите API, разработанный в главе 14. Создайте копию файла .env.example с именем .env. Убедитесь, что в файле .env указан корректный URL для доступа к API.

Папка `railway_cli` — это пакет Python, содержащий CLI-приложение для работы с API железнодорожной системы. В подпакете `api` находится код для взаимодействия с API. В подпакете `commands` реализованы подкоманды CLI-приложения. Модуль `exceptions.py` определяет исключения, которые могут возникнуть при работе приложения. Модуль `config.py` содержит код для обработки глобальных параметров конфигурации. Главная функция, запускающая CLI-приложение, находится в модуле `cli.py`. Файл `__main__.py` — специальный модуль, который делает пакет исполняемым. Если запустить пакет с помощью команды наподобие этой:

```
$ python -m railway_cli
```

то Python загрузит и выполнит модуль `__main__.py`. Содержимое данного файла показано ниже:

```
# project/railway_cli/__main__.py
from . import cli

cli.main()
```

Этот модуль только импортирует `cli` и вызывает функцию `cli.main()`, которая и является основной точкой входа в CLI-приложение.

Взаимодействие с API железнодорожной системы

Прежде чем перейти к коду интерфейса командной строки, кратко рассмотрим клиентскую часть, отвечающую за работу с API. Мы не будем подробно разбирать исходный код, а дадим лишь общий обзор. Остальное предлагаем изучить самостоятельно.

В подпакете `api` находятся два модуля — `client.py` и `schemas.py`:

- `schemas.py` содержит модели Pydantic, описывающие объекты, которые мы ожидаем получить от API (в этом проекте определены только модели для станций и поездов);
- `client.py` содержит три класса и несколько вспомогательных функций:
  - `HTTPClient` — универсальный класс для выполнения HTTP-запросов. Это обертка над объектом `Session` из библиотеки `requests`. В нем реализованы методы для работы с основными HTTP-глаголами (`GET`, `POST`, `PUT` и `DELETE`). Этот класс обрабатывает ошибки и извлекает данные из ответов API;
  - `StationClient` — более специализированный клиент, предназначенный для работы с конечными точками API, связанными со станциями;
  - `AdminClient` — клиент для работы с административными конечными точками. В частности, он предоставляет метод для аутентификации пользователя через `users/authenticate`, а также метод для удаления станции по адресу `admin/stations/{station_id}`.

Создание интерфейса командной строки

Начнем знакомство с кодом командного интерфейса с модуля `cli.py`. Мы будем разбирать его по частям, начиная с функции `main()`:

```
# project/railway_cli/cli.py
import argparse
import sys

from . import __version__, commands, config, exceptions
from .commands.base import Command

def main(cmdline: list[str] | None = None) -> None:
    arg_parser = get_arg_parser()
    args = arg_parser.parse_args(cmdline)

    try:
        # В переменной args.command ожидается класс команды (Command)
        command: Command = args.command(args)
        command.execute()
    except exceptions.APIError as exc:
        sys.exit(f"API error: {exc}")
    except exceptions.CommandError as exc:
        sys.exit(f"Command error: {exc}")
    except exceptions.ConfigurationError as exc:
        sys.exit(f"Configuration error: {exc}")
```

В начале модуля выполняются импорты: из стандартной библиотеки — модули `argparse` и `sys`, из текущего пакета — `__version__`, `config`, `commands` и `exceptions`, а также класс `Command` из модуля `commands.base`.

В Python принято, чтобы модули и пакеты предоставляли информацию о своей версии через переменную __version__. Обычно это строка, определенная в файле __init__.py верхнего уровня.

В функции `main()` сначала вызывается функция `get_arg_parser()`, чтобы можно было получить экземпляр `ArgumentParser`. Затем вызывается метод `parse_args()` — он разбирает аргументы командной строки. Ожидается, что возвращенный объект `Namespace` будет содержать атрибут `command`, который представляет собой класс, унаследованный от `Command`. Создается экземпляр этого класса, которому передается объект с разобранными аргументами. Затем вызывается метод `execute()` — именно он выполняет команду.

Обработка ошибок осуществляется через исключения `APIError`, `CommandError`, `ConfigurationError`. Если происходит одно из них, то вызывается `sys.exit()` с соответствующим сообщением об ошибке. Это единственные типы исключений, которые может сгенерировать пользовательский код CLI-приложения. Если произойдет любая непредусмотренная ошибка, то Python завершит выполнение программы и выведет полную трассировку исключения в консоль. Такой подход может показаться не слишком дружественным, но упрощает отладку. Кроме того, пользователи CLI-приложений, как правило, обладают большим объемом технических знаний, поэтому относятся к трассировкам не так настороженно, как пользователи графических интерфейсов или веб-приложений.

Стоит также отметить: функция main() принимает необязательный параметр cmdline, который передается в parse_args(). Если cmdline — None, то parse_args() будет использовать sys.argv. Но если передать список строк, то будет разобран именно он. Такая структура особенно полезна для модульного тестирования — она позволяет не изменять глобальное значение sys.argv в тестах.

Мы скоро вернемся к классу `Command` и увидим, как устроен разбор аргументов, который приводит к возврату экземпляра нужной команды. Но сначала рассмотрим саму функцию `get_arg_parser()`:

```
# project/railway_cli/cli.py
def get_arg_parser() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser(
        prog=__package__,
        description="Commandline interface for the Railway API",
        formatter_class=argparse.ArgumentDefaultsHelpFormatter,
    )
    parser.add_argument(
        "-V",
        "--version",
```

```
        action="version",
        version=f"%(prog)s {__version__}",
    )
    config.configure_arg_parser(parser)
    commands.configure_parsers(parser)
    return parser
```

Функция `get_arg_parser()` создает и настраивает экземпляр `ArgumentParser` для CLI-приложения. Аргумент `prog` указывает имя программы, которое будет отображаться в справке. Обычно `argparse` получает его из `sys.argv[0]`, но если приложение запускается как пакет через `python -m имя_пакета`, то имя по умолчанию будет `__main__.py`. Поэтому мы переопределяем `prog`, указывая имя пакета вручную. Аргумент `description` задает краткое описание программы, которое также выводится в справке. Аргумент `formatter_class` определяет формат справочного сообщения. Мы используем `ArgumentDefaultsHelpFormatter`, который добавляет значения по умолчанию для всех параметров прямо в текст справки. Мы добавляем параметр `-V` или `--version`, указывая действие `"version"`. Если этот параметр передан, то программа выведет строку с номером версии и завершит выполнение. В завершение вызываются две функции для дополнительной настройки парсера: `config.configure_arg_parser()` и `commands.configure_parsers()`. После этого настроенный парсер возвращается.

Система импорта Python автоматически устанавливает атрибут `__package__` для каждого импортируемого модуля — он указывает, к какому пакету принадлежит модуль.

Далее мы рассмотрим настройку аргументов командной строки в модулях `config` и `commands`.

Конфигурационные файлы и секреты

Помимо аргументов командной строки, многие CLI-приложения считывают настройки из конфигурационных файлов. Такие файлы обычно используются для параметров, которые редко меняются между запусками приложения, — например, это может быть URL API. Пользователю было бы неудобно вводить подобные параметры вручную каждый раз при запуске программы.

Еще один распространенный сценарий использования конфигурационных файлов — это хранение паролей и других секретов. Передавать пароли как аргументы командной строки — плохая практика с точки зрения безопасности, поскольку в большинстве операционных систем любой вошедший в систему пользователь может увидеть полную команду, с помощью которой запущено приложение. Кроме того, оболочки обычно сохраняют историю команд, что тоже может привести к утечке пароля, переданного через аргумент. Гораздо безопаснее хранить пароли в конфигурационных или специальных секретных файлах. Это файлы, имена которых соответствуют имени секрета, а содержимым является сам секрет (например, пароль).

Очень важно! Никогда не храните секреты вместе с исходным кодом. Особенно если вы используете систему контроля версий (например, Git или Mercurial) — ни в коем случае не коммитьте секреты в репозиторий.

В CLI-приложении `railway_cli` за работу с конфигурацией и секретами отвечает модуль `config`. Для управления настройками используется библиотека `pydantic-settings`, с которой вы познакомились в главе 14. Теперь разберем код этого модуля по частям:

```
# project/railway_cli/config.py
import argparse
from getpass import getpass

from pydantic import EmailStr, Field, SecretStr, ValidationError
from pydantic_settings import BaseSettings, SettingsConfigDict

from .exceptions import ConfigurationError

class Settings(BaseSettings):
    url: str
    secrets_dir: str | None = None

class AdminCredentials(BaseSettings):
    model_config = SettingsConfigDict(env_prefix="railway_api_")
    email: EmailStr
    password: SecretStr = Field(
        default_factory=lambda: SecretStr(
            getpass(prompt="Admin Password: ")
        )
    )
```

После операторов импорта в начале файла определяются два класса: `Settings` и `AdminCredentials`. Оба наследуются от `pydantic_settings.BaseSettings`. Класс `Settings` определяет общие настройки для CLI-приложения `railway_cli`:

- `url` — содержит URL API железнодорожной системы;
- `secrets_dir` — путь к каталогу, в котором хранятся файлы секретов. В частности, из этих файлов будет загружаться учетная запись администратора для доступа к API.

Класс `AdminCredentials` содержит параметры для аутентификации администратора в API. Внутри вложенного класса `SettingsConfigDict` задан параметр `env_prefix` — он добавляется к именам полей при поиске значений в каталоге секретов. Например, для поля `password` будет искаться файл `railway_api_password`. Класс `AdminCredentials` содержит следующие поля:

- `email` — адрес электронной почты администратора, с помощью которого будет выполняться вход. Мы используем тип `pydantic.EmailStr`, чтобы убедиться, что значение действительно является корректным адресом;

- `password` — пароль администратора. Тип `pydantic.SecretStr` позволяет маскировать значение (например, в логах — звездочками). Если пароль не передается явно (ни через файл секрета, ни через аргументы конструктора), то Pydantic вызывает функцию, переданную через `default_factory`. Здесь используется стандартная функция `getpass()` из библиотеки Python — она безопасно запрашивает пароль у пользователя в интерактивном режиме (без отображения ввода).

Под этими определениями находится функция `configure_arg_parser()`, код которой приведен ниже:

```
# project/railway_cli/config.py
def configure_arg_parser(parser: argparse.ArgumentParser) -> None:
    config_group = parser.add_argument_group(
        "configuration",
        description="""URL-адрес API должен быть указан
        в конфигурационном файле. Адрес электронной почты администратора
        и пароль должны задаваться через файлы в каталоге secrets —
        с именами email и password.""",
    )
    config_group.add_argument(
        "--config-file",
        help="Загрузить конфигурацию из файла",
        default=".env",
    )
    config_group.add_argument(
        "--secrets-dir",
        help="""Каталог secrets. Также может быть указан
        в конфигурационном файле.""",
    )
```

Мы используем метод `add_argument_group()` объекта `ArgumentParser`, чтобы создать группу аргументов с именем `configuration` и описанием. В нее добавляются параметры, позволяющие пользователю указать имя конфигурационного файла и каталог с файлами секретов.

Обратите внимание: группы аргументов не влияют на то, как аргументы разбираются или возвращаются. Они нужны исключительно для структурирования справки — чтобы эти параметры отображались под общим заголовком.

В целях упрощения в этом примере значение по умолчанию для конфигурационного файла — .env. Тем не менее рекомендуется использовать стандартные пути конфигурационных файлов, принятых на платформе, на которой работает приложение. Библиотека platformdirs (https://platformdirs.readthedocs.io) помогает автоматически определять корректные пути для хранения настроек и данных в зависимости от операционной системы.

В завершении модуля `config.py` представлены вспомогательные функции, позволяющие получать настройки и учетные данные администратора:

```
# project/railway_cli/config.py
def get_settings(args: argparse.Namespace) -> Settings:
    try:
        return Settings(_env_file=args.config_file)
    except ValidationError as exc:
        raise ConfigurationError(str(exc)) from exc

def get_admin_credentials(
    args: argparse.Namespace, settings: Settings
) -> AdminCredentials:
    secrets_dir = args.secrets_dir
    if secrets_dir is None:
        secrets_dir = settings.secrets_dir

    try:
        return AdminCredentials(_secrets_dir=secrets_dir)
    except ValidationError as exc:
        raise ConfigurationError(str(exc)) from exc
```

Функция `get_settings()` создает и возвращает экземпляр класса `Settings`. Аргумент `_env_file=args.config_file` сообщает `pydantic-settings`, из какого файла нужно загружать настройки. Этот путь задается с помощью параметра командной строки `--config-file` (по умолчанию `.env`). Функция `get_admin_credentials()` создает и возвращает экземпляр класса `AdminCredentials`. Аргумент `_secrets_dir` указывает каталог, в котором `pydantic-settings` будет искать файлы с секретами. Если параметр `--secrets-dir` был передан в командной строке, то используется он. Если нет — используется значение `settings.secrets_dir`. Если и оно равно `None`, то секреты из каталога загружаться не будут.

