

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ С ПОМОЩЬЮ**

PYTHON

ИРВ КАЛЬБ



 **БОМБОРА**
издательство





IRV KALB

OBJECT-ORIENTED
PYTHON

MASTER OOP BY
BUILDING GAMES AND GUIs



ИРВ КАЛЬБ

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ С ПОМОЩЬЮ
PYTHON**

УДК 004.42
ББК 32.973.26-018.2
К17

Object-Oriented Python: Master OOP by Building Games and GUIs
Irv Kalb

Copyright © 2022 by Irv Kalb. Title of English-language original: Object-Oriented Python: Master OOP by Building Games and GUIs, ISBN 9781718502062, published by No Starch Press Inc. 2458th Street, San Francisco, California United States 94103. The Russian-language edition. Copyright © 2024 by Eksmo Publishing House under license by No Starch Press Inc. All rights reserved.

Кальб, Ирв.

К17 Объектно-ориентированное программирование с помощью Python / Ирв Кальб ; [перевод с английского М. А. Райтмана]. — Москва : Эксмо, 2024. — 512 с. — (Мировой компьютерный бестселлер).

ISBN 978-5-04-186627-3

Объектно-ориентированное программирование (ООП) — это метод, основанный на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования, что позволяет по-другому думать о вычислительных задачах и решать их с возможностью многократного использования. «Объектно-ориентированное программирование с помощью Python» предназначено для программистов среднего уровня и представляет собой практическое руководство, которое глубоко изучает основные принципы ООП и показывает, как использовать инкапсуляцию, полиморфизм и наследование для написания игр и приложений с использованием Python.

Книга начинается с рассказа о ключевых проблемах, присущих процедурному программированию, затем вы познакомитесь с основами создания классов и объектов в Python.

Затем вы научитесь создавать графические интерфейсы с помощью Pygame, благодаря чему вы сможете писать интерактивные игры и приложения с виджетами графического пользовательского интерфейса (GUI), анимацией, различными сценами и многообразной игровой логикой.

УДК 004.42
ББК 32.973.26-018.2

ISBN 978-5-04-186627-3

© Райтман М.А., перевод на русский язык, 2024
© Оформление. ООО «Издательство «Эксмо», 2024

Моей замечательной жене Дорин.

Ты клей, который держит нашу семью вместе.

Много лет назад я сказал: «Я делаю», но я имел в виду: «Я сделаю».

ОБ АВТОРЕ

Ирв Кальб — профессор в UCSC Silicon Valley Extension и Университете Кремниевой долины (ранее Политехнический колледж Когсвелла), где он преподает вводные курсы программирования и курсы объектно-ориентированного программирования на языке Python. Ирв имеет степени бакалавра и магистра в области компьютерных наук, более 30 лет занимается объектно-ориентированным программированием на различных языках и более 10 лет преподает. У него десятилетний опыт разработки программного обеспечения с акцентом на образовательное ПО. Как Furry Pants Productions он и его жена создали и выпустили два обучающих диска с персонажем — далматинцем Дарби в главной роли. Ирв также является автором *Learn to Program with Python 3: A Step-by-Step Guide to Programming* («Учимся программировать на Python 3. Пошаговое руководство по программированию»).

Ирв активно участвовал в раннем развитии спорта Ultimate Frisbee®. Он возглавил создание многих версий официального сборника правил и стал соавтором и издателем первой книги об этом виде спорта — *Ultimate: Fundamentals of the Sport* («Ultimate: Основы спорта»).

О ТЕХНИЧЕСКОМ АВТОРЕ

Монте Давидовфф — независимый консультант по разработке программного обеспечения. Его области специализации включают DevOps и Linux. Монте программирует на Python уже более 20 лет. Он использовал Python для разработки разнообразного программного обеспечения, включая критически важные для бизнеса приложения и встроенное ПО.

КРАТКОЕ СОДЕРЖАНИЕ

Об авторе	6
О техническом авторе	7
Благодарности	17
Введение	19
Часть I. Введение в объектно-ориентированное программирование	29
1. Процедурные примеры Python	31
2. Моделирование физических объектов с помощью объектно-ориентированного программирования	55
3. Мысленные модели объектов и значение self	89
4. Управление несколькими объектами	101
Часть II. Графические пользовательские интерфейсы с pygame	139
5. Введение в pygame	141
6. Объектно-ориентированный pygame	183
7. Виджеты pygame GUI	211
Часть III. Инкапсуляция, полиморфизм и наследование	233
8. Инкапсуляция	235
9. Полиморфизм	263
10. Наследование	297
11. Управление памятью, используемой объектами	335
Часть IV. Использование ООП в разработке игр	363
12. Карточные игры	365
13. Таймеры	381

14. Анимация	399
15. Сцены	419
16. Полноценная игра: Dodger	457
17. Шаблоны проектирования и резюме	491
Предметный указатель	503

ПОДРОБНОЕ СОДЕРЖАНИЕ

Об авторе	6
О техническом авторе	7
Благодарности	17
Введение	19
Для кого эта книга?	20
Версия(-и) Python и установка	20
Как я объясняю ООП?	22
Структура книги	23
Среды разработки	26
Виджеты и примеры игр	26

Часть I. Введение в объектно-ориентированное программирование

29

1. Процедурные примеры Python	31
Карточная игра «Больше-меньше»	32
Представление данных	32
Реализация	33
Повторное использование кода	35
Моделирование банковского счета	36
Анализ необходимых операций и данных	36
Реализация 1. Одна учетная запись без функций	37
Реализация 2. Одна учетная запись с функциями	39
Реализация 3. Два счета	42
Реализация 4. Несколько счетов с использованием списков	44
Реализация 5. Список словарей учетных записей	47
Общие проблемы с процедурной реализацией	50
Объектно-ориентированное решение — первый взгляд на класс	51
Выводы	52
2. Моделирование физических объектов с помощью объектно-ориентированного программирования	55
Построение программных моделей физических объектов	56
Состояние и поведение: пример выключателя освещения	56
Введение в классы и объекты	58
Классы, объекты и экземпляры	60
Написание класса на Python	61

Область видимости и переменные экземпляра	63
Различия между функциями и методами	64
Создание объекта из класса	66
Вызов методов объекта	66
Создание нескольких экземпляров из одного класса	67
Типы данных Python реализованы как классы	69
Определение объекта	70
Создание несколько более сложного класса	70
Представление более сложного физического объекта как класса	73
Передача аргументов методу	79
Несколько экземпляров	81
Параметры инициализации	83
Использование классов	85
ООП как решение	86
Выводы	86
3. Мысленные модели объектов и значение self	89
Повторный обзор класса DimmerSwitch	90
Высокоуровневая мысленная модель № 1	91
Более глубокая мысленная модель № 2	92
В чем смысл слова «self»?	95
Выводы	99
4. Управление несколькими объектами	101
Класс банковского счета	101
Импорт кода класса	105
Создание тестового кода	107
Создание нескольких учетных записей	107
Несколько объектов учетной записи в списке	110
Несколько объектов с уникальными идентификаторами	112
Создание интерактивного меню	115
Создание объекта диспетчера объектов	118
Создание объекта диспетчера объектов	120
Основной код, создающий объект диспетчера объектов	123
Лучшая обработка ошибок с исключениями	125
try и except	125
Инструкция raise и пользовательские исключения	126
Использование исключений в нашей банковской программе	128
Класс счета с исключениями	128
Оптимизированный класс банка	130
Основной код, обрабатывающий исключения	132
Вызов одного и того же метода для списка объектов	134
Интерфейс по сравнению с реализацией	136
Выводы	137

Часть II. Графические пользовательские интерфейсы с pygame

139

5. Введение в pygame	141
Устанавливаем Pygame	142
Детали окон	143
Система координат окна	144
Цвета пикселей	147

Программы, управляемые событиями	148
Используем Pygame	150
Создаем пустое окно	151
Рисуем изображение	155
Обнаруживаем щелчок мыши	158
Обрабатываем клавиатуру	161
Создаем анимацию, основанную на местоположении	167
Используем rect pygame	169
Воспроизводим звуки	173
Воспроизводим звуковые эффекты	173
Воспроизведение фоновой музыки	175
Рисуем фигуры	176
Справка по примитивным фигурам	179
Выводы	181
6. Объектно-ориентированный pygame	183
Создаем заставку мяча с помощью Pygame	183
Создаем класс Ball	184
Используем класс Ball	186
Создаем много объектов Ball	188
Создаем много, много объектов Ball	190
Создаем многократно используемую объектно-ориентированную кнопку	190
Создаем класс кнопки	191
Основной код, использующий SimpleButton	194
Создаем программу с несколькими кнопками	196
Создаем многократно используемое отображение текста	197
Шаги для отображения текста	198
Создаем класс SimpleText	199
Демоверсия Ball с SimpleText и SimpleButton	201
Сравнение интерфейса и реализации	203
Обратные вызовы	204
Создаем обратный вызов	205
Используем обратный вызов с SimpleButton	206
Выводы	209
7. Виджеты pygame GUI	211
Передаем аргументы функции или методу	212
Позиционные параметры и параметры ключевых слов	213
Дополнительные примечания к параметрам ключевых слов	214
Используем None в качестве значения по умолчанию	216
Выбираем ключевые слова и значения по умолчанию	217
Значения по умолчанию в виджетах GUI	218
Пакет pygame.widgets	218
Установка	219
Общий подход к разработке	220
Добавляем изображение	221
Добавляем кнопки, флажки и переключатели	222
Вывод и ввод текста	226
Другие классы pygame.widgets	229
pygame.widgets в примере программы	231
Важность последовательного API	231
Выводы	231

Часть III. Инкапсуляция, полиморфизм и наследование

233

8. Инкапсуляция	235
Инкапсуляция с помощью функций	236
Инкапсуляция с помощью объектов	237
Объекты владеют своими данными	237
Интерпретации инкапсуляции	238
Прямой доступ и почему его следует избегать	238
Строгая интерпретация с помощью геттеров и сеттеров	244
Безопасный прямой доступ	246
Делаем переменные экземпляра более закрытыми	247
Неявно закрытый	247
Более явно закрытый	248
Декораторы и @property	249
Инкапсуляция в классах <code>pygame</code>	254
История из реального мира	255
Абстракция	257
Выводы	260
9. Полиморфизм	263
Отправляем сообщения объектам реального мира	264
Классический пример полиморфизма в программировании	265
Пример, использующий фигуры <code>pygame</code>	266
Класс квадратной формы	267
Класс круглой и треугольной формы	268
Основная программа, создающая фигуры	272
Расширяем шаблон	274
<code>pygame</code> проявляет полиморфизм	275
Полиморфизм для операторов	276
Магические методы	277
Магические методы оператора сравнения	278
Магические методы в классе <code>Rectangle</code>	280
Использование магических методов основной программой	282
Магические методы математических операторов	284
Векторный пример	285
Создаем строковое представление значений в объекте	288
Класс <code>Fraction</code> с магическими методами	291
Выводы	295
10. Наследование	297
Наследование в объектно-ориентированном программировании	298
Реализуем наследование	300
Пример работника и менеджера	301
Базовый класс: работник	301
Подкласс: менеджер	302
Тестовый код	305
Представление клиента о подклассе	306
Примеры наследования из реального мира	307
<code>InputNumber</code>	308
<code>DisplayMoney</code>	311
Пример использования	314
Наследование нескольких классов от одного базового класса	317

Абстрактные классы и методы	323
Как pygame применяет наследование	327
Иерархия классов	329
Сложность программирования с наследованием	331
Выводы	333
11. Управление памятью, используемой объектами	335
Жизненный цикл объекта	335
Подсчет ссылок	336
Сбор мусора	343
Переменные класса	343
Константы переменных класса	344
Переменные класса для подсчета	345
Собираем все воедино: пример программы «Шары»	347
Модуль констант	349
Код основной программы	350
Менеджер шаров	353
Класс шаров и объекты	356
Управляем памятью: слоты	359
Выводы	362

Часть IV. Использование ООП в разработке игр 363

12. Карточные игры	365
Класс Card	366
Класс Deck	369
Игра «Больше-меньше»	371
Основная программа	372
Объект Game	373
Тестируем с помощью __name__	377
Другие карточные игры	379
Колода для блек-джека	379
Игры с необычными колодами	379
Выводы	380
13. Таймеры	381
Демонстрационная программа таймера	382
Три подхода к реализации таймеров	383
Подсчет фреймов	383
Таймер событий	384
Создаем таймер путем вычисления прошедшего времени	386
Устанавливаем pygame helpers	388
Класс Timer	389
Отображаем время	392
CountUpTimer	393
CountDownTimer	397
Выводы	398
14. Анимация	399
Создаем классы анимации	400
Класс SimpleAnimation	400
Класс SimpleSpriteSheetAnimation	405
Объединяем два класса	410

Классы анимации в pygame	411
Класс Animation	412
Класс SpriteSheetAnimation	413
Общий базовый класс: PygAnimation	415
Пример программы анимации	416
Выводы	418
15. Сцены	419
Концепция конечного автомата	420
Пример pygame с конечным автоматом	423
Демоверсия программы, использующая менеджер сцен	430
Основная программа	432
Создаем сцены	433
Типичная сцена	437
Игра «Камень-ножницы-бумага», использующая сцены	439
Взаимодействие между сценами	444
Запрашиваем информацию у целевой сцены	445
Отправляем информацию целевой сцене	446
Отправляем информацию всем сценам	446
Проверяем взаимодействие между сценами	447
Реализация менеджера сцен	447
Метод run()	449
Основные методы	451
Взаимодействие между сценами	453
Выводы	454
16. Полноценная игра: Dodger	457
Модальные диалоговые окна	458
Диалоговые окна Yes/No и Предупреждения	458
Диалоговые окна с ответом	462
Создаем полноценную игру: Dodger	465
Обзор игры	465
Реализация	466
Дополнения к игре	488
Выводы	489
17. Шаблоны проектирования и резюме	491
Модель Представление Контроллер	492
Пример отображения файла	492
Пример статистического отображения	493
Преимущества шаблона MVC	499
Резюме	500
Предметный указатель	503

БЛАГОДАРНОСТИ

Я хотел бы поблагодарить людей, которые помогли создать эту книгу:

- Эла Свейгарта — за то, что приучил меня к `pygame` (особенно за его код `Pygbutton`), и за то, что позволил мне использовать концепцию его игры `Dodger`;
- Монте Давидовфа, который помог мне получить исходный код и документацию к нему для правильной сборки с помощью `GitHub`, `Sphinx` и `ReadTheDocs`. Этот человек творил чудеса, используя бесчисленное множество инструментов, чтобы управляться с файлами;
- Монте Давидовфа (да, это все тот же парень) — за то, что оказался выдающимся техническим рецензентом. Монте дал отличные технические и авторские предложения по всей книге, и многие примеры кода стали более `Pythonic` и более ООП-ориентированными именно после его комментариев;
- Тепа Сатъя Кхиейу, который проделал потрясающую работу, нарисовав все оригинальные схемы для этой книги.
Я не художник (я даже не играю ни одного на ТВ). Теп смог взять мои примитивные карандашные наброски и превратить их в четкие, последовательные произведения искусства;

- Харрисона Янга, Кевина Лая и Эмили Эллис — за их вклад в художественное оформление некоторых игр;
- ранних рецензентов: Илью Кацюка, Джейми Калб, Гергану Анджелову и Джо Лангмюра, которые нашли и устранили много опечаток и внесли отличные предложения по исправлениям и уточнениям;
- всех редакторов, которые работали над этой книгой: Лиз Чедвик (редактор по развитию), Рейчел Хед (редактор) и Кейт Камински (производственный редактор). Все они внесли огромный вклад, задавая вопросы и часто переписывая и реорганизуя некоторые из моих объяснений. Они также были чрезвычайно полезны в расставлении и удалении запятых [нужна ли она здесь?] и удлинении моих предложений, как здесь, чтобы убедиться, что точка встречается там, где нужна (ОК, я остановлюсь!). Я боюсь, что никогда не пойму, где следует использовать «который», а не «что», или как расставлять запятые и тире, но я рад, что они знают! Спасибо также Морин Форис (верстальщице) за ее ценный вклад в готовый продукт;
- всех студентов, которые побывали на моих занятиях в течение многих лет в UCSC Silicon Valley Extension и в Университете Кремниевой долины (ранее Политехнический колледж Когсвелла). Их отзывы, предложения, улыбки, недовольство, моменты озарения, разочарование, понимающие кивки и даже большие пальцы вверх (в классах Zoom в эпоху ковида) были чрезвычайно полезны в формировании содержания этой книги и моего общего стиля преподавания;
- наконец, мою семью, которая поддерживала меня в длительном процессе написания, тестирования, редактирования, переписывания, редактирования, отладки, редактирования, переписывания, редактирования (и так далее) этой книги и связанного с ней кода. Без вас я бы не справился. Я не был уверен, достаточно ли у нас книг в библиотеке, поэтому написал еще одну!

ВВЕДЕНИЕ



Эта книга о технике разработки программного обеспечения под названием *объектно-ориентированное программирование* (ООП) и о том, как его можно использовать с Python.

До ООП программисты применяли подход, известный как *процедурное* или *структурированное программирование*, который включает в себя построение набора функций (процедур) и передачу данных через вызовы к этим функциям. Парадигма ООП дает эффективный способ объединить код и данные в связанные единицы, которые подходят для повторного использования.

В процессе подготовки к написанию этой книги я подробно познакомился с существующей литературой и видеоматериалами, изучив конкретные подходы, используемые для объяснения этой важной и широкомасштабной темы. Я обнаружил, что учителя и писатели обычно начинают с определения некоторых ключевых терминов: *класс*, *переменная экземпляра*, *метод*, *инкапсуляция*, *наследование*, *полиморфизм* и так далее.

Хотя все это важные понятия и мы их подробно здесь рассмотрим, я все же начну с другого — с вопроса: «Какую проблему мы решаем?» То есть если решением является ООП, то в чем

проблема? Чтобы ответить, я покажу несколько примеров программ, построенных с использованием процедурного программирования, а затем выявлю сложности, присущие этому стилю. А после покажу вам, как объектно-ориентированный подход может упростить разработку и облегчить поддержку таких программ.

Для кого эта книга?

Эта книга предназначена для людей, которые уже знакомы с Python и используют основные функции из его стандартной библиотеки. Я предполагаю, что вы понимаете основной синтаксис языка и можете писать небольшие и средние программы с помощью переменных, инструкций присваивания, операторов `if/elif/else`, а также циклы, функции и вызовы функций, списки, словари и так далее. Если вы не знакомы со всеми этими понятиями, то я предлагаю вам сначала ознакомиться с моей предыдущей работой, *Learn to Program with Python 3**.

Издание, которое вы сейчас читаете, среднего уровня, поэтому есть ряд более продвинутых тем, их я не стану затрагивать. Например, чтобы книга оставалась практичной, я не часто буду вдаваться в подробности о внутренней реализации Python. Для простоты и ясности, а также для того, чтобы сосредоточиться на освоении методов ООП, примеры написаны с использованием ограниченного подмножества языка. Есть более продвинутые и лаконичные способы программирования на Python, которые выходят за рамки этой книги.

Я рассмотрю главные детали ООП, не зависящие от языка, но обозначу области, где существуют различия между Python и другими языками ООП. Ознакомившись с основами программирования в стиле ООП в этой книге, при желании вы сможете легко применить те же методы к другим языкам ООП.

Версия(-и) Python и установка

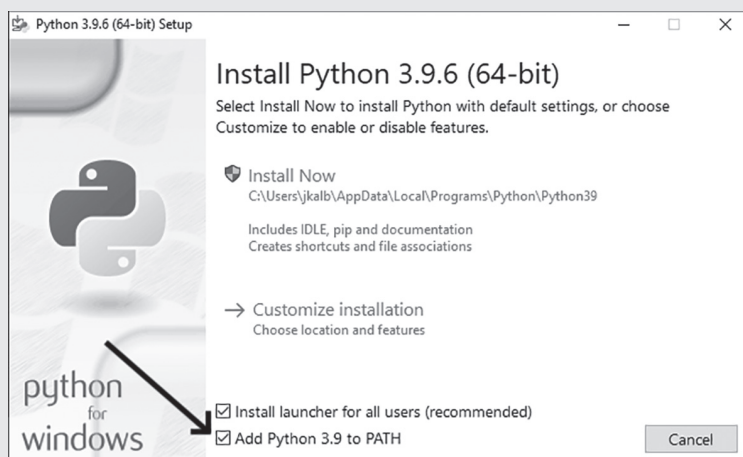
Все примеры кода в этой книге были написаны и протестированы с использованием версий Python с 3.6 по 3.9 и должны хорошо работать с версией 3.6 или новее.

* Не издавалась на русском языке. — Прим. ред.

Python доступен бесплатно по адресу <https://www.python.org>. Если у вас не установлен Python или вы хотите обновиться до последней версии, перейдите на этот сайт, найдите вкладку Downloads и нажмите кнопку **Download**. Так вы загрузите установочный файл на свой компьютер. Дважды щелкните по нему для установки Python.

УСТАНОВКА НА WINDOWS

При установке на систему Windows необходимо обратить внимание на один важный параметр. При выполнении шагов по установке вы увидите экран, как на рисунке ниже.



В нижней части диалогового окна находится флажок **Add Python 3.x to PATH** (Добавить Python 3.x в переменную PATH). Пожалуйста, не забудьте установить его (по умолчанию он не установлен). Данная настройка позволит правильно установить пакет `pygame` (который будет представлен в книге позже).

ПРИМЕЧАНИЕ Мне известен «PEP 8 — Руководство по стилю для кода Python» и его конкретная рекомендация использовать Snake Case (змеиный регистр) для имен переменных и функций. Тем не менее я использовал конвенцию об именовании Camel Case (горбатый регистр) в течение многих лет до того, как был написан

документ PEP 8, и мне было комфортно с ним на протяжении моей карьеры. Поэтому все имена переменных и функций в этой книге написаны с использованием горбатого регистра.

Как я объясняю ООП?

Примеры в первых нескольких главах используют текстовый Python; в них программы получают входные данные от пользователя и выводят ему информацию исключительно в виде текста. Я представляю ООП, показывая, как моделировать физические объекты в коде на основе текста. Для начала мы представим в виде объектов выключатели света, диммеры и телевизионные пульты дистанционного управления. Затем я покажу вам, как с помощью ООП моделировать банковские счета и банк в целом.

Как только мы рассмотрим основы ООП, я представляю модуль *pygame*, который позволяет программистам писать игры и приложения, использующие *графический пользовательский интерфейс* (GUI). В программах на основе графического интерфейса пользователь интуитивно взаимодействует с кнопками, флажками, полями ввода и вывода текста и другими удобными виджетами.

Я решил использовать *pygame* с Python, потому что эта комбинация позволяет мне демонстрировать концепции ООП визуально, используя элементы на экране. *Pygame* чрезвычайно портативна и работает практически на всех платформах и операционных системах. Все образцы программ, использующих пакет *pygame*, были протестированы с недавно выпущенной версией *pygame* 2.0.

Я создал пакет под названием *pygamewidgets*, который работает с *pygame* и реализует ряд базовых виджетов, все они построены с использованием подхода ООП. Я представляю этот пакет позже в книге и дам пример кода, с которым вы можете работать и экспериментировать. Такой подход позволит увидеть реальные, практические примеры ключевых объектно-ориентированных концепций, при этом внедряя эти приемы для создания забавных игр, в которые можно поиграть. Я также представляю мой пакет *pyghelpers*, содержащий код, что помогает написать более сложные игры и приложения.

Весь приведенный в книге код доступен для скачивания как один файл с сайта <https://addons.eksmo.ru/it/OOP-Code.zip>.

Структура книги

Эта книга состоит из четырех частей. Часть I знакомит с объектно-ориентированным программированием.

