Секреты Python Pro

Дейн Хиллард

MANNING

Practices of the Python Pro

DANE HILLARD

MANNING
SHELTER ISLAND

Дейн Хиллард

Секреты Python Pro

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2021

ББК 32.973.2-018.1
УДК 004.43
Х45

Хиллард Дейн

Х45 Секреты Python Pro. — СПб.: Питер, 2021. — 320 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1684-3

Код высокого качества — это не просто код без ошибок. Он должен быть чистым, удобочитаемым и простым в сопровождении. Путь от рядового питониста к профи не прост, для этого вам нужно разбираться в стиле, архитектуре приложений и процессе разработки.

Книга «Секреты Python Pro» научит проектировать ПО и писать качественный код, то есть делать его понятным, сопровождаемым и расширяемым. Дейн Хиллард — профессиональный питонист, с помощью примеров и упражнений он покажет вам, как разбивать код на блоки, повышать качество за счет снижения сложности и т. д. Только освоив основополагающие принципы, вы сможете сделать так, чтобы чтение, сопровождение и переиспользование вашего кода не доставляли проблем ни вам, ни вашим коллегам.

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

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617296086 англ.
978-5-4461-1684-3

© 2020 by Manning Publications Co. All rights reserved.
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Для профессионалов», 2021

Краткое содержание

Часть I.
Почему это важно?

Глава 1. Крупный план 28

Часть II.
Основы проектирования

Глава 2. Разделение ответственности 48

Глава 3. Абстракция и инкапсуляция 76

Глава 4. Создание дизайна для производительности 100

Глава 5. Тестирование ПО 127

Часть III.
Организация крупных систем

Глава 6. Разделение ответственности на практике 162

Глава 7. Расширяемость и гибкость 198

Глава 8. Правила (и исключения) наследования 218

Глава 9. Поддержание компактности 242

Глава 10. Достижение слабой сопряженности 265

Часть IV.
Что дальше?

Глава 11. Только вперед 292

Приложение. Установка языка Python 312

Оглавление

Предисловие **15**

Благодарности **17**

О книге 19

Для кого эта книга 19

Структура книги 20

О коде 21

Форум liveBook 22

Об авторе 24

Об обложке 25

От издательства 26

Часть I.
Почему это важно?

Глава 1. Крупный план 28

1.1. Python — язык для корпоративных приложений 30

1.1.1. Времена меняются 30

1.1.2. Что мне нравится в Python 30

1.2. Python — язык для обучения 31

1.3. Дизайн — это процесс 32
1.3.1. Пользовательский опыт 33
1.3.2. Вы уже были здесь раньше 35
1.4. Дизайн обеспечивает высокое качество ПО 35
1.4.1. Соображения по дизайну ПО 36
1.4.2. Органически выращенное ПО 37
1.5. Когда нужно уделять время дизайну 40
1.6. Новые начинания 41
1.7. Дизайн демократичен 41
1.7.1. Хладнокровие 42
1.8. Как пользоваться этой книгой 44
Итоги 45

Часть II.
Основы проектирования

Глава 2. Разделение ответственности 48
2.1. Организация пространства имен 49
2.1.1. Пространства имен и инструкция import 50
2.1.2. Лики импортирования 53
2.1.3. Пространства имен предотвращают коллизии 55
2.2. Иерархия разделения в Python 57
2.2.1. Функции 57
2.2.2. Классы 64
2.2.3. Модули 71
2.2.4. Пакеты 72
Итоги 75

Глава 3. Абстракция и инкапсуляция 76
3.1. Что такое абстракция? 77
3.1.1. Черный ящик 77
3.1.2. Абстракция подобна луковице 79
3.1.3. Абстракция упрощает 82
3.1.4. Декомпозиция обеспечивает возможность абстракции 83

3.2. Инкапсуляция 84
3.2.1. Конструкции инкапсуляции в Python 84
3.2.2. Ожидания приватности в Python 86
3.3. Попробуйте сами 86
3.3.1. Рефакторинг 88
3.4. Стили программирования тоже являются абстракцией 90
3.4.1. Процедурное программирование 90
3.4.2. Функциональное программирование 91
3.4.3. Декларативное программирование 93
3.5. Типизация, наследование и полиморфизм 94
3.6. Распознавание неправильной абстракции 97
3.6.1. Как корове седло 97
3.6.2. Подходите ко всему с умом 98
Итоги 99
Глава 4. Создание дизайна для производительности 100
4.1. Сквозь время и пространство 101
4.1.1. Сложность немного... сложна 102
4.1.2. Временная сложность 103
4.1.3. Пространственная сложность 107
4.2. Производительность и типы данных 109
4.2.1. Типы данных для постоянного времени 110
4.2.2. Типы данных для линейного времени 111
4.2.3. Пространственная сложность операций на типах данных 111
4.3. Работоспособность, правильность и быстрота 115
4.3.1. Сделайте работоспособным 115
4.3.2. Сделайте правильным 116
4.3.3. Сделайте быстрым 120
4.4. Инструменты 121
4.4.1. timeit 121
4.4.2. Профилирование ЦП 123

4.5. Попробуйте сами ... 125
Итоги ... 126

Глава 5. Тестирование ПО ... 127
5.1. Что такое тестирование ПО? ... 128
5.1.1. Соответствует ли содержимое этикетке? ... 128
5.1.2. Суть функционального тестирования ... 129
5.2. Подходы к функциональному тестированию ... 131
5.2.1. Ручное тестирование ... 131
5.2.2. Автоматизированное тестирование ... 131
5.2.3. Приемочное тестирование ... 132
5.2.4. Юнит-тестирование ... 134
5.2.5. Интеграционное тестирование ... 136
5.2.6. Пирамида тестирования ... 137
5.2.7. Регрессионное тестирование ... 138
5.3. Констатация фактов ... 139
5.4. Юнит-тестирование с помощью unittest ... 140
5.4.1. Организация тестов с помощью unittest ... 140
5.4.2. Выполнение тестов с помощью unittest ... 141
5.4.3. Написание первого теста с помощью unittest ... 141
5.4.4. Написание первого интеграционного теста с помощью unittest ... 145
5.4.5. Тестовые дублеры ... 148
5.4.6. Попробуйте сами ... 150
5.4.7. Написание дополнительных тестов ... 152
5.5. Тестирование с помощью фреймворка pytest ... 153
5.5.1. Организация тестов с помощью фреймворка pytest ... 154
5.5.2. Конвертирование тестов unittest в pytest ... 155
5.6. За пределами функционального тестирования ... 156
5.6.1. Тестирование производительности ... 156
5.6.2. Нагрузочное тестирование ... 157

5.7. Разработка на основе тестов: пример 158
5.7.1. Это образ мышления 158
5.7.2. Это философия 159
Итоги 160

Часть III. Организация крупных систем

Глава 6. Разделение ответственности на практике 162
6.1. Приложение командной строки для создания закладок 163
6.2. Обзор приложения Bark 164
6.2.1. Выгоды от разделения: реприза 165
6.3. Первоначальное разделение ответственности 166
6.3.1. Слой постоянства данных 168
6.3.2. Слой бизнес-логики 181
6.3.3. Слой визуализации 187
Итоги 197

Глава 7. Расширяемость и гибкость 198
7.1. Что такое расширяемый код? 199
7.1.1. Добавление новых форм поведения 199
7.1.2. Изменение существующего поведения 202
7.1.3. Слабая сопряженность 204
7.2. Решения проблемы жесткости 206
7.2.1. Отпустить на свободу: инверсия управления 206
7.2.2. Дьявол в мелочах: опора на интерфейсы 210
7.2.3. Борьба с энтропией: принцип надежности 211
7.3. Упражнение на растяжку 212
Итоги 217

Глава 8. Правила (и исключения) наследования 218
8.1. Наследование раньше 219
8.1.1. Серебряная пуля 219
8.1.2. Трудности иерархий 219

8.2. Наследование сейчас ... 222
8.2.1. Зачем нужно наследование? ... 222
8.2.2. Подстановка ... 223
8.2.3. Идеальный для наследования вариант использования ... 225
8.3. Наследование в Python ... 228
8.3.1. Проверка типов ... 228
8.3.2. Обращение к суперклассу ... 230
8.3.3. Множественное наследование и порядок разрешения методов ... 230
8.3.4. Абстрактные базовые классы ... 235
8.4. Наследование и композиция в приложении Bark ... 238
8.4.1. Рефакторинг для использования абстрактного базового класса ... 238
8.4.2. Окончательная проверка работы с наследованием ... 240
Итоги ... 241

Глава 9. Поддержание компактности ... 242
9.1. Насколько большим должен быть класс/функция/модуль? ... 243
9.1.1. Физический размер ... 243
9.1.2. Ограниченная ответственность ... 244
9.1.3. Сложность кода ... 244
9.2. Разложение сложности ... 250
9.2.1. Извлечение конфигурационной информации ... 250
9.2.2. Извлечение функций ... 253
9.3. Декомпозиция классов ... 256
9.3.1. Сложность инициализации ... 256
9.3.2. Извлечение классов и переадресация вызовов ... 259
Итоги ... 264

Глава 10. Достижение слабой сопряженности ... 265
10.1. Определение сопряженности ... 266
10.1.1. Сопряженность ... 266

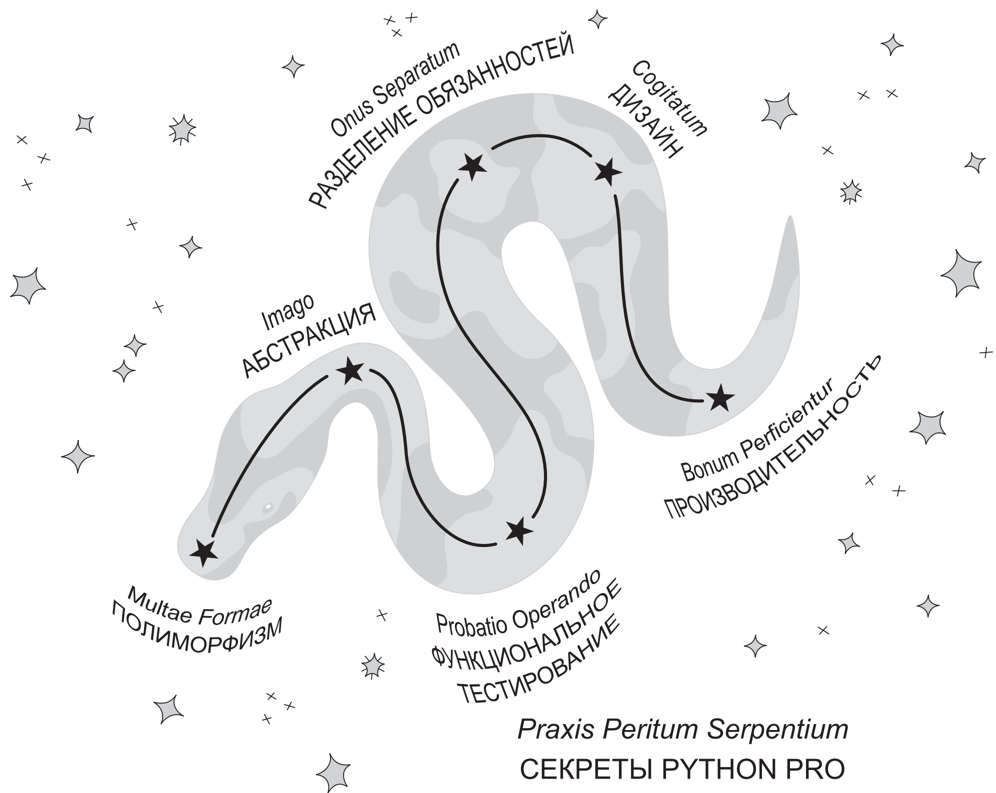
10.1.2. Тесная сопряженность ... 267
10.1.3. Слабая сопряженность ... 270
10.2. Распознавание сопряженности ... 274
10.2.1. Излишняя зависимость ... 274
10.2.2. Стрельба дробью ... 275
10.2.3. Дырявая абстракция ... 275
10.3. Сопряженность в приложении Bark ... 277
10.4. Решение проблемы сопряженности ... 280
10.4.1. Передача сообщений пользователям ... 281
10.4.2. Постоянство хранения закладок ... 284
10.4.3. Попробуйте сами ... 285
Итоги ... 289

Часть IV. Что дальше?

Глава 11. Только вперед ... 292
11.1. Что теперь? ... 292
11.1.1. Разработайте план ... 293
11.1.2. Исполните план ... 295
11.1.3. Отслеживайте свой прогресс ... 297
11.2. Паттерны проектирования ... 299
11.2.1. Взлеты и падения паттернов в Python ... 301
11.2.2. С чего начать ... 302
11.3. Распределенные системы ... 302
11.3.1. Режимы сбоя в распределенных системах ... 303
11.3.2. Обращение к состоянию приложения ... 304
11.3.3. С чего начать ... 305
11.4. Погружение в Python ... 305
11.4.1. Питоновский стиль ... 305
11.4.2. Языковые средства являются паттернами ... 306
11.4.3. С чего начать ... 307

11.5. Что вы узнали ... 308
11.5.1. Путешествие туда и обратно: рассказ разработчика ... 308
11.5.2. Выход из системы ... 310
Итоги ... 310

Приложение. Установка языка Python ... 312
А.1. Какую версию Python использовать? ... 313
А.2. «Системный» Python ... 313
А.3. Установка других версий Python ... 314
А.3.1. Скачайте официальный Python ... 314
А.3.2. Скачать с помощью Anaconda ... 316
А.4. Проверка установки ... 316


Onus Separatum
РАЗДЕЛЕНИЕ ОБЯЗАННОСТЕЙ
Cogitatum
ДИЗАЙН
Imago
АБСТРАКЦИЯ
Bonum Perficientur
ПРОИЗВОДИТЕЛЬНОСТЬ
Multae Formae
ПОЛИМОРФИЗМ
Probatio Operando
ФУНКЦИОНАЛЬНОЕ
ТЕСТИРОВАНИЕ
Praxis Peritum Serpentium
СЕКРЕТЫ PYTHON PRO

Предисловие

С языком Python мы родились в декабре 1989 года. За последующие три десятилетия я многого достиг, и Python тоже. Все больше людей выбирают его для достижения самых амбициозных целей в области data science, машинного обучения и многого другого. С тех пор как я освоил Python, этот «второй среди лучших языков для всего на свете» для меня стал первым и лучшим.

У меня был довольно традиционный путь в программирование через факультет электротехники и computer science Мичиганского университета. В то время все лабораторные на курсе были сосредоточены в основном на C++ и MATLAB — языках, которые я продолжал использовать на первом рабочем месте. На следующей должности, обрабатывая большой объем данных из области биоинформатики, я разработал несколько экземпляров скриптов для командной оболочки на SQL и начал использовать PHP для работы на персональном веб-сайте WordPress.

Хотя я и получал результаты (иногда даже крутые), ни один из используемых мною языков не находил во мне *отклика*. Я считал, что языки программирования — это просто средство достижения цели и у них мало шансов быть *клевыми*. Примерно в это время один товарищ пригласил меня присоединиться к хакатон-проекту по разработке библиотеки Ruby.

Внезапно мир взорвался красками, фрукты стали слаще, деревья выше и все такое. Простота использования интерпретируемого языка и удобный

для человека синтаксис Ruby действительно заставили меня задуматься о тех инструментах, которые я использовал. Правда, я не слишком долго задержался на Ruby, решив для следующей итерации персонального веб-сайта попробовать Python и веб-фреймворк Django. Это принесло мне ту же радость и плавный прогресс в обучении, который был с Ruby, и с тех пор к Ruby я не возвращался.

Теперь, когда Python широко признан наилучшим языком для многих задач, людям, приходящим в разработку ПО, не нужно переживать этап проб и ошибок, который прошел я, — они открывают новые интересные пути к карьере в области ПО. Но несмотря на эти различия, надеюсь, что мы поделимся общим опытом радости от программирования на Python, и книгой «Секреты Python Pro»я хочу внести свой вклад в эту радость.

Присоединяйтесь к чудесному путешествию по Python, в которое я попал случайно. Хочу увидеть, как вы создаете веб-сайт, конвейер данных или автоматизированную систему полива растений. Что бы вы ни нафантазировали, Python вас прикроет. Присылайте скриншоты и образцы кода ваших проектов по адресу python-pro-projects@danehillard.com.

Благодарности

Я писал эту книгу не один и глубоко признателен всем, кто как только мог помогал мне на разных этапах. Я ценю вас.

Почти каждый, кто участвовал в создании книги, может сказать, что работы всегда оказывается больше, чем ожидалось, — я слышал это много раз. Но настоящая борьба разворачивается там, где приходится уравновешивать чрезмерную работу с реальной жизнью.

Стефани! Твоя поддержка и то, что ты терпела мой треп, было безумно важно при создании книги. Благодарю тебя за то, что ты так легко воспринимала мое невнимание и вытаскивала меня из этого проекта в самые трудные времена. Я бы не справился без тебя.

Спасибо моим родителям, Киму и Донне, за то, что всегда направляли мою энергию в творческое русло.

Спасибо моему дорогому другу Винсенту Чжану за бесчисленные ночи в кофейне, когда мы писали код. Там и родилась концепция книги, и твое одобрение помогло мне взяться за дело.

Спасибо Джеймсу Нгуену за настойчивость — ты изменил свою жизнь, чтобы стать разработчиком, и олицетворяешь целевую аудиторию этой книги. Я горжусь твоими достижениями.

Выражаю благодарность всем моим коллегам в *ITHAKA* и за ее пределами за ваш вклад и поддержку — вы выдержали этот непростой период.

Тони Арритоле, редактору, спасибо за решимость подталкивать меня к более качественному изложению. Процесс написания книги чреват многими неожиданными препятствиями, но вы обеспечили мне стабильность. Спасибо.

Ник Уоттс, научный редактор, ваши отзывы превратили мой безумный бред в правдоподобные размышления о разработке ПО. Я очень ценю вашу искренность и проницательность.

Спасибо Майку Стивенсу и Марджан Бейс из издательства Manning за то, что они поверили в мою идею и доверили ее реализовать. Спасибо всем в Manning за неустанную работу по воплощению авторских замыслов.

Всем рецензентам — Элу Кринкеру, Бонни Бейли, Буркхарду Нестману, Крису Уэйману, Дэвиду Кернсу, Дэвиду Кадамуро, Эриксу Зеленке, Грэму Уилеру, Грегори Матушеку, Жану-Франсуа Морину, Йенсу Кристиану Бредалю Мадсену, Джозефу Перениа, Марку Томасу, Маркусу Маучеру, Майку Стивенсу, Патрику Ригану, Филу Соренсену, Рафаэлю Кассемиро Фрейре, Ричарду Филдсенду, Роберту Уолшу, Стивену Парру, Свену Стампфу и Уиллису Хэмптону — ваши предложения помогли сделать книгу лучше.

Последняя благодарность всем, кто оказал положительное влияние — намеренно, ненамеренно или иным образом — на мой путь в программировании и эту книгу. Просто невозможно составить исчерпывающий список. Спасибо Марку Брехобу, доктору Эндрю Деорио, Джесси Силафф, Треку Гловаки, всем в SAIC (в нашем маленьком офисе в Анн-Арборе), всем в Compendia Bioscience (и друзьям), Брэндону Родсу, Кеннету Лаву, Трею Ханнеру, Джеффу Триплетту, Мариатте Виджая, Али Спиттелу, Крису Койеру, Саре Драснер, Дэвиду Бизли, Дрону Аялону, Тиму Аллену, Санди Мец и Мартину Фаулеру.

О книге

В книге «Секреты Python Pro» есть несколько идей, которые разработчики на практически любом языке могут использовать для улучшения своей работы. Она будет отличной книгой после изучения основ языка Python.

ДЛЯ КОГО ЭТА КНИГА

Книга предназначена для тех, кто начинает свой путь в программировании или тех, кто использует ПО в качестве дополнительного инструмента в своей работе. Описанные здесь идеи помогут создавать ПО, улучшать его сопровождение и облегчат совместную работу.

В естественных науках воспроизводимость и работа с источниками являются важными аспектами исследовательского процесса. Все больше исследований опирается на софт, где особенно важен понятный код, который можно обновлять и совершенствовать. Но в учебных планах по-прежнему уделено слишком мало внимания связи ПО с другими дисциплинами. Тем, кто имеет небольшой опыт в формальной разработке софта, эта книга расскажет о принципах создания совместно переиспользуемого ПО.

Если вы хорошо разбираетесь в объектно-ориентированном программировании и предметно-ориентированном дизайне, то эта книга, возможно, покажется вам простой. Но начинающие разработчики обязательно найдут в ней много интересного.

СТРУКТУРА КНИГИ

Книга состоит из одиннадцати глав и разделена на четыре части. В частях I и II излагается тематический материал и приводятся краткие примеры и упражнения. Часть III основана на материалах из предыдущих глав и тоже содержит упражнения. В части IV содержатся рекомендации по дальнейшему изучению языка Python.

Часть I «Почему это важно?» рассказывает о восхождении языка Python к славе и отвечает на вопрос, в чем ценность разработки ПО.

- **Глава 1** посвящена недавней истории языка Python и его достоинствам. Объясняется, почему так важен дизайн программного обеспечения и как он проявляется в вашей повседневной работе.

Часть II «Основы проектирования» охватывает концепции высокого уровня, лежащие в основе дизайна и разработки ПО.

- **Глава 2** посвящена теме разделения ответственности — фундаментальной концепции, являющейся основополагающей для других, описанных в книге.
- **Глава 3** объясняет абстракцию и инкапсуляцию и показывает, как сокрытие информации и предоставление простых интерфейсов для более сложной логики помогает держать код под контролем.
- **Глава 4** предлагает подумать о производительности, разных структурах данных, подходах и инструментах создания быстрых программ.
- **Глава 5** учит тестированию программ с использованием различных подходов — от юнит-тестирования до сквозного тестирования.

Часть III «Организация крупных систем» проведет вас по процессу создания реального приложения с использованием принципов, которые вы усвоили.

- **В главе 6** вы самостоятельно спроектируете приложение. В главе также содержатся упражнения, которые помогут создать базу программы.
- **Глава 7** разбирает понятия расширяемости и гибкости и содержит упражнения на добавление расширяемости в приложение.
- **Глава 8** рассказывает о наследовании: когда и где его следует использовать. В главе также содержатся упражнения на проверку наследования в приложении.
- **Глава 9** знакомит с инструментами и подходом для предотвращения чрезмерного роста кода по мере развития проекта.
- **Глава 10** объясняет понятие слабой сопряженности и приводит несколько заключительных упражнений на уменьшение сопряженности в разрабатываемом приложении.

Часть IV «Что дальше?» дает несколько рекомендаций, как и чему учиться дальше.

- **В главе 11** я покажу, как составить план дальнейшего обучения, и затрону несколько тем, которые могут быть интересны, если вы настроены на последующее погружение в вопросы разработки ПО.

Рекомендую читать книгу «Секреты Python Pro» последовательно, хотя вы можете пропустить главы из частей I и II, если эти темы вам знакомы. Часть III лучше читать последовательно и делать упражнения по порядку.

В конце книги есть приложение, где рассказано, как установить Python:

- **В приложении** рассматривается вопрос о том, какую версию Python установить, и как это сделать.

О КОДЕ

Полный исходный код примеров и упражнений можно получить в репозитории книги на GitHub (https://github.com/daneah/practices-of-the-python-pro). Либо зайдите на домашнюю страницу книги (www.manning.com/books/prac-tices-of-the-python-pro) и скачайте код, нажав на кнопку Source Code (Исходный код).

Эта книга содержит много примеров исходного кода, как в пронумерованных листингах, так и в основном тексте. В обоих случаях код выделен `моноширинным шрифтом`.

Во многих случаях код был переформатирован: мы добавили разрывы строк и переработали отступы, чтобы уместиться в доступное пространство книжной страницы. В редких случаях даже этого было недостаточно, и листинги включали маркеры продолжения строк (➥). Кроме того, комментарии в коде часто удаляются из листингов, если в тексте есть описание. Большая часть листингов имеет дополнительные примечания.

В каждой главе код организован в модули Python, на которые даны ссылки в тексте. Мы надеемся, что вы напишете свою версию кода и будете использовать предоставленный исходный код только для самопроверки. В части III проекты каждой главы имеют в основе код из предыдущей, однако каждая глава содержит полную рабочую копию исходного кода.

Весь код этой книги написан на Python 3, точнее предназначен для работы с Python 3.7+. Бо́льшую часть кода можно организовать для работы в более ранних версиях без особых проблем, но все же подумайте об установке свежей версии Python для работы с этой книгой.

ФОРУМ LIVEBOOK

Покупка книги «Секреты Python Pro» включает в себя бесплатный доступ к веб-форуму от издательства Manning Publications, где вы можете комментировать книгу, задавать технические вопросы и получать

помощь от автора и других пользователей. Для того чтобы получить доступ к форуму, перейдите по ссылке https://livebook.manning.com/#!/book/practices-of-the-python-pro/discussion. Вы также можете узнать больше о форумах издательства Manning и правилах поведения на сайте по адресу https://livebook.manning.com/#!/discussion.

Издательство Manning предоставляет читателям возможность для содержательного диалога. Участие и вклад автора в работу форума остается добровольным (и неоплачиваемым). Попробуйте задать автору сложный вопрос! Форум и архивы предыдущих дискуссий будут доступны на веб-сайте издательства.

Об авторе

Дейн Хиллард — ведущий разработчик веб-приложений в *ITHAKA*, некоммерческой организации в области высшего образования. Ранее занимался созданием движков для данных телеметрии и конвейеров ETL для приложений в области биоинформатики.

Первые попытки Дейна программировать заключались в создании индивидуального стиля для своей страницы MySpace, написании скриптов для приложения 3D-моделирования Rhinoceros и создании настраиваемых под свои нужды скинов и оружия для игры Liero в MS-DOS. Дейн любит творческое написание кода и активно ищет способы объединить свою любовь к музыке, фотографии, еде и ПО.

Дейн неоднократно выступал на международных конференциях, посвященных Python и Django, и планирует продолжать, пока кто-нибудь его не остановит.

Об обложке

Рисунок на обложке книги «Секреты Python Pro» называется «Homme Finnois», или «Финский человек» (житель Финляндии). Иллюстрация взята из коллекции костюмов разных стран *Costumes de Différents Pays* Жака Грассе де Сен-Совера (1757–1810), опубликованной во Франции в 1797 году. Каждая иллюстрация нарисована и раскрашена от руки. Иллюстрации из каталога Грассе де Сен-Совера напоминают о культурных различиях между городами и весями мира, имевших место почти двести лет назад. Люди, проживавшие в изолированных друг от друга регионах, говорили на разных языках и диалектах. По одежде человека можно было определить, в каком городе, поселке или поселении он проживает.

С тех пор дресс-код сильно изменился, да и различия между разными регионами стали не столь выраженными. В наше время довольно трудно узнать жителей разных континентов, не говоря уже о жителях разных городов или регионов. Возможно, мы отказались от культурных различий в пользу более разнообразной личной жизни — и конечно, в пользу более разнообразной и стремительной технологической жизни.

Сейчас, когда все компьютерные книги похожи друг на друга, издательство Manning стремится к разнообразию и помещает на обложки книг иллюстрации, показывающие особенности жизни в разных странах мира два века назад.

В то время, когда трудно отличить одну компьютерную книгу от другой, издательство Manning демонстрирует изобретательность и инициативу компьютерного бизнеса с помощью книжных обложек, основанных на богатом разнообразии региональной жизни двухвековой давности, оживленной картинами Грассе де Сен-Совера.

ОТ ИЗДАТЕЛЬСТВА

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

Мы будем рады узнать ваше мнение! На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I

Почему это важно?

Когда вы начинаете изучать новые темы, важно учитывать общую картину, чтобы очертить рамки и сфокусировать внимание. Первая часть книги расскажет о значении языка Python в современной разработке ПО, а также поможет понять принципы и методы проектирования ПО, чтобы ваша карьера программиста была успешной.

Если вы новичок в программировании, или хотите изучить новый язык, или пытаетесь развить навыки для более крупных проектов, эта часть книги должна убедить вас в том, что Python — отличный выбор.

1 Крупный план

В этой главе:

- ✓ Использование Python в сложных проектах.
- ✓ Знакомство с высокоуровневым процессом дизайна ПО.
- ✓ Когда нужно вкладываться в в дизайн.

Я рад, что вы взяли в руки эту книгу, — это значит, что вы хотите сделать следующий шаг в разработке ПО. Или войти в софтверную индустрию, или улучшить работу. Вполне возможно, раньше вам даже платили за написание софта. Примите поздравления — вы уже профессионал! Создавать код как профи — значит применять идеи и стратегии создания и поддержки крупных проектов в долгосрочной перспективе.

Читая дальше, вы узнаете, как Python помогает мыслить масштабно и переходить от написания служебных скриптов к разработке сложного ПО. Я помогу вам заложить фундамент, на котором вы разовьете свои умения.

На карьерном пути вы, вероятно, будете сталкиваться с постоянно растущей сложностью ПО: длительное проектирование или куча кода,

навязанная вам в неподходящий момент. Как бы то ни было, вам захочется иметь в своем распоряжении набор утилит, чтобы во всем разобраться.

Книга поможет приобрести опыт и познакомит с работой сложных программных систем. Вы научитесь представлять разные системы еще до проектирования, чтобы свести к минимуму неожиданности и риски. После прочтения вы сможете с энтузиазмом погрузиться в то, что сейчас кажется непонятным и пугающим.

Я покажу, как поместить сложность вашего кода в легкую для понимания многоразовую оболочку. Вы организуете код так, чтобы он всегда был понятен, и поможете себе стать продуктивнее как в новых, так и в старых проектах!

Для примеров я использовал Python. Долгое время он был моим любимым языком программирования, и надеюсь, вы тоже его полюбите. Если у вас еще не было возможности познакомиться с Python поближе, то сначала найдите время, чтобы это сделать. Книга «Python. Экспресс-курс» Наоми Седер[1] — отличное пособие для старта.

ВЕЛИКИЙ ВОДОРАЗДЕЛ

Что вы используете: Python 2 или Python 3? Большое количество людей все еще используют Python 2, хотя Python 3 появился давным-давно, в 2008 году. В том году на вершине чартов были «Low» Флоу Райды и «No One» Алиши Киз.

Python 3 внес несколько обратно несовместимых изменений, последствия которых ощущаются и по сей день. Многие из этих изменений были перенесены в более поздние версии Python 2. Разработчики крупных проектов, использующие Python 2, преодолевают сложности, но некоторые из них, похоже, унесут софт Python 2 с собой в могилу.

---

[1] *Седер Наоми*. Python. Экспресс-курс. 3-е изд. — СПб.: Питер, 2020. — 480 с. — *Примеч. ред.*

Все примеры в книге написаны для версии Python 3. Настоятельно рекомендую установить Python 3, прежде чем продолжить работу (помощь по установке — в конце книги).

Если вам еще нужны доказательства, что Python хорош, то вот они.

1.1. PYTHON — ЯЗЫК ДЛЯ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Python исторически рассматривался как скриптовый язык. Разработчики негативно воспринимали его производительность и применимость, выбирая другие языки для своих корпоративных потребностей. Python использовался для малых задач обработки данных или персональных инструментов, а корпоративное ПО по-прежнему оставалось уделом Java, C или SAS.

1.1.1. Времена меняются

За последние несколько лет представление о Python резко изменилось. В настоящее время Python применяется практически во всех дисциплинах — от робототехники до машинного обучения и химии. Python дал старт некоторым из самых успешных интернет-компаний последнего десятилетия и не показывает никаких признаков замедления.

1.1.2. Что мне нравится в Python

Python — это глоток свежего воздуха. Как и многие мои друзья и коллеги, я очень много изучал C++ в школе, а также немного MATLAB, Perl и PHP. Я построил свой первый веб-сайт на PHP и даже в какой-то момент попробовал версию Java Spring. PHP и Java — это, как подтвердят многие успешные компании, вполне пригодные языки в этой области, но эти языки почему-то не пошли со мной на контакт.

Я нахожу синтаксис Python превосходным, и это одна из причин его растущей популярности. Он близок к письменному английскому языку,

чем не могут похвастаться другие языки программирования, и подходит как новичкам, так и тем, кто устал от сложных конструкций. Я замечал, как люди загораются от радости, когда поручают Python вывести приветствие `'Здравствуй, мир!'` с помощью `print('Здравствуй, мир!')`, и видят, как он делает именно это. Даже сейчас бывает, что я обнаруживаю стандартный библиотечный модуль, о котором раньше не знал.

Python удобочитаем — это ускоряет разработку. Хуэй Дин (Hui Ding), инженер из Instagram, проницательно отмечает, что «скорость исполнения больше не является главной заботой, а вот скорость выхода на рынок — да».[1] Python обеспечивает быстрое прототипирование и, как вы увидите, способность консолидировать софт в надежную сопроводимую кодовую базу. Вот что мне нравится в Python.

1.2. PYTHON — ЯЗЫК ДЛЯ ОБУЧЕНИЯ

В 2017 году опрос на Stack Overflow показал, что в странах с высоким уровнем дохода вопросы, связанные с Python, составляют более 10 % всех вопросов, опережая другие основные языки программирования.[2] На сегодняшний день Python является самым быстрорастущим языком программирования, что делает его удобным учебным инструментом. Процветающее сообщество разработчиков и огромное количество доступной информации означают, что он будет оставаться безопасным вариантом еще несколько лет.

В книге я исходил из того, что у вас есть базовые знания синтаксиса Python, его типов данных и классов. При этом я не жду, что вы питонист-чемпион (или чемпион?). Некоторый опыт программирования за плечами и несколько часов личного общения с Python не даст вам запутаться в представленном коде. Попробуйте применить концепции

---

[1] Мишель Гино, «Instagram плавно переходит на Python 3» (Michelle Gienow, *Instagram Makes a Smooth Move to Python 3*) (http://mng.bz/Ze0j) — это отличная рецензия на переход Instagram с Python 2 на Python 3.

[2] Дэвид Робинсон, «Невероятный рост популярности языка Python» (*The Incredible Growth of Python*), блог Stack Overflow (http://mng.bz/m48n).

из этой книги к другому языку, и вы обнаружите, что многие концепции разработки ПО выходят за рамки любой конкретной технологии.

1.3. ДИЗАЙН — ЭТО ПРОЦЕСС

Хотя слово «*дизайн*» часто описывает результат чего-либо, смысл дизайна заключается в *процессе* достижения этого результата. Возьмем дизайнеров одежды. Их цель — создать вещи, которые окажутся в руках потребителей. Однако для того, чтобы дизайнер смог донести до клиентов очередной тренд, требуется много шагов и участие многих людей (рис. 1.1).

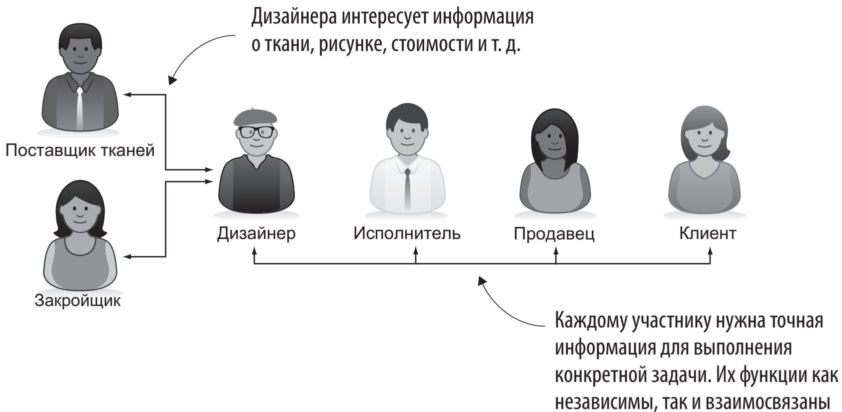


Рис. 1.1. Рабочий поток для дизайнера одежды. Дизайнер работает с целым рядом других людей, чтобы работа была выполнена

Дизайнеры работают с поставщиками тканей, чтобы получить нужные материалы. После того как изделие спроектировано, дизайнеры просят закройщиков изготовить выкройки. Готовые выкройки отправляются на производство, и далее одежда попадает в розничные магазины, где клиенты, наконец, покупают ее. Этот процесс может занимать месяцы!

Как и в моде, искусстве и архитектуре, дизайн в ПО представляет собой схематическую зарисовку планов, чтобы система могла выполнять задачи максимально эффективно. Планы помогают понять поток данных

и фрагменты системы, производящие операции с этими данными. Рисунок 1.2 показывает высокоуровневую диаграмму рабочего потока веб-сайта электронной коммерции, очерчивающую то, как пользователь будет продвигаться шаг за шагом.

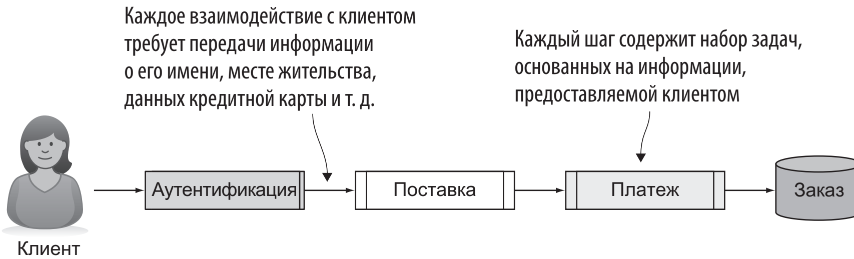


Рис. 1.2. Рабочий поток веб-сайта электронной коммерции. Система проходит ряд шагов, чтобы выполнить работу

При онлайн-покупке клиент обычно входит в систему, вводит информацию о доставке и оплачивает товар. В результате создается заказ, подлежащий обработке и отгрузке. Рабочие потоки, подобные этим, требуют выполнения огромного объема работ по дизайну. Программное обеспечение, которое управляет этими системами, соблюдает сложные правила, проверяет состояние ошибки и многое другое. И оно должно делать все это без промашек, потому что пользователи чувствительны к ошибкам. Они могут отказаться или даже активно выступать против продукта, который работает недостаточно хорошо.

1.3.1. Пользовательский опыт

Создание рабочих потоков затратно, даже если они кажутся небольшими и понятными. ПО, которое работает гладко для всех вариантов использования, требует исследования рынка, пользовательского тестирования и надежного дизайна. Некоторые продукты хорошо работают для предполагаемого варианта использования, но после их выпуска компании могут обнаружить, что пользователи делают с продуктом что-то совершенно неожиданное, на что ПО не заточено. В дизайне могут быть пробелы, которые необходимо учитывать.

Когда софт работает хорошо, мы этого почти не замечаем. Людям нравится контактировать без трений, и разработчики ПО не исключение. Работа с неподдерживаемым кодом, который не сопровождается технически, может привести к разочарованию, а незнание, как это исправить, — к гневу! Сделайте глубокий вдох.

ТРЕНИЕ

Представьте себе катание на коньках. Когда вы выходите на лед сразу после того, как ледовый комбайн заканчивает его выравнивание, катание почти не требует усилий. Вы можете чуть-чуть опираться на ноги, позволяя коньку делать свою работу. Через некоторое время коньки начинают резать лед. Скользить становится все труднее, и нужно сильно отталкиваться при каждом шаге.

Трение в пользовательском опыте похоже на грубый лед — пользователь по-прежнему способен делать то, что хочет, но он более напряжен. Опыт без трения — это опыт, который легко направляет пользователей вперед, до такой степени, что они едва замечают, как выполняется работа.

Скажем, вам поручено обновить ПО для отчетности в вашей компании. Вы видите, что значения в файлах экспорта разделены запятыми (CSV), но пользователи говорят, что табуляция (TSV) удобнее. Вы думаете: «Обновлю разделитель в выходной функции!» Затем открываете код и обнаруживаете, что все выводящие строки выглядят так:

```
print(col1_name + ',' + col2_name + ',' + col3_name + ',' + col4_name)
print(first_val + ',' + second_val + ',' + third_val + ',' + fourth_val)
```

Нужно убедиться, что вы заменили запятую на знак табуляции в шести местах. Здесь можно ошибиться. Вы заметили, что первая строка выводит заголовок, но пропустили строку, которая выводит строки данных. Для удобства разработчика, который будет использовать код после вас, сохраните значение разделителя в константе и используйте его там, где

необходимо. Еще можно применить функцию, чтобы облегчить себе построение строк. Тогда, если пользователи снова захотят запятые, изменение можно будет внести только в одном месте:

```
DELIMITER = '\t'
print(DELIMITER.join([col1_name, col2_name, col3_name, col4_name]))
print(DELIMITER.join([first_val, second_val, third_val, fourth_val]))
```

Обдумывая систему на высоком уровне, вы начнете замечать грубые участки, которые не видели раньше, и понимать, что некоторые ваши допущения неточны. Вы удивитесь еще не раз, и это побудит вас продолжать в том же духе. Как только вы начнете видеть повторяющиеся паттерны и часто встречающиеся оплошности, вы научитесь распознавать шипы, которые нужно вытащить.

1.3.2. Вы уже были здесь раньше

Осознанно или нет, вы наверняка уже проходили через процесс проектирования. Подумайте о том времени, когда вы на мгновение перестали писать код, чтобы вернуться к цели, которую вы пытались достичь. Замечали ли вы что-то, что заставило вас изменить направление движения? Видели ли вы более эффективный способ решения задачи?

Эти короткие моменты сами по себе являются процессами проектирования. Вы сопоставляли цель и текущее состояние ПО, чтобы определить дальнейшие действия. Выделение процессов проектирования на ранней стадии работы будет иметь как краткосрочные, так и долгосрочные преимущества.

1.4. ДИЗАЙН ОБЕСПЕЧИВАЕТ ВЫСОКОЕ КАЧЕСТВО ПО

Буду с вами откровенен: хороший дизайн требует времени и усилий. Он не бесплатный. Внедрение дизайнерского мышления в повседневную работу очень важно, но решающее значение имеет независимый дизайнерский шаг перед написанием (или переписыванием) кода.

Планирование системы поможет выявить участки, которые представляют риск и содержат уязвимости пользовательской информации. Оно поможет увидеть, какие части системы могут быть узкими местами для производительности или отдельными точками сбоя.

Вы сэкономите время и деньги, упростив, объединив или разделив фрагменты системы. Глядя на отдельные компоненты, вы не поймете, выполняют ли другие компоненты аналогичную работу. Обзор системы в целом позволяет принимать обоснованные решения относительно дальнейшего пути.

1.4.1. Соображения по дизайну ПО

Часто ПО пишут для «пользователя», но кто это? Человек, использующий продукт, частью которого является ПО, или человек, пытающийся разработать дополнительные характеристики ПО? Часто вы являетесь единственным пользователем вашего ПО! Тогда вы можете лучше выявить качества, которые хотите сконструировать.

Ниже приведено несколько часто встречающихся аспектов оценки свойств ПО для различных вариантов использования:

- *Скорость* — программа выполняет работу так быстро, как только может.
- *Целостность* — данные, используемые или созданные ПО, защищены от повреждения.
- *Ресурсы* — ПО эффективно использует дисковое пространство и пропускную способность сети.
- *Безопасность* — пользователи могут читать и записывать данные, доступные только после авторизации.

В дополнение вот некоторые часто встречающиеся результаты, которые вы как разработчик, возможно, захотите получить:

- *Слабая сопряженность* — компоненты ПО не находятся в сложной зависимости друг от друга.

- *Интуитивность* — разработчики могут понять суть ПО и то, как оно работает, прочитав код.
- *Гибкость* — разработчики могут адаптировать ПО к связанным или аналогичным задачам.
- *Расширяемость* — разработчики могут добавлять или изменять один аспект ПО, не затрагивая другие аспекты.

Достижение этих результатов часто сопряжено с издержками. Например, принятие обязательств по повышению безопасности ПО, скорее всего, замедлит его разработку. Это увеличит ваши расходы, и вы захотите продать свое приложение по более высокой цене. Эффективное планирование и понимание компромиссов между результатами поможет вам минимизировать издержки для вас и ваших потребителей.

Языки программирования обычно не решают эти вопросы непосредственно — они просто предоставляют инструменты, с помощью которых разработчик занимается обслуживанием. Например, *высокоуровневые языки*, такие как Python, позволяют разработчикам писать на языке, близком к естественному, что обеспечивает некоторую защиту с точки зрения повреждения памяти. Python также поощряет использование эффективных типов данных с помощью своего синтаксиса (глава 4).

Тем не менее существует еще много работы, которую мы можем делать своими силами, потому что даже Python не способен предсказать все пути, которыми разработчики могут все испортить. Именно здесь поможет тщательный дизайн и продумывание системы в целом.

1.4.2. Органически выращенное ПО

В отличие от фермерских продуктов, органически выращенное ПО вредно для здоровья. В контексте разработки софта система, которая органически выросла с течением времени, скорее всего, созрела для рефакторинга. *Рефакторинг кода* — это процесс обновления кода для его более оптимальной проработки и отражения последних передовых практик. Он связан с повышением производительности, сопроводимости или удобочитаемости кода.

Как следует из названия «органически выращенное», такое ПО стало организмом, укомплектованным нервной системой и собственным разумом. Возможно, в него были вживлены (и не раз) куски другого ПО, где-то там гниют методы, которые не использовались много лет, скорее всего, есть одна функция, которая выполняет около 150 % работы. Время для рефакторинга такой системы бывает трудно выбрать, но оно наступает задолго до того момента, когда вы кричите: «Жив еще, курилка!»

Пример такого явления показан на рис. 1.3. Процесс регистрации на веб-сайте электронной коммерции включает в себя несколько важных шагов:

1. Определить наличие товара на складе.
2. На основе цены товара рассчитать промежуточный итог.
3. На основе региона покупки рассчитать:
   - налог;
   - расходы на доставку и обработку.
4. На основе текущих стимулирующих акций рассчитать скидки.
5. Рассчитать итоговую сумму.
6. Обработать платеж.
7. Выполнить заказ.

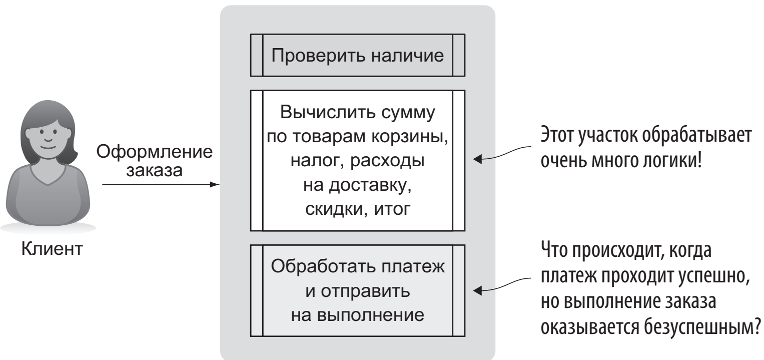


Рис. 1.3. Система электронной коммерции, которая выросла органически

В этой системе некоторые шаги четко разделены. Неплохо! Но есть и неровный участок посередине. Похоже, что вся логика, связанная с ценой, происходит в одном большом куске. Если в этом процессе есть дефект, то будет трудно понять, *на каком именно* шаге он происходит. Если цена окажется неправильной, придется просеять много кода, чтобы выяснить причину. Обработка платежей и их выполнение тоже смешаны в кучу, так что при наличии не вовремя выявленной ошибки вполне возможна ситуация, когда вы успешно обработаете платеж, но так и не выполните заказ. Это может вызвать недовольство клиента.

Хорошим началом на пути к повышению надежности этого рабочего потока станет вычленение его логических шагов (рис. 1.4). Если каждый шаг обрабатывается своей собственной службой, то служба для отдельного шага должна заниматься только одним заданием. Служба инвентаризации — отслеживать наличие товара. Служба ценообразования — знать его стоимость и налог. Изоляция шагов друг от друга снизит вероятность того, что каждый из них будет страдать от дефектов.

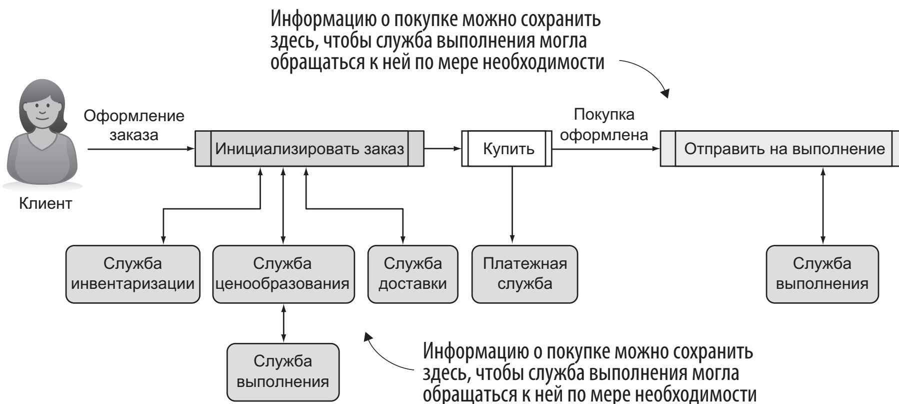


Рис. 1.4. Возможный вид тщательно спланированной системы электронной коммерции

Дизайн позволяет увидеть места, где существующие части системы могут быть разложены на более простые. Идея *декомпозиции* является лишь одним из инструментов, который мы подробнее рассмотрим в следующих главах. Имейте в виду, что рефакторинг исходного кода и редизайн

никогда не завершаются. Однако усвоив некоторые технические приемы, вы обнаружите, что со временем эти задачи будут решаться проще и быстрее. Будьте начеку и распознавайте возможности для улучшения существующего кода!

1.5. КОГДА НУЖНО УДЕЛЯТЬ ВРЕМЯ ДИЗАЙНУ

Мы склонны фокусировать усилия на создании нового ПО. Но по мере роста проектов мы забываем о реализации рабочего кода, который скорее создает проблемы, чем представляет ценность. Так проект накапливает технический долг, то есть требует дополнительной работы для сохранения продуктивности.

Чем чаще корявый фрагмент кода встает на пути и чем труднее с ним справиться *в эти моменты*, тем дольше нужно наводить порядок. Попробуйте найти проблему заранее.

Продуманный заранее дизайн ПО сэкономит время и избавит от головной боли в будущем. Когда ПО является гибким настолько, что его можно распространять на новые варианты использования, с ним приятно работать, поэтому обдумывайте написание каждой строки кода, чтобы поддержать продуктивность на высоком уровне. Мне нравится думать об этом как о технической *инвестиции* — вложениях ради последующего финансового возврата.

Возможно, вы сталкивались с проблемой в вычислительных *фреймворках* — крупных библиотеках кода, которые обеспечивают строительные блоки для реализации вашей цели (например, улучшения внешнего вида веб-сайта или создания нейросети для обнаружения лиц на видео). Фреймворк должен быть гибким в части обработки разных вариантов использования и расширяемым, чтобы вы могли дописать новую функциональность в старый код. Разработчики Python создали многочисленные вычислительные фреймворки: запросы для выполнения HTTP-вызовов, Flask и Django для веб-разработки, pandas для анализа данных и т. п. В каком-то смысле большинство кода — фреймворк. Изучите его заранее, чтобы не чинить препятствий самому себе.

Дизайн ПО — это инвестиция, которая помогает адаптировать программу к потребностям разработчиков и потребителей без больших накладных расходов и разочарований. Помните, что важна частота использования или обновления кода — тратить недели на совершенствование скрипта, применяемого пару раз в течение срока жизни проекта, неэкономично.

1.6. НОВЫЕ НАЧИНАНИЯ

Если вы зададитесь целью быть внимательнее к дизайну, улучшения быстро окажутся вам не по плечу. Возникнет так много всего, что нужно узнать и сделать, что попытки управлять всем этим сразу не доставят удовольствия. Необходимо постепенно внедрять в свою работу концепции дизайна, пока они не станут частью вашего мышления. В этой книге я буду вводить небольшие наборы понятий в каждую главу, и вы сможете в любое время вернуться к отдельным главам, чтобы закрепить знания.

1.7. ДИЗАЙН ДЕМОКРАТИЧЕН

Возможно, раньше вам *приходилось* работать над проектами самостоятельно, например во время учебы. В случае крупных проектов такое случается нечасто. В компаниях, разрабатывающих ПО для бизнеса, над одним продуктом могут работать десятки разработчиков, имеющих уникальный опыт. Разнообразие точек зрения хорошо отражается на надежности системы.

В ваших интересах прислушаться к мнению других разработчиков, особенно на ранних стадиях написания кода. Вы сможете выбрать из многочисленных вариантов самый подходящий и повысить свою продуктивность.

Если вам не выпала честь работать с активной командой разработчиков, то с особенностями совместного ПО вас могут познакомить проекты с открытым исходным кодом. Ищите дискуссии, где разработчики расходились (конструктивно!) во мнениях о какой-то задаче, и смотрите,

какие соображения вступали в игру. Мыслительный процесс, который ведет к решению, часто важнее самого решения. Способность рассуждать и обсуждать скорее поможет вам преодолеть трудности, чем знание конкретного алгоритма.

1.7.1. Хладнокровие

При написании ПО можно легко увлечься. Вспомните периоды своего энтузиазма. Вам не терпелось увидеть код в рабочем состоянии, и вы не могли сидеть спокойно и писать идеальный код.

При работе с небольшим скриптом или при выполнении некоторой предварительной работы цикл быстрой обратной связи может быть полезен для сохранения продуктивности. Я часто делаю подобную работу в цикле чтения, вычисления и печати (*read-eval-print loop*, REPL) на Python.

ИНТЕРАКТИВНАЯ СРЕДА REPL

Интерактивная среда REPL — это то, что скрывается за цепочкой символов `>>>`, когда вы набираете `python` в терминале. Она читает то, что вы набираете на клавиатуре, оценивает это, выводит результат и ждет, когда все это повторится (в цикле). Многие языки предоставляют среду REPL, поэтому разработчики имеют возможность интерактивно тестировать несколько строк кода.

Но будьте осторожны: в какой-то момент быстрое написание строки кода и наблюдение за тем, как она изменяет выходные данные, становится утомительной. Вы захотите написать более длинный или более долгосрочный код в файл и выполнить его с помощью интерпретатора. У каждого есть свой порог: я обычно упираюсь в свой, когда хочу использовать повторно строку кода, которую написал раньше, и это 15 строк назад в истории.

Пример в листинге 1.1 показывает, как преобразовать словарь данных, который увязывает штаты США с их столицами. Чтобы составить список всех столиц в алфавитном порядке, нужно:

1. Получить значения городов из словаря.
2. Отсортировать значения городов.

Листинг 1.1. Получение столиц штатов в алфавитном порядке

```
>>> us_capitals_by_state = {          ◄── Словарь, который увязывает названия штатов
    'Alabama': 'Montgomery',              с заглавных букв
    'Alaska': 'Juneau',
    ...
}
>>> capitals = us_capitals_by_state.values()   ◄── Только названия столиц
dict_values(['Montgomery', 'Juneau'])
>>> capitals.sort()                   ◄── Опа! Этот объект не имеет типа list,
Traceback (most recent call last):        поэтому нет никакого метода sort
  File "<stdin>", line 1, in <module>
AttributeError: 'dict_values' object has no attribute 'sort'
>>> sorted(capitals)                  ◄── Новый (отсортированный) список использует функцию
['Albany', 'Annapolis', ...]              sorted, которая принимает любой итерируемый объект
```

Эта задача была не так уж плоха, возникла только одна проблема. Но по мере роста проекта и расширения масштабов изменений полезно сделать шаг назад.

Продуманные планы экономят время в долгосрочной перспективе, вы же не будете делать два шага вперед и один шаг назад. Планирование на ранней стадии сформирует привычку подмечать возможности для рефакторинга. Когда я нахожусь в таком режиме, то обычно переключаюсь на написание своего кода в реальном модуле Python, даже если все еще пишу довольно короткий скрипт. Это побуждает меня немного притормозить и держать в уме более крупную цель.

Теперь представьте, что вам понадобился список столиц штатов в нескольких контекстах: в регистрационной форме, форме доставки и форме выставления счетов. Во избежание повтора вычисления можно обернуть это вычисление в функцию и вызывать ее всякий раз, когда нужно.

Листинг 1.2. Обертывание логики столицы штата в функцию

```
def get_united_states_capitals():   ◄── Фрагмент кода из листинга 1.1 в функции
    us_capitals_by_state = {'Alabama': ...}
    capitals = us_capitals_by_state.values()
    return sorted(capitals)
```

Получилась многоразовая функция. Но она работает с константными данными и делает некоторые вычисления при каждом вызове. Если она вызывается часто, можно ее изменить для повышения производительности.

На самом деле эта функция вообще не нужна. Переиспользования можно достичь с помощью только одного набора вычислений и сохранения результата в константе для последующего использования.

Листинг 1.3. Рефакторинг кода проявляет более лаконичное решение

```
US_CAPITALS_BY_STATE = {'Alabama': 'Montgomery', ...}
US_CAPITALS = sorted(US_CAPITALS_BY_STATE.values())
```

Константные данные, определяемые один раз

Тоже константа, нет необходимости в функции, нужна только ссылка US_CAPITALS

Двукратное сокращение числа строк кода без ущерба для удобочитаемости дает дополнительное преимущество.

Процесс, который мы прошли от постановки задачи до ее решения, представляет собой дизайн. Скоро вы обнаружите, что можно определить области, требующие улучшения, на ранней и более ранней стадии, а затем станете перед написанием кода чертить высокоуровневые диаграммы сложных фрагментов ПО. Конечно, не все так работают, решайте сами.

Если вам захотелось все бросить и начать свой проект заново, держитесь! Читая эту книгу, вы увидите, что процессы дизайна и рефакторинга ПО не только взаимосвязаны, но и являются двумя сторонами одной медали. Выполнение одного часто означает выполнение другого, и оба они непрерывны. К тому же ничто (и никто) не совершенно, поэтому возвращайтесь к коду почаще, особенно когда пошли трения.

А сейчас расслабьтесь. Впереди много всего интересного.

1.8. КАК ПОЛЬЗОВАТЬСЯ ЭТОЙ КНИГОЙ

Конечно, эту книгу лучше всего прочитать от корки до корки. Я структурировал части так, чтобы более поздний материал отражал идеи раннего. В части III каждая глава строится на базе проекта, который вы начнете

в главе 6. Но не стесняйтесь пролистывать или пропускать главы, с материалом которых уже знакомы, с оговоркой, что время от времени вам, возможно, понадобится вернуться к более ранней главе.

Идеи или практики из большинства глав можно легко внедрить в вашу повседневную работу. Если появится глава, идеи которой вы найдете особенно ценными, тщательно обдумайте их, а затем читайте дальше.

Помните, что код примеров и упражнений находится в репозитории GitHub этой книги (https://github.com/daneah/practices-of-the-python-pro), а также что бо́льшая часть исходного кода предназначена для проверки вашей собственной работы после выполнения упражнения. Используйте предоставленный код, если вы застряли или хотите сравнить решения, но сначала попробуйте выполнить упражнения самостоятельно.

Приятного написания кода!

ИТОГИ

- Python имеет такой же вес в сложных корпоративных проектах, как и другие основные языки программирования.
- Python имеет одну из самых быстрорастущих пользовательских баз из всех языков программирования.
- Дизайн — это не только внешний вид, а процесс, которому вы следуете, чтобы добраться до цели.
- Дизайн — это *инвестиция*, которая потом вознаградит вас чистым и гибким кодом.
- Необходимо создавать ПО с учетом разнообразной аудитории.

Часть II

Основы проектирования

Базисом эффективного программного обеспечения является продуманный дизайн, и в процессе разработки вы обнаружите, что одни и те же концепции встречаются вам снова и снова. Часть II осветит основы дизайна ПО и подготовит вас к тонкостям крупных проектов. Вы узнаете, как организовывать код, сделать его эффективнее и проверить, что он работает так, как вы ожидаете.

Попробуйте связать новые для вас сведения с описанными здесь концепциями своими силами. Частое повторение основ дизайна ПО поможет вам сделать их частью вашей повседневной работы.

2 Разделение ответственности

В этой главе:

- ✓ Использование средств Python для организации и разделения кода.
- ✓ Выбор способа и времени разделения кода на четко различимые фрагменты.
- ✓ Уровни гранулярности в разделении кода.

Краеугольным камнем *чистого кода* является разделение разнообразных форм его поведения на малые управляемые отрезки. Чистый код подразумевает, что вы не держите в голове всю связанную с ним информацию, когда обсуждаете его. Короткие отрезки кода с четко выраженными намерениями являются большим шагом в верном направлении, но эти отрезки не должны разбиваться произвольно. Необходимо разделить их по *зонам ответственности*.

ОПРЕДЕЛЕНИЕ

Зоной ответственности (concern) я называю совокупность правил отдельной области знаний, с которой имеет дело ПО. Эти правила могут варьироваться по сложности от вычисления квадратного корня до управления платежами в системе электронной коммерции.

В этой главе я расскажу об инструментах, встроенных в Python для разделения ответственности в вашем коде, а также о философии, которая помогает принимать решения, как и когда их использовать.

ПРИМЕЧАНИЕ

Если вы еще не установили Python на своем компьютере, то сделайте это, чтобы отслеживать код из книги (рекомендации по установке — в приложении в конце книги). Я подожду вас здесь. Получить полный исходный код примеров и упражнений можно в репозитории книги на GitHub (https://github.com/daneah/practices-of-the-python-pro).