Создание подкоманд

API железнодорожной системы содержит отдельные конечные точки для просмотра списка станций, создания станции, получения отправлений и других действий. Логично выстроить такую же структуру и в нашем CLI-приложении. Существует много способов организовать код для подкоманд. В этом проекте мы выбрали объектно-ориентированный подход: каждая подкоманда реализована в виде отдельного класса, содержащего метод для настройки разбора аргументов (`configure_arg_parser()`) и метод для выполнения команды (`execute()`). Все подкоманды наследуются от базового класса `Command`, который находится в модуле `commands/base.py`:

```
# project/railway_cli/commands/base.py
import argparse
from typing import ClassVar

from ..api.client import HTTPClient
from ..config import get_settings

class Command:
    name: ClassVar[str]
    help: ClassVar[str]
```

```
    def __init__(self, args: argparse.Namespace) -> None:
        self.args = args
        self.settings = get_settings(args)
        self.api_client = HTTPClient(self.settings.url)

    @classmethod
    def configure_arg_parser(
        cls, parser: argparse.ArgumentParser
    ) -> None:
        raise NotImplementedError

    def execute(self) -> None:
        raise NotImplementedError
```

Класс `Command` является обычным. Аннотация `ClassVar` для атрибутов `name` и `help` указывает, что они должны быть атрибутами класса, а не экземпляра. Конструктор `__init__()` принимает объект `argparse.Namespace`, полученный при разборе аргументов, и сохраняет его в `self.args`. Затем вызывается функция `get_settings()`, чтобы можно было загрузить конфигурацию из файла. Кроме того, создается экземпляр `HTTPClient` (из модуля `api/client.py`) и сохраняется в `self.api_client`.

Методы `configure_arg_parser()` (метод класса) и `execute()` по умолчанию вызывают `NotImplementedError`, то есть обязаны быть переопределены в подклассах.

Чтобы настроить разбор аргументов подкоманд, нужно создать парсер для каждой подкоманды, а затем вызвать `configure_arg_parser()` соответствующего класса. Этим занимается функция `commands.configure_parsers()`, и сейчас мы рассмотрим, как она устроена:

```
# project/railway_cli/commands/__init__.py
import argparse

from .admin import admin_commands
from .base import Command
from .stations import station_commands

def configure_parsers(parser: argparse.ArgumentParser) -> None:
    subparsers = parser.add_subparsers(
        description="Available commands", required=True
    )
    command: type[Command]
    for command in [*admin_commands, *station_commands]:
        command_parser = subparsers.add_parser(
            command.name, help=command.help
        )
        command.configure_arg_parser(command_parser)
        command_parser.set_defaults(command=command)
```

Метод `parser.add_subparsers()` возвращает объект, с помощью которого можно прикреплять подкоманды к основному парсеру. Аргумент `description` используется для генерации справки к подкомандам, а `required=True` гарантирует, что если пользователь не указал подкоманду, то будет выведена ошибка.

Далее выполняется перебор списков `admin_commands` и `station_commands`. Для каждой команды создается подпарсер с помощью метода `add_parser()`, в который передается имя подкоманды, а также дополнительные параметры, такие как описание, аргументы и т. д. Метод `add_parser()` возвращает экземпляр `ArgumentParser`, который передается в метод `configure_arg_parser()` соответствующего класса команды. Аннотация `command: type[Command]` означает, что каждый элемент в `admin_commands` и `station_commands` должен быть подклассом `Command`.

Метод `set_defaults()` позволяет задать значения атрибутов, которые будут присутствовать в объекте `Namespace`, возвращаемом парсером, — независимо от того, что именно передано в командной строке. В данном случае мы используем его, чтобы установить атрибут `command` у каждого подпарсера, присваивая ему соответствующий класс команды. Объект `Namespace`, получаемый после разбора аргументов через `parse_args()`, будет содержать атрибуты только от одного подпарсера — того, который соответствует выбранной пользователем подкоманде. Благодаря этому при вызове `args.command(args=args)` в функции `cli.main()` создается экземпляр нужного класса команды, соответствующий введенной в командной строке подкоманде.

Теперь, когда мы собрали весь код, настраивающий разбор аргументов, можно посмотреть, какая справка будет сгенерирована, если запустить приложение с флагом `-h`:

```
$ python -m railway_cli -h
usage: railway_cli [-h] [-V] [--config-file CONFIG_FILE]
                   [--secrets-dir SECRETS_DIR]
                   {admin-delete-station,get-station,...}
                   ...
Commandline interface for the Railway API

options:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
configuration:
  The API URL must be set in the configuration file. ...
  --config-file CONFIG_FILE
                        Load configuration from a file (default: .env)
  --secrets-dir SECRETS_DIR
                        The secrets directory. Can also be set
                        via the configuration file. (default: None)

subcommands:
  Available commands:
  {admin-delete-station,get-station,list-stations,...}
    admin-delete-station    Delete a station
    get-station             Get a station
    list-stations           List stations
    create-station          Create a station
    update-station          Update an existing station
    get-arrivals            Get arrivals for a station
    get-departures          Get departures for a station
```

Мы сократили вывод и удалили пустые строки, тем не менее из него видно следующее: вначале отображается сводка по использованию команды, затем — описание, которое мы указали при создании парсера аргументов, после чего идет секция глобальных параметров, вместе с `-h` или `--help`, а также `-V` или `--version`. Затем идет секция `configuration` с описанием и параметрами, которые были настроены в функции `config.configure_arg_parser()`. И наконец, отображается секция `subcommands` с описанием, переданным в `add_subparsers()` внутри `commands.configure_parsers()`, и списком всех доступных подкоманд, каждая из которых сопровождается краткой справочной строкой (`help`), указанной в соответствующем классе.

Итак, мы рассмотрели базовый класс для подкоманд и код, настраивающий парсер аргументов для работы с ними. Теперь перейдем к реализации одной из подкоманд.

Реализация подкоманд

Парсеры подкоманд полностью независимы друг от друга, поэтому при добавлении новых подкоманд не возникает конфликтов между их параметрами. Нужно лишь убедиться, что имена подкоманд уникальны. Благодаря этому можно расширять приложение, добавляя новые команды и при этом не изменяя существующий код. Подход на основе классов, который мы выбрали для этого приложения, упрощает эту процедуру: достаточно создать новый подкласс от `Command`, указать имя команды (`name`), справочную строку (`help`) и реализовать два метода — `configure_arg_parser()` и `execute()`. В качестве примера посмотрим на реализацию команды `create-station`:

```
# project/railway_cli/commands/stations.py
class CreateStation(Command):
    name = "create-station"
    help = "Create a station"

    @classmethod
    def configure_arg_parser(
        cls, parser: argparse.ArgumentParser
    ) -> None:
        parser.add_argument(
            "--code", help="The station code", required=True
        )
        parser.add_argument(
            "--country", help="The station country", required=True
        )
        parser.add_argument(
            "--city", help="The station city", required=True
        )

    def execute(self) -> None:
        station_client = StationClient(self.api_client)
        station = station_client.create(
            code=self.args.code,
            country=self.args.country,
            city=self.args.city,
        )
        print(station)
```

Операторы импорта из модуля `commands/stations.py` в этом фрагменте опущены. Как видите, код команды достаточно простой. Метод `configure_arg_parser()` добавляет три параметра: код станции, город и страну. Все три параметра отмечены как обязательные.

Документация по `argparse` не рекомендует использовать обязательные параметры, однако в некоторых случаях их наличие делает интерфейс более понятным. Когда команда требует несколько аргументов с разным значением, позиционный ввод может сбивать с толку — нужно помнить правильный порядок. Параметры с ключами (например, `--code`, `--city`) позволяют вводить аргументы в любом порядке и упрощают взаимодействие с интерфейсом.

Теперь посмотрим, что произойдет при запуске команды. Сначала используем параметр `-h`, чтобы увидеть справку:

```
$ python -m railway_cli create-station -h
usage: railway_cli create-station [-h] --code CODE --country
                                  COUNTRY --city CITY

options:
  -h, --help            show this help message and exit
  --code CODE           The station code
  --country COUNTRY     The station country
  --city CITY           The station city
```

В справке дано четкое объяснение того, как использовать команду. Теперь выполним саму команду для создания станции:

```
$ python -m railway_cli create-station --code LSB --city Lisbon \
    --country Portugal
id=12 code='LSB' country='Portugal' city='Lisbon'
```

Вывод показывает, что станция была успешно создана и ей присвоен идентификатор `12`.

На этом обзор CLI-приложения для работы с API железнодорожной системы завершен.

Дополнительные ресурсы и инструменты

В этом разделе приведено несколько ссылок на полезные ресурсы, содержащие больше сведений по теме текущей главы, а также указаны инструменты, которые пригодятся при разработке CLI-приложений.

- Мы постарались дать максимум информации, и тем не менее модуль `argparse` предоставляет гораздо больше возможностей, чем было показано в этой главе. Подробную документацию можно найти на сайте Python по адресу https://docs.python.org/3/library/argparse.html.

- Если `argparse` вам не нравится, то можете использовать несколько сторонних библиотек для парсинга аргументов командной строки. Мы рекомендуем попробовать следующие.
  - `Click` — самая популярная сторонняя библиотека для создания CLI-приложений на Python. Помимо анализа командной строки, она предлагает функции для создания интерактивных приложений (например, запросов к пользователю) и вывода с цветовым оформлением. Подробнее см. на сайте https://click.palletsprojects.com.
  - `Typer` — разработана командой, создавшей FastAPI. Переносит принципы FastAPI на разработку CLI-интерфейсов. Подробнее см. на сайте https://typer.tiangolo.com.
- Библиотека `pydantic-settings`, которую мы использовали для управления конфигурацией в текущей главе, а также в главе 14, тоже поддерживает парсинг аргументов командной строки. Подробнее см. на сайте https://docs.pydantic.dev/latest/concepts/pydantic_settings/#command-line-support.
- Современные оболочки поддерживают автодополнение аргументов. Это значительно упрощает работу с CLI-приложениями. Библиотека `argcomplete` (https://kislyuk.github.io/argcomplete/) добавляет автодополнение для `bash` и `zsh` в приложениях, использующих `argparse`.
- *Command Line Interface Guidelines* (https://clig.dev) — руководство с открытым исходным кодом, содержащее отличные рекомендации по проектированию удобных интерфейсов командной строки.

Резюме

В этой главе мы обсуждали разработку приложений с интерфейсом командной строки, создавая CLI-клиент для железнодорожного API, который был реализован в главе 14. Мы выяснили, как обрабатывать аргументы командной строки с помощью стандартного модуля `argparse`. Кроме того, мы обсудили, как структурировать CLI-приложение с помощью подкоманд. Такой подход помогает создавать модульные приложения, которые удобно поддерживать и расширять. В конце главы были приведены ссылки на сторонние библиотеки для создания CLI-приложений на Python, а также на ресурсы, содержащие больше информации по этой теме.

Работа с CLI-приложениями — отличный способ практиковаться в применении навыков, полученных по ходу чтения книги. Мы рекомендуем изучить код из этой главы, попробовать расширить его, добавив новые команды, а также улучшить, внедрив логирование и тесты.

В следующей главе вы узнаете, как упаковывать и публиковать приложения, написанные на Python.

16 Упаковка и публикация приложений

— Итак, есть у вас какой-нибудь сыр или нет?

— Нет.

«Монти Пайтон», скетч «Сырная лавка»

В этой главе вы узнаете, как создать устанавливаемый пакет для проекта на Python и опубликовать его, чтобы им могли пользоваться другие разработчики.

Существует множество причин публиковать свой код. В главе 1 мы говорили, что одно из главных преимуществ Python — это огромная экосистема сторонних пакетов, которые можно бесплатно установить через `pip`. Большинство из них были созданы такими же разработчиками, как вы. Выкладывая собственные проекты, вы помогаете поддерживать развитие сообщества Python. Со временем это пойдет на пользу и вашему коду: чем больше у него пользователей, тем выше вероятность, что ошибки будут обнаружены раньше. И наконец, если вы хотите устроиться на работу разработчиком, то будет очень полезно показать проекты, над которыми вы работали.

Лучший способ научиться упаковывать проекты — пройти весь процесс. Именно этим мы и займемся в данной главе. Мы будем работать с приложением `railway_cli`, которое разработали в главе 15.

Прежде чем перейти к упаковке и публикации, кратко обсудим Python Package Index (PyPI) и основные термины, связанные с упаковкой в Python.

Python Package Index

PyPI (Python Package Index) — это онлайн-репозиторий пакетов Python, расположенный по адресу https://pypi.org. У него есть веб-интерфейс, который позволяет просматривать или искать пакеты и изучать их описание. Кроме того, доступны API, которыми пользуются такие инструменты, как `pip`, чтобы находить и скачивать пакеты для установки. Любой желающий может зарегистрироваться

на PyPI и бесплатно распространять свои проекты, а также устанавливать любые доступные пакеты.

В репозитории находятся проекты, релизы и дистрибутивные пакеты. *Проект* — это библиотека, скрипт или приложение, к которым прилагаются связанные данные или ресурсы. Например, `FastAPI`, `Requests`, `pandas`, `SQLAlchemy` и наше приложение `railway_cli` — все это проекты. Сам `pip` тоже является проектом. *Релиз* — это конкретная версия проекта (моментальный снимок его состояния). Релизы обозначаются номерами версий. Например, `pip 24.2` — это релиз проекта `pip`. *Пакет для распространения (distribution package)* — это архив, помеченный номером релиза, который содержит модули Python, файлы данных и прочее, входящее в релиз. Кроме того, в пакет вносится метаинформация: имя проекта, авторы, номер версии, зависимости и т. д. Такие пакеты иногда называют *дистрибутивами* или просто *пакетами*.

Пакет для распространения важно не путать с импортируемым — последний в Python обозначает модуль, содержащий другие модули (обычно это папка с файлом __init__.py внутри). В этой главе под словом «пакет» мы будем в основном иметь в виду дистрибутивный пакет. Когда нужно будет различать значения, мы уточним, о каком пакете речь: импортируемом или дистрибутивном.

Дистрибутивы бывают двух типов: *исходные дистрибутивы* (source distributions, или *sdist*) — перед установкой требуют этапа сборки; *собранные дистрибутивы* (built distributions) — достаточно просто распаковать файлы в нужные каталоги. Современный формат исходных дистрибутивов был определен в PEP 517 (https://packaging.python.org/specifications/source-distribution-format/). А стандартный формат собранных дистрибутивов называется *wheel* и был впервые описан в PEP 427 (https://packaging.python.org/specifications/binary-distribution-format/). Он заменил устаревший формат *egg*, который больше не используется.