- В главе 1 представлен обзор кода с использованием процедурного программирования. Я покажу вам, как реализовать текстовую карточную игру и смоделировать банк, выполняющий операции по одному или нескольким счетам. Попутно я обсуждаю общие проблемы процедурного подхода.
- Глава 2 повествует о классах и объектах и показывает, как вы можете представлять реальные вещи, такие как выключатели света или пульт дистанционного управления телевизором, на Python с помощью классов. Вы увидите, как объектно-ориентированный подход решает проблемы, обозначенные в первой главе.
- В главе 3 представлены две ментальные модели, которые вы можете использовать, чтобы думать о том, что происходит за кулисами, когда вы создаете объекты на Python. Мы будем использовать Python Tutor, чтобы просмотреть код и увидеть, как создаются объекты.
- В главе 4 показан стандартный способ обработки нескольких объектов одного типа путем введения понятия объекта диспетчера объектов. Мы расширим моделирование банковского счета с помощью классов, и я покажу вам, как обрабатывать ошибки с помощью исключений.

Часть II посвящена созданию графических интерфейсов с помощью pygame.

- Глава 5 представляет пакет pygame и управляемую событиями модель программирования. Мы создадим несколько простых программ, чтобы вы могли начать с размещения графики в окне и обработки ввода с клавиатуры и мыши, а затем разработаем более сложную программу с прыгающими мячами.

- Глава 6 подробнее описывает использование ООП с программами `pygame`. Мы перепишем приложение с прыгающим мячом в стиле ООП и разработаем некоторые простые элементы графического интерфейса.
- В главе 7 представлен модуль `pygame.widgets`, который содержит полные реализации многих стандартных элементов графического интерфейса (кнопок, флажков и т. д.), каждый из которых разрабатывается как класс.

Часть III посвящена основным принципам ООП.

- В главе 8 обсуждается инкапсуляция, которая включает в себя сокрытие деталей реализации от внешнего кода и размещение всех связанных методов в одном месте — классе.
- Глава 9 вводит полиморфизм — идею о том, что несколько классов могут иметь методы с одинаковыми именами, — и показывает, как он позволяет вызывать методы в нескольких объектах, не зная типа каждого из них. Мы создадим программу `Shapes`, чтобы продемонстрировать эту концепцию.
- Глава 10 описывает наследование, которое позволяет создавать набор подклассов, использующих общий код, встроенный в базовый класс, а не «изобретать колесо» с похожими классами. Мы рассмотрим несколько реальных примеров, когда пригодится наследование, например реализацию поля ввода, которое принимает только числа, а затем перепишем наш пример программы `Shapes`, чтобы использовать эту функциональность.
- Глава 11 завершает эту часть книги обсуждением некоторых дополнительных важных тем ООП, в основном связанных с управлением памятью. Мы посмотрим на время жизни объекта и в качестве примера построим небольшую игру с шариками.

Часть IV посвящена нескольким темам, связанным с использованием ООП в разработке игр.

- Глава 12 демонстрирует, как мы можем перестроить карточную игру, разработанную в главе 1, в качестве программы с графическим интерфейсом на основе `pygame`. Я также

покажу вам, как создавать многоцветные классы колод и карт, которые вы можете использовать при создании других карточных игр.

- В главе 13 рассматривается вопрос о времени. Мы разработаем различные классы таймеров, которые позволят программе продолжать работу, одновременно проверяя заданный лимит времени.
- В главе 14 описываются классы анимации, которые можно использовать для отображения последовательностей изображений. Мы рассмотрим два метода: создание анимации из коллекции отдельных файлов изображений и извлечение и использование нескольких изображений из одного файла листа спрайта.
- Глава 15 объясняет концепцию конечного автомата, представляющего и контролирующего поток ваших программ, и менеджер сцен, который вы можете задействовать для создания программы с несколькими сценами. Чтобы продемонстрировать использование каждого из них, мы построим две версии игры «Камень, ножницы, бумага».
- В главе 16 обсуждаются различные типы модальных диалогов — еще одна важная функция взаимодействия с пользователем. Затем мы пройдемся по созданию полнофункциональной видеоигры на основе ООП под названием Dodger, которая демонстрирует многие методы, описанные в книге.
- В главе 17 представлена концепция шаблонов проектирования на примере шаблона контроллера представления модели, а затем показана программа бросания костей, в которой используется этот шаблон, чтобы позволить пользователю визуализировать данные различными способами. Завершается глава кратким подведением итогов работы над книгой.

Среды разработки

При работе с этой книгой вы будете использовать командную строку только для установки программного обеспечения. Все инструкции по установке четко прописаны, поэтому вам

не потребуется изучать дополнительный синтаксис командной строки.

Я твердо верю, что для разработки лучше использовать не командную строку, а интерактивную среду разработки (IDE). Интегрированная среда обрабатывает многие детали базовой операционной системы за вас и позволяет писать, редактировать и запускать код с помощью одной программы. Интегрированные среды, как правило, являются кросс-платформенными, что позволяет программистам легко перемещаться с Mac на компьютер под управлением Windows или наоборот.

Краткие примеры программ в книге могут быть запущены в среде разработки IDLE, которая устанавливается вместе с Python. IDLE очень проста в использовании и хорошо работает для программ, которые состоят из одного файла. Когда мы перейдем к более сложным программам, использующим несколько файлов Python, лучше применять что-то посложнее; лично я использую среду разработки JetBrains PyCharm, которая легче обрабатывает проекты с несколькими файлами. Community Edition доступен бесплатно по адресу <https://www.jetbrains.com/>, и я настоятельно рекомендую его. В PyCharm входит полностью интегрированный отладчик, который может быть чрезвычайно полезен при написании более крупных программ. Дополнительные сведения об использовании отладчика см. в моем видео на YouTube «Отладка Python 3 с помощью PyCharm» по адресу <https://www.youtube.com/watch?v=cxAOSQQwDJ4&t=43s/>.

Виджеты и примеры игр

В книге представлены и доступны два пакета Python: `pygame` и `pygame`. Используя их, вы сможете создавать полноценные программы с графическим интерфейсом пользователя, но, что более важно, вы получите представление о том, как каждый из виджетов написан как класс и используется как объект.

Примеры игр в книге, включающие различные виджеты, начинаются с относительно простых и становятся все более сложными. Глава 16 рассказывает о разработке и внедрении полнофункциональной видеоигры с таблицей результатов, которая сохраняется в файл.

К концу этой книги вы должны уметь писать собственные игры — карточные или видеоигры в стиле Pong, Hangman, Breakout, Space Invaders и так далее. Объектно-ориентированное программирование дает вам возможность писать программы, которые могут легко отображать и контролировать несколько элементов одного типа, что регулярно требуется при построении пользовательских интерфейсов и часто необходимо в игре.

Объектно-ориентированное программирование — это общий стиль, который можно применять во всех аспектах программирования, далеко за пределами игровых примеров, которые я использую для демонстрации техники ООП. Надеюсь, вам понравится этот подход к изучению ООП.

Итак, начнем.

ЧАСТЬ I

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Эта часть книги знакомит вас с объектно-ориентированным программированием. Мы обсудим проблемы, присущие процедурному коду, а затем посмотрим, как объектно-ориентированное программирование решает их. Мышление в объектах (с состоянием и поведением) даст вам новое представление о том, как писать код.

В главе 1 представлен обзор процедурного Python. Я начинаю с презентации текстовой карточной игры Higher or Lower («Больше-меньше»), затем прорабатываю несколько все более сложных реализаций банковского счета на Python, чтобы помочь вам лучше понять распространенные проблемы программирования в процедурном стиле.

Глава 2 показывает, как мы можем представлять объекты реального мира в Python с помощью классов. Мы напишем программу, имитирующую выключатель света, изменим ее, чтобы добавить возможности диммера (плавного затемнения), а затем перейдем к более сложному моделированию пульта от телевизора.

Глава 3 дает вам два разных способа думать о том, что происходит за кулисами, когда вы создаете объекты на Python.

Глава 4 демонстрирует стандартный способ обработки нескольких объектов одного типа (например, в такой простой игре, как шашки, где вы должны отслеживать много похожих игровых фигур). Мы расширим программу банковского счета из главы 1 и изучим, как обрабатывать ошибки.

1

ПРОЦЕДУРНЫЕ ПРИМЕРЫ PYTHON



Вводные курсы и книги обычно обучают разработке программного обеспечения с использованием стиля процедурного программирования, который включает в себя разделение программы на ряд функций (также известных как процедуры или подпрограммы). Вы передаете данные в функции, каждая из которых выполняет одно или несколько вычислений и, как правило, возвращает обратно результаты.

Эта книга о другой парадигме, известной как *объектно-ориентированное программирование* (ООП), которая позволяет иначе думать о том, как строить программное обеспечение. Объектно-ориентированное программирование дает возможность объединять код и данные, тем самым избегая некоторых осложнений, присущих процедурному программированию.

В этой главе я рассмотрю ряд концепций в Basic Python, создав две небольшие программы, которые включают в себя различные конструкции Python. Первой будет небольшая карточная игра под названием «Больше-меньше»; второй станет симуляция банка, выполняющего операции по одному, двум и нескольким счетам. Обе будут построены при помощи

процедурного программирования, то есть с использованием стандартных методов данных и функций. Позже я перепишу эти программы, применяя методы ООП. Цель данной главы — продемонстрировать некоторые ключевые проблемы, присущие процедурному программированию. В последующих главах будет объяснено, как ООП их решает.

Карточная игра «Больше-меньше»

Мой первый пример — простая карточная игра под названием «Больше-меньше». В ней восемь карт случайным образом выбираются из колоды. Первая отображается лицевой стороной вверх. Игра просит игрока предсказать, будет ли следующая карта в выборе иметь большее или меньшее достоинство, чем текущая. Допустим, что показанная карта имеет значение 3. Игрок отвечает «больше», и показывается вторая карта. Если ее достоинство выше, то игрок выиграл. В этом же примере, если бы игрок ответил «меньше», он бы проиграл.

За каждый правильный ответ игрок получает 20 очков, за неправильный — теряет 15. Если следующая карта, которую нужно перевернуть, имеет то же значение, что и предыдущая, игрок не угадал.

Представление данных

Программа должна представлять колоду из 52 карт, которую я построю в виде списка. Каждый из 52 элементов в списке станет словарем (набором пар ключ/значение). Чтобы представить любую карту, каждый словарь будет содержать три пары ключ/значение: 'ранг', 'масть' и 'значение'. Ранг — это название карты (туз, 2, 3 ... 10, валет, дама, король), но значение — это целое число, используемое для сравнения карт (1, 2, 3 ... 10, 11, 12, 13). Например, «валет треф» будет представлен в виде следующего словаря:

```
{ 'rank': 'Jack', 'suit': 'Clubs', 'value': 11 }
```

Перед раундом игрок создает список, представляющий колоду, и перетасовывает его, чтобы рандомизировать порядок карт. У меня нет графического представления карт, поэтому каждый раз, когда пользователь выбирает «больше» или

«меньше», программа получает словарь карт из колоды и печатает ранг и масть для пользователя. Затем она сравнивает достоинство новой карты с предыдущей и дает обратную связь на ответ пользователя.

Реализация

Листинг 1.1 показывает код игры «Больше-меньше».

ПРИМЕЧАНИЕ Напоминаем, что код, связанный со всеми основными листингами в этой книге, доступен для скачивания по адресу <https://addons.eksmo.ru/it/OOP-Code.zip>. Вы можете либо скачать и запустить код, либо ввести его самостоятельно.

Файл: HigherOrLowerProcedural.py

```
# HigherOrLower
import random

# Константы карт
SUIT_TUPLE = ('Spades', 'Hearts', 'Clubs', 'Diamonds')
RANK_TUPLE = ('Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10',
              'Jack', 'Queen', 'King')

NCARDS = 8

# Проходим по колоде, и эта функция возвращает случайную карту из колоды
def getCard(deckListIn):
    thisCard = deckListIn.pop() # Снимаем одну карту с верхней части
                                # колоды и возвращаем

    return thisCard

# Проходим по колоде, и эта функция возвращает перемешанную копию колоды
def shuffle(deckListIn):
    deckListOut = deckListIn.copy() # создаем копию стартовой колоды
    random.shuffle(deckListOut)
    return deckListOut

# Основной код
print('Welcome to Higher or Lower.')
print('You have to choose whether the next card to be shown will be
higher or lower than the current card.')
print('Getting it right adds 20 points; get it wrong and you lose
15 points.')
print('You have 50 points to start.')
print()
```

```

startingDeckList = []
❶ for suit in SUIT_TUPLE:
    for thisValue, rank in enumerate(RANK_TUPLE):
        cardDict = {'rank':rank, 'suit':suit, 'value':thisValue + 1}
        startingDeckList.append(cardDict)

score = 50

while True: # несколько игр
    print()
    gameDeckList = shuffle(startingDeckList)
    ❷ currentCardDict = getCard(gameDeckList)
    currentCardRank = currentCardDict['rank']
    currentCardValue = currentCardDict['value']
    currentCardSuit = currentCardDict['suit']
    print('Starting card is:', currentCardRank + ' of ' +
currentCardSuit)
    print()

    ❸ for cardNumber in range(0, NCARDS): # играем в одну игру
        # из этого количества карт
        answer = input('Will the next card be higher or lower than
the ' +
currentCardRank + ' of ' +
currentCardSuit + '? (enter h or l): ')
        answer = answer.casefold() # переводим в нижний регистр
    ❹ nextCardDict = getCard(gameDeckList)
    nextCardRank = nextCardDict['rank']
    nextCardSuit = nextCardDict['suit']
    nextCardValue = nextCardDict['value']
    print('Next card is:', nextCardRank + ' of ' + nextCardSuit)

    ❺ if answer == 'h':
        if nextCardValue > currentCardValue:
            print('You got it right, it was higher')
            score = score + 20
        else:
            print('Sorry, it was not higher')
            score = score - 15

    elif answer == 'l':
        if nextCardValue < currentCardValue:
            score = score + 20
            print('You got it right, it was lower')

        else:
            score = score - 15

```

```

        print('Sorry, it was not lower')
    print('Your score is:', score)
    print()
    currentCardRank = nextCardRank
    currentCardValue = nextCardValue # не нужна текущая масть

❸ goAgain = input('To play again, press ENTER, or "q" to quit: ')
    if goAgain == 'q':
        break

print('OK bye')
```

Листинг 1.1. Игра «Больше-меньше» с использованием процедурного Python

Программа начинается с создания колоды в виде списка ❶. Каждая карта – это словарь, состоящий из ранга, масти и значения. Для каждого раунда игры я извлекаю первую карту из колоды и сохраняю компоненты в переменных ❷. Для следующих семи карт пользователю предлагается предсказать, будет ли следующая карта выше или ниже, чем самая последняя ❸. Следующая карта извлекается из колоды, а ее компоненты сохраняются во втором наборе переменных ❹. Игра сравнивает ответ пользователя на текущую карту и выдает обратную связь и баллы на основе результата ❺. Когда пользователь сделал прогнозы для всех семи карт в выборе, мы спрашиваем, хочет ли он сыграть снова ❻.

Эта программа демонстрирует множество элементов программирования в целом и Python в частности: переменные, инструкции присваивания, функции и вызовы функций, инструкции if/else, инструкции print, а также циклы, списки, строки и словари. В этой книге предполагается, что вы уже знакомы со всем, что показано в примере. Если программа содержит что-то новое или непонятное для вас, вероятно, стоит уделить время изучению соответствующего материала, прежде чем двигаться дальше.

Повторное использование кода

Поскольку это игра на основе карт, код создает и использует модель карточной колоды. Если бы мы хотели написать еще одну игру такого рода, было бы здорово иметь возможность повторно использовать этот код.

В процедурной программе часто бывает трудно идентифицировать все фрагменты кода, связанные с одной частью программы, такие как колода и карты в этом примере. В листинге 1.1 код для колоды состоит из двух констант кортежей, двух функций, некоторого основного кода для построения глобального списка, который представляет стартовую колоду из 52 карт, и другого глобального списка, представляющего колоду, которая используется во время игры. Кроме того, обратите внимание, что даже в такой небольшой программе, как эта, данные и код, который обрабатывает данные, могут не быть тесно сгруппированы вместе.

Поэтому повторное использование кода колоды и карты в другой программе не так просто или очевидно. В главе 12 мы вернемся к этой программе и покажем, как решение ООП значительно упрощает повторное использование кода.

Моделирование банковского счета

Во втором примере процедурного программирования я представлю ряд вариантов программы, которая имитирует запуск банка. В каждой новой версии программы я добавлю больше функциональности. Обратите внимание, что эти программы не готовы к выпуску; недопустимые пользовательские данные или неправильное использование приведут к ошибкам. Цель состоит в том, чтобы вы сосредоточились на понимании, как код взаимодействует с данными, связанными с одним или несколькими банковскими счетами.

Для начала подумайте, какие операции клиент хотел бы совершить с банковским счетом и какие данные потребуются для представления счета.

Анализ необходимых операций и данных

Список операций, которые человек хотел бы совершить с банковским счетом, будет включать следующее.

- Открыть счет.
- Внести деньги.
- Снять деньги.
- Проверить баланс.

Далее приведен минимальный список данных, которые нам потребуются для представления банковского счета.

- Имя клиента.
- Пароль.
- Баланс.

Обратите внимание, что все операции — это слова действия (глаголы), а все элементы данных — это понятия (существительные). Реальный банковский счет, безусловно, способен выполнять гораздо больше операций и будет содержать дополнительные данные (такие как адрес владельца счета, номер телефона и номер социального страхования), но для ясности я начну с этих четырех действий и трех данных. Кроме того, чтобы не усложнять, я определяю все суммы целым числом долларов. Я также должен отметить, что в реальном банковском приложении пароли не будут храниться открытым текстом (незашифрованными), как в этих примерах.

Реализация 1. Одна учетная запись без функций

В стартовой версии в листинге 1.2 есть только один счет.

Файл: Bank1_OneAccount.py

```
# Без ООП
# Банк. Версия 1
# Единственный счет

❶ accountName = 'Joe'
   accountBalance = 100
   accountPassword = 'soup'

while True:
❷   print()
      print('Press b to get the balance')
      print('Press d to make a deposit')
      print('Press w to make a withdrawal')
      print('Press s to show the account')
      print('Press q to quit')
      print()

      action = input('What do you want to do? ')
      action = action.lower() # переводим в нижний регистр
      action = action[0] # используем первую букву
```

```

print()

if action == 'b':
    print('Get Balance:')
    userPassword = input('Please enter the password: ')
    if userPassword != accountPassword:
        print('Incorrect password')
    else:
        print('Your balance is:', accountBalance)

elif action == 'd':
    print('Deposit:')
    userDepositAmount = input('Please enter amount to deposit: ')
    userDepositAmount = int(userDepositAmount)
    userPassword = input('Please enter the password: ')

    if userDepositAmount < 0:
        print('You cannot deposit a negative amount!')

    elif userPassword != accountPassword:
        print('Incorrect password')

    else: # OK
        accountBalance = accountBalance + userDepositAmount
        print('Your new balance is:', accountBalance)

elif action == 's': # отображаем
    print('Show:')
    print('      Name', accountName)
    print('      Balance:', accountBalance)
    print('      Password:', accountPassword)
    print()

elif action == 'q':
    break

elif action == 'w':
    print('Withdraw:')

    userWithdrawAmount = input('Please enter the amount to
                                withdraw: ')
    userWithdrawAmount = int(userWithdrawAmount)
    userPassword = input('Please enter the password: ')

    if userWithdrawAmount < 0:
        print('You cannot withdraw a negative amount')

    elif userPassword != accountPassword:

```

```

        print('Incorrect password for this account')

    elif userWithdrawAmount > accountBalance:
        print('You cannot withdraw more than you have in your
              account')

    else: # OK
        accountBalance = accountBalance - userWithdrawAmount
        print('Your new balance is:', accountBalance)

print('Done')
```

Листинг 1.2. Моделирование банка для одного счета

Программа начинается с инициализации трех переменных для представления данных одной учетной записи ❶. Затем отображается меню, которое позволяет выбрать операции ❷. Основной код программы действует непосредственно на глобальные переменные счета.

В этом примере все действия находятся на основном уровне, в коде нет функций. Программа работает нормально, но может показаться немного длинной. Типичный способ сделать длинные программы более четкими — перемещение соответствующего кода в функции и вызов этих функций. Мы изучим это в следующей версии банковской программы.

Реализация 2. Одна учетная запись с функциями

В версии программы в листинге 1.3 код разбивается на отдельные функции, по одной для каждого действия. Опять же, эта симуляция предназначена для одного аккаунта.

Файл: Bank2_OneAccountWithFunctions.py

```

# Без ООП
# Банк 2
# Единственный счет
accountName = ''
accountBalance = 0
accountPassword = ''

❶ def newAccount(name, balance, password):
    global accountName, accountBalance, accountPassword
    accountName = name
```

```

    accountBalance = balance
    accountPassword = password

def show():
    global accountName, accountBalance, accountPassword
    print('        Name', accountName)
    print('        Balance:', accountBalance)
    print('        Password:', accountPassword)
    print()

❷ def getBalance(password):
    global accountName, accountBalance, accountPassword
    if password != accountPassword:
        print('Incorrect password')
        return None
    return accountBalance

❸ def deposit(amountToDeposit, password):
    global accountName, accountBalance, accountPassword
    if amountToDeposit < 0:
        print('You cannot deposit a negative amount!')
        return None

    if password != accountPassword:
        print('Incorrect password')
        return None

    accountBalance = accountBalance + amountToDeposit
    return accountBalance

❹ def withdraw(amountToWithdraw, password):
❺    global accountName, accountBalance, accountPassword
    if amountToWithdraw < 0:
        print('You cannot withdraw a negative amount')
        return None

    if password != accountPassword:
        print('Incorrect password for this account')
        return None

    if amountToWithdraw > accountBalance:
        print('You cannot withdraw more than you have in your account')
        return None

❻    accountBalance = accountBalance - amountToWithdraw
    return accountBalance

newAccount("Joe", 100, 'soup') # создаем аккаунт

```

```

while True:
    print()
    print('Press b to get the balance')
    print('Press d to make a deposit')
    print('Press w to make a withdrawal')
    print('Press s to show the account')
    print('Press q to quit')
    print()

    action = input('What do you want to do? ')
    action = action.lower() # переводим в нижний регистр
    action = action[0] # используем первую букву
    print()

    if action == 'b':
        print('Get Balance:')
        userPassword = input('Please enter the password: ')
        theBalance = getBalance(userPassword)
        if theBalance is not None:
            print('Your balance is:', theBalance)

    7 elif action == 'd':
        print('Deposit:')
        userDepositAmount = input('Please enter amount to deposit: ')
        userDepositAmount = int(userDepositAmount)
        userPassword = input('Please enter the password: ')

    8 newBalance = deposit(userDepositAmount, userPassword)
        if newBalance is not None:
            print('Your new balance is:', newBalance)

--- скрыты вызовы соответствующих функций ---

print('Done')

```

Листинг 1.3. Моделирование банка для одного счета с функциями

В этой версии я построил функцию для каждой из операций, которые мы определили для банковского счета (создать аккаунт ❶, проверить баланс ❷, внести деньги ❸, снять деньги ❹), и сделал так, чтобы основной код содержал вызовы различных функций.

В результате основная программа читается гораздо более удобно. Например, если пользователь вводит d, чтобы указать, что он хочет внести депозит ❷, код теперь вызывает функцию с именем `deposit()` ❸, передавая сумму депозита и пароль учетной записи, введенные пользователем.

Однако если вы посмотрите на определение любой из этих функций — например, `withdraw()`, — то увидите, что код использует оператор `global` ⑤ для доступа (чтения или записи) к переменным, которые представляют банковский счет.