2.1. ОРГАНИЗАЦИЯ ПРОСТРАНСТВА ИМЕН

Как и многие языки программирования, Python разделяет фрагменты кода с помощью *пространств имен*. При выполнении программы он отслеживает все известные пространства имен и информацию, доступную в них.

Пространства имен полезны в нескольких отношениях:

- По мере роста ПО несколько понятий будут нуждаться в одинаковых или идентичных именах. Пространства имен помогают снизить неопределенность и сохранить ясность, к какому понятию имя относится.

- По мере роста ПО становится экспоненциально труднее узнавать, какой код уже присутствует в кодовой базе. Пространства имен помогают делать обоснованные догадки о том, где может располагаться код, если он уже существует.
- При добавлении нового кода в крупную кодовую базу существующие пространства имен могут указывать на его соответствие старым или новому пространствам имен.

Пространства имен настолько важны, что они включены в «Дзен языка Python» в качестве последнего утверждения (если вы не знакомы с Дзеном, то попробуйте запустить интерпретатор Python и набрать `import this`).

> *Пространства имен — отличная штука! Будем делать их больше!*
>
> *Дзен языка Python*

Имена всех переменных, функций и классов, которые вы когда-либо использовали в Python, находились в том или ином пространстве имен. Имена, такие как `x`, `total` или `EssentialBusinessDomainObject`, являются ссылками на что-то. `x = 3` означает «назначить значение `3` имени `x`», то есть можно в дальнешем ссылаться на `x`. Так называемая «переменная» — это имя, которое ссылается на значение, хотя в Python имена могут ссылаться на функции, классы и др.

2.1.1. Пространства имен и инструкция import

Когда вы впервые открываете интерпретатор Python, *встроенное* пространство имен заполняется встроенной начинкой, например функциями без префикса, такими как `print()` и `open()`, которые можно сразу использовать в любом месте кода. Вот почему в Python знаменитая легкая инструкция `print('Здравствуй, мир!')` стала мемом *Just Works* (просто работает).

В отличие от других языков, в Python не нужно явно создавать пространства имен, но ваша структура кода повлияет на то, какие пространства имен создаются и как они будут взаимодействовать. Так, при создании модуля для него автоматически будет создано отдельное

пространство имен. В самом простом случае модулем является файл .py, содержащий некоторый код. Файл с именем sales_tax.py, например, является модулем sales_tax:

```
# sales_tax.py

def add_sales_tax(total, tax_rate):
    return total * tax_rate
```

Каждый модуль имеет глобальное пространство имен, к которому код в модуле может свободно обращаться. Ни во что не вложенные функции, классы и переменные находятся в глобальном пространстве имен модуля:

```
# sales_tax.py

TAX_RATES_BY_STATE = {
    'MI': 1.06,
    # ...
}

def add_sales_tax(total, state):
    return total * TAX_RATES_BY_STATE[state]
```

TAX_RATES_BY_STATE находится в глобальном пространстве имен модуля

Код в модуле может использовать TAX_RATES_BY_STATE без всякой суеты

Функции и классы в модуле также имеют локальное пространство имен, обращаться к которому могут только они:

```
# sales_tax.py

TAX_RATES_BY_STATE = {
    'MI': 1.06,
    ...
}

def add_sales_tax(total, state):
    tax_rate = TAX_RATES_BY_STATE[state]
    return total * tax_rate
```

tax_rate находится только в локальной области видимости для add_sales_tax ()

Код в add_sales_tax() может использовать tax_rate без особых усилий

Модуль, который хочет использовать переменную, функцию или класс из другого модуля, должен импортировать их в свое глобальное пространство имен. Импортирование — это перемещение имени в нужное пространство имен из другой части кода.

```
# receipt.py

from sales_tax import add_sales_tax

def print_receipt():
    total = ...
    state = ...
    print(f'TOTAL: {total}')
    print(f'AFTER TAX: {add_sales_tax(total, state)}')
```

Функция add_sales_tax добавляется в глобальное пространство имен receipt

add_sales_tax по-прежнему знает о TAX_RATES_BY_STATE и tax_rate из своего пространства имен

Таким образом, для того чтобы ссылаться на переменную, функцию или класс в Python, одно из следующих утверждений должно быть истинным:

- Это имя находится во встроенном в Python пространстве имен.
- Имя находится в глобальном пространстве имен текущего модуля.
- Имя находится в текущей кодовой строке локального пространства имен.

Приоритет для конфликтующих имен работает в обратном порядке: локальное имя будет переопределять глобальное имя, которое будет переопределять встроенное имя. Запомните, что наиболее специфичным для текущего кода является определение, которое используется сейчас (рис. 2.1).

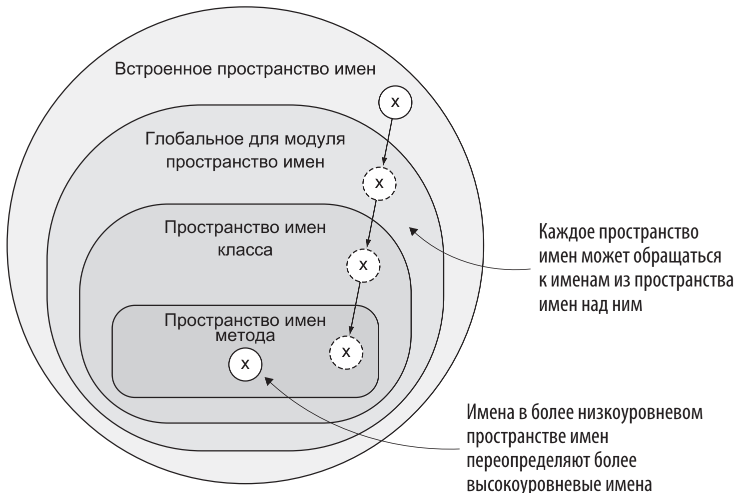


Рис. 2.1. Специфика пространств имен

Возможно, вы встречали ошибку NameError: name 'my_var' is not defined[1]. Она означает, что имя my_var не найдено ни в одном из пространств имен, известных коду. То есть, скорее всего, вы ни разу не передавали переменной my_var значение либо передали его где-то еще и должны его импортировать.

Модули представляют собой отличный способ начать дробить код. Если у вас есть один длинный файл script.py с кучей несвязанных функций в нем, попробуйте разбить эти функции на модули.

2.1.2. Лики импортирования

Синтаксис импортирования в Python на первый взгляд кажется прямолинейным, но есть несколько способов выполнения импорта, и каждый из них приводит к тонким различиям в информации, вносимой в пространство имен. Ранее вы импортировали функцию add_sales_tax() из модуля sales_tax в модуль receipt:

```
# receipt.py

from sales_tax import add_sales_tax
```

Эта инструкция добавляет функцию add_sales_tax() в глобальное пространство имен модуля receipt. Хорошо, но предположим, что вы добавляете еще десять функций в модуль sales_tax и хотите использовать их все в receipt. Если вы продолжите идти по тому же пути, то в итоге получите что-то вроде этого:

```
# receipt.py

from sales_tax import add_sales_tax, add_state_tax, add_city_tax,
➥ add_local_millage_tax, ...
```

Существует альтернативный синтаксис, который немного улучшает ситуацию:

```
# receipt.py

from sales_tax import (
    add_sales_tax,
```

[1] ОшибкаИмени: имя 'my_var' не определено

```
    add_state_tax,
    add_city_tax,
    add_local_millage_tax,
    ...
)
```

Все равно не очень складно. Если вам нужна масса функциональности из другого модуля, то вы можете импортировать этот модуль полностью:

```
# receipt.py

import sales_tax
```

Эта инструкция добавляет весь модуль `sales_tax` в текущее пространство имен, и на его функции можно ссылаться с помощью префикса `sales_tax.`:

```
# receipt.py

import sales_tax


def print_receipt():
    total = ...
    locale = ...
    ...
    print(f'AFTER MILLAGE: {sales_tax.add_local_millage_tax(total,
          locale)}')
```

Этот вариант позволяет избежать длинных инструкций `import` и, как вы увидите в следующем разделе, коллизий пространств имен.

ПРЕДУПРЕЖДЕНИЕ

Python позволяет импортировать все имена из модуля в укороченной форме с помощью инструкции `from themodule import *`. Заманчиво использовать эту форму вместо добавления префикса к именам с помощью `themodule.` на протяжении всего кода, но, *пожалуйста, не делайте этого!* Импорт с подстановочным знаком может приводить к коллизиям имен и затруднять отладку проблем, поскольку вы не увидите конкретные импортируемые имена. Придерживайтесь явных импортов!

2.1.3. Пространства имен предотвращают коллизии

Чтобы получить текущее время в программе Python, можно импортировать функцию `time()` из модуля `time`:

```
from time import time
print(time())
```

Результат будет примерно таким:

```
1546021709.3412101
```

`time()` возвращает текущее Unix-время[1]. Модуль `datetime` тоже содержит что-то с именем `time`, но делает нечто другое:

```
from datetime import time
print(time())
```

На этот раз вы должны увидеть вот такой результат:

```
00:00:00
```

Это определение `time` на самом деле является классом, и его вызов возвращает экземпляр класса `datetime.time`, который по умолчанию равен полуночи. Что происходит, когда вы импортируете их оба?

```
from time import time
from datetime import time
print(time())   ◄—— Это какое определение time?
```

В случаях двусмысленности Python использует самое последнее определение, о котором он знает. Если вы импортируете имя `time` из одного места, а затем импортируете другое имя `time` из другого места, то компилятор будет знать только о последнем. Без пространств имен трудно определить, на какое именно `time()`ссылается код, и по ошибке можно применить неправильное имя. Это веская причина для импортирования модулей целиком и добавления к ним префиксов.

---

[1] https://ru.wikipedia.org/wiki/Unix-время

```
import time
import datetime
now = time.time()                 ◄── Понятно, какое time имеется в виду
midnight = datetime.time()        ◄── Ссылка на это определение time тоже уникальна
```

Иногда столкновения имен трудно избежать, даже с теми инструментами, которые вы видели до сих пор. Если вы создадите модуль с тем же именем, что и модуль, встроенный в Python или из сторонней библиотеки, и вам нужно будет использовать их оба в одном модуле, то понадобится больше огневой мощи. К счастью, она совсем рядом, на расстоянии всего одного ключевого слова `as`, которое нужно назначить как псевдоним имени во время импортирования:

```
import datetime
from mycoollibrary import datetime as cooldatetime
```

Теперь модуль `datetime` доступен, как и ожидалось, и сторонний `datetime` доступен как `cooldatetime`.

Не переопределяйте встроенную в Python функциональность без веской причины и избегайте использования тех же имен, что и во встроенных модулях, если не собираетесь их заменять. Не зная всей стандартной библиотеки (я вот точно не знаю!), вы рискуете сделать это случайно. Тогда можно назначать своему модулю новое имя везде, где вы импортируете его в другие модули. Но еще лучше переименовывать модуль и обновлять любые ссылки на него во всем коде, чтобы импорт оставался согласованным с файловым именем модуля.

ПРИМЕЧАНИЕ

Большинство интегрированных сред разработки (IDE, integrated development environment) выдадут предупреждение, когда вы переопределите имя встроенного в Python модуля, чтобы вы не зашли слишком далеко.

Теперь вы готовы импортировать все, что нужно, без проблем. Помните, что префиксы имен модулей (подобные `time.` и `datetime.`) полезны в долгосрочной перспективе из-за риска коллизий пространств имен.

Когда вы столкнетесь с коллизиями, сделайте глубокий вдох и уверенно переработайте свои инструкции импорта или же создайте псевдоним и продолжайте путешествие!

2.2. ИЕРАРХИЯ РАЗДЕЛЕНИЯ В PYTHON

Один из способов отличить зоны ответственности — следовать философии Unix «делать что-то одно, и делать это хорошо».[1] Когда конкретная функция (или класс) работает по правилам, связанным с одной-единственной зоной ответственности, то можно улучшить ее независимо от кода. Но если формы поведения дублируются и смешиваются в коде, то обновить конкретное поведение без последствий для остальных форм поведения будет трудно. Например, функции на веб-сайте могут опираться на информацию от текущего аутентифицированного пользователя. Если они проверяют аутентификацию и сами получают информацию о пользователе, то абсолютно все они должны быть обновлены при изменении сведений об аутентификации. Это кропотливая работа.

Python поддерживает более широкий подход к разделению ответственности, чем пространства имен, а именно иерархию гранулярности. Нет жестких правил, насколько глубоко она должна распространяться. Иногда имеет смысл вызвать функцию, которая вызывает функцию, которая тоже вызывает функцию. Помните, что цель разделения ответственности состоит в том, чтобы сгруппировать похожие действия вместе и изолировать непохожие друг от друга.

Следующие разделы охватывают структурные инструменты организации кода и разделения ответственности. Если вы уверенно чувствуете себя с функциями и классами, переходите к разделу 2.2.3.

2.2.1. Функции

Если вы не слишком хорошо разбираетесь в *функциях*, вспомните урок математики. Математические функции — это формулы, обозначаемые

---

[1] https://ru.wikipedia.org/wiki/Философия_Unix

(не в питоновском синтаксисе) как `f(x) = x^2 + 3`, которые отображают входы в выходы. Получая на входе `x = 5`, указанная функция возвращает `f(5) = 5^2 + 3 = 25 + 3 = 28`. В программировании функции играют ту же роль. При заданном наборе входных переменных функция выполняет вычисление или преобразование и возвращает результат на выходе.

Это определение приводит к идее, что функции в ПО в основном должны быть краткими. Длинную функцию, которая делает слишком много, трудно охарактеризовать и назвать. Функция `f(x) = x^2 + 3` является квадратичной функцией от `x`, но как дать имя функции `f(x) = x^5 + + 17x ^ 9 - 2x + 7`? Смешение слишком большого числа концепций приводит к сложному для понимания коду.

Малые функции являются одним из первых инструментов, к которому можно обратиться при попытке разложить свой код на части. Функция обертывает несколько строк кода и дает им ясное название для последующей ссылки. Создание функции не только проясняет происходящее, но и позволяет использовать код многократно по мере необходимости. Если вы ввели `open()` для чтения файла или `len()` для получения длины списка, значит, вы использовали функциональность Python, достаточно важную, чтобы ее обернуть и дать ей имя.

Процесс разбиения задачи на малые управляемые фрагменты называется декомпозицией. Представьте гриб, разрушающий упавшее дерево. Он превращает сложные молекулы древесины в более простые материалы, такие как азот и углекислый газ, которые вернутся в экосистему. А код можно разложить на функции и затем многократно использовать их в экосистеме ПО (рис. 2.2).

Предположим, вы создаете веб-сайт для фанатов «Трех балбесов»[1]. На главной странице нужно отрекомендовать балбесов: Ларри, Керли и Мо. Имея список имен и название труппы, код должен произвести строковое значение `'Три балбеса: Ларри, Керли и Мо'`. Первоначальная реализация может выглядеть так:

```
names = ['Ларри', 'Керли', 'Мо']
message = 'Три балбеса: '
```

[1] https://ru.wikipedia.org/wiki/Три_балбеса

```
for index, name in enumerate(names):
    if index > 0:
        message += ', '
    if index == len(names) - 1:
        message += 'и '
    message += name
print(message)
```

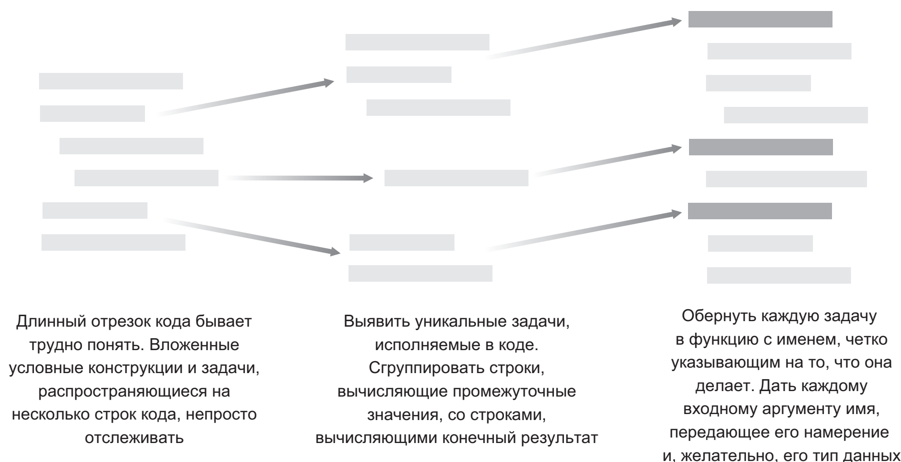


Рис. 2.2. Смысл декомпозиции

После некоторого изучения темы вы узнаете, что первоначальный состав балбесов был другим и для каждого состава нужны отдельные страницы. Вас посещает искушение добавить код для выполнения той же самой работы, но уже для первого состава:

```
names = ['Мо', 'Ларри', 'Шемп']
message = 'Три балбеса: '
for index, name in enumerate(names):
    if index > 0:
        message += ', '
    if index == len(names) - 1:
        message += 'и '
    message += name
print(message)

names = ['Ларри', 'Керли', 'Мо']
message = 'Три балбеса: '
for index, name in enumerate(names):
```

```
    if index > 0:
        message += ', '
    if index == len(names) - 1:
        message += 'и '
    message += name
print(message)
```

Это работает, но код был не очень ясен с самого начала, и теперь их стало *два*! Извлечение логики, знакомящей с составом участников, в функцию уменьшает дублирование и дает коду имя, уточняя то, что он делает:

```
def introduce_stooges(names):
    message = 'Три балбеса: '
    for index, name in enumerate(names):
        if index > 0:
            message += ', '
        if index == len(names) - 1:
            message += 'и '
        message += name
    print(message)

introduce_stooges(['Мо', 'Ларри', 'Шемп'])
introduce_stooges(['Ларри', 'Керли', 'Мо'])
```

Извлеченная функция берет символьные имена в качестве параметра

С одной и той же функцией можно использовать несколько наборов имен

Теперь поведение имеет четкое имя, и если у вас есть время сделать код еще чище, то можете сосредоточиться на теле функции `introduce_stooges`. Пока функция продолжает принимать список имен и выводить состав участников, который вы хотите, код работает как надо[1].

Довольный собой, вы решаете расширить страницу информацией о других известных группах. Однако начиная работать над черепашками-ниндзя[2], вы замечаете одну трудность: функция `introduce_stooges` (как вы можете догадаться) рекомендует только балбесов. Оказывается, у этой функции есть два правила:

- Рекомендация касается «Трех балбесов».
- В списке имен — балбесы.

---

[1] Для получения подробной информации об извлекающих функциях (и других ценных упражнений) я настоятельно рекомендую книгу Мартина Фаулера «Рефакторинг: улучшение проекта существующего кода» (М.; СПб.: Диалектика, 2019).

[2] https://ru.wikipedia.org/wiki/Черепашки_ниндзя

Как вы это обойдете? Можно обобщить функцию и вычленить первое правило путем извлечения названия группы («Три балбеса», «Черепашки-ниндзя» и т. д.) в качестве еще одного аргумента функции.

```
def introduce(title, names):
    message = f'{title}: '
    for index, name in enumerate(names):
        if index > 0:
            message += ', '
        if index == len(names) - 1:
            message += 'и '
        message += name
    print(message)

introduce('Три балбеса', ['Мо', 'Ларри', 'Шемп'])
introduce('Три балбеса', ['Ларри', 'Керли', 'Мо'])

introduce( 'Черепашки-Ниндзя',
    ['Донателло', 'Рафаэль', 'Микеланджело', 'Леонардо']
)

introduce('Бурундуки', ['Элвин', 'Саймон', 'Теодор'])
```

Суффикс _stooges исключен из имени функции, и внутрь передается название

Название группы передается внутрь при вызове функции

Разные группы могут рекомендоваться одной функцией (подробнее ниже)

Теперь эта функция соответствует требованиям веб-сайта: она знает только то, что у групп есть название и несколько именованных членов, и использует эту информацию для ознакомления пользователей с составом групп. Она может легко принимать новые группы по мере расширения веб-сайта. Чтобы изменить способ ознакомления пользователей с группами, вы просто перейдете к функции `introduce()`.

После декомпозиции кода на функции, скорее всего, код станет длиннее. Но если вы тщательно разложите код по зонам ответственности, давая явные имена разным понятиям, то повысите удобочитаемость. Совокупная длина кода менее важна, чем длины отдельных функций и методов.

Вернемся к функции `introduce`. Она формирует ознакомительное строковое значение из названия группы и имен и не знает, как списки имен должны быть соединены: запятыми или серийными запятыми и т. д. Мы можем извлечь этот фрагмент в самостоятельную функцию тоже.

```
def join_names(names):
    name_string = ''

    for index, name in enumerate(names):
```

Эта функция отвечает за способ соединения имен

```
        if index > 0:
            name_string += ', '
        if index == len(names) - 1:
            name_string += 'и '
        name_string += name
    return name_string

def introduce(title, names):
    print(f'{title}: {join_names(names)}')
```

Эта функция теперь знает только то, что рекомендации — это названия с последующими именами, соединенные знаком

Для кого-то это будет выглядеть излишеством — функция `introduce` не делает особо много. Но когда вы будете устранять дефекты, добавлять характеристики и тестировать код, то ощутите выгоду. Если дефект коснется способа соединения имен, вы легко отыщете ответственные за это строки кода и внесете изменения в `join_names`, не перекапывая `introduce`.

Разложение на функции, которые *сепарируют* зоны ответственности, позволяет проводить более точные изменения с меньшим влиянием на окружающий код, что экономит много времени.

Я уже упоминал, что дизайн, рефакторинг, а теперь декомпозиция и разделение ответственности — это практики, которые стоит включить в здоровый итеративный процесс разработки. Может показаться, что вы жонглируете тарелками, но по мере продвижения вперед к более крупному проекту вы обнаружите, что регулярно используете эти подходы. На долговечность и успех проекта влияет качество кода, которое зависит от тщательности, проявленной при его создании. Включайте полученные знания в свою работу в виде приправы, и скоро они станут основными ингредиентами.

Попробуйте сами

Соберите ваш опыт в извлечении функций и посмотрите, какие функции скрываются в листинге 2.1 реализации (вероятно, халтурной) игры «Камень, ножницы, бумага». Предлагаю выполнить этот код много раз, чтобы обеспечить согласованность поведения. Я извлек примерный набор функций в листинг 2.2. В качестве подсказки я разложил исходный код на шесть функций. Выбирайте по душе, но помните, что вы нацелены на функции, которые имеют только одну зону ответственности.

Листинг 2.1. Халтурный процедурный код

```
import random

options = ['камень', 'бумага', 'ножницы']
print('(1) Камень\n(2) Бумага\n(3) Ножницы')
human_choice = options[int(input('Введите число по вашему выбору: ')) - 1]
print(f'Вы выбрали {human_choice}')
computer_choice = random.choice(options)
print(f'Компьютер выбрал {computer_choice}')
if human_choice == 'камень':
    if computer_choice == 'бумага':
        print('К сожалению, бумага побеждает камень')
    elif computer_choice == 'ножницы':
        print('Да, камень побеждает ножницы!')
    else:
        print('Ничья!')
elif human_choice == 'бумага':
    if computer_choice == 'ножницы':
        print('К сожалению, ножницы побеждают бумагу')
    elif computer_choice == 'камень':
        print('Да, бумага побеждает камень!')
    else:
        print('Ничья!')
elif human_choice == 'ножницы':
    if computer_choice == 'камень':
        print('К сожалению, камень побеждает ножницы')
    elif computer_choice == 'бумага':
        print('Да, ножницы побеждают бумагу!')
    else:
        print('Ничья!')
```

Листинг 2.2. Код с извлеченными функциями

```
import random

OPTIONS = ['камень', 'бумага', 'ножницы']

def get_computer_choice():
    return random.choice(OPTIONS)

def get_human_choice():
    choice_number = int(input('Введите число по вашему выбору: '))
    return OPTIONS[choice_number - 1]

def print_options():
    print('\n'.join(f'({i}) {option.title()}' for i,
➥ option in enumerate(OPTIONS)))

def print_choices(human_choice, computer_choice):
    print(f'Вы выбрали {human_choice}')
```

```
    print(f'Компьютер выбрал {computer_choice}')

def print_win_lose(human_choice, computer_choice, human_beats,
➥ human_loses_to):
    if computer_choice == human_loses_to:
        print(f'К сожалению, {computer_choice} побеждает {human_
choice}')
    elif computer_choice == human_beats:
        print(f'Да, {human_choice} побеждает {computer_choice}!')

def print_result(human_choice, computer_choice):
    if human_choice == computer_choice:
        print('Ничья!')

    if human_choice == 'камень':
        print_win_lose('камень', computer_choice, 'ножницы', 'бумага')
    elif human_choice == 'бумага':
        print_win_lose('бумага', computer_choice, 'камень', 'ножницы')
    elif human_choice == 'ножницы':
        print_win_lose('ножницы', computer_choice, 'бумага', 'камень')

print_options()
human_choice = get_human_choice()
computer_choice = get_computer_choice()
print_choices(human_choice, computer_choice)
print_result(human_choice, computer_choice)
```

2.2.2. Классы

Код состоит из форм поведения и накапливающихся со временем данных. Вы уже видели, как оборачивать поведение в функцию, принимающую входные данные и возвращающую результат. Со временем вы начнете замечать, что некоторые функции работают в тандеме. Если вы часто передаете результат одной функции другой или несколько функций требуют одни и те же входные данные, то возможно, что целый *класс* уже ждет, когда его создадут.

Классы — это заготовки тесно связанных между собой данных и форм поведения. Их можно использовать для создания *объектов* или экземпляров класса. Данные становятся *состоянием* объекта, то есть представляют собой его *атрибуты*, поскольку *приписываются* ему. Формы поведения становятся *методами* — особыми функциями, получающими экземпляр объекта в качестве дополнительного аргумента (повсеместно именуемого разработчиками Python словом `self`). Это позволяет методам обращаться

к состоянию экземпляра или изменять его. Вместе атрибуты и методы являются *членами* класса.

Классы во многих языках содержат *конструктор* — особый метод, используемый для создания экземпляра класса. В Python чаще используется метод `__init__` (инициализатор). Когда экземпляр класса сконструирован, метод `__init__` вызывается и устанавливает начальное состояние экземпляра. Метод `__init__` принимает по крайней мере один аргумент `self` — ссылку на созданный экземпляр, а также дополнительные произвольные аргументы для задания начального состояния. Создание экземпляра класса по синтаксису напоминает функцию: вместо имени функции вводится имя класса, и аргументами функции выступают аргументы (исключая `self`) для конструктора `__init__`.

Взгляните еще раз на функции, которые вы вычленили из игры «Камень, ножницы, бумага» (листинг 2.3). Заметили, что все поведение и данные основываются на трех вариантах и выборе игрока? Некоторые функции используют одни и те же данные и связаны между собой. Значит, в этой игре есть класс, который рвется заявить о себе.

Листинг 2.3. Пересмотр кода игры «Камень, ножницы, бумага»

```
import random

OPTIONS = ['камень', 'бумага', 'ножницы']

def get_computer_choice():
    return random.choice(OPTIONS)

def get_human_choice():
    choice_number = int(input('Введите число по вашему выбору: '))
    return OPTIONS[choice_number - 1]

def print_options():
    print('\n'.join(f'({i}) {option.title()}' for i,
➥ option in enumerate(OPTIONS)))

def print_choices(human_choice, computer_choice):
    print(f'Вы выбрали {human_choice}')
    print(f'Компьютер выбрал {computer_choice}')

def print_win_lose(human_choice, computer_choice, human_beats,
➥ human_loses_to):
```

Функции используют OPTIONS для определения выбранных игроками вариантов

Для симуляции некоторые функции используют варианты выбора человека и компьютера

```
    if computer_choice == human_loses_to:
        print(f'К сожалению, {computer_choice} побеждает
                             {human_choice}')
    elif computer_choice == human_beats:
        print(f'Да, {human_choice} побеждает {computer_choice}!')

def print_result(human_choice, computer_choice):
    if human_choice == computer_choice:
        print('Ничья!')

    if human_choice == 'камень':
        print_win_lose('камень', computer_choice, 'ножницы', 'бумага')
    elif human_choice == 'бумага':
        print_win_lose('бумага', computer_choice, 'камень', 'ножницы')
    elif human_choice == 'ножницы':
        print_win_lose('ножницы', computer_choice, 'бумага', 'камень')
```

Варианты выбора человека и компьютера передаются по кругу неоднократно

Учитывая, что ответственность за сбор и вывод разных частей симуляции аккуратно разделена на функции, можно подумать о более высокоуровневых формах поведения. Выделение камня, ножниц и бумаги из других областей кода (возможно, вы создаете целую аркаду!) может быть выполнено с помощью класса, подобного показанному на рис. 2.3.

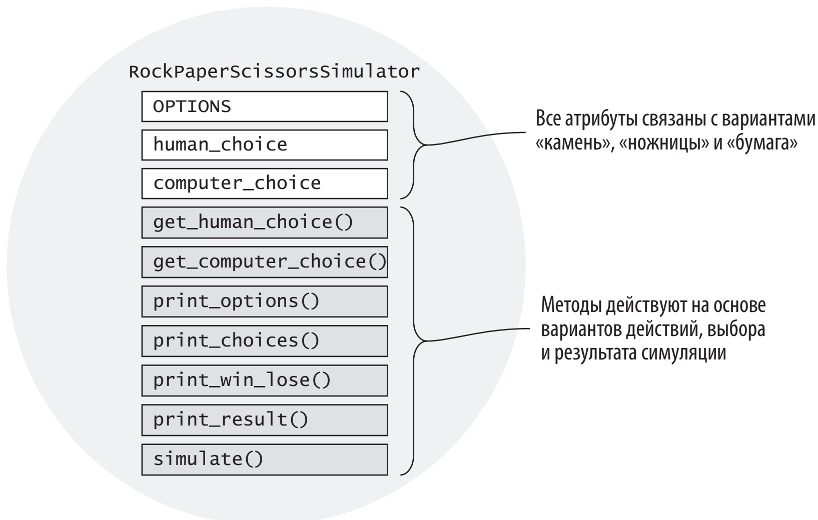


Рис. 2.3. Обертывание взаимосвязанных данных и форм поведения в классе

Обратите внимание на новый метод `simulate()`, который будет содержать код, вызывающий все остальные методы.

Начните с определения класса и перемещения в него функций в качестве методов (листинг 2.4). Помните, что методы берут `self` в качестве первого аргумента.

Листинг 2.4. Перемещение функций в класс в качестве методов

```
import random

OPTIONS = ['камень', 'бумага', 'ножницы']

class RockPaperScissorsSimulator:
    def get_computer_choice(self):        ◄── Методы нуждаются в аргументе self
        return random.choice(OPTIONS)

    def get_human_choice(self):
        choice_number = int(input('Введите число по вашему выбору: '))
        return OPTIONS[choice_number - 1]
                                                            Методы
    def print_options(self):                         с существующими
        print('\n'.join(f'({i}) {option.title()}' for i,    аргументами
➥ option in enumerate(OPTIONS)))                        по-прежнему
                                                     нуждаются в self
    def print_choices(self, human_choice, computer_choice):  ◄──
        print(f'Вы выбрали {human_choice}')
        print(f'Компьютер выбрал {computer_choice}')

    def print_win_lose(self, human_choice, computer_choice,
➥ human_beats, human_loses_to):
        if computer_choice == human_loses_to:
            print(f'К сожалению, {computer_choice} побеждает
                                 {human_choice}')
        elif computer_choice == human_beats:
            print(f'Да, {human_choice} побеждает {computer_choice}!')

    def print_result(self, human_choice, computer_choice):
        if human_choice == computer_choice:
            print('Ничья!')

        if human_choice == 'камень':
            self.print_win_lose('камень', computer_choice, 'ножницы',
                                'бумага')
        elif human_choice == 'бумага':
```

```
            self.print_win_lose('бумага', computer_choice, 'камень',
                                'ножницы')
        elif human_choice == 'ножницы':
            self.print_win_lose('ножницы', computer_choice, 'бумага',
                                'камень')
```

После перемещения функций вы можете создать новый метод `simulate` для вызова их всех. Внутри класса нужно писать `self.some_method()`, чтобы обозначить намерение вызывать метод `some_method`, заданный на классе (в отличие от какой-либо другой функции в пространстве имен). Обратите внимание, что Python передаст аргумент `self` в `some_method` автоматически. `simulate` вызовет функции для запуска симуляции:

```
...

    def simulate(self):
        self.print_options()
        human_choice = self.get_human_choice()
        computer_choice = self.get_computer_choice()
        self.print_choices(human_choice, computer_choice)
        self.print_result(human_choice, computer_choice)
```

Как видите, даже если все содержится в классе, данные опять передаются по всему классу. Зато теперь проблема внутри, и легче вносить изменения. Можно создать инициализатор, который задает необходимые для класса атрибуты, а именно `human_choice` и `computer_choice`, с дефолтным значением `None`:

```
...

    def __init__(self):
        self.computer_choice = None
        self.human_choice = None
```

Теперь методы могут обращаться к этим атрибутам с помощью аргумента `self`, а не передавать их по кругу, а вы сможете обновлять тела методов, используя `self.human_choice` вместо `human_choice`, и полностью удалить аргумент `human_choice`. Аргумент `computer_choice` получит такую же трактовку.

Код примет такой вид:

Листинг 2.5. Использование self для обращения к атрибутам

```
import random

OPTIONS = ['камень', 'бумага', 'ножницы']

class RockPaperScissorsSimulator:
    def __init__(self):
        self.computer_choice = None
        self.human_choice = None

    def get_computer_choice(self):      ◄── Методы могут устанавливать атрибуты на аргументе self
        self.computer_choice = random.choice(OPTIONS)

    def get_human_choice(self):
        choice_number = int(input('Введите число по вашему выбору: '))
        self.human_choice = OPTIONS[choice_number - 1]

    def print_options(self):
        print('\n'.join(f'({i}) {option.title()}' for i,
➥ option in enumerate(OPTIONS)))

    def print_choices(self):  ◄── Методы не должны брать атрибуты в качестве параметров
        print(f'Вы выбрали {self.human_choice}')      ◄── Методы могут читать атрибуты из аргумента self
        print(f'Компьютер выбрал {self.computer_choice}')


    def print_win_lose(self, human_beats, human_loses_to):
        if self.computer_choice == human_loses_to:
            print(f'К сожалению, {self.computer_choice} побеждает
                                 {self.human_choice}')
        elif self.computer_choice == human_beats:
            print(f'Да, {self.human_choice} побеждает
                        {self.computer_choice}!')

    def print_result(self):
        if self.human_choice == self.computer_choice:
            print('Ничья!')

        if self.human_choice == 'камень':
            self.print_win_lose('ножницы', 'бумага')
        elif self.human_choice == 'бумага':
            self.print_win_lose('камень', 'ножницы')
        elif self.human_choice == 'ножницы':
```

```
            self.print_win_lose('бумага', 'камень')

def simulate(self):
    self.print_options()
    self.get_human_choice()
    self.get_computer_choice()
    self.print_choices()
    self.print_result()
```

Потребовалось немного потрудиться, чтобы добавить `self.` в ссылки на атрибуты по всему классу, но бóльшая часть кода стала яснее. Метод `simulate` успешно склеивает другие методы, которые, в свою очередь, стали брать меньше аргументов. Еще один замечательный результат заключается в том, что код, симулирующий игру «Камень, ножницы, бумага», теперь выглядит так:

```
RPS = RockPaperScissorsSimulator()
RPS.simulate()
```

Довольно лаконично, не правда ли? Сначала вы разложили код на функции, отвечающие за разные вещи. Затем сгруппировали их в класс, чтобы сепарировать более высокоуровневые формы поведения. Теперь всю трудную закулисную работу можно легко вызвать коротким выражением. Это произошло благодаря тщательному отбору и группировке родственных данных и форм поведения.

Когда методы и атрибуты класса тесно связаны, принято говорить, что он обладает высокой *связностью*, или *когезией* (cohesion). Класс является когезивным, или связным, если все его содержимое имеет смысл как единое целое. Высокая связность приводит к четкому разделению ответственности, в то время как низкая затуманивает намерения класса слишком большим числом форм поведения. Обычно я создаю класс, когда мне ясна его связность и я вижу, из чего его можно сложить.

Когда класс зависит от другого класса, эти классы называются *сопряженными* (coupled). Если класс зависит от многих деталей другого класса настолько, что изменение одного из них требует изменения другого, то эти классы *тесно* сопряжены. Тесная сопряженность требует много времени

на управление цепными реакциями на изменения. Поэтому стремитесь к *слабой* сопряженности (глава 10).

Набор классов с высокой связностью служит во многом той же цели, что и набор четких функций. Он проясняет намерения, помогает ориентироваться в существующем коде и вносить в него дополнения быстро и просто без лишнего кордебалета.

2.2.3. Модули

Вы уже усвоили основы создания модулей в Python, например, файл .py является модулем. Я уже говорил о том, когда его нужно создавать, но давайте вернемся к этой теме.

Если бóльшая часть кода располагается в одной гигантской процедурной массе внутри script.py и вы, как и я, не в силах выучить код наизусть, то, возможно, вы уже извлекли из него ряд функций и классов.

Теперь код хорошо разделен на именованные функции, классы и методы, но все они по-прежнему располагаются в script.py. Рано или поздно минимальная структура, предоставляемая одним файлом, будет недостаточна для разумного хранения всего кода. Вы не запомните, находится ли искомая функция в строке 5 или в строке 205. Необходимо разбиение кода на категории.

Зоны ответственности удобно разделять в модулях. Подумайте, какими должны быть эти категории, примерьте разные варианты. И помните, что самый чистый код — это код, который вы не пишете, ведь каждая строка привносит дополнительную когнитивную нагрузку. На втором месте после *отсутствующего* кода стоит *хорошо организованный* код.

Модули создают дополнительную структуру вокруг кода, заявляя: «Содержащийся в нас код всецело касается статистики!». Для статистических вычислений применяйте `import statistics`. Если то, что вам нужно, еще не создано, то по крайней мере у вас будет место, куда это разместить. Может ли 500-строчный файл script.py предоставлять такие возможности? Да, но недолго.

2.2.4. Пакеты

Я похвалил модули за способность аккуратно разбивать код на части. Зачем нужно что-то еще?

Напомню, что разделение ответственности является иерархией, поэтому допускает коллизии имен. Предположим, ваш фанатский веб-сайт стал популярным и теперь вам нужна база данных (БД) и страница поиска, чтобы его отслеживать. Вы уже написали record.py — модуль для создания записей БД, а также query.py — модуль для запроса к БД:

```
.
├─ query.py
└─ record.py
```

Теперь нужно написать модуль для создания поисковых запросов. Как вы его назовете? Имя search_query.py (поисковый запрос) могло бы подойти, но тогда имело бы смысл для ясности переименовать query.py в database_query.py (запрос к БД):

```
.
├─ database_query.py
├─ record.py
└─ search_query.py
```

Если два модуля конфликтуют по имени или понятию, нужна дополнительная структура. *Пакеты* добавляют эту структуру, разделяя модули на родственные группы. В Python пакет — это каталог, содержащий модули (файлы .py) и особый файл, который дает команду Python рассматривать этот каталог как пакет (`__init__.py`). Указанный файл часто бывает пустым, но его можно использовать для более сложного управления импортами. Как и файл sales_tax.py, который становится модулем sales_tax, каталог ecommerce/ становится пакетом ecommerce.

Для модулей БД `database` и поиска `search` имеет смысл создать пакет `database` и пакет `search`. Тогда префиксы `database_` и `search_` для модулей будут избыточными и могут быть удалены.

> **ПРЕДУПРЕЖДЕНИЕ**
>
> Термин «пакеты» также относится к сторонним библиотекам Python, которые можно установить из каталога пакетов Python PyPI (Package Index). В этой книге я сделаю все возможное, чтобы не допустить неоднозначности терминов, но имейте в виду, что некоторые ресурсы не проводят между этими понятиями различия.

Вы можете развернуть иерархию кода в пакет, создающий удобную для чтения и перемещения структуру. Каждый пакет обращается к высокоуровневому участку зоны ответственности, и каждый модуль в пакете управляет меньшей зоной. Внутри каждого модуля классы, методы и функции дополнительно уточняют разные части приложения.

```
.
├─ database
│  ├─ _init_.py
│  ├─ query.py
│  └─ record.py
└─ search
   ├─ _init_.py
   └─ query.py
```

Если раньше вы писали бы `import query` для того, чтобы использовать *модуль* запросов к БД, то теперь вместо этого вам нужно будет импортировать его из *пакета* `database`. Вы можете написать `import database.query`, что потребует от вас предварять имена из этого модуля префиксом `database.query.`, либо написать `from database import query`. Если вы используете код БД только в конкретном модуле, то последний вариант лучше. Но если вам нужно использовать в модуле новый код поискового запроса и одновременно код БД, то вы должны устранить двусмысленность имен с помощью префиксов:

```
import database.query
import search.query
```

Используйте синтаксис `from` с псевдонимом каждого модуля:

```
from database import query as db_query
from search import query as search_query
```

Поскольку псевдонимы бывают многословными или совершенно запутанными, прибегайте к ним только по необходимости во избежание коллизий имен.

Вы можете вкладывать пакеты в процесс, аналогичный созданию исходного пакета. Создайте каталог с файлом `__init__.py` и разместите модули или пакеты внутри:

```
.
└ math
  ├ _init_.py
  ├ statistics
  │  ├ _init_.py
  │  ├ std.py
  │  └ cdf.py
  ├ calculus
  │  ├ _init_.py
  │  └ integral.py
  └  ...
```

В этом примере весь математический код находится в пакете math, каждый подраздел которого имеет свой подпакет с модулями. Если вы хотите посмотреть на код для вычисления интеграла, то представьте, что он находится в math/calculus/integral.py. Этот аспект пакетов — способность перемещаться туда, где код, скорее всего, будет располагаться, — становится бесценным по мере расширения проекта.

Импорт модуля `integral` работает так же, как и раньше, с дополнительными префиксами, чтобы добраться до интересующего модуля:

```
from math.calculus import integral
import math.calculus.integral
```

Обратите внимание, что `from math import calculus.integral` работать не будет. Можно импортировать только полный пунктирный путь с помощью `import ...` либо одиночное имя, используя `from ... import ...`.

ИТОГИ

- Разделение ответственности является главным ключом к пониманию кода, и многие идеи проектирования возникают непосредственно из этого принципа.
- Функции извлекают именованные понятия из процедурного кода. Чистота и разделение являются главными целями извлечения, а многократное использование — это вторичное преимущество.
- Классы группируют в объекте тесно связанные друг с другом данные и формы поведения.
- Модули группируют связанные друг с другом классы, функции и данные, сохраняя при этом разделение независимых зон ответственности. Явное импортирование кода из других модулей позволяет понять, что и где используется.
- Пакеты помогают создавать иерархию модулей, которая помогает в именовании и обнаружении кода.

3 Абстракция и инкапсуляция

В этой главе:

- ✓ Понимание сути абстракции в крупных системах.
- ✓ Инкапсуляция связанного между собой кода в классы.
- ✓ Использование инкапсуляции, наследования и композиции на языке Python.
- ✓ Понимание стилей программирования на языке Python.

Вы уже видели, что организация кода в функции, классы и модули — это отличный способ разделить *зоны ответственности*, но еще эти методы подходят для разделения *сложных участков* кода. Поскольку трудно постоянно помнить каждую деталь программы, предлагаю использовать абстракцию и инкапсуляцию для создания уровней гранулярности в коде, а о деталях вспоминать только по желанию.

3.1. ЧТО ТАКОЕ АБСТРАКЦИЯ?

О чем вы думаете, когда слышите слово *абстрактный*? У меня в голове сразу возникает какая-нибудь картина Джексона Поллока или скульптура Кальдера. Абстрактное искусство характеризуется свободой от конкретной формы и часто только подразумевает предмет. То есть *абстракция* — это процесс взятия чего-то конкретного и снятия с него специфики. В мире разработки ПО она именно такая!

3.1.1. Черный ящик

Когда вы разрабатываете ПО, его фрагменты полноценные. Например, созданную функцию можно использовать по назначению снова и снова, не задумываясь, как она работает. Функция превращается в черный ящик — вычисление или поведение, которое «просто работает», — его не нужно открывать и перепроверять (рис. 3.1).

Рис. 3.1. Трактовка рабочей программы как черного ящика

Предположим, вы проектируете систему обработки естественного языка, которая определяет отзыв о товаре как положительный, негативный или нейтральный. Такая система включает много шагов (рис. 3.2):

1. Разложить отзыв на предложения.
2. Разложить каждое предложение на слова или словосочетания, обычно именуемые лексемами, или *токенами*.
3. Выполнить *лемматизацию*, то есть определить корни слов.

4. Определить грамматическую структуру предложения.
5. Рассчитать полярность содержимого, сравнив его с тренировочными данными, помеченными вручную.
6. Вычислить совокупную магнитуду полярности.
7. Идентифицировать отзыв как положительный, негативный или нейтральный.

Каждый шаг в рабочем потоке сентиментного анализа состоит из большого числа строк кода. Свернув код в такие понятия, как «разложить на предложения» и «определить грамматическую структуру», вы сможете точнее его отслеживать и делиться информацией об отдельных шагах рабочего потока. Такая идея называется абстрагированием реализации.

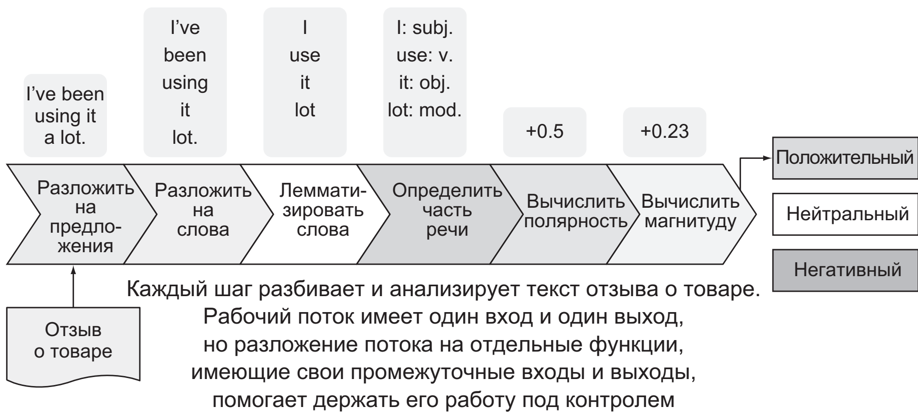


Рис. 3.2. Решение о том, каким является отзыв о товаре: положительным, отрицательным или нейтральным

Из главы 2 вы узнали, как выявлять зоны ответственности и извлекать их в функции. Абстрагирование поведения в функцию позволяет свободно менять то, *как* эта функция вычисляет результат, при условии, что входные и возвращаемые типы данных остаются неизменными. То есть если вы обнаружите дефект или более быстрый или точный способ выполнения вычисления, то сможете поменять поведение без последствий для остального кода и получите гибкость во время итераций.

3.1.2. Абстракция подобна луковице

Как видно из рис. 3.2, каждый шаг рабочего потока представляет собой некий код низкого уровня. Однако некоторые из этих шагов, такие как определение грамматической структуры, являются весьма сложными. Сложный код живет за счет *слоев* абстракции: низкоуровневые служебные подпрограммы поддерживают малые формы поведения, которые, в свою очередь, поддерживают более сложные формы поведения. По этой причине написание и чтение кода в больших системах часто похожи на чистку лука: открываются более мелкие, более плотно упакованные внутрь фрагменты кода (рис. 3.3).

Рис. 3.3. Абстракция работает, выстраиваясь в слои сложности

Самые малые формы поведения, которые используются снова и снова, сидят в нижних слоях и не нуждаются в частых изменениях. Большие концепции, бизнес-логика и сложные движущиеся части появляются по мере того, как вы идете дальше; они меняются чаще из-за меняющихся требований, но они по-прежнему пользуются малыми формами поведения.

Обычно вначале пишется одна длинная процедурная программа, которая выполняет свою работу. Она прекрасно работает во время прототипирования, но показывает плохую сопроводимость: вынуждает

прочитать 100 строк кода, чтобы найти что-либо. Введение абстракции языковыми средствами помогает выявить релевантный код. В Python такие средства, как функции, классы и модули, помогают абстрагировать поведение. Рассмотрим, как функция помогает справиться с первыми двумя шагами рабочего потока сентиментного анализа.

Во время отработки кода в листинге 3.1 вы, возможно, заметите, что он выполняет похожую работу дважды — разложение строки на предложения похоже на разложение предложения на слова. Если одна операция выполняется с разными входными данными, следует выделить поведение в самостоятельную функцию.

Листинг 3.1. Процедура разложения абзаца на предложения и лексемы

```
import re

product_review = '''Это прекрасное молоко, но товарная
линейка, похоже, ограничена спектром имеющихся цветов.
Я смог найти только белый цвет.'''   ← Отзыв о товаре в форме строки

sentence_pattern = re.compile(r'(.*?\.)(\s|$)', re.DOTALL)   ← Совпадает со всеми предложениями, оканчивающимися точкой
matches = sentence_pattern.findall(product_review)   ← Отыскивает все предложения в отзыве
sentences = [match[0] for match in matches]   ← findall возвращает список пар (предложение, пробел)

word_pattern = re.compile(r"([\w\-']+)([\s,.])?")   ← Совпадает с отдельными словами
for sentence in sentences:
    matches = word_pattern.findall(sentence)
    words = [match[0] for match in matches]   ← Получает все слова каждого предложения
    print(words)
```

Работа по поиску предложений (`sentences`) и слов (`words`) аналогична, причем строится на сопоставлении с паттерном. Но не забудьте о логистике: результат функции `findall` загромождает код, из-за чего намерения кода ясны не сразу.

Как абстракция поможет улучшить разбор предложений? Функция чуть-чуть его упростит. В следующем листинге сопоставление с паттерном абстрагируется в функцию `get_matches_for_pattern` (получить совпадения для паттерна).

> **ПРИМЕЧАНИЕ**
>
> В реальной обработке естественного языка настолько трудно разложить текст на предложения и слова, что в ПО используется *вероятностное моделирование*. Оно задействует крупный корпус входных тестовых данных для определения вероятной правильности того или иного результата. И результат не всегда единственный! Естественные языки многосложны, и компьютеры понимают их с трудом.

Листинг 3.2. Рефакторизованный разбор предложения

```
import re


def get_matches_for_pattern(pattern, string):   ◄ Новая функция для выполнения сопоставления с паттерном
    matches = pattern.findall(string)
    return [match[0] for match in matches]


product_review = '...'

sentence_pattern = re.compile(r'(.*?\.)(\s|$)', re.DOTALL)
sentences = get_matches_for_pattern(   ◄ Теперь вы можете попросить функцию сделать трудную работу
    sentence_pattern,
    product_review,
)

word_pattern = re.compile(r"([\w\-']+)([\s,.])?")
for sentence in sentences:
    words = get_matches_for_pattern(   ◄ Можете использовать эту функцию повторно, когда захотите
        word_pattern,
        sentence
    )
    print(words)
```

В обновленном коде разбора отзыва о товаре стало ясно, что указанный отзыв раскладывается на фрагменты. С хорошо поименованными переменными и четким, коротким циклом `for` двухэтапная структура процесса тоже приобрела ясность. Ваш коллега сможет прочитать главный код, только если углубится в работу функции `get_matches_for_pattern`.

Абстракция подарила программе чистоту и возможность переиспользования.

3.1.3. Абстракция упрощает

Хочу подчеркнуть, что абстракция облегчает понимание кода, скрывая детали функциональности, пока вы не захотите узнать больше. Этот прием используется при написании технической документации, а также в процессе разработки интерфейсов для взаимодействия с библиотеками кода.

Представьте отрывок из книги. В нем есть много предложений, похожих на строки кода, также могут встречаться незнакомые слова. В ПО это может быть строка, которая делает что-то новое или непривычное. Когда вы находите такие слова в книгах, то обращаетесь к словарю или комментарию в самой книге.

Но еще проще — абстрагировать связанные между собой фрагменты кода в функции, которые четко констатируют то, что они делают, как в листингах 3.1 и 3.2. Функция `get_matches_for_pattern` получает совпадения для заданного паттерна из строки. Однако до обновления намерение кода было не столь ясным.

> **СОВЕТ**
>
> Python позволяет добавлять дополнительный контекст в модуль, класс, метод или функцию с помощью литералов документирования `docstring`. *Литералы документирования* — это специальные строки в начале конструкций, которые рассказывают читателю (а также некоторому автоматизированному ПО) о характере поведения кода. Больше о литералах документирования — в Википедии (https://en.wikipedia.org/wiki/Docstring).

Абстракция уменьшает *когнитивную нагрузку* и объем ваших умственных затрат на постижение неизвестного, оставляя вам больше времени на поддержание качества ПО!

3.1.4. Декомпозиция обеспечивает возможность абстракции

Как я уже упоминал в главе 2, декомпозиция представляет собой разделение на компоненты. В разработке это означает объединение секций кода, делающих одно и то же, в функции. Фактически это также относится к обсуждению темы дизайна и рабочего потока из главы 1. Основная мысль в том, что софт, написанный мелкими частями, работающими в тандеме, удобнее в сопровождении, чем написанный одним крупным кодом. Рисунок 3.4 показывает, как огромная система может быть разложена на достижимые задачи.

Рис. 3.4. Разложение на гранулярные компоненты облегчает понимание

Видите, как компоненты уменьшаются слева направо? Пытаться создать что-то большое, как слева, — это все равно что упаковывать дом в транспортный контейнер. А проектировать как справа — это формировать каждую комнату в маленькие ящички, которые можно перемещать. Декомпозиция помогает обрабатывать большие идеи малыми инкрементами.

3.2. ИНКАПСУЛЯЦИЯ

Инкапсуляция является базисом объектно-ориентированного программирования. Она продвигает декомпозицию еще на один шаг вперед: в то время как декомпозиция группирует связанные между собой операции в функции, инкапсуляция группирует связанные между собой функции и данные в более крупную конструкцию. Эта конструкция действует для внешнего мира как барьер (или капсула). Какие конструкции доступны в Python?

3.2.1. Конструкции инкапсуляции в Python

Чаще всего инкапсуляция в Python делается с помощью класса. В классах функции становятся *методами*, то есть получают входные данные, которые являются либо экземпляром класса, либо самим классом.

В Python *модули* тоже являются формой инкапсуляции. Они группируют несколько связанных между собой классов и функций. Например, модуль, занимающийся взаимодействием по HTTP, может содержать классы для запросов и откликов, а также служебные функции для разбора URL-адресов. Большинство файлов `*.py` — модули.

Самая большая инкапсуляция в Python — это *пакет*. Пакеты инкапсулируют родственные модули в каталог и часто распределяются из каталога пакетов Python (PyPI) для установки и многократного применения другими пользователями.

Взгляните на рис. 3.5. Части корзины покупок разложены на зоны ответственности и не зависят друг от друга в выполнении задач. Любое

их сотрудничество координируется на более высоком уровне корзины покупок, которая изолирована внутри приложения и непосредственно получает необходимую информацию. Инкапсулированный код будто окружен крепостной стеной, где функции и методы — это подъемные мосты для входа и выхода.

Рис. 3.5. Разложив систему на малые части, вы можете инкапсулировать формы поведения и данные в изолированные фрагменты. Инкапсуляция побуждает вас уменьшать ответственность любой заданной части кода во избежание сложных зависимостей

Какие из этих фрагментов, по вашему мнению, будут:

- Методом?
- Классом?
- Модулем?
- Пакетом?

Три наименьших фрагмента — расчет налога, расчет расходов на доставку и вычитание скидки — по всей видимости, будут методами внутри класса, представляющего корзину покупок. Система электронной коммерции имеет достаточную функциональность, чтобы быть пакетом, потому что

корзина является только одной частью этой системы. Разные модули внутри пакета могут возникать в зависимости от того, насколько тесно они связаны. Но как они работают вместе, если каждый из них окружен стеной?

3.2.2. Ожидания приватности в Python

Крепостная стена инкапсуляции подразумевает *приватность*. Классы могут иметь *приватные* методы и данные, доступные только экземплярам класса. В свою очередь, с *публичными* методами и данными (*интерфейсом* класса) другие классы могут взаимодействовать.

Вместо строгой поддержки приватности Python доверяет разработчикам, следующим общепринятой договоренности: методы и переменные, предназначенные для использования только внутри класса, имеют префикс из символа подчеркивания. Но сторонние пакеты в своей документации часто громко заявляют о том, что такие методы могут меняться от версии к версии и на них не следует опираться явно.

Из главы 2 вы узнали о связи между классами и о стремлении к слабой сопряженности. Чем от большего числа методов и данных из другого класса зависит конкретный класс, тем более *сопряженными* эти классы становятся, что затрудняет внесение изменений.

Абстракция и инкапсуляция работают вместе, группируя связанную между собой функциональность и скрывая незначительные для других части. Такое «сокрытие информации» позволяет быстро изменять внутренности как класса, так и всей системы.

3.3. ПОПРОБУЙТЕ САМИ

Я бы хотел, чтобы вы сейчас немного попрактиковались в инкапсуляции. Предположим, вы пишете код для приветствия новых клиентов онлайн-магазина, вкладывая в него все гостеприимство и желание привлечь внимание. Напишите модуль `greeter`, который содержит один класс `Greeter`, имеющий три метода:

1) `_day(self)` — возвращает текущий день (например, воскресенье);
2) `_part_of_day(self)` — возвращает «утра», если текущий час не превышает 12:00, «дня» — если текущий час находится в периоде 12:00 – 17:00, и «вечера» — после 17:00;
3) `greet(self, store)` — с учетом названия магазина `store` и результата предыдущих двух методов выводит сообщение в форме:

```
Здравствуйте, добро пожаловать в <магазин>!
Желаем вам приятного <дня> <части дня>!
Дарим вам купон на скидку 20 %!
```

Методы `_day` и `_part_of_day` — приватные (с символом подчеркивания), поскольку единственной функциональностью, которую класс `Greeter` должен выставить наружу, является `greet`. Это помогает инкапсулировать внутренние компоненты класса `Greeter`, так что его единственным публичным делом будет выполнение приветствия.

СОВЕТ

Вы можете использовать `datetime.datetime.now()`, чтобы получить объект `datetime` с текущей датой/временем, используя атрибут `.hour` для времени суток и `.strftime ('%A')` — для дня недели.

Как все прошло? Ваше решение должно выглядеть примерно так, как показано в следующем примере.

Листинг 3.3. Модуль, который генерирует приветствия для онлайн-магазина

```
from datetime import datetime


class Greeter:
    def __init__(self, name):
        self.name = name

    def _day(self):  ◄── Форматирует дату и время в название текущего дня
        return datetime.now().strftime('%A')
```

```
    def _part_of_day(self):  ◄── Определяет часть дня, основываясь на текущем часе
        current_hour = datetime.now().hour

        if current_hour < 12:
            part_of_day = 'утра'
        elif 12 <= current_hour < 17:
            part_of_day = 'дня'
        else:
            part_of_day = 'вечера'

        return part_of_day          Выводит приветствие, используя
                                    все расчетные составляющие
    def greet(self, store):  ◄──┘
        print(f'Здравствуйте, меня зовут {self.name}, и добро пожаловать
                в {store}!')
        print(f'Желаем вам приятного {self._part_of_day()}
                                    {self._day()}?')
        print('Дарим вам купон на скидку 20 %!')

...
```

Класс `Greeter` выводит требуемое сообщение, так что все отлично, верно? Но присмотритесь: `Greeter` определяет день недели и часть дня, хотя должен только приветствовать! Инкапсуляция тут не идеальна. Что же делать?

3.3.1. Рефакторинг

Инкапсуляция и абстракция нередко являются итеративными процессами. Когда вы пишете больше кода, конструкции могут терять свой первоначальный смысл, что вполне естественно. Если код взбунтовался, нужен *рефакторинг* — обновление структуры кода для более полного удовлетворения ваших потребностей. Придется поменять способы представления понятий и форм поведения. Перемещение данных и имплементаций туда-сюда является необходимой частью совершенствования кода. Это так же полезно, как и перестановка мебели.

Теперь сделайте рефакторинг кода `Greeter`, переместив методы получения информации о дне и времени из класса `Greeter` в автономные функции внутри модуля.

Функции ни разу не использовали аргумент `self`, когда были методами, поэтому будут выглядеть как раньше, но без этого аргумента:

```
def day():
    return datetime.now().strftime('%A')


def part_of_day():
    current_hour = datetime.now().hour

    if current_hour < 12:
        part_of_day = 'утра'
    elif 12 <= current_hour < 17:
        part_of_day = 'дня'
    else:
        part_of_day = 'вечера'

    return part_of_day
```

Класс `Greeter` может вызывать эти функции, ссылаясь на них непосредственно, а не через префикс `self.`:

```
class Greeter:
    ...

    def greet(self, store):
        print(f'Здравствуйте, меня зовут {self.name}, и добро пожаловать
              в {store}!')
        print(f'Желаем вам приятного {day()} {part_of_day()}')
        print('Примите купон на скидку 20 %!')
```

Теперь `Greeter` осуществляет приветствие, не ведая, откуда оно берется. Также приятно то, что функции `day` и `part_of_day` можно использовать в другом месте, если нужно, и без необходимости ссылаться на класс `Greeter`. Две выгоды в одной!