PyPI изначально имел прозвище Cheese Shop — в честь знаменитого скетча комик-группы «Монти Пайтон», который мы цитировали в начале главы. Поэтому формат дистрибутива wheel назван в честь не автомобильного колеса, а сыра в форме колеса.

Чтобы вы могли лучше понять, как все это работает, разберем короткий пример: что происходит, когда мы выполняем команду `pip install`. Мы установим версию 2.32.3 библиотеки `requests`. Как увидеть процесс в `pip`? Мы добавим флаг `-v` трижды, чтобы получить максимально детализированный вывод, а также укажем `--no-cache`, чтобы принудительно скачать пакеты с PyPI (и не использовать локальный кэш). Результат будет выглядеть примерно так (мы убрали часть строк и сократили вывод, полный текст можно найти в исходном коде к этой главе — в файле `pip_install.txt`).

```
$ pip install -vvv --no-cache requests==2.32.3
Using pip 24.2 from ... (python 3.12)
...
1 location(s) to search for versions of requests:
* https://pypi.org/simple/requests/
```

pip сообщает, что нашел информацию о проекте requests по адресу https://pypi.org/simple/requests/. Далее в выводе дается список всех доступных дистрибутивов проекта requests:

```
Found link https://.../requests-0.2.0.tar.gz..., version: 0.2.0
...
Found link https://.../requests-2.32.3-py3-none-any.whl...
  (requires-python:>=3.8), version: 2.32.3
Found link https://.../requests-2.32.3.tar.gz...
  (requires-python:>=3.8), version: 2.32.3
```

Затем pip переходит к загрузке дистрибутива нужного нам релиза. В первую очередь он скачивает метаданные для wheel-дистрибутива:

```
Collecting requests==2.32.3
  ...
  Downloading requests-2.32.3-py3-none-any.whl.metadata (4.6 kB)
```

После этого pip извлекает список зависимостей из метаданных пакета и начинает тем же способом искать и собирать их метаданные. Как только все необходимые пакеты найдены, они скачиваются и устанавливаются:

```
Downloading requests-2.32.3-py3-none-any.whl (64 kB)
...
Installing collected packages: urllib3, ..., certifi, requests
...
Successfully installed ... requests-2.32.3 ...
```

Если бы для какого-либо из пакетов не оказалось подходящего wheel-дистрибутива и pip скачал бы исходный дистрибутив, то перед установкой потребовалось бы выполнить сборку пакета.

Теперь, когда мы выяснили, чем различаются проект, релиз и пакет, можно перейти к подготовке релиза и сборке дистрибутивов для приложения railway_cli.

Упаковка с помощью Setuptools

Для упаковки проекта мы будем использовать библиотеку Setuptools. Это самый старый инструмент упаковки в экосистеме Python, но библиотека до сих пор остается самой популярной и активно развиваемой. Она расширяет функциональность оригинальной системы упаковки из стандартной библиотеки — distutils.

Начиная с Python 3.10 модуль `distutils` был объявлен устаревшим, а в версии 3.12 полностью удален из стандартной библиотеки.

В этом разделе мы рассмотрим, как подготовить проект к сборке пакета с помощью `Setuptools`.

Структура проекта

Существует два распространенных подхода к организации файлов в проекте Python.

- В *структуре src* импортируемые пакеты, предназначенные для распространения, размещаются в папке `src` внутри основной папки проекта.
- В *плоской структуре* импортируемые пакеты располагаются прямо в корневой папке проекта.

Структура с папкой `src` удобна тем, что явно указывает, какие файлы попадут в дистрибутив. Благодаря этому снижается вероятность случайно добавить лишние файлы — например, вспомогательные сценарии, которые предназначены только для разработки. Однако такая структура может быть менее удобной в процессе разработки: если запустить скрипт или открыть консоль Python из корневой папки проекта, то импортировать пакет из `src` напрямую не получится — сначала нужно установить дистрибутив в виртуальное окружение.

Сторонники структуры src считают, что необходимость устанавливать дистрибутив даже во время разработки — это преимущество, поскольку ошибки в упаковке проекта можно выявить на раннем этапе.

В этом проекте используется структура с папкой `src`:

```
$ tree -a railway-project
railway-project
├── .flake8
├── CHANGELOG.md
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── railway_cli
│       ├── __init__.py
│       └── ...
└── test
    ├── __init__.py
    └── test_get_station.py
```

Папка `src/railway_cli` содержит код импортируемого пакета `railway_cli`. Кроме того, в проект добавлены некоторые тесты — их можно найти в папке `test`.

Они служат примером, на основе которого вы можете развивать покрытие кода. Файл `.flake8` содержит настройки для инструмента `flake8`, который проверяет стиль кода и помогает находить нарушения стандарта PEP 8.

Прежде чем перейти к описанию остальных файлов проекта, разберем, как установить проект в виртуальное окружение для разработки.

Установка в режиме разработки

При использовании структуры `src` чаще всего применяется установка в *режиме разработки* (она также называется *установкой в редактируемом режиме* (editable installation)). В результате создается и устанавливается wheel-пакет проекта, но вместо копирования файлов в виртуальное окружение создается ссылка на исходный код из вашей папки. Благодаря этому вы можете импортировать и запускать код, как будто он установлен, но при этом все внесенные изменения сразу вступают в силу — пакет не нужно пересобирать или переустанавливать.

Что ж, начнем. Откройте терминал, перейдите в папку с исходным кодом главы, создайте новое виртуальное окружение, активируйте его и выполните следующую команду:

```
$ pip install -e railway-project
Obtaining file:///.../ch16/railway-project
  Installing build dependencies ... done
  Checking if build backend supports build_editable ... done
  Getting requirements to build editable ... done
  Preparing editable metadata (pyproject.toml) ... done
...
Building wheels for collected packages: railway-cli
  Building editable for railway-cli (pyproject.toml) ... done
  Created wheel for railway-cli: filename=...
  ...
Successfully built railway-cli
Installing collected packages: ... railway-cli
Successfully installed ... railway-cli-0.0.1 ...
```

Как видно из вывода, при использовании параметра `-e` (или `--editable`) `pip` создает пакет и устанавливает его в виртуальное окружение в редактируемом режиме.

Теперь выполните команду:

```
$ python -m railway_cli
```

Она должна сработать так же, как и в предыдущей главе. Чтобы убедиться, что установка действительно редактируемая, измените что-нибудь в коде (например, добавьте вызов `print()`), а затем снова запустите программу — изменения отобразятся сразу же.

Теперь, когда рабочая установка проекта в режиме разработки выполнена, можно перейти к подробному разбору файлов в структуре проекта.

Лог изменений

Наличие лога изменений не является обязательным, но добавлять его в проект считается хорошей практикой. Такой файл помогает кратко зафиксировать изменения, внесенные в каждом выпуске. Это особенно полезно для пользователей: они могут быстро узнать о новых возможностях или о том, как изменилось поведение программы и на что теперь стоит обратить внимание.

В нашем проекте `railway_cli` лог-файл изменений называется `CHANGELOG.md` и представлен в формате *Markdown*.

Лицензия

Проект обязательно должен содержать лицензию — документ, определяющий условия распространения и использования кода. Существует множество лицензий, предусматривающих разную степень свободы действий. Если вы не уверены, какую выбрать, то посетите сайт choosealicense.com — там можно найти понятное сравнение популярных лицензий. Но если возникают сомнения по поводу юридических последствий, то лучше проконсультироваться с юристом.

Проект `railway_cli` распространяется по лицензии MIT. Это простая и широко используемая лицензия, допускающая свободное использование, распространение и изменение кода при условии, что сохраняется изначальное уведомление об авторских правах и сама лицензия.

По сложившейся традиции файл лицензии называется `LICENSE` или `LICENSE.txt`, хотя встречаются и другие варианты, например `COPYING`.

Файл README

В каждом проекте должен быть файл README — он объясняет, что это за проект, зачем создан и как им пользоваться. Такой файл помогает другим понять, что вы сделали, и быстро приступить к работе с вашим кодом. README может быть оформлен как обычный текст или с использованием разметки, такой как *Markdown* или *reStructuredText*. Наиболее распространенные названия: `README`, `README.txt` (если это обычный текст), `README.md` (для Markdown) и `README.rst` (для reStructuredText).

В нашем случае файл `README.md` содержит краткое описание цели проекта и несколько простых инструкций по его использованию.

Markdown и reStructuredText — это популярные языки разметки. Они разрабатывались так, чтобы тексты на них было удобно читать и писать в исходном виде, а при необходимости — легко преобразовывать в HTML для создания простых веб-страниц. Подробнее о них можно узнать на сайтах https://daringfireball.net/projects/markdown/ и https://docutils.sourceforge.io/rst.html.

Файл pyproject.toml

Данный файл был введен в рамках PEP 518 (https://peps.python.org/pep-0518/) и дополнен в PEP 517 (https://peps.python.org/pep-0517/). Эти документы определяют стандарты, позволяющие проектам указывать зависимости, необходимые для сборки, а также задавать инструмент, который должен использоваться для сборки пакета. В проекте, использующем библиотеку `Setuptools`, такой файл выглядит следующим образом:

```
# railway-project/pyproject.toml
[build-system]
requires = ["setuptools>=66.1.0", "wheel"]
build-backend = "setuptools.build_meta"
```

Здесь указано, что для сборки пакета требуется как минимум версия 66.1.0 библиотеки `Setuptools` (это минимально совместимая с Python 3.12 версия), а также любой выпуск проекта `wheel`, являющегося эталонной реализацией формата wheel-дистрибутива. Поле `requires` в данном разделе не описывает зависимости, необходимые для запуска проекта, — только те, которые требуются для сборки дистрибутива. О том, как указать зависимости для выполнения кода, мы поговорим чуть позже в этой главе.

Поле `build-backend` задает объект Python, который будет использоваться для сборки пакета. В случае с библиотекой `Setuptools` это модуль `build_meta`, входящий в импортируемый пакет `setuptools`.

PEP 518 также допускает размещение конфигураций других инструментов разработки в файле `pyproject.toml`. Разумеется, сами инструменты должны поддерживать чтение конфигурации из этого файла:

```
# railway-project/pyproject.toml
[tool.black]
line-length = 66

[tool.isort]
profile = 'black'
line_length = 66
```

Мы добавили в файл `pyproject.toml` настройки для `black`, популярного инструмента для автоматического форматирования кода, и `isort` — утилиты для упорядочения импортов по алфавиту. Оба инструмента настроены на длину строки 66 символов, чтобы код помещался на странице книги. Кроме того, для `isort` указана совместимость с `black`, чтобы оба инструмента работали согласованно[1].

---

[1] В 2025 году на смену black, isort и линтерам, например flake8, пришел ruff (https://docs.astral.sh/ruff/) — чрезвычайно быстрый линтер и форматтер для Python, написанный на Rust. — *Примеч. науч. ред.*

Больше информации о black и isort можно найти на их сайтах https://black.readthedocs.io/ и https://pycqa.github.io/isort.

PEP 621 (https://peps.python.org/pep-0621/) ввел возможность указывать всю метаинформацию о проекте непосредственно в файле `pyproject.toml`. Поддержка этой возможности появилась в `Setuptools` начиная с версии 61.0.0. Мы подробнее рассмотрим ее в следующем подразделе.

Метаданные пакета

Метаинформация о проекте задается в секции `[project]` файла `pyproject.toml`. Разберем ее по частям:

```
# railway-project/pyproject.toml
[project]
name = "railway-cli"
authors = [
  {name="Heinrich Kruger", email="heinrich@example.com"},
  {name="Fabrizio Romano", email="fabrizio@example.com"},
]
```

Секция начинается с заголовка `[project]`. Первые два параметра — это имя проекта и список авторов, включающий в себя имена и адреса электронной почты. В примере использованы фиктивные адреса, но в реальном проекте рекомендуется указывать действующий.

PyPI требует, чтобы имя каждого проекта было уникальным. Поэтому лучше проверить доступность имени еще в начале работы над проектом, чтобы убедиться, что оно не занято. Кроме того, стоит избегать названий, легко путающихся с другими, чтобы случайно не установить чужой пакет.

Следующие поля таблицы `[project]` описывают сам проект:

```
# railway-project/pyproject.toml
[project]
...
description = "A CLI client for the railway API"
readme = "README.md"
```

Поле `description` должно содержать краткое описание проекта — обычно в одну строку. Поле `readme` указывает на файл с более подробным описанием.

Кроме того, readme можно задать в виде вложенной таблицы TOML с ключом content-type и либо file (с путем к README-файлу), либо text (с полным текстом README прямо в pyproject.toml).

Лицензия проекта тоже указывается в метаинформации:

```
# railway-project/pyproject.toml
[project]
...
license = {file = "LICENSE"}
```

Поле `license` задается в виде таблицы TOML и может содержать либо ключ `file` с путем к файлу лицензии, либо ключ `text` с полным текстом лицензии.

Следующие поля метаданных помогают потенциальным пользователям найти ваш проект на сайте PyPI:

```
# railway-project/pyproject.toml
[project]
...
classifiers = [
    "Environment :: Console",
    "License :: OSI Approved :: MIT License",
    "Operating System :: MacOS",
    "Operating System :: Microsoft :: Windows",
    "Operating System :: POSIX :: Linux",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3.10",
    "Programming Language :: Python :: 3.11",
    "Programming Language :: Python :: 3.12",
]
keywords = ["packaging example", "CLI"]
```

Поле `classifiers` используется для указания списка *классификаторов trove*, применяемых для категоризации проектов на PyPI. При поиске на сайте PyPI пользователи могут отфильтровать результаты по классификаторам. Все классификаторы должны быть выбраны из официального списка, доступного по адресу https://pypi.org/classifiers/.