В Python оператор `global` требуется только в том случае, если вы хотите изменить значение глобальной переменной в функции. Тем не менее я использую его здесь, чтобы ясно показать, что эти функции относятся к глобальным переменным, даже если они просто получают значение.

Общий принцип программирования требует, чтобы функции *никогда* не изменяли глобальные переменные. Функция должна использовать только данные, которые передаются в нее, производить расчеты на основе их и, возможно, возвращать результат или результаты. Функция `withdraw()` в этой программе действительно работает, но она нарушает вышеуказанное правило, изменяя значение глобальной переменной `accountBalance` ⑥ (в дополнение к доступу к значению глобальной переменной `accountPassword`).

Реализация 3. Два счета

Версия программы моделирования банка в листинге 1.4 использует тот же подход, что и листинг 1.3, но добавляет возможность иметь два счета.

Файл: `Bank3_TwoAccounts.py`

```
# Без ООП
# Банк 3
# Два счета

account0Name = ''
account0Balance = 0
account0Password = ''
account1Name = ''
account1Balance = 0
account1Password = ''
nAccounts = 0
def newAccount(accountNumber, name, balance, password):
    ❶ global account0Name, account0Balance, account0Password
      global account1Name, account1Balance, account1Password

      if accountNumber == 0:
          account0Name = name
          account0Balance = balance
```

```

        account0Password = password
    if accountNumber == 1:
        account1Name = name
        account1Balance = balance
        account1Password = password

def show():
❷    global account0Name, account0Balance, account0Password
    global account1Name, account1Balance, account1Password

    if account0Name != '':
        print('Account 0')
        print('        Name', account0Name)
        print('        Balance:', account0Balance)
        print('        Password:', account0Password)
        print()
    if account1Name != '':
        print('Account 1')
        print('        Name', account1Name)
        print('        Balance:', account1Balance)
        print('        Password:', account1Password)
        print()

def getBalance(accountNumber, password):
❸    global account0Name, account0Balance, account0Password
    global account1Name, account1Balance, account1Password

    if accountNumber == 0:
        if password != account0Password:
            print('Incorrect password')
            return None
        return account0Balance
    if accountNumber == 1:
        if password != account1Password:
            print('Incorrect password')
            return None
        return account1Balance

--- скрыты дополнительные функции deposit() и withdraw() ---

--- скрыт основной код, который вызывает функции выше ---

print('Done')
```

Листинг 1.4. Моделирование банка для двух счетов с функциями

Уже при двух счетах вы можете увидеть, что этот способ быстро выходит из-под контроля. Во-первых, мы устанавливаем три глобальные переменные для каждого счета в ❶, ❷ и ❸. Кроме того, в каждой функции теперь есть оператор `if`,

выбирающий набор глобальных переменных для доступа или изменения. Всякий раз, когда мы хотим добавить еще один счет, нам нужно добавить еще один набор глобальных переменных и больше операторов `if` в каждой функции. Это просто непрактичный подход. Нам нужен другой способ обработки произвольного количества счетов.

Реализация 4. Несколько счетов с использованием списков

Чтобы упростить размещение нескольких счетов, в листинге 1.5 я буду представлять данные с помощью списков. В этой версии программы я использую три параллельных списка: `accountNamesList`, `accountPasswordsList` и `accountBalancesList`.

Файл: Bank4_N_Accounts.py

```
# Без ООП
# Банк 4
# Любое количество счетов — со списками
❶ accountNamesList = []
accountBalancesList = []
accountPasswordsList = []

def newAccount(name, balance, password):
    global accountNamesList, accountBalancesList, accountPasswordsList
    ❷ accountNamesList.append(name)
    accountBalancesList.append(balance)
    accountPasswordsList.append(password)

def show(accountNumber):
    global accountNamesList, accountBalancesList, accountPasswordsList
    print('Account', accountNumber)
    print('      Name', accountNamesList[accountNumber])
    print('      Balance:', accountBalancesList[accountNumber])
    print('      Password:', accountPasswordsList[accountNumber])
    print()

def getBalance(accountNumber, password):
    global accountNamesList, accountBalancesList, accountPasswordsList
    if password != accountPasswordsList[accountNumber]:
        print('Incorrect password')
        return None
    return accountBalancesList[accountNumber]
```

```

--- скрыты дополнительные функции ---

# создаем два образца учетных записей
❸ print("Joe's account is account number:", len(accountNamesList))
   newAccount("Joe", 100, 'soup')

❹ print("Mary's account is account number:", len(accountNamesList))
   newAccount("Mary", 12345, 'nuts')

while True:
    print()
    print('Press b to get the balance')
    print('Press d to make a deposit')
    print('Press n to create a new account')
    print('Press w to make a withdrawal')
    print('Press s to show all accounts')
    print('Press q to quit')
    print()

    action = input('What do you want to do? ')
    action = action.lower() # переводим в нижний регистр
    action = action[0] # используем первую букву
    print()

    if action == 'b':
        print('Get Balance:')
❺     userAccountNumber = input('Please enter your account number: ')
        userAccountNumber = int(userAccountNumber)
        userPassword = input('Please enter the password: ')
        theBalance = getBalance(userAccountNumber, userPassword)
        if theBalance is not None:
            print('Your balance is:', theBalance)

--- скрыт дополнительный пользовательский интерфейс ---

print('Done')

```

Листинг 1.5. Моделирование банка с параллельными списками

В начале программы я создал все три списка как пустой список ❶. Чтобы создать новую учетную запись, я добавляю соответствующее значение к каждому из них ❷.

Поскольку сейчас мы имеем дело с несколькими счетами, я использую базовую концепцию номера банковского счета. Каждый раз, когда пользователь создает учетную запись, код применяет функцию `len()` в одном из списков и возвращает

этот номер в качестве номера учетной записи пользователя ❸, ❹. При создании аккаунта для первого пользователя длина `accountNamesList` равна нулю. Таким образом, первой созданной учетной записи будет присвоен номер счета 0, второй учетной записи — номер счета 1 и так далее. Затем, как и в реальном банке, для выполнения любой операции после создания счета (например, пополнение или вывод средств) пользователь должен указать номер своего счета ❺.

Однако этот код все еще работает с глобальными данными; теперь это три глобальных списка данных.

Представьте, что вы видите эти данные в виде электронной таблицы. Пример приведен в табл. 1.1.

Таблица 1.1. Таблица наших данных

Номер счета	Имя	Пароль	Баланс
0	Joe	soup	100
1	Mary	nuts	3550
2	Bill	frisbee	1000
3	Sue	xyyzz	750
4	Henry	PW	10 000

Данные поддерживаются в виде трех глобальных списков Python, где каждый список представляет собой столбец в этой таблице. Например, как видно из выделенного столбца, все пароли сгруппированы в один список. Имена пользователей содержатся в другом списке, а остатки на балансе — в третьем. При таком подходе, чтобы получить информацию об одном аккаунте, вам нужно получить доступ к этим спискам с общим значением индекса.

Хотя этот способ работает, он кажется крайне неудобным. Данные не сгруппированы логически. Например, некорректно хранить пароли всех пользователей вместе. Кроме того, каждый раз, когда вы добавляете новый атрибут к учетной записи, например адрес или номер телефона, вам нужно создать и получить доступ к другому глобальному списку.

Вместо этого вы бы наверняка предпочли группировку, которая представляет собой строку в той же электронной таблице, как в табл. 1.2.

Таблица 1.2. Таблица наших данных

Номер счета	Имя	Пароль	Баланс
0	Joe	soup	100
1	Mary	nuts	3550
2	Bill	frisbee	1000
3	Sue	xxyyzz	750
4	Henry	PW	10 000

При таком подходе каждая строка содержит данные, связанные с одним банковским счетом. Хотя это одни и те же данные, такая группировка гораздо более естественно представляет счет.

Реализация 5. Список словарей учетных записей

Чтобы реализовать этот последний подход, я буду использовать немного более сложную структуру данных. Я создам список, где каждый счет (каждый элемент списка) — это словарь, который выглядит так:

```
{ 'name':<someName>, 'password':<somePassword>, 'balance':<someBalance> }
```

ПРИМЕЧАНИЕ Всякий раз, когда вы видите в этой книге значение, представленное в угловых скобках (<>), вы должны заменить этот элемент (включая скобки) на значение по вашему выбору. Например, в предыдущей строке кода <someName>, <somePassword> и <someBalance> — заполнители, их следует заменить фактическими значениями.

Код для окончательной реализации представлен в листинге 1.6.

Файл: Bank5_Dictionary.py

```
# Без ООП
# Банк 5
# Любое количество счетов - со списком словарей

accountsList = [] ❶

def newAccount(aName, aBalance, aPassword):
    global accountsList
```

```

newAccountDict = {'name':aName, 'balance':aBalance,
                  'password':aPassword}
accountsList.append(newAccountDict) ❷

def show(accountNumber):
    global accountsList
    print('Account', accountNumber)
    thisAccountDict = accountsList[accountNumber]
    print('        Name', thisAccountDict['name'])
    print('        Balance:', thisAccountDict['balance'])
    print('        Password:', thisAccountDict['password'])
    print()

def getBalance(accountNumber, password):
    global accountsList
    thisAccountDict = accountsList[accountNumber] ❸
    if password != thisAccountDict['password']:
        print('Incorrect password')
        return None
    return thisAccountDict['balance']

--- скрыты дополнительные функции deposit() и withdraw() ---

# создаем два образца учетных записей
print("Joe's account is account number:", len(accountsList))
newAccount("Joe", 100, 'soup')

print("Mary's account is account number:", len(accountsList))
newAccount("Mary", 12345, 'nuts')

while True:
    print()
    print('Press b to get the balance')
    print('Press d to make a deposit')
    print('Press n to create a new account')
    print('Press w to make a withdrawal')
    print('Press s to show all accounts')
    print('Press q to quit')
    print()

    action = input('What do you want to do? ')
    action = action.lower() # переводим в нижний регистр
    action = action[0] # используем первую букву
    print()

    if action == 'b':
        print('Get Balance:')
        userAccountNumber = input('Please enter your account number: ')

```

```

userAccountNumber = int(userAccountNumber)
userPassword = input('Please enter the password: ')
theBalance = getBalance(userAccountNumber, userPassword)
if theBalance is not None:
    print('Your balance is:', theBalance)

elif action == 'd':
    print('Deposit:')
    userAccountNumber= input('Please enter the account number: ')
    userAccountNumber = int(userAccountNumber)
    userDepositAmount = input('Please enter amount to deposit: ')
    userDepositAmount = int(userDepositAmount)
    userPassword = input('Please enter the password: ')

    newBalance = deposit(userAccountNumber, userDepositAmount,
                          userPassword)
    if newBalance is not None:
        print('Your new balance is:', newBalance)

elif action == 'n':
    print('New Account:')
    userName = input('What is your name? ')
    userStartingAmount = input('What is the amount of your initial
                               deposit? ')
    userStartingAmount = int(userStartingAmount)
    userPassword = input('What password would you like to use for
                          this account? ')

    userAccountNumber = len(accountsList)
    newAccount(userName, userStartingAmount, userPassword)
    print('Your new account number is:', userAccountNumber)

--- скрыт дополнительный пользовательский интерфейс ---

print('Done')

```

Листинг 1.6. Моделирование банка со списком словарей

При таком подходе все данные, связанные с одним счетом, можно найти в одном словаре ❶. Для каждого нового счета мы создаем словарь и добавляем его в список счетов ❷. Каждому счету присваивается номер (простое целое число), который должен быть указан при выполнении любого действия со счетом. Например, пользователь указывает номер своего счета при внесении депозита, а функция `getBalance()` использует его в качестве индекса в списке счетов ❸.

Это немного проясняет дело благодаря более логичной организации данных. Но каждая из функций в программе все равно должна иметь доступ к глобальному списку учетных записей. Как мы увидим в следующем разделе, предоставление функциям доступа ко всем данным учетной записи повышает потенциальные риски безопасности. В идеале каждая функция должна иметь возможность влиять только на данные одной учетной записи.

Общие проблемы с процедурной реализацией

Примеры, приведенные в этой главе, имеют общую проблему: все данные, которыми оперируют функции, хранятся в одной или нескольких глобальных переменных. Использование большого количества глобальных данных с процедурным программированием — плохая практика кодирования по следующим причинам.

1. Любую функцию, которая использует и/или изменяет глобальные данные, сложно повторно использовать в другой программе. Функция, получающая доступ к глобальным данным, работает с данными, которые живут на другом (более высоком) уровне, чем код самой функции. Для доступа к ним ей потребуется инструкция `global`. Вы не можете просто взять функцию, которая полагается на глобальные данные, и повторно использовать ее в другой программе; она может быть повторно использована только в программе с похожими глобальными данными.
2. Многие процедурные программы, как правило, имеют большие коллекции глобальных переменных. По определению глобальная переменная может быть использована или изменена любым фрагментом кода в любом месте программы. Назначения глобальным переменным часто широко разбросаны по процедурным программам как в основном коде, так и внутри функций. Поскольку значения переменных могут изменяться в любом месте, отладка и обслуживание программ, написанных таким образом, чрезвычайно сложны.
3. Функции, написанные для использования глобальных данных, часто имеют доступ к слишком большому объему данных. Когда функция использует глобальный список, словарь или любую

другую глобальную структуру данных, она имеет доступ ко всем данным в этой структуре. Однако, как правило, функция должна работать только на одной части (или только на небольшом количестве) этих данных. Наличие возможности считывать и изменять любые данные в большой структуре может привести к ошибкам, таким как случайное использование или перезапись данных, которые функция не должна была затрагивать.

Объектно-ориентированное решение — первый взгляд на класс

Листинг 1.7 — это объектно-ориентированный подход, который объединяет весь код и связанные с ним данные одного счета. Здесь много новых понятий, и я подробно рассмотрю их начиная со следующей главы. Хотя я не ожидаю, что вы полностью поймете этот пример, обратите внимание, что это комбинация кода и данных в одном скрипте (называемая *классом**). Вот ваш первый взгляд на объектно-ориентированный код.

Файл: Account.py

```
# Класс счета

class Account():
    def __init__(self, name, balance, password):
        self.name = name
        self.balance = int(balance)
        self.password = password

    def deposit(self, amountToDeposit, password):
        if password != self.password:
            print('Sorry, incorrect password')
            return None

        if amountToDeposit < 0:
            print('You cannot deposit a negative amount')
            return None

        self.balance = self.balance + amountToDeposit
        return self.balance
```

* Классом называется не весь скрипт, а именно комбинация кода и данных. — *Прим. науч. ред.*


```

def withdraw(self, amountToWithdraw, password):
    if password != self.password:
        print('Incorrect password for this account')
        return None

    if amountToWithdraw < 0:
        print('You cannot withdraw a negative amount')
        return None

    if amountToWithdraw > self.balance:
        print('You cannot withdraw more than you have in your
            account')
        return None

    self.balance = self.balance - amountToWithdraw
    return self.balance

def getBalance(self, password):
    if password != self.password:
        print('Sorry, incorrect password')
        return None
    return self.balance

# Код для отладки
def show(self):
    print('      Name:', self.name)
    print('      Balance:', self.balance)
    print('      Password:', self.password)
    print()

```

Листинг 1.7. Первый пример класса на Python

А сейчас взгляните на функции и посмотрите, как они похожи на наши предыдущие примеры процедурного программирования. Функции имеют те же имена, что и в предыдущем коде, — `show()`, `getBalance()`, `deposit()` и `withdraw()`, — но вы также увидите еще и слово `self` (или `self.`). О том, что это значит, вы узнаете в следующих главах.

Выводы

Эта глава начинается с процедурной реализации кода для карточной игры под названием «Больше-меньше». В главе 12

я покажу вам, как сделать объектно-ориентированную версию игры с графическим пользовательским интерфейсом.

Далее я представил проблему имитации банка с одним, а затем несколькими банковскими счетами. Я показал несколько различных способов использования процедурного программирования для реализации модели и описал некоторые проблемы, возникающие в результате этого подхода. Наконец, я дал первое представление о том, как будет выглядеть код, описывающий банковский счет, если бы он был создан с помощью класса.

2

МОДЕЛИРОВАНИЕ ФИЗИЧЕСКИХ ОБЪЕКТОВ С ПОМОЩЬЮ ОБЪЕКТНО- ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ



В этой главе мы рассмотрим общие понятия, лежащие в основе объектно-ориентированного программирования. Я покажу простой пример программы, написанной с использованием про-

цедурного программирования, представлю классы как основу для написания кода ООП и объясню, как элементы класса работают вместе. Затем я перепису первый процедурный пример как класс в объектно-ориентированном стиле и покажу, как вы создаете объект из класса. В остальной части главы я пройду через несколько все более сложных классов, представляющих физические объекты, чтобы продемонстрировать, как ООП решает проблемы процедурного программирования, с которыми мы столкнулись в главе 1. Это должно дать вам четкое понимание базовых объектно-ориентированных концепций и того, как они могут улучшить ваши навыки программирования.

Построение программных моделей физических объектов

Чтобы описать физический объект в нашем повседневном мире, мы часто ссылаемся на его атрибуты. Говоря о столе, вы можете описать его цвет, размеры, вес, материал и так далее. Некоторые объекты имеют атрибуты, которые применяются только к ним, а не к другим. Машину можно описать по количеству дверей, а рубашку — нет. Ящик может быть опечатан или открыт, пуст или заполнен, но эти характеристики нельзя применить к деревянному блоку. Кроме того, некоторые объекты способны выполнять действия. Автомобиль может ехать вперед, назад и повернуть влево или вправо.

Чтобы смоделировать реальный объект в коде, нам нужно решить, какие данные будут представлять атрибуты этого объекта и какие операции он может выполнять. Эти два понятия часто называют состоянием и поведением объекта соответственно: состояние — это данные, которые объект запоминает, а поведение — это действия, которые он может совершить.

Состояние и поведение: пример выключателя освещения

Листинг 2.1 — программная модель стандартного двухпозиционного выключателя освещения, написанная на процедурном языке Python. Это тривиальный пример, но он будет демонстрировать состояние и поведение.

Файл: LightSwitch_Procedural.py

Процедурный выключатель света

```
❶ def turnOn():
    global switchIsOn
    # включаем свет
    switchIsOn = True

❷ def turnOff():
    global switchIsOn
    # выключаем свет
    switchIsOn = False

# Основной код
❸ switchIsOn = False # глобальная логическая переменная
```

```
# код теста
print(switchIsOn)
turnOn()
print(switchIsOn)
turnOff()
print(switchIsOn)
turnOn()
print(switchIsOn)
```

Листинг 2.1. Модель выключателя света, написанная процедурным кодом

Переключатель может находиться только в одном из двух положений: «вкл.» или «выкл.». Чтобы смоделировать состояние, нам нужна только одна логическая переменная. Назовем эту переменную `switchIsOn` ❸ и скажем, что `True` означает «вкл.», а `False` — «выкл.». Когда переключатель поступает с завода, он находится в выключенном положении, поэтому мы изначально установили переключатель `IsOn` на `False`.

Далее мы посмотрим на поведение. Этот переключатель может выполнять только два действия: «включить» и «выключить». Поэтому мы строим две функции, `turnOn()` ❶ и `turnOff()` ❷, которые устанавливают значение одной логической переменной равным `True` и `False` соответственно.

Я добавил тестовый код в конце, чтобы включить и выключить переключатель несколько раз. Результат — это именно то, что мы ожидаем:

```
False
True
False
True
```

Это чрезвычайно простой пример, но он, начиная с таких вот небольших функций, упрощает переход к ООП. Как я объяснил в главе 1, поскольку мы использовали глобальную переменную для представления состояния (в данном случае переменную `switchIsOn`), этот код будет работать только для конкретного выключателя освещения, но одна из основных целей написания функций — создать многоразовый код. Поэтому я заново построю код выключателя света, используя объектно-ориентированное программирование, однако сначала мне нужно проработать часть базовой теории.

Введение в классы и объекты

Первым шагом к пониманию того, что такое объект и как он работает, является понимание взаимосвязи между классом и объектом. Я дам формальные определения позже, но на данный момент вы можете думать о классе как о шаблоне или плане, который определяет, как будет выглядеть объект при его создании. Мы создаем объекты из класса.

Представьте, что мы начали выпечку тортов по требованию. То есть мы делаем торт только тогда, когда поступает заказ на него. Мы специализируемся на тортах Bundt и потратили много времени на разработку формы как на рис. 2.1, чтобы убедиться, что наши торты не только вкусные, но и красивые.

Форма определяет, как будет выглядеть торт Bundt, когда мы создадим его, но она, конечно, не сам торт. Форма символизирует класс. Когда поступает заказ, мы создаем торт Bundt из нашей формы (рис. 2.2). Торт — это предмет, изготовленный с применением формы.

Используя ее, мы можем создать любое количество тортов. Они могут иметь разные атрибуты: различные вкусы, различные виды глазури и дополнительные опции, такие как шоколадные чипсы, но все торты будут одной и той же формы.



Рис. 2.1. Форма для торта как метафора для класса



Рис. 2.2. Торт как метафора для объекта, сделанного из класса формы

В табл. 2.1 приведены некоторые другие примеры из реального мира, которые помогают прояснить связь между классом и объектом.

Таблица 2.1. Примеры реальных классов и объектов

Класс	Объект, созданный из класса
Чертеж дома	Дом
Сэндвич в меню	Сэндвич в вашей руке
Штамп, используемый для изготовления 25-центовой монеты	Четвертак
Рукопись книги, написанной автором	Физическая или электронная копия книги

Классы, объекты и экземпляры

Давайте посмотрим, как это работает в коде.

Класс

Код, который определяет, что объект запомнит (его данные или состояние) и что он сможет сделать (его функции или поведение).

Чтобы получить представление о том, как выглядит класс, вот код выключателя света, записанный как класс:

```
# OO_LightSwitch

class LightSwitch():
    def __init__(self):
        self.switchIsOn = False

    def turnOn(self):
        # включаем переключатель
        self.switchIsOn = True

    def turnOff(self):
        # выключаем переключатель
        self.switchIsOn = False
```

Мы пройдемся по деталям чуть позже, но стоит заметить, что этот код определяет одну переменную, `self.switchIsOn`, которая инициализирована в одной функции и содержит две другие, определяющие поведение: `turnOn()` и `turnOff()`.

Если вы пишете код класса и пытаетесь запустить его, ничего не происходит, так же как и при запуске программы на Python, которая состоит только из функций и не вызывает их. Необходимо явно приказать Python создать объект из класса.

Чтобы создать объект `LightSwitch` из нашего класса `LightSwitch`, мы обычно используем такую строку:

```
oLightSwitch = LightSwitch()
```

Здесь говорится: найдите класс `LightSwitch`, создайте объект `LightSwitch` из этого класса и назначьте полученный объект переменной `oLightSwitch`.

ПРИМЕЧАНИЕ В этой книге для обозначения переменной, которая представляет объект, я, как правило, использую префикс в нижнем регистре `o`. Это не обязательно, но помогает напомнить себе, что переменная представляет объект.

Другое слово, которое вы найдете в ООП, — *экземпляр* (*instance*). Слова *экземпляр* и *объект* по существу взаимозаменяемы; однако, если быть точными, мы бы сказали, что объект `LightSwitch` является экземпляром класса `LightSwitch`.

Инстанцирование

Процесс создания объекта из класса.

В предыдущей инструкции назначения мы прошли процесс инстанцирования для создания объекта `LightSwitch` из класса `LightSwitch`. Мы также можем обозначить этот процесс как глагол; мы *инстанцируем* `LightSwitch` из класса `LightSwitch`.