Рано или поздно вы, возможно, разработаете еще больше функциональных средств, связанных с датой и временем, и тогда будет иметь смысл рефакторизовать все эти средства в самостоятельный модуль или класс. Я часто жду до тех пор, пока несколько функций или классов не будут представлять четкую связь, но некоторые разработчики любят держать вещи по отдельности с самого начала.

3.4. СТИЛИ ПРОГРАММИРОВАНИЯ ТОЖЕ ЯВЛЯЮТСЯ АБСТРАКЦИЕЙ

Целый ряд стилей (или *парадигм*) программирования годами набирал популярность, часто вырастая из определенных потребностей бизнеса или пользовательской базы. Python поддерживает несколько стилей, и каждый из них по-своему является абстракцией. Помните, что абстракция — это раздельное хранение понятий для упрощения восприятия. Каждый стиль программирования хранит информацию и поведение немного по-разному. Ни один стиль не является единственно «правильным», но некоторые лучше других подходят к решению определенных задач.

3.4.1. Процедурное программирование

В этой и предыдущих главах я уже показал несколько примеров *процедурного программирования*. Процедурное ПО использует *процедурные вызовы*, которые мы называем функциями. Эти функции не инкапсулированы в классы, поэтому часто опираются только на свои входные данные и иногда на какое-то глобальное состояние.

```
NAMES = ['Эбби', 'Дэйв', 'Кейра']

def print_greetings():
    greeting_pattern = 'Поприветствуй {name}!'
    nice_person_pattern = '{name} — замечательный человек!'
    for name in NAMES:
        print(greeting_pattern.format(name=name))
        print(nice_person_pattern.format(name=name))
```

Автономная функция, которая опирается на глобальную переменную NAMES

Если вы в программировании новичок, то этот стиль, вероятно, покажется вам знакомым, потому что с него начинают все. Переход от одной длинной процедуры к процедуре, которая вызывает несколько функций, как правило, ощущается естественным, поэтому этот подход хорош для преподавания. Выгоды от процедурного программирования пересекаются с теми, которые мы обсудили в разделе 3.1.4, поскольку оно фокусируется на функциях.

3.4.2. Функциональное программирование

Функциональное программирование *звучит* так, как будто оно является тем же самым, что и процедурное программирование, — термин *функция* вложен в название! Но хотя и верно, что функциональное программирование в значительной степени опирается на функции как форму абстракции, его модель образа мысли совершенно иная.

Функциональные языки требуют, чтобы вы думали о программах как о композициях функций. Например, циклы `for` заменяются функциями, которые оперируют на списках. В Python вы можете написать следующее:

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    print(i * i)
```

На функциональном языке вы могли бы написать это вот так:

```
print(map((i) => i * i, [1, 2, 3, 4, 5]))
```

В функциональном программировании функции иногда принимают другие функции в качестве аргументов или возвращают их в качестве результатов. В предыдущем фрагменте кода `map` принимает анонимную функцию, которая берет один аргумент и умножает его на себя.

Python имеет ряд инструментов функционального программирования. Многие из них доступны с использованием встроенных ключевых слов, а другие импортируются из встроенных модулей, таких как `functools` и `itertools`. Функциональное программирование в Python не является предпочтительным, хотя и поддерживается. Некоторые часто встречающиеся средства функциональных языков, такие как функция `reduce`, были вынесены в `functools`.

Многие считают, что императивный питоновский способ выполнения некоторых из этих операций выглядит более чистым. Использование средств функционального программирования языка Python будет выглядеть следующим образом:

```
from functools import reduce

squares = map(lambda x: x * x, [1, 2, 3, 4, 5])
should = reduce(lambda x, y: x and y, [True, True, False])
evens = filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5])
```

Предпочтительным в Python было бы следующее:

```
squares = [x * x for x in [1, 2, 3, 4, 5]]
should = all([True, True, False])
evens = [x for x in [1, 2, 3, 4, 5] if x % 2 == 0]
```

Попробуйте каждый подход, а затем выведите переменные. Вы увидите, что они производят одинаковые результаты. Выберите наиболее понятный вам стиль.

В Python мне нравится функция `functools.partial`. Она позволяет создавать из одной функции другую, наделенную несколькими аргументами исходной функции. Это иногда выглядит более понятно, чем писать новую функцию, вызывающую исходную функцию, в особенности в тех случаях, когда общеупотребительная функция ведет себя как более конкретно именованная функция. В случае возведения чисел в степень x^2 называется *квадратом* x, а x^3 — его *кубом*. Рассмотрите `partial`:

```
from functools import partial


def pow(x, power=1):
    return x ** power


square = partial(pow, power=2)
cube = partial(pow, power=3)
```

Новая функция square, которая действует как pow(x, power=2)

Новая функция cube, которая действует как pow(x, power=3)

Использование знакомых имен для описания форм поведения очень поможет тем, кто будет читать ваш код позже.

Функциональное программирование, используемое с умом, обеспечивает ряд преимуществ в производительности по сравнению с процедурным программированием, что делает его полезным в таких дорогостоящих областях вычислений, как математика и моделирование данных.

3.4.3. Декларативное программирование

Декларативное программирование сосредоточено на объявлении параметров задачи без указания способа ее выполнения. Детали решения задачи в основном или полностью абстрагируются от разработчика. Благодаря этому вы можете повторить высокопараметрическую задачу с незначительными изменениями параметров.

Часто этот стиль программирования реализуется посредством *предметно-ориентированных языков* (DSL, domain-specific languages) — языков (или языкоподобной разметки), которые заточены под конкретные множества задач. Например, на HTML разработчики могут описывать структуру страницы, ничего не говоря о том, как браузер должен преобразовывать тег `<table>` в строки и символы на экране. Python является *общецелевым языком* и требует руководства со стороны разработчика.

Подумайте о декларативном программировании, когда заметите, что пользователи делают что-то повторяющееся, например транслируют код в другую систему (SQL, HTML и т. д.) или создают похожие объекты для многоразового использования.

Широко используемым примером декларативного программирования в Python является пакет `plotly`. Это пакет, который позволяет создавать графики из данных, описывающих его возможный тип. Пример из документации `plotly` (https://plot.ly/python/) выглядит так:

```
import plotly.graph_objects as go

trace1 = go.Scatter(   ◄── Объявляет намерение построить график рассеяния
    x=[1, 2, 3],   ◄── Объявляет форму данных оси x
    y=[4, 5, 6],   ◄── Объявляет форму данных оси y, легко сопоставимую с x

    marker={'color': 'red', 'symbol': 104},   ◄── Объявляет внешний вид маркера прямой
    mode='markers+lines',   ◄── Объявляет, что на графике будут использоваться маркеры и прямые
    text=['one', 'two', 'three'],   ◄── Объявляет текст всплывающей подсказки для каждого маркера
    name='1st Trace',
)
```

Этот код задает данные для построения графика, а также визуальные характеристики. Каждый желаемый выход *декларируется*, а не добавляется процедурно.

Для сравнения: в процедурном подходе вместо предъявления нескольких фрагментов конфигурационных данных одной функции или классу вы выполняли бы каждый шаг конфигурирования как независимую строку более длинной процедуры:

```
trace1 = go.Scatter()
trace1.set_x_data([1, 2, 3])
trace1.set_y_data([4, 5, 6])
trace1.set_marker_config({'color': 'red', 'symbol': 104, 'size': '10'})
trace1.set_mode('markers+lines')
...
```

Каждый фрагмент информации задается явно с помощью методов

Декларативный стиль обеспечивает более лаконичный интерфейс, когда бо́льшая часть конфигурирования должна производиться пользователем.

3.5. ТИПИЗАЦИЯ, НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Типизация (не путайте термин *typing* с набором текста на клавиатуре) в языке, или система типов, — это то, как язык решает задачу управления типами переменных. Некоторые языки являются компилируемыми и проверяют типы данных во время компиляции. Другие проверяют типы во время выполнения программ. Третьи выводят логически, что выражение `x = 3` имеет целочисленный тип без явного `int x = 3`.

Python — *динамически типизируемый* язык. Он выясняет типы данных во время выполнения и применяет *утиную типизацию*, название которой происходит от идиомы «если оно ходит как утка и крякает как утка, то, должно быть, это утка». В отличие от многих языков, которые не смогут скомпилировать программу, если она ссылается на неизвестный метод в экземпляре класса, Python будет пытаться вызвать этот метод во время выполнения, выдавая ошибку `AttributeError`. Благодаря этому

Python может достигать некоторой степени полиморфизма — способности языка программирования поддерживать объекты разных типов, обеспечивающих специализированное поведение через согласованное имя метода.

С приходом объектно-ориентированного программирования возникла гонка по моделированию полных систем в виде наследуемых классов. `ConsolePrinter` наследовал от `Printer`, который наследовал от `Buffer`, который наследовал от `BytesHandler`, и т. д. Некоторые из этих иерархий имели смысл, но многие приводили к жесткому коду, который трудно обновлять. Попытка внести одно изменение могла привести к массивной ряби изменений на всем протяжении вверх или вниз по дереву.

Сегодня предпочтение сместилось к размещению в объекте *композиций* форм поведения. *Композиция* в противоположность декомпозиции сводит фрагменты функциональности вместе. На рис. 3.6 показано

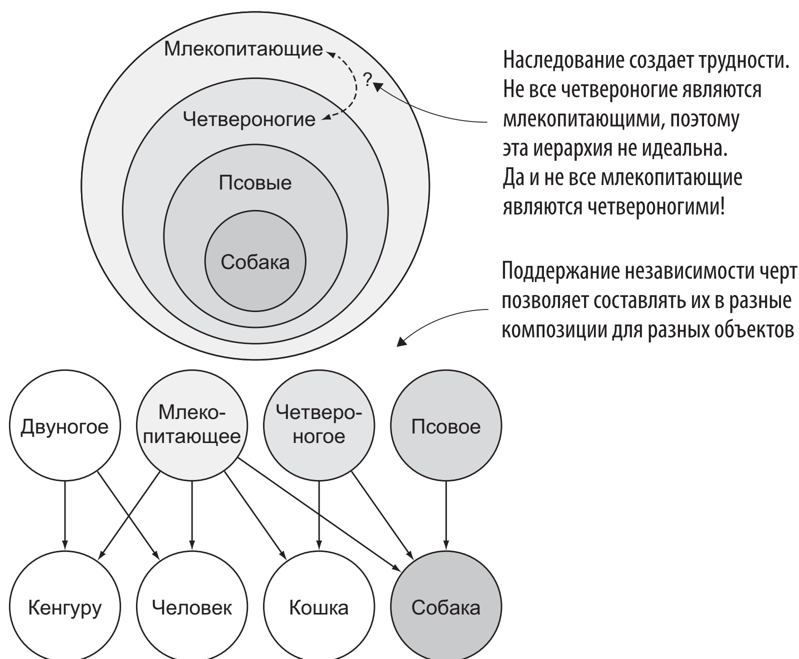


Рис. 3.6. Наследование против композиции

противопоставление жесткой структуры наследования той, в которой объекты состоят из многочисленных черт. Собака — это четвероногое млекопитающее семейства псовых. Наследование вынуждает создать из этих черт иерархию. Да, все псовые — млекопитающие, но не у всех млекопитающих четыре ноги. Композиция освобождает от ограничений иерархии без ущерба для связности между понятиями.

Композиция часто выполняется с помощью языкового средства под названием «интерфейс». *Интерфейсы* — это формальные определения методов и данных, которые конкретный класс должен имплементировать. Класс может имплементировать более одного интерфейса, транслируя повсюду, что у него есть объединение форм поведения этих интерфейсов.

О нет! В Python отсутствуют интерфейсы. Как же избежать глубокой иерархии наследования? К счастью, Python использует утиную типизацию и *множественное наследование* — наследование от произвольного числа классов, которое не поддерживается статически типизируемыми языками. Python создает что-то вроде интерфейса, который часто называют примесью, или *миксином* (mixin)[1].

Предположим, вы хотите создать модель собаки, которая может говорить и делать кувырок. Ну, и потом других животных. С использованием суффикса `Mixin` создайте класс собаки `Dog`, которая может говорить (`speak`) и кувыркаться (`roll_over`), как показано в следующем листинге.

Листинг 3.4. Множественное наследование, обеспечивающее интерфейсоподобное поведение

```
class SpeakMixin:
    def speak(self):
        name = self.__class__.__name__.lower()
        print(f'{name} говорит: "Привет!"')

class RollOverMixin:
    def roll_over(self):
        print('Сделала кувырок!')
```

Говорящее поведение инкапсулировано в SpeakMixin, чтобы показать, что оно составляется в композицию

Кувырковое поведение в RollOverMixin тоже составляется в композицию

[1] *Миксин* (или *примесь*) — это языковая концепция, которая позволяет программисту впрыскивать некий код в класс. Примесное программирование — это стиль разработки ПО, в котором функциональные единицы создаются в одном классе, а затем смешиваются с другими классами. — *Примеч. пер.*

```
class Dog(SpeakMixin, RollOverMixin):
    pass
```

Ваша собака Dog может speak, roll_over и все остальное, чему вы ее научите

Теперь, когда собака унаследовала от нескольких миксинов, вы можете проверить, знает ли она пару трюков:

```
dog = Dog()
dog.speak()
dog.roll_over()
```

Вы должны увидеть этот результат:

```
Собака говорит: "Привет!"
Сделала кувырок!
```

Тот факт, что собака знает человеческий язык, вызывает подозрение, а в остальном все отлично. В главах 7 и 8 мы еще глубже погрузимся в наследование и некоторые другие связанные с ним понятия, так что не торопитесь!

3.6. РАСПОЗНАВАНИЕ НЕПРАВИЛЬНОЙ АБСТРАКЦИИ

Распознавать ситуации, когда абстракции в существующем коде не работают, почти так же важно, как применять абстракции к новому коду. Возможно, это обусловлено тем, что новый код доказал, что абстракция не подходит для всех вариантов использования, или тем, что вы видите способ сделать код чище с помощью другой парадигмы. Как бы то ни было, забота о коде — это то, что ценится, даже если эта работа неявна для других.

3.6.1. Как корове седло

Как я уже говорил, абстракция нужна для упрощения. Если она заставляет вас рыть носом землю, только чтобы заставить что-то работать, подумайте об обновлении, чтобы устранить трение, или полностью измените подход. Я поздно понял, что иногда легче изменить среду, чем

к ней адаптироваться. Время и усилия на переписывание кода и его перепроверку, конечно, лучше потратить заранее.

Если интерфейс стороннего пакета вызывает трения и у вас нет возможности потратить время или усилия на обновление кода, создайте абстракцию вокруг этого интерфейса для своего кода. В ПО такая абстракция называется *адаптером*, и я приравниваю его к дорожному переходнику, незаменимому в аэропорту другой страны. Вы, разумеется, не можете поменять электрические переходники во Франции (никого не рассердив), и у вас нет французского переходника для ваших устройств под рукой. Так что даже если за дорожный переходник надо отдать 48 евро и вашего первенца, это выйдет дешевле, чем отыскать и купить французские источники питания для трех-четырех разных устройств. В ПО вы можете создать свой адаптерный класс, который имеет интерфейс, ожидаемый вашей программой, методы которого за кулисами выполняют вызовы несовместимого стороннего объекта.

3.6.2. Подходите ко всему с умом

Я долго разглагольствовал о написании прилизанного кода, но слишком умные решения тоже бывают болезненными. Пока вы колдуете, другие разработчики создают свои удачные решения, которые работают и без вашей единой рабочей реализации. Учитывайте частоту и влияние вариантов использования во время разработки. Часто встречающиеся варианты использования, как правило, максимально гладкие, в то время как редкие варианты бывают неуклюжими или не поддерживаются явно, когда нужны. Ваше решение должно быть *разумным в достаточной мере*, а эту цель, по общему признанию, поразить трудно.

С учетом сказанного, если что-то ощущается неудобным или громоздким, то дайте ему немного времени. Если через время это по-прежнему ощущается неудобным или громоздким, то спросите других, согласны ли они с этим. Если они говорят «нет», но неудобство или громоздкость так и ощущаются, то, вероятно, оно таковым и является. Сделайте шаг и сделайте мир чуточку лучше с помощью абстракции!

ИТОГИ

- Абстракция — это инструмент для упрощения понимания кода.
- Абстракция принимает много форм: декомпозиция, инкапсуляция, стиль программирования и наследование против композиции.
- Каждый подход к абстракции полезен, но контекст и частота варианта использования кода являются важными факторами при выборе подхода.
- Рефакторинг — это итеративный процесс. Поэтому даже рабочая абстракция подлежит пересмотру.

4 Создание дизайна для производительности

В этой главе:

- ✓ Понимание временной и пространственной сложности.
- ✓ Измерение сложности кода.
- ✓ Выбор типов данных для разных видов деятельности.

Написанный рабочий код — не конец пути. Важно, чтобы код не только выполнял задачу, но и делал это быстро. *Производительность* кода зависит от того, насколько хорошо он потребляет такие ресурсы, как память и время. ПО, которое работает на приемлемом уровне, то есть эффективно потребляет ресурсы и откликается на задачи в пределах желаемых временных рамок, считается *производительным*.

Производительность ПО влияет на нас каждый день: во время загрузки селфи в инстаграм или при анализе рынка ценных бумаг в режиме реального времени. Необходимый уровень производительности ПО часто

сводится к пользовательскому восприятию. Если что-то *ощущается* мгновенным, то, возможно, оно является достаточно быстрым.

Производительность ПО может влиять на конечный результат. Если ваше ПО требует, чтобы вы хранили что-то на диске или в БД, то минимизация объема требуемого хранилища сэкономит вам деньги. ПО, которое информирует о принятии денежных решений, может принести вам больше прибыли, если работает быстрее.

ЧЕЛОВЕЧЕСКОЕ ВОСПРИЯТИЕ

Люди воспринимают изменения, происходящие быстрее 100 мс, как мгновенные. Если они нажимают кнопку и экран отвечает через 50 мс, то они счастливы. Когда скорость отклика выходит за пределы 100 мс, задержка становится ощутимой.

Для длительных действий, таких как скачивание крупных файлов, важны обновления индикатора хода выполнения, поскольку они изменяют восприятие задержки, в результате чего процесс ощущается как более быстрый.

4.1. СКВОЗЬ ВРЕМЯ И ПРОСТРАНСТВО

Если вы почитываете статьи о высокопроизводительном ПО, то, скорее всего, сталкивались с такими словосочетаниями, как *временная сложность* и *пространственная сложность*. Термины звучат так, будто пришли из квантовой механики или астрофизики, но в мире разработки ПО им тоже нашлось свое место.

Временная и пространственная сложность — это мера того, насколько больше времени выполнения, памяти или дискового хранилища требуется проекту по мере роста объема его входных данных. Чем быстрее ваше ПО расходует время или занимает пространство, тем выше его сложность.

Сложность не претендует на статус *точной* количественной меры; скорее наоборот, она помогает вам понимать качество ПО — насколько

быстрым и большим оно будет в худшем случае. В этом разделе я научу вас интуитивно определять сложность, благодаря чему вы сможете по капельке выжимать производительность в своей работе. Правда, существует формальный процесс выявления сложности, но об этом позже.

4.1.1. Сложность немного… сложна

Не буду ходить вокруг да около и скажу прямо: измерять сложность бывает трудно, и эта процедура иногда сбивает с толку. Во время учебы она не имела для меня большого смысла — я научился тому, что знаю, только практикуясь. Будьте готовы, что у вас будет то же самое.

Виды сложности выявляются за счет процесса, который называется *асимптотическим анализом*. Он включает в себя наблюдение за кодом и определение границ его производительности в худшем случае.

> **ПРИМЕЧАНИЕ**
>
> Имейте в виду, что измерения сложности используются для сравнения способов выполнения какой-то конкретной задачи и они не очень-то полезны для сравнения несвязанных задач. Например, полезно сравнивать два алгоритма сортировки списка чисел, но вы не сможете сравнить алгоритм сортировки списка с деревом поиска. Убедитесь, что сравниваете яблоки с яблоками.

Система обозначений, используемая в асимптотическом анализе, на первый взгляд кажется загадочной, но ее составляющие легко переводятся на естественный язык. Сложность записывается как *O большое*, которое обозначает производительность анализируемого кода в худшем случае. Переводится это обозначение примерно как $O(n^2)$, или «порядок n в квадрате», где n — это число элементов входных данных, а n^2 — сложность. Эта укороченная форма означает, что «время, требующееся на то, чтобы код выполнился, увеличивается пропорционально квадрату числа элементов входных данных» (рис. 4.1). Гораздо быстрее

написать $O(n^2)$. В остальной части главы я буду использовать обозначение O большое.

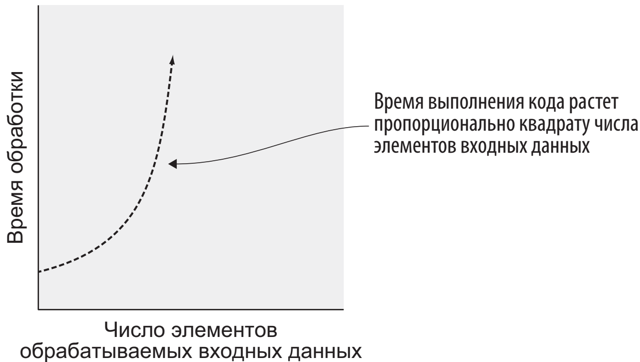


Рис. 4.1. $O(n^2)$ — это укороченная форма обозначения «O большое» для отношения $y \propto x^2$

4.1.2. Временная сложность

Временная сложность — это мера того, насколько быстро ваш код может выполнить задачу по отношению к своим входным данным. По мере увеличения числа элементов входных данных временная сложность говорит о том, в каком темпе ваш код будет замедляться. Это помогает рассуждать о том, сколько времени должна занимать задача.

Линейность

Линейная сложность — частое явление в ПО. Построение графика числа ее элементов входных данных в сопоставлении с временем производит прямую линию. То есть в математическом уравнении $y = mx + b$ значением x выступает число элементов входных данных, а y — время выполнения задачи. Задача может быть обременена накладными расходами независимо от входных данных (b, или пересечение), и каждый дополнительный элемент входных данных добавляет некое количество времени выполнения (m, или наклон)[1] (рис. 4.2).

[1] Наклон (*slope*) также именуется угловым коэффициентом, а пересечение (*intercept*) — сдвиговым коэффициентом (пересечением оси y). — *Примеч. пер.*

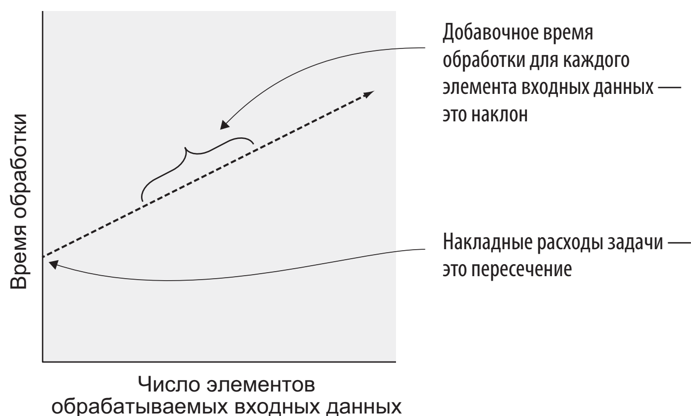


Рис. 4.2. Визуализация задачи с линейной сложностью

Такая сложность часто встречается, поскольку многие операции требуют выполнения некоторой задачи для каждого элемента в списке: печать списка имен, суммирование списка целых чисел и т. д.

По мере того как список растет, количество времени, которое компьютер должен потратить, растет пропорционально. Суммирование 1000 целых чисел занимает *примерно* вдвое меньше времени, чем суммирование 2000 целых чисел. Для n элементов эти виды деятельности «линейны относительно n» или обозначаются $O(n)$.

Вы можете опознать код, который, скорее всего, будет иметь $O(n)$ на Python, отыскав циклы `for`. Одиночный цикл над списком, множеством или другой последовательностью, по всей видимости, будет линейным:

```
names = ['Элайя', 'Бет', 'Дэвид', 'Карим']
for name in names:
    print(name)
```

Это верно даже при выполнении более чем одного шага внутри цикла:

```
names = ['Элайя', 'Бет', 'Дэвид', 'Карим']
for name in names:
    greeting = 'Привет, меня зовут'
    print(f'{greeting} {name}')
```

Это верно, даже если вы обходите в цикле один и тот же список заданное число раз:

```
names = ['Элайя',  'Бет',  'Дэвид',  'Карим']
for name in names:
    print(f'Это {name}!')

message = 'Давайте поприветствуем '
for name in names:
    message += f'{name} '
print(message)
```

Хотя вы прокручиваете список имен дважды, снова подумайте об этом в терминах уравнения прямой. Первый цикл занимает некоторое время f в расчете на элемент, и второй цикл занимает некое время g в расчете на элемент. Линейное уравнение было бы чем-то вроде $y = fx + gx + b$, что эквивалентно $y = (f + g)x + b$. Это по-прежнему прямая, даже если она более крутая.

Вот тут-то и возникает «асимптотическая» часть асимптотического анализа. Даже если какая-то конкретная деятельность может быть *круто* линейной, другие, более сложные операции все равно смогут превысить ее по величине, если элементов входных данных достаточно много (рис. 4.3).

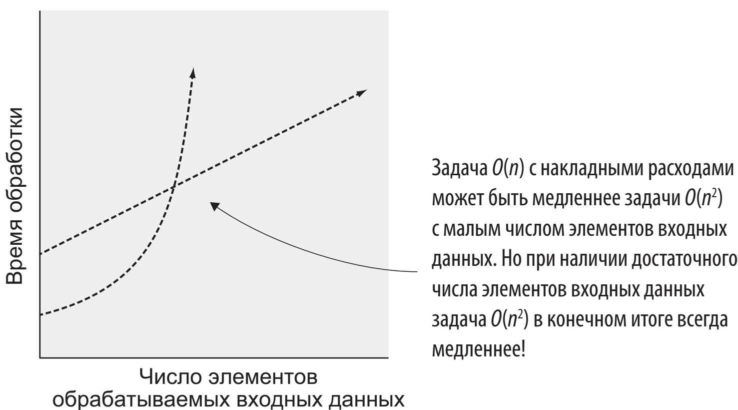


Рис. 4.3. Сложность более высокого порядка в крупных масштабах

Время растет пропорционально квадрату числа элементов выходных данных

В еще одном типе временной сложности время растет пропорционально *квадрату* числа элементов входных данных ($O(n^2)$). Это проявляется в тех случаях, когда для каждого элемента в списке нужно посмотреть на каждый второй элемент в списке. Когда вы добавляете новые элементы входных данных, код должен перебирать дополнительные элементы на каждой *такой* итерации. Увеличение во времени выполнения усугубляется.

Вы можете опознать это в коде Python по наличию вложенных циклов. Следующий код проверяет список на наличие дубликатов:

Внешний цикл перебирает каждый элемент последовательности

Внутренний цикл снова перебирает каждый элемент для каждого элемента во внешнем цикле

```
def has_duplicates(sequence):
    for index1, item1 in enumerate(sequence):
        for index2, item2 in enumerate(sequence):
            if item1 == item2 and index1 != index2:
                return True
    return False
```

Проверяет, что два разных элемента последовательности имеют одинаковое значение

ДОПОЛНИТЕЛЬНЫЕ ОБОЗНАЧЕНИЯ

Иногда бывает полезно рассчитать не только *худший* случай, но и *средний*, и *лучший*. Обозначение Ω большое» (омега) используется для анализа лучшего случая, а θ большое» (тета) — выражает, что верхняя и нижняя границы имеют заданную сложность. Расчеты с ними помогают выбрать лучший подход к выполнению задач. Сложность многих алгоритмов можно найти в интернете, к примеру, «сложность быстрой сортировки» (complexity of quicksort). Вы также можете найти временную сложность некоторых часто встречающихся операций в документации Python (https://wiki.python.org/moin/TimeComplexity).

$O(n^2)$ является для этого кода *худшим случаем*, потому что даже если только последние элементы окажутся дубликатами или если дубликатов

не существует, код все равно будет должен перебрать все элементы входных данных, прежде чем он завершится. Если первые два элемента окажутся дубликатами, код будет работать намного быстрее, потому что может немедленно остановиться, однако полезно изучить худший случай, чтобы лучше понять, на что код способен. По этой причине обозначение O всегда измеряет сложность кода для худшего случая.

Постоянное время

Идеальная сложность ($O(1)$) описывает постоянное время, которое не зависит от объема входных данных. Лучше постоянного времени только время, *ускоряющееся* по мере роста числа элементов входных данных! Постоянное время реализовано в некоторых типах данных в Python, о которых я расскажу позже.

Некоторые задачи, которые обычно были бы линейными (или хуже), можно сделать константными после предварительного вычисления. Это начальное вычисление само по себе может быть непостоянным, но если оно позволяет многим последующим шагам *стать* постоянными, то оно стоит проведения.

4.1.3. Пространственная сложность

Пространственная сложность — это мера того, как код использует дисковое пространство или память, когда растет число элементов его входных данных. Однако пространственную сложность легко проглядеть, потому что вы не всегда наблюдаете ее непосредственно.

СБОРЩИК МУСОРА

Так как вы редко управляете памятью сами, то это тоже приводит к увеличению пространственной сложности в Python. В некоторых языках необходимо явно выделять и высвобождать память, но Python использует автоматическую *сборку мусора*, которая высвобождает память, содержащую объекты, которые больше не используются работающей программой.

Иногда неэффективное использование дискового пространства поднимает свою уродливую голову только тогда, когда вы получаете всплывающее сообщение о том, что на компьютере не осталось места. Учитывайте пространство, когда пишете код, чтобы не съедать ресурсы.

Память

Обычно программы потребляют слишком много памяти, если читают крупные файлы данных целиком в память, когда от них это не требуется. Предположим, имеется текстовый файл, содержащий строку для каждого человека, живущего сегодня на Земле, и его любимый цвет. Вы хотели бы знать количество людей, которым больше всего нравится определенный цвет. Можно прочитать весь файл как список строк и работать с ним:

```
color_counts = {}

with open('all-favorite-colors.txt') as favorite_colors_file:
    favorite_colors = favorite_colors_file.read().splitlines()  ◄── Читает весь файл целиком в список строк

for color in favorite_colors:
    if color in color_counts:
        color_counts[color] += 1
    else:
        color_counts[color] = 1
```

На планете Земля много людей. Даже если файл содержит только один столбец любимых цветов и каждая строка использует 1 байт данных, размер файла будет чуть больше 7 Гб. Возможно, на вашем компьютере имеется такой объем памяти, но задача не требует, чтобы вы получили всю информацию о строках данных сразу.

В Python вы можете читать файл построчно в цикле `for`, и на каждой итерации цикла следующая строка будет *замещать* текущую строку в памяти. Попробуйте обновить код, читая по одной строке из файла за раз, и вернитесь, когда получите результат.

```
color_counts = {}

with open('all-favorite-colors.txt') as favorite_colors_file:
    for color in favorite_colors_file:  ◄── Читает только одну строку данных за раз
```

```
color = color.strip()

if color in color_counts:
    color_counts[color] += 1
else:
    color_counts[color] = 1
```

Удаляет замыкающий символ новой строки из каждой строки

Читая по одной строке за раз и выбрасывая их после того, как получите то, что нужно, вы потребляете память размером не больше самой большой строки в файле. Гораздо лучше! Пространственная сложность ушла от $O(n)$, придя к $O(1)$.

Дисковое пространство

В прошлом я сталкивался с проблемами дискового пространства в долгоживущих приложениях. Иногда их трудно увидеть сразу. Прежде чем закончится дисковое пространство, могут уйти недели или даже месяцы, так как либо программа пишет малые объемы данных за раз, либо в распоряжении имеется крупное хранилище.

Многие крупные веб-приложения используют журналы своих операций с целью их отладки или анализа. Если вы введете в код инструкцию журналирования, которая в процессе работы вызывается 1000 раз в минуту, то она начнет быстро съедать дисковое пространство. Возможно, вы захотите удалить эту строку кода, переместить ее туда, где она вызывается реже, или же улучшить свою стратегию хранения журналов.

Поиск возможностей уменьшения сложности почти всегда подразумевает рост производительности, которого не стоит ждать от изменения отдельных строк кода. Используйте анализ сложности в вашем ПО и читайте дальше, чтобы увидеть, какие средства поиска этих возможностей встроены в Python.

4.2. ПРОИЗВОДИТЕЛЬНОСТЬ И ТИПЫ ДАННЫХ

Дизайн кода строится на существующих в Python типах данных. Раздел 4.2 охватывает типы сложности и подходящие для них типы данных.

4.2.1. Типы данных для постоянного времени

Как вы помните, идеальная производительность достигается в случае постоянного времени, которое не увеличивается с увеличением числа элементов входных данных. Типы `dict` (словарь) и `set` (множество) Python демонстрируют такое поведение при добавлении, удалении и обращении к элементам. Под капотом они довольно похожи, причем главное отличие состоит в том, что *словари увязывают ключи со значениями*, тогда как *множества представляют собой коллекцию уникальных элементов*. Итерационный проход по элементам в любом из этих типов данных по-прежнему занимает время $O(n)$, поскольку оно зависит от числа элементов, содержащихся в объекте. Но извлечение конкретных элементов или проверка существования определенного элемента происходит быстро, независимо от суммарного числа элементов.

Предположим, что вместо подсчета любимых цветов вы заинтересовались, какие цвета людьми ни разу не упомянуты. Можно читать файл построчно, как и раньше, но как представить данные и проверить их на наличие конкретных значений цвета?

Попробуйте придумать решение сами, а затем сравните его со следующим листингом.

Листинг 4.1. Использование средств Python для минимизации пространства

```
all_colors = set()

with open('all-favorite-colors.txt') as favorite_colors_file:
    for line in favorite_colors_file:
        all_colors.add(line.strip())
print('Amber Waves of Grain' in all_colors)
```

Итеративный обход файла по-прежнему занимает время $O(n)$

Добавление в структуру данных set занимает время $O(1)$ и пространство $O(n)$

Принадлежность ко множеству является вопросом времени $O(1)$

Используя структуру данных `set` (множество) для хранения списка уникальных значений цвета, встречающихся в файле, вы можете проверять наличие конкретных значений цвета во множестве за константное время ($O(1)$) после цикла.

4.2.2. Типы данных для линейного времени

Списковый тип данных `list` в Python демонстрирует главным образом сложность $O(n)$ — определение членства или добавление нового элемента в произвольное место происходит медленнее в списках с бóльшим числом элементов. Добавление или удаление из *конца* списка занимает время $O(1)$. Списки полезны, когда хранящиеся элементы не поддаются уникальной идентификации.

Кортежный тип `tuple` аналогичен списку с точки зрения производительности, с ключевым отличием в том, что кортежи не могут быть изменены после их создания.

4.2.3. Пространственная сложность операций на типах данных

Теперь, когда вы знакомы с временной сложностью некоторых встроенных в Python структур данных, я научу вас паре трюков по их использованию. Все типы данных, о которых мы говорили, являются *итерируемыми* — объектами, поддерживающими итеративный перебор своего содержимого (например, в цикле `for`). Итеративный обход множества элементов почти всегда будет $O(n)$ по временной сложности, но перебор каждого элемента займет больше времени, если количество элементов растет. Но как быть с пространственной сложностью?

Все содержимое упомянутых выше типов данных хранится в памяти вместе. Если список состоит из 10 элементов, то он занимает примерно в 10 раз больше места в памяти, чем список с одним элементом (рис. 4.4). Из этого следует, что их *пространственная* сложность также равна $O(n)$. Это проблема, и проблемой также является чтение 7,6 млрд записей. Если все эти данные не нужны сразу, требуется более эффективный подход.

Введем *генераторы* — конструкции в Python, которые производят по одному значению за один раз, становясь на паузу до тех пор, пока не будет запрошено следующее значение (рис. 4.5). Это напоминает подход, который мы использовали, когда читали файл построчно. Производя

одно значение за один раз, генератор избегает хранения в памяти сразу всех значений, которые производит.

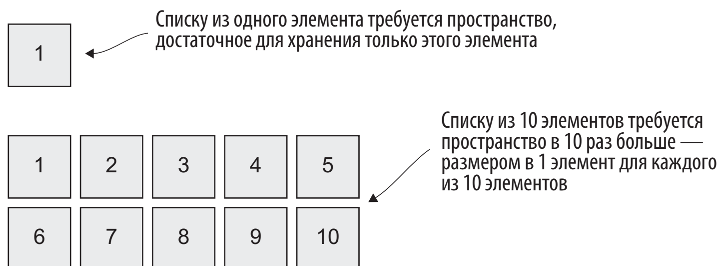


Рис. 4.4. След памяти, занимаемый списками

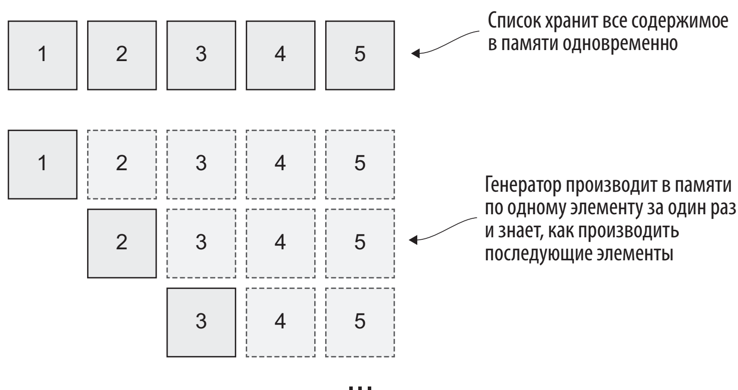


Рис. 4.5. Экономия пространства за счет генераторов

Если вы уже использовали функцию `range` в Python, это и был генератор. Функция `range` принимает аргументы, указывающие границы нужного интервала. Если бы `range` хранила все числа интервала в памяти, то код наподобие `range(100_000_000)` быстро съел бы всю память. Вместо этого `range` хранит только *границы* интервала и производит значения из него по одному за раз. Но как?

Генераторы используют ключевое слово `yield`. Произведя значение, они возвращают поток выполнения обратно вызывающему коду. То есть `yield` возвращает значение, а затем возвращает поток выполнения.

Ключевое слово `yield` работает во многом так же, как инструкция `return` языка Python, за исключением того, что оно позволяет выполнять операции *после* возврата значения. Его можно использовать для подготовки следующего значения, которое вы хотите произвести. Листинг 4.2 приближенно показывает поведение функции `range` под капотом. Обратите внимание на использование ключевого слова `yield` и приращение текущего значения *после* того, как `yield` было использовано.

Листинг 4.2. Использование ключевого слова yield для приостановки и подготовки

```
def range(*args):
    if len(args) == 1:  ◄── Разбирает аргументы с целью определения границ интервала
        start = 0
        stop = args[0]
    else:
        start = args[0]
        stop = args[1]

    current = start

    while current < stop:
        yield current  ◄── Порождает каждое значение (по одному за раз)
        current += 1  ◄── Выполняет всю необходимую настройку
                          для подготовки следующего значения
```

В этой имплементации есть паттерн, который вы будете часто встречать в генераторах:

1. Выполнить главную настройку, необходимую для получения всех значений.
2. Создать цикл.
3. Вернуть значение на каждой итерации цикла.
4. Обновить состояние для следующей итерации цикла.

Попробуйте проверить значения из интервального генератора `range` прямо сейчас. Вы можете превратить его в список с помощью `list(range(5, 10))` и двигаться вперед по одному значению за один раз, сохранив `range(5, 10)` в переменной и делая поочередные вызовы `next(my_range)`.

Напишите свой генератор, используя указанный паттерн. Ваша генераторная функция `squares` будет брать список целых чисел и производить квадрат каждого из них. Попробуйте и сравните ее со следующим листингом.

Листинг 4.3. Короткий генератор производит числа в квадрате

```
def squares(items):
    for item in items:
        yield item ** 2
```

Функция `squares` так компактна, потому что не требует подготовки или управления состоянием. Причем вместо списка вы можете передать еще один генератор. Вызов `squares(range(100_000_000))` работает так же хорошо. Эта функция будет хранить только один элемент из интервала и один результат в квадрате за раз, экономя еще больше места (рис. 4.6).

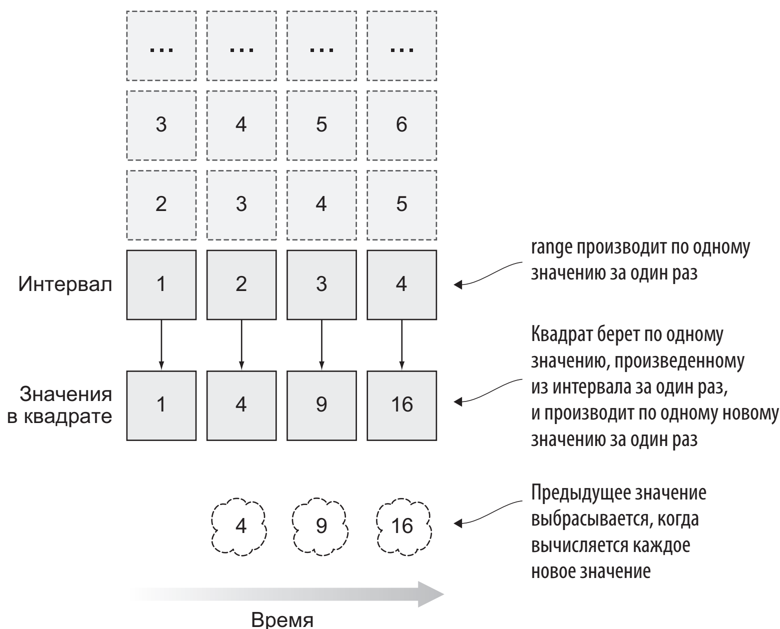


Рис. 4.6. Потребление памяти генераторами, соединенными в цепь

Рекомендую использовать генераторы над списками везде, где только можно, так как если нужно, всегда можно создать из генератора полный список прямо в памяти, написав `list(range(10000))` или `list(squares([1, 2, 3, 4]))`. Генератор экономит память и время, потому что код, потребляющий значения из генератора, не нуждается в них всех постоянно.

ЛЕНИВОЕ ОЦЕНИВАНИЕ

Идея получения по одному значению за один раз и отсутствия постоянной необходимости во всех значениях, которые можно создать, часто называется *ленивым оцениванием*. Оно *ленивое*, потому что подразумевает меньше работы, причем сделанной только по просьбе. Представьте себе, что ваши генераторы тяжело вздыхают всякий раз, когда их просят произвести (`yield`) значение.

4.3. РАБОТОСПОСОБНОСТЬ, ПРАВИЛЬНОСТЬ И БЫСТРОТА

Фраза «сделайте, чтобы это работало, затем — чтобы работало правильно, затем — чтобы работало быстро» принадлежит Кенту Беку, создателю экстремального программирования. На первый взгляд она может означать, что вы должны *сначала* написать рабочий код, затем переработать его, чтобы он был чистым и кратким, и только *потом* сделать его совершенным. Но мне нравится думать об этих трех правилах как о шагах, которые вы предпринимаете на каждой малой итерации, когда пишете код. Помните, что проектирование дизайна, реализация и рефакторинг происходят в сжатых циклах написания кода.

4.3.1. Сделайте работоспособным

Бóльшую часть времени разработчики (включая меня) стараются превратить постановку задачи или идею в код, который достигает цели. Часто задача отрабатывается с начала до конца, чтобы затем стать подверженной

рефакторингу. Напоминает дилемму о курице и яйце: *как «сделать это быстрым», если «это» еще не доделано?*

Декомпозиция, которая полезна для ПО в целом, также является инструментом для разложения задач на управляемые фрагменты. Каждая из меньших задач может быть реализована и проверена поступательно на пути к достижению более крупной цели. Кроме того, при таком подходе гораздо легче «сделать это работоспособным», ведь «это» — управляемая цель. Вы с большей готовностью набросаете идеи для вычисления скорости падающего тела, чем спроектируете движок.

4.3.2. Сделайте правильным

«Сделать работоспособным» — это как путь из пункта А в пункт Б. Если вы четко представляете себе цель задачи, то ответ на вопрос «работает ли это?» является бинарным.

«Сделать правильным» всецело касается рефакторинга. Рефакторинг направлен на повторную реализацию существующего кода более чистым или хорошо адаптированным способом без потери согласованного результата.[1]

Рефакторингу нет конца и края. Вы будете ловить себя на том, что выполняете итерацию за итерацией, когда реализуете или пересматриваете код для добавления новой функциональности. *Правило трех* Мартина Фаулера гласит, что когда вы трижды реализовали что-то схожее, то *должны* рефакторизовать код, обеспечив абстракцию для такого поведения. Мне близок этот подход, поскольку он организует баланс вокруг рефакторинга: не абстрагируй что-то сразу или даже после того, как ты это продублировал, — подожди, пока не увидишь, какие возникают варианты использования, и выбери самую подходящую абстракцию.

---

[1] Есть мнение, что если создать хорошие тесты для кода, на них можно опереться, чтобы безболезненно вносить изменения в код. Существует целый ряд фантастических текстов на эту тему, например книга Гарри Персиваля «Разработка на основе тестирования» (М.: ДМК Пресс, 2018. — *Примеч. ред.*).

Еще один аспект правильности — в использовании сильных сторон языка себе во благо. Взгляните на следующий код, который выявляет самое часто встречающееся целое число в списке:

```
def get_number_with_highest_count(counts):
    max_count = 0
    for number, count in counts.items():
        if count > max_count:
            max_count = count
            number_with_highest_count = number
    return number_with_highest_count

def most_frequent(numbers):
    counts = {}
    for number in numbers:
        if number in counts:
            counts[number] += 1
        else:
            counts[number] = 1

    return get_number_with_highest_count(counts)
```

Определяет число с наибольшим числом появлений в dict, увязывающим числа с количествами появлений

Подсчитывает количество появлений числа, чтобы увидеть, какое из них имеет самое большее количество появлений

Я сделал этот код *работоспособным*, но у Python есть несколько инструментов, позволяющих упростить его. Первый инструмент я применю к наращиванию количества. Он должен выяснить, не видели ли мы раньше число в списке, чтобы нарастить или инициализировать количество появлений числа. Я укажу встроенному словарю `defaultdict` тип значений, которые он будет хранить, чтобы по умолчанию он придерживался этого типа при обращении к новому ключу:

```
from collections import defaultdict


def get_number_with_highest_count(counts):
    max_count = 0
    for number, count in counts.items():
        if count > max_count:
            max_count = count
            number_with_highest_count = number
    return number_with_highest_count

def most_frequent(numbers):
    counts = defaultdict(int)
    for number in numbers:
```

Импортирует defaultdict из модуля collections

Словарный объект counts содержит целые числа, поэтому дефолтный тип каждого значения в defaultdict должен быть int

```
        counts[number] += 1

    return get_number_with_highest_count(counts)
```

Дефолтное значение для int равно 0, поэтому в первый раз, когда мы видим число, количество его появлений будет равно 0 + 1 = 1

Неплохо! Вы сэкономили одну строку кода, и функция стала чище. Но это еще не все. В Python также есть помощник для подсчета значений в последовательности:

```
from collections import Counter


def get_number_with_highest_count(counts):
    max_count = 0
    for number, count in counts.items():
        if count > max_count:
            max_count = count
            number_with_highest_count = number
    return number_with_highest_count


def most_frequent(numbers):
    counts = Counter(numbers)
    return get_number_with_highest_count(counts)
```

Словарь Counter тоже находится в модуле collections

Действует почти идентично словарю dict, который вы изготовили вручную

Вы сэкономили еще несколько строк кода и сделали четче функцию `most_frequent`: она считает уникальные числа и возвращает то, которое появлялось чаще других. Но как быть с функцией `get_number_with_highest_count` (получить число с наибольшим количеством появлений)? Она занимается поиском максимального значения в словаре, который увязывает числа с количеством их появлений. Python предоставляет два инструмента для ее упрощения.

Первый — это `max`. Функция `max` принимает итерируемый объект (списки, множества, словари и т. д.) и возвращает максимальное значение из этого объекта. В случае словаря функция `max` по умолчанию возвращает максимальное значение *ключей* `key`. Ключами словаря `count` являются сами числа, а не количества их появлений. `max` принимает второй аргумент `key` — функцию, которая говорит `max`, какую часть итерируемого объекта использовать.

Python будет передавать функции `key` только один аргумент: значение из итерируемого объекта. В случае словарей Python итерационно проходит по их ключам, поэтому функция, переданная аргументу `key` для `max`, будет получать только числа, а не количество их появлений. Вам нужно сообщить функции `key`, что при задании числа она должна индексировать словарь `counts` в этом числе, чтобы получить значение количества появлений. Написание отдельной функции в модуле не будет работать, потому что `counts` будет недоступен в его пространстве имен. Как это обойти?

В функциональном программировании принято передавать функции в качестве аргументов другим функциям, и иногда эти переданные функции коротки и понятны настолько, что им не требуются имена. В отличие от большинства функций, которые вы, вероятно, писали на Python, они являются *анонимными* и называются *лямбдами*. Лямбды являются функциями — принимают аргументы и возвращают значения. У них нет имен, их нельзя вызывать непосредственно, но вы можете использовать их в качестве встроенных аргументов для других функций.

В случае с функцией `get_number_with_highest_count` передайте лямбду в `max`, которая принимает число и возвращает `counts[number]`. Это решит проблему пространства имен и обеспечит желаемое поведение `max`:

```
from collections import Counter


def get_number_with_highest_count(counts):
    return max(
        counts,
        key=lambda number: counts[number]
    )


def most_frequent(numbers):
    counts = Counter(numbers)
    return get_number_with_highest_count(counts)
```

Во время итеративного перебора чисел в counts этот код использует counts[number] (количество появлений этого числа) в качестве сравниваемого значения

Коротко и ясно. Знание инструментов правда помогает сокращать код.

Разумеется, короче — не всегда значит лучше. Вы можете пойти дальше и переместить `max` непосредственно в функцию `most_frequent`, но, когда

я использую такие функции с непонятным поведением, как `max`, мне нравится иметь отдельную функцию с четким именем.

Когда вы достигли рабочего и *чистого для других* кода — вы «сделали его правильным».

4.3.3. Сделайте быстрым

Тщательная настройка производительности кода может занять столько же времени, сколько его написание. Она требует проверки типов данных и операций с учетом срочности вывода проекта на рынок. Оценивать производительность как *достаточную* придется вам. Лучшее — враг хорошего, поэтому поставить клиенту что-то ценное, но медленное лучше, чем вообще ничего не поставить.

Если вывод продукта на рынок в приоритете, установите чек-пойнты производительности, которые нужно достигнуть итеративно после первого релиза. Так вы сможете сосредоточиться на создании рабочего кода, открытого для усовершенствования. В продакшене код обнаружит неожиданные узкие места.

Приемлемый уровень производительности будет варьироваться в зависимости от ваших целей. Если вы показываете модальное окно для входа на веб-сайт после нажатия кнопки Войти, то это должно происходить мгновенно, иначе пользователи уйдут. Если же вы проектируете систему годовой отчетности, клиенты, которые хотят увидеть свои продажи, на входе могут немного подождать.

Архитектура системы — разные службы, страницы, взаимодействия и т. д. — будет информировать о производительности и на нее влиять. Более крупные системы требуют большей сетевой связи между API, БД и кэшами. Они также могут содержать процессы, происходящие вне рабочего потока пользователя, например накопление аналитических метрик в ночное время. Проверьте службы, которые выполняют задачи, аналогичные вашим, чтобы получить представление о базовом уровне, и создайте информированное ожидание на счет производительности вашего ПО, чтобы знать, к чему стремиться. Производительность крупных систем выходит за рамки кода.

По мере разрастания кода пускайте в ход знания о производительности типов данных и методов. Развивайте интуицию в отношении проблем с производительностью и вложенные циклы и огромные списки, пожирающие оперативную память, начнут бросаться в глаза.

4.4. ИНСТРУМЕНТЫ

Тестирование производительности должно быть основано на подтверждающих данных, поскольку системы с реальными пользователями неизбежно будут испытывать разное поведение: удачное сочетание неожиданных элементов данных на входе, временной координации, особенностей аппаратного обеспечения, задержек в сети и др.способствует повышению производительности системы. Поэтому ковыряние кода в надежде наткнуться на огромные выигрыши в производительности — не лучшее решение.

4.4.1. timeit

Модуль `timeit` в Python — это инструмент для тестирования времени выполнения фрагментов кода. Он может быть использован из командной строки или непосредственно в коде для большего контроля. Модуль `timeit` удобен для санитарной проверки планируемых изменений в производительности.

Давайте измерим, сколько времени нужно, чтобы просуммировать целые числа от 0 до 999. Для начала из командной строки активируйте модуль `timeit`:

```
python -m timeit "total = sum(range(1000))"
```

`timeit` выполнит код суммирования много раз и в итоге выведет некоторую статистику о времени выполнения:

```
20000 loops, best of 5: 18.9 usec per loop
```

Из этого результата можно сделать вывод, что суммирование от 0 до 999 обычно занимает менее 20 микросекунд.

Чтобы увидеть, изменится ли время при суммировании от 0 до 4999, измените команду и повторите ее:

```
python -m timeit "total = sum(range(5000))"
2000 loops, best of 5: 105 usec per loop
```

Как оказалось, суммирование целых чисел 0–4999 происходит более чем в пять раз дольше суммирования целых чисел 0–999.

Имейте в виду, что `timeit` действительно выполняет код, а реальное выполнение зависит от многих переменных, таких как уровень заряда батареи и тактовая частота процессора. Поэтому лучше проводить измерения несколько раз, чтобы оценить стабильность их результатов и последствия вносимых в них изменений. Хотя `timeit` дает количественные измерения, используйте его для качественного сравнения разных реализаций, сосредоточившись на тренде, и поймете, что именно ускоряет код.

Командный интерфейс для `timeit` великолепен, но в случае тестирования больших или сложных фрагментов кода выглядит громоздким. Если вам нужен больший контроль над предметом измерений, используйте `timeit` изнутри кода. Измеряя время выполнения конкретной порции кода (без ее группы поддержки), вычлените настроечный шаг. Тогда время, которое он требует, не будет учитываться:

```
from timeit import timeit

setup = 'from datetime import datetime'
statement = 'datetime.now()'
result = timeit(setup=setup, stmt=statement)
print(f'Took an average of {result}ms')
```


Этот код в итоге приведет к измерению времени, необходимого для вызова `datetime.now()`, без учета инструкции `import`, необходимой для его выполнения.

Предположим, что вы хотите доказать, что проверка наличия элемента во множестве выполняется быстрее, чем проверка его наличия в списке. Как бы вы это сделали, используя модуль `timeit`? Постройте свои входные данные, используя `set(range (10000))` и `list(range(10000))`, и измерьте

время выполнения задачи поиска в них числа 300. Насколько быстрее будет множество?

Модуль `timeit` несколько раз спасал меня от падения в кроличью нору, сообщая о том, что моя гипотеза об ускорении того или иного кода была ошибочной. Он поистине является хранителем времени (такой вот каламбур).

4.4.2. Профилирование ЦП

Когда вы использовали модуль `timeit`, он выполнял *профилирование* кода — анализ кода во время его работы для сбора некоторых метрик о его поведении. Модуль `timeit` измерял суммарное время выполнения кода, однако еще один проницательный способ измерения производительности состоит в профилировании центрального процессора (ЦП). Профилирование ЦП позволяет увидеть, какие *части* кода выполняют дорогостоящие вычисления и как часто они вызываются. Оно помогает понять, куда нужно обратиться в первую очередь, чтобы ускорить код.

Предположим, вы написали функцию, не слишком дорогую, но вызываемую в приложении много раз. Вы также написали функцию, которая обходится дорого, но вызывается только один раз. Если у вас есть время исправить одну из них, какую выберете? Без профилирования трудно узнать, какая функция ускорит код. Вы можете выяснить это с помощью модуля `cProfile`.

> **ПРИМЕЧАНИЕ**
>
> Если вы пытаетесь импортировать модуль `cProfile`, но получаете ошибку, то вместо него используйте модуль `profile`.

Модуль `cProfile` выводит несколько фрагментов информации о каждом методе или функции, вызываемых во время выполнения программы. Для каждого вызова он будет показывать вам:

- число произошедших таких вызовов (`ncall`);

- время, проведенное в этом вызове, не включая вещи, которые он сам вызвал (`tottime`);
- среднее время проведения в этом вызове, если он вызывался `ncall` раз (`percall`);
- кумулятивное время, проведенное в этом вызове, включая любое время, проведенное в подвызовах (`cumtime`).

Эта информация покажет как медленные вызовы (которые имеют большую величину `cumtime`), так и быстрые, но многократные. Посмотрите на программу, которая вызывает функцию 1000 раз. Исполнение вызова функции занимает произвольное количество времени до 10 миллисекунд.

Листинг 4.4. Профилирование процессной производительности программы Python

```
import random
import time


def an_expensive_function():
    execution_time = random.random() / 100
    time.sleep(execution_time)


if __name__ == '__main__':
    for _ in range(1000):
        an_expensive_function()
```

Исполнение занимает произвольное количество времени (вплоть до 10 миллисекунд)

Выполняет функцию 1000 раз

Сохраните эту программу в модуле cpu_profiling.py. Тогда вы сможете выполнить ее профилирование из командной строки с помощью `cProfile`:

```
python -m cProfile --sort cumtime cpu_profiling.py
```

При большом количестве вызовов можно ожидать, что функция, которая занимает 0–10 миллисекунд, будет занимать в среднем около 5 миллисекунд (`percall`). Вызывая ее 1000 раз (`ncalls`), можно ожидать, что она займет около 5 секунд в целом (`cumtime`). Выполните `cProfile` в программе, чтобы проверить, соответствует ли она вашим ожиданиям. Вы увидите много результатов, но сортировка по общему времени покажет, что вызовы функции `an_expensive_function` будут находиться ближе к верху:

```
$ python -m cProfile --sort cumtime cpu_profiling.py
         5138 function calls (5095 primitive calls) in 5.644 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      4/1    0.000    0.000    5.644    5.644 {built-in method builtins.exec}
        1    0.002    0.002    5.644    5.644 cpu_profiling.py:1(<module>)
     1000    0.003    0.000    5.625    0.006 cpu_profiling.py:5
➥ (an_expensive_function)
     1000    5.622    0.006    5.622    0.006 {built-in method time.sleep}
     ...
```

В этом прогоне функция `an_expensive_function` занимала в среднем около 6 миллисекунд в расчете на вызов в течение 1000 вызовов, что привело к кумулятивным 5,625 секунды, которые были потрачены внутри этой функции.

Глядя на результаты `cProfile`, вы захотите найти вызовы с высоким значением `percall` или большим скачком в `cumtime`. Эти характеристики означают, что вызов занимает бóльшую часть времени выполнения программы. Ускорение медленной функции может значительно повысить быстродействие программы, а сокращение времени, необходимого для многократного вызова функции, станет настоящим выигрышем.

4.5. ПОПРОБУЙТЕ САМИ

Рассмотрите следующий код. Он содержит функцию `sort_expensive`, которая должна сортировать список из 1000 целых чисел в интервале 0–999999, и функцию `sort_cheap`, сортирующую список из 10 целых чисел в интервале 0–999.

Алгоритмы сортировки обычно обходятся дороже, чем $O(1)$, так что функция `sort_expensive` займет больше времени, чем функция `sort_cheap`. Если выполнить каждую функцию только один раз, то `sort_cheap` наверняка выиграет. Но выполнение `sort_cheap` 1000 раз уравняет шансы двух функций на победу.