В примере классификаторы указывают, что проект предназначен для использования в консоли, распространяется по лицензии MIT, работает в macOS, Windows и Linux, а также совместим с Python 3 (в частности, с версиями 3.10, 3.11 и 3.12). Классификаторы носят исключительно информационный характер и не влияют на то, где именно можно установить пакет.

Поле `keywords` служит для добавления произвольных ключевых слов, которые могут облегчить поиск проекта. В отличие от `classifiers` здесь нет ограничений — можно указать любые подходящие слова.

Версионирование и динамические метаданные

В метаданных проекта обязательно должно быть указано значение версии. Вместо того чтобы задавать его напрямую (через ключ `version`), мы сделали возможность задавать версию *динамически*. Спецификация `pyproject.toml` допускает, чтобы любые

метаданные проекта, кроме имени, задавались динамически с помощью сторонних инструментов. Названия таких динамических полей перечисляются в ключе `dynamic`:

```
# railway-project/pyproject.toml
[project]
...
dynamic = ["version"]
```

Чтобы использовать динамические метаданные с библиотекой `Setuptools`, нужно задать таблицу `tool.setuptools.dynamic`, в которой указывается, как получить значение. Версия может быть считана либо из файла (с помощью таблицы с ключом `file`), либо из атрибута модуля Python (с помощью таблицы с ключом `attr`). В этом проекте версия берется из атрибута `version` внутри импортируемого пакета `railway_cli`:

```
# railway-project/pyproject.toml
[tool.setuptools.dynamic]
version = {"attr" = "railway_cli.__version__"}
```

Атрибут `__version__` определен в файле `railway_cli/__init__.py`:

```
# railway-project/src/railway_cli/__init__.py
__version__ = "0.0.1"
```

Благодаря использованию динамического поля мы можем задавать номер версии в одном месте — и в коде, и в метаданных проекта, — не дублируя значение.

Вы можете выбрать любую схему версионирования, которая подходит для вашего проекта, но она должна соответствовать правилам, описанным в PEP 440 (https://peps.python.org/pep-0440/). Совместимая с PEP 440 версия состоит из последовательности чисел, разделенных точками, за которыми могут следовать обозначения предварительного, пост-релиза или релиза в стадии разработки. Предварительный релиз обозначается буквой `a` *(альфа)*, `b` *(бета)* или `rc` *(release candidate)*, за которыми следует число. Пост-релиз указывается с помощью слова `post` и числа. Релиз в разработке обозначается словом `dev` и числом. Версия без дополнительных индикаторов считается *финальной*. Например:

- `1.0.0.dev1` — первый релиз в разработке для версии 1.0.0;
- `1.0.0.a1` — первый альфа-релиз;
- `1.0.0.b1` — первый бета-релиз;
- `1.0.0.rc1` — первый кандидат в релизы;
- `1.0.0` — финальный релиз версии 1.0.0;
- `1.0.0.post1` — первый пост-релиз.

Релизы в разработке, предварительные, финальные и пост-релизы с одним и тем же основным номером версии упорядочиваются в том порядке, в котором перечислены выше.

Среди популярных схем версионирования можно выделить *семантическое версионирование*, которое позволяет судить о совместимости релизов по номеру

версии, а также *версионирование по дате*, при котором номер версии обычно отражает год и месяц выпуска.

Семантическое версионирование использует три числа: основную, второстепенную и корректирующую версии (major.minor.patch), разделенные точками. Если новая версия полностью совместима с предыдущей, то увеличивается только корректирующее число — такие релизы, как правило, содержат только исправления ошибок. Если в релиз добавлены новые функции без нарушения совместимости, то увеличивается второстепенное число. Основное увеличивается, когда релиз несовместим с предыдущими версиями. Полное описание семантического версионирования приведено на сайте https://semver.org.

Указание зависимостей

Как мы уже видели в начале главы, в дистрибутивный пакет можно добавить список проектов, от которых он зависит. При установке такого пакета с помощью `pip` соответствующие версии этих зависимостей будут установлены автоматически. Эти зависимости указываются в ключе `dependencies` таблицы `project`:

```
# railway-project/pyproject.toml
[project]
...
dependencies = [
    "pydantic[email]>=2.8.2,<3.0.0",
    "pydantic-settings~=2.4.0",
    "requests~=2.0",
]
```

Наш проект зависит от пакетов `pydantic`, `pydantic-settings` и `requests`. Квадратные скобки вокруг слова `email` после `pydantic` обозначают, что нам также нужны дополнительные (необязательные) зависимости, связанные с обработкой адресов электронной почты. Такие зависимости мы подробнее обсудим чуть позже.

Указывать нужные версии можно с помощью *спецификаторов версий*. Помимо обычных операторов сравнения, Python поддерживает оператор `~=` — он указывает на *совместимую версию* в рамках схемы семантического версионирования. Например, `requests~=2.0` означает, что подойдут любые версии `requests` 2.x, начиная с 2.0 и до (не включительно) 3.0. Кроме того, можно использовать список условий, разделенных запятыми: `pydantic>=2.8.2,<3.0.0` означает, что нас устраивает любая версия от 2.8.2 до 3.0.0 (не включительно). Это не то же самое, что `pydantic~=2.8.2`, поскольку в этом случае допустимыми будут версии до 2.9.0 (не включительно).

Подробно синтаксис зависимостей и правила сопоставления версий описаны в PEP 508 (https://peps.python.org/pep-0508/).

Не стоит задавать слишком жесткие ограничения на версии зависимостей. Помните, что ваш пакет, скорее всего, будет устанавливаться в среде, уже содержащей другие пакеты. Это особенно важно для библиотек и инструментов,

предназначенных для разработчиков. Чем шире вы укажете диапазон допустимых версий зависимостей, тем меньше вероятность, что у тех, кто использует ваш пакет, возникнут конфликты между вашими зависимостями и зависимостями других библиотек. Кроме того, если ограничения окажутся слишком строгими, то пользователи не смогут получать обновления и исправления безопасности в зависимостях, пока вы сами не выпустите новую версию с обновленным ограничением.

Помимо зависимостей от других проектов, можно указать, с какими версиями Python совместим ваш проект. В нашем случае используются функции, появившиеся в Python 3.10, поэтому мы указываем, что проект требует как минимум Python 3.10:

```
# railway-project/pyproject.toml
[project]
...
requires-python = ">=3.10"
```

Как и в случае с зависимостями, не стоит чрезмерно ограничивать поддерживаемые версии Python. Указывать минимальную версию языка следует только тогда, когда точно известно, что код не будет работать на других, все еще поддерживаемых версиях Python 3.

Актуальный список поддерживаемых версий Python доступен на официальной странице скачиваний: https://www.python.org/downloads/.

Важно убедиться, что ваш код действительно работает во всех версиях Python и с теми версиями зависимостей, которые вы указываете в конфигурации проекта. Один из подходов — создавать виртуальные окружения с разными версиями Python и зависимостей, а затем запускать тесты в каждом из них. Проверка вручную заняла бы слишком много времени, но существуют инструменты, которые автоматизируют этот процесс. Самый популярный из них — `tox`. Подробнее о нем можно узнать на сайте https://tox.wiki.

В вашем проекте можно также указать дополнительные (необязательные) зависимости. Система `pip` будет устанавливать такие зависимости только в том случае, если пользователь запросит их специально. Это удобно, когда определенные зависимости нужны лишь для функций, которые большинство пользователей, скорее всего, не будут задействовать. Пользователи, которым эти функции нужны, смогут установить дополнительные зависимости, а все остальные разработчики сэкономят место на диске и трафик.

Например, в проекте PyJWT, о котором шла речь в главе 9, используется библиотека `cryptography` для подписывания JWT с помощью асимметричных ключей. Данная возможность нужна далеко не всем пользователям, поэтому разработчики сделали `cryptography` дополнительной зависимостью.

Дополнительные зависимости (или *extra*) указываются в секции `project.optional-dependencies` файла `pyproject.toml`. В ней можно указать любое количество списков, каждый из которых описывает определенный набор дополнительных зависимостей. Эти списки называются *extras*. В нашем проекте определена одна extra-зависимость — с именем `dev`:

```
# railway-project/pyproject.toml
dev = [
    "black",
    "isort",
    "flake8",
    "mypy",
    "types-requests",
    "pytest",
    "pytest-mock",
    "requests-mock",
]
```

Такое имя часто используют для зависимостей, которые полезны в процессе разработки. Во многих проектах также создается extra-зависимость с именем `test` — она добавляет пакеты, которые требуются только для запуска тестов.

Чтобы установить проект с дополнительными зависимостями, нужно указать имена extra-зависимостей в квадратных скобках при вызове `pip install`. Например, чтобы установить наш проект в режиме редактирования вместе с зависимостями для разработки, можно выполнить команду:

```
$ pip install -e './railway-project[dev]'
```

Обратите внимание: кавычки в команде необходимы, чтобы оболочка не интерпретировала квадратные скобки как шаблон имени файла.

URL проекта

Помимо всего вышеперечисленного, вы можете указать список URL, связанных с вашим проектом, в подтаблице `urls` внутри таблицы `project` в файле `pyproject.toml`:

```
# railway-project/pyproject.toml
[project.urls]
Homepage = "https://github.com/PacktPublishing/Learn-Python-..."
"Learn Python Programming Book" = "https://www.packtpub.com/..."
```

Ключи таблицы `urls` — это произвольные строки, описывающие назначение каждой ссылки. Часто сюда добавляют ссылку на исходный код проекта, размещенный на GitHub, GitLab или другом хостинге. Кроме того, многие проекты добавляют сюда ссылки на документацию и трекеры ошибок.

В нашем примере мы использовали эту секцию, чтобы добавить ссылку на GitHub-репозиторий с исходным кодом к книге, а также на страницу книги на сайте издательства Packt.

Скрипты и точки входа

До сих пор мы запускали наше приложение, вводя такую команду:

```
$ python -m railway_cli
```

Это не слишком удобно. Было бы куда лучше, если бы приложение запускалось просто по команде:

```
$ railway-cli
```

Такое поведение можно настроить с помощью скриптовых точек входа в файле `pyproject.toml`. *Точка входа (entry point)* — это функция, которую нужно сделать доступной как команду в терминале (или графическое приложение). При установке пакета `pip` автоматически создает скрипты, которые импортируют указанные функции и запускают их.

Скриптовые точки входа настраиваются в таблице `project.scripts`:

```
# railway-project/pyproject.toml
[project.scripts]
railway-cli = "railway_cli.cli:main"
```

Каждый ключ в этой таблице задает имя скрипта, который должен быть создан при установке пакета. Значение указывает на функцию, которую нужно вызвать при запуске скрипта (в формате `модуль:функция`). Если бы мы упаковывали графическое приложение, то вместо `project.scripts` использовалась бы таблица `project.gui-scripts`.

Операционная система Windows различает консольные и графические приложения. Консольные запускаются в окне терминала и могут выводить текст или принимать ввод с клавиатуры через консоль. Графические приложения запускаются без консольного окна. В других операционных системах такого различия нет — скрипты работают одинаково.

После установки проекта в виртуальное окружение `pip` создаст скрипт `railway-cli` в папке `bin` виртуального окружения (или в папке `Scripts` на Windows). Этот скрипт будет выглядеть примерно так:

```
#!/.../ch16/railway-project/venv/bin/python
# -*- coding: utf-8 -*-
import re
import sys
from railway_cli.cli import main
if __name__ == '__main__':
    sys.argv[0] = re.sub(
        r'(-script\.pyw|\.exe)?$', '', sys.argv[0]
    )
    sys.exit(main())
```

Комментарий `#!/.../ch16/railway-project/venv/bin/python` в начале файла называется *шебангом* (от слов *hash* и *bang*; bang является альтернативным названием восклицательного знака). Шебанг указывает путь к исполняемому файлу Python, который будет использоваться для запуска скрипта. Последний импортирует функцию `main()` из модуля `railway_cli.main`, немного изменяет значение `sys.argv[0]`, а затем вызывает `main()` и передает результат в `sys.exit()`.

Помимо скриптовых точек входа, можно определять и произвольные группы точек входа. Такие группы описываются в подтаблицах таблицы project.entry-points. Для точек входа из этих групп pip не создает исполняемые скрипты, однако они полезны в других сценариях. Например, во многих проектах с плагинами точки входа используются для обнаружения и подключения расширений. Это уже более сложная тема, которую мы не будем разбирать подробно. Если вы хотите получить больше информации, то обратитесь к спецификации точек входа (https://packaging.python.org/specifications/entry-points/).

Определение содержимого пакета

Для проектов, использующих структуру src, библиотека `setuptools` обычно может автоматически определить, какие модули Python следует добавить в дистрибутив. В проектах с плоской структурой автоматическое обнаружение работает только в том случае, если в корневой папке проекта содержится ровно один импортируемый пакет. Если же в проекте есть другие файлы Python, то придется явно указать, какие пакеты и модули следует добавлять или исключать, используя таблицу `tool.setuptools` в файле `pyproject.toml`.

Для структуры src дополнительная конфигурация потребуется только в том случае, если папка с исходным кодом называется не `src` или если в папке `src` находятся модули, которые не нужно добавлять в дистрибутив. В проекте `railway_cli` мы полагаемся на автоматическое обнаружение, поэтому в нашем файле `pyproject.toml` нет конфигурации, связанной с обнаружением пакетов. Подробную информацию об автоматическом обнаружении пакетов `Setuptools` и способах его настройки можно найти в руководстве пользователя по адресу https://setuptools.pypa.io/en/latest/userguide/.