Написание класса на Python

Обсудим различные части класса и детали создания экземпляров и использования объекта. В листинге 2.2 показана общая форма класса на Python.

```
class <ClassName>():

    def __init__(self, <optional param1>, ..., <optional paramN>):
        # код инициализации

    # Функции, которые обращаются к данным
    # Каждая имеет вид:

    def <functionName1>(self, <optional param1>, ..., <optional paramN>):
        # тело функции

    # ...другие функции

    def <functionNameN>(self, <optional param1>, ..., <optional paramN>):
        # тело функции
```

Листинг 2.2. Типичная форма класса на Python

Определение класса начинается с оператора `class`, указывающего имя, которое вы хотите дать классу. Для таких названий принято использовать горбатый регистр с первой заглавной буквой (например, `LightSwitch`). После имени вы можете добавить набор скобок, но оператор должен заканчиваться двоеточием, чтобы указать, что вы собираетесь начать тело класса. (Я объясню, что может быть в скобках, в главе 10, когда мы будем обсуждать наследование.)

В теле класса можно определить любое количество функций. Все функции считаются частью класса, и код, который их определяет, должен начинаться отступом. Каждая функция представляет определенное поведение, которое может выполнять объект, созданный из класса. Все функции должны иметь хотя бы один параметр, который по соглашению называется `self` (я объясню, что означает это имя, в главе 3). Функциям ООП дано специальное название: *метод*.

Метод

Функция, определенная внутри класса. Метод всегда имеет хотя бы один параметр, который обычно называется `self`.

Первый метод в каждом классе должен иметь специальное имя `__init__`. Всякий раз, когда вы создаете объект из класса, этот метод будет вызываться автоматически. Поэтому он — логичное место для размещения любого кода инициализации, который вы хотите запустить, когда создаете экземпляр объекта из класса. Имя `__init__` зарезервировано Python для этой самой задачи и должно выглядеть именно так, с двумя подчеркиваниями до и после слова `init` (которое пишется строчными буквами). На самом деле метод `__init__()` не является обязательным. Однако, как правило, считается хорошей практикой включать его и использовать для инициализации.

ПРИМЕЧАНИЕ Когда вы создаете экземпляр объекта из класса, Python заботится о создании объекта (выделении памяти) за вас. Специальный метод `__init__()` называется методом *initializer*, где вы задаете переменным начальные значения. (Для большинства других языков ООП требуется метод с именем `new()`, который часто называют конструктором.)

Область видимости и переменные экземпляра

В процедурном программировании существует два основных уровня области видимости: переменные, созданные в основном коде, имеют *глобальную* область видимости и доступны в любом месте программы, в то время как переменные, созданные внутри функций, имеют *локальную* область видимости и живут только до тех пор, пока выполняется функция. При выходе из функции все локальные переменные (переменные с локальной областью видимости) буквально исчезают.

Объектно-ориентированное программирование и классы представляют третий уровень области видимости, обычно называемый *областью видимости объекта*, иногда *областью видимости класса* или, реже, *областью видимости экземпляра*. Все они означают одно и то же: область видимости состоит из всего кода внутри определения класса.

Методы могут использовать как локальные переменные, так и *переменные экземпляра*. В методе любая переменная, имя которой не начинается с `self`, является локальной переменной и исчезает при выходе из него, то есть другие методы в классе больше не могут ее использовать. *Переменные экземпляра* имеют область видимости объекта, то есть они доступны всем методам, определенным в классе. Переменные экземпляра и область видимости объекта — это ключи к пониманию того, как объекты запоминают данные.

Переменная экземпляра

В методе любая переменная, имя которой начинается, по соглашению, с префикса `self`. (например, `self.x`). Переменные экземпляра имеют область видимости объекта.

Подобно локальным и глобальным переменным, переменные экземпляра создаются, когда им в первый раз присваивается значение, и не требуют специального объявления. Метод `__init__()` — логичное место для инициализации переменных экземпляра. Вот пример класса, где метод `__init__()` инициализирует переменную экземпляра `self.count` (читается как «`self dot count`») в ноль, и другой метод, `increment()`, который просто добавляет 1 к `self.count`:

```
class MyClass():
    def __init__(self):
        self.count = 0 # создаем self.count и установим ее на 0
    def increment(self):
        self.count = self.count + 1 # инкремент переменной
```

При создании экземпляра объекта из класса `MyClass` запускается метод `__init__()` со значением переменной экземпляра `self.count`, равным нулю. При вызове метода `increment()` значение `self.count` изменяется с нуля на единицу. При повторном вызове `increment()` значение изменяется с одного на два и так далее.

Каждый объект, созданный из класса, получает свой набор переменных экземпляра независимо от любых других объектов, созданных из этого класса. В случае класса `LightSwitch` существует только одна переменная экземпляра, `self.switchIsOn`, поэтому каждый объект `LightSwitch` будет иметь собственный `self.switchIsOn`. Таким образом, вы можете иметь несколько объектов `LightSwitch`, каждый из которых имеет свое независимое значение `True` или `False` для своей переменной `self.switchIsOn`.

Различия между функциями и методами

Подводя итог, можно отметить три ключевых различия между функцией и методом.

1. Все методы класса должны иметь отступы под инструкцией `class`.
2. Все методы имеют специальный первый параметр, который (так принято) называется `self`.
3. Методы в классе могут использовать переменные экземпляра, записанные в виде `self.<variableName>`.

Теперь, когда вы знаете, что такое методы, я покажу вам, как создать объект из класса и как использовать различные методы, доступные в классе.

ПРОЦЕСС СОЗДАНИЯ ЭКЗЕМПЛЯРА

На рис. 2.3 показаны этапы создания экземпляра объекта `LightSwitch` из класса `LightSwitch`, начинающегося в инструкции присваивания в Python, затем переходящего в код класса, затем обратно через Python и, наконец, обратно в инструкцию присваивания.

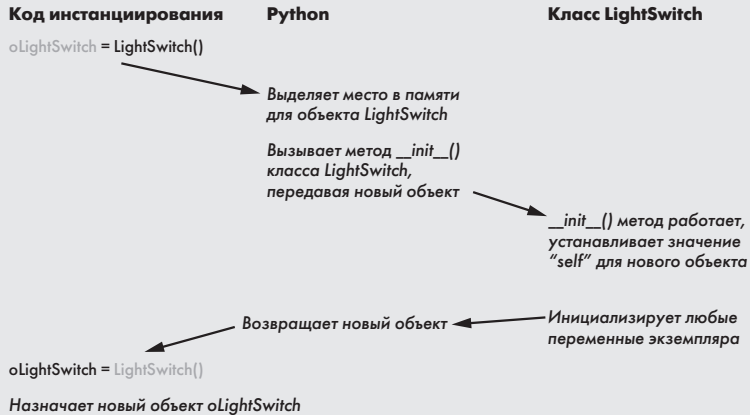


Рис. 2.3. Процесс создания экземпляра объекта

Процесс состоит из пяти этапов.

1. Наш код просит Python создать объект из класса `LightSwitch`.
2. Python выделяет место в памяти для объекта `LightSwitch`, затем вызывает метод `__init__()` класса `LightSwitch`, передавая вновь созданный объект.
3. Метод `__init__()` класса `LightSwitch` выполняется. Новый объект присваивается параметру `self`. Код `__init__()` инициализирует любые переменные экземпляра в объекте (в данном случае переменную экземпляра `self.switchIsOn`).
4. Python возвращает новый объект исходному вызывающему.
5. Результат исходного вызова присваивается переменной `oLightSwitch`, поэтому теперь она представляет объект.

Создание объекта из класса

Как я уже говорил, класс просто определяет, как будет выглядеть объект. Чтобы использовать класс, вы должны сказать Python создать объект из класса. Типичный способ сделать это — использовать оператор присваивания, как тут:

```
<object> = <ClassName>(<optional arguments>)
```

Эта единственная строка кода вызывает последовательность шагов, которая заканчивается тем, что Python возвращает вам новый экземпляр класса, который вы обычно храните в переменной. Эта переменная ссылается на результирующий объект.

Вы можете сделать класс доступным двумя способами: поместить код класса в один файл с основной программой или во внешний файл и использовать оператор импорта, чтобы подключить содержимое файла. В этой главе я покажу первый подход, а второй — в главе 4. Единственное правило заключается в том, что определение класса должно предшествовать любому коду, который создает экземпляр объекта из класса.

Вызов методов объекта

После создания объекта из класса для вызова метода объекта используется обобщенный синтаксис:

```
<object>.<methodName>(<any arguments>)
```

В листинге 2.3 показан класс `LightSwitch`, код для создания экземпляра объекта из класса и код для включения и выключения объекта `LightSwitch` путем вызова его методов `turnOn()` и `turnOff()`.

Файл: `OO_LightSwitch_with_Test_Code.py`

```
# OO_LightSwitch
class LightSwitch():
    def __init__(self):
        self.switchIsOn = False

    def turnOn(self):
        # включаем выключатель
        self.switchIsOn = True
```

```

def turnOff(self):
    # выключаем выключатель
    self.switchIsOn = False

def show(self): # добавлено для тестирования
    print(self.switchIsOn)

# Основной код
oLightSwitch = LightSwitch() # создаем объект LightSwitch

# Вызовы методов
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
oLightSwitch.turnOff()
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()

```

Листинг 2.3. Класс `LightSwitch` и тестовый код для создания объекта и вызова его методов

Сначала мы создаем объект `LightSwitch` и назначаем его переменной `oLightSwitch`. Затем мы используем эту переменную для вызова других методов, доступных в классе `LightSwitch`. Мы бы прочитали эти строки как «`oLightSwitch dot show`», «`oLightSwitch dot turnOn`» и так далее. Если запустим этот код, он выведет следующее:

```

False
True
False
True

```

Напомним, что этот класс имеет одну переменную экземпляра с именем `self.switchIsOn`, но ее значение запоминается и легкодоступно, когда выполняются разные методы одного и того же объекта.

Создание нескольких экземпляров из одного класса

Одной из ключевых особенностей ООП является то, что вы можете создать сколько угодно объектов из одного класса, так же как вы можете испечь бесконечное количество тортов в форме Bundt.

Итак, если вам нужно два объекта переключения света, или три, или больше, вы можете просто создать дополнительные объекты из класса `LightSwitch` примерно так:

```
oLightSwitch1 = LightSwitch() # создаем объект LightSwitch
oLightSwitch2 = LightSwitch() # создаем еще один объект LightSwitch
```

Важно то, что каждый объект, который вы создаете из класса, поддерживает *свою собственную версию* данных. В этом случае каждый из `oLightSwitch1` и `oLightSwitch2` имеет свою переменную экземпляра, `self.switchIsOn`. Любые изменения, внесенные в данные одного объекта, не повлияют на данные другого. Вы можете вызвать любой из методов в классе с помощью любого объекта. Пример в листинге 2.4 создает два объекта переключателей света и вызывает методы для разных объектов.

Файл: `OO_LightSwitch_Two_Instances.py`

```
# OO_LightSwitch

class LightSwitch():
    --- пропущенный код класса LightSwitch, как в листинге 2-3 ---

    # Основной код
    oLightSwitch1 = LightSwitch() # создаем объект LightSwitch
    oLightSwitch2 = LightSwitch() # создаем еще один объект LightSwitch

    # код теста
    oLightSwitch1.show()
    oLightSwitch2.show()
    oLightSwitch1.turnOn() # Переключатель 1 включен
    # Переключатель 2 должен быть выключен при запуске,
    # этот код делает это более очевидным
    oLightSwitch2.turnOff()
    oLightSwitch1.show()
    oLightSwitch2.show()
```

Листинг 2.4. Создайте два экземпляра класса и вызовите методы каждого

Вот вывод, который вы получите при запуске программы:

```
False
False
True
False
```

Код предписывает `oLightSwitch1` включить себя и предписывает `oLightSwitch2` выключить себя. Обратите внимание, что код в классе не имеет глобальных переменных. Каждый объект `LightSwitch` получает собственный набор любых переменных экземпляра (в данном случае только одну), определенных в классе.

Хотя это может показаться небольшим улучшением по сравнению с наличием двух простых глобальных переменных, которые могут быть использованы для одного и того же, последствия этого изменения огромны. Вы получите лучшее представление об этом в главе 4, где я буду рассказывать, как создавать и поддерживать большое количество экземпляров, созданных из класса.

Типы данных Python реализованы как классы

Возможно, вас не удивит, что все встроенные типы данных в Python реализованы как классы. Можно привести простой пример:

```
>>> myString = 'abcde'
>>> print(type(myString))
<class 'str'>
```

Мы присваиваем переменной строковое значение. Когда вызываем функцию `type()` и выводим результаты, мы видим, что у нас есть экземпляр класса `str` `string`. Класс `str` дает нам ряд методов, которые можно вызвать для строк, включая `myString.upper()`, `myString.lower()`, `myString.strip()` и так далее.

Списки работают аналогичным образом:

```
>>> myList = [10, 20, 30, 40]
>>> print(type(myList))
<class 'list'>
```

Все списки являются экземплярами класса `list`, который имеет множество методов, включая `myList.append()`, `myList.count()`, `myList.index()` и так далее.

Когда пишете класс, вы определяете новый тип данных. Ваш код предоставляет подробную информацию, показывая, какие данные он хранит и какие операции может выполнять. После создания экземпляра вашего класса и назначения его

переменной вы можете использовать встроенную функцию `type()` для определения класса, используемого для создания экземпляра, так же как со встроенным типом данных. Здесь мы создаем экземпляр объекта `LightSwitch` и выводим его тип данных:

```
>>> oLightSwitch = LightSwitch()
>>> print(type(oLightSwitch))
<class 'LightSwitch'>
```

Как и со встроенными типами данных Python, мы можем использовать переменную `oLightSwitch` для вызова методов, доступных в классе `oLightSwitch`.

Определение объекта

Подводя итог этому разделу, я дам свое официальное определение *объекту*.

Объект

Данные плюс код, который действует на них с течением времени.

Класс определяет, как будет выглядеть объект при создании экземпляра. Объект — это набор переменных экземпляра и код методов в классе, из которого был создан экземпляр объекта. Любое количество объектов может быть инстанцировано из класса, и каждый из них имеет собственный набор переменных экземпляра. При вызове метода объекта метод запускается и использует набор переменных экземпляра в этом объекте.

Создание несколько более сложного класса

Давайте построим на концепции, представленной ранее, второй, немного более сложный пример, в котором мы сделаем класс выключателя-диммера. Выключатель-диммер имеет переключатель включить/выключить, но также у него есть мультипозиционный ползунок, который влияет на яркость света.

Ползунок может перемещаться по диапазону значений яркости. Для простоты наш цифровой ползунок диммера имеет 11 положений, от 0 (полностью выключено) до 10 (полностью включено). Чтобы увеличить или уменьшить яркость лампы

в максимальной степени, вы должны перемещать ползунок через каждую возможную настройку.

Класс `DimmerSwitch` обладает большей функциональностью, чем класс `LightSwitch`, и должен запоминать больше данных:

- состояние переключателя (включено или выключено);
- уровень яркости (от 0 до 10).

А вот поведение объекта `DimmerSwitch`:

- включить;
- выключить;
- увеличить яркость;
- уменьшить яркость;
- вывести яркость (для отладки).

Класс `DimmerSwitch` использует стандартный шаблон, показанный ранее в листинге 2.2: он начинается с оператора класса и первого метода с именем `__init__()`, а затем определяет ряд дополнительных методов, по одному для каждого из перечисленных действий. Полный код для этого класса представлен в листинге 2.5.

Файл: `DimmerSwitch.py`

```
# Класс DimmerSwitch

class DimmerSwitch():
    def __init__(self):
        self.switchIsOn = False
        self.brightness = 0

    def turnOn(self):
        self.switchIsOn = True

    def turnOff(self):
        self.switchIsOn = False

    def raiseLevel(self):
        if self.brightness < 10:
            self.brightness = self.brightness + 1

    def lowerLevel(self):
        if self.brightness > 0:
            self.brightness = self.brightness - 1
```

```
# Дополнительный метод для отладки
def show(self):
    print('Switch is on?', self.switchIsOn)
    print('Brightness is:', self.brightness)
```

Листинг 2.5. Немного более сложный класс DimmerSwitch

В этом методе `__init__()` мы имеем две переменные экземпляра: знакомую `self.switchIsOn` и новую, `self.brightness`, которая запоминает уровень яркости. Мы присваиваем начальные значения обоим переменным экземпляра. Все остальные методы могут получить доступ к текущему значению каждой из них. В дополнение к `turnOn()` и `turnOff()` мы включаем два новых метода для этого класса: `raiseLevel()` и `lowerLevel()`, — которые делают именно то, что подразумевают их имена. Метод `show()` используется во время разработки и отладки и просто выводит текущие значения переменных экземпляра.

Основной код в листинге 2.6 тестирует наш класс, создавая объект `DimmerSwitch` (`oDimmer`), затем вызывая различные методы.

Файл: `OO_DimmerSwitch_with_Test_Code.py`

```
# Класс DimmerSwitch с тестовым кодом

class DimmerSwitch():
    --- скрыт фрагмент кода класса DimmerSwitch, как
    в листинге 2.5 ---

    # Основной код
    oDimmer = DimmerSwitch()

    # включаем переключатель и поднимаем уровень яркости 5 раз
    oDimmer.turnOn()
    oDimmer.raiseLevel()
    oDimmer.raiseLevel()
    oDimmer.raiseLevel()
    oDimmer.raiseLevel()
    oDimmer.raiseLevel()
    oDimmer.show()

    # Уменьшаем уровень яркости 2 раза и выключаем переключатель
    oDimmer.lowerLevel()
    oDimmer.lowerLevel()
```

```
oDimmer.turnOff()  
oDimmer.show()  
  
# включаем переключатель и поднимаем уровень яркости 3 раза  
oDimmer.turnOn()  
oDimmer.raiseLevel()  
oDimmer.raiseLevel()  
oDimmer.raiseLevel()  
oDimmer.show()
```

Листинг 2.6. Класс `DimmerSwitch` с тестовым кодом

Когда мы запускаем этот код, в результате получаем вывод:

```
Switch is on? True  
Brightness is: 5  
Switch is on? False  
Brightness is: 3  
Switch is on? True  
Brightness is: 6
```

Основной код создает объект `oDimmer`, затем вызывает различные методы. Каждый раз, когда мы вызываем метод `show()`, состояние включения/выключения и уровень яркости выводятся на экран. Здесь важно помнить, что `oDimmer` представляет объект. Он разрешает доступ ко всем методам в классе, из которого он был создан (класс `DimmerSwitch`), и он имеет набор всех переменных экземпляра, определенных в классе (`self.switchIsOn` и `self.brightness`). Опять же, переменные экземпляра сохраняют свои значения между вызовами методов объекта, поэтому переменная экземпляра `self.brightness` увеличивается на 1 для каждого вызова `oDimmer.raiseLevel()`.

Представление более сложного физического объекта как класса

Рассмотрим более сложный физический объект — телевизор. В этом примере мы рассмотрим подробнее, как работают аргументы в классах.

Телевизору требуется гораздо больше данных, чем выключателю света, чтобы представлять его состояние, и у него больше действий. Чтобы создать класс телевизора, мы должны подумать о том, как обычно используется телевизор и что он должен

помнить. Рассмотрим некоторые важные кнопки на типичном пульте дистанционного управления телевизором (рис. 2.4).

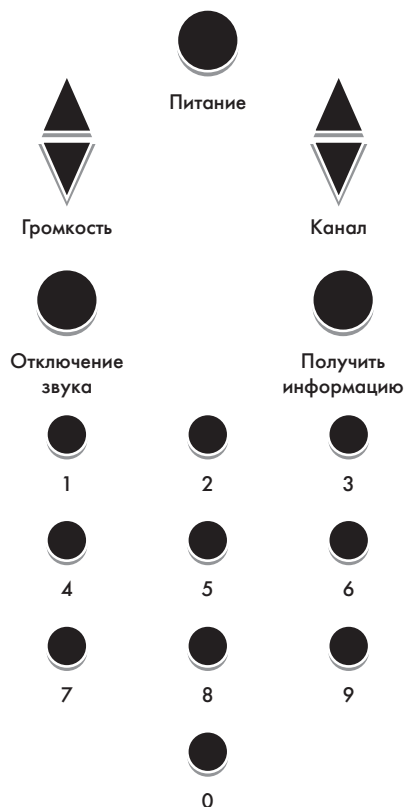


Рис. 2.4. Упрощенный пульт от телевизора

Исходя из этого, мы можем определить, что для отслеживания своего состояния класс ТВ должен будет поддерживать следующие данные:

- состояние питания (вкл. или выкл.);
- состояние отключения звука (отключено?);
- список доступных каналов;
- текущая настройка канала;
- текущая настройка громкости;
- диапазон доступных уровней громкости.

А действия, которые должен обеспечить телевизор, включают в себя:

- включение и выключение питания;
- повышение и понижение громкости;

- переключение каналов вверх и вниз;
- выключение/включение звука;
- получение информации о текущих настройках;
- переход к указанному каналу.

Код для нашего класса телевизора показан в листинге 2.7. Мы включаем метод инициализации `__init__()`, за которым следует метод для каждого действия.

Файл: TV.py

```
# класс TV
class TV():
    def __init__(self): ❶
        self.isOn = False
        self.isMuted = False
        # Некий список каналов по умолчанию
        self.channelList = [2, 4, 5, 7, 9, 11, 20, 36, 44, 54, 65]
        self.nChannels = len(self.channelList)
        self.channelIndex = 0
        self.VOLUME_MINIMUM = 0 # константа
        self.VOLUME_MAXIMUM = 10 # константа
        self.volume = self.VOLUME_MAXIMUM // # целочисленная переменная

    def power(self): ❷
        self.isOn = not self.isOn # переключатель

    def volumeUp(self):
        if not self.isOn:
            return
        if self.isMuted:
            self.isMuted = False # изменение громкости включает звук,
                                # если тот отключен
        if self.volume < self.VOLUME_MAXIMUM:
            self.volume = self.volume + 1

    def volumeDown(self):
        if not self.isOn:
            return
        if self.isMuted:
            self.isMuted = False # изменение громкости включает звук,
                                # если тот отключен
        if self.volume > self.VOLUME_MINIMUM:
            self.volume = self.volume - 1

    def channelUp(self): ❸
```



```

    if not self.isOn:
        return
    self.channelIndex = self.channelIndex + 1
    if self.channelIndex > self.nChannels:
        self.channelIndex = 0 # после последнего канала вернуться
                               # к первому каналу

def channelDown(self): ❹
    if not self.isOn:
        return
    self.channelIndex = self.channelIndex - 1
    if self.channelIndex < 0:
        self.channelIndex = self.nChannels - 1 # перед первым
                                                # каналом - последний

def mute(self): ❺
    if not self.isOn:
        return
    self.isMuted = not self.isMuted

def setChannel(self, newChannel):
    if newChannel in self.channelList:
        self.channelIndex = self.channelList.index(newChannel)
    # если newChannel нет в нашем списке каналов, ничего не делать

def showInfo(self): ❻
    print()
    print('TV Status:')
    if self.isOn:
        print('  TV is: On')
        print('  Channel is:', self.channelList[self.channelIndex])
        if self.isMuted:
            print('  Volume is:', self.volume, '(sound is muted)')
        else:
            print('  Volume is:', self.volume)
    else:
        print('  TV is: Off')

```

Листинг 2.7. Класс телевизора со множеством переменных экземпляров и методов

Метод `__init__()` ❶ создает все переменные экземпляра, используемые во всех методах, и присваивает каждой из них разумные начальные значения. Технически вы можете создать переменную экземпляра внутри любого метода, однако хорошая практика программирования — определять все переменные

экземпляра в методе `__init__()`. Это позволяет избежать риска ошибки при попытке использовать переменную экземпляра в методе до того, как она была определена.