```
import random

def sort_expensive():
    the_list = random.sample(range(1_000_000), 1_000)
```

```
    the_list.sort()


def sort_cheap():
    the_list = random.sample(range(1_000), 10)
    the_list.sort()

if __name__ == '__main__':
    sort_expensive()
    for i in range(1000):
        sort_cheap()
```

Нужно выполнить профилирование кода, чтобы понять производительность. Посмотрите на поведение каждой задачи с привлечением модулей `timeit` и `cProfile`.

ИТОГИ

- Создавайте проект с учетом итеративного повышения производительности как на начальном этапе, так и на протяжении всей разработки.
- Тщательно продумывайте правильный тип данных для поставленной задачи.
- Предпочитайте генераторы спискам, если вам не нужны все значения сразу, чтобы сэкономить на потреблении памяти.
- Используйте модули `timeit` и `cProfile` (`profile`) языка Python для проверки ваших гипотез о сложности и производительности.

5 Тестирование ПО

В этой главе:

- ✓ Понимание сути тестирования.
- ✓ Использование разных подходов к тестированию приложения.
- ✓ Написание тестов с помощью фреймворка `unittest`.
- ✓ Написание тестов с помощью фреймворка `pytest`.
- ✓ Внедрение методологии разработки на основе тестирования.

В предыдущих главах я уже говорил о написании чистого кода с использованием хорошо проименованных функций для обеспечения сопроводимости, но это только часть картины. Когда вы добавляете характеристику за характеристикой, можете ли вы быть уверены, что приложение по-прежнему делает то, что вы хотели? Любое приложение, которое, как вы надеетесь, проживет долго, нуждается в неких гарантиях долговечности. Тесты помогают обеспечивать правильность внесения новых характеристик. Они должны быть доступны для выполнения всякий раз, когда вы обновляете код, убеждаясь в том, что он *остается* правильным.

Тесты для запуска шаттлов и поддержания самолетов в полете являются строгими и часто математически доказуемыми. Это довольно круто, но выходит за рамки того, что нужно для большинства приложений Python. Из этой главы вы узнаете о методологии и инструментах, используемых разработчиками Python для тестирования кода, а также получите возможность написать несколько тестов самостоятельно.

5.1. ЧТО ТАКОЕ ТЕСТИРОВАНИЕ ПО?

Грубо говоря, *тестирование ПО* — это практические приемы проверки, оправдывает ли ПО ваши ожидания. Оно может проверять результат функции при передаче конкретного входного значения или способность приложения справляться с пиковым наплывом в сто пользователей одновременно — диапазон его работы широк. Как разработчики, мы подсознательно постоянно что-то тестируем, например запускаем сервер локально, чтобы убедиться в работоспособности своего веб-сайта.

В краткосрочной перспективе тестирование отнимает ресурсы, однако в будущем оно *сэкономит* вам время, ограничив повторное возникновение ошибок поведения и производительности и предоставив фреймворк для уверенного рефакторинга в будущем. Чем важнее фрагмент кода, тем больше времени вы захотите потратить на его тщательное тестирование.

5.1.1. Соответствует ли содержимое этикетке?

Одна из причин тестирования ПО заключается в том, чтобы определить, действительно ли оно делает то, что заявлено. Хорошо названная функция описывает свое намерение читателю, но, как говорится, благими намерениями вымощена дорога в ад. Не счесть, сколько раз я писал функцию, будучи уверен, что она добросовестно выполняет свое предназначение, только чтобы позже узнать, что я совершил оплошность.

Опечатку или исключение во фрагменте кода, с которым вы знакомы, легко выловить. Труднее отыскать те ошибки, которые становятся не

причинами мгновенных проблем, а распространяются по мере роста приложения. При хорошем тестировании проблемы можно обнаружить на ранней стадии, чтобы огородить приложение от подобных казусов в будущем. Существует целый ряд категорий тестирования, каждая из которых направлена на выявление конкретных видов проблем. Я коснусь здесь некоторых из них, но будьте уверены — этот список не исчерпывающий.

5.1.2. Суть функционального тестирования

Тестирование гарантирует, что ПО выдает нужный результат для заданных входных данных. Этот тип тестирования называется функциональным тестированием, поскольку обеспечивает правильную работу части кода. Оно отличается от других типов тестирования, таких как тестирование производительности, о котором я расскажу в разделе 5.6.

Хотя стратегии функционального тестирования различаются по масштабу и подходу, базовая анатомия функционального теста остается стабильной. Поскольку они выполняют верификацию того, что ПО дает нужный результат на основе заданных входных данных, все функциональные тесты должны выполнять несколько специфических задач, включая следующие:

1. *Подготовить входные данные* для ПО.
2. *Выявить ожидаемый результат работы* ПО.
3. *Получить фактические результаты работы* ПО.
4. *Сравнить фактические и ожидаемые результаты работы*, чтобы увидеть их совпадение либо расхождение.

Подготовка входных данных и выявление ожидаемых выходных данных — это то место, где вы будете выполнять бóльшую часть тестов, тогда как получение и сравнение фактического результата — вопрос выполнения кода (рис. 5.1).

Структурирование тестов подобным образом позволяет читать их как спецификацию кода. Это полезно, когда вы возвращаетесь к коду, который

написали давно (или на прошлой неделе, как я). Хороший тест для функции `calculate_mean` может выглядеть следующим образом:

> Если дать список целых чисел `[1, 2, 3, 4]`, то ожидаемый результат функции `calculate_mean` будет равен 2,5. Проверить, что фактический результат функции `calculate_man` совпадает с этим ожиданием.

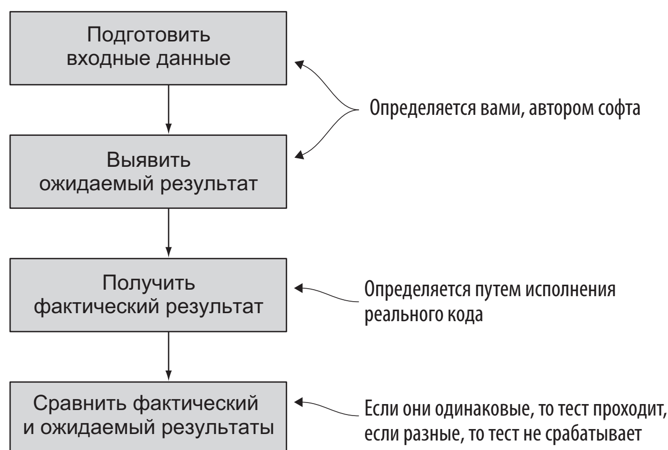


Рис. 5.1. Базовый поток функционального теста

Указанный формат масштабируется до более крупных функциональных рабочих потоков. В системе электронной коммерции «входными данными» может быть щелчок по товару, а затем нажатие кнопки Добавить в корзину. Ожидаемым «результатом» — товар, добавленный в корзину. Функциональный тест для этого рабочего потока будет выглядеть так:

> Если я зайду на страницу товара `53-DE-232` и нажму кнопку Добавить в корзину, то ожидаю увидеть `53-DE-232` в своей корзине.

Итак, тесты не только проверяют работоспособность кода, но и выступают в качестве документации по его эксплуатации. В следующем разделе вы увидите, как этот рецепт написания функционального теста применяется к разным подходам к тестированию.

5.2. ПОДХОДЫ К ФУНКЦИОНАЛЬНОМУ ТЕСТИРОВАНИЮ

Функциональное тестирование на практике принимает много форм: от постоянных мелких проверок, которые делают разработчики, до автоматизированных тестов, которые приводятся в действие перед каждым производственным развертыванием. Даже если вы знаете некоторые из приведенных типов тестирования, рекомендую почитать о каждом из них, чтобы понять сходства и различия между ними.

5.2.1. Ручное тестирование

Ручное тестирование — это практика выполнения приложения, при которой коду дается несколько элементов входных данных, чтобы проверить, что произойдет дальше. Например, если вы пишете рабочий поток регистрации на веб-сайте, то нужно ввести имя и пароль и убедиться, что новый пользователь создан. Если у вас есть требования к паролю, то нужно проверить, что использование неверного пароля *не* создает нового пользователя. Точно так же вы проверили бы уникальность имени пользователя.

Регистрация на веб-сайте, как правило, является малой (и одноразовой) частью опыта работы с продуктом для большинства пользователей, но, как вы можете видеть, вам уже нужно проверить несколько вариантов. Если что-то из этого пойдет не так, то ваши пользователи либо не смогут зарегистрироваться, либо их учетная запись будет перезаписана. Поскольку этот код так важен, опора на ручное тестирование в течение слишком долгого времени в конечном итоге приведет к тому, что вы что-то упустите. Поэтому при всех его достоинствах ручное тестирование лучше рассматривать как дополнение к другим типам тестирования.

5.2.2. Автоматизированное тестирование

В отличие от ручного тестирования, *автоматизированное тестирование* позволяет писать огромное число тестов, которые затем могут быть выполнены любое количество раз без риска пропустить проверку, когда

вы пытаетесь улизнуть из офиса в пятницу. Если такая гипотетическая ситуация кажется до предела специфичной, то это потому, что она не является гипотетической. Я уже имел несчастье ее пережить.

Автоматизированное тестирование сжимает цикл обратной связи, благодаря чему вы можете сразу увидеть, нарушило или нет внесенное изменение ожидаемое поведение. Время, которое вы сэкономите на автоматизированном тестировании, освободит вас для творческого поиска. Обнаружив то, что нужно исправить, вы должны включить это в автоматизированные тесты. Можно думать об этом как о блокировке проверки, которая гарантирует, что конкретная ошибка больше не повторится. Большинство типов тестирования, о которых мы поговорим, часто являются автоматизированными.

5.2.3. Приемочное тестирование

Самый близкий по своей природе к тесту рабочего потока Добавить в корзину *приемочный тест* проверяет высокоуровневые требования системы. ПО, которое проходит эти тесты, является *приемлемым* на основе указанных требований. Как показано на рис. 5.2, приемочные тесты отвечают на такие вопросы, как «может ли пользователь успешно пройти процесс покупки и купить желаемый товар?».

Рис. 5.2. Приемочные тесты проверяют рабочие потоки с точки зрения пользователя

Эти проверки являются критически важными для миссии бизнеса — помогают оставаться на плаву.

Приемочные тесты часто проводятся вручную стейкхолдерами бизнеса, но также тесты могут быть до некоторой степени автоматизированы с помощью сквозного тестирования. *Сквозное тестирование* проверяет, выполняется ли набор действий (от начала и до конца) с участием необходимых данных в правильных местах. Если рабочий поток с точки зрения пользователя приемлем, то он начинает выглядеть так же просто, как поток добавления в корзину.

ТЕСТИРОВАНИЕ ПРЕДНАЗНАЧЕНО ДЛЯ ВСЕХ

Библиотеки вроде Cucumber (https://cucumber.io) предоставляют возможность описывать сквозные тесты на естественном языке как высокоуровневые действия, например: «нажать кнопку отправки». Эти тесты гораздо легче понять, чем месиво из кода. Написание шагов на естественном языке документирует систему таким образом, что почти любой в организации может ее понять.

Эта идея *разработки на основе поведения* (BDD, behavior-driven development) позволяет сотрудничать с коллегами, не имеющими опыта в написании кода. Разработка на основе поведения часто используется как способ определения желаемых исходов через реализацию кода, чтобы обеспечить прохождение тестов позже.

Сквозные тесты обычно проверяют участки с высокой ценностью для бизнеса: если корзина не работает, то никто не сможет купить товары и вы лишитесь выручки, но они также наиболее подвержены взлому, потому что охватывают очень широкий сегмент функциональности. Если какой-либо один шаг в рабочем потоке не работает, то весь сквозной тест не будет пройден. Попробуйте создать набор тестов, различающихся по степени гранулярности, чтобы быстрее засекать проблемы.

Если сквозные тесты — *наименее* гранулярны, то что находится на другом конце спектра?

5.2.4. Юнит-тестирование

Юнит-тестирование (или модульное тестирование) — это, пожалуй, самое важное, что вы извлечете из этой главы. Юнит-тесты следят за тем, чтобы работали все маленькие фрагменты ПО, и закладывают прочную основу для более крупных мероприятий по тестированию, таких как сквозное тестирование. Я покажу, как начать работу с юнит-тестированием на Python в разделе 5.4.

> **ОПРЕДЕЛЕНИЕ**
>
> *Модуль* (unit) — это небольшой фундаментальный фрагмент ПО, как «единица» в термине «единичная окружность». То, что составляет модуль, является источником для многочисленных философских разглагольствований, но хорошее рабочее определение состоит в том, что это фрагмент кода, который можно изолировать в целях тестирования. Функции обычно считаются модулями кода — их можно исполнять изолированно, вызывая их с соответствующими входными данными. Строки кода внутри этих функций не могут быть изолированы, поэтому они меньше модуля. Классы содержат много частей, которые могут быть изолированы дальше, поэтому они обычно больше модуля, но иногда они рассматриваются как модули.

Юнит-тестирование направлено на проверку правильности работы всех отдельных модулей кода. Что может быть основательнее?

Функции являются наиболее часто встречающейся целью функциональных юнит-тестов. В конце концов, слово «функция» есть в названии. Благодаря их входно-выходной природе, ответственность, разделенная на малые функции, позволяет применять к ним функциональное тестирование непосредственно.

Структурирование кода на модули посредством разделения ответственности, инкапсуляции и слабой сопряженности облегчает тестирование. Тестирование может быть утомительным, поэтому приветствуется любая

возможность уменьшить трение, ведь чистота и простота теста — залог уверенности в нем.

Большинство юнит-тестов в Python сравнивают ожидаемые и фактические результаты, используя простое сравнение на эквивалентность. Попробуйте сами! Откройте интерактивную среду REPL и создайте функцию `calculate_mean`:

```
>>> def calculate_mean(numbers):
...     return sum(numbers) / len(numbers)
```

Теперь можно проверить эту функцию с помощью нескольких разных входных данных, сопоставляя их с ожидаемыми результатами:

```
>>> 2.5 == calculate_mean([1, 2, 3, 4])
True
>>> 5.5 == calculate_mean([5, 5, 5, 6, 6, 6])
True
```

Теперь попробуйте несколько других списков чисел в REPL, чтобы убедиться, что `calculate_mean` дает правильные результаты. Подумайте о наборах входных данных, которые могут изменить поведение функции:

- Правильно ли она работает с отрицательными числами?
- Работает ли она, когда список чисел содержит `0`?
- Работает ли она, когда список является пустым?

Для таких приступов любопытства стоит писать тесты. Они иногда вскрывают то, что вы не учли.

```
>>> 0.0 == calculate_mean([-1, 0, 1])
True
>>> 0.0 == calculate_mean([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in calculate_mean
ZeroDivisionError: division by zero
```

Возбуждает исключение для варианта, который вы еще не рассматривали

Вы можете исправить функцию `calculate_mean`, вернув `0`, если список пустой:

```
>>> def calculate_mean(numbers):
...     if not numbers:
...         return 0
...     return sum(numbers) / len(numbers)
>>> 0.0 == calculate_mean([])
True
```

Отлично! Функция `calculate_mean` прошла все наши испытания. Помните, что юнит-тесты — это фундамент для более масштабного тестирования. Поэтому двигаемся дальше.

5.2.5. Интеграционное тестирование

Интересом *интеграционного тестирования* является работа модулей в тандеме (рис. 5.3). Зачем нужны десять полнофункциональных модулей, если их нельзя собрать вместе? В то время как сквозные тесты рабочих потоков обычно формируются с точки зрения пользователя, интеграционные тесты больше фокусируются на поведении кода — это другой уровень абстракции.

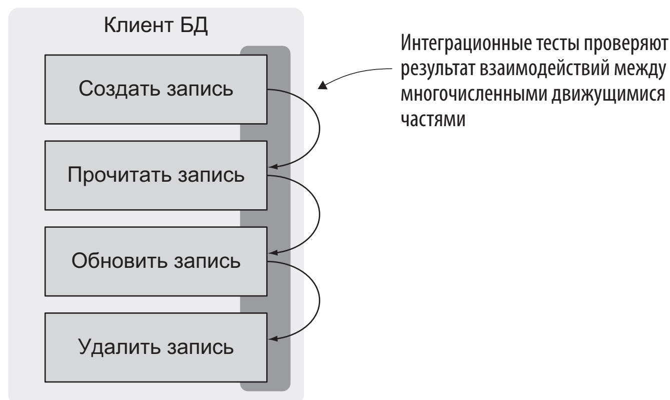


Рис. 5.3. Интеграционные тесты фокусируются на том, как операции работают вместе

Будьте осторожны, интеграционные тесты должны нанизывать несколько фрагментов кода вместе, поэтому структурированы так же, как тестируемый код. Это вводит тесную сопряженность между тестами

и кодом — изменения в коде, которые производят тот же результат, не гарантируют срабатывания тестов, поскольку тесты сфокусированы на том, *как* результат достигается.

На интеграционные тесты требуется значительно больше времени, чем на юнит-тесты. Они не просто выполняют некие функции и проверяют выходные данные, то есть используют БД для создания записей и манипулирования ими. Тестируемое взаимодействие является более многосложным, поэтому время, необходимое для его выполнения, может расти. По этим причинам число интеграционных тестов обычно меньше, чем юнит-тестов.

5.2.6. Пирамида тестирования

Теперь, когда вы ознакомились с ручным, юнит- и интеграционным тестированием, давайте кратко рассмотрим взаимодействие между ними. Идея пирамиды тестирования (рис. 5.4) состоит в том, что вы должны широко применять функциональные тесты, такие как юнит- и интеграционные тесты, но быть более консервативными с длинными, хрупкими и ручными тестами[1].

Рис. 5.4. Пирамида тестирования

[1] Пирамиды тестирования были впервые описаны Майком Коном в книге «AGILE: оценка и планирование проектов».

Каждый имеет свои достоинства, и все будет зависеть от приложения и ресурсов, находящихся в вашем распоряжении, но эта пирамида является полезным эмпирическим правилом, помогающим решить, куда инвестировать время.

Вы получите максимум эффекта за свои деньги, добившись того, чтобы все маленькие фрагменты ПО работали как надо, а затем работали сообща. Опять же, автоматизация этого процесса даст возможность использовать освободившееся время для обдумывания новых способов нарушения работы вашего ПО. Затем вы можете использовать эти идеи в качестве новых тестов и постепенно укреплять уверенность, которая будет вести вас вперед.

5.2.7. Регрессионное тестирование

Регрессионное тестирование — это не столько подход к тестированию как таковому, сколько порядок, которому нужно следовать во время разработки приложений. Когда вы пишете тест, предполагается, что вы говорите: «Я хочу, чтобы код продолжал работать именно так». Если из-за внесенных вами дополнений тестируемый код изменил поведение, такая ситуация называется регрессией. *Регрессия* — это переход в нежелательное (или, по крайней мере, неожиданное) состояние, и обычно это плохо.

Регрессионное тестирование — это практика выполнения существующих тестовых наборов после каждого изменения кода перед его отправкой в продакшен. *Тестовый набор* — это коллекция тестов, которые вы накопили в течение долгого времени, написанных для проверки кода в качестве юнит- или интеграционных тестов либо для устранения дефектов, обнаруживаемых в ходе разведывательного ручного тестирования. Многие команды разработчиков выполняют эти тестовые наборы в среде *непрерывной интеграции* (CI, continuous integration), где изменения в приложении часто комбинируются и тестируются перед релизом. Полное обсуждение непрерывной интеграции выходит за рамки этой книги, но ее идея в том, чтобы настроиться на успех, выполнив все свои тесты относительно всех изменений. Рекомендую обратиться к среде Travis CI (https://docs.travis-ci.com/user/for-beginners/) либо CircleCI (https://circleci.com/docs/2.0/about-circleci/), чтобы узнать больше.

ПЕРЕХВАТЧИКИ УПРАВЛЕНИЯ ВЕРСИЯМИ

Одним из способов автоматизации юнит-тестов в системах управления версиями является использование прекоммит-хука (precommit hook). При каждом комите кода срабатывает хук, который запускает тесты на выполнение. Если возникают какие-либо сбои, то комит не срабатывает и вы получаете напоминание о том, что их необходимо исправить перед комитом кода. Большинство инструментов юнит-тестирования должны довольно хорошо интегрироваться с этим подходом. Повторный прогон тестов в среде непрерывной интеграции обеспечивает непосредственное прохождение перед развертыванием кода.

По мере добавления новых характеристик тестовый набор получает новые тесты. Они фиксируются как регрессионные тесты для будущих изменений. Схожим образом, принято добавлять тесты на обнаруживаемые ошибки, чтобы убедиться, что та или иная ошибка не появится снова. Как и код, тестовые наборы не всегда идеальны. Но опора на надежный набор, который вовремя сигнализирует о проблемах, поможет сосредоточиться на других участках, таких как инновации и производительность.

С учетом всего сказанного давайте посмотрим, как начать писать тесты на Python.

5.3. КОНСТАТАЦИЯ ФАКТОВ

Следующий шаг к созданию реальных тестов состоит в проверочном *утверждении*, что то или иное сравнение является истинным. Проверочное утверждение — это констатация факта. Если оно не истинно, то неверно либо какое-то допущение, либо само утверждение. «Каждое утро я вижу на горизонте солнце» истинно, пока не появились облака. Допущение «при чистом небе» сделает утверждение полностью, а не частично истинным.

В Python проверочные утверждения можно писать с помощью ключевого слова `assert`. Когда проверочные утверждения не соответствуют реальности, они вызывают ошибку `AssertionError`.

Вы можете проверить функцию `calculate_mean` с помощью проверочных утверждений, добавив `assert` перед сравнениями. Успешное проверочное утверждение не будет иметь результата, а неуспешное покажет обратную трассу для ошибки `AssertionError`:

```
>>> assert 10.0 == calculate_mean([0, 10, 20])
>>> assert 1.0 == calculate_mean([1000, 3500, 7_000_000])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Вокруг такого поведения в Python строятся многие инструменты тестирования. Используя рецепт для функционального теста (настроить входные данные, выявить ожидаемые выходные данные, получить фактические выходные данные и т. д.), эти инструменты помогают выполнять сравнение и предоставляют ценный контекст, когда проверочные утверждения не срабатывают. Читайте дальше, чтобы увидеть, как два наиболее широко используемых инструмента тестирования в Python обрабатывают проверочные утверждения.

5.4. ЮНИТ-ТЕСТИРОВАНИЕ С ПОМОЩЬЮ UNITTEST

`unittest` — это встроенный в Python фреймворк тестирования. Несмотря на название, его можно использовать для интеграционного тестирования. Он предоставляет средства для создания проверочных утверждений о коде, а также инструмент для *выполнения* тестов. В этом разделе вы увидите то, каким образом тесты организуются и как их выполнять, и наконец немного попрактикуетесь в написании реальных тестов. Давайте приступим к делу!

5.4.1. Организация тестов с помощью unittest

Фреймворк `unittest` предоставляет набор средств для выполнения проверочных утверждений. Вы уже видели, как писать сырые инструкции

`assert` для тестирования кода, а `unittest` предоставляет класс `TestCase` со специализированными методами проверочного утверждения для более понятных результатов тестирования. Ваши тесты будут наследовать от этого класса и использовать методы для создания проверочных утверждений.

Я призываю вас использовать эти тестовые классы в качестве стратегии для группирования тестов. Эти классы гибкие — их можно применить к любым тестам. Если у вас много тестов для класса, то удобно поместить их в самостоятельный класс `TestCase`. Если же у вас много тестов для отдельного метода внутри класса, то вы можете создать `TestCase` только для этих тестов. Вы научились использовать связность, пространства имен и разделение ответственности, и пора применить эти идеи к тестам.

5.4.2. Выполнение тестов с помощью unittest

Фреймворк `unittest` предоставляет исполнителя тестов, который доступен после набора в терминале команды `python -m unittest`. После запуска он начинает:

1) искать модули с именем `test_*` или `*_test` в текущем каталоге (и любых подкаталогах);
2) искать классы, которые наследуют от `unittest.TestCase`, в этих модулях;
3) искать методы, которые начинаются с `test_`, в этих классах.

Некоторые разработчики любят размещать свои тесты как можно ближе к релевантному коду, облегчая поиск тестов для конкретного модуля, вызывающего интерес. Другие предпочитают помещать все свои тесты в каталог tests/ в корне проекта, чтобы держать их отдельно от кода. Я делал по-разному, и вы делайте так, как удобно вам и вашей команде или сообществу, вместе с которым вы пишете ПО.

5.4.3. Написание первого теста с помощью unittest

Теперь, когда у вас есть представление, как работает `unittest`, начинайте тестировать. Следующий листинг разворачивает класс, который можно использовать, чтобы попрактиковаться в тестировании.

Листинг 5.1. Класс Product для системы электронной коммерции

```
class Product:
    def __init__(self, name, size, color):    ◄── Атрибуты товара указываются,
        self.name = name                          когда создается экземпляр
        self.size = size                          класса Product
        self.color = color

    def transform_name_for_sku(self):
        return self.name.upper()

    def transform_color_for_sku(self):
        return self.color.upper()

    def generate_sku(self):    ◄── Артикул уникально идентифицирует атрибуты товара
        """
        Генерирует артикул (SKU) для этого товара.

        Пример:
            >>> small_black_shoes = Product('shoes', 'S', 'black')
            >>> small_black_shoes.generate_sku()
            'SHOES-S-BLACK'
        """
        name = self.transform_name_for_sku()
        color = self.transform_color_for_sku()
        return f'{name}-{self.size}-{color}'
```

Этот класс представляет товар для покупки в системе электронной коммерции. У товара есть название и атрибуты размера и цвета, и каждая комбинация этих атрибутов производит артикул, или *модуль хранения запасов* (SKU, stock keeping unit). Артикул — это уникальный внутренний идентификатор, используемый компаниями для ценообразования и инвентаризации, в котором часто используются символы только в верхнем регистре. Поместите это определение класса в модуль product.py.

После того как вы создали модуль product, создайте модуль test_product.py в том же каталоге, что и product.py. Начните с импорта `unittest` и создания пустого класса `ProductTestCase`, который наследует от базового класса `TestCase`:

```
import unittest

class ProductTestCase(unittest.TestCase):
    pass
```

Если в этом месте вы выполните команду `python -m unittest` только с product.py и вашим пустым тестовым случаем в test_product.py, то исполнитель тестов сообщит, что никакие тесты не выполнил:

```
$ python -m unittest
----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

Видимо, он нашел модуль `test_product` и класс `ProductTestCase`, но вы еще не написали там тесты. Проверьте это, добавив в класс пустой тестовый метод:

```
import unittest

class ProductTestCase(unittest.TestCase):
    def test_working(self):
        pass
```

Попробуйте запустить исполнителя тестов еще раз. Вы должны увидеть, что на этот раз он выполнил один тест:

```
$ python -m unittest
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Теперь вы готовы к настоящему волшебству. Вспомните рецепт функционального теста:

1) настроить результаты;
2) выявить ожидаемый результат;
3) получить фактический результат;
4) сравнить ожидаемый и фактический результаты.

А если вы хотите протестировать метод `transform_name_for_sku` из класса `Product`, то рецепт будет такой:

1) создать экземпляр класса `Product` с именем, размером и цветом;

2) обратить внимание, что `transform_name_for_sku` возвращает `name.upper()`, — ожидаемым результатом является имя в верхнем регистре;
3) вызвать метод `transform_name_for_sku` экземпляра класса `Product` и сохранить его в переменной;
4) сравнить ожидаемый результат с сохраненным фактическим результатом.

Вы можете написать первые три шага, используя регулярный код для создания экземпляра класса `Product` и получения значения из метода `transform_name_for_sku`. Использование инструкции `assert` для четвертого шага будет работать, но ошибка `AssertionError` по умолчанию даст не много информации в обратной трассировке. Именно здесь в игру вступают специализированные методы проверочного утверждения в `unittest`. Наиболее распространенным методом сравнения двух значений является `assertEqual`, который принимает в качестве аргументов ожидаемое и фактическое значения. Он вызывает ошибку `AssertionError` и предоставляет дополнительную информацию, показывающую разницу между двумя значениями, если они не равны. Этот добавленный контекст поможет вам быстрее находить проблемы.

Вот как этот тест мог бы выглядеть при первом проходе:

```
import unittest

from product import Product

class ProductTestCase(unittest.TestCase):
    def test_transform_name_for_sku(self):
        small_black_shoes = Product('shoes', 'S', 'black')
        expected_value = 'SHOES'
        actual_value = small_black_shoes.transform_name_for_sku()
        self.assertEqual(expected_value, actual_value)
```

Подготавливает условия для transform_name_for_sku: товар с его атрибутами

Констатирует ожидаемый результат для generate_sku с заданными входными данными

Использует специальный метод утверждения с проверкой эквивалентности для сравнения двух значений

Получает фактический результат функции generate_sku для сравнения

В результате выполнения исполнителя тестов должно появиться сообщение `Ran 1 test`, и если тест проходит (он должен пройти), то вы увидите не много дополнительных выходных данных.

Важно посмотреть, как тесты не срабатывают, чтобы проверить, действительно ли они отловят проблему. Измените ожидаемое значение `'SHOES'` на `'SHOEZ'` и снова выполните тест. Теперь `unittest` вызывает ошибку `AssertionError`, констатируя, что `SHOEZ'!='SHOES'`:

```
$ python -m unittest
F
======================================================================
FAIL: test_transform_name_for_sku (test_product.ProductTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/dhillard/test/test_product.py", line 11, in
➥ test_transform_name_for_sku
    self.assertEqual(expected_value, actual_value)
AssertionError: 'SHOEZ' != 'SHOES'
- SHOEZ
?     ^
+ SHOES
?     ^

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Убедившись, что тест присматривает за кодом, вы можете изменить его обратно на надлежащее значение и перейти к еще одному тесту.

5.4.4. Написание первого интеграционного теста с помощью unittest

Вы увидели, что такое модули кода и как их можно протестировать, и пришло время посмотреть, как можно протестировать интеграцию многочисленных модулей. Юнит-тесты предназначены для проверки поведения малых фрагментов ПО, поэтому без интеграционных тестов трудно сказать, работают ли эти малые фрагменты вместе, чтобы произвести что-то полезное как единое целое (рис. 5.5).

Теперь, когда вы можете управлять товарами в инвентарной описи с помощью системы артикулов, клиенты должны получить возможность их покупать. Новый класс `ShoppingCart` с возможностью добавления

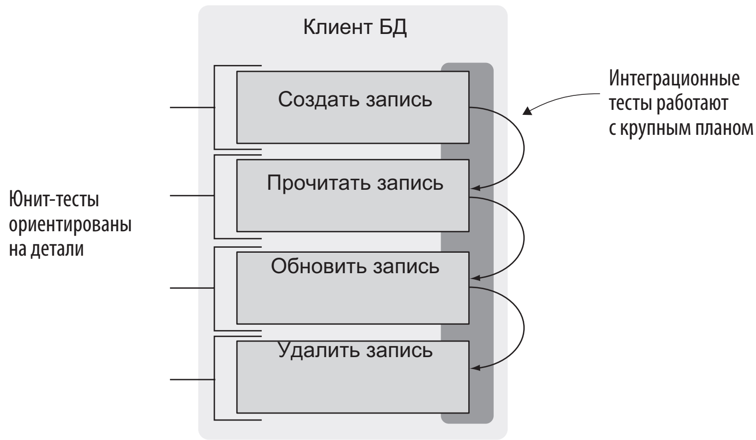


Рис. 5.5. Юнит- и интеграционные тесты

и удаления товаров станет хорошим первым шагом. Корзина хранит товары в форме словаря:

```
{
    'SHOES-S-BLACK': {        ← Ключи являются артикулами товаров
        'quantity': 2,        ← Вложенный словарь метаданных о позиции
        ...                     в корзине, например количестве
    },
    'SHOES-M-BLUE': {
        'quantity': 1,
        ...
    },
}
```

Класс `ShoppingCart` содержит методы добавления и удаления товара путем управления данными в словаре.

```
from collections import defaultdict


class ShoppingCart:
    def __init__(self):
        self.products = defaultdict(lambda: defaultdict(int))   ← Используя defaultdict, упрощает логику проверки наличия товара в корзине

    def add_product(self, product, quantity=1):   ← Добавляет количество товара в корзину
        self.products[product.generate_sku()]['quantity'] += quantity
```

```
    def remove_product(self, product, quantity=1):  ◄── Удаляет количество товара из корзины
        sku = product.generate_sku()
        self.products[sku]['quantity'] -= quantity
        if self.products[sku]['quantity'] == 0:
            del self.products[sku]
```

Поведение класса `ShoppingCart` теперь представляет пару интеграционных точек, которые должны быть протестированы:

- корзина опирается на метод `generate_sku` экземпляра класса `Product` (*интегрируется с этим методом*);
- добавление и удаление товаров должно работать в тандеме — товар, который был добавлен, также необходимо иметь возможность удалить.

Тестирование этих интеграций похоже на юнит-тестирование, разница лишь в том, сколько кода выполняется во время тестирования. Если юниттест обычно выполняет код только в одном методе и утверждает, что выход соответствует ожиданиям, то интеграционный тест может выполнять много методов и делать проверочные утверждения сразу для нескольких вещей.

В случае `ShoppingCart` полезным тестом была бы инициализация корзины, добавление товара, его удаление и проверка корзины на отсутствие товаров:

Листинг 5.2. Интеграционный тест для класса ShoppingCart

```
import unittest

from cart import ShoppingCart
from product import Product

class ShoppingCartTestCase(unittest.TestCase):  ◄── Настройка теста сравнима с более ранним юнит-тестом
    def test_add_and_remove_product(self):
        cart = ShoppingCart()  ◄── Создает корзину для добавления в нее товаров

        product = Product('shoes', 'S', 'blue')  ◄── Создает маленькие синие туфельки

        cart.add_product(product)  ◄── Добавляет обувь в корзину
        cart.remove_product(product)  ◄── Удаляет обувь из корзины

        self.assertDictEqual({}, cart.products)  ◄── Корзина должна быть пустой!
```

Этот тест вызывает метод `__init__` корзины, метод `generate_sku` товара и методы `add_product` и `remove_product` корзины. Там много чего происходит. Как и следовало ожидать, интеграционные тесты длиннее.

5.4.5. Тестовые дублеры

Вам придется часто писать тесты для кода, который взаимодействует с другой системой, будь то БД или вызов API, и выполнение тестов может разрушить реальные данные. Тестовый набор выполняет участок кода многократно, и связь этого участка с другой системой сильно замедлит процесс тестирования. Эти другие системы могут не находиться под вашим контролем, и лучше их сымитировать, вместо того чтобы использовать что-то настоящее.

Существует несколько тонко различающихся способов имитации сторонних систем с использованием *тестовых дублеров*:

- *Подделка* (faking) — использование системы, которая ведет себя очень похоже на настоящую, но избегает дорогостоящих или разрушительных действий.
- *Установка заглушек (стабы)* (stubbing) — использование предопределенного значения в качестве отклика вместо получения ответа от живой системы.
- *Подставка (моки)* (mocking) — использование системы с тем же интерфейсом, что и реальная, но которая в дополнение к этому регистрирует взаимодействия для последующего контроля и проверочных утверждений.

Подделка и установка стабов в Python предусматривают, что вы сами напишете имитации в форме функций или классов, которые будут использоваться во время выполнения теста. Мок же делается с помощью модуля `unittest.mock`.

Предположим, что ваш код вызывает конечную точку API, чтобы получить налоговую информацию о продажах товара. Вы не очень-то хотите использовать эту конечную точку в своем тесте, потому что видели, что у нее на отклик уходит несколько секунд. Вдобавок ко всему она

возвращает динамические данные, поэтому нельзя быть уверенным, о каком значении следует делать проверочные утверждения в тесте. Если код выглядит так:

```
from urllib.request import urlopen


def add_sales_tax(original_amount, country, region):
    sales_tax_rate =
➥ urlopen(f'https://tax-api.com/{country}/{region}').read().decode()
    return original_amount * float(sales_tax_rate)
```

то юнит-тест с моком (mock) может выглядеть так:

```
import io
import unittest
from unittest import mock

from tax import add_sales_tax


class SalesTaxTestCase(unittest.TestCase):
    @mock.patch('tax.urlopen')
    def test_get_sales_tax_returns_proper_value_from_api(
            self,
            mock_urlopen
    ):
        test_tax_rate = 1.06
        mock_urlopen.return_value = io.BytesIO(
            str(test_tax_rate).encode('utf-8')
        )

        self.assertEqual(
            5 * test_tax_rate,
            add_sales_tax(5, 'USA', 'MI')
        )
```

Декоратор mock.patch подставляет мок в виде указанного объекта или метода

Тестовая функция получает мок или метод

Вызов мока urlopen теперь вернет мокированный отклик с ожидаемой тестовой налоговой ставкой

Утверждает, что метод add_sales_tax вычисляет новое значение из налоговой ставки, возвращаемой API-интерфейсом

Такое тестирование позволяет вам объявить: «Код, которым я управляю, ведет себя так с учетом этих допущений», где допущения создаются с использованием тестовых дублеров. Если вы уверены, что библиотека requests работает так, как ожидается, то можно использовать тестовые дублеры, чтобы с ней не связываться. Если в будущем потребуется использовать другую клиентскую библиотеку HTTP или изменить API, из которого вы получаете налоговую информацию, то тест не придется менять.

Не злоупотребляйте тестовыми дублерами, как это время от времени делаю я. Иногда возникает соблазн подставить собственный код, чтобы изолировать модуль. Это может привести к *хрупким* тестам, которые часто ломаются при изменении кода, отчасти потому, что они слишком точно отражают структуру реализации. Изменяя реализацию, изменяйте и свои тесты.

Попробуйте писать тесты, гибкие в плане изменений в базовой реализации. Такой подход, опять же, является слабо сопряженнным. Слабая сопряженность применима к коду тестирования в той же мере, что и к коду реализации.

5.4.6. Попробуйте сами

Как бы вы проверили другие методы в классах `Product` и `ShoppingCart`? С учетом рецепта для функциональных тестов попробуйте добавить дополнительные тесты для остальных методов. Тщательно продуманный тестовый набор будет содержать проверочные утверждения для каждого метода и для каждого отличающегося ожидаемого результата. Возможно, вы даже найдете едва уловимую ошибку! В качестве подсказки попробуйте проверить, что происходит, когда вы удаляете из корзины больше предметов, чем она содержит.

Среди тех нескольких значений, которые вам приходится проверять, есть словари. `unittest` содержит специальный метод `assertDictEqual`, который обеспечивает полезный результат, характерный для словарей, когда тест не срабатывает.

В случае коротких тестов, подобных тому, который вы уже написали, вы можете пропустить сохранение ожидаемого и фактического значений в качестве переменных. Введите выражения в `assertEqual` непосредственно в качестве аргументов:

```
def test_transform_name_for_sku(self):
    small_black_shoes = Product('shoes', 'S', 'black')
    self.assertEqual(
        'SHOES',
        small_black_shoes.transform_name_for_sku(),
    )
```

Когда вы сделаете попытку, вернитесь и сверьтесь со следующим листингом. Не забудьте использовать исполнитель тестов фреймворка `unittest` после написания или изменения теста, чтобы увидеть, что код продолжает его проходить.

Листинг 5.3. Тестовый набор для классов Product и ShoppingCart

```
class ProductTestCase(unittest.TestCase):
    def test_transform_name_for_sku(self):
        small_black_shoes = Product('shoes', 'S', 'black')
        self.assertEqual(
            'SHOES',
            small_black_shoes.transform_name_for_sku(),
        )

    def test_transform_color_for_sku(self):
        small_black_shoes = Product('shoes', 'S', 'black')
        self.assertEqual(
            'BLACK',
            small_black_shoes.transform_color_for_sku(),
        )

    def test_generate_sku(self):
        small_black_shoes = Product('shoes', 'S', 'black')
        self.assertEqual(
            'SHOES-S-BLACK',
            small_black_shoes.generate_sku(),
        )


class ShoppingCartTestCase(unittest.TestCase):
    def test_cart_initially_empty(self):
        cart = ShoppingCart()
        self.assertDictEqual({}, cart.products)

    def test_add_product(self):
        cart = ShoppingCart()
        product = Product('shoes', 'S', 'blue')

        cart.add_product(product)
        self.assertDictEqual({'SHOES-S-BLUE': {'quantity': 1}},
➥ cart.products)

    def test_add_two_of_a_product(self):
        cart = ShoppingCart()
        product = Product('shoes', 'S', 'blue')

        cart.add_product(product, quantity=2)

        self.assertDictEqual({'SHOES-S-BLUE': {'quantity': 2}},
```

```
➥ cart.products)

    def test_add_two_different_products(self):
        cart = ShoppingCart()
        product_one = Product('shoes', 'S', 'blue')
        product_two = Product('shirt', 'M', 'gray')

        cart.add_product(product_one)
        cart.add_product(product_two)

        self.assertDictEqual(
            {
                'SHOES-S-BLUE': {'quantity': 1},
                'SHIRT-M-GRAY': {'quantity': 1}
            },
            cart.products
        )

    def test_add_and_remove_product(self):
        cart = ShoppingCart()
        product = Product('shoes', 'S', 'blue')

        cart.add_product(product)
        cart.remove_product(product)

        self.assertDictEqual({}, cart.products)

    def test_remove_too_many_products(self):
        cart = ShoppingCart()
        product = Product('shoes', 'S', 'blue')

        cart.add_product(product)
        cart.remove_product(product, quantity=2)

        self.assertDictEqual({}, cart.products)
```

Вы можете устранить ошибку в корзине покупок, обновив метод `remove_product` в части удаления товара из корзины, если его количество *меньше или равно* `0`:

```
if self.products[sku]['quantity'] <= 0:
            del self.products[sku]
```

5.4.7. Написание дополнительных тестов

Хорошие тесты будут использовать входные данные, влияющие на поведение тестируемого метода. Все артикулы в типичной ситуации пишутся

в верхнем регистре и обычно не содержат пробелов — только буквы, цифры и тире. Но что делать, если название товара содержит пробел? Вы захотите удалить пробелы до того, как название будет помещено в артикул. Например, артикул маек должен начинаться с `'TANKTOP'`.

Это требование является новым, и поэтому вы можете написать новый тест, который описывает то, как код должен себя вести.

```
def test_transform_name_for_sku(self):
    medium_pink_tank_top = Product('tank top', 'M', 'pink')
    self.assertEqual(
        'TANKTOP',
        medium_pink_tank_top.transform_name_for_sku(),
    )
```

Тест не пройден, потому что текущий код возвращает `'TANK TOP'`. Это нормально, так как еще не создана поддержка для товаров с пробелами в названии. Видя, что этот тест не сработал по ожидаемой причине, вы приходите к выводу, что нужно написать код с правильной обработкой пробелов.

Придумывание дополнительных тестов подобного рода само по себе имеет свою ценность, потому что выводит такие вопросы на поверхность на более раннем этапе в процессе разработки. Затем вы можете опросить других заинтересованных лиц и спросить: «Каковы все возможные форматы названий товаров, которые нам, возможно, нужно будет поддержать?». Если их ответ даст вам новую информацию, то вы можете включить ее в код и тесты, добившись более высокого качества конечного продукта.

Теперь, когда вы поняли преимущества фреймворка `unittest`, пришло время узнать о фреймворке `pytest`.

5.5. ТЕСТИРОВАНИЕ С ПОМОЩЬЮ ФРЕЙМВОРКА PYTEST

Хотя `unittest` — это полнофункциональный и зрелый фреймворк тестирования, встроенный в Python, у него есть несколько недостатков. Для некоторых он ощущается «непитоновским», потому что использует верблюжийРегистр вместо змеиного_стиля для имен методов (реликвия JUnit-овского периода его истории). `unittest` также требует изрядного

объема стереотипного кода, что несколько затрудняет понимание базовых тестов.

ПИТОНОВСКИЙ КОД

Исходный код часто называют *питоновским*, если в нем используются средства и общие стилевые рекомендации языка Python. В питоновском коде используется змеиный_регистр имен переменных, методов и включений в список вместо простых циклов `for` и т. д.

Для тех, кто любит краткие, прямолинейные тесты, решением станет фреймворк `pytest` (https://docs.pytest.org/en/latest/getting-started.html). После его установки вы можете вернуться к сырым инструкция `assert`, которые уже знаете. Фреймворк `pytest` выполняет под капотом небольшую скрытую «магию», и работа выглядит гладкой.

Фреймворк `pytest` по умолчанию производит более удобочитаемый результат, сообщая вам о системе, числе найденных тестов, результате отдельных тестов и сводке совокупных результатов тестирования:

```
$ pytest
========== test session starts ==========
platform darwin -- Python 3.7.3, pytest-5.0.1, py-1.8.0,
➥ pluggy-0.12.0
rootdir: /path/to/ecommerce/project
collected 15 items

test_cart.py ............    [ 80%]
test_product.py ..           [ 93%]
test_tax.py .                [100%]

======= 15 passed in 0.12 seconds =======
```


5.5.1. Организация тестов с помощью фреймворка pytest

Фреймворк `pytest` обнаруживает ваши тесты автоматически, как и `unittest`. Он даже находит любые завалявшиеся тесты фреймворка

`unittest`. Одно из его ключевых отличий заключается в том, что его правильные тестовые классы называются `Test*` и для работы не должны наследовать от базового класса (как, например, `unittest.TestCase`).

Команда для запуска тестов с помощью фреймворка `pytest` выглядит проще:

```
pytest
```

Поскольку `pytest` не требует наследования от базового класса или каких-либо специальных методов, отпадает строгая необходимость в организации тестов в классы. Однако я по-прежнему рекомендую это делать, потому что класс остается хорошим организационным инструментом. Фреймворк `pytest` будет включать имя тестового класса в вывод сообщения о сбое и тому подобной информации, что поможет понять, где тесты располагаются и о чем они говорят. В целом тесты фреймворка `pytest` могут быть организованы аналогично тестам фреймворка `unittest`.

5.5.2. Конвертирование тестов unittest в pytest

Поскольку `pytest` обнаруживает ваши существующие тесты фреймворка `unittest`, вы можете поступательно конвертировать тесты в `pytest`, как вы хотите (и *если* вообще хотите). Для тестового набора, который вы уже писали, это конвертирование выглядит так:

- Удалить импорт фреймворка `unittest` из test_product.py.
- Переименовать класс `ProductTestCase` в `TestProduct` и удалить наследование от `unittest.TestCase`.
- Заменить любые инструкции `self.assertEqual(expected, actual)` на `assert actual == expected`.

Тестовый случай из более ранних версий больше похож на описанный ниже в рамках фреймворка `pytest`.

Листинг 5.4. Тестовый случай во фреймворке pytest

```
class TestProduct:    ◄——— Нет необходимости наследовать от любого базового класса
    def test_transform_name_for_sku(self):
        small_black_shoes = Product('shoes', 'S', 'black')
```

```
        assert small_black_shoes.transform_name_for_sku() == 'SHOES'

    def test_transform_color_for_sku(self):
        small_black_shoes = Product('shoes', 'S', 'black')
        assert small_black_shoes.transform_color_for_sku() == 'BLACK'

    def test_generate_sku(self):
        small_black_shoes = Product('shoes', 'S', 'black')
        assert small_black_shoes.generate_sku() == 'SHOES-S-BLACK'
```

self.assertEqual уходит, и вместо него используются сырые инструкции assert

Как видите, `pytest` приводит к более короткому и, возможно, более удобочитаемому тестовому коду. Он также предоставляет свой собственный фреймворк средств, которые облегчают настройку среды и зависимостей для тестов. Для углубления в тему `pytest` настоятельно рекомендую книгу Брайана Оккена «Тестирование на Python с помощью pytest: простое, быстрое, эффективное и масштабируемое» (*Okken Brian, Python Testing with pytest: Simple, Rapid, Effective, and Scalable, Pragmatic Bookshelf, 2017*).

Теперь у вас за плечами есть несколько проведенных юнит- и интеграционных тестов. Читайте дальше, чтобы узнать о нефункциональном тестировании.

5.6. ЗА ПРЕДЕЛАМИ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ

Бóльшая часть этой главы посвящена функциональным тестам. Сначала мы приводим код в работоспособное состояние и только потом делаем его быстрым, поэтому функциональное тестирование предшествует тестированию скорости выполнения кода. После того как вы убедились что код работает, приступайте к обеспечению его производительности.

5.6.1. Тестирование производительности

Тестирование производительности показывает, как вносимые изменения влияют на потребление памяти, процессора и диска. Из главы 4 вы узнали о нескольких инструментах тестирования производительности

модулей кода, таких как модуль `timeit`, который выручает меня при поиске вариантов для конкретных строк кода и функций. Это не автоматические замеры — они предназначены для нерегламентированного сравнения двух подходов и пишутся быстро, чтобы выявить, какая из двух реализаций быстрее.

Разработка крупных приложений с критически важными операциями требует интеграции автоматизированных тестов производительности. Такие тесты на практике похожи на регрессионные тесты: если вы развертываете изменение и замечаете, что приложение начинает потреблять на 20 % больше памяти, то изменение нужно тщательно изучить. Тестирование производительности также позволяет полюбоваться ускорением приложения, когда вы исправляете медленный фрагмент кода.

В отличие от юнит-тестирования, которое дает однозначный результат «прошел/не прошел», тестирование производительности делает более подробные выводы. Если вы видите, что приложение имеет тенденцию к замедлению (либо резко скачет после развертывания), обратите на это внимание. Природа такого рода тестирования немного затрудняет его автоматизацию и мониторинг, но решения уже существуют.

5.6.2. Нагрузочное тестирование

Нагрузочное тестирование — это один из видов тестирования производительности, который дает информацию, как далеко вы можете подтянуть приложение, пока оно не упадет. Вполне возможно, что оно потребляет большое количество мощности процессора, памяти или пропускной способности сети или оно становится слишком медленным для пользователей и не обеспечивает надежность. Однако метрики, которые оно обеспечивает, можно использовать для тонкой настройки предоставляемых приложению ресурсов. В особых случаях оно может побудить вас изменить дизайн части системы, чтобы повысить ее эффективность.

Нагрузочное тестирование влечет за собой больше инфраструктуры и стратегии, чем, например, юнит-тестирование. Для того чтобы получить четкое представление о производительности в условиях нагрузки, нужно выполнить точную имитацию производственной среды в архитектуре и поведения пользователя. Из-за сложности нагрузочного тестирования

на прикладном уровне оно, на мой взгляд, находится в пирамиде тестирования где-то выше интеграционного тестирования (рис. 5.6).

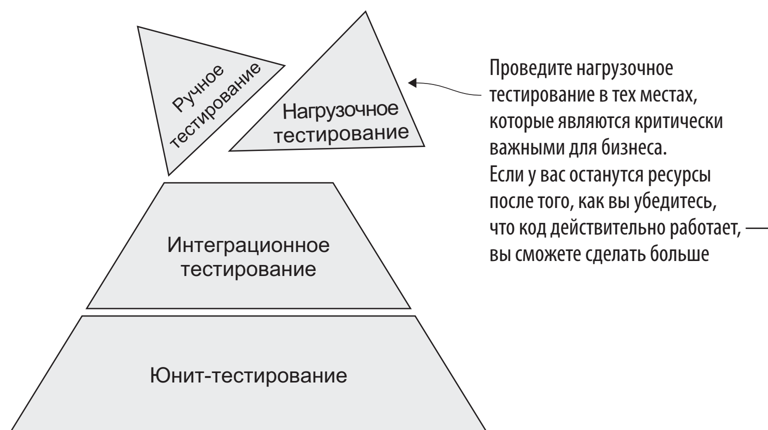


Рис. 5.6. Нагрузочное тестирование в пирамиде тестирования

Нагрузочное тестирование помогает тестировать производительность приложений в сценариях, которые более точно имитируют поведение реальных пользователей.

5.7. РАЗРАБОТКА НА ОСНОВЕ ТЕСТОВ: ПРИМЕР

Вокруг разработки с использованием юнит- и интеграционного тестирования существует целое научное направление. Общим названием указанной методологии является разработка на основе тестирования (TDD, test-driven development). Разработка на основе тестов помогает вам с самого начала погрузиться в тестирование, суля выгоды.

5.7.1. Это образ мышления

Для меня настоящая выгода от разработки на основе тестов состоит в образе мышления, который она формирует. Инженер по безопасности всегда найдет в коде что-то, что можно сломать. Обычно об этом говорится с пренебрежением, но я думаю, что это замечательно. Перечисление

всех путей, когда система может взорваться, одновременно приносит пользу и впечатляет.

Компания Netflix доводит это до крайности с помощью идеи инженерии хаоса. Они активно размышляют о путях, когда системы могут давать сбой, но при этом также вводят некоторое число невероятных случаев.[1] Это приводит к инновационным способам реагирования на системный сбой.

Когда вы пишете тесты, старайтесь быть инженером хаоса и думать о крайностях, которые код способен выдержать, и намеренно набрасывайте их на него. Разумеется, всему есть предел — нет смысла добиваться, чтобы весь код предсказуемо откликался на все входные данные, однако на редкие или неожиданные можно — благодаря системе исключений в Python.

5.7.2. Это философия

Вокруг разработки на основе тестов существует целая субкультура, и мнениями, более категоричными, чем те, как сделать правильно, являются те, как *не* сделать правильно. Это настоящее искусство, где столько же стилей и критиков, сколько и в любом другом движении. Я наблюдаю за тем, как разные команды справляются с тестовыми аспектами своего процесса, и какие-то варианты решений использую в своей работе.

Некоторые сторонники разработки на основе тестов рекомендуют каждую строку кода покрывать тестами. Хотя неплохо иметь сильное покрытие разных случаев, после определенного момента оно может привести к уменьшению отдачи. Иногда покрытие последних нескольких строк кода тестами означает введение более тесной сопряженности между тестами и реализацией со стороны интеграционного теста.

Если вы обнаружите, что тестирование какого-либо аспекта поведения функции становится неудобным или трудным, узнайте, обусловлено это плохим разделением ответственности или же его изначально неудобно

[1] Больше о достижениях Netflix в области инженерии хаоса можно найти по адресу https://medium.com/netflix-techblog/tagged/chaos-engineering.

тестировать. Если неудобство *должно быть* включено, то лучше, чтобы оно было в тестах, чем в реальном коде. Не делайте рефакторинг кода *только* для того, чтобы облегчить тестирование или усилить покрытие, — делайте его, чтобы облегчить тестирование *и* сделать код согласованным.

ИТОГИ

- Функциональные тесты обеспечивают ожидаемый результат от заданного элемента входных данных.
- Тестирование экономит время в долгосрочной перспективе, отлавливая ошибки и облегчая рефакторинг кода.
- Ручное тестирование не является масштабируемым и должно использоваться в дополнение к автоматизированному тестированию.
- `unittest` и `pytest` — это два популярных фреймворка юнит- и интеграционного тестирования кода на языке Python.
- Разработка на основе тестирования ставит тесты на первое место, направляя вас к рабочей реализации, основанной на требованиях.

Часть III

Организация крупных систем

Из части II вы узнали о концепциях, которые составляют основную часть дизайна ПО, а в части III начнете их применять. Создав приложение с нуля, вы увидите, как концепции дизайна ПО могут применяться в разных точках жизненного цикла его разработки.

Цель разработчика — создать работоспособное и быстрое ПО, которое он сам и другие разработчики смогут понимать и сопровождать. Эта часть книги покажет вам, что дизайн представляет собой итеративный процесс с некоторым пространством для маневра, где не всегда имеется правильный или неправильный ответ и редко видна точка «готово». Вы узнаете, как выявлять в коде слабые места, чтобы снизить усилия и повысить понимание.

6 Разделение ответственности на практике

В этой главе:

- ✓ Разработка приложения с разделением ответственности на высоком уровне.
- ✓ Использование специфических типов инкапсуляции для ослабления сопряженности.
- ✓ Создание хорошо разделенного фундамента для расширения в будущем.

В главе 2 я показал несколько самых лучших практик по разделению ответственности в Python. *Разделение ответственности* означает очерчивание контура вокруг кода, занятого конкретной операцией. Вы узнали, как функции, классы, модули и пакеты полезны при декомпозиции кода на фрагменты, о которых легче думать. Пришло время получить некоторый опыт применения инструментов, помогающих разделить ответственность.

Как и многие другие, я лучше учусь на практике. Когда я работаю над реальным проектом, то часто обнаруживаю связи, которых не видел раньше, или нахожу новые вопросы для изучения. Этим мы сейчас и займемся. Вы усовершенствуете начатый здесь проект в следующих главах и в конечном итоге получите то, что можно расширить для персонального использования.

ПРИМЕЧАНИЕ

В этой и следующих главах мы немного воспользуемся языком структурированных запросов SQL — доменно-специфичным языком для манипулирования данными и их извлечения из БД. Если вы раньше не использовали SQL или вам нужно освежить память, то предлагаю пробежаться по учебному пособию, прежде чем продолжить. Видеокурс Бена Брумма «SQL в движении» (*Ben Brumm, SQL in Motion,* www.manning.com/livevideo/sql-in-motion) станет хорошим вводным материалом.

6.1. ПРИЛОЖЕНИЕ КОМАНДНОЙ СТРОКИ ДЛЯ СОЗДАНИЯ ЗАКЛАДОК

В этой главе вы разработаете приложение для сохранения и организации закладок (подробнее об этом чуть позже).

Я не очень хорошо умею делать заметки. На протяжении всей учебы и карьеры я изо всех сил старался найти способ записывать для себя то, что помогает мне учиться и удерживать информацию. Когда я нахожу великолепный ресурс, который освещает концепцию с нового ракурса или содержит крутые примеры, я нахожу золотую жилу, но обычно приходится выделять время на чтение и практическую работу с информацией с этого ресурса. В результате за последние несколько лет я накопил массу закладок. Может быть, когда-нибудь у меня найдется время их прочитать!

Стандартное средство ведения закладок в большинстве браузеров отсутствует. Хотя материал можно вкладывать в проименованные папки, часто довольно трудно вспомнить, почему вы вообще его сохранили. Когда

я нахожу интересный репозиторий на GitHub, я пользуюсь функциональностью звезд хостинга GitHub, чтобы сохранять его на потом. Но звезды GitHub тоже ограничены — на момент написания книги они представляют собой один большой плоский список, который можно фильтровать только по языку программирования. Какие бы реализации ведения закладок вы ни использовали, они построены на одних и тех же принципах.

Закладки являются примером небольшого рабочего потока *CRUD*: создание, чтение, обновление и удаление (create, read, update, delete) (рис. 6.1). Эти четыре операции повсеместны. Вы можете *создать* закладку, чтобы сохранить ее на потом, а затем *прочитать* ее информацию, чтобы получить URL-адрес. А также можете *обновить* заголовок закладки, если тот, который вы дали ей изначально, сбивал с толку, и затем *удалить* ее, когда закончите с ней работать. Хорошее начало приложения!

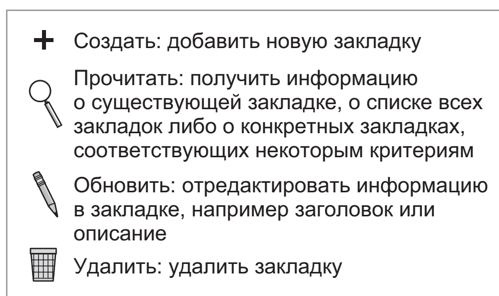


Рис. 6.1. Операции CRUD — основа многих приложений по управлению пользовательскими данными

Поскольку длинное описание отсутствует в некоторых существующих инструментах ведения закладок, ваше приложение сразу включит ее. В последующих главах вы продолжите добавлять ей нужные характеристики.

6.2. ОБЗОР ПРИЛОЖЕНИЯ BARK

Вы собираетесь разработать приложение командной строки для ведения закладок под названием Bark (примерно переводится как «слоеная

оболочка», «корка»). Оно позволяет создавать закладки, которые на данный момент будут состоять из нескольких фрагментов информации:

- *ИД* — уникальный числовой идентификатор каждой закладки.
- *Заголовок* — краткое текстовое название закладки, например GitHub.
- *URL-адрес* — ссылка на сохраненную статью или веб-сайт.
- *Примечание* — необязательное длинное описание или пояснение.
- *Дата добавления* — метка времени, показывающая, сколько закладке лет (доктор-прокрастинолог уже выехал).

Приложение Bark также позволяет выводить список всех добавленных закладок, а затем удалять одну из них по идентификатору. Оно управляется через *интерфейс командной строки* (CLI, command-line interface) — приложение, с которым вы взаимодействуете в терминале. При запуске CLI приложение Bark представит меню вариантов действий. Каждый выбранный вариант запустит действие, которое будет читать или модифицировать БД.

> **ПРИМЕЧАНИЕ**
>
> Об *обновлении* закладок читайте в главе 7.

6.2.1. Выгоды от разделения: реприза

Даже несмотря на то что CRUD-подобные операции, которые поддерживаются Bark, встречаются довольно часто, в приложении такой величины важно помнить, какие выгоды принесет разделение ответственности:

- *Уменьшенное дублирование* — если каждый фрагмент ПО делает одну вещь, то будет легче увидеть, когда два фрагмента делают одно и то же. Проанализируйте подобные фрагменты кода, чтобы понять, имеет ли смысл объединить их.
- *Улучшенная сопроводимость* — код читается гораздо чаще, чем пишется. Код, который является последовательным, в связи с тем, что каждый фрагмент имеет четкую ответственность, позволяет

разработчикам заскакивать в интересующие их участки, узнавать то, что им нужно, и выскакивать обратно.

- *Простота обобщения и расширения* — код с одной зоной ответственности может быть обобщен с другими вариантами использования либо разделен для поддержки более разнообразного поведения. Коду, который делает много всего, трудно быть гибким.

Помните об этих идеях во время отработки упражнения этой главы. Моя цель — дать вам то, что вы сможете развивать. Поэтому сначала подумайте о высокоуровневой архитектуре, которая будет поддерживать результат, а затем реализуйте ее.

6.3. ПЕРВОНАЧАЛЬНОЕ РАЗДЕЛЕНИЕ ОТВЕТСТВЕННОСТИ

Я стараюсь начинать разработку приложений, таких как Bark, с краткого объяснения того, как и что они делают. Это приводит меня к первоначальной архитектуре.

Например, как работает Bark? Как кратко описать его? Возможно, следующее утверждение содержит ответы на эти вопросы: *используя интерфейс командной строки, пользователь выбирает варианты действий, в частности добавить, удалить и вывести список закладок, хранящихся в БД*.

А теперь давайте немного разберемся:

- *Интерфейс командной строки* — способ представить пользователю варианты действий и понять, какие варианты он выбирает.
- *Выбор вариантов действий* — запуск какого-то действия или бизнес-логики.
- *Хранение в БД* — размещение данных для последующего использования.

Эти точки представляют высокоуровневые слои абстракции Bark. Интерфейс командной строки (CLI) — это слой *визуализации*. БД — это слой постоянства данных. Действия и *бизнес-логика* — это своего рода клей, который соединяет слои *визуализации* и *постоянства данных*. Каждый из

них имеет свою зону ответственности (рис. 6.2). Этот вид *многослойной архитектуры*, где каждый слой (ярус) приложения имеет возможность эволюционировать, используется многими компаниями. Команды разработчиков могут собираться вокруг каждого яруса, основываясь на областях специализации, и каждый слой может потенциально использоваться с другими приложениями многократно, если это необходимо.

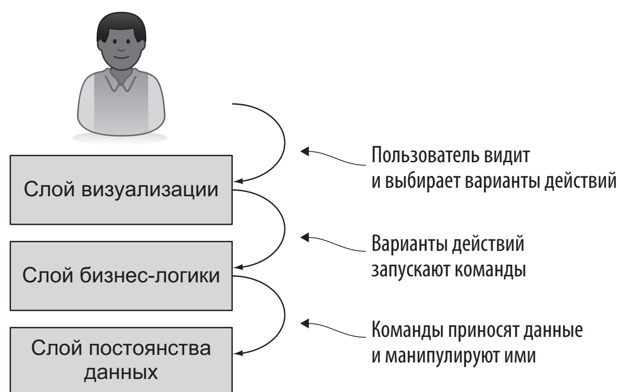


Рис. 6.2. Многослойная архитектура часто используется в веб- и настольных приложениях

ПАТТЕРНЫ АРХИТЕКТУРЫ ПРИЛОЖЕНИЯ

Паттерн разделения приложений на слои визуализации, постоянства данных и действий (или правил) встречается очень часто. Некоторые варианты этого подхода настолько распространены, что получили собственные названия. *Модель-представление-контроллер* (MVC, model-view-controller) — это способ моделирования данных для их постоянства, обеспечивающий пользователю их представление и позволяющий ему управлять их изменениями с помощью некоторого набора действий. *Модель-представление-модель представления* (MVVM, model-view-viewmodel) делает акцент на том, чтобы позволить представлению и модели данных свободно взаимодействовать. Эти и другие многослойные архитектуры являются прекрасными примерами разделения ответственности.

В этой главе вы разобьете каждый из этих слоев Bark. Поскольку каждый из них имеет свою зону ответственности, рассматривайте их как отдельные модули Python:

- модуль `database` (модуль БД);
- модуль `commands` (модуль команд);
- модуль `bark`, содержащий код, который фактически *выполняет* приложение Bark.

Мы начнем со слоя постоянства данных и будем продвигаться вверх.

6.3.1. Слой постоянства данных

Слой постоянства данных является нижним слоем Bark (рис. 6.3). Этот слой будет заниматься получением информации и ее передачей в БД.

Рис. 6.3. Слой постоянства данных занимается хранением данных — это самый нижний слой приложения

Вы будете использовать межплатформенную систему управления БД SQLite, которая по умолчанию хранит данные в одном-единственном файле (www.sqlite.org/index.html). По сравнению с более сложными системами управления базами данных (СУБД) это удобнее, потому что можно начать с нуля, удалив файл, если что-то пойдет не так.

> **ПРИМЕЧАНИЕ**
>
> Несмотря на то что SQLite — одна из самых широко используемых СУБД, по умолчанию она устанавливается только в некоторых операционных системах. Рекомендую вам скачать ее скомпилированный двоичный файл с официальной веб-страницы (https://sqlite.org/download.html).

Начиная с модуля `database`, вы создадите класс `DatabaseManager` для управления данными в БД. Python предоставляет встроенный модуль `sqlite3`, который можно использовать для соединения с БД, выполнения запросов и перебора результатов. БД в SQLite обычно представляют собой единственный файл с расширением .db, поэтому если вы создадите `sqlite3`-соединение с несуществующим файлом, то модуль создаст этот файл за вас.

Модуль `database` предоставляет большинство из того, что вам нужно для управления данными закладок, включая следующее:

- создание таблицы (для инициализации БД);
- добавление или удаление записи;
- вывод списка записей из таблицы;
- отбор и сортировка записей из таблицы на основе некоторых критериев;
- подсчет числа записей в таблице.

Как разложить эти задачи дальше? Каждая из них выделена с точки зрения описанной ранее бизнес-логики, но как быть со слоем постоянства данных? Большинство описанных действий достигаются путем конструирования инструкции SQL и ее выполнения, которое требует соединения с БД через указание пути к ее файлу.

Управление постоянством данных — задача высокого уровня и, отрывая слой постоянства данных, вы будете вынуждены работать с высокоуровневыми задачами. Они должны быть разделены. Но обо всем по порядку — прежде всего нужно соединиться с БД.

РАБОТА С БД

Умные люди создали надежные пакеты, которые облегчают работу с БД в Python. Пакет SQLAlchemy (www.sqlalchemy.org) широко используется не только для взаимодействия с БД, но и для абстрагирования моделей данных с помощью *объектно-реляционного отображения* (ОРМ, object-relational mapping). Это отображение позволяет рассматривать записи БД как объекты, не беспокоясь о деталях БД. Веб-фреймворк Django тоже обеспечивает механизм объектно-реляционного отображения для написания моделей данных.

Вы сами напишете код взаимодействия с БД. Он будет ограничен областью применения Bark, но его можно будет дополнять или менять, если вы захотите сделать больше с остальной частью приложения. Если вам нужно будет использовать БД в будущих проектах, то подумайте, хотите ли вы писать код БД с нуля либо вместо этого использовать один из сторонних пакетов.

Создание и закрытие соединения с БД

Пока приложение Bark работает, ему нужно только одно соединение с БД, которое можно использовать многократно. Для его установки используйте функцию `sqlite3.connect`, принимающую путь к файлу интересующей БД.

Опять же, если файл не существует, он будет создан.

Метод `__init__` класса `DatabaseManager` должен:

1. Принять аргумент, содержащий путь к файлу БД (не кодируйте его жестко и разделите ответственность!).
2. Использовать путь к файлу БД для создания SQLite-соединения с помощью `sqlite3.connect(path)` и сохранить его как атрибут экземпляра.

Закрывайте соединение с БД в SQLite при завершении работы программы, чтобы ограничить возможность повреждения данных. Для симметрии

метод `__del__` класса `DatabaseManager` должен закрыть соединение с помощью метода `.close()`.

Вот что послужит фундаментом для выполнения инструкций.