Конфигурация метаданных нашего пакета завершена. Прежде чем перейти к его сборке и публикации, кратко рассмотрим, как получить доступ к метаданным прямо в коде.

Доступ к метаданным в коде

Ранее мы уже использовали динамические метаданные, чтобы общее значение версии использовалось и в конфигурации дистрибутива, и в самом коде. Однако для других метаданных библиотека `setuptools` поддерживает загрузку только из файлов, а не из атрибутов в коде. Такие файлы не будут добавлены

в wheel-дистрибутив, если явно не указать это в конфигурации. К счастью, существует более удобный способ доступа к метаданным дистрибутива из кода.

Модуль стандартной библиотеки `importlib.metadata` предоставляет интерфейсы для доступа к метаданным любого установленного пакета. Чтобы продемонстрировать это, мы добавили в приложение `railway_cli` параметр командной строки для вывода лицензии проекта:

```
from importlib.metadata import metadata, packages_distributions

def get_arg_parser() -> argparse.ArgumentParser:
    ...
    parser.add_argument(
        "-L",
        "--license",
        action="version",
        version=get_license(),
    )
    ...

def get_license() -> str:
    default = "License not found"

    all_distributions = packages_distributions()
    try:
        distribution = all_distributions[__package__][0]
    except KeyError:
        return default

    meta = metadata(distribution)
    return meta["License"] or default
```

Мы используем действие парсера аргументов `"version"`, чтобы вывести текст лицензии и завершить выполнение, если в командной строке есть параметр `-L` или `--license`. Чтобы получить текст лицензии, сначала нужно определить дистрибутив, соответствующий нашему импортируемому пакету. Функция `packages_distributions()` возвращает словарь, в котором ключами являются имена импортируемых пакетов в виртуальном окружении, а значениями — списки дистрибутивов, предоставляющих эти пакеты. Мы предполагаем, что ни один другой установленный дистрибутив не содержит пакет с тем же именем, что и наш, поэтому просто берем первый элемент из списка `all_distributions[__package__]`.

Функция `metadata()` возвращает объект, похожий на словарь, содержащий метаданные. Ключи в этом объекте частично совпадают с названиями полей в файле `pyproject.toml`. Полный список допустимых ключей и их значения приведены в спецификации метаданных на сайте Python Packaging Authority (https://packaging.python.org/specifications/core-metadata/).

Если наш пакет не установлен, то при попытке получить его из словаря, возвращаемого функцией `packages_distributions()`, будет выброшено исключение `KeyError`. В этом случае, а также если в метаданных проекта не указана лицензия

(отсутствует ключ "License"), мы возвращаем значение по умолчанию, чтобы показать, что лицензию найти не удалось.

Посмотрим, как выглядит вывод:

```
$ railway-cli -L
Copyright (c) 2024 Heinrich Kruger, Fabrizio Romano

Permission is hereby granted, free of charge, ...
```

В данном примере мы сократили вывод, чтобы сэкономить место на странице, но если вы выполните эту команду у себя, то будет выведен полный текст лицензии.

Теперь можно переходить к созданию и публикации дистрибутивов проекта.

Создание и публикация пакетов

Создавать дистрибутив пакета мы будем с помощью инструмента `build`, который можно найти на сайте PyPI (https://pypi.org/project/build/). Для загрузки пакетов на PyPI потребуется также утилита `twine` (https://pypi.org/project/twine/). Обе программы можно установить, воспользовавшись файлом `requirements/build.txt`, который прилагается к исходному коду, сопровождающему эту главу. Рекомендуется устанавливать их в новое виртуальное окружение.

Имена проектов на PyPI должны быть уникальными, поэтому загрузить проект railway-cli не получится, если только вы не измените его имя. Измените его в файле pyproject.toml на уникальное значение, прежде чем создавать дистрибутив. Имейте в виду, что имена файлов дистрибутива также изменятся.

Сборка

Проект `build` предоставляет простой сценарий для создания пакетов в соответствии со спецификацией PEP 517. Он берет на себя все технические детали, связанные со сборкой дистрибутивов. При запуске `build` выполняет следующие шаги.

1. Создает виртуальное окружение.
2. Устанавливает в него зависимости сборки, указанные в файле `pyproject.toml`.
3. Импортирует указанный в `pyproject.toml` механизм сборки и использует его для создания исходного дистрибутива.
4. Создает другое виртуальное окружение и снова устанавливает зависимости сборки.
5. Импортирует механизм сборки и использует его для создания wheel-пакета на основе исходного дистрибутива, полученного на шаге 3.

Чтобы увидеть все это в действии, перейдите в папку `railway-project` из исходного кода к данной главе и выполните следующую команду:

```
$ python -m build
* Creating isolated environment: venv+pip...
* Installing packages in isolated environment:
  - setuptools>=66.1.0
  - wheel
* Getting build dependencies for sdist...
...
* Building sdist...
...
* Building wheel from sdist
* Creating isolated environment: venv+pip...
* Installing packages in isolated environment:
  - setuptools>=66.1.0
  - wheel
* Getting build dependencies for wheel...
...
* Building wheel...
...
Successfully built railway_cli-0.0.1.tar.gz and
railway_cli-0.0.1-py3-none-any.whl
```

Мы опустили множество строк вывода, чтобы было проще сопоставить происходящее с описанными выше шагами. После выполнения команды в папке `railway-project` появится новая папка `dist`, в которой будут находиться два файла: `railway_cli-0.0.1.tar.gz` (исходный дистрибутив) и `railway_cli-0.0.1-py3-none-any.whl` (wheel-пакет).

Перед публикацией пакета рекомендуется выполнить несколько проверок, чтобы убедиться, что сборка прошла корректно. Сначала с помощью `twine` можно проверить, корректно ли будет отображаться файл README на сайте PyPI:

```
$ twine check dist/*
Checking dist/railway_cli-0.0.1-py3-none-any.whl: PASSED
Checking dist/railway_cli-0.0.1.tar.gz: PASSED
```

Если `twine` сообщит о проблемах, то нужно исправить их и собрать пакет повторно. В нашем случае проверка прошла успешно, поэтому можно установить wheel-пакет в отдельное виртуальное окружение и протестировать:

```
$ pip install dist./railway_cli-0.0.1-py3-none-any.whl
```

Убедитесь, что запускаете приложение за пределами каталога проекта. Если при установке или запуске возникнут ошибки, то внимательно проверьте конфигурацию на наличие опечаток.

Поскольку сборка прошла успешно, то можно переходить к публикации пакета.

Публикация

Наш проект учебный, поэтому мы будем публиковать его не в настоящем хранилище PyPI, а в TestPyPI — это отдельный экземпляр индексного репозитория, созданный специально для тестирования загрузки пакетов и экспериментов с инструментами и процессами упаковки.

Прежде чем загружать пакеты, нужно зарегистрировать учетную запись. Перейдите на сайт TestPyPI https://test.pypi.org и нажмите кнопку Register (Зарегистрироваться). После завершения регистрации и подтверждения адреса электронной почты необходимо сгенерировать токен API. Это можно сделать на странице Account Settings (Настройки аккаунта). Скопируйте токен и сохраните его до закрытия страницы. Затем создайте файл `.pypirc` в домашней папке пользователя. Его содержимое должно выглядеть так:

```
[testpypi]
username = __token__
password = pypi-...
```

Замените значение `pypi-...` на свое фактическое значение токена, полученного на сайте.

Настоятельно рекомендуется включить двухфакторную аутентификацию для учетной записи на TestPyPI и особенно для основной учетной записи на PyPI.

Теперь можно использовать `twine` для загрузки дистрибутивов:

```
$ twine upload --repository testpypi dist/*
Uploading distributions to https://test.pypi.org/legacy/
Uploading railway_cli-0.0.1-py3-none-any.whl
100% ━━━━━━━━━━━━━━━━━━━━ 19.3/19.3 kB • 00:00 • 7.3 MB/s
Uploading railway_cli-0.0.1.tar.gz
100% ━━━━━━━━━━━━━━━━━━━━ 17.7/17.7 kB • 00:00 • 9.4 MB/s

View at:
https://test.pypi.org/project/railway-cli/0.0.1/
```

`twine` выдает индикаторы прогресса, показывающие, как идет загрузка. После завершения загрузки он выводит URL, по которому можно посмотреть сведения о вашем пакете. Откройте его в браузере. Вы увидите описание проекта, сформированное на основе содержимого файла `README.md`. Слева на странице находятся ссылки на URL проекта, сведения об авторе, лицензия, ключевые слова и классификаторы. На рис. 16.1 показано, как выглядит эта страница для проекта `railway-cli`.

Внимательно проверьте всю информацию на странице и убедитесь, что она соответствует ожидаемой. Если что-то указано неправильно, то нужно будет исправить метаданные в файле `pyproject.toml`, пересобрать пакет и загрузить его заново.

Рис. 16.1. Страница нашего проекта на сайте TestPyPI

PyPI не позволяет повторно загружать дистрибутивы с теми же именами файлов, которые уже были размещены. Чтобы изменить метаданные, необходимо увеличить версию пакета. Пока вы не убедитесь, что все указано правильно, имеет смысл использовать номера предварительных версий. Это поможет избежать ненужного увеличения номера версии из-за мелких ошибок в упаковке.

Теперь можно установить пакет из репозитория TestPyPI. Выполните следующую команду в новом виртуальном окружении:

```
pip install --index-url https://test.pypi.org/simple/ \
--extra-index-url https://pypi.org/simple/ railway-cli==1.0.0
```

Параметр `--index-url` указывает `pip`, что основным индексом пакетов нужно считать `https://test.pypi.org/simple/`. Мы также добавляем `--extra-index-url https://pypi.org/simple/`, чтобы `pip` мог находить зависимости в обычном репозитории PyPI — на случай, если нужные зависимости отсутствуют в TestPyPI. Пакет был установлен успешно, а значит, он корректно собран и загружен.

Если бы это был настоящий проект, то следующим шагом стала бы публикация в основном репозитории PyPI. Процедура точно такая же, как и для TestPyPI. При сохранении API-ключа PyPI его нужно добавить в существующий файл `.pypirc` под заголовком `[pypi]`, примерно так:

```
[pypi]
  username = __token__
  password = pypi-…
```

Кроме того, нет необходимости указывать `--repository`, чтобы загрузить пакет в основной PyPI. Достаточно выполнить следующую команду:

```
$ twine upload dist/*
```

Как видите, упаковка и публикация проекта — не слишком сложный процесс, но есть немало нюансов, которым следует уделить внимание. Хорошая новость в том, что бóльшая часть этой работы выполняется только раз — при публикации первой версии. В последующих релизах, как правило, достаточно обновить номер версии и, возможно, изменить список зависимостей. В следующем разделе вы найдете несколько советов, которые помогут упростить процесс.

Советы по запуску новых проектов

Подготовка проекта к упаковке может показаться утомительной, особенно если делать все сразу. Легко допустить ошибку — например, забыть указать зависимость, если пытаться описать конфигурацию пакета непосредственно перед первой публикацией. Намного удобнее начать с простого файла `pyproject.toml`, в котором указаны только базовая информация и минимальная конфигурация. По мере развития проекта вы будете постепенно добавлять в него новые данные.

Например, каждый раз, когда в коде появляется новая сторонняя библиотека, имеет смысл сразу же добавлять ее в список зависимостей. Кроме того, полезно заранее начать писать файл README и постепенно его дополнять. Иногда бывает, что несколько абзацев с описанием проекта помогают лучше сформулировать цели и общее направление работы.

Чтобы помочь вам, мы подготовили шаблон проекта. Он находится в папке `skeleton-project` в исходном коде к этой главе:

```
$ tree skeleton-project
skeleton-project
├── README.md
├── pyproject.toml
├── src
│   └── example
│       └── __init__.py
└── tests
    └── __init__.py
```

Скопируйте его, адаптируйте под свои нужды и используйте как отправную точку для собственного проекта.

Кроме того, существует проект cookiecutter (https://cookiecutter.readthedocs.io), который позволяет создавать шаблоны для старта новых проектов. Благодаря этому можно значительно упростить начальную настройку.

Другие файлы

Конфигурационных файлов, рассмотренных в этой главе, как правило, вполне хватает для упаковки и распространения большинства современных Python-проектов. Однако при изучении других проектов на PyPI вы можете столкнуться и с другими файлами.

- На ранних этапах развития библиотеки `setuptools` каждому проекту требовался сценарий `setup.py`, с помощью которого происходила сборка проекта. Обычно он содержал только вызов функции `setuptools.setup()` с метаданными проекта в виде аргументов. Использование `setup.py` не устарело — это по-прежнему допустимый способ конфигурации `Setuptools`.
- `Setuptools` может считывать настройки из файла `setup.cfg`. До того момента, пока `pyproject.toml` не получил широкое распространение, именно использование `setup.cfg` считалось предпочтительным способом конфигурации.
- Если вы хотите добавить в дистрибутив нестандартные файлы, например дополнительные данные, то потребуется файл `MANIFEST.in`, в котором указываются такие файлы. Подробнее об этом можно прочитать на сайте https://packaging.python.org/guides/using-manifest-in/.

Альтернативные инструменты

В завершение главы кратко рассмотрим альтернативные варианты упаковки проектов Python. До появления стандартов PEP 517 и PEP 518 практически невозможно было использовать что-либо кроме `Setuptools`. Не существовало способа задать, какие библиотеки требуются для сборки проекта или как именно он должен собираться, поэтому `pip` и другие инструменты по умолчанию использовали `Setuptools`.