Метод `power()` ❷ показывает, что происходит при нажатии кнопки питания на пульте управления. Если телевизор выключен, нажатие кнопки питания включает его; если телевизор включен, нажатие кнопки питания выключает его. Для кодирования этого поведения я использовал *переключатель*, который является логической переменной, используется для представления одного из двух состояний и может легко переключаться между ними. С помощью этого переключателя оператор `not` переключает переменную `self.isOn` с `True` на `False` или с `False` на `True`. Код метода `mute()` ❸ выполняет аналогичные действия, при этом переменная `self.muted` переключается между выключенным и включенным звуком, но сначала должна проверить, что телевизор включен. Если телевизор выключен, вызов метода `mute()` не имеет никакого эффекта.

Интересно отметить, что мы не следим за текущим каналом. Вместо этого мы отслеживаем индекс текущего канала, который позволяет нам получить текущий канал в любое время с помощью `self.channelList[self.channelIndex]`.

Методы `channelUp()` ❹ и `channelDown()` ❺ в основном увеличивают и уменьшают индекс канала, но в них также есть некоторый умный код, позволяющий идти по кругу. Если вы находитесь на последнем индексе в списке каналов и пользователь просит перейти к следующему каналу вверх, телевизор переходит к первому каналу в списке. Если вы находитесь на первом индексе в списке каналов и пользователь просит перейти к следующему каналу вниз, телевизор переходит к последнему каналу в списке.

Метод `showInfo()` ❻ выводит текущее состояние телевизора на основе значений переменных экземпляра (вкл./выкл., текущий канал, текущая настройка громкости и настройка отключения звука).

В листинге 2.8 мы создадим объект телевизора и вызовем методы этого объекта.

Файл: OO_TV_with_Test_Code.py

```
# Класс TV с тестовым кодом

--- фрагмент кода класса TV, как в листинге 2.7 ---

# Основной код
oTV = TV() # создаем ТВ-объект

# включаем телевизор и показываем статус
oTV.power()
oTV.showInfo()

# Дважды меняем канал, дважды увеличиваем громкость, показываем статус
oTV.channelUp()
oTV.channelUp()
oTV.volumeUp()
oTV.volumeUp()
oTV.showInfo()

# Выключаем телевизор, показываем статус, включаем телевизор,
# показываем статус
oTV.power()
oTV.showInfo()
oTV.power()
oTV.showInfo()

# Убавляем громкость, отключаем звук, показываем состояние
oTV.volumeDown()
oTV.mute()
oTV.showInfo()

# Переключаем канал на 11, отключаем звук, показываем состояние
oTV.setChannel(11)
oTV.mute()
oTV.showInfo()
```

Листинг 2.8. Класс телевизора с тестовым кодом

Когда запускаем этот код, вот что мы получаем в качестве вывода:

```
TV Status:
    TV is: On
    Channel is: 2
    Volume is: 5
```

```
TV Status:
  TV is: On
  Channel is: 5
  Volume is: 7

TV Status:
  TV is: Off

TV Status:
  TV is: On
  Channel is: 5
  Volume is: 7

TV Status:
  TV is: On
  Channel is: 5
  Volume is: 6 (sound is muted)

TV Status:
  TV is: On
  Channel is: 11
  Volume is: 6
```

Все методы работают корректно, и мы получаем ожидаемый результат.

Передача аргументов методу

При вызове любой функции количество аргументов должно соответствовать количеству параметров, перечисленных в операторе `def`:

```
def myFunction(param1, param2, param3):
    # тело функции

# вызов функции:
myFunction(argument1, argument2, argument3)
```

То же правило применяется к методам и вызовам методов. Тем не менее вы можете заметить, что всякий раз, когда мы делаем вызов методу, кажется, что мы указываем на один аргумент меньше, чем количество параметров. Например, определение метода `power()` в нашем телевизионном классе выглядит следующим образом:

```
def power(self):
```

Это означает: метод `power()` ожидает, что будет передано одно значение и все переданное будет присвоено переменной `self`. Тем не менее, когда мы начали с включения телевизора в листинге 2.8, мы сделали такой вызов:

```
oTV.power()
```

Когда мы делаем вызов, мы не передаем ничего явно внутри скобок.

Это может показаться еще более странным в случае метода `setChannel()`. Метод написан так, что принимает два параметра:

```
def setchannel(self, newchannel):  
    if newChannel in self.channelList:  
        self.channelIndex = self.channelList.index(newChannel)
```

Но мы вызвали `setChannel()` следующим образом:

```
oTV.setChannel(11)
```

Похоже, что передается только одно значение.

Вы можете ожидать, что Python сгенерирует здесь ошибку из-за несоответствия в количестве переданных аргументов (один) и количестве ожидаемых параметров (два). На практике Python делает небольшую закулисную работу, чтобы облегчить синтаксис.

Давайте рассмотрим это подробнее. Ранее я говорил, что для вызова метода объекта вы используете следующий общий синтаксис:

```
<object>.<method>(<any arguments>)
```

Python берет `<object>`, указанный в вызове, и подставляет его в качестве первого аргумента. Любые значения в скобках вызова метода считаются последующими аргументами. Таким образом Python делает вид, что вы написали это вместо своего вызова:

```
<method of object>(<object>, <any arguments>)
```

На рис. 2.5 показано, как это работает в нашем примере кода, опять же с помощью метода `setChannel()` класса `TV`.

```
# Метод в классе TV
def setChannel(self, newChannel):
    ...

# Вызов
oTV.setChannel(11)
```

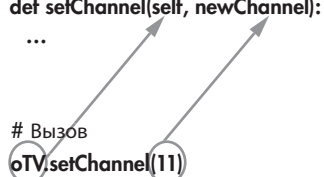


Рис. 2.5. Вызов метода

Хотя похоже, что мы предоставляем только один аргумент (для `newChannel`), на самом деле передано два аргумента: `oTV` и `11`, — и метод предоставляет два параметра для получения этих значений (`self` и `newChannel` соответственно). Python меняет за нас порядок аргументов при вызове. Поначалу это может показаться странным, но очень быстро войдет в привычку. Запись вызова с помощью объекта в первую очередь значительно упрощает для программиста определение того, на какой объект выполняется действие.

Это тонкая, но важная особенность. Помните, что объект (в данном случае `oTV`) сохраняет текущие настройки всех своих переменных экземпляра. Передача объекта в качестве первого аргумента позволяет методу работать со значениями переменных экземпляра этого объекта.

Несколько экземпляров

Каждый метод записывается с собой в качестве первого параметра, поэтому переменная `self` получает объект, используемый в каждом вызове. Это имеет большое значение: оно позволяет любому методу в классе работать с *различными* объектами. Я объясню, как это работает, используя пример.

В листинге 2.9 мы создадим два объекта телевизора и сохраним их в двух переменных, `oTV1` и `oTV2`. Каждый телевизионный объект имеет настройку громкости, список каналов, настройку канала и так далее. Мы будем вызывать несколько разных методов для разных объектов. В конце мы вызовем метод `showInfo()` для каждого объекта `TV`, чтобы увидеть результирующие настройки.

Файл: OO_TV_TwoInstances.py

```
# Два объекта TV с вызовами к их методам
class TV():

    --- фрагмент кода класса TV, как в листинге 2.7 ---

    # Основной код
    oTV1 = TV() # создаем один объект TV
    oTV2 = TV() # создаем еще один объект TV

    # включаем оба телевизора
    oTV1.power()
    oTV2.power()

    # увеличиваем громкость TV1
    oTV1.volumeUp()
    oTV1.volumeUp()

    # увеличиваем громкость TV2
    oTV2.volumeUp()
    oTV2.volumeUp()
    oTV2.volumeUp()
    oTV2.volumeUp()
    oTV2.volumeUp()

    # Переключаем канал TV2, затем отключаем звук
    oTV2.setChannel(44)
    oTV2.mute()

    # Теперь отображаем состояние обоих телевизоров
    oTV1.showInfo()
    oTV2.showInfo()
```

Листинг 2.9. Создание двух экземпляров класса TV и вызов методов каждого

Если мы запустим этот код, он сгенерирует следующий вывод:

```
Status of TV:
    TV is: On
    Channel is: 2
    Volume is: 7

Status of TV:
    TV is: On
    Channel is: 44
    Volume is: 10 (sound is muted)
```

Каждый объект TV поддерживает собственный набор переменных экземпляра, определенных в классе. Таким образом, переменными экземпляра каждого объекта TV можно манипулировать независимо от переменных любого другого объекта TV.

Параметры инициализации

Возможность передавать аргументы вызовам метода также работает при создании экземпляра объекта. До сих пор, когда мы создавали наши объекты, мы всегда устанавливали переменные их экземпляров на постоянные значения. Однако часто нужно создавать разные объекты с разными стартовыми значениями. Например, представьте, что мы хотим сделать экземпляры различных телевизоров и идентифицировать их по их бренду и местоположению. Таким образом, мы можем различать телевизор Samsung в гостиной и телевизор Sony в спальне. Константы не сработали бы для нас в этой ситуации.

Чтобы инициализировать объект с разными значениями, мы добавляем параметры в определение метода `__init__()`, например вот так:

```
# класс TV

class TV():
    def __init__(self, brand, location): # передаем бренд
                                         # и расположение телевизора
        self.brand = brand
        self.location = location
        --- оставшаяся инициализация телевизора ---
        ...
```

Во всех методах параметры являются локальными переменными, поэтому они буквально исчезают, когда метод заканчивается. Например, в методе `__init__()` показанного здесь класса TV бренд и местоположение — локальные переменные, которые исчезают по завершении метода. Однако мы часто хотим сохранить значения, передающиеся через параметры, чтобы использовать их в других методах.

Для того чтобы объект запомнил начальные значения, стандартный подход заключается в сохранении любых значений, переданных в переменные экземпляра. Поскольку переменные экземпляра имеют область видимости объекта, их можно использовать в других методах класса. Соглашение Python

заключается в том, что имя переменной экземпляра должно совпадать с именем параметра, но с префиксом `self` и точкой:

```
def __init__(self, someVariableName):  
    self.someVariableName = someVariableName
```

В классе `TV` строка после оператора `def` говорит Python взять значение параметра `brand` и назначить его переменной экземпляра с именем `self.brand`. Следующая строка делает то же самое с параметром `location` и переменной экземпляра `self.location`. После этих назначений мы можем использовать `self.brand` и `self.location` в других методах.

Используя этот подход, мы можем создать несколько объектов из одного класса, но начать каждый с разных данных. Таким образом мы можем создать два наших телевизионных объекта, как тут:

```
oTV1 = TV('Samsung', 'Family room')  
oTV2 = TV('Sony', 'Bedroom')
```

При выполнении первой строки Python сначала выделяет место для объекта `TV`. Затем он переупорядочивает аргументы, как обсуждалось в предыдущем разделе, и вызывает метод `__init__()` класса `TV` с тремя аргументами: вновь выделенным объектом `oTV1`, брендом и местоположением.

При инициализации объекта `oTV1` для `self.brand` задается строка `'Samsung'`, а для `self.location` — строка `'Family room'`. При инициализации `oTV2` его переменной `self.brand` присваивается строка `'Sony'`, а его `self.location` присваивается строка `'Bedroom'`.

Можно изменить метод `showInfo()`, чтобы сообщить модель и местоположение телевизора.

Файл: `OO_TV_TwoInstances_with_Init_Params.py`

```
def showInfo(self):  
    print()  
    print('Status of TV:', self.brand)  
    print(' Location:', self.location)  
    if self.isOn:  
        ...
```

И тогда увидим это в выводе:

```
Status of TV: Sony
  Location: Family room
  TV is: On
  Channel is: 2
  Volume is: 7

Status of TV: Samsung
  Location: Bedroom
  TV is: On
  Channel is: 44
  Volume is: 10 (sound is muted)
```

Мы выполнили те же вызовы метода, что и в предыдущем примере в листинге 2.9. Разница заключается в том, что каждый ТВ-объект теперь инициализирован брендом и местоположением, и вы можете видеть эту информацию, напечатанную в ответ на каждый вызов измененного метода `showInfo()`.

Использование классов

Используя все, что узнали из этой главы, теперь мы можем создавать классы и несколько независимых экземпляров из этих классов. Вот примеры того, как это можно делать.

- Скажем, мы хотели смоделировать студента на курсе. У нас может быть класс `Student`, который имеет переменные экземпляра для `name`, `emailAddress`, `currentGrade` и так далее. Каждый объект ученика, который мы создаем из этого класса, будет иметь собственный набор переменных экземпляра, и значения, данные переменным экземпляра, будут различными для каждого ученика.
- Рассмотрим игру, где есть несколько игроков. Игрок может быть смоделирован классом `Player` с переменными экземпляра для `name`, `points`, `health`, `location` и так далее. Каждый игрок будет иметь одинаковые возможности, но методы могут работать по-разному на основе различных значений в переменных экземпляра.
- Представьте себе адресную книгу. Можно создать класс `Person` с переменными экземпляра `name`, `address`,

phoneNumber и birthday. Мы можем создать столько объектов из класса Person, сколько захотим, по одному для каждого знакомого человека. Переменные экземпляра в каждом объекте Person будут содержать разные значения. Затем мы можем написать код для поиска по всем объектам Person и извлечь информацию об одном или нескольких из них, которые мы ищем.

В будущих главах я изучу эту концепцию создания нескольких объектов из одного класса и дам вам инструменты, которые помогут управлять коллекцией объектов.

ООП как решение

В конце главы 1 я упомянул три проблемы, которые присущи процедурному программированию. Будем надеяться, что после проработки примеров в этой главе вы увидите, как объектно-ориентированное программирование решает такие задачи.

1. Хорошо написанный класс может быть легко повторно использован во многих различных программах. Классы не нуждаются в доступе к глобальным данным. Вместо этого объекты предоставляют код и данные на одном уровне.
2. Объектно-ориентированное программирование может значительно сократить количество требуемых глобальных переменных, поскольку класс обеспечивает структуру, в которой данные и код, действующий на них, существуют в одной группировке. Это также облегчает отладку кода.
3. Объекты, созданные из класса, имеют доступ только к своим данным — их набору переменных экземпляра в классе. Даже если у вас есть несколько объектов, созданных из одного класса, они не имеют доступа к данным друг друга.

Выводы

В этой главе я представил введение в объектно-ориентированное программирование, продемонстрировав связь между классом и объектом. Класс определяет форму и возможности объекта. Объект — это одиночный экземпляр класса, который имеет собственный набор всех данных, определенных в переменных экземпляра класса. Каждый фрагмент данных,

который должен содержать объект, хранится в переменной экземпляра, которая имеет область видимости объекта. Это означает, что он доступен во всех методах, определенных в классе. Все объекты, созданные из одного класса, получают свой набор всех переменных экземпляра, и, поскольку они могут содержать разные значения, вызов методов на разных объектах может привести к разному поведению.

Я показал, как создавать объект из класса через инструкцию присваивания. После создания экземпляра объекта его можно использовать для вызова любого метода, определенного в классе этого объекта. Я также показал, как можно создавать экземпляры нескольких объектов из одного класса.

В этой главе в демонстрационных классах реализованы физические объекты (выключатели света, телевизоры). Это хороший способ познакомиться с понятиями класса и объекта. Однако в будущих главах я покажу объекты, которые не представляют физические сущности.

3

МЫСЛЕННЫЕ МОДЕЛИ ОБЪЕКТОВ И ЗНАЧЕНИЕ SELF



Надеюсь, новые концепции и терминология, которые я ввел ранее, начинают обретать смысл. Некоторые люди, не знакомые с ООП, испытывают трудности с представлением о том, что такое объект и как методы объекта работают с его переменными экземпляра. Специфика довольно сложна, поэтому может быть полезно разработать ментальную модель того, как работают объекты и классы.

В этой главе я покажу две ментальные модели ООП. Прежде всего я хочу прояснить, что ни одна из этих моделей не является точным представлением того, как объекты работают в Python. Эти модели предназначены просто для того, чтобы дать вам возможность подумать, как выглядит объект и что происходит при вызове метода. В этой главе также будет более подробно рассказано о слове `self` и показано, как оно используется для работы методов с несколькими объектами, созданными из одного класса. На протяжении всей остальной части книги вы получите гораздо более глубокое понимание объектов и классов.

Повторный обзор класса DimmerSwitch

В следующих примерах мы продолжим работу с классом DimmerSwitch из главы 2 (см. листинг 2.5). Класс DimmerSwitch уже имеет две переменные экземпляра: `self.isOn` и `self.brightness`. Единственное изменение, которое мы сделаем, — это добавим переменную экземпляра `self.label`, чтобы каждый создаваемый нами объект можно было легко идентифицировать в выходных данных при запуске программы. Эти переменные создаются и получают начальные значения в методе `__init__()`. Затем к ним обращаются или изменяют их в пяти других методах класса.

В листинге 3.1 приведен тестовый код для создания трех объектов DimmerSwitch из класса DimmerSwitch, которые мы станем использовать в наших ментальных моделях. Я буду вызывать различные методы для каждого из объектов DimmerSwitch.

Файл: OO_DimmerSwitch_Model1.py

```
# создаем первый DimmerSwitch, включаем его и поднимаем уровень
# яркости дважды
oDimmer1 = DimmerSwitch('Dimmer1')
oDimmer1.turnOn()
oDimmer1.raiseLevel()
oDimmer1.raiseLevel()

# создаем второй DimmerSwitch, включаем его и поднимаем уровень
# яркости в 3 раза
oDimmer2 = DimmerSwitch('Dimmer2')
oDimmer2.turnOn()
oDimmer2.raiseLevel()
oDimmer2.raiseLevel()
oDimmer2.raiseLevel()

# создаем третий DimmerSwitch, используя настройки по умолчанию
oDimmer3 = DimmerSwitch('Dimmer3')

# просим каждый переключатель вывести свои показатели
oDimmer1.show()
oDimmer2.show()
oDimmer3.show()
```

Листинг 3.1. Создание трех объектов DimmerSwitch и вызов различных методов для каждого из них

При запуске с классом DimmerSwitch этот код выдает следующие выходные данные:

```
Label: Dimmer1
Light is on? True
Brightness is: 2

Label: Dimmer2
Light is on? True
Brightness is: 3

Label: Dimmer3
Light is on? False
Brightness is: 0
```

Это именно то, чего мы и ожидали. Каждый объект DimmerSwitch не зависит от любых других объектов DimmerSwitch, и каждый объект содержит и изменяет собственные переменные экземпляра.

Высокоуровневая мысленная модель № 1

В этой первой модели каждый объект можно рассматривать как автономную единицу, содержащую тип данных, набор переменных экземпляра, определенных в классе, и копию всех методов, определенных в классе (рис. 3.1).

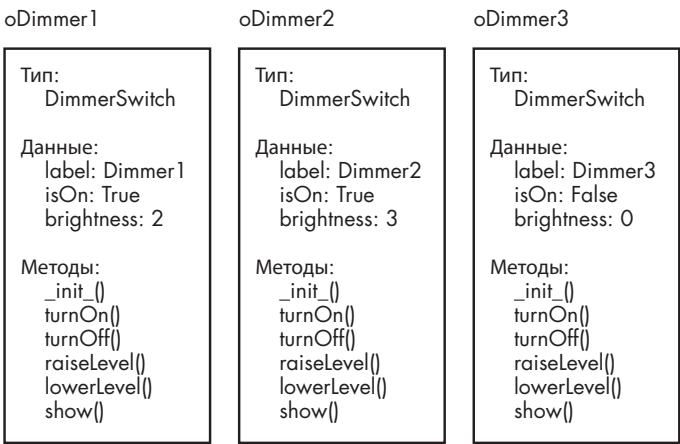


Рис. 3.1. В мысленной модели № 1 каждый объект является единицей, имеющей тип, данные и методы

Данные и методы каждого объекта упакованы вместе. Область видимости переменной экземпляра определяется как все

методы в классе, поэтому все методы имеют доступ к переменным экземпляра, связанным с этим объектом.

Если эта мысленная модель делает концепции ясными, то вы разобрались. Хотя это *не* то, как на самом деле объекты реализованы, это вполне разумный способ *представлять*, как переменные и методы экземпляра объекта работают вместе.

Более глубокая мысленная модель № 2

Эта вторая модель исследует объекты на более низком уровне и лучше объясняет, что такое объект.

Каждый раз, когда создаете экземпляр объекта, вы получаете обратно значение из Python. Обычно возвращаемое значение хранится в переменной, которая ссылается на объект. В листинге 3.2 мы создадим три объекта DimmerSwitch. После создания каждого из них мы добавим код для проверки результата, напечатав тип и значение каждой переменной.

Файл: OO_DimmerSwitch_Model2_Instantiation.py

```
# создаем три объекта DimmerSwitch
oDimmer1 = DimmerSwitch('Dimmer1')
print(type(oDimmer1))
print(oDimmer1)
print()
oDimmer2 = DimmerSwitch('Dimmer2')
print(type(oDimmer2))
print(oDimmer2)
print()
oDimmer3 = DimmerSwitch('Dimmer3')
print(type(oDimmer3))
print(oDimmer3)
print()
```

Листинг 3.2. Создание трех объектов DimmerSwitch и печать типа и значения каждого из них

Вот результат:

```
<class '__main__.DimmerSwitch'>
<__main__.DimmerSwitch object at 0x7ffe503b32e0>
<class '__main__.DimmerSwitch'>
<__main__.DimmerSwitch object at 0x7ffe503b3970>
<class '__main__.DimmerSwitch'>
<__main__.DimmerSwitch object at 0x7ffe503b39d0>
```

Первая строка в каждой группе сообщает нам тип данных. Вместо встроенного типа, такого как `integer` или `float`, мы видим, что все три объекта имеют определенный программистом тип `DimmerSwitch`. (Значение `__main__` указывает, что код `DimmerSwitch` был найден в нашем единственном файле Python, а не импортирован из какого-либо другого файла.)

Вторая строка каждой группировки содержит строку символов. Каждая строка представляет собой расположение в памяти компьютера. Место в памяти — это место, где можно найти все данные, связанные с каждым объектом. Обратите внимание, что каждый объект находится на собственном месте в памяти. Если вы запустите этот код на своем компьютере, то, скорее всего, получите другие значения, но фактические адреса не принципиальны для понимания концепции.

Все объекты `DimmerSwitch` имеют одинаковый тип: класс `DimmerSwitch`. Чрезвычайно важным выводом является то, что все объекты ссылаются на код одного и того же класса, который действительно существует только в одном месте. Когда программа начинает работать, Python считывает все определения классов и запоминает расположение всех классов и их методов.

Веб-сайт Python Tutor (<http://PythonTutor.com>) предоставляет некоторые полезные инструменты, которые могут помочь вам визуализировать работу небольших программ, позволяя пошагово выполнять каждую строку вашего кода. На рис. 3.2 показан снимок экрана с запуском класса `DimmerSwitch` и тестового кода с помощью инструмента визуализации, останавливающего выполнение перед созданием экземпляра первого объекта `DimmerSwitch`.

Здесь видно, что Python запоминает расположение класса `DimmerSwitch` и всех его методов. Хотя классы могут содержать сотни или даже тысячи строк кода, ни один объект фактически не получает копию кода класса. Наличие только одной копии кода очень важно, так как это сохраняет размер ООП-программ небольшим. При создании экземпляра объекта Python выделяет достаточно памяти для каждого объекта, чтобы представить собственный набор переменных экземпляра, определенных в классе. Как правило, создание экземпляра объекта из класса эффективно использует память.