```
import sqlite3


class DatabaseManager:
    def __init__(self, database_filename):
        self.connection = sqlite3.connect(database_filename)

    def __del__(self):
        self.connection.close()
```

Создает и сохраняет соединение с БД для последующего использования

Закрывает соединение, когда дело сделано, на всякий случай

Выполнение инструкций

Класс `DatabaseManager` нуждается в способе выполнения инструкций, и лучше всего инкапсулировать их в многоразовый метод из-за их общих черт, которые не стоит дублировать.

Инструкции SQL, которые возвращают данные, называются *запросами*. Модуль `sqlite3` управляет результатами запроса с использованием *курсора* для прокручивания результатов, возвращаемых инструкциями при выполнении. Инструкции, которые не являются запросами (`INSERT`, `DELETE` и т. д.), не возвращают результаты, но курсор управляет этим, возвращая пустой список.

Напишите метод `_execute` в классе `DatabaseManager`, который можно использовать для выполнения всех инструкций с помощью курсора, чтобы затем применять результаты там, где это необходимо. Метод `_execute` должен:

1) принять инструкцию в качестве аргумента;
2) получить курсор из соединения с БД;
3) выполнить инструкцию с помощью курсора (подробнее об этом чуть позже);
4) вернуть курсор, который сохранил результат выполненной инструкции (если таковой имеется).

```
def _execute(self, statement):
    cursor = self.connection.cursor()  ◄── Создает курсор
    cursor.execute(statement)  ◄── Использует курсор для выполнения инструкций SQL
    return cursor  ◄── Возвращает курсор, который сохранил результаты
```

Инструкции, которые не являются запросами, манипулируют данными, и если во время их выполнения произойдет сбой, данные могут повредиться. СУБД защищают от этого сценария с помощью средства, именуемого *транзакцией*. Если инструкция, выполняемая в рамках транзакции, завершится безуспешно или иным образом прервется, то СУБД откатится назад к своему последнему известному рабочему состоянию. Модуль `sqlite3` позволяет использовать объект `connection` для создания транзакции посредством *контекстного менеджера* — питоновского блока, использующего ключевое слово `with`, который обеспечивает некое специальное поведение, когда код входит в блок или выходит из него.

Обновите `_execute`, чтобы поместить создание, выполнение и возврат курсора `cursor` внутрь транзакции:

```
def _execute(self, statement):
    with self.connection:  ◄── Создает контекст транзакции БД
        cursor = self.connection.cursor()
        cursor.execute(statement)  ◄── Происходит внутри транзакции БД
        return cursor
```

Использование `.execute` внутри транзакции приведет вас, функционально говоря, к цели. Но для безопасности используйте в инструкциях SQL плейсхолдеры для реальных значений, чтобы не позволить пользователям атаковать через специально изготовленные запросы[1]. Обновите `_execute`, чтобы принять:

- инструкцию SQL в качестве строкового значения, возможно, содержащую плейсхолдеры;
- список значений для вставки в плейсхолдеры внутри инструкции.

Затем этот метод должен выполнить инструкцию, передав оба аргумента в метод `execute` курсора, который примет те же аргументы. Он должен выглядеть так:

---

[1] Об инъекции SQL читайте на https://ru.wikipedia.org/wiki/Внедрение_SQL_кода.

```
def _execute(self, statement, values=None):
        with self.connection:
            cursor = self.connection.cursor()
            cursor.execute(statement, values or [])
            return cursor
```

Аргумент values необязателен — у некоторых инструкций нет плейсхолдеров для вставки значений

Исполняет инструкцию, предоставляя плейсхолдерам все переданные внутрь значения

Теперь у вас есть соединение с БД и возможность выполнять произвольные инструкции на нем. Помните, что соединение управляется автоматически при создании экземпляра класса `DatabaseManager`, поэтому не нужно думать о том, как оно открывается и закрывается, если только вы не хотите его изменить. Теперь выполнение инструкции контролируется методом `_execute`, и поэтому вам также не нужно думать, как выполняется инструкция, — необходимо только указать, *какую* инструкцию выполнить. В этом и заключается мощь разделения ответственности.

Теперь, когда у вас есть эти строительные блоки, пришло время разработать несколько взаимодействий с БД.

Создание таблиц

Одна из первых вещей, которые вам понадобятся, — это таблица БД, в которой будут храниться данные закладок. Придется *создать* ее с помощью инструкции SQL. Поскольку зоны ответственности за соединение с БД и выполнение инструкций теперь абстрагированы, работа по созданию таблицы будет включать в себя следующее:

1) определиться с именами столбцов;
2) определиться с типом данных для каждого столбца;
3) создать правильную инструкцию SQL для создания таблицы с этими столбцами.

Помните, что каждая закладка имеет идентификатор, заголовок, URL-адрес, необязательные примечания и дату ее добавления. Ниже приведены типы данных и ограничения для каждого столбца:

- *ID* — идентификатор является *первичным ключом* таблицы, или главным идентификатором каждой записи. Он должен автоматически наращиваться при каждом добавлении новой записи,

с помощью ключевого слова `AUTOINCREMENT`. Этот столбец имеет целочисленный тип `INTEGER`, остальные имеют тип `TEXT`.

- *Заголовок* (Title) является обязательным. Вы можете сообщить SQLite, что столбец не может быть пустым, используя ключевое слово `NOT NULL`.
- *URL-адрес* тоже является обязательным, и поэтому тоже получает `NOT NULL`.
- *Примечания* (Notes) необязательны и им нужен только спецификатор типа `TEXT`.
- *Дата* (Date) добавления закладки обязательна и получает тип `NOT NULL`.

Инструкция создания таблицы в SQLite использует ключевые слова `CREATE TABLE` с последующим именем таблицы, списком столбцов и информацией о типе данных в круглых скобках. Чтобы приложение Bark создавало таблицу при запуске, если она еще не существует, примените инструкцию `CREATE TABLE IF NOT EXISTS`.

Итак, как будет выглядеть инструкция SQL для создания таблицы закладок `bookmarks`? Посмотрите, сможете ли вы ее написать, а затем вернитесь, чтобы сверить свою работу со следующим листингом.

Листинг 6.1. Инструкция создания таблицы закладок bookmarks

```
CREATE TABLE IF NOT EXISTS bookmarks
(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    url TEXT NOT NULL,
    notes TEXT,
    date_added TEXT NOT NULL
);
```

ID каждой записи, который автоматически наращивается по мере добавления записей

NOT NULL требует, чтобы столбец был заполнен значениями

Теперь вы можете написать свой метод для создания таблиц. Каждый столбец идентифицируется именем, например `title`, которое увязывается с типом данных и ограничениями, например `TEXT NOT NULL`, поэтому словарь будет подходящим типом Python для представления столбцов.

Метод должен:

1) принять два аргумента: имя создаваемой таблицы и словарь имен столбцов, увязанных с типами и ограничениями;
2) сконструировать инструкцию SQL `CREATE TABLE`, как показано выше;
3) исполнить эту инструкцию с вызовом `DatabaseManager._execute`.

Попробуйте написать метод `create_table` прямо сейчас, а затем сравните его со следующим листингом.

Листинг 6.2. Создание таблицы БД SQLite

```
def create_table(self, table_name, columns):
        columns_with_types = [
            f'{column_name} {data_type}'
            for column_name, data_type in columns.items()
        ]
        self._execute(
            f'''
            CREATE TABLE IF NOT EXISTS {table_name}
            ({', '.join(columns_with_types)});
            '''
        )
```

Конструирует определения столбцов с их типами и ограничениями

Конструирует полную инструкцию создания таблицы и выполняет ее

ОБ ОБОБЩЕНИИ

Сейчас для Bark нужна только таблица `bookmarks`. Я уже доказывал в этой книге, что ранняя оптимизация является грубейшим просчетом, и то же самое верно для обобщения. Тогда зачем вообще использовать метод `create_table`?

Когда я начинаю проектировать метод с жестко закодированными значениями, то проверяю, много ли работы нужно сделать, чтобы параметризовать эти значения аргументами метода. Например, не составит большого труда заменить строковое значение `'bookmarks'` на строковый аргумент `table_name`. То же самое относится и к столбцам и их типам. Используя этот подход, метод `create_table` можно сделать достаточно общим, чтобы получить возможность создать почти любую таблицу.

Позже вы будете использовать этот метод для создания таблицы `bookmarks`, с которой будет взаимодействовать Bark, для управления закладками при разработке вашего приложения.

Добавление записей

Теперь, когда вы можете создавать таблицу, должна быть возможность добавлять в нее записи закладок (рис. 6.4).

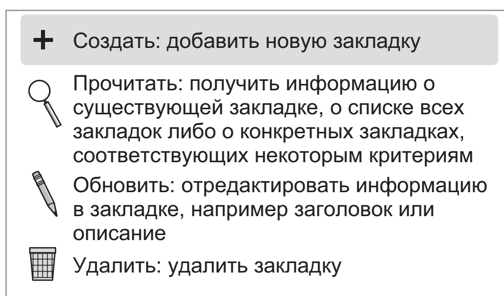


Рис. 6.4. Создание — самая элементарная операция, необходимая для CRUD, и суть многих систем

SQLite ожидает, что ключевые слова `INSERT INTO` с последующим именем таблицы будут указывать на намерение добавить новую запись. Далее последует список столбцов — ключевое слово `VALUES`, а затем значения в скобках. Инструкция вставки записи в SQLite выглядит так:

```
INSERT INTO bookmarks
(title, url, notes, date_added)
VALUES ('GitHub', 'https://github.com',
➥ 'Место для хранения репозиториев кода', '2019-02-01T18:46:32.125467');
```

Неплохо применить плейсхолдеры, как в предыдущем методе `_execute`. Но где?

1. `bookmarks`?
2. `title`, `url` и т. д.?
3. `'GitHub'`, `'https://github.com'` и т. д.?
4. Все вышеперечисленное?

Плейсхолдеры можно использовать только в тех местах инструкций, где располагаются литеральные значения, поэтому правильный ответ — 3. Инструкция `INSERT` для таблицы `bookmarks` с плейсхолдерами выглядит так:

```
INSERT INTO bookmarks
(title, url, notes, date_added)
VALUES (?, ?, ?, ?);
```

Чтобы ее сконструировать, нужно написать метод `add` в классе `DatabaseManager`, который:

1) принимает два аргумента: имя таблицы и словарь, увязывающий имена и значения столбцов;
2) конструирует строку плейсхолдеров (`?` для каждого столбца);
3) конструирует строку имен столбцов;
4) возвращает значения столбцов в форме кортежа (`.values()` словаря возвращает объект `dict_ values`, который, как оказывается, не работает с методом `execute` модуля `sqlite3`);
5) выполняет инструкцию с помощью `_execute`, передавая ее SQL с плейсхолдерами и значениями столбцов в качестве отдельных аргументов.

Напишите метод `add` прямо сейчас и сравните его со следующим листингом.

Листинг 6.3. Добавление записи в таблицу БД SQLite

```
def add(self, table_name, data):
        placeholders = ', '.join('?' * len(data))
        column_names = ', '.join(data.keys())        ← Ключами являются имена столбцов
        column_values = tuple(data.values())       ← .values() возвращает объект dict_values, но execute требует списка или кортежа

        self._execute(
            f'''
            INSERT INTO {table_name}
            ({column_names})
            VALUES ({placeholders});
            ''',
            column_values,        ← Передает необязательный аргумент values в execute
        )
```

Использование предикатных условий для ограничения области действия

Информация для вставки в БД некоторыми инструкциями используется в тандеме с одним или несколькими *предикатными условиями*, которые влияют на то, с какими записями будет работать инструкция. Например, инструкция `DELETE` без предикатного условия может привести к удалению всех записей в таблице. Вы же этого не хотите?

Предикатные условия `WHERE` могут добавляться в несколько видов инструкций, чтобы ограничить действия инструкций определенными критериями. Вы можете объединять несколько критериев `WHERE`, используя `AND` или `OR`. В Bark, например, каждая запись закладки имеет ID, поэтому ограничить инструкцию действием на конкретную запись по ее ID можно с помощью предикатного условия вроде `WHERE id = 3`.

Эти ограничения полезны как для запросов (поиска записей), так и для регулярных инструкций. Еще предикатные условия помогают удалять конкретные записи.

Удаление записей

После того как закладка себя изжила, нужно ее удалить (рис. 6.5). Для этого можно создать инструкцию `DELETE` для БД, используя предикатное условие `WHERE` для указания закладки по ее ID.

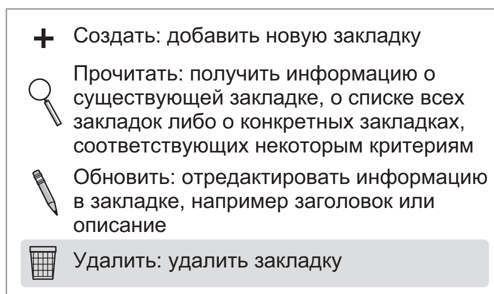


Рис. 6.5. Удаление является противоположностью создания, поэтому большинство систем поддерживают эту операцию

В SQLite инструкция удаления закладки с идентификатором `3` выглядит так:

```
DELETE FROM bookmarks
WHERE ID = 3;
```

Как и в методах `create_table` и `add`, вы можете представить критерии в форме словаря для сопоставления имен столбцов со значением на удаление. Напишите метод удаления `delete`, который:

1) принимает два аргумента: имя таблицы, из которой удаляются записи, и словарь, увязывающий имена столбцов со значением на удаление. Критерии должны быть обязательным аргументом, чтобы не удалить сразу все записи;

2) конструирует строку плейсхолдеров для предикатного условия `WHERE`;

3) конструирует полный запрос `DELETE FROM` и выполняет его с помощью `_execute`.

Сверьте свои результаты со следующим листингом.

Листинг 6.4. Удаление записей в SQLite

```
def delete(self, table_name, criteria):
        placeholders = [f'{column} = ?' for column in criteria.keys()]
        delete_criteria = ' AND '.join(placeholders)
        self._execute(
            f'''
            DELETE FROM {table_name}
            WHERE {delete_criteria};
            ''',
            tuple(criteria.values()),
        )
```

Аргумент criteria здесь обязателен, поскольку без критериев все записи будут удалены

Использует аргумент values метода _execute в качестве значений на удаление

Отбор и сортировка записей

Теперь вы можете добавлять и удалять записи из таблицы, но как их извлекать? Помимо создания и удаления информации, нужна возможность читать то, что сохранено (рис. 6.6).

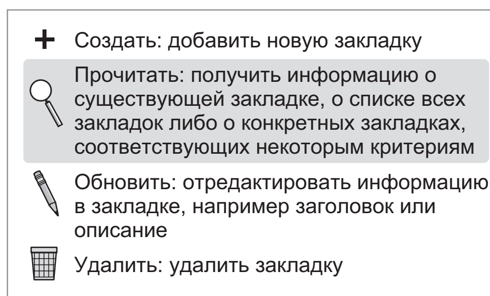


Рис. 6.6. Чтение существующих данных является необходимой частью CRUD

Вы можете создать инструкцию с запросом к SQLite, используя `SELECT * FROM bookmarks` (* означает «все столбцы») и несколько критериев:

```
SELECT * FROM bookmarks
WHERE ID = 3;
```

Вдобавок можно отсортировать эти результаты по определенному столбцу с помощью предикатного условия `ORDER BY`:

```
SELECT * FROM bookmarks
WHERE ID = 3
ORDER BY title;   ◄—— Упорядочивает результаты по столбцу title по возрастанию
```

Опять-таки используйте плейсхолдеры там, где в запросе есть литеральные значения:

```
SELECT * FROM bookmarks
WHERE ID = ?
ORDER BY title;
```

Ваш метод `select` будет выглядеть примерно так же, как метод `delete`, за исключением того, что критерии необязательны. (По умолчанию он будет извлекать все записи.) Он также должен принимать необязательный аргумент `order_by`, задающий столбец для сортировки результатов (по умолчанию используется первичный ключ таблицы). Используя `delete` в качестве руководства, напишите `select` прямо сейчас и вернитесь, чтобы сравнить его со следующим листингом.

Листинг 6.5. Метод отбора данных из таблицы SQL

```
def select(self, table_name, criteria=None, order_by=None):
        criteria = criteria or {}

        query = f'SELECT * FROM {table_name}'

        if criteria:
            placeholders = [f'{column} = ?' for column in
                            criteria.keys()]
            select_criteria = ' AND '.join(placeholders)
            query += f' WHERE {select_criteria}'

        if order_by:
            query += f' ORDER BY {order_by}'

        return self._execute(
            query,
            tuple(criteria.values()),
        )
```

По умолчанию критерии могут быть пустыми, потому что совершенно нормально, если в таблице отбираются все записи

Конструирует предикатное условие ORDER BY для сортировки результатов

На этот раз вы хотите вернуть значение из _execute для прокручивания результатов в цикле

Конструирует предикатное условие WHERE для ограничения результатов

Теперь вы создали соединение с БД — написали как метод `_execute` для выполнения произвольных инструкций SQL с плейсхолдерами в транзакции, так и методы для добавления, запроса и удаления записей. Это почти все, что нужно для манипулирования БД SQLite на данный момент. Вы только что завершили написание менеджера БД менее чем в 100 строках кода. Отличная работа.

Далее разработаем бизнес-логику, которая будет взаимодействовать со слоем постоянства данных.

6.3.2. Слой бизнес-логики

Теперь, когда слой постоянства данных для Bark находится на своем месте, вы можете заняться слоем, на котором выясняется, что нужно вставлять и что получать из слоя постоянства данных (рис. 6.7).

Когда пользователь взаимодействует с чем-то в слое визуализации, приложение Bark должно вызывать что-то в слое бизнес-логики и затем в слое постоянства данных. Заманчиво сделать следующее:

```
if user_input == 'add bookmark':
    # добавить закладку
elif user_input == 'delete bookmark #4':
    # удалить закладку
```

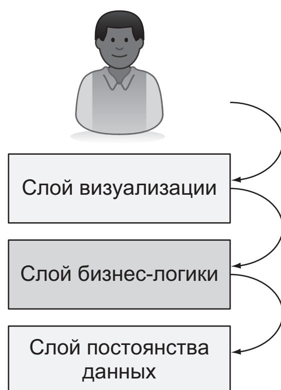


Рис. 6.7. Слой бизнес-логики определяет, когда и как данные читаются из слоя постоянства данных или пишутся туда

Но теперь представляемый пользователю текст сопряжен с действиями, которые должны быть запущены. В меню будут новые условия для каждого варианта действий, и если вы захотите, чтобы несколько вариантов запускали одну и ту же команду, или вам понадобится изменить текст, вам придется проводить рефакторинг какого-то кода. Было бы неплохо, если бы слой визуализации был единственным местом, которое знает о тексте варианта действий, показываемого пользователю в меню.

Каждое действие — это своего рода *команда*, которая должна выполняться в ответ на выбор пользователем варианта из меню. Инкапсулируя логику каждого действия в форме командного объекта и обеспечивая согласованный способ их запуска с помощью метода `execute`, вы отсоедините эти действия от слоя визуализации. Тогда слой визуализации будет направлять варианты действий из меню к командам, не беспокоясь о том, как эти команды работают. Такой подход называется *паттерном Команда*.[1]

[1] https://ru.wikipedia.org/wiki/Команда_(шаблон_проектирования)

Вы разработаете каждое действие CRUD и некоторую периферийную функциональность в качестве команд в слое бизнес-логики.

Создание таблицы bookmarks

В слое бизнес-логики создайте новый модуль `commands` для размещения команд, которые собираетесь написать. Поскольку большинство команд будут использовать класс `DatabaseManager`, импортируйте его из модуля `database` и создайте его экземпляр `db`, который будет использоваться во всем модуле `commands`. Помните, что его метод `__init__` требует путь к файлу БД SQLite (предлагаю назвать его bookmarks.db). Если пропустить любой из названных шагов, файл БД будет создан в том же каталоге, что и код Bark.

Чтобы инициализировать таблицу закладок `bookmarks` БД, если она еще не существует, напишите класс `CreateBookmarksTableCommand`, метод `execute` которого создаст таблицу закладок. Либо воспользуйтесь методом `db.create_table`, который написали ранее. Далее в этой главе вы будете вызывать эту команду при запуске Bark. Сверьте свою работу со следующим листингом.

Листинг 6.6. Команда для создания таблицы

```
db = DatabaseManager('bookmarks.db')  ◄── Помните, что модуль sqlite3 автоматически
                                          создаст файл БД, если тот не существует

class CreateBookmarksTableCommand:
    def execute(self):  ◄── Будет вызвано при запуске приложения Barks
        db.create_table('bookmarks', {  ◄── Создает таблицу bookmarks с необходимыми столбцами и ограничениями
            'id': 'integer primary key autoincrement',
            'title': 'text not null',
            'url': 'text not null',
            'notes': 'text',
            'date_added': 'text not null',
        })
```

Обратите внимание, что команда осведомлена только о своих задачах (вызывая логику слоя постоянства данных) и об интерфейсе метода `DatabaseManager.create_table`, от которого она зависит. Это пример слабой сопряженности, полученной отчасти благодаря разделению логики постоянства данных и в конечном счете логики визуализации. По мере отработки этих упражнений вы должны более ясно видеть выгоды от разделения ответственности.

Добавление закладок

Чтобы добавить закладку, нужно передать данные, полученные из слоя визуализации, в слой постоянства данных. Данные будут передаваться в форме словаря, увязывающего имена столбцов со значениями. Это отличный пример кода, опирающегося не на специфику имплементации, а на совместный интерфейс. Если слой постоянства данных и слой бизнес-логики достигают согласия о формате данных, то каждый из них может делать то, что ему нужно, до тех пор, пока формат данных остается согласованным.

Напишите класс `AddBookmarkCommand` для выполнения этой операции. Он будет:

1) ожидать словарь, содержащий заголовок, URL-адрес и, возможно, примечание;

2) добавлять текущую дату и время в словарь в `date_added`. Чтобы получить текущее время в стандартизированном формате UTC, используйте `datetime.datetime.utcnow().isoformat()`[1];

3) вставлять данные в таблицу `bookmarks` методом `DatabaseManager.add`;

4) возвращать сообщение об успехе, которое потом будет показываться в слое визуализации.

Сверьте свою работу со следующим листингом.

Листинг 6.7. Команда для добавления закладки

```
from datetime import datetime

...

class AddBookmarkCommand:
    def execute(self, data):
        data['date_added'] = datetime.utcnow().isoformat()
        db.add('bookmarks', data)
        return 'Закладка добавлена!'
```

Добавляет текущую дату и время при добавлении записи

Использование метода .add класса DatabaseManager укорачивает добавление записи

Вы будете использовать это сообщение позже в слое визуализации

Теперь вы написали всю бизнес-логику, необходимую для создания закладок. Пора выводить список закладок, которые вы добавили.

[1] https://ru.wikipedia.org/wiki/ISO_8601

Вывод списка закладок

Приложение Bark должно уметь показывать сохраненные закладки — это его назначение. Напишите команду ListBookmarksCommand для вывода на экран закладок из БД.

За получение закладок из БД отвечает метод DatabaseManager.select. По умолчанию SQLite сортирует записи по порядку их создания (то есть по первичному ключу таблицы), но также, возможно, будет полезно сортировать закладки по дате или заголовку. Идентификаторы закладок и даты сортируются идентично, потому что они оба увеличиваются строго по мере добавления закладок, но неплохой практикой будет явная сортировка по интересующему столбцу в случае, если тот изменяется.

Команда ListBookmarksCommand должна делать следующее:

- принимать столбец для упорядочения и сохранять его как атрибут экземпляра. Если хотите, можете установить в date_added дефолтное значение;
- передавать эту информацию в db.select в его методе execute;
- возвращать результат (используя метод .fetchall() курсора), потому что select — это запрос.

Напишите команду для вывода списка закладок и вернитесь, чтобы сверить свой результат со следующим листингом.

Листинг 6.8. Команда для вывода на экран списка существующих закладок

```
class ListBookmarksCommand:
    def __init__(self, order_by='date_added'):
        self.order_by = order_by

    def execute(self):
        return db.select('bookmarks', order_by=self.order_by).fetchall()
```

Вы можете создать версию этой команды для сортировки по дате и по заголовку

db.select возвращает курсор, который вы можете прокрутить в цикле для получения записей

Теперь у вас есть достаточно возможностей для добавления новых закладок и просмотра существующих. Последним шагом в управлении закладками станет команда их удаления.

Удаление закладок

Подобно добавлению закладки, ее удаление требует передачи некоторых данных из слоя визуализации. Однако на этот раз данные будут просто целочисленным значением, представляющим ID удаляемой закладки.

Напишите команду `DeleteBookmarkCommand`, которая принимает эту информацию в своем методе `execute` и передает ее в метод `DatabaseManager.delete`. Помните, что `delete` принимает словарь, увязывающий имена столбцов со значениями на удаление, поэтому нужно будет сопоставить заданное значение в столбце `id`. После того как запись будет удалена, верните сообщение об успехе для использования в слое визуализации.

Вернитесь и сверьте свою работу со следующим листингом.

Листинг 6.9. Команда для удаления закладок

```
class DeleteBookmarkCommand
:
    def execute(self, data):
        db.delete('bookmarks', {'id': data})
        return 'Bookmark deleted!'
```

delete принимает словарь имен столбцов и сопоставляет пары значений

Выход из Bark

Осталась команда для выхода из Bark. Пользователь может применить обычный метод Ctrl-C для остановки программы Python, но вариант из меню будет немного приятнее.

Python предоставляет функции `sys.exit` для остановки выполнения программы. Напишите команду `QuitCommand` и метод `execute`, которой завершит работу, используя этот подход, а затем вернитесь и сверьте свой код со следующим листингом.

Листинг 6.10. Команда для выхода из программы

```
import sys

...

class QuitCommand
:
    def execute(self):
        sys.exit()
```

Приводит к немедленному выходу из Bark

Теперь можно вытереть пот со лба. Однако это еще не конец, и далее мы разработаем слой визуализации.

6.3.3. Слой визуализации

В Bark используется интерфейс командной строки (CLI). Его слой визуализации (та часть, которую видит пользователь, как показано на рис. 6.8) представляет собой текст в терминале. В зависимости от приложения интерфейс командной строки может работать до выполнения конкретной задачи или до явного завершения работы пользователем. Вы написали команду `QuitCommand` — догадываетесь, что дальше?

Рис. 6.8. Слой визуализации показывает пользователям, какие действия можно предпринять и как их запускать

Слой визуализации Bark содержит бесконечный цикл:

1) очищает экран;
2) выводит элементы меню;
3) получает выбранный пользователем вариант;
4) очищает экран и выполняет команду по выбору пользователя;
5) ждет до тех пор, пока пользователь не просмотрит результат и не нажмет Enter.

В слое визуализации нужно создать новый модуль `bark`. Поместив код для приложений командной строки в блок `if name == 'main':`, вы исключите случайное выполнение кода в модуле при импортировании модуля `bark` куда-то еще. Если вы начнете с программы типа `Здравствуй, мир!`, то быстро убедитесь, что все настроено правильно.

Начните модуль `bark` со следующего:

```
if __name__ == '__main__':
    print('Добро пожаловать в Bark!')
```

Попробуйте выполнить python bark.py в терминале, и вы увидите: `Добро пожаловать в Bark!`. Теперь можно подключить слой визуализации к некоторой бизнес-логике.

Инициализация БД

Помните, что приложение Bark инициализирует БД, создавая таблицу `bookmarks`, если она еще не существует. Импортируйте модуль `commands` и обновите код до выполнения команды `CreateBookmarksTableCommand`, как показано в следующем фрагменте кода. После внесения этого обновления и выполнения python bark.py на экране не будет текста, но вы увидите, что файл bookmarks.db создан.

```
import commands


if __name__ == '__main__':
    commands.CreateBookmarksTableCommand().execute()
```

Внешне это выглядит как нечто незначительное, но вы только что совершили полный проход через все слои вашей *многослойной архитектуры*. Слой визуализации (в виде выполнения модуля bark.py) запустил команду в бизнес-логике, которая, в свою очередь, настроила таблицу в слое постоянства данных, пригодную для хранения закладок. Каждый слой знает о своем окружении ровно столько, чтобы выполнять свою работу — все хорошо разделено и слабо сопряжено. Вы испытаете полный проход еще не раз, когда начнете добавлять варианты действий в меню Bark и будете запускать все больше команд.

Элементы меню

Когда вы запускаете Bark, приложение должно предоставить меню вариантов действий, которое выглядит примерно так:

```
(A) Добавить закладку
(B) Показать список закладок по дате
(T) Показать список закладок по заголовку
(D) Удалить закладку
(Q) Выйти
```

Каждый вариант содержит комбинацию клавиш и описательный заголовок. Если приглядеться, варианты соответствуют командам, которые вы написали ранее с использованием паттерна Команда, поэтому каждая команда может быть запущена так же, как и другие, — с помощью своего метода `execute`. Команды отличаются только тем, какие настройки и ввод они требуют, а затем с точки зрения слоя визуализации они работают одинаково.

Основываясь на знаниях об инкапсуляции, как бы вы подключили элементы из слоя визуализации к бизнес-логике, которую они контролируют?

1. Использовали бы условную логику для вызова метода `execute` правильного класса `Command` на основе пользовательского ввода.
2. Создали бы класс, который связывает текст, выводимый на экран пользователю, с командой, которую он запускает.

Рекомендую вариант 2. Для того чтобы подключить каждый вариант действий в меню к команде, которую он должен запускать, можно создать класс `Option`. Метод `__init__` указанного класса будет принимать имя, показываемое пользователю в меню, экземпляр команды, выполняемой по выбору пользователя, и необязательный подготовительный шаг (например, получение от пользователя дополнительных входных данных). Все это можно хранить как атрибуты экземпляра.

При выборе экземпляру класса `Option` нужно:

1. Выполнить заданный подготовительный шаг, если таковой имеется.
2. Передать значение, возвращаемое из подготовительного шага, если таковое имеется, в метод `execute` заданной команды.

3. Вывести результат выполнения в виде сообщения об успехе или информации о закладках, возвращаемых из бизнес-логики.

Экземпляр класса `Option` должен быть представлен в форме его текстового описания для пользователя. Вы можете применить `__str__` для переопределения дефолтного поведения. Абстрагирование этой работы от остальной части кода, который получает и проверяет допустимость вводимых пользователем данных, позволяет удерживать разделение ответственности.

Попробуйте написать класс `Option`, а затем сверьте его со следующим листингом.

Листинг 6.11. Подключение текста меню к командам бизнес-логики

```
class Option:
    def __init__(self, name, command, prep_call=None):
        self.name = name             ◄── Имя, показываемое в меню
        self.command = command       ◄── Экземпляр выполняемой команды
        self.prep_call = prep_call   ◄── Необязательный подготовительный шаг, который
                                         вызывается перед выполнением команды

    def choose(self):    ◄── Метод choose будет вызван, когда
                             вариант действия выбран из меню
        data = self.prep_call() if self.prep_call else None   ◄── Вызывает подготовительный шаг, если он указан
        message = self.command.execute(data) if data
➥ else self.command.execute()   ◄── Выполняет команду, переданную в данных из подготовительного шага, если таковая имеется
        print(message)

    def __str__(self):   ◄── Представляет вариант действия в форме имени
        return self.name     вместо дефолтного поведения Python
```

Имея класс `Option` на своем месте, подключите остальную бизнес-логику, которую создали ранее. Помните, что с каждым вариантом нужно:

1) вывести клавишу клавиатуры, которую пользователь нажмет, чтобы выбрать нужный вариант действия;

2) вывести текст варианта;

3) проверить соответствие пользовательского ввода какому-либо варианту действия и при их совпадении выбрать вариант.

Какая структура данных Python будет хорошо работать, поддерживая все варианты?

1. `list`;
2. `set`;
3. `dict`.

Каждая клавиша клавиатуры сопоставляется с вариантом действия в меню, поэтому проверьте соответствия пользовательского ввода с имеющимися вариантами и держите эти пары наготове. Ответ 3 — удачное решение, потому что `dict` может предоставлять пары «клавиша — вариант действия», которые можно прокручивать методом `.items()` словаря, для вывода текста варианта действия. Также рекомендую использовать словарь `collections.OrderedDict`, который позволяет выводить варианты действия в меню в указанном порядке.

Поместите словарь `options` после команды `CreateBookmarksTableCommand`, добавив в меню элемент для каждого варианта действия. Как только словарь будет на своем месте, создайте функцию `print_options`, которая будет перебирать варианты действия и выводить их в том формате, который вы видели ранее:

```
(A) Добавить закладку
(B) Показать список закладок по дате
(T) Показать список закладок по заголовку
(D) Удалить закладку
(Q) Выйти
```

Сверьте свой результат со следующим листингом.

Листинг 6.12. Детализация и вывод вариантов действий в меню

```
def print_options(options):
    for shortcut, option in options.items():
        print(f'({shortcut}) {option}')
    print()

...

if __name__ == '__main__':
    ...
```

```
    options = {
        'A': Option('Добавить закладку', commands.AddBookmarkCommand()),
        'B': Option('Показать список закладок по дате',
➥ commands.ListBookmarksCommand()),
        'T': Option('Показать список закладок по заголовку',
➥ commands.ListBookmarksCommand(order_by='title')),
        'D': Option('Удалить закладку',
                    commands.DeleteBookmarkCommand()),
        'Q': Option('Выйти', commands.QuitCommand()),
    }
    print_options(options)
```

Если после добавления в меню вариантов действий выполнить приложение Bark, то все добавленные варианты действий будут выведены на экране. Пока их нельзя вызывать, поскольку нужно получить пользовательский ввод.

Пользовательский ввод

Для нанизывания визуализации на бизнес-логику, а бизнес-логики на постоянство данных осталось добавить совсем немного — чуть-чуть интерактивности для пользователей Bark. Подход к получению желаемого пользователем варианта выглядит следующим образом:

1. Предложить пользователю ввести свой вариант действий, используя встроенную в Python функцию ввода `input`.
2. Если выбранный пользователем вариант совпадает с одним из перечисленных, то вызвать метод `choose` этого варианта.
3. В противном случае повторить всю процедуру.

Какой подход использовать в Python, чтобы получить повторяющееся поведение?

1. Цикл `while`.
2. Цикл `for`.
3. Вызов рекурсивной функции.

Поскольку исчерпывающее состояние для получения от пользователя входных данных отсутствует (он может ввести недопустимый вариант

четыре миллиарда раз), цикл `while` имеет наибольший смысл — продолжать предлагать пользователю ввести вариант до тех пор, *пока* этот вариант остается недопустимым. Или поступить проще — принять версии каждого варианта в верхнем и нижнем регистре.

Напишите функцию `get_option_choice` и используйте ее после вывода вариантов действий, чтобы получить выбранный пользователем вариант. Затем вызовите метод `choose` этого варианта. А затем сравните свою работу со следующим листингом.

Листинг 6.13. Получение выбранного пользователем варианта из меню

```
def option_choice_is_valid(choice, options):    ◄ Вариант является допустимым, если буква совпадает с одним из ключей в словаре options
    return choice in options or choice.upper() in options

def get_option_choice(options):
    choice = input('Выберите вариант действия: ')    ◄ Получает от пользователя первоначальный вариант
    while not option_choice_is_valid(choice, options):    ◄ Пока вариант пользователя остается допустимым, продолжать предлагать ему ввести данные
         print('Недопустимый вариант')
        choice = input('Выберите вариант действия: ')
    return options[choice.upper()]    ◄ Возвращает совпадающий вариант, когда сделан правильный выбор

if __name__ == '__main__':
    ...

    chosen_option = get_option_choice(options)
    chosen_option.choose()
```

Здесь можно выполнить приложение Bark, и некоторые команды, такие как вывод списка закладок и выход из системы, будут реагировать на ваш ввод. Но несколько вариантов, как я уже говорил, требуют дополнительной подготовки: указания заголовка, описания и т. д., чтобы добавить закладку, а также указания ID закладки, чтобы ее удалить. Вам нужно будет запросить у пользователя эти данные так же, как вы получали пользовательский ввод для выбора в меню варианта действий.

Есть еще одна возможность инкапсулировать поведение. По каждому требуемому фрагменту информации вы должны:

1. Предложить пользователю ввести метку — к примеру, «Заголовок» или «Описание».
2. Если информация является обязательной и пользователь нажимает Enter, не введя никакой информации, то продолжить ее запрашивать.

Напишите три функции — одну для обеспечения поведения с повторяющимся запросом информации и две берущие информацию от первой функции для добавления или удаления закладки. Затем добавьте каждую функцию извлечения информации в качестве `prep_call` в соответствующий экземпляр класса `Option`. Сверьте свои результаты со следующим листингом.

Листинг 6.14. Сбор информации о закладках от пользователя

```
def get_user_input(label, required=True):          ◄ Общая функция, которая предлагает пользователю ввести данные
    value = input(f'{label}: ') or None             ◄ Получает первоначальный ввод от пользователя
    while required and not value:                   ◄ При необходимости продолжает предлагать ввод до тех пор, пока входные данные остаются пустыми
        value = input(f'{label}: ') or None
    return value

def get_new_bookmark_data():                        ◄ Функция, которая получает необходимые данные для добавления новой закладки
    return {
        'title': get_user_input('Title'),
        'url': get_user_input('URL'),
        'notes': get_user_input('Notes', required=False),   ◄ Примечания для закладки являются необязательными, поэтому не продолжает предлагать их ввести
    }

def get_bookmark_id_for_deletion():                 ◄ Получает необходимую информацию для удаления закладки
    return get_user_input('Enter a bookmark ID to delete')

if __name__ == '__main__':
    ...
    'A': Option('Добавить закладку', commands.AddBookmarkCommand(),
➥ prep_call=get_new_bookmark_data),
    ...
    'D': Option('Удалить закладку', commands.DeleteBookmarkCommand(),
➥ prep_call=get_bookmark_id_for_deletion),
```

Если все хорошо, то теперь вы можете запустить приложение Bark и добавлять, смотреть список закладок или удалять их. Поздравляю! Вы отлично потрудились!

УГОЛОК ГИКА

Мы рассмотрели много материала, но хочу отметить кое-что важное. Если вы захотите добавить новую функциональность, вот вам четкий алгоритм.

1. Добавьте любые новые методы манипулирования БД, которые могут понадобиться, в модуль database.py.
2. Добавьте класс `command`, который выполняет требующуюся вам бизнес-логику, в модуль commands.py.
3. Подключите новую команду к новому варианту действия в меню в модуле bark.py.

Насколько это круто? Разделение ответственности позволяет четко понимать, какие участки кода необходимо дополнить при добавлении новой функциональности.

В завершение еще несколько слов по доработке.

Очистка экрана

Очистка экрана непосредственно перед печатью меню или выполнением команды облегчит пользователю просмотр текущего контекста. Чтобы очистить экран, обратитесь к программе командной строки операционной системы по очистке текста терминала, такой как `clear` или `cls` в Windows. Чтобы выяснить, работаете ли вы в Windows, проверьте, что значение `os.name` — это `'nt'`. (Windows NT относится к Windows 10 как macOS к Mojave.)

Напишите функцию `clear_screen`, которая выполняет соответствующий вызов, используя `os.system` как в следующем коде:

```
import os


def clear_screen():
    clear = 'cls' if os.name == 'nt' else 'clear'
    os.system(clear)
```

Вызывайте ее непосредственно перед вызовом `print_options` и метода `.choose()` выбранного пользователем варианта действия:

```
if __name__ == '__main__':
    ...

    clear_screen()
    print_options(options)
    chosen_option = get_option_choice(options)
    clear_screen()
    chosen_option.choose()
```

Она будет наиболее полезной, когда меню и результаты команд печатаются многократно.

Цикл приложения

Последний шаг состоит в том, чтобы гонять приложение Bark в цикле, давая пользователям возможность выполнять несколько действий подряд. Для этого создайте метод `loop` и переместите в него все, кроме инициализации БД из блока `if __name__ == '__main__'`. Вернувшись в блок `if __name__ == '__main__'`, вызовите метод `loop` внутри блока `while True:`. В конце метода `loop` добавьте строку кода для паузы и дождитесь, пока пользователь не нажмет клавишу Enter, прежде чем продолжить.

```
def loop():
    # Все шаги для показа / выбора вариантов действий
    ...
    _ = input('Нажмите ENTER для возврата в меню')

if __name__ == '__main__':
    commands.CreateBookmarksTableCommand().execute()

    while True:
        loop()
```

Все, что происходит для каждой итерации цикла меню > опция > результат уходит сюда

Предлагает пользователю нажать ENTER и просматривает результат перед продолжением работы (_ означает «неиспользуемое значение»)

Повторяет в бесконечном цикле (до тех пор, пока пользователь не выберет вариант, соответствующий команде QuitCommand)

Теперь приложение Bark даст пользователю возможность возвращаться в меню после каждого взаимодействия, а меню позволит выйти из приложения. Это все, что необходимо для успешной работы. Как вы думаете, этим приложением можно пользоваться? Я думаю, самое время.

ИТОГИ

- Разделение ответственности — это инструмент для достижения более удобочитаемого и сопроводимого кода.
- Приложения для конечных пользователей часто разделяются на слои постоянства данных, бизнес-логики и визуализации.
- Разделение ответственности тесно сотрудничает с инкапсуляцией, абстракцией и слабой сопряженностью.
- Применение эффективного разделения ответственности позволяет добавлять, изменять и удалять функциональность, не затрагивая окружающий код.

7 Расширяемость и гибкость

В этой главе:

- ✓ Использование инверсии управления для обеспечения гибкости кода.
- ✓ Использование интерфейсов для обеспечения расширяемости кода.
- ✓ Добавление новых функциональных характеристик в существующий код.

Во многих компаниях повседневная работа разработчика предусматривает не только написание новых приложений, но и обновление существующих. Добавление новой характеристики в существующее приложение — это *расширение* функциональности этого приложения путем добавления кода.

Некоторые приложения реагируют на изменения *гибко* и легко адаптируются к новым требованиям. Другие будут биться с вами, пуская в ход

зубы и когти. В этой главе вы найдете стратегии создания гибкого и расширяемого ПО: мы добавим в приложение Bark импорт звезд GitHub.

7.1. ЧТО ТАКОЕ РАСШИРЯЕМЫЙ КОД?

Код считается *расширяемым*, или пригодным для расширения, если добавление новых характеристик практически не влияет на существующие в нем формы поведения.

Подумайте о веб-браузере, например Google Chrome или Mozilla Firefox. Наверняка вы установили блокировщик рекламы в браузере или инструмент, позволяющий легко сохранять статью в программе ведения заметок вроде Evernote. Firefox называет такие устанавливаемые части программного обеспечения *надстройками*, а Chrome называет их *расширениями*. Они оба являются примерами системы плагинов. *Системы плагинов* — это реализации расширяемости. Chrome и Firefox не строились *специально* с учетом блокировщиков рекламы или Evernote, но они были спланированы так, чтобы позволять создавать такие расширения.

Массивные проекты, такие как веб-браузеры, будут успешны, если могут удовлетворять потребностям сотен тысяч пользователей. Предсказать все эти потребности невозможно, и разумно отвечать на них после того, как продукт выведен на рынок. Расширяемость ПО является дальновидным решением разработчика.

Как и во многих других аспектах разработки ПО, расширяемость — это диапазон, над которым вы будете работать. Практикуя разделение ответственности и слабую сопряженность, вы сможете со временем улучшать расширяемость кода и добавлять новые функциональные характеристики быстрее, не беспокоясь об их влиянии на окружающий код. Изолированность изменений также облегчает тестирование кода.

7.1.1. Добавление новых форм поведения

В предыдущей главе вы написали основы Bark и использовали многослойную архитектуру, разделяющую ответственность на сохранение закладок, манипулирование ими и вывод на экран, а также небольшой

набор функциональных характеристик поверх слоев абстракции. Готовы добавить новую функциональность?

В расширяемой системе добавление новых классов, методов, функций или данных должно инкапсулировать новое поведение без изменения существующего кода (рис. 7.1).

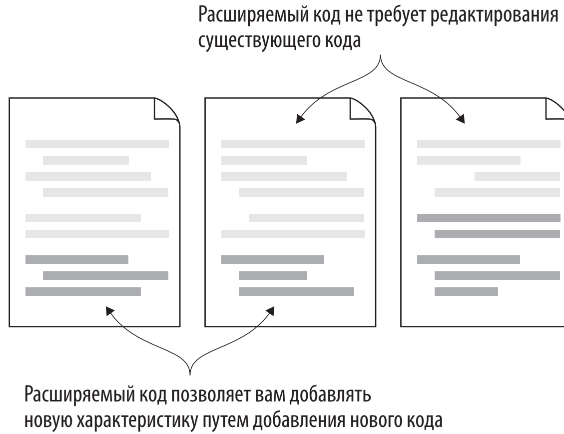


Рис. 7.1. Добавление нового поведения в расширяемый код

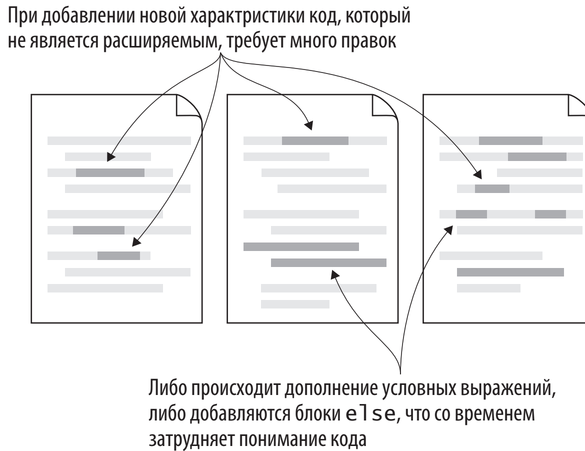


Рис. 7.2. Добавление нового поведения в код, который не является расширяемым

В менее расширяемой системе новая функциональность может потребовать добавления условных блоков в функцию, метод и т. д. (рис. 7.2). Широта изменений и их гранулярность иногда называются *хирургией дробовика (или стрельбой дробью)*[1], потому что новая характеристика вынуждает усеять изменениями весь код, как мелкой дробью[2]. Это вскрывает смешение ответственности и необходимость другого подхода.

В конце предыдущей главы я отметил, что добавить новую характеристику в Bark относительно просто — нужно:

- добавить новую логику постоянства данных в модуль `database`, если это необходимо;
- добавить новую бизнес-логику в модуль `command` для опорной функциональности;
- добавить новый вариант действий в модуль `bark` для обработки взаимодействия с пользователем.

СОВЕТ

Дублировать некий код и обновлять копию для его выполнения — значит делать исходный код более расширяемым. Изменяя копию, я смотрю, как отличаются две версии, чтобы проще рефакторизовать дублированный код обратно в единую многоцелевую версию. Если вы попытаетесь устранить дублирование кода без полного понимания всех путей его использования, то рискуете принять слишком много допущений и сделать код негибким. Поэтому помните, что дублирование лучше, чем неправильная абстракция.

---

[1] https://en.wikipedia.org/wiki/Shotgun_surgery

[2] Подробнее о хирургии дробовика и других видах кода с душком читайте в работе «Расследование плохих запахов в объектно-ориентированном дизайне» (*An Investigation of Bad Smells in Object-Oriented Design*), *Third International Conference on Information Technology: New Generations*, 2006 (https://ieeexplore.ieee.org/document/1611587).

Если приложение Bark почти идеально выполняет эти три действия, то вам нужно только добавить новый код, не затрагивая старый. Поскольку реальные системы редко бывают идеальными, вы все равно обнаружите необходимость в регулярных изменениях существующего кода (рис. 7.3). Как гибкость вам поможет?

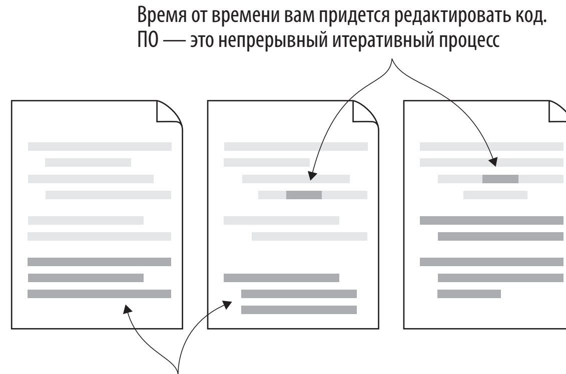


Рис. 7.3. Как расширяемость выглядит на практике

7.1.2. Изменение существующего поведения

Есть целый ряд причин изменить код: устранение ошибки, реакция на новые требования к коду, упрощение работы с кодом и сохранение согласованного поведения. Вы не всегда стремитесь сделать код расширяемым за счет нового поведения, но *гибкость* кода по-прежнему играет большую роль.

Гибкость — это мера сопротивления кода изменениям. Идеальная гибкость означает, что любой фрагмент кода может быть легко заменен на другую реализацию. Код, который во время изменений требует стрельбы дробью, является *жестким*. Кент Бек остроумно заметил: «Каждое желаемое изменение сделайте легким (предупреждение: это может быть трудно), а затем внесите эти легкие изменения»[1]. Разрушение сопротивления кода с самого начала — посредством декомпозиции, инкапсуляции и т. д. — позволит внести только нужное изменение.

[1] Твит от 25 сентября 2012 г., https://twitter.com/kentbeck/status/250733358307500032.

В своей работе я делаю малые, непрерывные рефакторинги в текущем участке кода. Например, код, в котором вы работаете, может содержать замысловатый набор инструкций `if…else` (листинг 7.1). Чтобы найти место для изменений в наборе условных блоков, придется его читать почти полностью. И есть риск, что изменение придется вносить многократно.

Листинг 7.1. Жесткое отображение (mapping) условий с результатами

```
if choice == 'A':
    print('A — для яблок')
elif choice == 'B':
    print('B — для летучих мышей')
...
```

Эта условная инструкция должна быть обновлена правильно для каждой альтернативы

Обязанности отображения варианта сообщения и вывода

Как это можно улучшить?

1. Извлечь информацию из условных проверок и тел в `dict`.
2. Использовать цикл `for` для проверки каждой имеющейся альтернативы.

Поскольку каждый вариант отображается с конкретным результатом, извлечение связки форм поведения в словарь `dict` будет правильным подходом. Связывая букву варианта со словом, содержащимся в сообщении, новая версия кода сможет извлечь нужное слово из связки независимо от выбранной альтернативы. И больше не потребуется постоянно добавлять инструкции `elif` в условный блок и задавать поведение для нового случая. Вместо этого вы сможете добавить одну-единственную новую связку выбранной буквы со словом, которое вы будете использовать в сообщении, выделив печать в конце (листинг 7.2). Увязывание вариантов с сообщениями действует как *конфигурационная* информация, которую программа использует, чтобы выбрать способ выполнения. Конфигурацию часто легче понять, чем логику условных блоков.

Листинг 7.2. Более гибкий способ отображение (mapping) условий с исходами

```
choices = {
    'A': 'яблоки',
    'B': 'летучие мыши',
    ...
}

print(f'{choice} — для {choices[choice]}')
```

Извлечение связки вариантов с сообщениями облегчает добавление нового варианта

Исход централизован, и поведение Print отделено

Эта версия кода удобочитаема. В то время как пример в листинге 7.1 требовал от вас понимания условий, здесь версия четче структурирована как набор вариантов и строка кода, которая выводит информацию о конкретном варианте. Добавлять новые варианты и изменять выводимое сообщение также проще, потому что они разделены. Все это мы делаем в погоне за слабой сопряженностью.

7.1.3. Слабая сопряженность

Расширяемость вырастает в слабосопряженных системах. Без слабой сопряженности большинство изменений в системе будут происходить в виде стрельбы дробью. Предположим, что вы написали приложение Bark без слоев абстракции вокруг БД и бизнес-логики — что-то вроде следующего листинга. Эту версию трудно читать из-за ее физической компоновки (обратите внимание на глубокую вложенность) и сосредоточения разных зон ответственности в одном фрагменте кода.

Листинг 7.3. Процедурный подход к приложению Bark

```
if __name__ == '__main__':
    options = [...]

    while True:
        for option in options:
            print(option)

        choice = input('Выберите вариант: ')

        if choice == 'A':
            ...
            sqlite3.connect(...).execute(...)
        elif choice == 'D':
            ...
            sqlite3.connect(...).execute(...)
```

Глубокая вложенность — это недвусмысленный намек на то, что зоны ответственности нуждаются в дальнейшем разделении

О блоках if, elif и else сложно рассуждать

Поведение БД является повторяющимся и перемешано с пользовательским взаимодействием

Этот код будет работать, но попробуйте реализовать изменение, которое влияет на соединение с БД, или вообще изменить опорную БД. Это будет болезненно. В коде есть много взаимозависимых фрагментов, поэтому добавление нового поведения потребует введения еще одного блока `elif`, некоторого сырого SQL и т. д. Поскольку вы будете нести эти расходы

всякий раз, когда захотите добавить новое поведение, эта система не будет хорошо масштабироваться.

Представьте атомы в железной болванке — они плотно упакованы, крепко держатся друг за друга. Это делает железо *жестким*, и оно сопротивляется изгибу или изменению формы. Но кузнецы придумали, как преодолевать эту жесткость путем плавления железа, в результате которого связь атомов ослабляется и они могут свободно обтекать друг друга. Даже когда железо остывает, оно становится *ковким*, или подвижным, и способно сгибаться не ломаясь.

Того же нужно добиться от кода (рис. 7.4). Слабосопряженные фрагменты могут свободнее перемещаться с места на место, чего нельзя сказать о плотно упакованном и опирающемся на окружение коде.

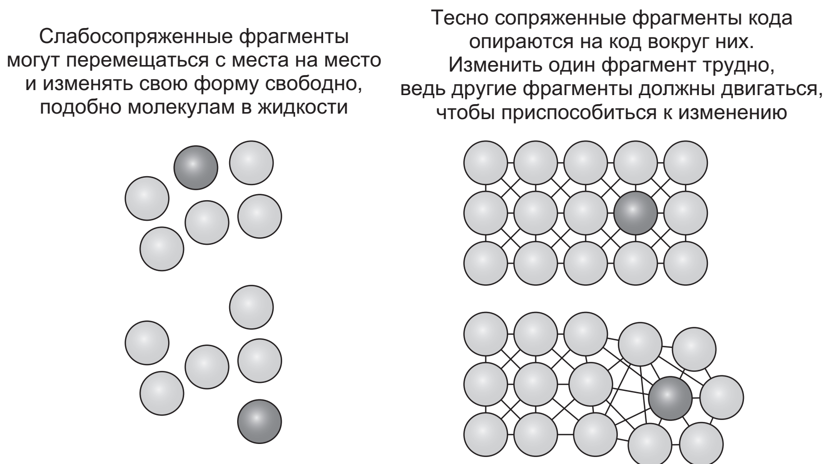


Рис. 7.4. Гибкость в контрасте с жесткостью

Слабая сопряженность, которую вы использовали при написании приложения Bark, означает, что новая функциональность БД может быть добавлена с помощью новых методов в классе `DatabaseManager` или целенаправленными изменениями в существующем (централизованном) методе. Новая бизнес-логика может быть инкапсулирована в новые классы `Command`, а добавление в меню останется вопросом создания нового варианта действий в словаре `options` в модуле `bark` и подключения

его к команде. Это похоже на браузерные плагины. Приложение Bark не нацелено обрабатывать новые характеристики, но они могут быть добавлены вполне ограниченным числом усилий.

Этот краткий обзор слабой сопряженности показывает, каким образом то, что вы узнали до сих пор, поможет вам разрабатывать гибкий код. А теперь я научу вас нескольким новым техническим приемам для достижения еще большей гибкости.

7.2. РЕШЕНИЯ ПРОБЛЕМЫ ЖЕСТКОСТИ

Жесткость в коде развивается, как артрит у людей, — с возрастом. По мере устаревания ПО код, который используется меньше всего, тяготеет к наибольшей жесткости, и требуется индивидуальный подход, чтобы снова ее ослабить.

Из следующих разделов вы узнаете несколько способов снижения жесткости.

7.2.1. Отпустить на свободу: инверсия управления

Мы уже говорили, что по сравнению с наследованием композиция обеспечивает объектам выгоду многократного использования поведения, не ограничивая их иерархией. Когда вы разделяете ответственность на большое число малых классов и хотите составить композицию из этих форм поведения, вы можете написать класс, который использует экземпляры меньших классов. Эта практика часто встречается в объектно-ориентированных кодовых базах.

Представьте, что вы работаете в модуле, который занимается велосипедами и их деталями. Вы открываете модуль `bicycle` и видите код из следующего листинга. Попробуйте оценить в нем инкапсуляцию и абстракцию.

Листинг 7.4. Композитный класс, зависящий от других, более мелких классов