Благодаря информации о системе сборки, задаваемой в файле `pyproject.toml`, теперь легко использовать любую библиотеку упаковки. Доступно несколько популярных альтернатив[1].

---

[1] В 2025 году самым популярным инструментом является uv (https://docs.astral.sh/uv/) — чрезвычайно быстрый пакетный менеджер Python, написанный на Rust (от создателей ruff). Он ориентирован на замену таких средств, как pip, pip-tools, pipx, poetry, pyenv, twine, virtualenv и др. Поддерживает бэкенд сборки для проектов на чистом Python по стандарту PEP 517 с возможностью использования сторонних бэкендов для сборки. Поддерживает метаданные проекта по стандарту PEP 621. Отличительной чертой является управление версиями Python и публикация пакетов на PyPi «из коробки». — *Примеч. науч. ред.*

- Flit (https://flit.pypa.io) вдохновил разработку стандартов PEP 517 и PEP 518 (его автор был соавтором PEP 517). Flit ориентирован на простые проекты на чистом Python, не требующие сложных этапов сборки (например, компиляции кода C). В комплект входит CLI-инструмент для сборки пакета и загрузки на PyPI, так что вам не понадобятся `build` и `twine`.
- Poetry (https://python-poetry.org) тоже предоставляет CLI для сборки и публикации пакетов, а кроме того — облегченный бэкенд сборки, совместимый с PEP 517. Главным преимуществом Poetry считается расширенная система управления зависимостями. Он может даже автоматически управлять виртуальными окружениями.
- Hatch (https://hatch.pypa.io) — расширяемый инструмент управления проектами Python. Он содержит средства для установки и управления версиями Python, работы с виртуальными окружениями, запуска тестов и многого другого. В него также входит hatchling — бэкенд сборки по стандарту PEP 517.
- PDM (https://pdm-project.org) — еще один менеджер пакетов и зависимостей, содержащий бэкенд сборки. Подобно Hatch, он позволяет управлять виртуальными окружениями и устанавливать версии Python. Для PDM доступно множество плагинов, расширяющих его функциональность.
- Enscons (https://dholth.github.io/enscons/) основан на универсальной системе сборки SCons (https://scons.org/). В отличие от Flit и Poetry enscons можно использовать для сборки дистрибутивов с расширениями на языке C.
- Meson (https://mesonbuild.com) — еще одна популярная универсальная система сборки. Проект meson-python (https://mesonbuild.com/meson-python/) предоставляет бэкенд сборки по стандарту PEP 517 на основе Meson. С его помощью можно собирать дистрибутивы, содержащие расширения, скомпилированные с использованием Meson.
- Maturin (https://www.maturin.rs) — инструмент сборки, предназначенный для создания дистрибутивов с расширениями, написанными на языке Rust.

Все инструменты, рассмотренные в этой главе, ориентированы на распространение пакетов через PyPI. Однако в зависимости от целевой аудитории такой способ может подойти не всегда. PyPI в первую очередь предназначен для публикации библиотек и инструментов разработки, которыми пользуются разработчики на Python. Чтобы установить и использовать пакеты с PyPI, требуется рабочая установка Python и хотя бы минимальные знания о том, как использовать `pip`.

Если ваш проект — это приложение, рассчитанное на пользователей, обладающих малым объемом технических знаний, то стоит рассмотреть другие способы распространения. На сайте Python Packaging User Guide есть полезный обзор

различных подходов к дистрибуции приложений (https://packaging.python.org/overview/#packaging-applications).

На этом наше путешествие по упаковке проектов подходит к концу.

Дополнительные материалы

Завершим главу подборкой полезных ресурсов, содержащих более подробную информацию по теме упаковки проектов Python.

- Страница об истории упаковки, расположенная на сайте Python Packaging Authority (https://www.pypa.io/en/latest/history/), поможет узнать, как развивались инструменты упаковки в Python.
- Руководство по упаковке Python (https://packaging.python.org) содержит полезные учебные материалы, глоссарий, ссылки на спецификации упаковки и обзор интересных проектов, связанных с этой темой.
- Документация по `Setuptools` (https://setuptools.readthedocs.io) — еще один важный источник подробной информации.
- Если вы распространяете библиотеку с поддержкой аннотаций типов, то имеет смысл добавить в пакет типовую информацию, чтобы разработчики могли выполнять проверку типов в своем коде. Подробнее об этом — в документации по аннотациям типов на сайте https://typing.readthedocs.io/en/latest/spec/distributing.html.
- В этой главе мы показали, как получить доступ к метаданным дистрибутива в коде. Кроме того, большинство инструментов упаковки позволяют добавлять в дистрибутив дополнительные данные. Для доступа к таким *ресурсам пакета* можно использовать модуль стандартной библиотеки `importlib.resources`. Подробнее об этом — на странице https://docs.python.org/3/library/importlib.resources.html.

При чтении этих (и других) материалов стоит помнить: стандарты PEP 517, PEP 518 и PEP 621 были утверждены несколько лет назад, однако многие проекты до сих пор не перешли полностью на сборку по PEP 517 и не используют `pyproject.toml` для хранения метаданных. Кроме того, в значительной части документации до сих пор описываются устаревшие подходы.

Резюме

В этой главе мы разбирали, как упаковывать и распространять проекты на Python через PyPI. Сначала мы рассмотрели теоретическую часть — понятия проекта, релиза и дистрибутива на платформе PyPI.

Вы изучили `Setuptools` — самую популярную библиотеку для упаковки проектов на Python — и пошагово прошли процесс подготовки проекта, выполняемый

с ее помощью. В ходе работы вы увидели, какие файлы нужно добавить в проект и какую роль играет каждый из них.

Далее мы обсудили, какие метаданные стоит указывать, чтобы описать проект и упростить его поиск на PyPI. Кроме того, мы выяснили, как добавлять в дистрибутив код, как задавать зависимости и определять точки входа, чтобы `pip` мог автоматически создавать исполняемые скрипты. Вдобавок мы рассмотрели способы обращения к метаданным дистрибутива прямо из кода.

Затем речь зашла о сборке дистрибутивов и загрузке их на PyPI с помощью `twine`. Кроме того, мы поделились советами по началу новых проектов, а в завершение кратко рассказали об альтернативах `Setuptools` и привели список полезных ресурсов по теме упаковки.

Мы рекомендуем вам попробовать опубликовать свой код на PyPI. Даже если проект кажется вам незначительным, где-то в мире обязательно найдется человек, которому он пригодится. Это по-настоящему вдохновляет — вносить вклад в сообщество, и к тому же упоминание такого проекта отлично смотрится в резюме.

В следующей (финальной) главе мы немного отойдем от профессиональной разработки и заглянем в мир соревновательного программирования. Вы узнаете, почему задачки на скорость могут быть не только веселым хобби, но и полезным опытом.

17 Задачи по программированию

> Опасайтесь багов в приведенном выше коде; я только доказал корректность, но не запускал его.
>
> *Дональд Кнут*

Задачи по программированию — это задачи, которые можно решить с помощью относительно короткой программы. Существует множество сайтов, предлагающих отличные подборки таких задач. Как правило, они классифицируют задачи по уровню сложности и другим параметрам, таким как:

- тип алгоритма, необходимого для решения;
- тип абстрактных структур данных (например, деревья, связанные списки, *n*-мерные векторы);
- тип конкретных структур данных (например, списки, кортежи, словари в Python);
- подход к решению (например, динамическое программирование, рекурсия).

Этот список далеко не полный, но дает общее представление о том, чего можно ожидать.

Разные сайты публикуют задачи на программирование по разным причинам. Одни помогают программистам готовиться к собеседованиям, другие делают это ради удовольствия, третьи стремятся научить программированию.

Вдобавок некоторые из этих сайтов проводят соревнования, в которых решения участников оцениваются по скорости выполнения, корректности, объему используемой памяти и другим критериям.

Мир соревновательного программирования действительно захватывающий, а решение задач — отличный способ выучить новый язык и улучшить навыки программирования.

В этой главе мы решим две задачи с сайта *Advent of Code* (https://adventofcode.com) — это наш любимый ресурс с задачами. Решение таких задач не только увлекает, но и позволяет увидеть возможности Python на примере компактных и выразительных программ.

Проект Advent of Code

Этот проект был создан Эриком Уэстлом в 2015 году. На сайте он описан так:

> «Advent of Code — это адвент-календарь с небольшими задачами по программированию, рассчитанными на разные уровни подготовки и навыки. Их можно решать на любом языке программирования. Люди используют их как подготовку к собеседованиям, для обучения в компаниях, в рамках университетских курсов, для практики, соревнований на скорость или просто чтобы бросить вызов друг другу.
>
> Для участия не требуется специальное образование в области компьютерных наук — достаточно базовых знаний программирования и умения решать задачи. Не нужен и мощный компьютер — все задачи можно решить менее чем за 15 секунд даже на оборудовании десятилетней давности».

Каждая задача состоит из двух частей. По словам Эрика, первая часть помогает убедиться, что вы поняли условие, а вторая и есть настоящая задача. Обычно вторая часть сложнее первой. К каждой задаче прилагаются входные данные, которые можно скачать с сайта.

У этого сайта есть особенности. Для начала, каждая задача вплетена в сюжет, в котором вы, решая задачи, помогаете Санте или эльфам спасти Рождество. Таким образом, каждое задание становится частью приключения, которое длится 25 дней: с 1 по 25 декабря. Материал всегда подается с юмором, каждая задача уникальна и требует как логического, так и творческого подхода. Это выделяет Advent of Code среди других подобных сайтов, на которых задачи обычно подаются в более сухом и академичном стиле, а решения зачастую сводятся к применению стандартных алгоритмов, структур данных или шаблонов.

Прежде чем добавить эту главу в книгу, мы связались с Эриком Уэстлом. Его единственным пожеланием было не воспроизводить задачи полностью дословно. Мы уважаем его позицию и потому представим лишь краткое описание сути задач — только их инструкции.

Решения, которые мы покажем в этой главе, представляют собой лишь один из возможных способов решения. Они выбраны так, чтобы мы могли обсудить с вами несколько заключительных концепций. Наш совет: после прочтения данной главы попробуйте разработать собственные решения этих задач.

Camel Cards

Первая задача называется *Camel Cards* и относится к седьмому дню Advent of Code 2023. Ее полное описание можно найти по адресу https://adventofcode.com/2023/day/7. Нужно написать программу, которая решает вариант карточной игры, похожей на покер.

Часть первая — условие задачи

Ниже приведена сокращенная версия оригинального текста.

В игре Camel Cards вы получаете список раздач, и ваша задача — упорядочить их по силе. Каждая раздача состоит из пяти карт с номиналами `A`, `K`, `Q`, `J`, `T`, `9`, `8`, `7`, `6`, `5`, `4`, `3` или `2`. Старшинство карт — от `A` (самая сильная) до `2` (самая слабая).

Каждая раздача относится ровно к одному типу. Типы перечислены ниже — от самого сильного к самому слабому.

- *Пятерка одного номинала* (five of a kind): все пять карт одинаковые — `AAAAA`.
- *Каре* (four of a kind): четыре карты одного номинала, пятая другого — `AA8AA`.
- *Фул-хаус* (full house): три карты одного номинала и две другого — `23332`.
- *Тройка* (three of a kind): три одинаковые карты и две разные — `TTT98`.
- *Две пары* (two pair): две карты одного номинала, две другого, одна третьего — `23432`.
- *Одна пара* (one pair): две одинаковые карты, три оставшиеся все разные — `A23A4`.
- *Старшая карта* (high card): все карты разного номинала — `23456`.

Раздачи в первую очередь упорядочиваются по типу. Например, любой фул-хаус сильнее любой тройки.

Если две раздачи относятся к одинаковому типу, то применяется дополнительное правило: карты сравниваются по порядку. Сначала сравниваются первые. Если они разные, то выигрывает та, где карта сильнее. Если первые одинаковые — сравниваются вторые и так далее до пятой карты.

Например, `33332` и `2AAAA` — обе раздачи типа каре, но `33332` сильнее, так как первая карта `3` старше, чем `2`. `77888` и `77788` — оба фул-хауса, но `77888` сильнее, так как третья карта `8` старше, чем `7`.

Чтобы сыграть в Camel Cards, вы получаете список раздач и соответствующих им ставок (это и есть входные данные задачи). Например:

```
32T3K 765
T55J5 684
KK677 28
KTJJT 220
QQQJA 483
```

В этом примере даны пять раздач. После каждой указывается ставка. Каждая раздача выигрывает сумму, равную произведению ставки на ранг раздачи: са-

мая слабая получает ранг 1, следующая — ранг 2, и так далее, до самой сильной, которая получает ранг 5. Поскольку в примере пять раздач, самая сильная будет умножаться на 5.

Итак, сначала упорядочим раздачи по силе.

- 32T3K — единственная с одной парой, а остальные — более сильного типа, поэтому она получает ранг 1.
- KK677 и KTJJT — обе с двумя парами. Первая карта у обеих — K, но вторая у KK677 — тоже K, а у KTJJT — T. Поскольку K сильнее, KTJJT получает ранг 2, а KK677 — 3.
- T55J5 и QQQJA — обе с тройкой. У QQQJA первая карта — Q, у T55J5 — T, поэтому QQQJA получает ранг 5, а T55J5 — 4.
- Теперь можно подсчитать общий выигрыш, перемножив ставки на ранги и сложив результат: 765 * 1 + 220 * 2 + 28 * 3 + 684 * 4 + 483 * 5 = 6440.

Задача: упорядочите все раздачи в вашем наборе и найдите общий выигрыш, следуя этим правилам.