Python 3.6
(known limitations)

```

27 print('Label:', self.label)
28 print('Light is on?', self.isOn)
29 print('Brightness is:', self.brightness)
30 print()
31
32
33 # Main code
34
35 # Create three DimmerSwitch objects
36 oDimmer1 = DimmerSwitch('Dimmer1')
37 print(type(oDimmer1))
38 print(oDimmer1)
39 print()
40 oDimmer2 = DimmerSwitch('Dimmer2')
41 print(type(oDimmer2))
42 print(oDimmer2)
43 print()
44 oDimmer3 = DimmerSwitch('Dimmer3')
45 print(type(oDimmer3))
46 print(oDimmer3)
47 print()

```

Edit this code

→ line that just executed

→ next line to execute

Print output (drag lower right corner to resize)

Frames

Global frame
DimmerSwitch

Objects

DimmerSwitch class

__init__	function	__init__(self, label)
lowerLevel	function	lowerLevel(self)
raiseLevel	function	raiseLevel(self)
show	function	show(self)
turnOff	function	turnOff(self)
turnOn	function	turnOn(self)

Рис. 3.2. Python запоминает все классы и все методы в каждом классе

На снимке экрана на рис. 3.3 показан результат выполнения всего тестового кода, приведенного в листинге 3.2.

Python 3.6
(known limitations)

```

27 print('Label:', self.label)
28 print('Light is on?', self.isOn)
29 print('Brightness is:', self.brightness)
30 print()
31
32
33 # Main code
34
35 # Create three DimmerSwitch objects
36 oDimmer1 = DimmerSwitch('Dimmer1')
37 print(type(oDimmer1))
38 print(oDimmer1)
39 print()
40 oDimmer2 = DimmerSwitch('Dimmer2')
41 print(type(oDimmer2))
42 print(oDimmer2)
43 print()
44 oDimmer3 = DimmerSwitch('Dimmer3')
45 print(type(oDimmer3))
46 print(oDimmer3)
47 print()

```

Edit this code

→ line that just executed

→ next line to execute

<< First

< Prev

Next >

Last >>

Step 28 of 28

Customize visualization (NEW!)

Print output (drag lower right corner to resize)

<_main_.DimmerSwitch object at 0x7f483d72...
<class '_main_.DimmerSwitch'>
<_main_.DimmerSwitch object at 0x7f483d73...

Frames

Global frame
DimmerSwitch
oDimmer1
oDimmer2
oDimmer3

Objects

DimmerSwitch class

__init__	function	__init__(self, label)
lowerLevel	function	lowerLevel(self)
raiseLevel	function	raiseLevel(self)
show	function	show(self)
turnOff	function	turnOff(self)
turnOn	function	turnOn(self)

DimmerSwitch instance

brightness	0
isOn	False
label	"Dimmer1"

DimmerSwitch instance

brightness	0
isOn	False
label	"Dimmer2"

DimmerSwitch instance

brightness	0
isOn	False
label	"Dimmer3"

Рис. 3.3. Выполнение кода из листинга 3.2 демонстрирует, что объекты не включают код в соответствии с мысленной моделью № 2

Это соответствует нашей второй мысленной модели. В правой части снимка экрана код для класса DimmerSwitch отображается только один раз. Каждый объект знает класс,

из которого был создан экземпляр, и содержит собственный набор переменных экземпляра, определенных в классе.

ПРИМЕЧАНИЕ Хотя ниже приведена деталь реализации, это может помочь в дальнейшем понимании объектов. Внутренне все переменные экземпляра объекта хранятся в словаре Python в виде пар имя/значение. Можно проверить, какие переменные экземпляра существуют в объекте, вызвав встроенную функцию `vars()` для любого объекта. Например, в тестовом коде из листинга 3.2, если хотите увидеть внутреннее представление переменных экземпляра, вы можете добавить следующую строку в конце:

```
print('oDimmer1 variables:', vars(oDimmer1))
```

Когда запустите этот код, вы увидите следующие выходные данные:

```
oDimmer1 variables: {'label': 'Dimmer1', 'isOn':  
True, 'brightness': 2}
```

В чем смысл слова «self»?

Философы боролись с вопросом «Что означает Я» на протяжении веков, поэтому с моей стороны было бы довольно претенциозно пытаться объяснить его всего на нескольких страницах. В Python, однако, переменная с именем `self` (я) имеет узкоспециализированное и четкое значение. В этом разделе я покажу, как самому себе присваивается значение и как код методов в классе работает с переменными экземпляра любого объекта, созданного из класса.

ПРИМЕЧАНИЕ Имя переменной `self` не является ключевым словом в Python, но используется по соглашению — может быть использовано любое другое имя, и код будет работать нормально. Тем не менее использование `self` — общепринятая практика в Python, и я буду придерживаться ее на протяжении всей этой книги. Если вы хотите, чтобы ваш код был понят другими программистами Python, используйте имя `self` в качестве первого параметра во всех методах класса. (Другие языки ООП имеют ту же концепцию, но используют другие имена, такие как `this` или `me`.)

Предположим, вы пишете класс с именем `SomeClass`, а затем создаете объект из этого класса, как показано ниже:

```
oSomeObject = SomeClass(<optional arguments>)
```

Объект `oSomeObject` содержит набор всех переменных экземпляра, определенных в классе. Каждый метод класса `SomeClass` имеет определение, которое выглядит следующим образом:

```
def someMethod(self, <any other parameters>):
```

А вот общая форма вызова такого метода:

```
oSomeObject.someMethod(<any other arguments>)
```

Как мы знаем, Python переупорядочивает аргументы при вызове метода таким образом, чтобы объект передавался в качестве первого аргумента. Это значение получается в первом параметре метода и помещается в переменную `self` (рис. 3.4).

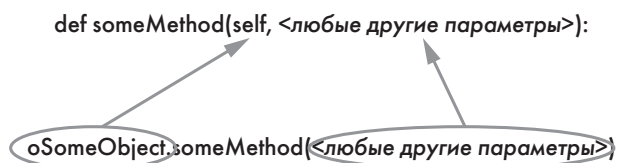


Рис. 3.4. Как Python переупорядочивает аргументы при вызове метода

Поэтому всякий раз, когда вызывается метод, `self` будет установлен на объект в вызове. То есть код метода может работать с переменными экземпляра любого объекта, созданного из класса. Это делается с помощью формы:

```
self.<instanceVariableName>
```

Это по существу говорит об использовании объекта, на который ссылается `self`, и доступе к переменной экземпляра, указанной в *<имя переменной экземпляра>*. Поскольку каждый метод использует `self` в качестве первого параметра, каждый метод в классе использует один и тот же подход.

Для иллюстрации этой концепции воспользуемся классом `DimmerSwitch`. В следующем примере мы создадим экземпляр двух объектов `DimmerSwitch`, а затем пройдемся по тому, что

происходит, когда мы поднимаем уровень яркости этих объектов, вызывая метод `raiseLevel()` для каждого из них.

Код метода, который мы вызываем:

```
def raiseLevel(self):
    if self.brightness < 10:
        self.brightness = self.brightness + 1
```

В листинге 3.3 показан пример тестового кода для двух объектов `DimmerSwitch`.

Файл: `OO_DimmerSwitch_Model2_Method_Calls.py`

```
# создаем два объекта DimmerSwitch
oDimmer1 = DimmerSwitch('Dimmer1')
oDimmer2 = DimmerSwitch('Dimmer2')

# просим oDimmer1 поднять свой уровень яркости
oDimmer1.raiseLevel()

# просим oDimmer2 поднять свой уровень яркости
oDimmer2.raiseLevel()
```

Листинг 3.3. Вызов одного и того же метода на разных объектах `DimmerSwitch`

В этом листинге сначала создадим два объекта `DimmerSwitch`. Затем идут два вызова метода `raiseLevel()`: сперва мы вызываем его для `oDimmer1`, а после — для `oDimmer2`.

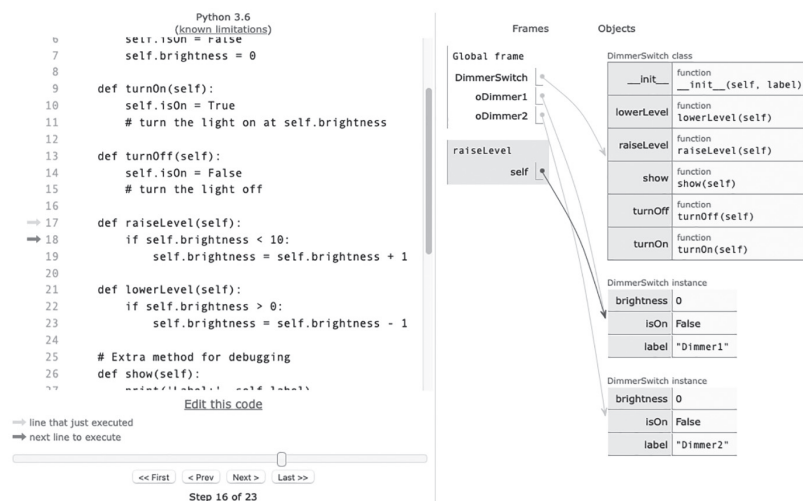


Рис. 3.5. Программа в листинге 3.3 остановилась при вызове `oDimmer1.raiseLevel()`

На рис. 3.5 показан результат выполнения тестового кода из листинга 3.3 в Python Tutor, при этом оно было остановлено во время первого вызова `raiseLevel()`.

Обратите внимание, что `self` и `oDimmer1` относятся к одному и тому же объекту. Когда метод выполняется и использует любую `self`. <переменная экземпляра>, он будет использовать переменные экземпляра `oDimmer1`. Следовательно, когда этот метод выполняется, `self.brightness` относится к переменной экземпляра яркости в `oDimmer1`.

Если мы продолжим выполнять тестовый код в листинге 3.3, то переходим ко второму вызову `raiseLevel()` для `oDimmer2`. На рис. 3.6 я остановил выполнение внутри этого вызова метода.

Обратите внимание, что на этот раз `self` относится к тому же объекту, что и `oDimmer2`. Теперь `self.brightness` относится к переменной экземпляра яркости `oDimmer2`.

Независимо от того, какой объект мы используем или какой метод вызываем, значение объекта присваивается переменной `self` в вызываемом методе. Вы должны думать о `self` как о текущем объекте — том, с которым был вызван метод. Всякий раз, когда метод выполняется, он использует набор переменных экземпляра для объекта, указанного в вызове.

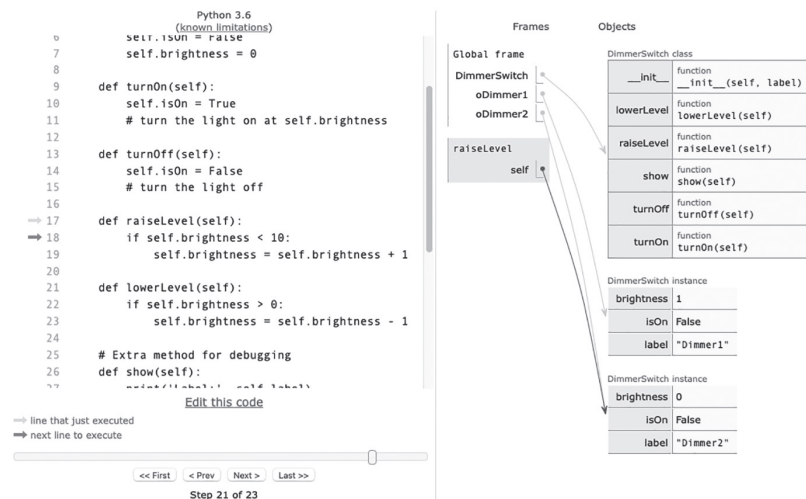


Рис. 3.6. Программа в листинге 3.3 остановилась при вызове `oDimmer2.raiseLevel()`

Выводы

В этой главе я представил два разных способа мышления об объектах. Эти мысленные модели должны помочь в развитии базового понимания того, что происходит, когда вы создаете несколько экземпляров объекта из класса.

Первая модель показала, что вы можете думать об объекте как о наборе всех переменных экземпляра и всех методов класса, завернутых в одну упаковку.

Вторая модель дает гораздо более подробную информацию о реализации, объясняя, что код класса существует только в одном месте. Важный вывод: создание новых объектов из класса космически эффективно. При создании нового экземпляра объекта Python выделяет память для представления переменных экземпляра, определенных в классе. Дубликаты кода класса не создаются, да и не требуются.

Ключом к работе методов с несколькими объектами является то, что первый параметр всех методов, `self`, всегда устанавливается в объект, используемый при вызове этого метода. При таком подходе каждый метод использует переменные экземпляра для текущего объекта.

4

УПРАВЛЕНИЕ НЕСКОЛЬКИМИ ОБЪЕКТАМИ



В этой главе будут показаны методы управления любым количеством объектов, созданных из одного класса. Сначала я рассмотрю пример ООП реализации банковского счета из главы 1.

Подход ООП позволяет данным и коду счета быть на одном уровне, устраняя необходимость зависеть от глобальных данных. Затем я разделю программу на основной код, который представляет меню верхнего уровня, и отдельный объект банка, управляющий счетами, в дополнение к любому количеству объектов счета. Мы также обсудим лучший способ обработки ошибок с помощью исключений.

Класс банковского счета

Нашему классу банковского счета потребуются как минимум имя, пароль и баланс в качестве его данных. В качестве поведения пользователь должен иметь возможность создавать счет, вносить и выводить деньги, проверять свой баланс.

Мы определим и инициализируем переменные для имени, пароля и баланса, а также построим методы для реализации

каждой из операций. После этого сможем создать любое количество объектов Account. Как и начальный класс из главы 1, это упрощенный класс Account, который использует только целые числа для баланса и сохраняет пароль в открытом виде. Хотя вы не будете использовать такие упрощения в реальных банковских приложениях, сейчас это позволит нам сосредоточиться на соответствующих аспектах ООП.

Новый код для класса Account представлен в листинге 4.1.

Файл: Account.py

```
# Класс счета

class Account():
    ❶ def __init__(self, name, balance, password):
        self.name = name
        self.balance = int(balance)
        self.password = password

    ❷ def deposit(self, amountToDeposit, password):
        if password != self.password:
            print('Sorry, incorrect password')
            return None

        if amountToDeposit < 0:
            print('You cannot deposit a negative amount')
            return None

        self.balance = self.balance + amountToDeposit
        return self.balance

    ❸ def withdraw(self, amountToWithdraw, password):
        if password != self.password:
            print('Incorrect password for this account')
            return None

        if amountToWithdraw < 0:
            print('You cannot withdraw a negative amount')
            return None

        if amountToWithdraw > self.balance:
            print('You cannot withdraw more than you have in your
                  account')
            return None

        self.balance = self.balance - amountToWithdraw
        return self.balance
```

```

4 def getBalance(self, password):
    if password != self.password:
        print('Sorry, incorrect password')
        return None
    return self.balance

# Добавлено для отладки
5 def show(self):
    print('        Name:', self.name)
    print('        Balance:', self.balance)
    print('        Password:', self.password)
    print()

```

Листинг 4.1. Минимальный код класса счета

ПРИМЕЧАНИЕ Обработка ошибок в листинге 4.1 очень проста. При обнаружении условия ошибки мы выводим сообщение об ошибке и возвращаем специальное значение `None`. Позже в этой главе я покажу лучший способ обработки ошибок.

Обратите внимание, как методы манипулируют данными и запоминают их. Данные передаются в каждый метод через параметры, которые являются локальными переменными и существуют только во время выполнения метода. Данные запоминаются в переменных экземпляра, которые имеют область видимости объекта и поэтому запоминают свои значения при вызовах различных методов.

Во-первых, мы имеем метод ❶ `__init__()` с тремя параметрами. При создании объекта из этого класса требуются три элемента данных: имя, баланс и пароль. Инстанцирование может выглядеть следующим образом:

```
oAccount = Account('Joe Schmoe', 1000, 'magic')
```

При создании экземпляра объекта значения трех аргументов передаются в метод `__init__()`, который в свою очередь присваивает эти значения переменным экземпляра с аналогичными именами: `self.name`, `self.balance` и `self.password`. Мы будем обращаться к этим переменным экземпляра в других методах.

Метод `deposit()` ❷ позволяет пользователю вносить депозит на счет. После создания экземпляра объекта `Account`

и сохранения его в `oAccount` мы можем вызвать метод `deposit()` следующим образом:

```
newBalance = oAccount.deposit(500, 'magic')
```

Этот вызов говорит внести \$500 и задает пароль «magic». Метод выполняет две проверки валидности запроса при внесении депозита. Первый гарантирует правильность пароля путем сравнения пароля, предоставленного в вызове, с паролем, установленным при создании объекта `Account`. Это хороший пример того, как используется оригинальный пароль, сохраненный в переменной экземпляра `self.password`. Вторая проверка валидности позволяет убедиться, что мы не вносим отрицательную сумму (что на самом деле является выводом средств).

Если одна из этих проверок не прошла, то сейчас мы возвращаем специальное значение `None`, чтобы показать, что произошла какая-то ошибка. Если оба теста пройдены, мы увеличиваем переменную экземпляра `self.balance` на сумму депозита. Поскольку баланс хранится в `self.balance`, он запоминается и доступен для будущих вызовов. Наконец, мы возвращаем новый баланс.

Метод `withdraw()` ❸ работает очень похоже и будет вызван следующим образом:

```
oAccount.withdraw(250, 'magic')
```

Метод `withdraw()` проверяет, что мы указали правильный пароль, сравнивая его с переменной экземпляра `self.password`. Он также проверяет, что мы не просим вывести отрицательную сумму или больше, чем у нас есть на счете, с использованием переменной экземпляра `self.balance`. Как только эти проверки пройдены, метод уменьшает `self.balance` на сумму вывода. Возвращает результирующий баланс.

Чтобы проверить баланс ❹, нам нужно только указать правильный пароль для счета:

```
currentBalance = oAccount.getBalance('magic')
```

Если указанный пароль совпадает с сохраненным в переменной экземпляра `self.password`, метод возвращает значение в параметре `self.balance`.

Наконец, для отладки мы добавили метод `show()` ⁵ для отображения текущих значений `self.name`, `self.balance` и `self.password`, сохраненных для счета.

Класс `Account` — наш первый пример представления чего-либо, что не является физическим объектом. Банковский счет — это не то, что вы можете видеть, чувствовать или трогать. Тем не менее он прекрасно вписывается в мир компьютерных объектов, потому что имеет данные (имя, баланс, пароль) и действия, которые работают с этими данными (создать, внести, вывести, получить баланс, показать).

Импорт кода класса

Существует два способа использования класса, который вы создали в собственном коде. Как мы видели в предыдущих главах, самый простой способ — поместить весь код класса непосредственно в основной исходный файл Python. Но это затрудняет повторное использование класса.

Второй подход заключается в том, чтобы поместить код класса в отдельный файл и импортировать его в программу, которая его использует. Мы поместили весь код для нашего класса `Account` в `Account.py`, но если мы попытаемся запустить `Account.py` самостоятельно, ничего не произойдет, потому что это просто определение класса. Чтобы использовать его, мы должны создать экземпляр одного или нескольких объектов и выполнить вызовы методов объекта. По мере того как наши классы становятся больше и сложнее, сохранение каждого из них в отдельный файл становится предпочтительным способом работы с ними.

Чтобы использовать класс `Account`, мы должны собрать другой файл `.py` и импортировать код из `Account.py`, как мы делаем с другими встроенными пакетами, такими как `random` и `time`. Часто программисты Python называют главную программу, которая импортирует другие файлы классов, `main.py` или `Main_<SomeName>.py`. Затем мы должны убедиться, что `Account.py` и основной файл программы находятся в одной папке. В начале основной программы мы добавляем код класса `Account`, начиная с выражения `import` (обратите внимание, что мы оставляем расширение файла `*.py`):

```
from Account import *
```

Использование оператора импорта со звездочкой (*) обозначает «все содержимое импортируемого файла». Импортируемый файл может содержать несколько классов. В таком случае, где это возможно, следует указывать конкретный класс или классы, которые вы хотите импортировать, вместо того чтобы импортировать весь файл. Вот синтаксис для импорта конкретных классов:

```
from <ExternalFile> import <ClassName1>, <ClassName2>, ...
```

Импорт кода класса имеет два преимущества.

1. Модуль можно использовать повторно, поэтому, если мы хотим использовать *Account.py* в другом проекте, нам просто нужно сделать копию файла и поместить его в папку этого проекта. Повторное использование кода таким образом является одним из основных элементов объектно-ориентированного программирования.
2. Если ваш код класса включен в основную программу, каждый раз, когда вы запускаете программу, Python компилирует весь код в вашем классе (переводит его на язык более низкого уровня, который легче запускать на вашем компьютере), даже если вы не внесли никаких изменений в класс.

Однако, когда вы запускаете свою основную программу с импортированным кодом класса, Python оптимизирует этап компиляции без необходимости делать что-либо. Он создает папку с именем `__pycache__` в папке проекта, затем компилирует код в файл класса и сохраняет его в папке `__pycache__` с вариантом оригинального имени файла Python. Например, для файла *Account.py* Python создаст файл, используя имя *Account.cpython-39.pyc* (или аналогичный на основе используемой версии Python). Расширение *.pyc* обозначает *Python Compiled*. Python перекомпилирует ваш файл класса только в том случае, если изменится исходный файл класса. Если исходник вашей учетной записи *Account.py* не изменился, Python знает, что не нужно его перекомпилировать, и может более эффективно использовать версию файла *pyc*.

Создание тестового кода

Мы протестируем наш новый класс в четырех основных программах. Первая создаст объекты `Account`, используя отдельно именованные переменные. Вторая будет хранить объекты в списке, а третья — номера счетов и объекты в словаре. Наконец, четвертая версия разделит функциональность, поэтому у нас будет основная программа, которая взаимодействует с пользователем, и объект `Bank`, управляющий различными счетами.

В каждом примере основная программа импортирует *Account.py*. Папка проекта должна содержать основную программу и файл *Account.py*. В следующем обсуждении различные версии основной программы будут называться *Main_Bank_VersionX.py*, где X — номер версии.

Создание нескольких учетных записей

В этой первой версии мы создадим два примера счетов и заполним их жизнеспособными данными для тестирования. Сохраним каждый счет в явной переменной, представляющей объект.

Файл: `BankOOP1_IndividualVariables/Main_Bank_Version1.py`

```
# Тестовая программа, использующая счета
# Версия 1, использующая явные переменные для каждого объекта Account

# Берем весь код из файла класса Account
from Account import *

# создаем два счета
❶ oJoesAccount = Account('Joe', 100, 'JoesPassword')
print("Created an account for Joe")
❷ oMarysAccount = Account('Mary', 12345, 'MarysPassword')
print("Created an account for Mary")
❸ oJoesAccount.show()
oMarysAccount.show()
print()

# вызываем разные методы для разных счетов
print('Calling methods of the two accounts ...')
❹ oJoesAccount.deposit(50, 'JoesPassword')
oMarysAccount.withdraw(345, 'MarysPassword')
oMarysAccount.deposit(100, 'MarysPassword')
```



```
# отображаем счета
oJoesAccount.show()
oMarysAccount.show()
```

Листинг 4.2. Основная программа для тестирования класса Account

Мы создаем счет для Джо ❶ и счет для Мэри ❷ и храним результаты в двух объектах Account. Затем мы вызываем метод `show()`, чтобы показать, что счета были созданы правильно ❸. Джо вкладывает \$50. Мэри снимает \$345, а затем вносит \$100 ❹. Если мы запустим программу сейчас, то получим вот такой вывод:

```
Created an account for Joe
Created an account for Mary
    Name: Joe
    Balance: 100
    Password: JoesPassword

    Name: Mary
    Balance: 12345
    Password: MarysPassword

Calling methods of the two accounts ...
    Name: Joe
    Balance: 150
    Password: JoesPassword

    Name: Mary
    Balance: 12100
    Password: MarysPassword
```

Теперь расширим тестовую программу, чтобы создать третий счет в интерактивном режиме, попросив пользователя внести некоторые данные. В листинге 4.3 показан код для этого.

```
# создаем новый счет с информацией от пользователя
print()
userName = input('What is the name for the new user account? ') ❶
userBalance = input('What is the starting balance for this account? ')
userBalance = int(userBalance)
userPassword = input('What is the password you want to use for this
    account? ')
oNewAccount = Account(userName, userBalance, userPassword) ❷
```

```
# отображаем вновь созданный счет пользователя
oNewAccount.show() ❸

# вносим 100 на новый счет
oNewAccount.deposit(100, userPassword) ❹
usersBalance = oNewAccount.getBalance(userPassword)
print()
print('After depositing 100, the user's balance is:', usersBalance)

# отображаем новый счет
oNewAccount.show()
```

Листинг 4.3. Расширение тестовой программы для создания счета на лету

Тестовый код запрашивает у пользователя имя, начальный баланс и пароль ❶. Он использует эти значения для создания нового счета и хранит вновь созданный объект в переменной `oNewAccount` ❷. Затем мы вызываем метод `show()` для нового объекта ❸. Вносим \$100 на счет и выводим новый баланс, вызывая метод `getBalance()` ❹. Когда запускаем полную программу, мы получаем вывод из листинга 4.2, а также следующий вывод:

```
What is the name for the new user account? Irv
What is the starting balance for this account? 777
What is the password you want to use for this account?
IrvsPassword
    Name: Irv
    Balance: 777
    Password: IrvsPassword

After depositing 100, the user's balance is: 877
    Name: Irv
    Balance: 777
    Password: IrvsPassword
```

Здесь важно отметить, что каждый объект `Account` поддерживает собственный набор переменных экземпляра. Каждый объект (`oJoesAccount`, `oMarysAccount` и `oNewAccount`) является глобальной переменной, которая содержит набор из трех переменных экземпляра. Если мы расширим наше определение класса `Account`, включив в него такую информацию, как адрес, номер телефона и дата рождения, каждый объект получит набор этих дополнительных переменных экземпляра.