```
class Tire:  ◄── Малые классы для использования в композиции
    def __repr__(self):
```

```
        return 'Резиновая шина'

class Frame:
    def __repr__(self):
        return 'Алюминиевая рама'

class Bicycle:
    def __init__(self):    ◄—— Bicycle создает детали, которые ему нужны
        self.front_tire = Tire()
        self.back_tire = Tire()
        self.frame = Frame()

    def print_specs(self):    ◄—— Вывод всех деталей велосипеда
        print(f'Рама: {self.frame}')
        print(f'Передняя шина: {self.front_tire}, задняя шина:
                               {self.back_tire}')

if __name__ == '__main__':    ◄—— Создает велосипед и выводит его технические характеристики
    bike = Bicycle()
    bike.print_specs()
```

Выполнение этого кода позволит вывести технические характеристики велосипеда:

```
Рама: алюминиевая рама
Передняя шина: резиновая шина, задняя шина: резиновая шина
```

В результате вы, безусловно, получите велосипед. Инкапсуляция выглядит хорошо: каждая деталь велосипеда располагается в своем отдельном классе. Уровни абстракции тоже имеют смысл: на верхнем уровне находится `Bicycle`, и каждая его деталь доступна уровнем ниже. Что может вызвать трудность?

1. Добавление новых деталей в велосипед.
2. Обновление деталей велосипеда.

Добавлять новые детали в велосипед не очень сложно. Можно создать экземпляр новой детали и сохранить его на экземпляре `Bicycle` в методе `__init__` вместе с другими деталями. А вот обновлять (изменять) детали экземпляра `Bicycle` динамически в этой структуре трудно, поскольку классы для деталей жестко закодированы и инициализированы.

Вы могли бы сказать, что `Bicycle` *зависит* от `Tire`, `Frame` и других необходимых ему деталей. Без них велосипед не может функционировать.

Но если вам нужна карбоновая рама `CarbonFiberFrame`, придется взломать код класса `Bicycle`, чтобы его обновить. По этой причине класс `Tire` в настоящее время является жесткой зависимостью класса `Bicycle`.

Инверсия управления подразумевает, что вместо создания экземпляров зависимостей в классе вы можете передать существующие экземпляры для использования этим классом (рис. 7.5). *Контроль* над созданием зависимостей *инвертируется* путем предоставления контроля любому коду, создающему `Bicycle`. Это очень мощный прием.

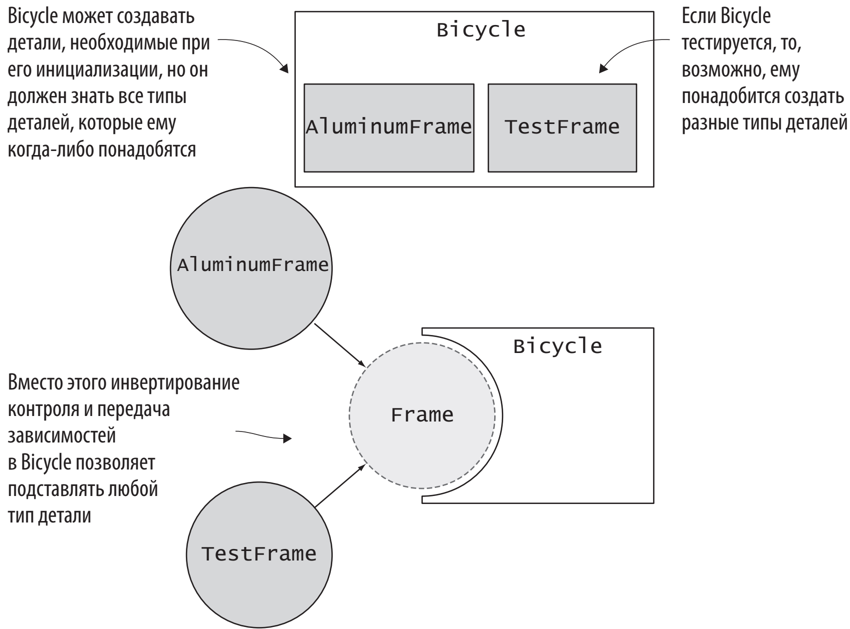


Рис. 7.5. Использование инверсии управления для достижения гибкости

Попробуйте обновить метод `Bicycle.__init__`, чтобы он принимал аргумент для каждой своей зависимости и передавал их в метод, а затем вернитесь к следующему листингу.

Листинг 7.5. Использование инверсии управления

```
class Tire:
    def __repr__(self):
        return 'Резиновая шина'
```

```
class Frame:
    def __repr__(self):
        return 'Алюминиевая рама'


class Bicycle:
    def __init__(self, front_tire, back_tire, frame):
        self.front_tire = front_tire
        self.back_tire = back_tire
        self.frame = frame

    def print_specs(self):
        print(f'Рама: {self.frame}')
        print(f'Передняя шина: {self.front_tire}, задняя шина:
                              {self.back_tire}')


if __name__ == '__main__':
    bike = Bicycle(Tire(), Tire(), Frame())
    bike.print_specs()
```

Зависимости передаются в класс после инициализации

Код, который создает Bicycle, снабжает его надлежащими экземплярами

Этот код должен дать вам тот же результат, что и раньше. Может показаться, что мы переложили проблему, но это обеспечило определенную степень свободы. Теперь вы можете создавать любую причудливую шину или раму, которую пожелаете, и использовать ее вместо базовых версий. До тех пор пока класс `FancyTire` имеет те же методы и атрибуты, что и любая другая шина, велосипеду будет все равно.

Попробуйте создать новый класс карбоновых шин `CarbonFiberFrame` и обновить (изменить) свой велосипед. Затем вернитесь к следующему листингу.

Листинг 7.6. Использование нового вида рамы для велосипеда

```
class CarbonFiberFrame:
    def __repr__(self):
        return 'Карбоновая рама'

...

if __name__ == '__main__':
    bike = Bicycle(Tire(), Tire(), CarbonFiberFrame())
    bike.print_specs()
```

Карбоновая рама может использоваться так же просто, как и обычная

Теперь вы должны увидеть карбоновую раму в выведенных технических характеристиках

Такая способность замены зависимостей с минимальными усилиями полезна при тестировании кода. Чтобы точно изолировать поведение в классах, вы от случая к случаю захотите заменить реальную реализацию зависимости тестовым дублером. Наличие жесткой зависимости от `Tire` заставляет вас выполнять мок класса `Tire` для каждого теста класса `Bicycle`, чтобы достичь изоляции. Инверсия управления освобождает вас от этого ограничения, позволяя, к примеру, передавать внутрь экземпляр класса `MockTire`. Благодаря этому вы не забудете что-то сымитировать, потому что вы должны передать *какую-то* шину создаваемым вами экземплярам `Bicycle`.

Упрощение тестирования является одной из главных причин, почему надо следовать принципам, изложенным в этой книге. Если код трудно тестировать, то, возможно, его также трудно понять, и наоборот.

7.2.2. Дьявол в мелочах: опора на интерфейсы

Жесткость проявляется, когда высокоуровневый код слишком сильно опирается на детали более низкоуровневых зависимостей. Я предположил, что причудливая шина `FancyTire` может быть поставлена, *если* имеет те же методы и атрибуты, что и другие шины. Выражаясь формальнее, можно поместить в класс `Bicycle` любой объект, если он имеет *интерфейс* шины.

`Bicycle` не так много знает о деталях *конкретной* шины (или интересуется ими). Ему важно только наличие у шины конкретного набора информации и поведения, а в остальном шины вольны делать все, что угодно.

Совместное использование согласованных интерфейсов (вместо ожидания соответствия специфичных для класса деталей шины) между высокоуровневым и низкоуровневым кодом дает свободу для обмена имплементациями. Помните, что утиная типизация Python против строгих интерфейсов. Какие методы и атрибуты составят интерфейс, решаете вы. Разработчик должен сделать так, чтобы классы придерживались интерфейсов, которые ожидают потребители.

В Bark классы `Command` в бизнес-логике предоставляют метод `execute` как часть своего интерфейса. Слой визуализации использует этот интерфейс, когда пользователь выбирает тот или иной вариант действий. Реализация

конкретной команды может измениться настолько, насколько это необходимо, и никаких изменений в слое визуализации не потребуется до тех пор, пока интерфейс остается прежним. Вам нужно будет изменить слой визуализации, только если, например, методы `execute` классов `Command` потребуют дополнительного аргумента.

Для тесно связанных фрагментов кода вставка интерфейса будет ощущаться излишней мерой. А для разделенных фрагментов из разных классов или модулей использование совместных интерфейсов облегчит обмен данными в программе.

7.2.3. Борьба с энтропией: принцип надежности

Энтропия — это тенденция к постепенному растворению организации в дезорганизации. Вначале код всегда небольшой, аккуратный и понятный, но со временем он тяготеет в сторону сложности. Одна из причин этого — необходимость приспособить код к разным видам входных данных.

Принцип надежности, также именуемый законом Постела, гласит: «Консервативно относитесь к своей деятельности и либерально — ко вкладам других». Согласно этому утверждению вы должны обеспечить поведение, необходимое для достижения желаемых выходных данных, но открытое для несовершенных или неожиданных входных данных. Не нужно принимать любые входные данные, какие только существуют в природе, но наличие гибкости облегчит потребителям кода жизнь. Увязывая крупный диапазон входных данных с меньшим диапазоном выходных данных, вы можете направить и ограничить поток информации (рис. 7.6).

Рассмотрим встроенную функцию `int()`, которая конвертирует входные данные в целое число. Она работает для входных данных, которые являются целыми числами:

```
>>> int(3)
3
```

строковыми значениями:

```
>>> int('3')
3
```

и числами с плавающей точкой, возвращая только целую часть числа:

```
>>> int(6.5)
6
```

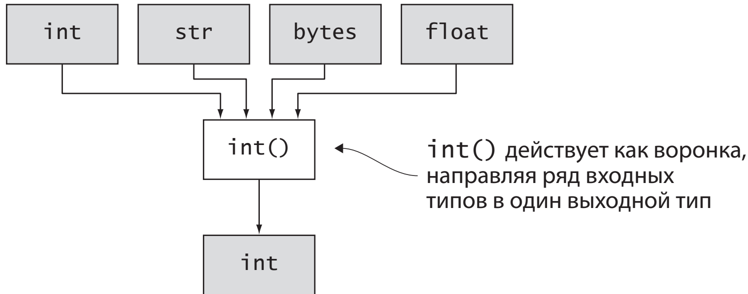


Рис. 7.6. Уменьшение энтропии при отображении входов в выходы

Функция `int` принимает несколько типов данных и направляет их к возвращаемому целочисленному типу, возбуждая исключение только в том случае, если действительно неясно, как действовать дальше[1]:

```
>>> int('датчанин')
ValueError: invalid literal for int() with base 10: 'датчанин'
```

Потратьте некоторое время на понимание диапазона ожидаемых входных данных, а затем обуздайте их, чтобы они возвращали только то, что ожидает остальная часть системы. Это обеспечит гибкость потребителям, которые находятся в точках входа в систему, сохраняя управляемым число ситуаций, с которыми опорный код должен справляться.

7.3. УПРАЖНЕНИЕ НА РАСТЯЖКУ

Понимая, что входит в расширяемый и гибкий дизайн, вы можете аккуратно добавить функциональность в Bark. Сейчас Bark представляет собой довольно простой инструмент — вы можете поочередно добавлять закладки, а пользователи должны сами вводить все URL-адреса

[1] `ОшибкаЗначения: недопустимый литерал для int() с основанием 10: 'датчанин'`

и описания. Эта работа утомительна, особенно если у них уже есть куча закладок, которые хранятся в другом инструменте.

Предлагаю разработать импортер звезд GitHub для Bark (рис. 7.7). Этот новый вариант действий по импортированию в слое визуализации должен выполнять следующее:

1) запрашивать у пользователя Bark его пользовательское имя GitHub;
2) узнавать, нужно ли сохранять временные метки исходных звезд;
3) запускать соответствующую команду.

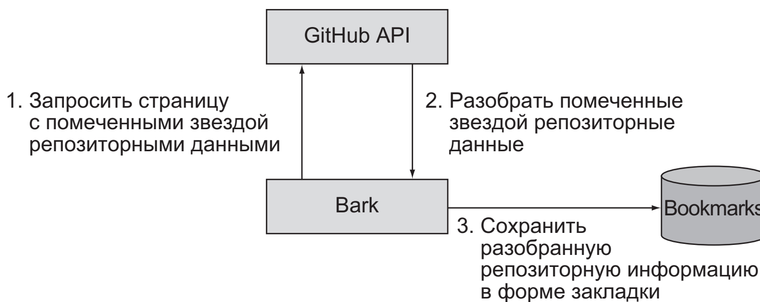


Рис. 7.7. Рабочий поток импортера звезд GitHub для Bark

Запускаемая команда должна использовать API хостинга GitHub для получения данных о звездах. Рекомендую установить и использовать пакет `requests` (https://github.com/psf/requests).

Данные о звездах разделены на страницы, поэтому процесс будет выглядеть примерно так:

1) получить начальную страницу результатов со звездами (конечная точка: https://api.github.com/users/{github_username}/starred);
2) разобрать данные из отклика, чтобы выполнить команду `AddBookmarkCommand` для каждого помеченного звездой репозитория;
3) получить заголовок `Link: <…>; rel=next`, если он есть;
4) повторить для следующей страницы. Если таковой нет — остановиться.

> **ПРИМЕЧАНИЕ**
>
> Чтобы получать временные метки звезд GitHub, нужно передавать заголовок `Accept: application/vnd.github.v3.star+json` в запросах к API.

С точки зрения пользователя, взаимодействие должно выглядеть примерно так:

```
$ ./bark.py
(A) Добавить закладку
(B) Показать список закладок по дате
(T) Показать список закладок по заголовку
(D) Удалить закладку
(G) Импортировать звезды GitHub
(Q) Выйти

Выберите вариант действий: G
Пользовательское имя GitHub: daneah
Сохранить метки времени [Д/н]: Д
Импортировано 205 закладок из помеченных звездами репо!
```

Оказывается, Bark не *полностью* расширяем, особенно в отношении временных меток закладок. Сейчас Bark делает временную метку создания закладки (с помощью `datetime.datetime.utcnow().isoformat()`). Но чтобы оставлять временные метки звезд GitHub нетронутыми, используйте инверсию контроля.

Обновите команду `AddBookmarkCommand` так, чтобы она принимала необязательную временную метку, используя ее исходное поведение в качестве запасного варианта. Сверьте свой результат со следующим листингом.

Листинг 7.7. Инверсия управления для временной метки закладки

```
class AddBookmarkCommand:

    def execute(self, data, timestamp=None):
        data['date_added'] = timestamp or datetime.utcnow().isoformat()
        db.add('bookmarks', data)
        return 'Bookmark added!'
```

Добавляет необязательный аргумент timestamp для выполнения

Задействует переданный аргумент timestamp, если он предоставлен, используя текущее время в качестве запасного варианта

Теперь вы улучшили гибкость команды `AddBookmarkCommand`, и она достаточно расширена, чтобы обрабатывать то, что нужно для импортера звезд GitHub. Вам не потребуется никакая новая функциональность в слое постоянства данных, и поэтому вы можете сосредоточиться на визуализации и бизнес-логике для новой характеристики. Сделайте свой код и сравните его со следующими двумя листингами.

Листинг 7.8. Команда импортирования звезд GitHub

```
class ImportGitHubStarsCommand:
    def _extract_bookmark_info(self, repo):    ◄── Имея словарь repository, извлечь
        return {                                    необходимые фрагменты для
            'title': repo['name'],                  создания закладки
            'url': repo['html_url'],
            'notes': repo['description'],
        }

    def execute(self, data):
        bookmarks_imported = 0

        github_username = data['github_username']     URL-адрес первой страницы
        next_page_of_results =                         результатов со звездами
➥ f'https://api.github.com/users/{github_username}/starred'   ◄──

        while next_page_of_results:    ◄── Продолжает получать результаты со звездами,
                                           пока существуют страницы результатов
            stars_response = requests.get(    ◄── Получает следующую страницу результатов,
                next_page_of_results,              используя правильный заголовок, чтобы
                headers={'Accept': 'application/   сообщить API о необходимости
                       vnd.github.v3.star+json'},  возврата временных меток
            )
            next_page_of_results =
➥ stars_response.links.get('next', {}).get('url')   ◄── Заголовок Link с rel=next
                                                         содержит ссылку на следу-
            for repo_info in stars_response.json():      ющую страницу, если она
                repo = repo_info['repo']    ◄──          имеется
                                 Информация о помеченном звездами
                                 репозитории

                if data['preserve_timestamps']:
                    timestamp = datetime.strptime(       Метка времени
                        repo_info['starred_at'],    ◄── создания звезды
                        '%Y-%m-%dT%H:%M:%SZ'    ◄── Форматирует метку времени
                    )                               в том же формате, что и суще-
                else:                               ствующие закладки Bark
                    timestamp = None

                bookmarks_imported += 1
```

```
            AddBookmarkCommand().execute(
                self._extract_bookmark_info(repo),
                timestamp=timestamp,
            )

        return f'Импортировано {bookmarks_imported} закладок
               из помеченных звездами репо!'
```

Выполняет команду AddBookmarkCommand, заполняя Bark репозиторными данными

Возвращает сообщение с указанием количества импортированных звезд

Листинг 7.9. Вариант с импортированием звезд GitHub в меню

```
...

def get_github_import_options():
    return {
        'github_username': get_user_input('Пользовательское
                                           имя GitHub'),
        'preserve_timestamps':
            get_user_input(
                'Сохранить метки времени [Д/н]',
                required=False
            ) in {'Д', 'д', None},
    }


def loop():
    ...


    options = OrderedDict({
        ...
        'G': Option(
            'Импортировать звезды GitHub',
            commands.ImportGitHubStarsCommand(),
            prep_call=get_github_import_options
        ),
    })
```

Функция получения пользовательского имени GitHub источника импорта

Нужно ли сохранить время изначального создания звезды?

Принимает «Д», «д» либо просто нажатие Enter, когда пользователь говорит «да»

Добавляет в меню вариант с импортированием из GitHub с правильным командным классом и функцией

Должно быть, вы заметили, что добавление поведения в расширяемую систему сопровождается низким трением. Как же приятно почти полностью сосредоточиться на выполнении желаемого поведения, составляя композиции из частей существующей инфраструктуры и подключая прочие «фишки». Редкий момент, когда вы, как разработчик, можете почувствовать себя дирижером оркестра, который добавляет смычковые, духовые и ударные, чтобы достичь гармонии. Если вдруг вы услышите

какофонию, не отчаивайтесь. Найдите точки жесткости, вызывающие диссонанс, и вспомните, как можно освободиться из оков.

БОЛЬШЕ ПРАКТИКИ

Если хотите получить еще немного опыта в расширении Bark, попробуйте выполнить правку существующей закладки.

Добавьте в `DatabaseManager` новый метод для обновления записей, укажите изменяемую запись, а также имя столбца и новое значение. В качестве руководства используйте все, что знаете о методах `add`, `select` и `delete`.

Слой визуализации должен запрашивать у пользователя ID обновляемой закладки, обновляемый столбец и новое значение. Это позволит подключиться к новой команде `EditBookmarkCommand` в слое бизнес-логики.

Во всем этом вы теперь профи, так что попробуйте! Моя версия находится в исходном коде этой главы (https://github.com/daneah/practices-of-the-python-pro).

Из следующей главы вы больше узнаете о наследовании.

ИТОГИ

- Создавайте код так, чтобы добавление новых характеристик означало добавление новых функций, методов или классов без изменения существующих.
- Инверсия управления позволяет другому коду адаптировать поведение к потребностям без изменения низкоуровневой имплементации.
- Совместное использование согласованных интерфейсов между классами вместо предоставления им подробных знаний друг о друге уменьшает сопряженность.
- Относитесь взвешенно к тому, какие типы входных данных вы хотите обрабатывать, и будьте строги в отношении выходных типов.

8 Правила (и исключения) наследования

В этой главе:

- ✓ Использование наследования и композиции вместе для моделирования систем.
- ✓ Использование встроенных модулей Python для проверки типов объектов.
- ✓ Придание интерфейсам большей строгости с помощью абстрактных базовых классов.

Если вы сами писали классы или использовали фреймворк на основе классов в Python, то скорее всего столкнулись с *наследованием*. Классы могут наследовать от других классов, перенимая поведение родительского класса. Из этой главы вы больше узнаете о наследовании в Python и о том, когда оно уместно или неуместно.

8.1. НАСЛЕДОВАНИЕ РАНЬШЕ

Наследование было задумано в первые дни компьютерного программирования, но люди по-прежнему ведут оживленные споры о том, когда и как его использовать. Большую часть истории объектно-ориентированного программирования наследование было его сутью. Приложения представляли собой модели реального мира с тщательно подобранной иерархией объектов. Так объектно-ориентированное программирование и наследование стали практически неразделимыми.

8.1.1. Серебряная пуля

Хотя наследование иногда становится наилучшим инструментом для достижения цели, оно использовалось на практике как молоток для каждого гвоздя, как неуловимая серебряная пуля. Но как и серебряная пуля, парадигма, отвечающая сразу всем требованиям, — это художественный вымысел.

Разочарование от наследования привело многих разработчиков к отказу от объектно-ориентированного программирования. Это печально, ведь объектная ориентация имеет ряд преимуществ для мысленного моделирования задач, и наследование имеет свое место во время создания правильных иерархий. Мы применим наследование к конкретному набору вариантов использования, о которых вы узнаете из этой главы.

Но сначала несколько слов о том, почему наследование классов привело к такому большому разочарованию.

8.1.2. Трудности иерархий

Объектно-ориентированное программирование всецело касается разделения, инкапсуляции и классификации информации и форм поведения. Я работаю со многими программистами, отвечающими за библиотеки, которые забыли о классификации больше, чем я когда-либо о ней узнаю, — эти люди работают, чтобы выявлять связи между явлениями,

создавая таксономии или даже онтологии[1]. Эти подходы хорошо работают для организации необработанной информации, но вызывают головную боль, как только дело касается поведения ПО. По мере роста ПО становится все труднее поддерживать четкое понимание отношений «родитель — ребенок» между классами.

ПРИМЕЧАНИЕ

Родительские классы в Python (и во многих других языках) называются *суперклассами*. *Дочерние* классы называются *подклассами*. Я буду использовать эту терминологию на протяжении всей остальной части главы.

Класс наследует всю информацию и поведение от своего суперкласса, а затем может переопределить их, чтобы выполнить что-то новое (рис. 8.1). Это, пожалуй, самая тесная сопряженность, которая существует в программировании. Класс является полностью сопряженным со своим суперклассом, потому что все, что он знает и делает по умолчанию, привязано к суперклассу.

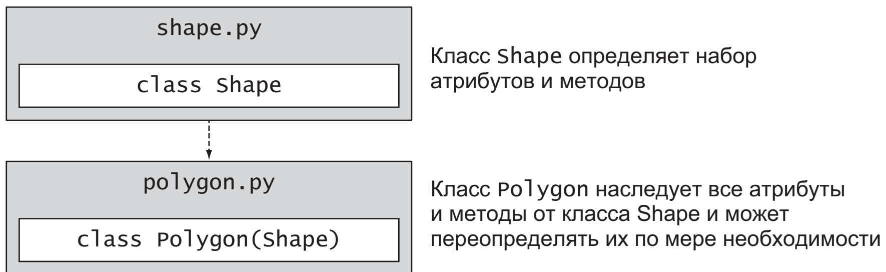


Рис. 8.1. Наследование с одним суперклассом и одним подклассом

Увидеть эту сопряженность *очень трудно*, когда иерархии классов растут, потому что, глядя на конкретный класс, вы не увидите, наследует ли

[1] Подробнее об онтологии в контексте информатики читайте в статье https://ru.wikipedia.org/wiki/Онтология_(информатика).

другой класс от него или нет. Это приводит к ошибкам из-за неожиданных изменений в поведении (рис. 8.2).

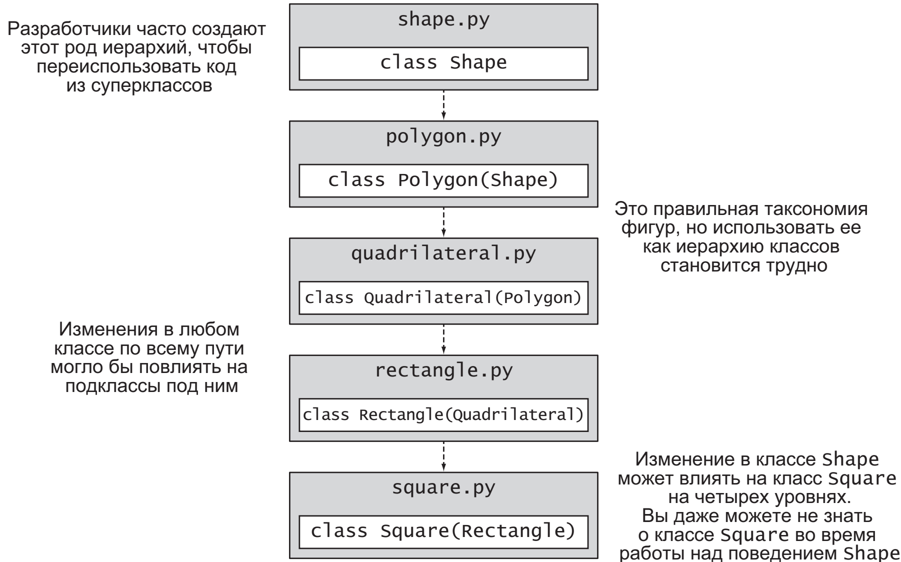


Рис. 8.2. Как глубокая иерархия наследования приводит к увеличению числа ошибок

Если провести аналогию, то в квантовой физике две частицы могут быть *запутаны* таким образом, что изменение одной из них приведет к такому же изменению в другой, независимо от того, как далеко они друг от друга находятся в пространстве. Это «жуткое действие на расстоянии», как называл его Эйнштейн, означает, что вы не можете достоверно определить состояние частицы, потому что оно может измениться в любой момент из-за ее близнеца. В ПО это явление представляет серьезную опасность. Изменяя один класс, вы можете непреднамеренно изменить — или, что еще хуже, — нарушить функциональность в другом подклассе, о котором вы не знали. Как в фильме «Эффект бабочки». (Внимание, спойлер: эта проблема мешает персонажу Эштона Кутчера.)

Иногда наследование применяется для переиспользования кода, но оно создает проблемы в дальнейшем. При наличии глубокой иерархии классы на разных уровнях могут переопределять или дополнять поведение своих

суперклассов. Очень скоро вы обнаружите, что ходите вверх и вниз по классам, пытаясь проследить поток информации. Я уже говорил, что труд разработчика направлен на улучшение понимания кода и уменьшение его когнитивной нагрузки, но глубокие иерархии работают против этой цели. Тогда почему мы все еще используем наследование?

8.2. НАСЛЕДОВАНИЕ СЕЙЧАС

Из-за головной боли, вызываемой сложными иерархиями, наследование приобрело плохую репутацию. Но это зло не было врожденным. Просто наследование использовалось слишком часто и не по назначению.

8.2.1. Зачем нужно наследование?

Многие по-прежнему через наследование хотят переиспользовать код в каком-то классе, но в реальности оно предназначено для *специализации поведения*. Другими словами, боритесь с желанием создать подкласс только ради переиспользования кода. Создавайте подклассы для того, чтобы метод возвращал другое значение либо под капотом работал по-другому.

Рассматривайте подклассы как *специальные случаи* суперкласса. Они будут многократно использовать код из суперкласса, но не повторять его действия.

Когда класс *B* наследует от класса *A*, мы часто говорим *B* «является» *A*. Это подчеркивает, что экземпляры *B* на самом деле являются экземплярами *A* и выглядят так же (подробнее об этом чуть позже). Сравните это с композицией, где, если экземпляр класса *C* использует экземпляр класса *D*, мы говорим, что *C* «имеет» *D*, чтобы подчеркнуть, что *C* состоит из *D* (среди прочего, в потенциальном плане).

Вспомните пример с `Bicycle` из предыдущей главы. Вы ввели несколько типов велосипедных рам, обновив алюминиевую раму `AluminumFrame` до карбоновой рамы `CarbonFiberFrame`, а шину `Tire` — до причудливой шины `FancyTire`. Предположим, что `CarbonFiberFrame` и `FancyTire` наследуют соответственно от `Frame` и `Tire`. Что из нижеследующего можно сказать о том, как вы моделировали велосипеды, используя наследование и композицию?

1. `Tire` имеет `Bicycle`.
2. `Bicycle` имеет `Tire`.
3. `CarbonFiberFrame` является `Tire`.
4. `CarbonFiberFrame` имеет `Frame`.

Поскольку шина не состоит из велосипеда (все наоборот), вариант 1 неверен, тогда как вариант 2 имеет смысл — это композиция. А карбоновая рама *является* рамой (у нее *нет* рамы), и вариант 4 тоже неверен, тогда как вариант 3 имеет смысл — это наследование. Опять же, наследование предназначено для специализации, тогда как композиция — для многократного использования одной формы поведения (рис. 8.3).

Использование наследования для специализации поведения — это только первый шаг. Представьте, что вы заменяете алюминиевую раму на карбоновую. Каждая рама имеет одинаковые точки стыковки, без которых велосипед развалится. Так же и в ПО.

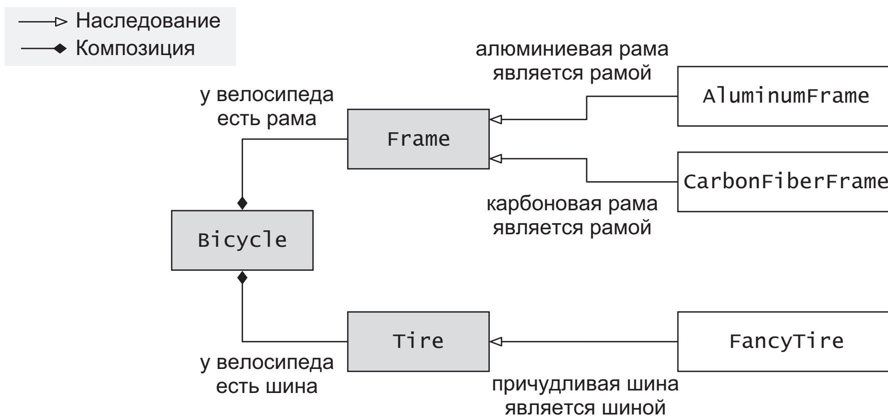


Рис. 8.3. Как наследование и композиция работают вместе

8.2.2. Подстановка

Барбара Лисков, профессор Массачусетского технологического института (MIT), разработала принцип, описывающий концепцию *подстановки* (замещения) в наследовании. Этот принцип гласит, что в программе любой

экземпляр класса должен быть пригодным для замещения экземпляром одного из его подклассов без ущерба для правильности программы.[1] *Правильность* в этом контексте означает, что программа остается безошибочной и достигает тех же базовых результатов, хотя точный результат может быть другим или достигнут другим способом. Подстановка возникает из-за того, что подклассы строго придерживаются интерфейса своих суперклассов.

В Python нетрудно отойти от этого принципа. Рассмотрим следующий листинг, где совершенно правильный код Python моделирует слизней и улиток (две разновидности брюхоногих моллюсков). Класс улиток `Snail` наследует от класса слизняков `Slug` (улитки и слизняки одинаковы во всем, кроме раковины), и можно сказать, что класс `Snail` специализирует, или детализирует, класс `Slug`, добавляя информацию о раковине. Но класс `Snail` нарушает подстановку, потому что программа, использующая класс `Slug`, **не может заменить его классом** `Snail`, не добавив аргумент `shell_size` в метод `__init__`.

Листинг 8.1. Подкласс, нарушающий принцип подстановки

```
class Slug:
    def __init__(self, name):
        self.name = name

    def crawl(self):
        print('slime trail!')


class Snail(Slug):     ◄—— Класс Snail наследует от класса Slug
    def __init__(self, name, shell_size):  ◄—— Использование сигнатуры создания
    super().__init__(name)                     отличающегося экземпляра часто
    self.name = name                           ведет к нарушению подстановки
    self.shell_size = shell_size


def race(gastropod_one, gastropod_two):    Вы можете создать два экземпляра клас-
    gastropod_one.crawl()                  са Slug и организовать гонки
    gastropod_two.crawl()
                                           Попытка использовать класс Snail
                                           без аргумента shell_size возбуждает
race(Slug('Джеффри'), Slug('Рамона'))  ◄   исключение
race(Snail('Джеффри'), Snail('Рамона'))  ◄
```

[1] https: //ru.wikipedia.org/wiki/Принцип_подстановки_Барбары_Лисков.

Можно вытащить из рукава еще больше трюков, чтобы заставить все работать, но учтите, что все это больше подходит для композиции. В конце концов, у улитки *есть* раковина.

Мне нравится сомневаться в *роли*, которую выполняет конкретный набор классов. Если каждый класс в иерархии может выполнять одну и ту же роль, то они пригодны для замещения. Если подкласс изменяет любую свою сигнатуру метода или вызывает исключение в рамках своей специализации, то он может и не выполнять эту роль, но иерархию классов тогда придется переорганизовать.

8.2.3. Идеальный для наследования вариант использования

Сэнди Метц, программист Ruby, который первоначально пришел из сообщества Smalltalk (Smalltalk — язык программирования, частично созданный Аланом Кеем, одним из пионеров объектно-ориентированного программирования), изложил большой набор базовых правил о том, когда следует использовать наследование:[1]

- задача, которую вы решаете, имеет мелкую узкую иерархию;
- подклассы находятся в листьях графа объектов и пользуются другими объектами;
- подклассы используют (специализируют, детализируют) *все* поведение суперкласса.

Я расскажу о каждом из них подробнее.

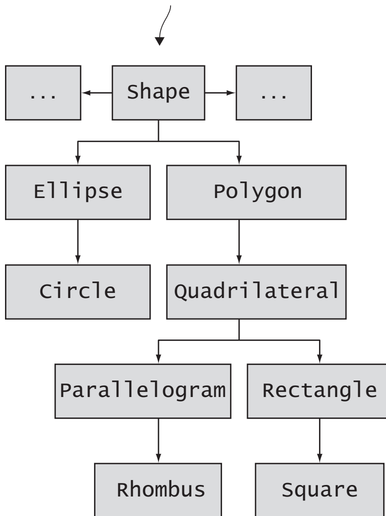
Мелкая узкая иерархия

Мелкая часть этого правила гласит, что глубоко вложенные иерархии классов приводят к трудностям в управлении ими и вводят дефекты. Поэтому, поддерживая иерархию малой и ограниченной, вам становится понятно, зачем это нужно (рис. 8.4).

---

[1] Подробнее об этом в выступлении Сэнди Метца *All the Little Things* («Все эти мелочи») на RailsConf, 2014 г., www.youtube.com/watch?v=8bZh5LMaSmE.

Узкая часть этого правила означает, что ни один класс в иерархии не должен иметь слишком много подклассов. По мере роста числа подклассов становится трудно понимать, какие из них обеспечивают ту или иную специализацию, и другие разработчики могут дублировать подклассы, если не найдут что искали.



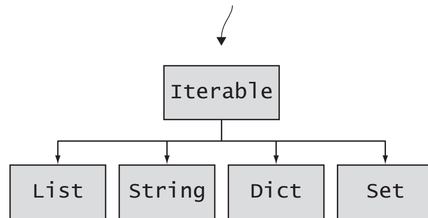


Рис. 8.4. Эффективнее размышлять в терминах узких мелких иерархий наследования

Подклассы в листьях графа объектов

Вы можете думать об объектах ПО как о вершинах в графе, указывающих на другие объекты, от которых они наследуют или которые используют посредством композиции. В наследовании класс может указывать на другие объекты, но его *подклассы* обычно не должны иметь дополнительных зависимостей. Подклассы предназначены для *специализированного* поведения, но если подкласс имеет уникальную зависимость, которой нет у суперкласса или другого подкласса, используйте композицию.

Проверяйте, что подклассы специализируют поведение без добавления большого объема новой сопряженности.

Подклассы используют все поведение суперкласса

Это результат отношения «является». Если подкласс *не* использует все поведение суперкласса, разве он является экземпляром суперкласса? Рассмотрим класс, который представляет птицу:

```
class Bird:
    def fly(self):
        print('летает!')
```

Вы можете подклассировать его так, чтобы метод fly делал что-то другое для некоторых видов птиц, например колибри:

```
class Hummingbird(Bird):
    def fly(self):
        print('вжжжжух!')
```

Что происходит, когда вы добираетесь до пингвина, киви или страуса? Ни одна из этих птиц не летает. Одним из возможных решений является переопределение fly вот таким образом:

```
class Penguin(Bird):
    def fly(self):
        print('никто не может.')
```

Также можно переопределить метод fly, чтобы ничего не делать (`pass`) или вызывать какое-то исключение. Однако это будет противоречить принципу подстановки. Любой код, который знает, что он имеет дело с классом `Penguin`, вряд ли вызовет метод fly, и поэтому указанное поведение не будет использоваться. Опять же, композиция «полетного» поведения в классы, которые в нем нуждаются, здесь была бы более подходящим вариантом.

Упражнение

Теперь попробуйте применить правила наследования и композиции к примеру с классом `Bicycle`. Модуль `bicycle` можно найти в исходном коде этой главы (https://github.com/daneah/practices-of-the-python-pro).

Посмотрим, сможете ли вы выяснить, следуют ли объекты в модуле `bicycle` каждому из правил Метца или нет.

Вернитесь сюда, чтобы оценить, что у вас получилось:

- Классы `Frame` и `Tire` имеют узкую мелкую иерархию, и у каждого из них есть ниже один уровень с двумя подклассами максимум.
- Разные типы шин и рам не зависят от каких-либо других объектов.
- Разные типы шин и рам используют или специализируют все поведение суперклассов.

Ура! Созданная модель, как положено, использует наследование там, где это необходимо, и композицию для сведения разных частей в единое целое. Читайте дальше, чтобы узнать, какие инструменты Python предоставляет для проверки и использования наследования.

8.3. НАСЛЕДОВАНИЕ В PYTHON

Python предоставляет набор инструментов для проверки классов и их структуры наследования, а также ряд путей к наследованию и композиции. Этот раздел посвящен каждому из них и содержит ноу-хау для отладки и тестирования кода.

8.3.1. Проверка типов

При отладке кода вас в первую очередь интересует тип объекта, с которым вы имеете дело в конкретной строке кода. Динамическая типизация Python означает, что тип не всегда сразу очевиден, и поэтому неплохо провести осмотр.

Элементарным способом проверки типа объекта является использование встроенной функции `type()`. Вызов `type(some_object)` сообщит, экземпляром какого класса этот объект является:

```
>>> type(42)
<class 'int'>
>>> type({'десерт': 'печенье', 'вкус': 'шоколадная крошка'})
<class 'dict'>
```

ПРОВЕРКА ТИПОВ

Последние версии Python поддерживают механизм *подсказок типов*, то есть способ указывать разработчикам и автоматизированным инструментам на то, какие типы объектов функция или метод ожидают получить. Инструменты могут проверять наличие вызовов, нарушающих эти типы, не выполняя код. Обратите внимание, что Python не обеспечивает соблюдение типов во время выполнения, а лишь предоставляет вспомогательные средства для разработки.

Чтобы узнать, является ли объект экземпляром конкретного класса или любого из его подклассов, используйте функцию `isinstance()`:

```
>>> isinstance(42, int)
True
>>> isinstance(FancyTire(), Tire)
True
```

Любые классы, на которые вы ссылаетесь, необходимо импортировать в пространство имен

Наконец, чтобы знать, является ли класс подклассом другого, примените функцию `issubclass()`:

```
>>> issubclass(int, int)
True
>>> issubclass(FancyTire, Tire)
True
>>> issubclass(dict, float)
False
```

ПРИМЕЧАНИЕ

Функция `issubclass()` имеет несколько сбивчивое название. Поскольку она рассматривает класс как подкласс самого себя, она будет возвращать `True`, даже если два класса, которые вы предлагаете, на самом деле являются одним и тем же классом.

Эти инструменты могут пригодиться в реальном коде, но их присутствие часто настораживает, потому что для изменения поведения на основе типа данных существуют подклассы поведения. Эти встроенные функции

хороши для проверки объектов извне, но Python также предоставляет полезные средства для обработки наследования *внутри* классов.

8.3.2. Обращение к суперклассу

Предположим, что вы создаете подкласс, поведение которого должно зависеть от исходного поведения суперкласса. Как это сделать в Python? Можно использовать встроенную функцию `super()`, как показано в следующем листинге, которая перенаправляет к суперклассу любые обращения к методу или атрибуту.

Листинг 8.2. Использование функции super() для доступа к поведению суперкласса

```
class Teller:
    def deposit(self, amount, account):
        account.deposit(amount)

class CorruptTeller(Teller):
    def __init__(self):
        self.coffers = 0

    def deposit(self, amount, account):
        self.coffers += amount * 0.01
        super().deposit(amount * 0.99, account)
```

Коррумпированный кассир является кассиром

Коррумпированный кассир переопределяет поведение по умолчанию по внесении вклада

Коррумпированный кассир снимает сверху чуть-чуть для себя

Он вкладывает остальное так же, как и любой кассир, но используя другое количество денежных средств

Код, использующий `super()`, может стать запутанным, если нарушена подстановка. Переопределение методов для получения разного числа аргументов и передача только некоторых из них с помощью `super()` может усложнить сопровождение кода. Подстановка в Python особенно важна в случае *множественного наследования*.

8.3.3. Множественное наследование и порядок разрешения методов

До сих пор мы обсуждали *одиночное* наследование, при котором подкласс имеет только один суперкласс. Но Python также поддерживает идею

множественного наследования, позволяющего подклассу иметь несколько прямых суперклассов (рис. 8.5).

Множественное наследование используется в архитектурах плагинов или при реализации более одного интерфейса в одном классе. Например, водное транспортное средство имеет интерфейс и лодки, и автомобиля.

Вы можете наследовать от нескольких классов, представив их в определении класса (листинг 8.3). Попробуйте поместить этот код в модуль `cats`.

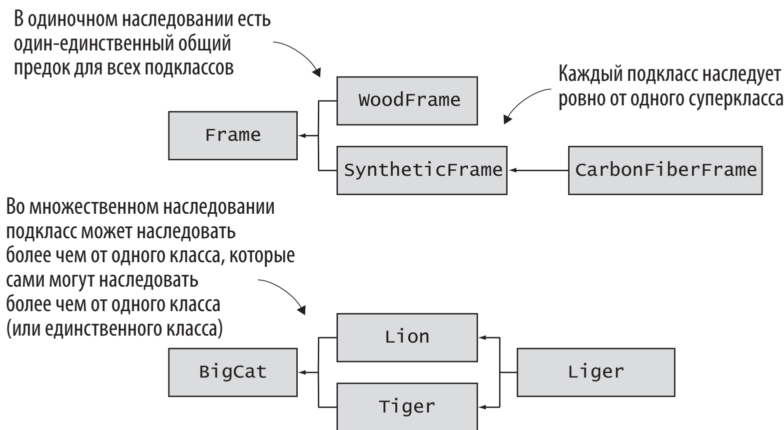


Рис. 8.5. Одиночное и множественное наследование

Угадайте, что делает `print(liger.eats())` перед выполнением кода?

Листинг 8.3. Множественное наследование в Python

```
class BigCat:
    def eats(self):
        return ['грызуны']

class Lion(BigCat):
    def eats(self):
        return ['антилопа гну']

class Tiger(BigCat):
    def eats(self):
        return ['индийский буйвол']

class Liger(Lion, Tiger):
```

Класс Lion является классом BigCat, полученным посредством одиночного наследования

Класс Tiger тоже является классом BigCat, полученным посредством одиночного наследования

Класс Liger (лигр — гибрид льва и тигрицы) использует множественное наследование, то есть является как классом Lion, так и классом Tiger

```
    def eats(self):
        return super().eats() + ['кролик', 'корова', 'свинья', 'курица']

if __name__ == '__main__':
    lion = Lion()
    print('Лев питается', lion.eats())
    tiger = Tiger()
    print('Тигр питается', tiger.eats())
    liger = Liger()
    print('Лигр питается', liger.eats())
```

Будет ли лигр питаться добычей, которую вы ожидали?

```
Лигр питается ['антилопа гну', 'кролик', 'корова', 'свинья', 'курица']
```

Он должен был есть добычу, подходящую `Lion` и `Tiger`, но при множественном наследовании метод `super()` работает несколько иначе. Когда вызывается `super().eats()`, Python начинает искать определение метода `eats()` с помощью процесса, именуемого *порядком разрешения методов*, или линеаризацией. Указанный процесс определяет список классов, которые Python будет искать, по порядку[1].

Вот шаги процесса разрешения методов:

1. Сгенерировать упорядоченную расстановку суперклассов сперва в глубину слева направо. Для класса `Liger` это класс `Lion` (крайний слева родитель), класс `BigCat` (единственный родитель класса `Lion`), класс `object` (неявный родитель класса `BigCat`), класс `Tiger` (следующий родитель класса `Liger`), класс `BigCat` и класс `object` (рис. 8.6).

2. Удалить все дубликаты. Список принимает вид `Liger`, `Lion`, `BigCat`, `object` и `Tiger`.

3. Переставить каждый класс так, чтобы он появлялся после всех своих подклассов. Окончательный список таков: `Liger`, `Lion`, `Tiger`, `BigCat`, `object`.

Как это выглядит для класса `Liger`? Полный процесс показан на рис. 8.7.

---

[1] То есть определяет порядок поиска в базисных классах при выполнении метода. — *Примеч. пер.*

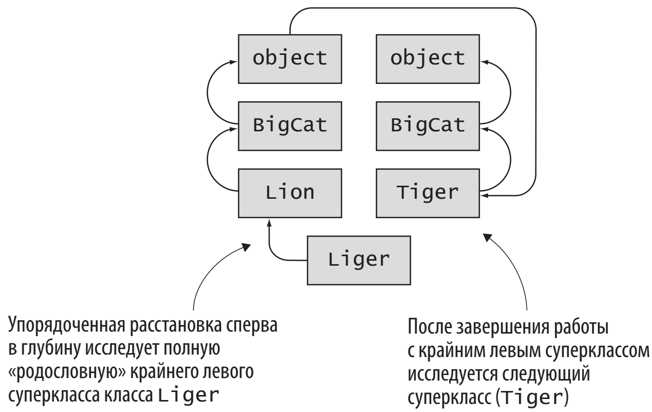


Рис. 8.6. Упорядоченная расстановка сперва в глубину для иерархии наследования классов

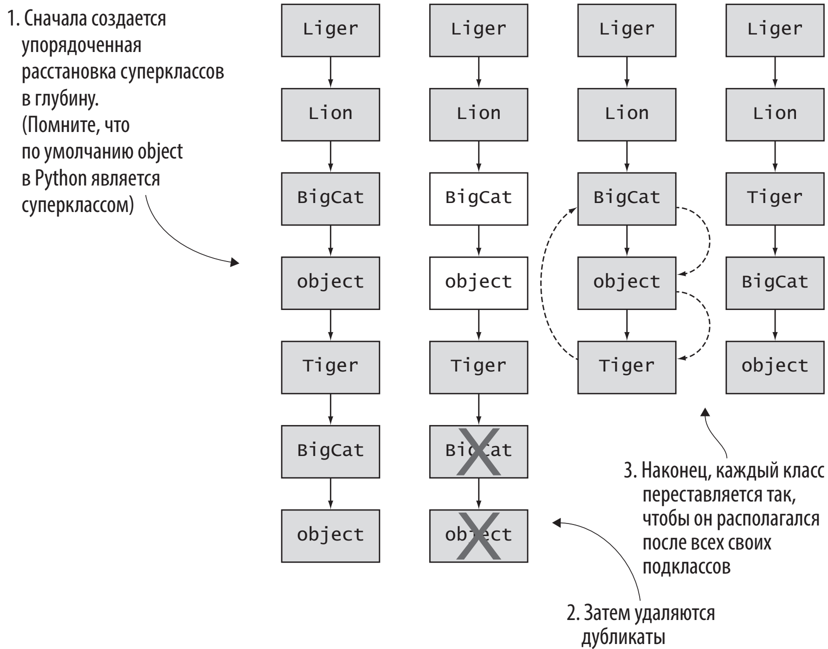


Рис. 8.7. Процедура определения в Python порядка разрешения методов для класса

Когда вы запрашиваете `super().eats()`, Python начинает путь по процессу разрешения методов до тех пор, пока он не найдет метод `eats()`, определенный на одном из классов (помимо того, из которого вы вызвали `super()`). Как видите, сначала он находит класс `Lion`, который возвращает `['антилопа гну']`. Затем класс `Liger` добавляет свой собственный список хищных животных, в результате чего получается список, который вы видели на выходе.

ИНСПЕКТИРОВАНИЕ ПОРЯДКА РАЗРЕШЕНИЯ МЕТОДОВ

Увидеть порядок разрешения методов для любого класса позволяет его атрибут `__mro__`:

```
>>> Liger.__mro__
(<class '__main__.Liger'>, <class '__main__.Lion'>,
➥ <class '__main__.Tiger'>, <class '__main__.BigCat'>,
   <class 'object'>)
```

Вы можете реализовать свои ожидания, практикуя *кооперативное* множественное наследование. Каждый класс возьмет на себя ответственность иметь одинаковые сигнатуры методов (подстановка) и вызывать `super().some_method ()` из своего `some_method()`. Наличие `super()` в каждом методе означает, что Python будет продолжать проходить процесс разрешения методов даже после нахождения метода. Так ни один класс не заблокирует выполнение и ничего не нарушит неожиданным интерфейсом. Классы прекрасно поладят.

Попробуйте обновить классы `Lion` и `Tiger`, чтобы вызвать `super().eats()`, так же, как делает метод `Liger.eats()`. Выполните код повторно и вернитесь сюда, чтобы убедиться в его совпадении со следующим результатом.

```
Лигр питается ['грызуны', 'индийский буйвол', 'антилопа гну', 'кролик',
➥'корова', 'свинья', 'курица']
```

Хотя множественное наследование используется не каждый день, важно знать, как с ним справиться, когда вы его встретите. По мере роста ПО

учащается вероятность использования разных парадигм, поэтому будьте готовы.

8.3.4. Абстрактные базовые классы

До сих пор я привирал о том, что интерфейсы недоступны в Python. Сначала нужно было разобраться, когда и как эффективно использовать наследование и композицию, но сейчас самое подходящее время копнуть немного глубже.

Абстрактные базовые классы в Python представляют собой способ использования чего-то, что выглядит как наследование, для достижения чего-то, что фактически является интерфейсом. Абстрактный базовый класс, как и формальный интерфейс в других языках, описывает то, какие методы и атрибуты должны имплементироваться его подклассами. Это возвращает к идее выполнения ролей из раздела 8.2.2. Вы не можете создать экземпляр абстрактного базового класса непосредственно, поскольку он действует как заготовка для поведения других классов.

Python предоставляет модуль `abc` для создания абстрактных базовых классов, который обеспечивает несколько полезных конструкций:

- Вы можете наследовать от класса `ABC`, указав на то, что ваш класс является абстрактным базовым.
- Вы можете пометить методы, определенные в вашем абстрактном базовом классе, как абстрактные, используя декоратор `@abstractmethod`. (Декораторы выходят за рамки темы этой книги, но вы можете думать об `abstractmethod` как о метке для метода, который определяете.) Этим соблюдается правило: методы должны быть определены в любом подклассе абстрактного класса.

Предположим, что вы моделируете пищевую цепочку и хотите, чтобы все классы хищников придерживались интерфейса, включающего метод `eat` для поедания добычи. Вы можете создать абстрактный базовый класс `Predator`, который определит этот метод и его сигнатуру, а затем создать подкласс, чтобы любой подкласс, который не определяет `eat`, возбуждал исключение.

Листинг 8.4. Использование абстрактного базового класса для реализации интерфейса

```
from abc import ABC, abstractmethod


class Predator(ABC):
    @abstractmethod
    def eat(self, prey):
        pass

class Bear(Predator):
    def eat(self, prey):
        print(f'Терзает {prey}!')


class Owl(Predator):
    def eat(self, prey):
        print(f'Налетает на {prey}!')

class Chameleon(Predator):
    def eat(self, prey):
        print(f'Выстреливает язык в {prey}!')

if __name__ == '__main__':
    bear = Bear()
    bear.eat('оленя')
    owl = Owl()
    owl.eat('мышь')
    chameleon = Chameleon()
    chameleon.eat('муху')
```

Наследование от ABC делает этот класс абстрактным базовым классом

Указывает на то, что метод должен быть определен для любых подклассов

Эта сигнатура метода может быть проверена интегрированными средами разработки в любых подклассах

Абстрактные методы не имеют реализацию по умолчанию

Констатирует ваше намерение реализовывать интерфейс путем создания подкласса абстрактного базового класса

Этот метод должен быть определен, или будет возбуждено исключение

> **СОВЕТ**
>
> Если вы используете интегрированную среду разработки, то она, возможно, предупредит вас о неправильной сигнатуре метода. Python не будет проверять ее во время выполнения, но может все же вызвать исключение в случае обычных ошибок, например, если аргументов слишком много или, наоборот, мало.

Попробуйте создать новый класс `Predator` без метода `eat` и его экземпляр в конце модуля. Вы должны увидеть ошибку `TypeError`, указывающую на то, что создать экземпляр не получается, потому что он не определяет реализацию абстрактного метода `eat()`.

Теперь попробуйте добавить метод в класс `Bear`, который заставляет его реветь. Что вы ожидаете увидеть?

1. Появляется ошибка `TypeError` при создании экземпляра, поскольку `Predator` не определяет `roar` как абстрактный метод.
2. Появляется ошибка `RuntimeError` при вызове `roar()`, поскольку `Predator` не определяет `roar` как абстрактный метод.
3. Он работает как любой нормальный метод класса.

Определение дополнительных методов в подклассе абстрактного базового класса работает на отлично (вариант 3). Абстрактный базовый класс требует, чтобы его подклассы *минимально* реализовывали методы, которые он определяет, но с дополнительным поведением все прекрасно, потому что подкласс все-таки реализует желаемый интерфейс. Кроме того, есть возможность вложить дополнительное поведение в сам базовый класс и получить его в подклассах как обычное наследование. Однако воздержитесь от этой практики, потому что вложение реального поведения в класс, который претендует на то, чтобы быть абстрактным, может сбить с толку любого, кто читает код.

Абстрактные базовые классы эффективно сотрудничают с утиной типизацией, предоставляя дополнительные защитные и гарантийные меры вокруг интерфейсов, которым классы должны следовать. Но я не часто ловлю себя на том, что тянусь к ним. Композиции через инверсию контроля обычно мне хватает. Попробуйте использовать оба варианта и посмотрите, какой из них уместнее в вашей ситуации.

Теперь, когда вы хорошо разбираетесь в разных аспектах наследования, давайте взглянем, какие возможности имеет Bark для наследования и композиции.

8.4. НАСЛЕДОВАНИЕ И КОМПОЗИЦИЯ В ПРИЛОЖЕНИИ BARK

Приложение Bark до сих пор не пользовалось наследованием. Видите, как далеко вы можете зайти без него? Но в этом последнем разделе главы вы увидите, как использовать наследование, чтобы сделать приложение надежнее.

8.4.1. Рефакторинг для использования абстрактного базового класса

Интерфейсы могут объявлять, что класс реализует конкретный набор методов и атрибутов. Еще вы только что узнали, что абстрактные базовые классы могут использоваться для усиления идеи интерфейсов в Python. Какие из следующих элементов придерживаются интерфейса в приложении Bark?

1. Команды в модуле `commands`.
2. Выполнение инструкции БД в модуле `database`.
3. Варианты действий в модуле `bark`.

Все варианты действий в модуле `bark` ведут себя одинаково, но для каждого варианта нет отдельного *класса*, только отдельные *экземпляры* класса `Option`. Это не похоже на интерфейс. Выполнение инструкции БД аналогичным образом ограничено действием внутри отдельного класса. А вот команды пользуются интерфейсами: каждый командный класс реализует метод `execute()`, который вызывается при запуске команды.

Чтобы все будущие команды не забывали реализовать метод `execute()`, рекомендую рефакторизовать модуль `commands` так, чтобы он использовал абстрактный базовый класс.

Вы можете назвать базовый класс `Command`, и он будет определять метод `execute()` как `abstractmethod`, который по умолчанию вызывает ошибку `NotImplementedError`. Каждый существующий командный класс должен наследовать от класса `Command`.

Обратите внимание, что все существующие командные классы уже реализуют `execute()`, поэтому они прикрыты на этом фронте. Но есть несколько отличающихся сигнатур метода `execute()`, что плохо отражается на подстановке и работе с абстрактными базовыми классами. Некоторые из них вызываются с аргументом `data`, тогда как другие не принимают никаких аргументов. Подумайте, как привести методы к одинаковой сигнатуре?

1. Удалить аргумент `data` из тех методов `execute()`, которые его принимают.
2. Добавить `data` в качестве необязательного именованного аргумента в методы `execute ()`, которые его еще не принимают.
3. Сделать так, чтобы все методы `execute()` принимали переменное число позиционных аргументов (`*args`).

Удаление аргумента `data` лишит вас возможности действовать на данных внутри команд, что приведет к удалению большого объема функциональности приложения. Хотя вариант 3 и будет работать, лучше четко указывать аргументы, которые вы принимаете, до тех пор, пока не понадобится гибкость для обработки разного количества аргументов. Прямо сейчас `execute()` либо нуждается в одном аргументе, либо не нуждается в никаком, поэтому я бы выбрал путь добавления `data` в качестве аргумента в каждый метод `execute()`.

Попробуйте создать абстрактный базовый класс `Command` и наследовать от него. Временно переименуйте методы `execute()` или измените их сигнатуры, чтобы посмотреть, как IDE (или приложение Bark) реагирует на нарушенный интерфейс. Проверьте результат по следующему листингу.

Листинг 8.5. Абстрактный базовый класс для паттерна Команда

```
from abc import ABC, abstractmethod   ◄── Импортирует необходимые
                                          инструменты из abc

class Command(ABC):   ◄── Определяет базовый класс Command
    @abstractmethod
    def execute(self, data):   ◄── Определяет execute как абстрактный метод,
                                   который принимает аргумент data
```

```
class CreateBookmarksTableCommand(Command):
    def execute(self, data=None):
        ...

class AddBookmarkCommand(Command):
    ...
```

Каждая команда наследует от Command

Добавляет аргумент data (None по умолчанию, чтобы вызывающие участки кода могли его опускать)

Командам, которые уже принимают аргумент data, нужно лишь унаследовать от Command

Поскольку `execute()` имеет согласованную сигнатуру, можно упростить строку кода в модуле `bark`, где вариант действий запускает команду в методе `choose()`:

```
class Option:
    ...

    def choose(self):
        ...