Часть первая — решение

Реализация решения достаточно прямолинейна. Вторая часть задачи во многом похожа на первую, поэтому общая логика вынесена в базовый класс Solver. В нем реализованы все необходимые методы, кроме одного — type(), который определяется в подклассах PartOne и PartTwo. Такой подход — лишь один из способов избежать дублирования кода. Рассмотрим первую часть кода:

```
# day7.py
from collections import Counter
from functools import cmp_to_key
from util import get_input

class Solver:
    strengths: str = ""

    def __init__(self) -> None:
        self.ins: list[str] = get_input("input7.txt")

    def solve(self) -> int:
        hands = dict(self.parse_line(line) for line in self.ins)
        sorted_hands = sorted(
            hands.keys(), key=cmp_to_key(self.cmp)
        )
        return sum(
            rank * hands[hand]
            for rank, hand in enumerate(sorted_hands, start=1)
        )

    def parse_line(self, line: str) -> tuple[str, int]:
        hand, bid = line.split()
        return hand, int(bid)
```

Мы импортируем `Counter` и `cmp_to_key()` из стандартной библиотеки, а также `get_input()` из модуля `util.py`. Последняя функция — вспомогательная: она читает входной файл и возвращает список строк, например: `["9A35J 469", "75T32 237", ...]`

Далее определен класс `Solver` (приведен не полностью), который в конструкторе читает входные данные и содержит метод `solve()`, запускающий алгоритм. Этот алгоритм состоит из следующих шагов.

С помощью метода `parse_line()` мы преобразуем строки в словарь `hands`, где ключи — это сами раздачи, а значения — соответствующие ставки, уже приведенные к типу `int`. Затем создаем список раздач, отсортированных по их силе, как указано в условии задачи. Для этого используется собственная функция-компаратор `cmp()` — мы рассмотрим ее чуть позже. Пока обратите внимание, как она применяется для сортировки: поскольку `cmp()` принимает два аргумента, ее нельзя напрямую передать в параметр `key` функции `sorted()`. Поэтому Python предоставляет функцию `cmp_to_key()`, которая преобразует компаратор в объект, подходящий для сортировки по ключу.

Получив отсортированный список раздач, возвращаем сумму произведений: для каждой раздачи — ее ранг, умноженный на ставку.

Теперь разберем наиболее интересную часть класса:

```
# day7.py
…
class Solver:
    …
    def type(self, hand: str) -> list[int]:
        raise NotImplementedError

    def cmp(self, hand1: str, hand2: str) -> int:
        """-1 if hand1 < hand2 else 1, or 0 if hand1 == hand2"""
        type1 = self.type(hand1)
        type2 = self.type(hand2)
        if type1 == type2:
            for card1, card2 in zip(hand1, hand2):
                strength1 = self.strengths.index(card1)
                strength2 = self.strengths.index(card2)
                if strength1 == strength2:
                    continue
                return -1 if strength1 < strength2 else 1
            return 0
        return -1 if type1 < type2 else 1
```

В оставшейся части класса реализована функция-компаратор `cmp()`. Она принимает две раздачи и сравнивает их в соответствии с правилами из условия задачи. Сначала с помощью метода `type()` вычисляется тип каждой раздачи. При разных типах возвращается `-1`, если вторая раздача сильнее первой, или `1` в противоположном случае. Если же типы одинаковые, то сравнивается сила карт по позициям — от первой до пятой. Результат сравнения используется по тем же правилам: `-1`, если вторая раздача сильнее, `1` — если первая сильнее. Для полноты картины предусмотрен и вариант, когда типы и сила всех карт совпадают, — в этом

случае возвращается `0`. Однако, как указано в условии, такой сценарий невозможен: при одинаковых раздачах итоговый порядок мог бы зависеть от порядка входных данных, чего задача не допускает.

Компаратор использует метод `type()`, логика которого в базовом классе не реализована. Так сделано потому, что сила карт и способ определения типа раздачи — это как раз то, что меняется между первой и второй частями задачи. Поэтому данные элементы реализованы в двух отдельных классах-наследниках.

Теперь посмотрим реализацию класса `PartOne`:

```
# day7.py
…
class PartOne(Solver):
    strengths: str = "23456789TJQKA"

    def type(self, hand: str) -> list[int]:
        return [count for _, count in Counter(hand).most_common()]
```

В PartOne мы устанавливаем атрибут класса strengths и реализуем метод type(). При вызове этого метода на нескольких примерах рук получаем такие результаты:

```
type("KK444") = [3, 2]
type("QQQ6Q") = [4, 1]
type("5JA22") = [2, 1, 1, 1]
type("7A264") = [1, 1, 1, 1, 1]
type("TTKTT") = [4, 1]
```

Вот как работает метод `type()`: сначала раздача передается в объект `Counter`. Он подсчитывает количество вхождений каждого символа в строке. В первом примере вызов `Counter("KK444")` вернет объект, *похожий на словарь*: `{'K': 2, '4': 3}` — как и ожидалось, две карты `K` и три карты `4`. С помощью метода `most_common()` мы получаем список количеств, отсортированный по убыванию. В данном случае — `[3, 2]`. Подобные списки — `[1, 1, 1, 1, 1]`, `[3, 2]`, `[4, 1]` и т. д. — можно напрямую сравнивать между собой при вычислении силы руки (в покере под рукой понимается набор карт, а сила руки — это сила комбинации). Это и делается при формировании `sorted_hands` в методе `solve()`.

Теперь можно создать экземпляр класса `PartOne` и вызвать у него метод `solve()`:

```
# day7.py
part_one = PartOne()
print(part_one.solve())
```

Обратите внимание, что в `PartOne.strengths` задается строка `"23456789TJQKA"` — это порядок старшинства карт согласно условию: `2` — самая слабая, `A` — самая сильная. Мы используем позицию символа в этой строке как его «вес» — благодаря этому можно легко сравнивать карты, просто обращаясь к их индексу в строке. Чем выше индекс, тем сильнее карта.

На этом первая часть задачи завершена, переходим ко второй.

Часть вторая — условие задачи

После того как мы ввели на сайте правильный ответ на первую часть, открывается вторая часть задачи. В ней правила немного меняются — в игру вводится карта-джокер.

Теперь карты `J` считаются джокерами — они могут заменять любую карту, которая усилит руку, делая ее максимально сильной. В то же время при сравнении отдельных карт джокеры считаются самыми слабыми — даже слабее, чем `2`. Остальной порядок карт остается прежним: `A`, `K`, `Q`, `T`, `9`, `8`, `7`, `6`, `5`, `4`, `3`, `2`, `J`.

Карты `J` могут притворяться любой картой — так, чтобы усилить тип руки. Например, раздача `QJJQ2` теперь считается четверкой одного номинала. Однако при сравнении карт одинакового типа для разрешения ничьей `J` остается `J`, то есть не считается той картой, которую она имитирует. Например, `JKKK2` считается слабее `QQQQ2`, поскольку `J` слабее `Q`.

Теперь уже знакомый нам пример разыгрывается иначе:

```
32T3K 765
T55J5 684
KK677 28
KTJJT 220
QQQJA 483
```

- `32T3K` — по-прежнему единственная раздача с одной парой, джокеров в ней нет, так что сила не меняется.
- `KK677` — теперь единственная раздача с двумя парами, становится второй по силе.
- `T55J5`, `KTJJT` и `QQQJA` — теперь все считаются четверками. `T55J5` получает ранг `3`, `QQQJA` — ранг `4`, а `KTJJT` — ранг `5`.

С учетом новых правил для джокеров общий выигрыш в этом примере составляет `5905`. Задача: примените новое правило для джокеров, найдите ранги всех раздач в вашем наборе и определите новый общий выигрыш.

Теперь приступим к решению.

Часть вторая — решение

Для решения второй части нужно изменить всего два аспекта: задать новый порядок старшинства карт и реализовать другую версию метода `type()`.

Продолжим код из того же модуля:

```
# day7.py
…
class PartTwo(Solver):
    strengths: str = "J23456789TQKA"

    def type(self, hand: str) -> list[int]:
        card_counts = Counter(hand.replace("J", "")).most_common()
        h = [count for _, count in card_counts]
        return [h[0] + hand.count("J"), *h[1:]] if h else [5]
```

```
part_two = PartTwo()
print(part_two.solve())
```

Этого достаточно, чтобы решить вторую часть задачи. Вот как работает новая версия метода `type()`: мы по-прежнему передаем строку раздачи в `Counter`, но теперь дополнительно удаляем все джокеры `J` из раздачи перед обработкой. Затем, как и раньше, сортируем оставшиеся значения по убыванию с помощью `most_common()`. В результате возможны три ситуации.

- Если раздача состоит только из джокеров — например, `"JJJJJ"`, — то словарь частот будет пустым и метод `type()` вернет `[5]`. Это правильно: максимальная возможная комбинация — пятерка одного номинала.
- Если в раздаче нет джокеров, то логика полностью совпадает с реализацией из `PartOne.type()`.
- Если в раздаче есть хотя бы один джокер, но не все пять карт — `J`, то мы сначала вычисляем тип руки без учета джокеров. После вычисления базового типа руки (без джокеров) мы возвращаем измененную версию этого результата: к первому элементу списка добавляется количество джокеров. Например, для руки `"KKJJ4"` (если игнорировать джокеры) `Counter` выдаст список `[2, 1]` — две карты `K` и одну `4`. Чтобы максимизировать силу этой руки, логично считать джокеры тоже картами `K`, что даст комбинацию `"KKKK4"`. Для этого мы просто прибавляем 2 (количество джокеров) к первому элементу списка `[2, 1]`, получая `[4, 1]` — это и есть правильный тип руки `"KKKK4"`.

Еще один важный момент во второй части — новый порядок старшинства карт, заданный строкой `"J23456789TQKA"` Теперь `J` является самой слабой картой и стоит первой в списке. Благодаря этому при сравнении раздач с одинаковым типом `J` всегда будет считаться слабее любой другой карты.

На этом вторая часть задачи завершена. Мы выбрали именно эту задачу, поскольку она позволила показать, как использовать объектно-ориентированный подход (ООП), чтобы избежать дублирования кода, а также как применять `Counter` и `cmp_to_key()` в реальных задачах.

Входные данные для этой задачи находятся в папке `ch17` в коде к этой книге. Реализация функции `get_input()` размещена в модуле `util.py`.

Теперь перейдем ко второму заданию.

Cosmic Expansion

Вторая задача — *Cosmic Expansion* — взята из 11-го дня Advent of Code 2023. Полное условие можно найти по адресу https://adventofcode.com/2023/day/11. Вам предстоит расширить Вселенную и вычислить сумму длин кратчайших путей между всеми парами галактик.

Часть первая — условие задачи

Ниже приведена сокращенная версия оригинального описания.

Исследователь собрал большое количество данных и скомпоновал их в одно гигантское изображение (это и есть входные данные). В нем есть пустое пространство (`.`) и галактики (`#`). Например:

```
...#......
.......#..
#.........
..........
......#...
.#........
.........#
..........
.......#..
#...#.....
```

Исследователь хочет вычислить сумму длин кратчайших путей между всеми парами галактик. Но есть нюанс: пока свет от этих галактик шел к обсерватории, Вселенная расширилась.

Расширяется, однако, не все пространство. Только те строки и столбцы, в которых нет ни одной галактики, должны увеличиться вдвое.

В приведенном примере три столбца и две строки не содержат галактик. Эти строки и столбцы считаются расширенными, и результат «космического расширения» выглядит так:

```
....#........
.........#...
#............
.............
.............
........#....
.#...........
............#
.............
.............
.........#...
#....#.......
```

Во Вселенной, расширенной таким образом, можно найти кратчайшие пути между всеми парами галактик. Удобно присвоить каждой галактике уникальный номер. В приведенном примере таких галактик девять, а значит, пар всего 36. Каждую пару считаем только раз — порядок в паре не имеет значения. Для каждой пары галактик нужно найти один из кратчайших путей, двигаясь только вверх, вниз, влево или вправо, переходя по клеткам, содержащим `.` или `#`. Путь может проходить через другие галактики.

Если пронумеровать галактики и выделить крестиками (x) один из кратчайших путей, например от галактики 5 к галактике 9, то получится следующее:

```
....1........
.........2...
3............
.............
.............
........4....
.5...........
.xx.........6
..xx.........
...xx........
....xx...7...
8....9.......
```

Минимальный путь от галактики 5 до галактики 9 занимает 9 шагов (8 шагов по x плюс один шаг на саму галактику 9).

Вот еще несколько примеров.

Между галактикой 1 и галактикой 7 — 15 шагов.

Между галактикой 8 и галактикой 9 — 5 шагов.

В этом примере после расширения Вселенной сумма кратчайших расстояний между всеми 36 парами галактик составляет 374.

Задача: расширьте Вселенную по описанным правилам, затем найдите длину кратчайшего пути для каждой пары галактик и определите суммарную длину всех этих путей.

Теперь перейдем к решению этой части.

Часть первая — решение

Начнем с базовых компонентов:

```
# day11.py
from itertools import combinations
from typing import NamedTuple, Self
from util import get_input

universe = get_input("input11.txt")

class Galaxy(NamedTuple):
    x: int
    y: int

    @classmethod
    def expand(
        cls, galaxy: Self, xfactor: int, yfactor: int
    ) -> Self:
        return cls(galaxy.x + xfactor, galaxy.y + yfactor)
```

```
    @classmethod
    def manhattan(cls, a: Self, b: Self) -> int:
        return abs(a.x - b.x) + abs(a.y - b.y)

class ExpansionCoeff(NamedTuple):
    x: int = 0
    y: int = 0
```

Код начинается с операторов импорта. Чтобы перебрать все пары галактик, нам понадобится функция `combinations()` из модуля `itertools`. Мы также импортируем `NamedTuple` и `Self` из модуля `typing`, а еще нашу функцию `get_input()`, которая читает входной файл. Прочитанные данные сохраняются в переменную `universe`.