Несколько объектов учетной записи в списке

Представление каждой учетной записи в отдельной глобальной переменной работает, но это не очень хороший подход, когда нам нужно обрабатывать большое количество объектов. Банку необходим способ обработки произвольного количества счетов. Всякий раз, когда нам нужно произвольное количество фрагментов данных, типичным решением будет список.

В этой версии тестового кода мы начнем с пустого списка объектов Account. Каждый раз, когда пользователь открывает счет, мы создаем экземпляр объекта Account и добавляем полученный объект в наш список. Номер счета для любого данного счета будет индексом счета в списке, начиная с 0. Мы снова начнем с создания одного тестового аккаунта для Джо и другого для Мэри, как показано в листинге 4.4.

Файл: BankOOP2_ListOfAccountObjects/Main_Bank_Version2.py

```
# Тестовая программа, использующая счета
# Версия 2, с использованием списка счетов

# Берем весь код из файла класса Account
from Account import *

# начинаем с пустого списка счетов
accountsList = [ ] ❶

# создаем два счета
oAccount = Account('Joe', 100, 'JoesPassword') ❷
accountsList.append(oAccount)
print("Joe's account number is 0")

oAccount = Account('Mary', 12345, 'MarysPassword') ❸
accountsList.append(oAccount)
print("Mary's account number is 1")

accountsList[0].show() ❹
accountsList[1].show()
print()

# вызываем разные методы для разных счетов
print('Calling methods of the two accounts ...')
accountsList[0].deposit(50, 'JoesPassword') ❺
accountsList[1].withdraw(345, 'MarysPassword') ❻
accountsList[1].deposit(100, 'MarysPassword') ❼
```

```

# отображаем счета
accountsList[0].show() ❸
accountsList[1].show()

# создаем новый счет с информацией от пользователя
print()
userName = input('What is the name for the new user account? ')
userBalance = input('What is the starting balance for this account? ')
userBalance = int(userBalance)
userPassword = input('What is the password you want to use for this
                      account? ')
oAccount = Account(userName, userBalance, userPassword)
accountsList.append(oAccount) # добавляем к списку счетов

# отображаем вновь созданный счет пользователя
print('Created new account, account number is 2')
accountsList[2].show()

# вносим 100 на новый счет
accountsList[2].deposit(100, userPassword)
usersBalance = accountsList[2].getBalance(userPassword)
print()
print('After depositing 100, the user's balance is:', usersBalance)

# отображаем новый счет
accountsList[2].show()

```

Листинг 4.4. Изменен тестовый код для хранения объектов в списке

Начнем с создания пустого списка учетных записей ❶. Создаем учетную запись для Джо, сохраняем возвращенное значение в переменную `oAccount` и немедленно добавляем этот объект в наш список счетов ❷. Так как он первый в списке, номер счета Джо равен 0. Как и в реальном банке, в любое время, когда Джо хочет совершить какие-либо операции со своим счетом, он указывает его номер. Мы используем этот номер, чтобы показать баланс ❹, внести депозит ❺, а затем показать баланс снова ❸. Мы также создаем для Мэри счет с номером 1 ❸ и выполняем некоторые тестовые операции с ее счетом в ❻ и ❼.

Результаты идентичны коду теста из листинга 4.3. Однако между двумя тестовыми программами есть одно очень существенное различие: теперь существует только одна глобальная переменная `accountsList`. Каждый аккаунт имеет уникальный номер, который используется для доступа к определенному счету. Мы сделали важный шаг в сокращении числа глобальных переменных.

Еще одна важная вещь, которую стоит отметить здесь, это то, что мы внесли некоторые довольно значительные изменения в основную программу, но ничего не трогали в файле класса Account. ООП часто позволяет скрывать детали на разных уровнях. Если предположить, что код класса Account сам заботится о реквизитах, связанных с индивидуальным счетом, то можно сосредоточиться на том, как сделать основной код лучше.

Обратите внимание, что мы используем переменную `oAccount` как *временную* переменную. То есть всякий раз, когда создаем новый объект Account, мы назначаем результат переменной `oAccount`. И сразу после этого добавляем `oAccount` к нашему списку учетных записей. Мы никогда не используем переменную `oAccount` в вызовах любого метода конкретного объекта пользователя. Таким образом мы можем повторно использовать переменную `oAccount` для получения значения следующей создаваемой учетной записи.

Несколько объектов с уникальными идентификаторами

Объекты Account должны быть индивидуально идентифицируемыми, чтобы каждый пользователь мог вносить депозиты, выводить средства и получать баланс своего конкретного счета. Использование списка для наших банковских счетов работает, но есть серьезный недостаток. Представьте, что есть пять счетов, пронумерованных 0, 1, 2, 3 и 4. Если владелец аккаунта 2 решит закрыть его, мы, скорее всего, используем стандартную операцию `pop()` в списке, чтобы удалить аккаунт 2. Это вызовет эффект домино: счет, который был на позиции 3, теперь находится на позиции 2, а счет, который был на позиции 4, теперь находится на позиции 3. Однако пользователи этих счетов по-прежнему имеют свои первоначальные номера счетов: 3 и 4. В результате клиент, владеющий счетом 3, теперь получит информацию о бывшем счете 4, а номер счета 4 становится недействительным индексом.

Для работы с большим количеством объектов с уникальными идентификаторами мы обычно используем словарь. В отличие от списка, словарь позволит нам удалять счета без изменения связанных с ними номеров. Мы строим каждую пару ключ/значение с номером счета в качестве ключа и объектом Account в качестве значения. Таким образом, если нам необходимо

удалить тот или иной счет, это никак не повлияет на все остальные. Словарь счетов будет выглядеть следующим образом:

```
{0 : <object for account 0>, 1 : <object for account 1>, ... }
```

Затем мы можем легко получить связанный объект Account и вызвать такой метод:

```
oAccount = accountsDict[accountNumber]
oAccount.someMethodCall()
```

Кроме того, мы можем использовать accountNumber непосредственно для вызова метода для любого объекта Account:

```
accountsDict[accountNumber].someMethodCall()
```

В листинге 4.5 показан тестовый код с использованием словаря объектов Account. Опять же, мы, хотя и вносим множество изменений в наш тестовый код, не меняем ни одной строки в классе Account. В тестовом коде вместо использования жестко закодированных номеров учетных записей мы добавляем счетчик nextAccountNumber, который будем увеличивать после создания нового объекта Account.

Файл: BankOOP3_DictionaryOfAccountObjects/ Main_Bank_Version3.py

```
# Тестовая программа, использующая счета
# Версия 3 с использованием словаря счетов

# Берем весь код из файла класса Account
from Account import *
accountsDict = {} ❶
nextAccountNumber = 0 ❷

# создаем два счета
oAccount = Account('Joe', 100, 'JoesPassword')
joesAccountNumber = nextAccountNumber
accountsDict[joesAccountNumber] = oAccount ❸
print('Account number for Joe is:', joesAccountNumber)
nextAccountNumber = nextAccountNumber + 1 ❹

oAccount = Account('Mary', 12345, 'MarysPassword')
marysAccountNumber = nextAccountNumber
accountsDict[marysAccountNumber] = oAccount ❺
```

```

print('Account number for Mary is:', marysAccountNumber)
nextAccountNumber = nextAccountNumber + 1

accountsDict[joesAccountNumber].show()
accountsDict[marysAccountNumber].show()
print()

# вызываем разные методы для разных счетов
print('Calling methods of the two accounts ...')
accountsDict[joesAccountNumber].deposit(50, 'JoesPassword')
accountsDict[marysAccountNumber].withdraw(345, 'MarysPassword')
accountsDict[marysAccountNumber].deposit(100, 'MarysPassword')

# отображаем счета
accountsDict[joesAccountNumber].show()
accountsDict[marysAccountNumber].show()

# создаем новый счет с информацией от пользователя
print()
userName = input('What is the name for the new user account? ')
userBalance = input('What is the starting balance for this account? ')
userBalance = int(userBalance)
userPassword = input('What is the password you want to use for this
                      account? ')
oAccount = Account(userName, userBalance, userPassword)
newAccountNumber = nextAccountNumber
accountsDict[newAccountNumber] = oAccount
print('Account number for new account is:', newAccountNumber)
nextAccountNumber = nextAccountNumber + 1

# отображаем вновь созданный счет пользователя
accountsDict[newAccountNumber].show()

# вносим 100 на новый счет
accountsDict[newAccountNumber].deposit(100, userPassword)
usersBalance = accountsDict[newAccountNumber].getBalance(userPassword)
print()
print('After depositing 100, the user's balance is:', usersBalance)

# отображаем новый счет
accountsDict[newAccountNumber].show()

```

Листинг 4.5. Изменен тестовый код для хранения номеров и объектов счетов в словаре

Запуск этого кода дает результаты, почти идентичные результатам предыдущих примеров. Мы начинаем с пустого словаря счетов ❶ и инициализируем нашу переменную

nextAccountNumber на 0 ❷. Каждый раз, когда создаем новый счет, мы добавляем запись в словарь счетов, используя текущее значение nextAccountNumber в качестве ключа и объект Account в качестве значения ❸. Это делается для каждого клиента, например Мэри ❹. Каждый раз, когда создаем новый счет, мы увеличиваем nextAccountNumber, чтобы подготовиться к созданию следующего счета ❺. Номера счетов в словаре являются ключами, и, если клиент закрывает свой счет, мы можем удалить этот ключ и значение из словаря, не затрагивая других.

Создание интерактивного меню

Если наш класс Account работает правильно, мы сделаем основной код интерактивным, попросив пользователя сообщить нам, какую операцию он хотел бы сделать: получить баланс, внести депозит, сделать вывод или открыть новый счет. В ответ наш основной код будет собирать необходимую информацию от пользователя, начиная с его номера счета, и вызывать соответствующий метод объекта пользователя Account.

В качестве кратчайшего пути мы снова заполним два счета, один для Джо и один для Мэри. В листинге 4.6 показан расширенный основной код, который использует словарь для отслеживания всех счетов. Для краткости я пропустил код, создающий счета для Джо и Мэри и добавляющий их в словарь счетов (так как это то же самое, что в листинге 4.5).

Файл: BankOOP4_InteractiveMenu/Main_Bank_Version4.py

```
# Интерактивная тестовая программа создания словаря счетов
# Версия 4 с интерактивным меню

from Account import *

accountsDict = {}
nextAccountNumber = 0

--- пропущено создание учетных записей, добавление их в словарь ---

while True:
    print()
    print('Press b to get the balance')
    print('Press d to make a deposit')
    print('Press o to open a new account')
    print('Press w to make a withdrawal')
```



```

print('Press s to show all accounts')
print('Press q to quit')
print()

action = input('What do you want to do? ') ❶
action = action.lower()
action = action[0] # захватываем первую букву
print()

if action == 'b':
    print('*** Get Balance ***')
    userAccountNumber = input('Please enter your account number: ')
    userAccountNumber = int(userAccountNumber)
    userAccountPassword = input('Please enter the password: ')
    oAccount = accountsDict[userAccountNumber]
    theBalance = oAccount.getBalance(userAccountPassword)
    if theBalance is not None:
        print('Your balance is:', theBalance)

elif action == 'd': ❷
    print('*** Deposit ***')
    userAccountNumber = input('Please enter the account number: ') ❸
    userAccountNumber = int(userAccountNumber)
    userDepositAmount = input('Please enter amount to deposit: ')
    userDepositAmount = int(userDepositAmount)
    userPassword = input('Please enter the password: ')
    oAccount = accountsDict[userAccountNumber] ❹
    theBalance = oAccount.deposit(userDepositAmount, userPassword) ❺
    if theBalance is not None:
        print('Your new balance is:', theBalance)

elif action == 'o':
    print('*** Open Account ***')
    userName = input('What is the name for the new user account? ')
    userStartingAmount = input('What is the starting balance for
                                this account? ')
    userStartingAmount = int(userStartingAmount)
    userPassword = input('What is the password you want to use for
                           this account? ')
    oAccount = Account(userName, userStartingAmount, userPassword)
    accountsDict[nextAccountNumber] = oAccount
    print('Your new account number is:', nextAccountNumber)
    nextAccountNumber = nextAccountNumber + 1
    print()

elif action == 's':
    print('Show:')
    for userAccountNumber in accountsDict:

```

```

        oAccount = accountsDict[userAccountNumber]
        print(' Account number:', userAccountNumber)
        oAccount.show()

elif action == 'q':
    break

elif action == 'w':
    print('*** Withdraw ***')
    userAccountNumber = input('Please enter your account number: ')
    userAccountNumber = int(userAccountNumber)
    userWithdrawalAmount = input('Please enter the amount to
                                withdraw: ')
    userWithdrawalAmount = int(userWithdrawalAmount)
    userPassword = input('Please enter the password: ')
    oAccount = accountsDict[userAccountNumber]
    theBalance = oAccount.withdraw(userWithdrawalAmount,
                                    userPassword)
    if theBalance is not None:
        print('Withdrew:', userWithdrawalAmount)
        print('Your new balance is:', theBalance)

else:
    print('Sorry, that was not a valid action. Please try again.')

print('Done')

```

Листинг 4.6. Добавление интерактивного меню

В этой версии мы представляем пользователю меню опций. Когда он выбирает действие ❶, код задает вопросы о предполагаемой транзакции, чтобы собрать всю информацию, необходимую для вызова учетной записи. Например, если пользователь хочет внести депозит ❷, программа запрашивает номер счета, сумму для внесения и пароль счета ❸. Мы используем номер счета в качестве ключа в словаре объектов Account, чтобы получить соответствующий объект Account ❹. С его помощью мы затем вызываем метод `deposit()`, передавая сумму депозита и пароль пользователя ❺.

Опять же, мы изменили код на уровне основного кода и оставили наш класс Account нетронутым.

Создание объекта диспетчера объектов

Код в листинге 4.6 на самом деле делает две разные вещи. Сначала программа предоставляет простой интерфейс меню. Затем, когда выбрано действие, она собирает данные и вызывает метод объекта Account. Вместо того чтобы делать одну большую основную программу, выполняющую две разные задачи, мы можем разделить этот код на две более мелкие логические единицы, каждая из которых имеет четко определенную роль. Система меню становится основным кодом, решающим, какие действия предпринять, а остальная часть кода имеет дело с тем, чем занимается банк. Банк может быть смоделирован как объект, управляющий другими объектами (счетами), известный как *объект диспетчера объектов*.

Объект диспетчера объектов

Объект, который ведет список или словарь управляемых объектов (обычно одного класса) и вызывает методы этих объектов.

Это разделение может быть сделано легко и логично: мы берем весь код, связанный с банком, и помещаем его в новый класс Bank. Затем, в начале основной программы, создаем экземпляр одного объекта Bank из нового класса Bank.

Класс Bank будет управлять списком или словарем объектов Account. Таким образом объект Bank будет единственным кодом, который напрямую взаимодействует с объектами Account (рис. 4.1).

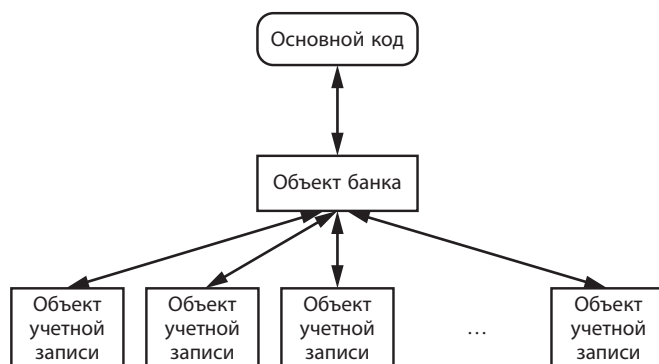


Рис. 4.1. Основной код управляет объектом Bank, который в свою очередь управляет множеством объектов Account

Чтобы создать эту иерархию, нам нужен некоторый основной код, который обрабатывает систему меню самого высокого уровня. В ответ на выбор действия основной код вызовет метод объекта `Bank` (например, `deposit()` или `withdraw()`). Объект `Bank` будет собирать необходимую информацию (номер счета, пароль, сумму для внесения или вывода), обращаться к своему словарию счетов, чтобы найти соответствующий счет пользователя, и вызывать соответствующий метод для этого счета.

В этом разделении труда есть три уровня:

- 1) основной код, который создает и связывается с одним объектом `Bank`;
- 2) объект `Bank`, управляющий словарем объектов `Account` и вызывающий методы этих объектов;
- 3) сами объекты `Account`.

При таком подходе у нас есть только одна глобальная переменная — объект `Bank`. На самом деле основной код понятия не имеет, что объекты учетной записи вообще существуют. И наоборот, каждый объект `Account` не имеет понятия (и не заботится) о том, что такое пользовательский интерфейс верхнего уровня программы. Объект `Bank` получает сообщения от основного кода и связывается с соответствующим объектом `Account`.

Ключевое преимущество этого подхода в том, что мы разбили большую программу на более мелкие подпрограммы: в данном случае основной код и два класса. Это значительно упрощает программирование, поскольку объем работ меньше, а обязанности для каждого элемента более понятны. Кроме того, наличие только одной глобальной переменной гарантирует, что код более низкого уровня не будет случайно влиять на данные на глобальном уровне.

В компьютерной литературе конструкция, показанная на рис. 4.1, часто известна как *композиция* или *композиция объектов*.

Композиция

Логическая структура, в которой один объект управляет одним или несколькими другими объектами.

Вы можете думать, что один объект состоит из других объектов. Например, объект автомобиля состоит из объекта двигателя, объекта рулевого колеса, некоторого количества объектов двери, четырех объектов колеса и шины и так далее. Обсуждение часто сосредоточено на отношениях между объектами. В этом примере можно сказать, что автомобиль «имеет» рулевое колесо, двигатель, некоторое количество дверей и так далее. Следовательно, объект автомобиля представляет собой составной элемент других объектов.

У нас будет три отдельных файла. Основной код живет в собственном файле. Он импортирует код нашего нового файла *Bank.py*, который содержит класс Банка (листинг 4.7). Класс Bank импортирует код файла *Account.py* и использует его для создания экземпляров объектов Account по мере необходимости.

Создание объекта диспетчера объектов

В листинге 4.7 представлен код нового класса Bank, который является объектом диспетчера объектов.

Файл: BankOOP5_SeparateBankClass/Bank.py

```
# Банк, управляющий словарем объектов Account

from Account import *

class Bank():

    def __init__(self):
        self.accountsDict = {} ❶
        self.nextAccountNumber = 0

    def createAccount(self, theName, theStartingAmount, thePassword): ❷
        oAccount = Account(theName, theStartingAmount, thePassword)
        newAccountNumber = self.nextAccountNumber
        self.accountsDict[newAccountNumber] = oAccount
        # увеличиваем на единицу для подготовки к созданию следующей
        # учетной записи
        self.nextAccountNumber = self.nextAccountNumber + 1
        return newAccountNumber

    def openAccount(self): ❸
        print('*** Open Account ***')
        userName = input('What is the name for the new user account? ')
        userStartingAmount = input('What is the starting balance for
                                   this account? ')
```

```

userStartingAmount = int(userStartingAmount)
userPassword = input('What is the password you want to use for
                      this account? ')

userAccountNumber = self.createAccount(userName,
                                         userStartingAmount, userPassword) ❹
print('Your new account number is:', userAccountNumber)
print()

def closeAccount(self): 5
    print('*** Close Account ***')
    userAccountNumber = input('What is your account number? ')
    userAccountNumber = int(userAccountNumber)
    userPassword = input('What is your password? ')
    oAccount = self.accountsDict[userAccountNumber]
    theBalance = oAccount.getBalance(userPassword)
    if theBalance is not None:
        print('You had', theBalance, 'in your account, which is
              being returned to you.')
        # удаляем учетную запись пользователя из словаря учетных
        # записей
        del self.accountsDict[userAccountNumber]
        print('Your account is now closed.')

def balance(self):
    print('*** Get Balance ***')
    userAccountNumber = input('Please enter your account number: ')
    userAccountNumber = int(userAccountNumber)
    userAccountPassword = input('Please enter the password: ')
    oAccount = self.accountsDict[userAccountNumber]
    theBalance = oAccount.getBalance(userAccountPassword)
    if theBalance is not None:
        print('Your balance is:', theBalance)

def deposit(self):
    print('*** Deposit ***')
    accountNum = input('Please enter the account number: ')
    accountNum = int(accountNum)
    depositAmount = input('Please enter amount to deposit: ')
    depositAmount = int(depositAmount)
    userAccountPassword = input('Please enter the password: ')
    oAccount = self.accountsDict[accountNum]
    theBalance = oAccount.deposit(depositAmount, userAccountPassword)
    if theBalance is not None:
        print('Your new balance is:', theBalance)

def show(self):
    print('*** Show ***')

```

```

for userAccountNumber in self.accountsDict:
    oAccount = self.accountsDict[userAccountNumber]
    print(' Account:', userAccountNumber)
    oAccount.show()

def withdraw(self):
    print('*** Withdraw ***')
    userAccountNumber = input('Please enter your account number: ')
    userAccountNumber = int(userAccountNumber)
    userAmount = input('Please enter the amount to withdraw: ')
    userAmount = int(userAmount)
    userAccountPassword = input('Please enter the password: ')
    oAccount = self.accountsDict[userAccountNumber]
    theBalance = oAccount.withdraw(userAmount, userAccountPassword)
    if theBalance is not None:
        print('Withdrew:', userAmount)
        print('Your new balance is:', theBalance)

```

Листинг 4.7. Класс Bank с отдельными методами для различных банковских операций

Я сосредоточусь на наиболее важных вещах, на которые стоит обратить внимание в классе Bank. Во-первых, в методе `__init__()` класса Bank инициализируются две переменные: `self.accountsDict` и `self.nextAccountNumber` ❶. Префикс `self.` обозначает их как переменные экземпляра, то есть класс Bank может ссылаться на них в любом из своих методов.

Во-вторых, существуют два метода создания учетной записи: `createAccount()` и `openAccount()`. Метод `createAccount()` создает экземпляр новой учетной записи ❷ с именем пользователя, начальной суммой и паролем, переданным для новой учетной записи. Метод `openAccount()` запрашивает у пользователя вопросы для получения этих трех элементов информации ❸ и вызывает метод `createAccount()` в рамках одного класса.