        message = self.command.execute(data)
```

Всегда передает data для выполнения

Приложение Bark должно продолжить работать как раньше. Добавление абстрактного базового класса просто сделало его безопаснее для создания будущих команд. Если вы решите, что команды должны реализовать дополнительные методы или принимать дополнительные аргументы, то сможете начать с добавления их в `Command`, а интегрированная среда разработки поможет отыскать места, которые необходимо обновить. Это очень удобный способ разработки.

8.4.2. Окончательная проверка работы с наследованием

Вы успешно применили наследование, чтобы сделать композицию надежнее. Проверьте еще раз, проходит ли код тесты Метца:

- *У команд мелкая узкая иерархия.* Семь командных классов в ширину, каждый из которых имеет один уровень иерархии в глубину.
- *Команды не знают о других объектах.* Они действительно пользуются объектом соединения с БД, но это является частью глобального состояния, которое придерживается интерфейса БД.

- *Команды используют или определяют всю функциональность из суперкласса.* `Command` является абстрактным классом, не имеющим собственного поведения.

Превосходно. Вы используете наследование там, где оно имеет смысл, и добавляет ценность, не навязывая эту структуру тому, что в ней не нуждается. Этот вид критического анализа еще не раз пригодится вам при написании и рефакторинге кода.

Переходите к следующей главе, чтобы узнать, как поддерживать классы в состоянии, пригодном для сопровождения, сохраняя их компактными.

ИТОГИ

- Используйте наследование для представления отношения «является» (хорошо подходит для специализации поведения).
- Используйте композицию для отношения «имеет» (хорошо подходит для многократного использования кода).
- Порядок разрешения методов играет ключевую роль в поддержании четкого понимания множественного наследования.
- Абстрактные базовые классы обеспечивают интерфейсноподобный контроль и безопасность в Python.

9 Поддержание компактности

В этой главе:

- ✓ Использование мер сложности для выявления кода, подлежащего рефакторингу.
- ✓ Использование средств языка Python для разделения кода.
- ✓ Использование средств языка Python для поддержки обратной совместимости.

До разделения ответственности вы будете наблюдать за тем, как фрагменты организуются сами, чтобы на основе увиденного создать более точные абстракции. Это означает, что ваши классы будут расти, пока не станут неуправляемыми.

Проведем аналогию с выращиванием дерева бонсай. Нужно дать ему время вырасти, и только после того, как оно куда-то «устремится», поддержать его на этом пути. Слишком частое подрезание дерева вызовет у него стресс, а придание ему неестественной формы может помешать цветению.

Так и с кодом. Из этой главы вы узнаете, как подрезать код, чтобы поддерживать его цветущий вид.

9.1. НАСКОЛЬКО БОЛЬШИМ ДОЛЖЕН БЫТЬ КЛАСС/ФУНКЦИЯ/МОДУЛЬ?

Многие онлайн-форумы по сопровождению ПО содержат вопросы такого рода. А я задаюсь вопросом: а не продолжаем ли мы его задавать, потому что надеемся выйти на какой-то новый уровень понимания? Каждый тред обычно содержит смесь мнений и забавных случаев.

Желание найти окончательный ответ на этот вопрос не так уж плохо, ведь имея в арсенале руководящие принципы, проще управлять своим временем. Но также важно понимать сильные и слабые стороны метрик, которые мы используем для решения этой проблемы.

9.1.1. Физический размер

Некоторые пытаются прописать лимит для функций, методов и классов в строках кода. Указанная метрика кажется хорошей, потому что ее легко измерить: «В моей функции 17 строк». Я не согласен с таким подходом, потому что он заставляет разработчика разбивать понятную функцию, увеличивая когнитивную нагрузку.

Если вы выдвинете ультиматум в пять строк, то о шестистрочной функции не может быть и речи. Это побуждает разработчиков играть в «кодовый гольф», пытаясь вместить одинаковый объем логики в меньшее число строк кода. Python тоже дает возможность поиграть в такую игру:

```
def valuable_customers(customers):
    return [customer for customer in customers if customer.active and
➥ sum(account.value for account in customer.accounts) > 1_000_000]
```

Удалось ли вам сразу разобраться в этом коде? Он не ужасен, но разве сведение всего этого кода в одну строку повышает его ценность?

Взгляните на переписанную версию, где каждой составной части дана своя строка:

```
def valuable_customers(customers):
    return [
        customer
        for customer in customers
        if customer.active
        and sum(account.value for account in customer.accounts) >
                1_000_000
    ]
```

Разложение частей в логическом порядке дает человеку, читающему код, возможность переварить каждую составляющую, формируя мысленную модель того, что происходит, по порядку.

Еще одно негласное правило: «Класс должен умещаться на экране». Оно учитывает некоторые болевые точки его более строгой версии, но в то же время менее измеримо из-за разных размеров экрана и разрешающей способности.

Суть этих метрик в упрощении. Но есть и другие способы упрощать.

9.1.2. Ограниченная ответственность

Более открытое измерение класса, метода или функции заключается в том, за сколько дел они отвечают. Как вы уже поняли из разделения ответственности, для функций и методов это означает выполнение одного вычисления или задачи, а для классов — работа со сфокусированным аспектом более крупной бизнес-задачи.

Если вы заметили функцию, выполняющую две задачи, или класс, содержащий два четко различимых участка фокуса, то это сигнал о возможности разделения. Бывают моменты, когда единственная задача оказывается многосложной, что тоже оправдывает ее дальнейшее разложение.

9.1.3. Сложность кода

Когнитивная нагрузка и сопроводимость кода определяются понятием *сложности*. Как и временная и пространственная сложности, сложность кода — это количественное измерение характеристик кода, а не просто субъективная мера того, насколько вы запутываетесь, когда его читаете.

Инструменты для измерения сложности нужно иметь под рукой. Я нахожу, что они часто точно указывают на малопонятный код. В следующих нескольких разделах вы увидите сложность кода и несколько инструментов для ее измерения.

Измерение сложности кода

Часто встречающейся мерой сложности является цикломатическая сложность. Хотя это название звучит пугающе и заумно, измерение *цикломатической* сложности предусматривает определение числа путей выполнения кода посредством функции или метода. Структура (а следовательно, и сложность) функции зависит от числа содержащихся в ней условных блоков и циклов.

Чем выше балл сложности функции или метода, тем больше условных блоков и циклов вы должны от нее или него ожидать. Конкретный балл не всегда полезен, гораздо важнее его тенденция и то, как он реагирует на изменения. Стремитесь снизить свои баллы сложности со временем и учитывайте фрагменты кода с высокой сложностью при определении места для рефакторинга.

Вы можете сами измерить сложность функции. Создав граф *потока управления* или путь, который код выберет по мере выполнения инструкций, вы сможете подсчитать число вершин и ребер в графе и вычислить цикломатическую сложность. Ниже представлены вершины в графе потока управления программой:

- «Старт» функции (место, куда входит поток управления).
- Условия `if` / `elif` / `else` (каждое из них является самостоятельной вершиной).
- Циклы `for`.
- Циклы `while`.
- «Конец» цикла (место, где вы чертите путь выполнения обратно к началу цикла).
- Инструкции `return`.

Рассмотрим функцию в следующем листинге, которая принимает предложение либо как строку, либо как список слов и определяет, есть ли в предложении длинные слова. Она содержит цикл и несколько условных блоков.

Листинг 9.1. Функция с условными блоками и циклом

```
def has_long_words(sentence):
    if isinstance(sentence, str):        ← Разбивает слова в предложении, если оно имеет строковый тип (условный блок)
        sentence = sentence.split(' ')

    for word in sentence:                ← Выполняет работу для каждого слова (цикл)
        if len(word) > 10:               ← Возвращает True, если найдено длинное слово (условный блок)
            return True

    return False                         ← Возвращает False, если ни одно слово не было длинным
```

Ребра — это стрелки, которые следуют разными путями выполнения кода. Цикломатическая сложность M для функции или метода равна числу ребер минус число вершин плюс 2. Вы можете добавить вершины и ребра для строк кода, которые не находятся внутри условного блока или цикла, если это поможет вам построить диаграмму функции, но они не повлияют на общую сложность — каждый из них добавляет одну вершину и одно ребро, которые в математике взаимно исключаются.

В функции `has_long_words` один условный блок проверяет, что входное значение имеет строковый тип, цикл для каждого слова в предложении и условный блок внутри цикла с проверкой, является ли слово длинным (рис. 9.1). Схематизируя поток управления и упрощая граф в виде простых вершин и ребер, вы можете их подсчитать и включить результаты в уравнение цикломатической сложности. В этом случае граф функции `has_long_words` имеет 8 вершин с 10 ребрами, поэтому его сложность равна $M = E - N + 2 = 10 - 8 + 2 = 4$.

Большинство источников рекомендуют нацеливаться на сложность 10 или ниже для заданной функции или метода. Это примерно соответствует количеству информации, которое разработчики могут понять сразу.

Цикломатическая сложность также полезна в тестировании, поскольку выступает минимальным числом четко различимых тестовых случаев, которые нужно описать, чтобы покрыть каждый путь выполнения. Это

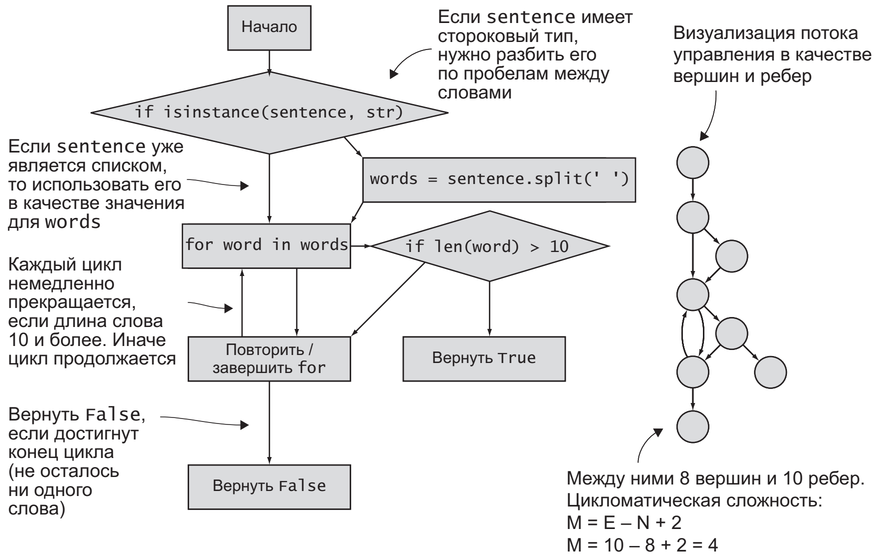


Рис. 9.1. Схема потока управления для измерения цикломатической сложности

следует из того факта, что инструкции `if`, `while` и т. д. требуют подготовки разного набора предусловий для проверки их работы в разных случаях.

Помните, что идеальное тестовое покрытие не гарантирует, что код работает, оно лишь означает, что тесты обусловили выполнение некоторой части кода. Но обеспечение покрытия интересующих вас путей выполнения важно. Непротестированные ветви выполнения обычно называются «случаями, о которых мы не подумали». Отличный пакет Coverage от Неда Бэтчелдера (https://coverage.readthedocs.io) поможет выводить метрики покрытия ветвей выполнения для тестов.

СЛОЖНОСТЬ ПО ХОЛСТЕДУ

Для некоторых приложений снижение риска поставки дефектного ПО является таким же приоритетом, как и сопроводимость. Хотя уменьшение числа ветвей в коде делает его более удобочитаемым и понятным, не доказано, что оно уменьшает число багов. Цикломатическая сложность предсказывает число дефек-

тов примерно так же, как и число строк кода. Но существует по крайней мере один набор показателей, который пытается решить проблему частоты дефектов.

Сложность по Холстеду количественно измеряет идеи уровня абстракции, сопроводимости и частоты дефектов. Замер сложности по Холстеду предусматривает проверку использования программой встроенных инструкций языка программирования и числа переменных и выражений, которые они содержат. Эта мера выходит за рамки темы данной книги, но я рекомендую прочитать о ней больше (https://ru.qwe.wiki/wiki/Halstead_complexity_measures). Radon (https://radon.readthedocs.io) может измерять сложность по Холстеду программ на Python, если вы заинтересованы в ее определении.

Вспомните код, который вы написали для импорта звезд GitHub в Bark (воспроизведенный в листинге ниже). Попробуйте построить диаграмму потока управления и рассчитать цикломатическую сложность.

Листинг 9.2. Код для импортирования звезд GitHub в приложение Bark

```
def execute(self, data):
    bookmarks_imported = 0

    github_username = data['github_username']
    next_page_of_results =
➥ f'https://api.github.com/users/{github_username}/starred'

    while next_page_of_results:                         ◄── Цикл, в который вернется код,
        stars_response = requests.get(                      расположенный дальше внизу
            next_page_of_results,
            headers={'Accept': 'application/vnd.github.v3.star+json'},
        )
        next_page_of_results = stars_response.links.get('next',
                               {}).get('url')           Еще один цикл, в который
                                                        вернется код, расположен-
        for repo_info in stars_response.json():   ◄──   ный дальше внизу
            repo = repo_info['repo']

            if data['preserve_timestamps']:   ◄── Одна ветвь исполнения
                timestamp = datetime.strptime(
```

```
                repo_info['starred_at'],
                '%Y-%m-%dT%H:%M:%SZ'
            )
        else:   ◄─── Еще одна ветвь исполнения
            timestamp = None

        bookmarks_imported += 1
        AddBookmarkCommand().execute(
            self._extract_bookmark_info(repo),
            timestamp=timestamp,
        )   ◄─── Место, которое возвращается к for либо, если завершено, к while
return f'Импортировано {bookmarks_imported} закладок из помеченных
         звездами репо!'
```

Когда закончите, сверьте свою работу с решением, представленным на рис. 9.2.

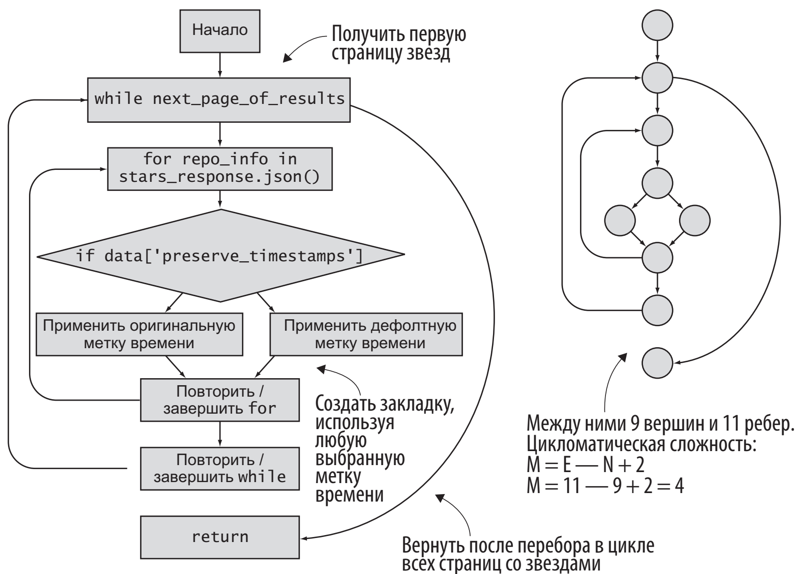


Рис. 9.2. Цикломатическая сложность функции из приложения Bark

К счастью, не нужно строить диаграмму каждой функции и метода, которые вы пишете. Целый ряд инструментов, таких как SonarQube (www.sonarqube.org) и Radon (https://radon.readthedocs.io), может сделать это за вас. Эти инструменты можно даже интегрировать в редакторы кода,

в результате чего вы получите возможность разбивать сложный код по мере разработки.

Теперь, когда вы усвоили некоторые способы обнаружения повышенной сложности кода, давайте попрактикуемся в разложении этой сложности.

9.2. РАЗЛОЖЕНИЕ СЛОЖНОСТИ

У меня для вас парочка трудных заданий. Распознать повышенную сложность кода — самое щадящее из них. Следующая трудность — вести дела с конкретными видами сложности. На протяжении остальной части главы я буду указывать на паттерны сложности, которые сам встречал в Python, и покажу варианты их решения.

9.2.1. Извлечение конфигурационной информации

Начну с примера, который вы уже видели: по мере роста ПО некоторые участки кода должны продолжать адаптироваться к новым требованиям.

Представьте, что вы создаете веб-сервис, у которого нерешительные пользователи будут спрашивать, что им съесть на обед. Если пользователь переходит в конечную точку сервиса `/random`, то должен получить в ответ случайную еду. Начальная функция-обработчик принимает запрос пользователя в качестве аргумента:

```
import random

FOODS = [  ◄── Список продуктов питания (в конечном итоге это может уйти в БД)
    'пицца',
    'бургеры',
    'салат',
    'суп',
]

def random_food(request):  ◄── Эта функция принимает HTTP-запрос пользователя (в настоящее время не используется)
    return random.choice(FOODS)  ◄── Возвращает случайный продукт питания из списка в качестве строкового значения
```

Когда ваш сервис становится популярным (*все* люди нерешительны), некоторые пользователи хотят создать вокруг него полноценное приложение.

Они говорят вам, что хотят получать от вас отклик в формате JSON, потому что с ним легко работать. Вы не хотите менять поведение по умолчанию для остальных пользователей, и поэтому говорите им, что будете возвращать ответ JSON, если в своем запросе они будут отправлять заголовок `Accept: application/json`. (Не беспокойтесь о том, как работают заголовки HTTP, просто допустим, что `request.headers` — это словарь пар «имя заголовка — значение заголовка».) Обновите функцию, чтобы это учесть:

```
import json
import random

...

def random_food(request):
    food = random.choice(FOODS)

    if request.headers.get('Accept') == 'application/json':
        return json.dumps({'еда': food})
    else:
        return food
```

Выбирает продукт питания наугад и сохраняет его для немедленного использования

Возвращает, к примеру, {«еда»: «пицца»}, если запрос имеет заголовок Accept: application/json

Продолжает возвращать, к примеру, «пицца» по умолчанию

Подумайте об этом изменении в терминах цикломатической сложности: какова сложность до и после изменения?

1. 1 до, 2 после.
2. 2 до, 2 после.
3. 1 до, 3 после.
4. 2 до, 1 после.

Первоначальная функция не имела ни условий, ни циклов, поэтому сложность была равна 1. Поскольку вы добавили только одно новое условие (случай, когда пользователь запрашивает JSON), то сложность выросла с 1 до 2.

Ничего страшного, если сначала вы получите увеличение сложности на 1, чтобы справиться с новым требованием. Но если вы продолжите двигаться по этой траектории в течение длительного времени, линейно увеличивая сложность с каждым требованием, то скоро будете иметь дело с пугающе разросшимся кодом:

```
...

def random_food(request):
    food = random.choice(FOODS)

    if request.headers.get('Accept') == 'application/json':
        return json.dumps({'food': food})
    elif request.headers.get('Accept') == 'application/xml':
        return f'<response><food>{food}</food></response>'
    else:
        return food
```

Каждое дополнительное требование является новым условием, увеличивающим сложность

Помните, как решить эту проблему? Подскажу: обратите внимание на то, что условия увязывают значение (значение заголовка `Accept`) с еще одним значением (отклик с возвратом). Какая структура данных имеет смысл?

1. `list`.
2. `tuple`.
3. `dict`.
4. `set`.

Словарь Python (вариант 3) увязывает значения с другими значениями, поэтому он хорошо подходит для рефакторинга этого кода. Переработка потока выполнения в форму конфигурационных пар «значение заголовка — формат отклика», а затем выбор правильного из них на основе запроса пользователя упростит дело.

Попробуйте извлечь разные значения заголовков и типы откликов в словарь, используя поведение по умолчанию в качестве запасного варианта, если пользователь не запросит формат отклика (или запросит неизвестный формат). Когда закончите работу, сверьте ее со следующим листингом.

Листинг 9.3. Конечная точка с извлеченной конфигурацией

```
...

def random_food(request):
    food = random.choice(FOODS)

    formats = {
        'application/json': json.dumps({'food': food}),
        'application/xml': f'<response><food>{food}</food></response>',
    }

    return formats.get(request.headers.get('Accept'), food)
```

Извлечено из предыдущих условий if / elif

Получает запрошенный формат отклика, если таковой имеется, в противном случае откатывает назад, возвращая обычное строковое значение

Хотите верьте, хотите нет, но это новое решение приводит обратно к цикломатической сложности 1. И даже если вы продолжите добавлять записи в словарь форматов, то никакой дополнительной сложности добавлено не будет. Это тот самый выигрыш, о котором я говорил в главе 4. Вы перешли от линейного алгоритма к константному.

По моему опыту, извлечение конфигурационной информации в словарь делает код удобочитаемым. Попытка просеять ряд условий `if / elif` утомляет, даже если они похожи. Напротив, ключи словаря можно просканировать и быстро отыскать.

Куда уж лучше?

9.2.2. Извлечение функций

Мы победили растущую цикломатическую сложность, но две другие вещи по-прежнему продолжают расти внутри функции `random_food`:

- код, который знает, *что* делать (форматировать ответ как JSON, XML и т. д.);
- код, который знает, *как решать*, что делать (на основе значений заголовка `Accept`).

Хорошая возможность разделить ответственность. Если вы посмотрите на каждый элемент в словаре `formats`, то заметите, что их значения могут стать функцией, принимающей аргумент `food` и возвращающей отформатированный отклик, который будет возвращен пользователю (рис. 9.3).

Попробуйте изменить свою функцию `random_food`, чтобы использовать эти разделенные функции отформатированного отклика. Теперь словарь будет увязывать форматы с функцией, которая может возвращать отклик для этого формата, и `random_food` будет вызывать эту функцию со значением `food`.

Если ни одна функция не доступна после вызова `formats.get(…)`, вернитесь к функции, которая возвращает значение `food` неизменным. Это

можно сделать с помощью лямбды. Когда закончите, сверьте результат со следующим листингом.

Листинг 9.4. Конечная точка сервиса с функциями форматирования откликов

```
def to_json(food):                      ← Извлеченные функции форматирования
    return json.dumps({'food': food})


def to_xml(food):
    return f'<response><food>{food}</food></response>'


def random_food(request):
    food = random.choice(FOODS)

    formats = {                         ← Теперь увязывает форматы данных с соответству-
                                          ющими функциями форматирования
        'application/json': to_json,
        'application/xml': to_xml,
    }

    format_function = formats.get(      ← Получает соответствующую функцию
                                          форматирования, если она имеется
        request.headers.get('Accept'),
        lambda val: val                 ← Использует лямбду в качестве запасного варианта
                                          для возврата неизменного значения переменной food
    )
    return format_function(food)        ← Вызывает функцию форматирования
                                          и возвращает ответ
```

Рис. 9.3. Извлечение внутристрочных выражений в качестве функций

Чтобы разделить обязанности полностью, извлеките `formats` и работу по получению из нее нужной функции в отдельную функцию `get_format_function`. Новая функция будет принимать значение заголовка `Accept` от пользователя и возвращать правильную функцию форматирования. Сверьтесь со следующим листингом.

Листинг 9.5. Разделение ответственности на две функции

```
def get_format_function(accept=None):
    formats = {
        'application/json': to_json,
        'application/xml': to_xml,
    }

    return formats.get(accept, lambda val: val)


def random_food(request):
    food = random.choice(FOODS)
    format_function = get_format_function(request.headers.get('Accept'))
    return format_function(food)
```

Определяется с тем, какую функцию форматирования использовать

random_food теперь имеет три коротких шага

Ранее перемешанные зоны ответственности теперь абстрагированы в вызовы функций

Возможно, вы думаете, что код стал сложнее: была одна функция, а стало четыре. Но каждая из этих функций имеет цикломатическую сложность 1, вполне удобочитаема и имеет хорошее разделение ответственности.

У вас на руках теперь *расширяемый* код, потому что создание новых форматов откликов будет выглядеть так:

1) добавить новую функцию для форматирования отклика;
2) добавить связку требуемого значения заголовка `Accept` и новой функции форматирования;
3) получить прибыль.

Вы можете создать новое бизнес-значение, просто добавив новый код и обновив конфигурацию. Идеально.

Теперь, когда вы знаете несколько трюков для функций, я покажу кое-что для классов.

9.3. ДЕКОМПОЗИЦИЯ КЛАССОВ

Классы могут расти неуправляемо, как функции, а возможно, и еще быстрее. Но иногда кажется, что разложить класс куда сложнее, чем разложить функцию. Функции ощущаются как строительные блоки, а классы — как готовые изделия. И это тот барьер, через который я часто стараюсь перешагнуть.

У вас должно быть достаточно уверенности, чтобы разбивать классы так же часто, как функции. Классы — это просто еще один инструмент. Когда вы обнаруживаете, что класс начинает расти в сложности, это обычно происходит из-за перемешивания зон ответственности. Как только вы выявляете внутри класса самостоятельный объект, начинайте разбивать.

9.3.1. Сложность инициализации

Я часто встречаю классы, которые содержат сложные процедуры инициализации. Хорошо это или плохо, но эти классы обычно являются сложными, потому что они имеют дело со сложными структурами данных. Вы когда-нибудь встречали такой класс, как в следующем листинге?

Листинг 9.6. Класс со сложной доменной логикой при его конструировании

```
class Book:
    def __init__(self, data):
        self.title = data['заголовок']
        self.subtitle = data['подзаголовок']

        if self.title and self.subtitle:
            self.display_title = f'{self.title}: {self.subtitle}'
        elif self.title:
            self.display_title = self.title
        else:
            self.display_title = 'Не озаглавлена'
```

Извлекает некоторые поля из передаваемых данных

Сложность, возникающая из доменной логики бизнеса

Когда логика домена, с которой вы имеете дело, является сложной, код, скорее всего, будет это отражать. В таких случаях для разработчиков

более чем когда-либо важно опираться на полезные абстракции, чтобы уловить смысл происходящего.

Один из подходов, который вы можете здесь использовать, заключается в извлечении логики для `display_title` в метод `set_display_title`, который можно вызывать из метода `__init__`, как показано в следующем листинге. Попробуйте создать модуль `book` и добавить в него класс `Book`, извлекая сеттер (метод-модификатор) для `display_title`.

Листинг 9.7. Использование сеттера для упрощения процедуры конструирования класса

```
class Book:
    def __init__(self, data):
        self.title = data['заголовок']
        self.subtitle = data['подзаголовок']
        self.set_display_title()  ◄── Вызывает извлеченную функцию

    def set_display_title(self):  ◄── Извлеченная функция задает display_title
        if self.title and self.subtitle:
            self.display_title = f'{self.title}: {self.subtitle}'
        elif self.title:
            self.display_title = self.title
        else:
            self.display_title = 'Не озаглавлена'
```

Этот код очистил метод `__init__`, но возникло несколько трудностей:

- геттеры и сеттеры в Python обычно не поощряются, потому что загромождают класс;
- хорошей практикой является установка всех необходимых атрибутов в некоторое начальное значение непосредственно внутри `__init__`, но `display_title` устанавливается в другом методе.

Вы можете исправить последнее, установив переменную `display_title` по умолчанию равной значению `'Не озаглавлена'`, но читатель может из этого заключить, что показываемый заголовок в типичной ситуации (или даже всегда) равен значению `'Не озаглавлена'`.

Существует один подход, который дает преимущество в читабельности от извлечения метода, не заставляя никого страдать от недостатков. Он

предусматривает создание функции, которая возвращает значение для `display_title`.

Но подождите! Если подумать о том, как используется класс `Book`, то это может быть что-то вроде этого:

```
...
book = Book(data)
return book.display_title
```

Как сделать логику `display_title` функцией без необходимости обновлять вторую строку кода, вместо этого возвращая `book.display_title()`? К счастью, для этого случая Python предоставляет особый инструмент. Декоратор `@property` можно использовать для обозначения того, что метод класса должен быть доступен в качестве атрибута.

Теперь создайте *метод* `display_title`, декорированный атрибутом `@property`, который использует существующую логику для возврата надлежащего показываемого заголовка. Когда закончите, сравните ваши изменения с листингом 9.8.

> **ПРИМЕЧАНИЕ**
>
> Методы могут использоваться в качестве свойств только в том случае, если `self` является их единственным аргументом, поскольку при обращении к атрибуту вы не можете передать ему никакие аргументы.

Листинг 9.8. Использование атрибута @property для упрощения конструирования классов

```
class Book:
    def __init__(self, data):
        self.title = data['заголовок']
        self.subtitle = data['подзаголовок']

    @property
    def display_title(self):    ◄── property — это функция, на которую
                                    можно ссылаться как на атрибут
        if self.title and self.subtitle:
            return f'{self.title}: {self.subtitle}'
```

```
    elif self.title:
        return self.title
    else:
        return 'Untitled'
```

Используя атрибут `@property`, вы все равно можете обратиться к `book.display_title` как к атрибуту, но вся его сложность абстрагируется в самостоятельную функцию. Это уменьшит сложность и увеличит удобочитаемость метода `__init__`. Я часто использую `@property`.

> **ПРИМЕЧАНИЕ**
>
> Поскольку свойства являются методами, многократное к ним обращение означает, что методы вызываются каждый раз. Иногда это может влиять на производительность, если вычисление свойств обходится дорого.

Что нужно делать, когда имеется много функциональности, достаточной для того, чтобы абстрагировать целый *класс* методов?

9.3.2. Извлечение классов и переадресация вызовов

Когда вы извлекали функцию `get_format_function` из `random_food` в разделе 9.2.2, вы все равно *вызывали* ее из исходного местоположения. С классами нужно поступить так же, если нужно сохранить обратную совместимость. *Обратная совместимость* — это практика эволюции ПО без нарушения реализации, на которую потребители опирались ранее. Если вы измените аргументы функции, имя класса и т. д., потребителям понадобится обновить свой код, чтобы он продолжал работать. Во избежание этих проблем вы можете воспользоваться сторонней подсказкой от системы пересылки почты в почтовом отделении.

Когда вы переезжаете на новый адрес, то можете попросить почтовое отделение пересылать вашу почту (рис. 9.4). И всякий раз, когда вы получаете письмо, адресованное по вашему старому месту жительства, то можете уведомлять отправителя о своем новом адресе, чтобы он мог обновить свои записи. Как только вы будете уверены, что больше не

получаете почту, оформленную на старый адрес, то можете прекратить пересылку почты.

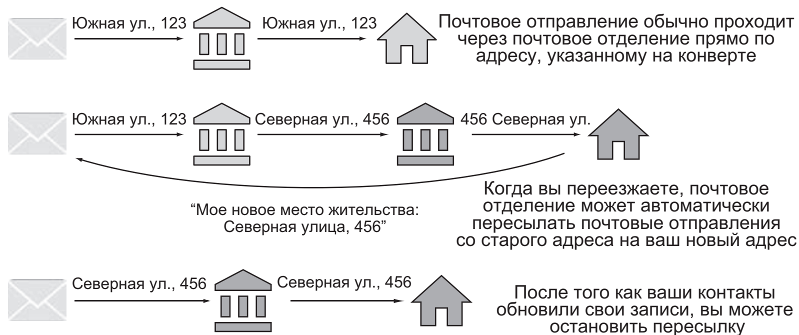


Рис. 9.4. Почтовые отправления могут переадресовываться почтовым отделением при переезде на новое место

Выделив один класс из другого, вы захотите продолжать предоставлять ранее существовавшую функциональность по старой схеме в течение некоторого времени, благодаря чему потребителям не нужно будет немедленно обновлять ПО. Как и в случае с почтой, вы сможете продолжать принимать вызовы в одном классе и передавать их в другой класс под капотом. Это называется *переадресацией*.

Предположим, что ваш класс `Book` вырос настолько, что требуется отслеживать информацию об авторе. Это кажется естественным с самого начала: что такое книга без автора? Но по мере того как класс приобретает все большую функциональность, автор начинает ощущаться как отдельная зона ответственности. Как показано в следующем листинге, вскоре возникнут методы для имени автора, для того, как оно должно показываться на веб-сайте, а также в цитировании научной статьи.

Листинг 9.9. Класс Book с чрезмерной ответственностью относительно данных об авторе

```
class Book:
    def __init__(self, data):
        # ...

        self.author_data = data['автор']   # Сохраняет автора в форме словаря из data
```

```
    @property
    def author_for_display(self):
        return f'{self.author_data["имя"]}
➥ {self.author_data["фамилия"]}'

    @property
    def author_for_citation(self):
        return f'{self.author_data["фамилия"]},
➥ {self.author_data["имя"][0]}.'
```

Показывает автора как, например, «Дейн Хиллард»

Получает подходящее для цитирования имя автора как, например, «Хиллард, Д.»

Предположим, что вы бы использовали этот класс `Book` вот так:

```
book = Book({
    'заголовок': 'Бриллиант',
    'подзаголовок': 'Камень, который все изменил',
    'автор': {
        'имя': 'Расти',
        'фамилия': 'Поттс',
    }
})

print(book.author_for_display)
print(book.author_for_citation)
```

Способность ссылаться на `book.author_for_display` и `book.author_for_citation` была великолепной, и вы хотели бы ее сохранить. Но ссылка на словарь `author` в этих свойствах начинает выглядеть громоздкой, если с классом `author` нужно будет делать что-то еще. Как вы поступите дальше?

1. Извлечете класс `AuthorFormatter` для форматирования имен авторов разными способами.
2. Извлечете класс `Author` для инкапсуляции форм поведения и информации, связанных с автором.

Хотя класс для форматирования имен авторов (вариант 1) мог бы представлять ценность, извлечение класса `Author` (вариант 2) обеспечит более подходящее разделение ответственности. Когда несколько методов в классе имеют общий префикс или суффикс, особенно тот, который не совпадает с именем класса, вполне возможно, что новый класс требует извлечения. Пришло время попробовать свои силы в извлечении класса.

Создайте класс Author (либо в том же модуле, либо импортированный из нового модуля). Он должен содержать всю ту же информацию, что и раньше, но в более структурированной форме, а также:

- Принимать author_data в форме словаря в __init__, сохраняя каждое соответствующее значение (имя, фамилию и т. д.) из словаря в качестве атрибута.
- Иметь два свойства, for_display и for_citation, которые возвращают надлежаще отформатированное строковое значение с данными автора.

Помните, что класс Book должен продолжать работать для пользователей, поэтому сохраните существующие формы поведения author_data, author_for_display и author_for_citation, определенные в данный момент на классе Book. Инициализируя экземпляр класса Author данными author_data, вы можете переадресовывать вызовы из Book.author_for_display в Author.for_display и т. д. Благодаря этому класс Book позволит классу Author делать основную часть работы, оставляя только временную систему на своем месте ради того, чтобы убедиться, что вызовы продолжают работать. Попробуйте прямо сейчас и вернитесь к следующему листингу, чтобы проверить результат.

Листинг 9.10. Извлечение класса Author из класса Book

```
class Author:
    def __init__(self, author_data):
        self.first_name = author_data['имя']
        self.last_name = author_data['фамилия']

    @property
    def for_display(self):
        return f'{self.first_name} {self.last_name}'

    @property
    def for_citation(self):
        return f'{self.last_name}, {self.first_name[0]}.'


class Book:
    def __init__(self, data):
        # ...

        self.author_data = data['автор']
        self.author = Author(self.author_data)
```

То, что раньше хранилось только в форме словаря, теперь является структурированными атрибутами

Свойства уровня класса Author теперь проще, чем исходные

Продолжает хранить author_data до тех пор, пока потребители не перестанут им пользоваться

Хранит экземпляр класса Author для переадресации вызовов

```
    @property
    def author_for_display(self):
        return self.author.for_display

    @property
    def author_for_citation(self):
        return self.author.for_citation
```

Заменяет предшествующую логику на переадресацию к экземпляру класса Author

Заметили ли вы, что, несмотря на то что код теперь содержит больше строк, каждая строка упрощена? И глядя на эти классы, теперь легче сказать, какую информацию они содержат. Рано или поздно бо́льшая часть кода, все еще находящегося в классе `Book`, тоже будет удалена, и с этого момента класс `Book` будет задействовать композицию класса `Author` для предоставления информации об авторах.

Если вы *по-настоящему* хорошо относитесь к потребителям, то при разборе класса на части попросите их переключиться на новый код. Чтобы потребители книги перешли, например, с `book.author_for_display` на `book.author.for_display`, Python предлагает отправить им предупреждения `warning`.

Также существует предупреждение об устаревании кода `DeprecationWarning`, которое тоже необходимо отправлять пользователям. Обычно оно выводит сообщение среди выходных данных программы, сообщая пользователю, что он должен внести изменения, например:

```
import warnings

warnings.warn('Больше это не используйте!', DeprecationWarning)
```

Вы можете помочь потребителям плавно обновить свой код, добавив предупреждение об устаревании в каждый метод, который собираетесь удалить.[1] Теперь попробуйте добавить его в свойства, связанные с автором, в классе `Book`. Добавьте полезную информацию: `'Вместо этого используйте book.author.for_display'`. Если вы выполните код

[1] Бретт Слаткин, «Рефакторинг Python: зачем и как реструктурировать свой код» (*Refactoring Python: Why and how to restructure your code*), PyCon, 2016 г., www.youtube.com/watch?v=D_6ybDcU5gc — сокровищница трюков по выведению кода из эксплуатации и его извлечению.

прямо сейчас, то среди выходных данных увидите предупреждающие сообщения:

```
/path/to/book.py:24: DeprecationWarning: Вместо этого используйте book.
author.for_display
```

Поздравляю! Вы извлекли новый класс, разложив на части класс с повышенной сложностью. Вы сделали это обратносовместимым путем, оставив подсказки для пользователей, чтобы они знали, что происходит и как это исправить. Теперь у вас есть хорошо структурированный и удобочитаемый код с разделенной ответственностью и высокой связностью. Отличная работа!

ИТОГИ

- Сложность кода и разделенная ответственность являются более точными метриками для разложения кода на части, чем физический размер.
- Цикломатическая сложность измеряет число путей выполнения в коде.
- Извлекайте конфигурационную информацию, функции, методы и классы без ограничений для разложения сложности на части.
- Используйте переадресацию и предупреждения об устаревании кода для одновременной поддержки новых и старых способов выполнения работы.

10 Достижение слабой сопряженности

В этой главе:

- ✓ Распознавание признаков тесно сопряженного кода.
- ✓ Стратегии уменьшения сопряженности.
- ✓ Программирование, ориентированное на обработку сообщений.

Слабая сопряженность позволяет безопасно вносить изменения в разные участки кода. С ее помощью вы можете работать над одной характеристикой, пока ваш коллега занимается другой. Она также является основой для других желательных характеристик, таких как расширяемость. Без слабой сопряженности работа по сопровождению кода может быстро выйти из-под контроля.

В этой главе вы познакомитесь с некоторыми неудобствами тесной сопряженности и узнаете, как с ними бороться.

10.1. ОПРЕДЕЛЕНИЕ СОПРЯЖЕННОСТИ

Поскольку идея сопряженности (coupling) играет такую большую роль в эффективной разработке ПО, очень важно четко понимать, что она означает. Что такое сопряженность? Вы можете думать о ней как о соединительной ткани между разными участками вашего кода.

10.1.1. Сопряженность

Поначалу понятие сопряженности может показаться запутанным. Это своего рода сетка, которая пронизывает весь код (рис. 10.1). Там, где два фрагмента кода имеют высокую взаимозависимость, эта сетка тесно сплетена и натянута. Перемещение любого фрагмента кода требует, чтобы другой тоже перемещался. Сетка между участками с малой взаимозависимостью или без нее является гибкой — возможно, она сделана из резинок. Вам пришлось бы изменять код в этой более свободной части сетки гораздо более радикально, чтобы он повлиял на код вокруг нее.

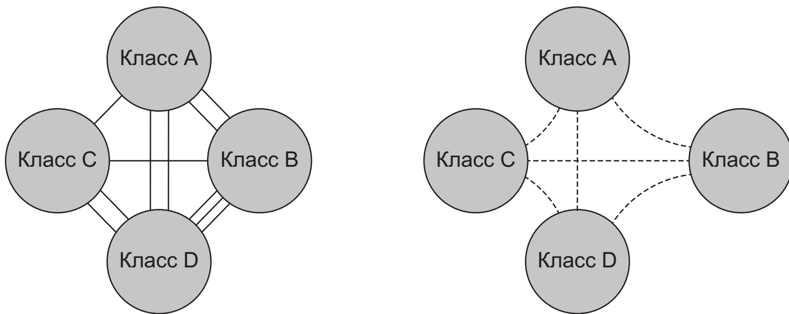


Рис. 10.1. Сопряженность — это мера взаимосвязи четко различимых фрагментов ПО

Мне нравится эта аналогия, потому что она подразумевает, что тесная сопряженность не является изначально *плохой* во всех случаях. Суть в том, что появляется больше работы, когда вы хотите что-то перетасовать.

Из этой аналогии также вытекает, что сопряженность — это непрерывный процесс, а не двоичная величина «все или ничего».

Хотя сопряженность измеряется на протяжении всего процесса, *существуют* общепризнанные пути его проявления. Вы можете научиться распознавать их и уменьшать сопряженность в своих программах по своему усмотрению. Но сначала я хочу дать вам более точное определение понятий тесной и слабой сопряженности.

10.1.2. Тесная сопряженность

Сопряженность между двумя фрагментами кода (модулями, классами и т. д.) считается тесной, если эти фрагменты кода тесно взаимосвязаны. Но как выглядит эта взаимосвязь? Она создается:

- классом, который хранит другой объект в качестве атрибута;
- классом, методы которого вызывают функции из другого модуля;
- функцией или методом, который выполняет много процедурной работы с использованием методов из другого объекта.

Всякий раз, когда классу, методу или функции нужно нести в себе много информации о другом модуле или классе, это является тесной сопряженностью. Рассмотрите код в следующем листинге. Функция `display_book_info` должна знать разнообразные фрагменты информации, которые содержатся в экземпляре класса `Book`.

Листинг 10.1. Функция, тесно сопряженная с объектом

```
class Book:
    def __init__(self, title, subtitle, author):
        self.title = title
        self.subtitle = subtitle
        self.author = author

def display_book_info(book):
        print(f'{book.title}: {book.subtitle} by {book.author}')
```

Книга хранит несколько фрагментов информации в качестве атрибутов

Эта функция содержит в себе знание обо всех атрибутах книги

Если класс `Book` и функция `display_book_info` находятся в одном модуле, то этот код может быть допустимым. Он работает на связанной информации,

и все это делается в одном месте. Но по мере роста кодовой базы вы рано или поздно обнаружите, что такие функции, как display_book_info, в одном модуле будут работать на классах из других модулей.

Тесная сопряженность — это не так уж плохо по своей сути. Иногда она просто пытается вам что-то сказать. Поскольку display_book_info работает только на информации из класса Book и делает что-то связанное с книгой, эта функция и этот класс имеют высокую связность. Она настолько *тесно сопряжена* с классом Book, что имеет смысл переместить ее в класс Book в качестве метода, как показано в следующем листинге.

Листинг 10.2. Уменьшение сопряженности за счет увеличения связности

```
class Book:
    def __init__(self, title, subtitle, author):
        self.title = title
        self.subtitle = subtitle
        self.author = author

def display_info(self):
    print(f'{self.title}: {self.subtitle} by {self.author}')
```


Как правило, тесная сопряженность становится проблемой, когда появляется между двумя разделенными зонами ответственности и выявляет неверно структурированную связность.

Возможно, вы встречали или писали код, похожий на код из листинга 10.3. Представьте поисковый индекс, в который пользователи могут направлять запросы. Модуль search предоставляет функциональность для предварительной очистки запросов, чтобы они производили из индекса согласующиеся результаты. Вы пишете главную процедуру, которая получает запрос от пользователя, очищает его и выводит очищенную версию.

Является ли главная процедура тесно сопряженной с модулем search?

1. Нет, потому что легко могла бы сделать эту работу сама.
2. Да, потому что вызывает некоторые функции внутри модуля search.
3. Да, потому что должна измениться, если изменится процесс очистки запросов.

Листинг 10.3. Процедура, тесно сопряженная с деталями класса

```
import re


def remove_spaces(query):                  ◄── Преобразовывает ' Джордж Вашингтон '
    query = query.strip()                      в 'Джордж Вашингтон'
    query = re.sub(r'\s+', ' ', query)
    return query


def normalize(query):                      ◄── Преобразовывает 'Universitätsstraße'
    query = query.casefold()                   (Университетская улица) в 'universitätsstrasse'
    return query


if __name__ == '__main__':                                          Получает запрос
    search_query = input('Введите поисковый запрос: ')        ◄──   от пользователя
    search_query = remove_spaces(search_query)
    search_query = normalize(search_query)
    print(f'Выполняется поиск "{search_query}"')   ◄── Выводит очищенный запрос

Удаляет пробелы и нормализует регистр букв
```

Вы можете правильно выявить сопряженность, оценив вероятность того, что любое изменение в модуле потребует изменений в коде, который его использует (вариант 3). Хотя главная процедура *может* выполнять ту же работу, что и функции очистки, важно обсудить сопряженность в том виде, в каком она существует в настоящее время. Вариант 1 является гипотетическим и недостоверным. Вызов нескольких функций из модуля (вариант 2) иногда является признаком сопряженности, но более важной метрикой остается степень вероятности того, что изменение в модуле `search` потребует внесения изменений в главную процедуру.

Предположим, что пользователи получают несогласующиеся результаты от незначительных изменений в своих запросах. Вы выяснили, что некоторые пользователи любят ставить кавычки вокруг запросов, думая, что это сделает запросы конкретнее, но ваш поисковый индекс обрабатывает кавычки буквально. Вы решаете отбросить кавычки перед выполнением запроса.

Сейчас код написан таким образом, что потребует добавления новой функции в модуль `search` *и* обновления всех мест, где вы очищаете

запросы, чтобы они вызывали новую функцию, как показано в следующем листинге. Все эти точки в коде тесно сопряжены с модулем `search`.

Листинг 10.4. Тесная сопряженность, приводящая к наружным изменениям

```
def remove_quotes(query):  ◄── Новая функция для удаления кавычек
    query = re.sub(r'"', '', query)
    return query


if __name__ == '__main__':
    ...
    search_query = remove_quotes(search_query)◄── Вызывает новую функцию везде, где вы нормализуете запросы
    ...
```

Читайте дальше, чтобы понять, что такое слабая сопряженность и чем она помогает в подобных ситуациях.

10.1.3. Слабая сопряженность

Слабая сопряженность — это способность двух фрагментов кода взаимодействовать ради выполнения задачи, не опираясь в значительной степени на детали друг друга. Она часто достигается за счет использования совместных абстракций. Вы узнали об интерфейсах из предыдущих глав и использовали совместную абстракцию в приложении Bark для реализации паттерна Команда.

Слабо сопряженный код реализует и использует интерфейсы, по крайней мере для обобщения. Динамическая типизация Python позволяет немного расслабиться, но здесь есть философия, которую я бы очень хотел для вас подчеркнуть.

Если вы начнете думать об общении между частями кода в терминах *сообщений*, которые объекты посылают друг другу (рис. 10.2), не сосредоточиваясь на самих объектах, то начнете выявлять более чистые абстракции и более сильную связность. Что такое сообщения? Это вопросы или приказы, направленные к объекту.

Еще раз взгляните на главную процедуру очистителя запросов в следующем листинге. Вы достигаете каждого преобразования, вызывая функции

для получения нового запроса. Каждый из них — это сообщение, которое вы посылаете.

Листинг 10.5. Вызов функций из модуля

```
if __name__ == '__main__':
    search_query = input('Введите ваш поисковый запрос: ')
    search_query = remove_spaces(search_query)
    search_query = remove_quotes(search_query)
    search_query = normalize(search_query)
    print(f'Выполняется поиск "{search_query}"')
```

Дает команду модулю search удалить пробелы

Дает команду модулю search удалить кавычки

Дает команду модулю search нормализовать регистр букв

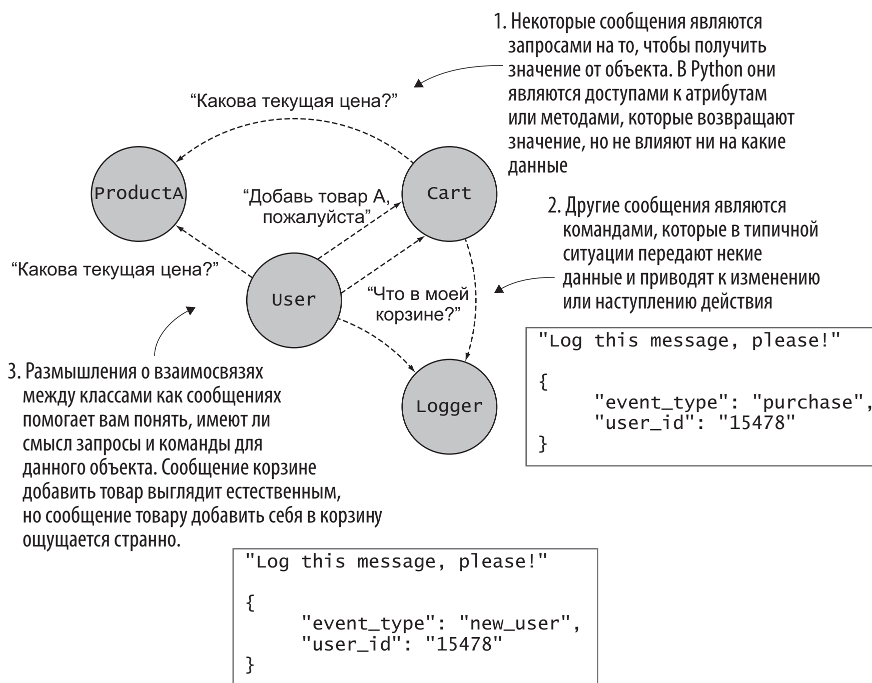


Рис. 10.2. Воображаемые взаимосвязи между классами в форме сообщений, которыми они обмениваются

То, что вы написали, решает задачу — очищает запрос, — но как вы воспринимаете эти сообщения? Не кажется ли вам, что вызов разнообразных функций из модуля `search` похож на прыжки через многочисленные обручи?

Если бы я увидел этот код, то мог бы сказать: «Я просто хочу очистить запрос, и мне все равно как!» Утомительно проходить через все шажки вызова каждой функции, особенно если вы очищаете запросы по всему коду.

Подумайте об этом с точки зрения сообщения или сообщений, которые вы *хотели бы* отправить. Более чистым подходом будет отправка одного-единственного сообщения: «Вот мой запрос; очисти его, пожалуйста». Какой подход вы использовали бы для достижения этой цели?

1. Совместить функции очистки запросов в единую функцию для удаления пробелов и кавычек и нормализации регистра букв.
2. Обернуть существующие вызовы функций в еще одну функцию, которую вы можете вызывать в любом месте.
3. Использовать класс для инкапсуляции логики очистки запросов.

Любой из этих вариантов мог бы сработать. Поскольку мы стремимся к разделению ответственности, вариант 1 может оказаться не лучшим. Перенос существующих функций в другую оставил бы обязанности раздельными, обеспечивая при этом единую точку входа для поведения очистки, что хорошо. Инкапсуляция этой логики далее в класс может иметь смысл позже, когда понадобится логика очистки для поддержания информации между шагами.

Попробуйте провести рефакторинг модуля `search`, чтобы сделать каждую функцию преобразования приватной, выставив наружу функцию `clean_query` (запрос), которая выполняет всю очистку и возвращает очищенный запрос. Вернитесь и сверьте свою работу со следующим листингом.

Листинг 10.6. Упрощение совместного интерфейса

```
import re


def _remove_spaces(query):   ◄── Преобразования сделаны приватными,
                                 потому что обусловливают детали очистки
    query = query.strip()
    query = re.sub(r'\s+', ' ', query)
    return query


def _normalize(query):
    query = query.casefold()
    return query
```

```
def _remove_quotes(query):
    query = re.sub(r'"', '', query)
    return query


def clean_query(query):
    query = _remove_spaces(query)
    query = _remove_quotes(query)
    query = _normalize(query)
    return query


if __name__ == '__main__':
    search_query = input('Введите свой поисковый запрос: ')
    search_query = clean_query(search_query)
    print(f'Выполняется поиск "{search_query}"')
```

Единая точка входа получает исходный запрос, очищает его и возвращает

Потребляющий код должен теперь вызывать только одну-единственную функцию, уменьшая сопряженность

Теперь, когда вы подумаете еще об одном способе очистки запросов, вы сможете сделать следующее (рис. 10.3):

1) создать функцию для выполнения нового преобразования на запросе;
2) вызвать новую функцию внутри `clean_query`;
3) поставить на этом точку и быть уверенными в том, что все потребители очищают запросы надлежащим образом.

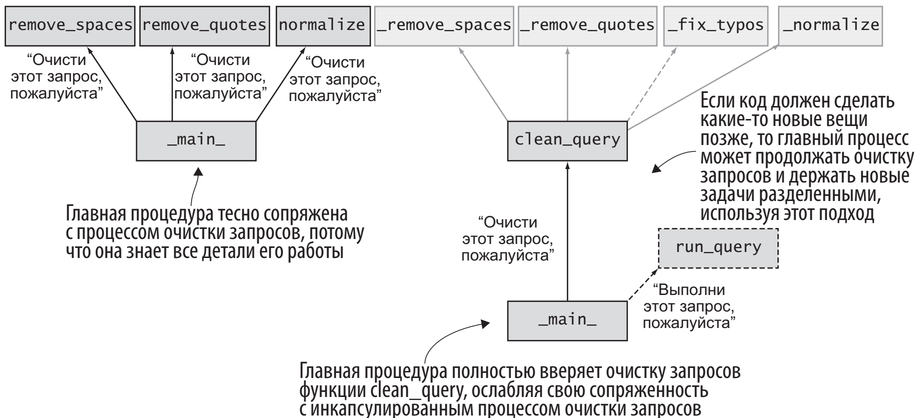


Рис. 10.3. Использование инкапсуляции и разделения ответственности для поддержания слабой сопряженности

Вы видите, что слабая сопряженность, разделение ответственности и инкапсуляция работают вместе. Разделение и инкапсуляция поведения с тщательно продуманным интерфейсом для внешнего мира помогают достичь слабой сопряженности.

10.2. РАСПОЗНАВАНИЕ СОПРЯЖЕННОСТИ

Вы уже видели примеры тесной и слабой сопряженности, но на практике сопряженность может принимать несколько специфических форм. Дав имя этим формам и распознав признаки каждой формы, вы обеспечите себе возможность ослабить тесную сопряженность на ранней стадии, поддерживая при этом продуктивность в долгосрочной перспективе.

10.2.1. Излишняя зависимость

В ранней версии кода очистки запросов посетителю требовалось вызывать несколько функций из модуля `search`. Когда код выполняет несколько задач, используя данные из другого участка кода, принято говорить, что в коде есть *излишняя зависимость*, или, другими словами, присутствуют «завистливые функции». Главная процедура тяготеет к превращению в модуль `search`, потому что явным образом использует все его средства. Это часто встречается и в классах (рис. 10.4).

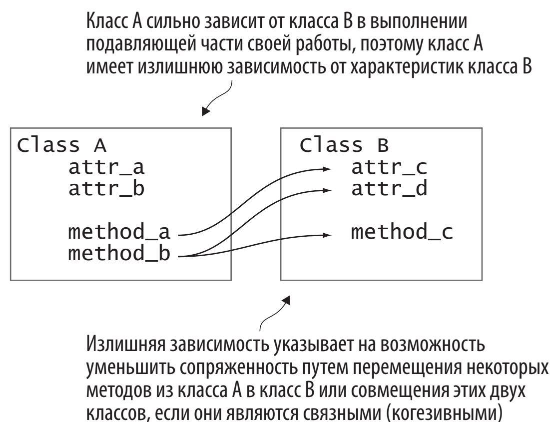


Рис. 10.4. Зависимость одного класса от данных другого

Решить проблему излишней зависимости можно тем же путем, каким вы исправили логику очистки запросов: свернуть ее в единую точку входа в источнике. В предыдущем примере вы создали функцию `clean_query` в модуле `search`. Модуль `search` — это то место, куда идет логика очистки запросов, так что функция `clean_query` там вполне уместна. Другой код может продолжать использовать `clean_query`, не осознавая, что происходит под ним, и полагая, что он получит в ответ правильно очищенный запрос. Этот код больше не имеет завистливых функций, что позволяет модулю `search` руководить поиском.

Когда вы выполняете рефакторинг, чтобы удалить излишнюю зависимость, вам будет казаться, что вы отказываетесь от некоторого контроля. До рефакторинга вы точно видите то, как информация течет по коду, но после него она скрывается под слоем абстракции. Это требует некоторого доверия к коду, с которым вы взаимодействуете, чтобы делать то, что он говорит. Иногда это будет неудобно, но тщательный тестовый набор поможет вам сохранить уверенность.

10.2.2. Стрельба дробью

Вы узнали о хирургии дробовика из главы 7 как о последствии тесной сопряженности, которое выглядит как мелкая дробь необходимых изменений по всему коду, вызванная однократным изменением в классе или модуле.

Решая проблему излишней зависимости, разделяя обязанности и практикуя хорошую инкапсуляцию и абстракцию, вы сведете к минимуму необходимый объем стрельбы дробью. Всякий раз, когда вы обнаружите, что переходите к разным функциям, методам или модулям, чтобы реализовать изменения, спросите себя, чувствуете ли вы тесную сопряженность между этими участками кода. Затем поищите возможности для перемещения метода в более подходящий класс, функции — в более подходящий модуль и т. д. — всему есть свое место.

10.2.3. Дырявая абстракция

Цель абстракции, как вы уже поняли, состоит в том, чтобы скрыть детали конкретной задачи от потребителя. Потребитель запускает поведение

и получает результат, но не заботится о том, что происходит под капотом. Если вы начинаете замечать излишнюю зависимость, то это может быть обусловлено *дырявой абстракцией*.

Дырявая абстракция — это абстракция, которая недостаточно скрывает свои детали. Абстракция претендует на звание простого способа сделать что-то, но в конечном счете она требует от вас понимания, что под ней скрыто, когда вы ее используете. Иногда она проявляет излишнюю зависимость, но также может быть незаметной.

Представьте себе пакет Python для выполнения HTTP-запросов (возможно, `requests`). Если ваша цель состоит исключительно в том, чтобы сделать запрос GET к некоторому URL-адресу и получить назад отклик, то самую лучшую службу вам сослужит абстракция на поведении HTTP-метода GET, например `requests.get('https://www.google.com')`.

Эта абстракция работает хорошо в *большинстве* случаев. Но что происходит, когда пропадает подключение к интернету, Google недоступен, все вдруг выглядит «как-то странно», а GET-запрос никуда не уходит? В этих случаях пакет `requests` обычно возбуждает исключение, указывающее на проблему (рис. 10.5). Это полезно для обработки ошибок, но требует, чтобы вызывающий код знал хоть немного о *возможных* ошибках, чтобы их обработать. Процесс обработки ошибок из пакета `requests` в нескольких местах происходит в условиях тесной сопряженности, потому что код ожидает некоторый набор возможных исходов, специфичных для пакета `requests`.

Утечки информации происходят, если не учтен компромисс с абстракциями: чем дальше вы абстрагируете концепцию в коде, тем меньше у вас возможностей для его настройки под свои нужды. Это происходит потому, что абстракция изначально предназначена для удаления доступа к деталям: чем меньше деталей вы можете получить, тем меньше способов существует для их изменения. Как разработчики, мы часто хотим подправить что-то, чтобы оно лучше соответствовало нашим потребностям, поэтому иногда предоставляем доступ более низкого уровня к тем самым деталям, которые пытались скрыть.

Предоставляя доступ к низкоуровневым деталям из высокоуровневого слоя абстракции, вы вводите сопряженность. Помните, что слабая

сопряженность опирается на *интерфейсы* — совместные абстракции, а не на конкретные низкоуровневые детали. Читайте дальше, чтобы увидеть несколько конкретных стратегий достижения слабой сопряженности в коде.

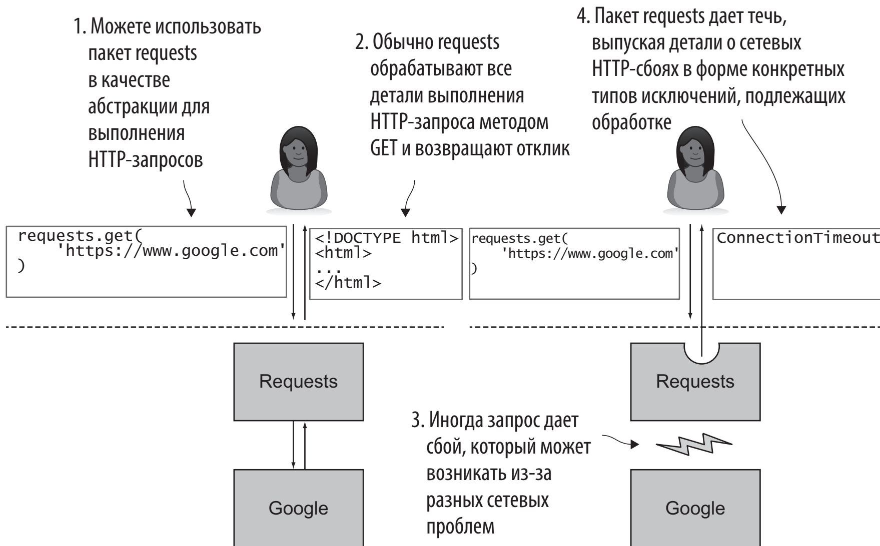


Рис. 10.5. Абстракции иногда дают течь, выпуская те детали, которые они пытаются скрыть

10.3. СОПРЯЖЕННОСТЬ В ПРИЛОЖЕНИИ BARK

Вы можете разделять обязанности и инкапсулировать поведение сколько угодно, но эти обязанности неизбежно должны взаимодействовать друг с другом. Сопряженность является необходимой частью разработки ПО, но она не обязательно должна быть *тесной*. Теперь, когда вы знакомы с некоторыми признаками тесной сопряженности, пришло время рассмотреть технические приемы ее уменьшения, сохраняя при этом код в рабочем состоянии. Некоторые из них вам знакомы, и вы увидите, как применить их к приложению Bark.

Вспомните многослойную архитектуру, которую вы использовали для Bark, приведенную снова на рис. 10.6. Каждый слой имеет четко различимый набор зон ответственности:

- слой визуализации показывает информацию пользователю и получает информацию от него;
- слой бизнес-логики содержит «мозги» приложения — логику, связанную с решаемой задачей;
- слой постоянства данных хранит данные для приложения, которые впоследствии будут многократно использоваться.

Вы подключили слой визуализации к слою бизнес-логики с помощью паттерна Команда. Каждый вариант действий в меню запускает соответствующую команду в бизнес-логике с помощью метода `execute` этой команды. Набор команд с их совместной абстракцией `execute` является хорошим примером слабой сопряженности.

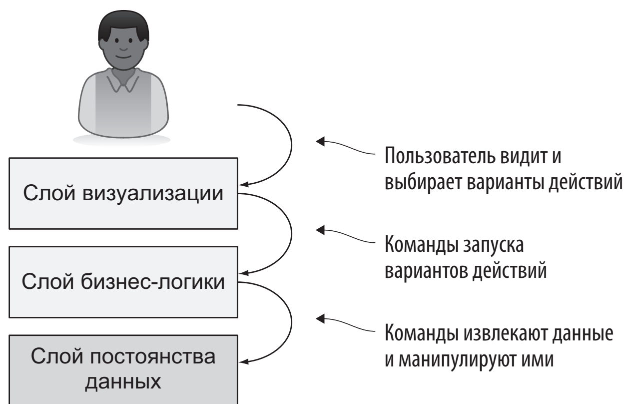


Рис. 10.6. Разделение ответственности в многослойную архитектуру

Слой визуализации очень мало что знает о командах, к которым он подключен, и команды не интересуются тем, почему были запущены, коль скоро они получают данные, которые ожидают. Это позволяет каждому слою изменяться самостоятельно, адаптируясь к новым требованиям.

Теперь подумайте, как слой бизнес-логики взаимодействует со слоем постоянства данных. Вспомните созданную вами команду `AddBookmarkCommand`, показанную в листинге 10.7. Эта команда выполняет следующие действия:

1) получает данные для закладки вместе с дополнительной меткой времени;
2) генерирует метку времени, если это необходимо;
3) предлагает слою постоянства данных сохранить закладку;
4) возвращает сообщение о том, что добавление прошло успешно.

Листинг 10.7. Команда для добавления новой закладки