Для представления галактики мы создаем класс `Galaxy`, который наследуется от `NamedTuple`. Благодаря такому подходу упрощается реализация и автоматически предоставляются полезные функции, такие как сравнение и отображение. Альтернативными вариантами могли бы быть `dataclass`, комплексные числа или обычный класс.

Класс `Galaxy` содержит координаты `x` и `y`, а также метод `expand()`, который возвращает новую галактику со смещенными координатами (для моделирования расширения Вселенной), и метод `manhattan()`, который вычисляет расстояние до другой галактики по манхэттенской метрике (или *геометрии таксиста*). Подробности можно найти по адресу https://en.wikipedia.org/wiki/Taxicab_geometry (https://ru.wikipedia.org/wiki/Расстояние_городских_кварталов). Если коротко, то расстояние вычисляется как сумма модулей разностей координат — при движении строго по вертикали и горизонтали в дискретной сетке.

Кроме того, мы определяем вспомогательный класс `ExpansionCoeff`, представляющий коэффициент расширения, который будет использоваться при преобразовании координат.

Теперь перейдем к основному блоку логики решения:

```
# day11.py
…
def coords_to_expand(universe: list[str]) -> list[int]:
    return [
        coord
        for coord, row in enumerate(universe)
        if set(row) == {"."}
    ]

def expand_universe(universe: list[str], coeff: int) -> set:
    galaxies = parse(universe)

    rows_to_expand = coords_to_expand(universe)
    galaxies = expand_dimension(
        galaxies, rows_to_expand, ExpansionCoeff(y=coeff - 1)
    )
```

```
    transposed_universe = ["".join(col) for col in zip(*universe)]
    cols_to_expand = coords_to_expand(transposed_universe)
    return expand_dimension(
    galaxies, cols_to_expand, ExpansionCoeff(x=coeff - 1)
)

def parse(ins: list[str]) -> set:
    return {
        Galaxy(x, y)
        for y, row in enumerate(ins)
        for x, val in enumerate(row)
        if val == "#"
    }

def solve(universe: list[str], coeff) -> int:
    expanded_universe = expand_universe(universe, coeff)
    return sum(
        Galaxy.manhattan(g1, g2)
        for g1, g2 in combinations(expanded_universe, 2)
    )
```

Функция `solve()` отвечает за расчет расширенной Вселенной и возвращает сумму кратчайших путей между каждой парой галактик. Основная работа выполняется в методе `expand_universe()`.

Сначала мы разбираем входные данные, чтобы получить множество объектов `Galaxy`. Расширение Вселенной выполняется в два этапа: сначала по вертикали, затем по горизонтали. Вселенная представлена в виде списка строк, поэтому ее можно воспринимать как двумерную матрицу. Расширить ее по двум ортогональным направлениям можно с помощью двух подходов. Первый — написать отдельную функцию для каждого направления и вызывать их, передавая Вселенную как аргумент. Второй (то, что мы реализовали) — написать код расширения только для вертикального направления и вызвать его дважды: один раз со Вселенной в исходном виде, второй — с ее транспонированной версией. Это дает тот же эффект, что и расширение по горизонтали.

Если вы не знакомы с понятием транспонированной матрицы, то можете узнать о нем по адресу https://en.wikipedia.org/wiki/Transpose. Проще говоря, транспонированная матрица получается путем «переворачивания» исходной матрицы по диагонали. В результате строки становятся столбцами и наоборот.

В выделенной строке кода, где транспонируется Вселенная, вы можете увидеть, как это делается. Строго говоря, можно было бы использовать `zip(*universe)`, но такой подход возвращает не список строк, а набор кортежей, что требует дополнительной аннотации типов. Чтобы избежать этого и сохранить совместимость с оригинальным форматом (список строк), мы преобразуем транспонированную матрицу обратно в список строк. Так решение остается типобезопасным и согласованным.

Стоит отметить, что в профессиональной разработке предпочтительнее адаптировать аннотации типов под код, а не наоборот. Однако в данном случае мы стремились к максимальной читабельности кода, поэтому выбрали упрощенный вариант.

Несколько слов о методе `coords_to_expand()`. Алгоритм, реализующий расширение Вселенной вдоль одного направления, зависит от того, что координаты для расширения отсортированы. Мы просто проходим по Вселенной сверху вниз и находим те строки, в которых нет ни одной галактики, поэтому итоговый список координат получается отсортированным — нет нужды вызывать `sorted()` вручную.

Осталось рассмотреть ключевую часть — расширение в одномерном направлении:

```
# day11.py
…
def expand_dimension(
    galaxies: set,
    coords_to_expand: list[int],
    expansion_coeff: ExpansionCoeff,
) -> set:
    dimension = "x" if expansion_coeff.y == 0 else "y"
    for coord in reversed(coords_to_expand):
        new_galaxies = set()
        for galaxy in galaxies:
            if getattr(galaxy, dimension) >= coord:
                galaxy = Galaxy.expand(galaxy, *expansion_coeff)
            new_galaxies.add(galaxy)
        galaxies = new_galaxies
    return new_galaxies
```

Функция `expand_dimension()` может показаться неочевидной, поэтому разберем ее. Сначала определяем, по какому измерению будем выполнять расширение, — это зависит от объекта `expansion_coeff`. Если его атрибут `y` равен `0`, значит, расширяемся по оси *X* (вдоль строк). Если `x` равен `0`, значит, расширяемся по оси *Y* (вдоль столбцов).

Далее начинается вложенный цикл. Внешний цикл проходит по всем координатам, которые нужно расширить. Обратите внимание: мы обходим координаты в обратном порядке. Благодаря этому все координаты смещаются строго корректно и галактики не «перепрыгивают» через собственные новые позиции.

Перед входом во внутренний цикл мы создаем множество `new_galaxies`, которое будет содержать все галактики после обработки текущей координаты. Одни из них будут смещены, другие останутся на месте.

После завершения внутреннего цикла мы присваиваем `galaxies = new_galaxies` и переходим к следующей координате для расширения. По завершении этого процесса все галактики будут смещены корректно.

На этом все. Теперь остается только вызвать решатель с нужным коэффициентом расширения:

```
# day11.py
print(solve(universe, coeff=2))
```

Так мы получим решение для первой части.

Часть вторая — условие задачи

Как и в первой задаче (Camel Cards), вторая часть представляет собой небольшое усложнение первой.

Теперь вместо удвоения пустых строк и столбцов каждую пустую строку и каждый пустой столбец нужно увеличить в миллион раз. То есть каждая пустая строка заменяется на 1 000 000 строк, а каждый пустой столбец — на 1 000 000 столбцов.

(В предыдущем примере, если бы каждая пустая строка и столбец увеличивались всего лишь в 100 раз, сумма кратчайших расстояний между всеми парами галактик составила бы 8410. Но во входных данных вашей задачи Вселенная должна расширяться гораздо сильнее.)

Начните с того же самого начального изображения, расширьте Вселенную по новым правилам, а затем найдите сумму кратчайших расстояний между всеми парами галактик. Какова эта сумма?

Во многих задачах Advent of Code при переходе ко второй части становится ясно, насколько удачной была реализация первой. В нашем случае благодаря тому, что мы представляли галактики в виде множества объектов `Galaxy`, а не работали напрямую со строками или матрицами, никаких изменений в логике кода не требуется. Достаточно просто заменить коэффициент расширения при вызове функции `solve()`.

Часть вторая — решение

Вот как вызывается функция `solve()` во второй части:

```
# day11.py
print(solve(universe, coeff=int(1e6)))
```

На этом все. Теперь вместо коэффициента 2 мы передаем 1 000 000 — и получаем правильный ответ для второй части.

Благодаря правильному выбору структуры данных в первой части мы можем расширять Вселенную, используя любой коэффициент, без ущерба для производительности и корректности.

Используемый прием — частный случай концепции, называемой *«разреженная матрица» («разреженный массив»)*. Подробнее о ней можно прочитать на https://en.wikipedia.org/wiki/Sparse_matrix (https://ru.wikipedia.org/wiki/Разреженная_матрица).

Когда данные представлены в виде матрицы, обычно используют список списков или, как в этой задаче, список строк. Но иногда матрица почти полностью пуста и значимая информация занимает лишь небольшую часть пространства.

Так обстоит дело и в нашей задаче: галактики занимают лишь малую долю Вселенной, остальное — пустота. В таких случаях не имеет смысла хранить все; гораздо лучше использовать другую структуру. В данном решении мы использовали множество координат, и этого вполне достаточно.

В других ситуациях можно было бы применить словарь. Например, если бы у каждой галактики был связан показатель светимости, то координаты можно было бы использовать в качестве ключей, а значения — для хранения светимости. Идея остается прежней: мы храним только информацию о галактиках, а не пустое пространство, чего не избежать при использовании списка списков.

Как уже говорилось в предыдущих главах, выбор подходящих структур данных критически важен. В этой задаче, если бы мы хранили информацию о расширенной Вселенной в виде нового списка строк (или списков), в первой части такой подход сработал бы, но во второй — уже нет.

Заключение

Прежде чем завершить эту главу, выскажем несколько заключительных соображений.

Как мы упоминали в начале главы, представленные решения двух задач не претендуют на звание самых элегантных или самых быстрых. Существуют более эффективные алгоритмы, особенно когда дело касается обработки больших объемов данных.

Кроме того, структура кода в задачах различалась намеренно: в задаче *Camel Cards* мы использовали объектно-ориентированный подход, а в *Cosmic Expansion* — более функциональный. Это было сделано специально, чтобы показать вам разные подходы к организации решения одной и той же задачи.

Разумеется, можно было распределить код между классами и функциями по-другому, а также выбрать иные структуры данных для представления информации.

Мы старались отдавать приоритет читабельности кода и его простоте, но при этом дать вам возможность познакомиться с концепциями, которые не были затронуты в других главах, — с пользовательскими функциями-компараторами и разреженными матрицами.

Надеемся, вам понравилось это небольшое отступление от темы *профессиональной разработки на Python* и, может быть, мы зажгли в вас искру интереса.

Мы искренне советуем вам зарегистрироваться на Advent of Code и попробовать придумать собственные решения задач, приведенных в этой главе. Попробуйте

применить ООП там, где мы его не использовали, и наоборот. Протестируйте разные алгоритмы и другие способы организации данных. А главное — получайте удовольствие. Решать такие задачи действительно интересно, а если вы похожи на нас, то это еще и очень затягивает!

В последнем разделе этой главы приведены списки сайтов с задачами по программированию, которые, как мы надеемся, покажутся вам полезными и интересными.

Сайты с задачами по программированию

Ниже представлены списки некоторых наших любимых сайтов с задачами. Одни сайты бесплатные, другие — нет. Одни ориентированы на математику, другие — на чистое программирование. Есть сайты для подготовки к собеседованиям, а есть — просто для удовольствия.

Подготовка к собеседованиям

- LeetCode: https://leetcode.com.
- HackerRank: https://www.hackerrank.com.
- CodeSignal: https://codesignal.com.
- Coderbyte: https://coderbyte.com.
- HackerEarth: https://www.hackerearth.com.

Соревновательное программирование

- Codeforces: https://codeforces.com.
- Topcoder: https://www.topcoder.com.
- AtCoder: https://atcoder.jp.
- Sphere Online Judge (SPOJ): https://www.spoj.com.

Развитие навыков и обучение

- Project Euler: https://projecteuler.net (Фабрицио был членом команды Project Euler — он создавал свои задачи и участвовал в разработке других).
- Exercism: https://exercism.org.
- Codewars: https://www.codewars.com.

Получение удовольствия и общение

- Advent of Code: https://adventofcode.com.
- CodinGame: https://www.codingame.com.

Машинное обучение

- Kaggle: https://www.kaggle.com.

Надеемся, вы найдете время заглянуть хотя бы на некоторые из этих сайтов. Они помогут поддерживать остроту ума, освежить в памяти или изучить алгоритмы, структуры данных и выучить новые языки.

Резюме

В этой главе вы заглянули в мир задач по программированию. Вы разобрали две задачи с сайта Advent of Code и немного познакомились с другой Вселенной, в которой программирование используется для обучения, развлечения, подготовки к собеседованиям и соревнований.

Кроме того, вы узнали, что такое пользовательские функции-компараторы и разреженные матрицы, и увидели, как можно применять некоторые из концепций, описанных в предыдущих главах, для решения реальных задач.

Теперь наше путешествие подошло к концу. Дальше все зависит от вас. Применяйте полученные знания и извлекайте максимум из того, чему вы научились, читая эти страницы.

Мы старались дать надежную основу, на которую можно опираться, делая следующие шаги, — и в плане знаний, и в плане подходов к разработке.

Надеемся, что нам удалось передать нашу страсть и опыт, и верим, что они будут сопровождать вас, куда бы вы ни отправились в дальнейшем.

Спасибо, что были с нами. Желаем удачи и надеемся, что вам было интересно читать эту книгу.

Read IT Club

Комьюнити рецензентов
и переводчиков ИТ-литературы

Миссия участников клуба — обеспечить высокое качество профессиональной переводной литературы в России. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.

присоединиться к нам

Аарон Максвелл

МОЩНЫЙ PYTHON: ПАТТЕРНЫ И СТРАТЕГИИ СОВРЕМЕННОГО ПРОГРАММИРОВАНИЯ

Как стать экспертом в создании сложных и мощных приложений на Python, не тратя время на повторение уже известных основ или перечисление ненужных функций? Аарон Максвелл фокусируется на первопринципах Python, которые действуют подобно катализаторам для всего остального: достаточно получить 5 % знаний в области программирования, чтобы остальные 95 % подтянулись автоматически.

SPRINT book