```
class AddBookmarkCommand(Command):
    def execute(self, data, timestamp=None):    ◄── Получает данные закладок
        data['date_added'] = timestamp or datetime.utcnow().isoformat()
        db.add('bookmarks', data)
        return 'Закладка добавлена!'
```


А что, если я скажу, что здесь есть некоторая тесная сопряженность? Весь класс имеет пять строк — и вы, возможно, спросите: «И сколько же сопряженности может быть в пяти строках?». Как оказалось, последние две строки метода `execute` демонстрируют признаки тесной сопряженности.

Первая нарушающая строка, которая вызывает `db.add`, демонстрирует тесную сопряженность не только со слоем постоянства данных, но и с самой БД. Другими словами, если вы захотели бы в будущем хранить свои закладки не в БД — например, в файле JSON, — то `db.add` уже не очень хорошо подходит. Существует также некоторая излишняя зависимость — большинство команд напрямую пользуются одной операцией из класса `DatabaseManager`.

Второй строкой, представляющей сопряженность, является инструкция `return`. Каково ее предназначение? Она возвращает сообщение о том, что добавление прошло успешно. Это сообщение предназначено для пользователя. Обработка фрагмента информации слоя визуализации в слое бизнес-логики является примером дырявой абстракции. Слой визуализации должен руководить тем, что показывается пользователям. Некоторые другие команды, которые вы написали, имеют такую же структуру, которую вы вскоре исправите.

Другая команда, `CreateBookmarksTableCommand`, вводит еще более тесную сопряженность. Слово `Table` в ее названии подразумевает наличие БД и объекта слоя постоянства данных. Во время запуска приложения в слое визуализации происходит обращение к этой команде, которая охватывает все слои абстракции, так тщательно вами построенные! Не переживайте, вы вскоре сможете убрать и это.

Читайте дальше, чтобы узнать, как эта сопряженность может стать причиной проблем в реальности и как решать подобные проблемы.

10.4. РЕШЕНИЕ ПРОБЛЕМЫ СОПРЯЖЕННОСТИ

Представьте себе теперь, что вам нужно перенести приложение Bark на мобильные устройства. (И представьте телефоны, которые работают на Python!) Вы хотели бы использовать как можно больше кода приложения Bark многократно, чтобы оптимизировать работу пользователей на их телефонах, сохраняя при этом существующий интерфейс командной строки (рис. 10.7).

Рис. 10.7. Как стержневая бизнес-логика поддерживает разные варианты использования

Обращение к новым требованиям часто вскрывает тесно сопряженные участки кода. Новые варианты использования требуют, чтобы вы поменяли поведение и неизбежно раскрыли точки кода, в которых отсутствует гибкость. Что вы найдете в Bark?

10.4.1. Передача сообщений пользователям

Поскольку мобильные приложения обычно фокусируются на визуальных и тактильных элементах, наверняка в дополнение к сообщениям вы захотите использовать иконки, которые будут указывать на успех. Передача сообщений в Bark тесно сопряжена со слоем бизнес-логики. Чтобы устранить это ограничение, необходимо полностью освободить контроль над передачей сообщений для слоя визуализации. Как поддерживать взаимодействие между командами и слоем визуализации, если команды не знают о сообщениях, которые показывают?

Обратите внимание, что исходом одних команд является сообщение об успешности, а других — результат (например, список закладок). В слое визуализации вы можете разделить понятия «успех» и «результат», и каждая команда будет возвращать кортеж, представляющий статус и результат *одновременно*.

Все построенные команды должны исполняться успешно, поэтому на данный момент состояние каждой команды — `True`. Рано или поздно появятся команды, которые возвратят `False`, если не сработают. Команды, которые сейчас возвращают результат, смогут идти тем же путем, а команды без результата будут возвращать `None`.

Обновите каждую вашу команду так, чтобы та возвращала кортеж `status, result`. Вам также потребуется обновить класс `Option` в слое визуализации, чтобы учесть новое поведение при возврате. Что дальше?

1. Снабдить класс `Option` возможностью печати разных сообщений об успешности в зависимости от исполняемой команды.
2. Сконфигурировать каждый экземпляр класса `Option` с конкретным сообщением для использования при успешном выполнении команды.

3. Подклассировать класс `Option` для каждого вида сообщения, которое вы хотите показывать.

Вариант 1 мог бы сработать, но каждая новая команда будет добавлять к условной логике свой вариант того, какое сообщение показывать. Вариант 3 также мог бы сработать, но наследование должно быть ограниченным, а в данном случае не ясно, чем оправдано создание подклассов. Вариант 2 дает нужный объем настроек без больших дополнительных усилий. Помните, что приложение Bark должно продолжать функционировать так же по мере того, как вы рефакторизуете передачу сообщений, — вы выполняете рефакторинг только для того, чтобы упростить разработку.

Попробуйте сами и сверьтесь со следующими двумя листингами либо просмотрите полный исходный код этой главы (см. https://github.com/daneah/practices-of-the-python-pro).

Листинг 10.8. Отсоединение слоев абстракции с помощью интерфейсов

```
class AddBookmarkCommand(Command):
    def execute(self, data, timestamp=None):
        data['date_added'] = timestamp or datetime.utcnow().isoformat()
        db.add('bookmarks', data)
        return True, None

class ListBookmarksCommand(Command):
    def __init__(self, order_by='date_added'):
        self.order_by = order_by

    def execute(self, data=None):
        return True, db.select('bookmarks', order_by=self.order_by).
                    fetchall()
```

Команда AddBookmarkCommand выполняется успешно, но не возвращает результат

Возвращаемое значение равно статусу True и результату None

Команда ListBookmarksCommand выполняется успешно и возвращает список закладок

Возвращаемое значение равно статусу True и списку закладок

Листинг 10.9. Использование статусов и результатов в слое визуализации

```
def format_bookmark(bookmark):
    return '\t'.join(
        str(field) if field else ''
        for field in bookmark
    )
```

```
class Option:
    def __init__(self, name, command, prep_call=None,
➥ success_message='{result}'):
        self.name = name
        self.command = command
        self.prep_call = prep_call
        self.success_message = success_message

    def choose(self):
        data = self.prep_call() if self.prep_call else None
        success, result = self.command.execute(data)

        formatted_result = ''

        if isinstance(result, list):
            for bookmark in result:
                formatted_result += '\n' + format_bookmark(bookmark)
        else:
            formatted_result = result

        if success:
            print(self.success_message.format(result=
                                          formatted_result))

    def __str__(self):
        return self.name

def loop():
    ...

    options = OrderedDict({
        'A': Option(
            'Добавить закладку',
            commands.AddBookmarkCommand(),
            prep_call=get_new_bookmark_data,
            success_message='Закладка добавлена!',
        ),
        'B': Option(
            'Показать список закладок по дате',
            commands.ListBookmarksCommand(),
        ),
        'T': Option(
            'Показать список закладок по заголовку',
            commands.ListBookmarksCommand(order_by='title'),
        ),
        'E': Option(
            'Редактировать закладку',
            commands.EditBookmarkCommand(),
            prep_call=get_new_bookmark_info,
```

Сообщение по умолчанию для команд, которые возвращают результат, является самим результатом

Сохраняет сконфигурированное сообщение об успешности для этого варианта действий в целях последующего использования

Получает статус и результат из выполненной команды

Форматирует результат для показа, если это необходимо

Выводит сообщение об успешности, вставляя отформатированный результат, если это необходимо

Варианты действий без результата могут указывать статическое сообщение об успешности

Варианты действий, которые должны выводить только результат, не должны задавать сообщение

```
        ),
        'D': Option(
            'Удалить закладку',
            commands.DeleteBookmarkCommand(),
            prep_call=get_bookmark_id_for_deletion,
            success_message='Закладка удалена!',
        ),
        'G': Option(
            'Импортировать звезды GitHub',
            commands.ImportGitHubStarsCommand(),
            prep_call=get_github_import_options,
            success_message='Импортировано {result} закладок
                             из помеченных звездами репо!',
        ),
        'Q': Option(
            'Выйти',
            commands.QuitCommand()
        ),
    })
```

Варианты действий, которые имеют результат и специализированное сообщение, могут помещать оба элемента вместе

Примите мои поздравления! Вы разъединили слои бизнес-логики и визуализации. Теперь они взаимодействуют, используя идею статуса и результата вместо жестко закодированного сообщения. В будущем, когда вы создадите новый мобильный интерфейс для приложения Bark, он может использовать статусы и результаты для определения иконок и сообщений, которые будут показываться на телефонах.

10.4.2. Постоянство хранения закладок

Ваши мобильные пользователи всегда находятся в движении, и поэтому важно, чтобы они имели доступ к своим закладкам из любого места. БД должна жить в облаке за фасадом API, чтобы пользователи видели закладки на любом из своих устройств.

Некоторые участки командного кода специфичны для локальных операций с БД. Замените модуль `database` на новый слой постоянства данных, который взаимодействует с новым API. Помните, что совместные абстракции представляют собой верный путь к уменьшению сопряженности. Хотя это может показаться чересчур сложной задачей, поиск сходств и отличий между локальной БД и API поможет вам концептуализировать абстракцию, которая справится и с тем и с другим (рис. 10.8).

База данных	API
Данные представлены как объекты-записи	Данные представлены как объекты-записи
Операции CRUD с SQL (INSERT, SELECT, UPDATE, DELETE)	Операции CRUD с SQL (POST, GET, PUT, DELETE)
Конфигурация нужна для файлов и таблиц базы данных	Конфигурация нужна для домена API и URL-адресов

Рис. 10.8. БДи API имеют несколько общих характеристик

Слои постоянства — как БД, так и API — должны иметь дело с аналогичной ответственностью. Вот где абстракция видна во всей красе. Точно так же как вы сократили каждую свою команду до интерфейса `execute`, который возвращает статус и результат, отсоединив их от слоя визуализации, уменьшите слой постоянства данных до более общего набора операций CRUD, отсоединив его от команд. Тогда любой новый слой постоянства, который вы создадите, сможет использовать ту же абстракцию.

10.4.3. Попробуйте сами

У вас есть инструменты и знания, необходимые для того, чтобы отсоединить команды от класса `DatabaseManager`.

Используя абстрактный базовый класс `PersistenceLayer`, чтобы определить интерфейс, создайте слой постоянства `BookmarkDatabase`, который будет находиться между командами и классом `DatabaseManager` (рис. 10.9).

Создайте эти классы в новом модуле постоянства данных. Вы сможете рефакторизовать свои команды, чтобы использовать его вместо класса `DatabaseManager` непосредственно. Вместо специфичных для БД или API имен методов интерфейс должен предоставлять методы, которые будут применяться практически к любому слою постоянства данных:

- `__init__` для первоначальной конфигурации;
- `create(data)` для создания новой закладки;

- `list(order_by)` для показа всех закладок;
- `edit(bookmark_id, data)` для обновления закладки;
- `delete(bookmark_id)` для удаления закладки.

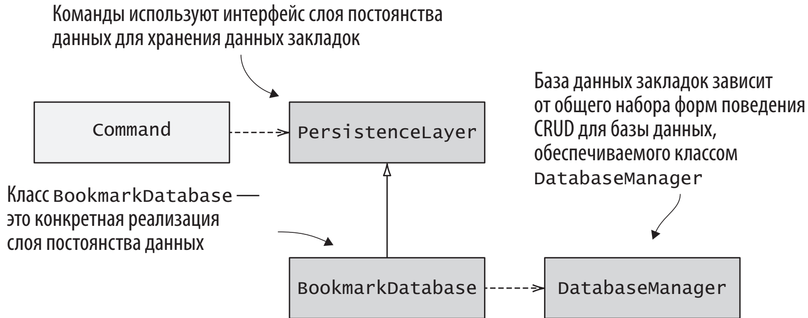


Рис. 10.9. Отсоединение команд от специфики БД с помощью интерфейса и конкретной реализации

Логика в `CreateBookmarksTableCommand` в действительности является первоначальной конфигурацией для слоя постоянства БД закладок, поэтому вы можете переместить ее в `BookmarksDatabase.__init__`. Создание экземпляра класса `DatabaseManager` тоже хорошо для этого подходит. Затем вы можете написать реализацию для каждого метода абстракции `PersistenceLayer` в классе `BookmarksDatabase`. Каждый вызов метода, ориентированного на БД (`db.add`, например), из исходных команд можно переместить в надлежащий метод, освободив команды для вызова методов из класса `BookmarksDatabase`. Сделайте попытку, ссылаясь на следующий листинг и полный исходный код для этой главы.

Листинг 10.10. Интерфейс постоянства данных и его реализация

```
from abc import ABC, abstractmethod

from database import DatabaseManager


class PersistenceLayer(ABC):
    @abstractmethod
    def create(self, data):
        raise NotImplementedError('Слои постоянства данных должны
➥ реализовать метод create')
```

Абстрактный базовый класс, который определяет интерфейс слоя постоянства данных

Каждый метод соответствует CRUD-подобной операции для поддержки постоянства данных

```
    @abstractmethod
    def list(self, order_by=None):
        raise NotImplementedError('Слои постоянства данных должны
➥ реализовать метод list')

    @abstractmethod
    def edit(self, bookmark_id, bookmark_data):
        raise NotImplementedError('Слои постоянства данных должны
➥ реализовать метод edit')

    @abstractmethod
    def delete(self, bookmark_id):
        raise NotImplementedError('Слои постоянства данных должны
➥ реализовать метод delete')


class BookmarkDatabase(PersistenceLayer):
    def __init__(self):
        self.table_name = 'bookmarks'
        self.db = DatabaseManager('bookmarks.db')

        self.db.create_table(self.table_name, {
            'id': 'integer primary key autoincrement',
            'title': 'text not null',
            'url': 'text not null',
            'notes': 'text',
            'date_added': 'text not null',
        })

    def create(self, bookmark_data):
        self.db.add(self.table_name, bookmark_data)

    def list(self, order_by=None):
        return self.db.select(self.table_name,
               order_by=order_by).fetchall()

    def edit(self, bookmark_id, bookmark_data):
        self.db.update(self.table_name, {'id': bookmark_id},
                bookmark_data)

    def delete(self, bookmark_id):
        self.db.delete(self.table_name, {'id': bookmark_id})
```

Специфичная для слоя постоянства реализация, которая использует БД

Обрабатывает создание БД с помощью класса DatabaseManager

Специфичная для БД реализация для каждой формы поведения интерфейса

Теперь, когда у вас есть интерфейс для слоя постоянства данных и конкретная реализация этого интерфейса, знающая, как использовать класс `DatabaseManager` для сохранения закладок, вы готовы обновить команды, чтобы они зависели от интерфейса `PersistenceLayer`

вместо DatabaseManager. В модуле commands замените экземпляр db класса DatabaseManager на persistence — экземпляр класса BookmarkDatabase. Затем пройдитесь по остальной части модуля, заменив вызовы методов класса DatabaseManager (например, db.select) методами из класса PersistenceLayer (например, persistence.list). Обратитесь к следующему листингу, чтобы проверить себя.

Листинг 10.11. Обновление бизнес-логики для использования абстракции

```
from persistence import BookmarkDatabase   ◄ Импортирует класс BookmarksDatabase вместо класса DatabaseManager

persistence = BookmarkDatabase()   ◄ Настраивает слой постоянства (он может быть замещен в будущем)

class AddBookmarkCommand(Command):
    def execute(self, data, timestamp=None):
        data['date_added'] = timestamp or datetime.utcnow().isoformat()
        persistence.create(data)   ◄ persistence.create занимает место метода db.add
        return True, None


class ListBookmarksCommand(Command):
    def __init__(self, order_by='date_added'):
        self.order_by = order_by

    def execute(self, data=None):
        return True, persistence.list(order_by=self.order_by)   ◄ persistence.list занимает место метода db.select

class DeleteBookmarkCommand(Command):
    def execute(self, data):
        persistence.delete(data)   ◄ persistence.delete занимает место метода db.delete
        return True, None


class EditBookmarkCommand(Command):
    def execute(self, data):
        persistence.edit(data['id'], data['update'])   ◄ persistence.edit занимает место метода db.update
        return True, None
```

Приложение Bark расширяется на новые варианты использования, например импорт звезд из GitHub. Его зоны ответственности хорошо разделены, благодаря чему вы можете рассуждать о визуализации, бизнес-логике и постоянстве данных изолированно. Теперь можно поменять

местами любой из этих слоев, чтобы реализовать четко различимые новые варианты использования.

Вы могли бы поменять БД `BookmarksDatabase`, скажем, на службу `BookmarksStorageService`, которая отправляет данные закладок через API HTTP в облако. Вы также можете подставить фиктивную БД `DummyBookmarksDatabase` для тестирования, которая сохраняет закладки в памяти только на время прохождения тестов. Слабая сопряженность изобилует возможностями! Настоятельно рекомендую вам изучить некоторые из них самостоятельно.

Принципы, которые вы применили к приложению Bark, легко переносятся на многие реальные проекты. Применяя то, что вы усвоили, вы сможете повысить сопроводимость, а также помочь другим людям подхватить ваш код и понять его суть. А это очень важно, когда вы проектируете ПО.

В последней части этой книги кратко резюмируем все то, чему вы научились, и рассмотрим некоторые рекомендации, что изучать дальше. Увидимся там!

ИТОГИ

- Разделите обязанности, выполните инкапсуляцию данных и поведения, а затем создайте совместные абстракции, чтобы ослабить сопряженность.
- Классы, которые знают и используют многочисленные детали другого класса, возможно, должны быть поглощены этим классом.
- Тесная сопряженность может быть устранена путем повторной инкапсуляции с более высокой связностью, но она может быть использована для введения новой абстракции, совместной для обеих сторон. (Например, меню и команда могут опираться на то, что команда возвращает статус и результат вместо конкретного сообщения.)

Часть IV

Что дальше?

Хотя мне и было приятно учить вас, в ограниченном пространстве книги удается охватить мало. В этой части поговорим о том, что следует изучать дальше. Здесь также содержится поверхностное введение в еще несколько концепций, которые помогут вам в дальнейшем писать первоклассный софт. Предложения по самоподготовке сгруппированы по темам, таким образом вы можете узнать о каждой теме на высоком уровне либо перейти к одной из них и изучить ее более подробно.

11 Только вперед

В этой главе:

- ✓ Выбор перспективных направлений для дальнейшего развития карьеры разработчика ПО.
- ✓ Разработка плана дальнейших действий.

Что ж, вы дошли до последней главы этой книги. Разве она не была увлекательной? Из этой книги вы узнали многие аспекты продуманного дизайна ПО, но предстоит еще открыть для себя целый мир. Иногда бывает трудно понять, что делать дальше. Если вы не уверены в том, какие траектории развития следует выбрать, то прочитайте эту главу для вдохновения и идей.

11.1. ЧТО ТЕПЕРЬ?

По мере накопления опыта вы продолжите многому учиться. Вы также столкнетесь с вещами, которым захотите научиться, но времени или опыта для изучения пока нет. Кроме того, есть просто бесконечный набор

вещей, о которых вы вообще не знаете. Это понятия, которые либо еще не пришли вам в голову, либо у вас нет слов, чтобы их выразить.

Бывший министр обороны США Дональд Рамсфелд лаконично (и с юмором) выразился примерно так:

> Есть известные известные — вещи, о которых мы знаем, что знаем их. Есть также известные неизвестные — вещи, о которых мы знаем, что не знаем. Но еще есть неизвестные неизвестные — это вещи, о которых мы не знаем, что не знаем их.
>
> *Дональд Рамсфелд*

Быть эффективным инженером — не значит быть исчерпывающе осведомленным в каком-либо предмете. Чаще всего вы способны эффективно работать, зная, что конкретно нужно искать и какие ресурсы вам доступны. Короче говоря, находчивость ценится больше, чем опыт.

По мере профессионального роста вы, у вас, скорее всего, будут копиться статьи, инструменты и темы, которые вас интересуют. Также придется усваивать какие-то новые концепции при написании кода. Рано или поздно, когда вы решите, что пришло время углубиться в какие-то темы, настроиться на успех поможет учебный план.

11.1.1. Разработайте план

Вы когда-нибудь спускались в кроличью нору Википедии? Начинаете читать статью на какую-то тему, и вдруг уже 2:37 утра, а у вас в браузере 37 открытых вкладок. Вы щелкаете по ссылкам, представляющим интерес, уходя в тему все глубже. Хотя вы и чувствуете, что впустую потратили вечер, такая стратегия оказывается эффективной для знакомства с информацией.

Интеллект-карта организует информацию в иерархическую структуру, которую можно воспринимать визуально. Она начинается с центральной *вершины* — совокупного понятия, которое вам интересно. Затем она разветвляется, и каждая вершина представляет подтемы или связанные с ними понятия для изучения — где-то среди них «космический латте»

на странице о «беж». Используя интеллект-карту для перечисления концепций, о которых хотите узнать, вы разработаете четкую картину разных областей, которые нужно охватить.

ФИЛОСОФСКАЯ ИГРА

Вы также можете пойти в обратном направлении — вверх — по Википедии. Начиная практически с любой статьи, нажатие на первую (а иногда и вторую) ссылку в первом полном абзаце каждой последующей статьи, скорее всего, приведет вас на страницу «философия». Это связано с тем, что первая ссылка обычно является одной из самых широких или общих ссылок. Попробуйте:

- Беж > французский > романский язык > вульгарная латынь > нестандартный > языковое разнообразие > социолингвистика > общество > группа > социальные науки > академические дисциплины > знание > факты > реальность > воображаемое > объект > философия
- Python (язык программирования) > интерпретируемый > язык программирования > формальный язык > математика > величина > множество > число > математический объект > абстрактный объект > философия

Если вы хотите изучить процесс обработки естественного языка, набросайте интеллект-карту как на рис. 11.1. Несколько категорий деятельности высокого уровня в итоге разветвляются на конкретные и связанные между собой темы, такие как лемматизация и марковские цепи. Запишите их, даже если уже многое о них слышали или не знаете, в какую ветвь они попадут, — потом вы доберетесь до них.

Такое визуальное представление помогает подчеркнуть связи между темами, что даст возможность удерживать информацию. Она подобна географической карте, на которой обозначены неисследованные территории, и очень удобна.

Рис. 11.1. Интеллект-карта для изучения обработки естественного языка

Если у вас нет достаточного опыта в какой-то области, чтобы начертить ее полную карту, не переживайте. Составление короткого списка тем по-прежнему остается эффективной мерой. Главное, чтобы у вас была запись, которая напоминала бы вам, что вы уже сделали и что осталось сделать.

Наметив следующие шаги, вы максимально подготовитесь к учебе.

11.1.2. Исполните план

После того как темы для изучения будут обозначены (или перечислены), можете собрать ресурсы: книги, онлайн-курсы, друзья или опытные коллеги. Определитесь также со своим стилем обучения. Некоторые могут учиться, просто читая, в то время как другим нужно писать код. Будьте изобретательными.

Интеллект-карта работает хорошо, потому что ее можно заполнять нелинейно. Если вы пока что знакомитесь с терминологией и понятиями, то сначала можно разведать понятия на одном уровне от центра, как показано на рис. 11.2. Это поможет оценить рельеф местности для дальнейшей заливки фундамента.

Рис. 11.2. Изучение широты темы

Когда вы выберете тему, особенно интересную для углубленного изучения (рис. 11.3), приток новой информации начнет наполнять вас энергией.

> **СОВЕТ**
>
> Распространенная ошибка — глубоко погрузиться в в одну тему, не имея достаточного контекста для ее дальнейшего изучения. Убедитесь, что вы держите баланс. Слишком много внимания к одному предмету может привести вас к закреплению неточного или неполного понимания, что воспрепятствует будущему обучению.

Успешное обучение требует итеративного подхода — по мере того как вы приобретаете больше опыта в теме, вы естественным образом будете находить все больше понятий, требующих добавления в интеллект-карту (или список). Добавлять что-то по ходу дела абсолютно нормально, но убедитесь, что вы уверенно себя чувствуете в ранее изученном, прежде чем переходить к новым темам. Очень легко взять на себя слишком много!

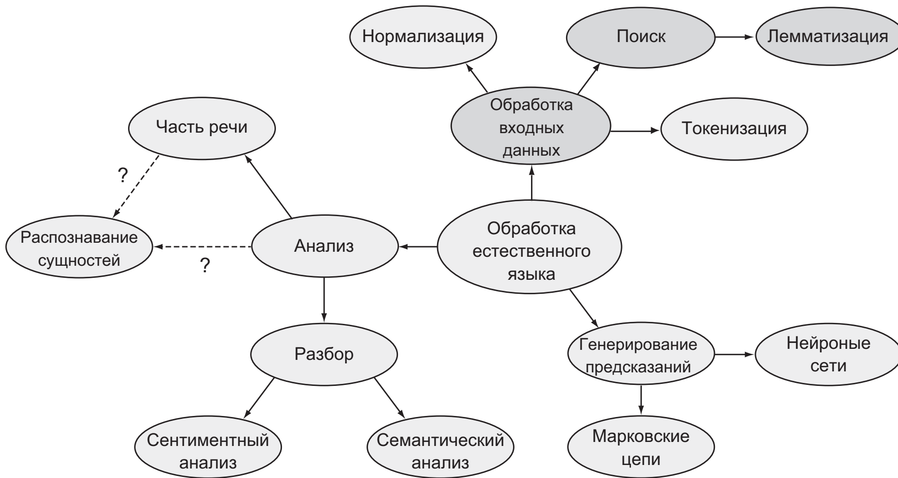


Рис. 11.3. Изучение темы в глубину

Держите все это в порядке, отслеживая свой прогресс.

11.1.3. Отслеживайте свой прогресс

Процесс обучения является субъективным, его невозможно «закончить». В изучении той или иной темы есть несколько четко различимых состояний:

1. *Желание или потребность учиться* — у вас есть список тем, которые нужно охватить, но вы еще не приступили к нему.
2. *Активная учеба* — вы нашли и прочитали некоторые ресурсы по этой теме и ищете больше.
3. *Осведомленность* — вы понимаете тему в целом, и у вас есть некоторое представление о том, как вы могли бы ее применить.
4. *Комфортное ощущение* — вы уже несколько раз применяли идеи из этой темы и уже разбираетесь в ней.
5. *Компетентность* — вы применяли идеи уже достаточно, чтобы знать некоторые нюансы, и знаете, к каким ресурсам следует обращаться, когда сталкиваетесь с новыми видами задач.

Эксперты выделяют еще больше состояний, каждое из которых представляет собой заметный сдвиг в вашем поведении. Неплохо представлять себе, на каком уровне вы находитесь, чтобы лучше понимать, в какие темы инвестировать свое время. Вполне возможно, что вы вообще *не захотите* достигать «компетентного» уровня для тем, с которыми сталкиваетесь редко или задачи которых не вдохновляют на поиск решения. Записывая это в явной форме, вы сможете поддерживать свой план в актуальном состоянии (рис. 11.4).

Рис. 11.4. Отслеживание прогресса в обучении по каждой теме

На каждом уровне обучения вы, вероятно, усвоите несколько важных моментов о теме. Они могут быть недостаточно большими, чтобы оправдать добавление новых вершин в интеллект-карту (хотя программы по составлению интеллект-карт делают это занятие менее хлопотным), но записать их все же можно. Вы можете использовать эти заметки, чтобы определить, на каком уровне изучили ту или иную тему, и они, возможно, побудят вас вернуться к идеям, требующим дополнительной работы.

Как в учебе, так и в работе мне стоило больших усилий, чтобы удерживать поступающую информацию даже после неоднократных повторений.

Составление карт и отслеживание прогресса оказались эффективным подспорьем в изучении многих идей, высказанных в этой книге и не только. Если вы еще не отслеживали свое обучение подобным образом, то попробуйте это сделать.

ПРОГРАММЫ ДЛЯ СОСТАВЛЕНИЯ ИНТЕЛЛЕКТ-КАРТ

ПО для составления интеллект-карт помогает создавать визуальные представления ваших мыслей и связей между ними. Простейшие интеллект-карты — это вершины с некоторым текстом, соединенные линиями. Существует несколько коммерческих инструментов, таких как Lucidchart (www.Lucidchart.com) и MindMup (www.mindmup.com), которые имеют более продвинутые функциональные средства, однако любая программа для построения диаграмм, такая как draw.io (https://draw.io), обеспечит необходимый минимум для начала работы. Попробуйте что-нибудь простое и бесплатное, пока не освоитесь с составлением карт.

Теперь, когда у вас в голове есть фреймворк для исследований, читайте дальше — я дам советы о том, что делать после чтения этой книги.

11.2. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

За последние несколько десятилетий разработчики неоднократно решали одни и те же задачи. Если посмотреть на все эти решения, то обнаруживаются некоторые паттерны: как со слабой сопряженностью, так и с тесной.

Указанные *паттерны проектирования* являются проверенными решениями, и их поименное перечисление позволит поговорить о них конкретнее. *Единый язык*, или совместный словарь понятий, которые команда должна понимать, развивается вместе с текущей работой команды.

Вы использовали паттерн проектирования, когда создавали команды для приложения Bark. Паттерн Команда, как известно, часто используется в подобных приложениях, чтобы отсоединить код, который запрашивает

действие, от самого действия. Паттерн Команда всегда имеет несколько часто встречающихся фрагментов, независимо от ситуации, в которой он используется:

1. *Получатель* — сущность, которая предпринимает действие, в частности осуществляет постоянное хранение данных в БД или выполняет вызов API.
2. *Команда* — сущность, содержащая информацию, необходимую получателю для того, чтобы предпринять действие.
3. *Активатор* — сущность, запускающая команду для того, чтобы оповестить получателя.
4. *Клиент* — сущность, которая собирает активаторов, команды и получателей вместе для того, чтобы выполнить задачу.

В приложении Bark эти фрагменты выглядят следующим образом:

1. *Классы* `PersistenceLayer` — это получатели. Они получают достаточно информации для хранения или извлечения данных (из БД в случае класса `BookmarkDatabase`).
2. *Классы* `Command` — это команды. Они хранят информацию, необходимую для взаимодействия со слоем постоянства данных.
3. *Экземпляры класса* `Option` — это активаторы. Они запускают команду на выполнение, когда пользователь выбирает в меню вариант действий.
4. *Клиентский модуль* — это клиент. Он надлежащим образом подключает варианты действий к командам, в результате чего выбор пользователей в меню приводит к желаемому действию.

Диаграмма на унифицированном языке моделирования (UML, unified modeling language) этих классов показана на рис. 11.5[1]. Диаграммы UML — это общепринятый способ изображения связей между сущностями в программе. В этой книге я намеренно почти не затрагивал UML, так как он усложнил бы понимание тем новичками. Однако когда вы узнаете о паттернах проектирования, вы увидите, что диаграммы UML возникают там часто. Помните, что паттерны сами по себе важны

[1] https://ru.wikipedia.org/wiki/UML

для понимания: если диаграммы UML для вас не работают, почитайте о них подробнее.

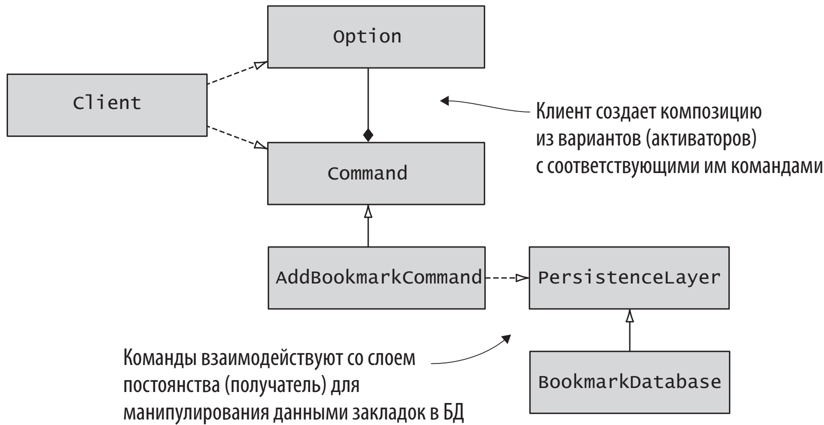


Рис. 11.5. Паттерн команд, используемый в приложении Bark

11.2.1. Взлеты и падения паттернов в Python

Вы уже видели некоторые выгоды от использования конкретного паттерна в Python. Паттерн Команда помог вам отсоединить слои абстракции в Bark, что привело к гибкому постоянству данных, бизнес-логике и визуализации. Многие другие паттерны, о которых вы узнаете, также представляют ценность.

Для того чтобы понять, какие паттерны вы должны усвоить и применять в Python, важно представлять себе контекст, в котором они разрабатывались и использовались, а именно язык или языки, из которых они возникли. Ряд паттернов происходит из Java, статически типизируемого языка, который не может создавать экземпляры классов и т. д. В результате многие паттерны являются *порождающими*. Динамическая типизация Python освобождает паттерн от ограничений, поэтому многие порождающие паттерны просто не нужны в Python.

Если вы пытаетесь использовать паттерн проектирования для решения задачи и он кажется вынужденным, не бойтесь отказаться от него. Тем временем на вас может наскочить более удачный паттерн.

Канонической книгой о паттернах является «Паттерны объектно-ориентированного проектирования»[1]. В онлайн-сообществе по разработке ПО также есть много дискуссий на эту тему, часто с полезными примерами, которые помогают лучше понять, когда и где использовать тот или иной паттерн.

11.2.2. С чего начать

Вы можете начать самостоятельное изучение паттернов со следующих:

- Порождающие паттерны.
- Фабрики.
- Поведенческие паттерны.
- Паттерн Команда.
- Структурные паттерны.
- Паттерн адаптеров.

11.3. РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

При разработке современных веб-приложений, возможно, вам понадобится сервер, обрабатывающий HTTP-трафик, БД для хранения данных, кэш для хранения часто используемых данных и т. д. Эти элементы образуют систему — группу взаимосвязанных частей, составляющих единое целое. Элементы этой системы часто располагаются на разных машинах, в отдельных центрах обработки данных, а иногда даже на разных континентах (рис. 11.6). Эти *распределенные* системы добавляют слои ценности, сложности и риска, которыми будет заниматься разработчик.

Некоторые наиболее интересные сложности распределенных систем проявляются при сбоях.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — 448 с.: ил. — *Примеч. ред.*

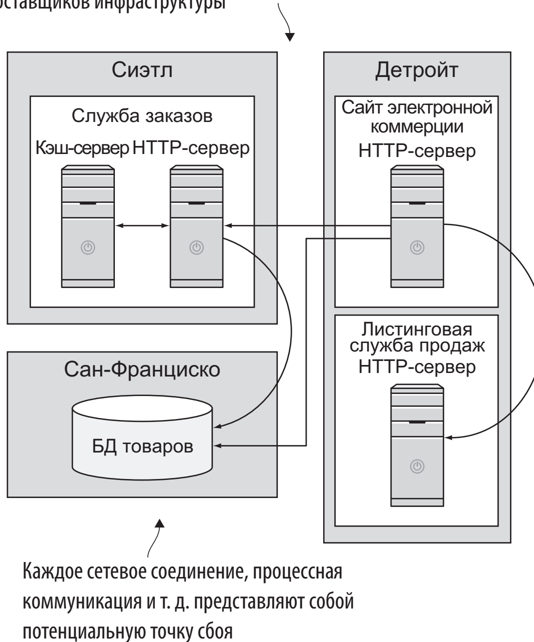


Рис. 11.6. Система распределена по нескольким местоположениям

11.3.1. Режимы сбоя в распределенных системах

Даже на одной машине программа может неожиданно завершиться аварийно. Другие программы, которые ожидают, что эта программа продолжит работу, также могут завершиться аварийно, если не учли эту ситуацию.

Отщипывание кусочков приложения для размещения их в новых местах приводит к новым, экзотическим режимам сбоя. Все приложения могут работать правильно, но сетевое соединение между ними может давать сбой. Или большинство приложений имеют доступ к БД, но один из них — нет. Технологии распределенных систем стремятся выдержать эти режимы отказа и восстановиться после сбоев.

Я обнаружил, что размышления об отказах распределенных систем похожи на размышления о функциональном тестировании. Из главы 5 вы узнали о творческом разведывательном тестировании как способе перечисления большого числа аспектов уязвимости. Распределенные системы требуют такого же образа мышления в большем масштабе, потому что там больше движущихся частей.

11.3.2. Обращение к состоянию приложения

Серьезный вопрос в распределенных системах — как справиться с аварийным сбоем части системы. Возможно, вы сможете жить без каких-то частей системы и данных, которые ими предоставлялись. Другие части системы могут быть необходимы, но не чувствительны ко времени, поэтому запросы к ним, пока они не работают, могут храниться и откладываться до восстановления. Оставшиеся части системы имеют решающее значение для операций — система без них останавливается. Это *одиночные точки сбоя*.

Распределенные системы спроектированы таким образом, чтобы свести к минимуму единичные точки отказа, что обеспечивает *плавную деградацию* — продолжение работы без какого-то действия или информации. Такие инструменты, как Kubernetes (https://kubernetes.io/), дополняют подход к обработке сбоев с помощью *окончательной согласованности*, которая позволяет определять состояние, требуемое вашей системе, обеспечивая гарантию того, что система рано или поздно достигнет определенного состояния. Сопряжение плавной деградации с окончательной согласованностью приводит к появлению гибких систем, которые выходят из строя реже.

Хотя распределенные системы не являются чем-то новым, в последнее время появилось много разработок в области их инструментария и философии. Платформа распределенных вычислений Kubernetes и ее экосистема могут успешно применяться к малым системам по мере усвоения, но она наиболее хороша в крупных, многосложных системах. Вы можете начать с принципов и методов, а затем получить некоторую практику построения нескольких распределенных систем, прежде чем перейти к специализированным инструментам.

11.3.3. С чего начать

Вы можете начать самостоятельное изучение распределенных систем со следующего:

- Отказоустойчивость.
- Окончательная согласованность (eventual consistency).
- Желаемое состояние.
- Конкурентность.
- Очередность обработки сообщений.

11.4. ПОГРУЖЕНИЕ В PYTHON

Кажется очевидным, но еще одна область, в которой вы можете продолжать расти, — это язык Python. Хотя в этой книге Python использовался в примерах для передачи идей о разработке ПО, из нее можно узнать еще массу интересного о функциях, синтаксисе и мощи этого языка.

11.4.1. Питоновский стиль

По мере того как вы будете все больше работать на языке Python, рано или поздно вы разовьете чутье в отношении форматирования кода. Вы будете писать свой код в этом стиле, потому что вам будет легче читать его потом. Но когда кто-то другой, кто следовал своему собственному стилю, будет читать ваш код, ему может быть трудно его понять. PEP 8 — руководство по написанию кода на Python — предлагает стандартный стиль форматирования кода Python, избавляя вас от необходимости тратить время на мучительные размышления над ним[1]. Такие инструменты, как Black (https://github.com/psf/black), помогают в форматировании кода. Это высвобождает ваше время, давая возможность думать о более серьезных проблемах, таких как более

[1] «PEP 8 Руководство по стилю кода Python» можно найти на веб-сайте Python: www.python.org/dev/peps/pep-0008/.

масштабный дизайн ПО и бизнес-потребности, пути решения которых вы пытаетесь нащупать.

11.4.2. Языковые средства являются паттернами

Паттерны проектирования традиционно обсуждаются в терминах объектов и взаимодействий между ними. Но паттерны также часто встречаются в том, каким образом те или иные идеи выражаются в синтаксисе Python. Вещи, которые многократно делаются некоторым образом на языке Python, потому что они изящны, кратки, четки или удобочитаемы, называются «питоновскими». Эти паттерны бывают так же важны для тех, кто пытается понять ваш код, как и паттерны проектирования.

Некоторые паттерны в Python предусматривают использование типов данных, присущих данной ситуации, например, использование `dict` для увязывания ключей со значениями. Некоторые паттерны подразумевают использование операции включения в список либо тернарных операторов с целью сокращения многострочных инструкций в пользу кратких. Очень важно знать, что именно у вас есть и когда пользоваться каждым паттерном. Важно также знать, когда *не* следует ими пользоваться.

Дзен языка Python предоставляет хороший набор общих принципов написания кода Python.

```
>>> import this
The Zen of Python, by Tim Peters

Красивое лучше, чем уродливое.
Явное лучше, чем неявное.
Простое лучше, чем сложное.
Сложное лучше, чем запутанное.
Плоское лучше, чем вложенное.
Разреженное лучше, чем плотное.
Читаемость имеет значение.
Особые случаи не настолько особые, чтобы нарушать правила.
При этом практичность важнее безупречности.
Ошибки никогда не должны замалчиваться.
Если не замалчиваются явно.
Встретив двусмысленность, отбрось искушение угадать.
```

```
Должен существовать один — и желательно только один — очевидный способ
сделать это.
Хотя он поначалу может быть и неочевиден, если вы не голландец.
Сейчас лучше, чем никогда.
Хотя никогда зачастую лучше, чем прямо сейчас.
Если реализацию сложно объяснить — идея плоха.
Если реализацию легко объяснить — идея, возможно, хороша.
Пространства имен — отличная вещь! Давайте будем делать их больше!
```

Если вы просматриваете эти рекомендации как простые заметки на полях, то можете критически взглянуть на те участки кода, которые вас раздражают или кажутся нелепыми. Видя корявый синтаксис, понимая его предназначение и пытаясь нагуглить «лучший способ сделать X в Python», вы сможете придумать альтернативные идеи его реализации. Еще одна тактика, которую я использовал для усвоения советов и трюков, заключается в следовании приемам известных пользователей Python — таких, как разработчики ядра Python, — в Twitter. Благодаря этому вы сможете находить на удивление полезную информацию.

Всестороннее руководство по языку, такие книги, как «Быстрый Python» Дэрила Хармса и Кеннета Макдональда (*Harms, D., McDonald, K., The Quick Python Book*, *Manning,* 1999)[1]; «Автостопом по Python» Кеннета Рейтца и Тани Шлюссер (*Reitz, K., Schlusser, T. The Hitchhiker's Guide to Python*, O'Reilly, 2016)[2]; https://docs.python-guide.org/) и «Книга рецептов Python» Дэвида Ашера и Алекса Мартелли (*Ascher, D., Martelli, A. Python Cookbook*, O'Reilly, 2002)[3] помогут вам погрузиться в этот язык.

11.4.3. С чего начать

Вы можете начать со следующих концепций, когда начнете погружаться в изучение примеров, паттернов и рекомендаций по питоновскому коду:

- Питоновский код.
    - Питоновский способ сделать X.

---

[1] Третье издание книги: *Седер Наоми*. Python. Экспресс-курс. 3-е изд. — СПб.: Питер, 2019. — 480 с.

[2] *Рейтц К., Шлюссер Т.* Автостопом по Python. — СПб.: Питер, 2017. — 336 с.

[3] Второе издание книги: *Бизли Д., Джонс Б. К.* Python. Книга рецептов. — М.: ДМК Пресс, 2019. — 648 с.

- Идиоматический Python.
- Питоновские антипаттерны.
- Питоновские статические анализаторы кода (линтеры).

11.5. ЧТО ВЫ УЗНАЛИ

Как автор, я могу только догадываться, что побудило вас взяться за эту книгу или какой опыт вы имели до этого. Но если вы все-таки читаете ее, то я, вероятно, могу представить, где вы сейчас находитесь. Вы сами себе строжайший критик, поэтому здесь, в конце, важно перечислить основные пункты всего, что вы узнали. Имейте в виду вот что:

- Разработка ПО — это не одна, а мириады практик, которые в итоге сливаются в программном проекте.
- Баланс всех этих практик будет бесконечной задачей, а некоторые практики будут приходить и уходить по мере того, как вы сосредоточиваетесь на улучшении других.
- Это не точная наука. Принимайте утверждения, заявляющие о том, что нечто является «единственным истинным путем», с большой долей скепсиса.
- Вы можете применять принципы, которые усвоили в этой книге, практически к любому языку, структуре или задаче. Python — прекрасен, но не загоняйте себя в рамки.

11.5.1. Путешествие туда и обратно: рассказ разработчика

В главе 1 вы сразу же окунулись в идею разработки ПО. Понимание того, что разработка ПО может быть продуманным, глубокомысленным процессом, легло в основу всех последующих глав. Иногда будет трудно найти время из-за дедлайнов и прочего, чтобы продумать проект заранее, но постарайтесь как можно чаще находить возможность, чтобы можно было обдуманно подходить к ПО, которое вы создаете. Результаты

остаются самой важной целью, но дизайн поможет вам продолжать достигать результатов успешнее.

В главе 2 было рассказано о разделении ответственности. Большинство современных языков программирования поощряют использование функций, методов, классов и модулей, и не без оснований. Разложение ПО на составные части помогает снизить когнитивную нагрузку, а также улучшить сопроводимость кода. Обязанности могут быть разделены на самых низких уровнях кода, вплоть до более широкой архитектуры ПО.

Опираясь на структуры Python для разделения ответственности, вы научились использовать их для абстракции и инкапсуляции в главе 3. Освобождение себя и других разработчиков от мелких подробностей конкретной задачи, если только они не заинтересованы в том, чтобы знать больше, приносит желанное облегчение. Показывая только критически важные детали другим участкам команды, вы также снижаете точки интеграции и вероятность ломанного кода для потребителей.

В рамках более подробного рассмотрения, в главе 4 вы узнали о дизайне с учетом производительности. Вы увидели несколько предоставляемых языком Python структур данных и то, в каких ситуациях они полезны. Вы также узнали о некоторых инструментах для количественного измерения производительности ПО.

Если в главе 4 было показано, как тестировать эффективность программ, то в главе 5 основное внимание было уделено тестированию *правильности* программ. Функциональное тестирование помогает обеспечивать совпадение ожиданий и реальной разработки. Вы узнали то, каким образом структурированы функциональные тесты и как писать тесты с помощью инструментов Python. Паттерны функционального тестирования у языков и тестовых фреймворков весьма похожи, поэтому эту информацию можно применять практически в любой области.

Вооружившись некоторыми основами проектирования, в следующей части вы создавали приложение Bark, где достигли ряда важных этапов:

- Создали многослойную архитектуру для поддержки отдельных слоев визуализации, бизнес-логики и постоянства данных.
- Открыли приложение Bark для расширения, упростив добавление новой функциональности, а затем добавили новую функциональную характеристику по импортированию звезд из GitHub в форме закладок.
- Использовали интерфейсы и паттерн Команда, еще больше сократив работу, необходимую для добавления или изменения характеристик.
- Ослабили сопряженность между разными участками приложения Bark, открыв новые возможности, например перенос на мобильную платформу.

Скромный инструмент для ведения закладок останется для вас кладезем ярких технических приемов. Применение накопленных знаний к будущим проектам для решения реальных задач неизбежно даст вам столь же эффективные результаты. Вы можете попрактиковаться в освоении любых новых концепций, применив их и к Bark: можно добавить новые характеристики, улучшить существующий код или написать для него тесты. Приятного полета!

11.5.2. Выход из системы

Вы закончили читать эту книгу. Мне было очень приятно вас учить, и я надеюсь услышать ваши рассказы, когда вы перейдете к более крупным и качественным программным проектам. Празднуйте победы, учитесь и развивайтесь.

Приятного написания кода!

ИТОГИ

- Обучение — это не пассивный процесс. Составьте план, который будет работать на вас, запишите его или набросайте интеллект-карту и следите за своим прогрессом. Это может породить больше

идей или следующих шагов, которые помогут сохранять мотивацию и любопытство.

- Постарайтесь выявлять закономерности и подходы к решению задач. По мере того как вы будете решать однотипные задачи, попробуйте несколько разных подходов на ранней стадии, чтобы увидеть, какие из них работают лучше и эффективнее. Паттерны — это инструменты, и они должны улучшать работу, а не мешать ей.
- Хорошо знайте свой язык программирования. Не нужно браться за все сразу, но сохраняйте любопытство и почаще спрашивайте себя, есть ли более идиоматический способ выразить мысль в коде.
- Вы прошли долгий путь от начала этой книги, поэтому возьмите паузу, чтобы как следует обдумать прочитанное.

Приложение. Установка языка Python

В приложении:

- ✓ Версии Python и их использование.
- ✓ Установка языка Python на компьютер.

Python — это переносимое ПО, которое может быть скомпилировано из исходного кода в большинстве систем. К счастью для вас, Python, скорее всего, также имеется в предварительно собранном виде для вашей операционной системы. Это приложение поможет настроить Python так, чтобы вы могли запускать любой код, приводимый в этой книге, из командной строки.

СОВЕТ

Если вы уже установили версию Python 3 на компьютер, то вам повезло. Тут просто больше нечего делать. Вы свободно можете вернуться к чтению и работать с кодом из этой книги.

ПРИМЕЧАНИЕ

Если вы устанавливаете версию Python 3, то во время выполнения кода вам почти наверняка придется использовать команду `python3`. Ключевое слово `python` зарезервировано для совсем другой установки Python во многих операционных системах (раздел А.2).

А.1. КАКУЮ ВЕРСИЮ PYTHON ИСПОЛЬЗОВАТЬ?

Первая версия Python 2.7 вышла в 2010 году, и на момент написания этих строк macOS поставляется с Python 2.7.10, что на несколько версий отстает от последней версии Python 2.7. С 1 января 2020 года Python 2.7 официально не будет поддерживаться.

Если вы уже знакомы с Python 2 и беспокоитесь об обновлении до Python 3, то знайте, что большинство изменений в коде, которые вам нужно будет внести, невелики. При запуске новых проектов я рекомендую использовать Python 3. Это настроит вас на написание кода, который продержится дольше в будущем.

СОВЕТ

Существуют инструменты, которые помогут вам с обновлением Python 2 до Python 3. Python предоставляет модуль `__future__`, который позволяет использовать более новые функциональные средства Python 3, которые были перенесены обратно в Python 2. Благодаря этому при обновлении ваш синтаксис будет уже правильным и вы сможете просто удалить инструкцию импорта `future`. Пакет `Six` (дважды три, поняли?) (https://six.readthedocs.io/) также помогает оседлать две версии.

А.2. «СИСТЕМНЫЙ» PYTHON

Во многих операционных системах Python уже может быть заранее установлен, потому что система нуждается в нем для решения некоторых

своих задач. Такую установку Python часто называют «системным» языком Python. Например, в macOS установлен и доступен для использования Python 2.7.

Использование системного Python усложняется, когда нужно установить пакеты, потому что все они будут установлены под глобальной версией Python. Если вы устанавливаете пакет, который переопределяет то, что нужно операционной системе, или у вас есть несколько проектов, которым нужны разные версии пакета, то могут возникнуть проблемы. Настоятельно рекомендую избегать системного Python.

А.3. УСТАНОВКА ДРУГИХ ВЕРСИЙ PYTHON

Если вы еще не установили свою версию Python, то в вашем распоряжении имеется несколько вариантов. Все они функционально эквивалентны, поэтому ваш выбор будет зависеть от того, что соответствует вашему рабочему потоку или имеет для вас смысл.

Единственное, что важно, так это убедиться, что у вас есть относительно недавняя версия Python. Рекомендую Python 3.6+ для сильной совместимости с большинством существующих библиотек, но на момент написания этих строк уже доступен Python 3.8. Если у вас нет каких-либо специальных требований, то постарайтесь ставить последнюю версию.

А.3.1. Скачайте официальный Python

Можно скачать Python непосредственно с официального сайта Python (www.python.org/downloads). Сайт распознает тип вашей операционной системы и покажет большую кнопку скачивания Python (рис. А.1). Если по каким-то причинам система не распознается или распознается неверно, то можно перейти по прямым ссылкам на различные операционные системы.

Скачанный файл такой же, как и большинство других приложений, которые вы устанавливаете в своей системе. В macOS открытие скачанного файла приведет вас к мастеру установки (рис. А.2). Какие варианты

установки вы выберете в мастере, зависит от вас, но значения, выставленные по умолчанию, обычно являются нормальными.

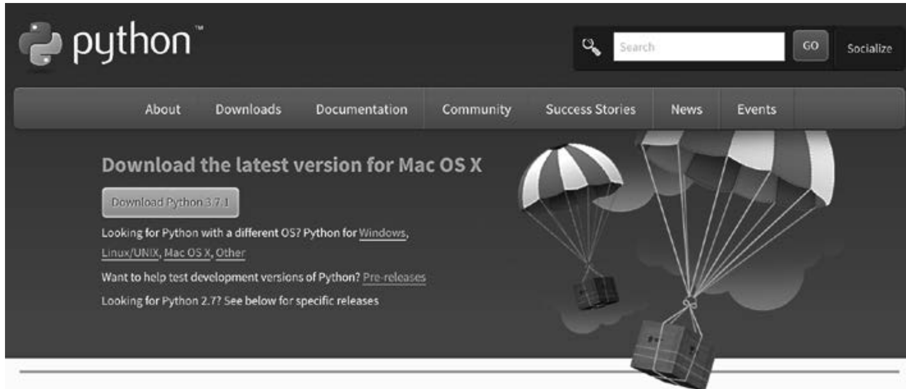


Рис. А.1. После нажатия на большую желтую кнопку будет загружена последняя версия Python. Вы можете найти более старые версии или версии для других операционных систем в ссылках ниже под ней

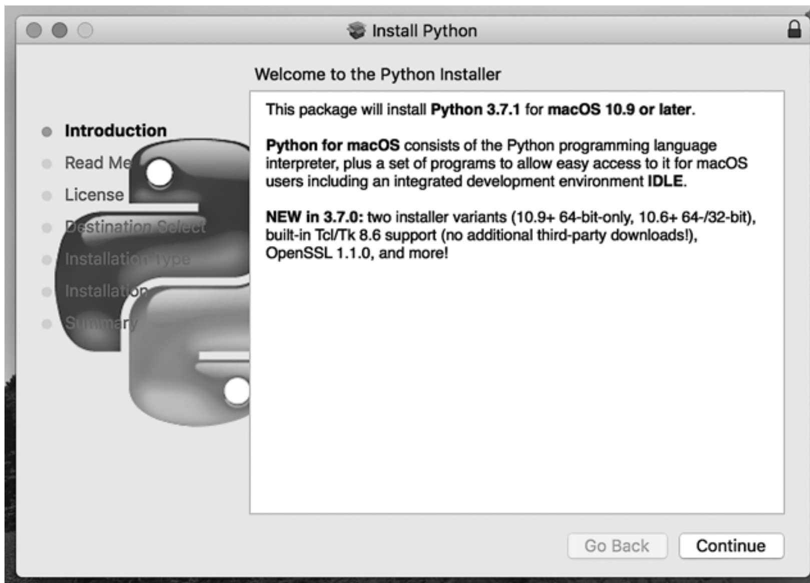


Рис. А.2. Обычно я просто безрассудно нажимаю кнопку «Continue» (Продолжить)

А.3.2. Скачать с помощью Anaconda

Если вы принадлежите к научному вычислительному сообществу, то, скорее всего, знакомы с Anaconda (www.anaconda.com). Anaconda — это комплект инструментов, который включает в себя Python. На момент написания этих строк Anaconda может быть установлена либо с Python 2, либо с Python 3. Проверьте, что у вас установлено, и убедитесь, что у вас есть версия Python 3.

С помощью команды `conda` можно установить большинство версий Python, используя, к примеру, `conda install python=3.7.3`. Ознакомьтесь с официальной документацией, чтобы разобраться с процессом установки в вашей системе.

А.4. ПРОВЕРКА УСТАНОВКИ

После завершения установки откройте терминал (приложение «Терминал» в macOS). Попробуйте выполнить команду `python` (либо `python3`, как вариант). Если язык Python установлен успешно, то вы увидите приветствие интерактивной среды Python REPL, которое должно где-то внутри содержать сообщение о `Python 3`:

```
$ python3
Python 3.7.3 (default, Jun 17 2019, 14:09:05)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Попробуйте ввести свой любимый фрагмент кода и посмотрите, что произойдет:

```
>>> print('Здравствуй, мир!')
Здравствуй, мир!
```

Теперь вы готовы поработить мир!

Тестовые дублеры
Юнит-тестирование
Тестирование производительности
Нагрузочное тестирование
Функциональное тестирование
Тестирование
Временная сложность
Пространственная сложность
Рефакторинг
Производительность
Сопроводимость
Расширяемость
Разделение задач
Многослойная архитектура
Архитектура ПО
Секреты Python Pro
Абстракция
Структура кода
Парадигмы программирования
Организация пространств имен
Объектно-ориентированное программирование
Наследование
Композиция
Инкапсуляция

Дейн Хиллард

Секреты Python Pro

Перевел с английского *А. Логунов*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

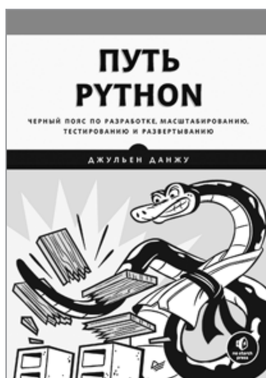
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 18.11.20. Формат 70х100/16. Бумага офсетная. Усл. п. л. 25,800. Тираж 700. Заказ 0000.

Отпечатано в типографии ООО «Комбинат программных средств».
420044, РТ, г. Казань, пр. Ямашева, д. 36Б.

Джульен Данжу

ПУТЬ PYTHON. ЧЕРНЫЙ ПОЯС ПО РАЗРАБОТКЕ, МАСШТАБИРОВАНИЮ, ТЕСТИРОВАНИЮ И РАЗВЕРТЫВАНИЮ



«Путь Python» позволяет отточить ваши профессиональные навыки и узнать как можно больше о возможностях самого популярного языка программирования. Эта книга написана для разработчиков и опытных программистов. Вы научитесь писать эффективный код, создавать лучшие программы за минимальное время и избегать распространенных ошибок. Пора познакомиться с многопоточными вычислениями и мемоизацией, получить советы экспертов в области дизайна API и баз данных, а также заглянуть внутрь Python, чтобы расширит понимание языка.

Вам предстоит начать проект, поработать с версиями, организовать автоматическое тестирование и выбрать стиль программирования для конкретной задачи. Потом вы перейдете к изучению эффективного объявления функции, выбору подходящих структур данных и библиотек, созданию безотказных программ, пакетам и оптимизации программ на уровне байт-кода.

КУПИТЬ

Лейн Хобсон, Хапке Ханнес, Ховард Коул

ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА В ДЕЙСТВИИ

Последние достижения в области глубокого обучения позволяют создавать приложения, с исключительной точностью распознающие текст и речь. Что в результате? Появляются чат-боты, ведущие диалог не хуже реальных людей, программы, эффективно подбирающие резюме под заданную вакансию, развивается превосходный предиктивный поиск, автоматически генерируются аннотации документов. Благодаря новым приемам и инструментам, таким как Keras и Tensorflow, сегодня возможно как никогда просто реализовать качественную обработку естественного языка (NLP).

«Обработка естественного языка в действии» станет вашим руководством по созданию программ, способных распознавать и интерпретировать человеческий язык. В издании рассказано, как с помощью готовых пакетов на языке Python извлекать из текста смыслы и адекватно ими распоряжаться. В книге дается расширенная трактовка традиционных методов NLP, что позволит задействовать нейронные сети, современные алгоритмы глубокого обучения и генеративные приемы при решении реальных задач, таких как выявление дат и имен, составление текстов и ответов на неожиданные вопросы.

КУПИТЬ