Вызов одного метода другим в том же классе — обычное явление. Но вызываемый метод не знает, был ли он вызван изнутри или снаружи класса; ему известно только, что первый аргумент — это объект, на котором он должен работать. Поэтому вызов метода должен начинаться с `self.`, ведь `self` всегда относится к текущему объекту. Как правило, для вызова из одного метода в другой в том же классе необходимо записать:

```

def myMethod(self, <other optional parameters>):
    ...
    self.methodInSameClass(<any needed arguments>)

```

После сбора информации от пользователя для `openAccount()` у нас есть следующая строка ❹:

```
userAccountNumber = self.createAccount(userName, userStartingAmount,  
                                         userPassword)
```

Здесь `openAccount()` вызывает `createAccount()` из того же класса для создания учетной записи. Метод `createAccount()` запускается, создает экземпляр объекта `Account` и возвращает номер счета в `openAccount()`, который возвращает этот номер счета пользователю.

Наконец, новый метод `closeAccount()` позволяет пользователю закрыть существующий счет ❺. Это дополнительный функционал, который мы предлагаем из нашего основного кода.

Класс `Bank` воплощает абстрактный банк, а не физический объект. Это еще один хороший пример класса, который не представляет физическую структуру.

Основной код, создающий объект диспетчера объектов

Основной код, который создает и совершает вызовы объекта `Bank`, представлен в листинге 4.8.

Файл: BankOOP5_SeparateBankClass/Main_Bank_Version5.py

```
# Основная программа, контролирующая банк, состоящий из счетов  
  
# Берем весь код класса банка  
from Bank import *  
  
# создаем экземпляр банка  
oBank = Bank()  
  
# Основной код  
# создаем два тестовых счета  
joesAccountNumber = oBank.createAccount('Joe', 100, 'JoesPassword')  
print("Joe's account number is:", joesAccountNumber)  
  
marysAccountNumber = oBank.createAccount('Mary', 12345, 'MarysPassword')  
print("Mary's account number is:", marysAccountNumber)  
  
while True:  
    print()  
    print('To get an account balance, press b')  
    print('To close an account, press c')
```



```

print('To make a deposit, press d')
print('To open a new account, press o')
print('To quit, press q')
print('To show all accounts, press s')
print('To make a withdrawal, press w ')
print()

```

```

❶ action = input('What do you want to do? ')
   action = action.lower()
   action = action[0] # берем первую букву
   print()

❷ if action == 'b':
    oBank.balance()

❸ elif action == 'c':
    oBank.closeAccount()

   elif action == 'd':
       oBank.deposit()

   elif action == 'o':
       oBank.openAccount()

   elif action == 's':
       oBank.show()

   elif action == 'q':
       break

   elif action == 'w':
       oBank.withdraw()

   else:
       print('Sorry, that was not a valid action. Please try again.')

print('Done')

```

Листинг 4.8. Основной код, который создает объект `Bank` и совершает вызовы к нему

Обратите внимание, что код в листинге 4.8 представляет систему меню верхнего уровня. Он запрашивает у пользователя действие ❶, затем вызывает соответствующий метод в объекте `Bank` для выполнения работы ❷. Вы можете легко расширить объект `Bank`, чтобы обработать некоторые дополнительные запросы, например запросить часы работы банка, адрес или номер телефона. Эти данные могут просто храниться как

дополнительные переменные экземпляра внутри объекта Bank. Bank ответит на эти вопросы без необходимости связываться с каким-либо объектом Account.

При выполнении запроса на закрытие **3** основной код вызывает метод `closeAccount()` объекта Bank для закрытия счета. Объект Bank удаляет конкретный счет из своего словаря счетов, используя следующую строку:

```
del self.accountsDict[userAccountNumber]
```

Напомним, что объект — это данные плюс код, который действует на них с течением времени. Возможность удаления объекта проистекает из этого определения. Мы можем создать объект (в данном случае объект Account), когда захотим, а не только когда запустится программа. Мы создаем новый объект Account всякий раз, когда пользователь решает открыть счет. Наш код может использовать этот объект, вызывая его методы. Мы также можем удалить объект в любое время, когда пользователь решит закрыть свой счет. Это пример того, что объект (например, Account) имеет продолжительность жизни, начиная с момента его создания и заканчивая удалением.

Лучшая обработка ошибок с исключениями

До сих пор в классе Account, если метод обнаруживал ошибку (например, когда пользователь вносит отрицательную сумму, вводит неправильный пароль, выводит отрицательную сумму и т. д.), мы выбирали решение вернуть None в качестве сигнала о том, что что-то пошло не так. В этом разделе обсудим лучший способ обработки ошибок, используя блоки `try/except` и вызывая исключения.

try и except

Когда в функции или методе из стандартной библиотеки Python возникает ошибка времени выполнения или ненормальное состояние, эта функция или метод сигнализирует об ошибке, вызывая исключение (иногда называется *выбросить* или *сгенерировать* исключение). Мы можем обнаруживать исключения и реагировать на них, используя конструкцию `try/except`. Вот общая форма:

```
try:
    # код, который может вызвать ошибку (вызвать исключение)
except <some exception name>: # если случается исключение
    # код для обработки исключения
```

Если код внутри блока `try` работает корректно и не генерирует исключение, исключаящее предложение пропускается и выполнение продолжается после блока `except`. Однако, если код в блоке `try` приводит к исключению, управление передается оператору `except`. Если исключение соответствует одному из нескольких исключений, перечисленных в операторе `except`, управление передается в код блока `except`. Это часто называют *поймать* исключение. Такой блок с отступом обычно содержит код для сообщения и/или восстановления после ошибки.

Вот простой пример, где мы запрашиваем число у пользователя и пытаемся преобразовать его в целое:

```
age = input('Please enter your age: ')
try: # попытка преобразования в целое число
    age = int(age)
except ValueError: # если возникло исключение при попытке преобразования
    print('Sorry, that was not a valid number')
```

Вызовы стандартной библиотеки Python могут генерировать стандартные исключения, такие как `TypeError`, `ValueError`, `NameError`, `ZeroDivisionError` и т. д. В этом примере, если пользователь вводит буквы или число с плавающей точкой, встроенная функция `int()` вызывает исключение `ValueError` и управление передается коду в блоке `except`.

Инструкция `raise` и пользовательские исключения

Если код обнаруживает ошибку во время выполнения, можно использовать инструкцию `raise`, чтобы подать сигнал об исключении. Существует много форм оператора `raise`, но стандартный подход заключается в использовании этого синтаксиса:

```
raise <ExceptionName>('Any string to describe the error')
```

Для `<ExceptionName>` у вас есть три варианта. Во-первых, если есть стандартное исключение, которое соответствует

обнаруженной ошибке (`TypeError`, `ValueError`, `NameError`, `ZeroDivisionError` и т. д.), его можно использовать. Вы также можете добавить собственную строку описания:

```
raise ValueError('You need to specify an integer')
```

Во-вторых, можно использовать общее исключение `Exception`:

```
raise Exception('The amount cannot be a floating-point number')
```

Однако такой подход не слишком удачен, потому что стандартная практика написания исключений состоит в поиске исключений по имени, а этот способ не дает конкретного имени.

Третий вариант, и, возможно, лучший, — создать свое исключение. Это легко сделать, но необходима техника, называемая наследованием (которую мы подробно обсудим в главе 10). Вот все, что вам нужно, чтобы создать собственное исключение:

```
# Определите пользовательское исключение
class <CustomExceptionName>(Exception):
    pass
```

Вы указываете уникальное имя для вашего исключения, которое затем можете вызвать в коде. Создание собственных исключений означает, что вы можете явно проверить их по имени на более высоком уровне кода. В следующем разделе мы перепишем код примера из нашего банка, чтобы создать пользовательское исключение в классах `Bank` и `Account` и проверить и сообщить об ошибке в основном коде. Основной код сообщит об ошибке, но позволит программе продолжить работу.

В типичном случае оператор `raise` вызывает выход из текущей функции или метода и возвращает управление вызывающему абоненту. Если вызывающий объект содержит исключяющее условие, которое ловит исключение, выполнение продолжается внутри этого исключения. В противном случае данная функция или метод завершается. Этот процесс повторяется до тех пор, пока исключение не попадет в исключение само. Управление передается обратно через последовательность вызовов, и, если исключение не ловится никаким `except`, программа завершается и Python отображает ошибку.

Использование исключений в нашей банковской программе

Теперь мы можем переписать все три уровня программы (основной, Банк и Счет), чтобы сигнализировать об ошибках с помощью операторов `raise` и обрабатывать ошибки с помощью блоков `try/except`.

Класс счета с исключениями

Листинг 4.9 представляет собой новую версию класса `Account`, переписанную для использования исключений и оптимизированную таким образом, чтобы код не повторялся. Мы начнем с определения пользовательского исключения

`AbortTransaction`, которое будет создано, если мы обнаружим какую-либо ошибку, когда пользователь пытается выполнить транзакцию в нашем банке.

Файл: BankOOP6_UsingExceptions/Account.py (изменено для работы с предстоящим Bank.py)

```
# Класс счета
# Ошибки обозначены операторами "raise"

# Определяем пользовательское исключение
class AbortTransaction(Exception): ❶
    '''raise this exception to abort a bank transaction'''
    pass

class Account():
    def __init__(self, name, balance, password):
        self.name = name
        self.balance = self.validateAmount(balance) ❷
        self.password = password

    def validateAmount(self, amount):
        try:
            amount = int(amount)
        except ValueError:
            raise AbortTransaction('Amount must be an integer') ❸
        if amount <= 0:
            raise AbortTransaction('Amount must be positive') ❹
        return amount

    def checkPasswordMatch(self, password): ❺
        if password != self.password:
```

```

        raise AbortTransaction('Incorrect password for this account')

def deposit(self, amountToDeposit): ❹
    amountToDeposit = self.validateAmount(amountToDeposit)
    self.balance = self.balance + amountToDeposit
    return self.balance

def getBalance(self):
    return self.balance

def withdraw(self, amountToWithdraw): ❺
    amountToWithdraw = self.validateAmount(amountToWithdraw)
    if amountToWithdraw > self.balance:
        raise AbortTransaction('You cannot withdraw more than you
                                have in your account')
    self.balance = self.balance - amountToWithdraw
    return self.balance

# Добавлено для отладки
def show(self):
    print('        Name:', self.name)
    print('        Balance:', self.balance)
    print('        Password:', self.password)

```

Листинг 4.9. Измененный класс счета, который вызывает исключения

Мы начинаем с определения пользовательского исключения `AbortTransaction` ❶, чтобы применить его в этом классе и в другом коде, который импортирует этот класс.

В методе `__init__()` класса `Account` мы гарантируем, что сумма, указанная в качестве начального баланса, действительна, вызвав `validateAmount()` ❷. Этот метод использует блок `try/except`, чтобы гарантировать, что начальная сумма может быть успешно преобразована в целое число. Если вызов `int()` завершается неудачей, возникает исключение `ValueError`, которое попадает в секцию `except`. Вместо того чтобы просто разрешить общему `ValueError` быть возвращенным вызывающему, код этого блока `except` ❸ выполняет инструкцию `raise`, вызывая наше исключение `AbortTransaction`, и вмещает в себя более содержательную строку сообщения об ошибке. Если преобразование в целое число успешно, мы выполняем еще один тест. Если пользователь задал отрицательную сумму, мы также вызываем исключение `AbortTransaction` ❹, но с другой строкой сообщения об ошибке.

Метод `checkPasswordMatch()` ⑤ вызывается методами объекта `Bank` для проверки соответствия пароля, введенного пользователем, паролю, сохраненному в `Account`. В случае несоответствия мы выполняем другую инструкцию `raise` с тем же исключением, но предоставляем более содержательную строку сообщения об ошибке.

Это позволяет упростить код `deposit()` ⑥ и `withdraw()` ⑦, поскольку данные методы предполагают, что сумма и пароль были проверены до их вызова. Существует дополнительная проверка в `withdraw()`, чтобы убедиться, что пользователь не пытается вывести больше денег, чем находится на счете; если это не так, мы вызываем исключение `AbortTransaction` с соответствующим описанием.

Поскольку в данном классе нет кода для обработки исключения `AbortTransaction`, каждый раз, когда оно возникает, управление передается обратно вызывающему. Если у вызывающего нет кода для обработки исключения, то управление передается обратно предыдущему вызывающему и так далее вверх по стеку вызовов. Как мы увидим, наш основной код справится с этим исключением.

Оптимизированный класс банка

Полный код класса `Bank` доступен для скачивания. В листинге 4.10 я показываю некоторые примеры методов, которые демонстрируют методы `try/except` с вызовами методов в ранее обновленном классе `Account`.

Файл: BankOOP6_UsingExceptions/Bank.py (изменено для работы с предыдущим `Account.py`)

```
# Банк, управляющий словарем объектов Account

from Account import *

class Bank():
    def __init__(self, hours, address, phone): ①
        self.accountsDict = {}
        self.nextAccountNumber = 0
        self.hours = hours
        self.address = address
        self.phone = phone

    def askForValidAccountNumber(self): ②
```

```

accountNumber = input('What is your account number? ')
try: ❸
    accountNumber = int(accountNumber)
except ValueError:
    raise AbortTransaction('The account number must be an
                           integer')
if accountNumber not in self.accountsDict:
    raise AbortTransaction('There is no account ' +
                           str(accountNumber))

return accountNumber

def getUsersAccount(self): ❹
    accountNumber = self.askForValidAccountNumber()
    oAccount = self.accountsDict[accountNumber]
    self.askForValidPassword(oAccount)
    return oAccount

--- пропущены дополнительные методы ---

def deposit(self): ❺
    print('*** Deposit ***')
    oAccount = self.getUsersAccount()
    depositAmount = input('Please enter amount to deposit: ')
    theBalance = oAccount.deposit(depositAmount)
    print('Deposited:', depositAmount)
    print('Your new balance is:', theBalance)

def withdraw(self): ❻
    print('*** Withdraw ***')
    oAccount = self.getUsersAccount()
    userAmount = input('Please enter the amount to withdraw: ')
    theBalance = oAccount.withdraw(userAmount)
    print('Withdrew:', userAmount)
    print('Your new balance is:', theBalance)

def getInfo(self): ❼
    print('Hours:', self.hours)
    print('Address:', self.address)
    print('Phone:', self.phone)
    print('We currently have', len(self.accountsDict), 'account(s)
          open.')

# Специальный метод только для администратора банка
def show(self):
    print('*** Show ***')
    print('(This would typically require an admin password)')
    for userAccountNumber in self.accountsDict:
        oAccount = self.accountsDict[userAccountNumber]

```



```
print('Account:', userAccountNumber)
oAccount.show()
print()
```

Листинг 4.10. Измененный класс Bank

Класс Bank начинается с метода `__init__()` ❶, который сохраняет всю необходимую информацию в переменных экземпляра.

Новый метод `askForValidAccountNumber()` ❷ вызывается из ряда других, чтобы запросить у пользователя номер счета и попытаться проверить заданный номер. Для начала он включает блок `try/except` ❸, чтобы убедиться, что число является целым. Если это не так, блок `except` обнаруживает ошибку как исключение `ValueError`, но сообщает об ошибке более четко, создавая пользовательские исключение `AbortTransaction` с содержательным сообщением. Далее он проверяет, что указанный номер счета банку знаком. В противном случае он также вызывает исключение `AbortTransaction`, но выдает другую строку сообщения об ошибке.

Обновленный метод `getUsersAccount()` ❹ сначала вызывает предыдущий `askForValidAccountNumber()`, а затем использует номер счета, чтобы найти соответствующий объект `Account`. Обратите внимание, что в этом методе нет `try/except`. Если исключение создано в `askForValidAccountNumber()` (или на более низком уровне), этот метод немедленно вернется к своему вызывающему.

Методы `deposit()` ❺ и `withdraw()` ❻ вызывают `getUsersAccount()` в том же классе. Аналогичным образом, если их вызов `getUsersAccount()` вызывает исключение, метод завершится и передаст исключение в цепочку вызывающему методу. Если все тесты проходят успешно, код `deposit()` и `withdraw()` вызывает аналогично именованные методы в указанном объекте `Account` для выполнения фактической транзакции.

Метод `getInfo()` ❼ сообщает информацию о банке (часы, адрес, телефон) и не имеет доступа к индивидуальным счетам.

Основной код, обрабатывающий исключения

В листинге 4.11 показан обновленный основной код, переписанный для обработки пользовательского исключения. Здесь о любых возникших ошибках сообщается пользователю.

Файл: BankOOP6_UsingException/Main_Bank_Version6.py

```
# Основная программа, контролирующая банк, состоящий из счетов
from Bank import *

# создаем экземпляр банка
❶ oBank = Bank('9 to 5', '123 Main Street, Anytown, USA', '(650) 555-1212')

# Основной код
❷ while True:
    print()
    print('To get an account balance, press b')
    print('To close an account, press c')
    print('To make a deposit, press d')
    print('To get bank information, press i')
    print('To open a new account, press o')
    print('To quit, press q')
    print('To show all accounts, press s')
    print('To make a withdrawal, press w')
    print()

    action = input('What do you want to do? ')
    action = action.lower()
    action = action[0] # берем первую букву
    print()

❸    try:
        if action == 'b':
            oBank.balance()
        elif action == 'c':
            oBank.closeAccount()
        elif action == 'd':
            oBank.deposit()
        elif action == 'i':
            oBank.getInfo()
        elif action == 'o':
            oBank.openAccount()
        elif action == 'q':
            break
        elif action == 's':
            oBank.show()
        elif action == 'w':
            oBank.withdraw()
❹    except AbortTransaction as error:
        # выводим текст сообщения об ошибке
        print(error)

print('Done')
```

Листинг 4.11. Основной код, который обрабатывает ошибки с try/except

Основной код начинается с создания единственного объекта `Bank` ❶. Затем в цикле он представляет пользователю меню верхнего уровня и спрашивает его, какое действие он хочет выполнить ❷. Вызывает подходящий метод для каждой команды.

Важно, что в этом списке мы добавили блок `try` вокруг всех вызовов методов объекта `oBank` ❸. Таким образом, если какой-либо вызов метода вызывает исключение `AbortTransaction`, управление будет передано оператору `except` ❹.

Исключения являются объектами. В части `except` обрабатывается исключение `AbortTransaction`, которое было вызвано на любом более низком уровне. Мы присваиваем значение исключения переменной `error`. Когда напечатаем эту переменную, пользователь увидит соответствующее сообщение об ошибке. Поскольку исключение было обработано в части `except`, программа продолжает действовать, и пользователю задается вопрос, что он хочет сделать.

Вызов одного и того же метода для списка объектов

В отличие от нашего банковского примера, в случаях, когда отдельные объекты не нужно однозначно идентифицировать, использование списка объектов работает очень хорошо. Предположим, вы кодируете игру, и вам нужно иметь некоторое количество плохих парней, космических кораблей, пуль, зомби или чего-то еще. Каждый подобный объект, как правило, будет иметь какие-то данные, которые он запоминает, и какие-то действия, которые он может выполнить. До тех пор пока для каждого объекта не требуется уникальный идентификатор, стандартный способ обработки — создать множество экземпляров объекта из класса и поместить их в список:

```
objectList = [] # начинаем с пустого списка
for i in range(nObjects):
    oNewObject = MyClass() # создаем новый экземпляр
    objectList.append(oNewObject) # сохраняем объект в списке
```

В игре мы представляем мир как большую сетку, электронную таблицу. Мы хотим разместить монстров в случайных местах. В листинге 4.12 показано начало класса `Monster` с его методом `__init__()` и методом `Move()`. Когда создается

экземпляр `Monster`, ему сообщается количество строк и столбцов в сетке и максимальная скорость, и он выбирает случайное начальное местоположение и скорость.

Файл: `MonsterExample.py`

```
import random

class Monster():
    def __init__(self, nRows, nCols, maxSpeed):
        self.nRows = nRows # сохраняем значение
        self.nCols = nCols # сохраняем значение
        self.myRow = random.randrange(self.nRows) # выбираем
                                                    # случайную строку
        self.myCol = random.randrange(self.nCols) # выбираем
                                                    # случайную колонку
        self.mySpeedX = random.randrange(-maxSpeed, maxSpeed + 1)
                                                    # выбираем скорость по оси X
        self.mySpeedY = random.randrange(-maxSpeed, maxSpeed + 1)
                                                    # выбираем скорость по оси Y
        # задаем другие переменные экземпляра, такие как здоровье,
        # сила и т. д.

    def move(self):
        self.myRow = (self.myRow + self.mySpeedY) % self.nRows
        self.myCol = (self.myCol + self.mySpeedX) % self.nCols
```

Листинг 4.12. Класс монстров, который может быть использован для создания многих экземпляров монстров

С помощью этого класса `Monster` мы можем создать список объектов `Monster`, вот так:

```
N_MONSTERS = 20
N_ROWS = 100 # может быть любого размера
N_COLS = 200 # может быть любого размера
MAX_SPEED = 4
monsterList = [] # начинаем с пустого списка
for i in range(N_MONSTERS):
    oMonster = Monster(N_ROWS, N_COLS, MAX_SPEED) # создаем монстра
    monsterList.append(oMonster) # добавляем монстра в наш список
```

Этот цикл создаст 20 экземпляров класса `Monster`, и каждый из них будет знать собственное начальное местоположение в сетке и свою индивидуальную скорость. Когда у вас уже есть список объектов и вы хотите, чтобы каждый выполнил одно

и то же действие, можно написать простой цикл, где вы вызываете один и тот же метод каждого объекта в списке:

```
for objectVariable in objectVariablesList:
    objectVariable.someMethod()
```

Например, если мы хотим, чтобы каждый из наших объектов `Monster` перемещался, мы могли бы использовать цикл, как этот:

```
for oMonster in monsterList:
    oMonster.move()
```

Поскольку каждый объект `Monster` запоминает свое местоположение и скорость, в методе `move()` каждый монстр может переместиться и запомнить свое новое местоположение.

Эта техника построения списка объектов и вызова одного и того же метода всех объектов в списке чрезвычайно полезна, и это стандартный подход к работе с коллекцией похожих объектов. Мы будем использовать его довольно часто, когда позднее доберемся до создания игр с помощью `pygame`.

Интерфейс по сравнению с реализацией

Наш предыдущий класс `Account`, кажется, имеет методы и переменные экземпляра, которые работают хорошо. Когда вы уверены, что ваш код действует исправно, не нужно беспокоиться о деталях внутри класса. Когда класс делает то, что вы хотите, все, что вам нужно помнить, это какие методы в нем доступны. Существует два различных способа взглянуть на класс: сосредоточиться на том, что он способен делать (*интерфейс*) и как он работает внутри (*реализация*).

Интерфейс

Коллекция методов, предоставляемых классом (и параметры, которые ожидает каждый метод). Интерфейс показывает, что может сделать объект, созданный из класса.

Реализация

Фактический код класса, который показывает, как объект делает то, что он делает.

Если вы создатель или сопровождающий класса, вам нужно полностью понять реализацию — код всех методов и то, как они работают вместе, чтобы повлиять на переменные экземпляра. Если вы просто пишете код для использования класса, вам нужно только позаботиться об интерфейсе — различных методах, доступных в классе, значениях, которые необходимо передать в каждый метод, и любых значениях, возвращающихся из методов. Если вы пишете код самостоятельно (как «команда из одного человека»), то будете как исполнителем класса, так и пользователем его интерфейса.

Пока интерфейс класса не изменяется, реализацию класса можно изменить в любое время. То есть, если вы обнаружите, что метод может быть реализован быстрее или эффективнее, изменение соответствующего кода внутри класса не будет иметь никаких плохих побочных эффектов ни на какую другую часть программы.

Выводы

Объект диспетчера объектов — это объект, управляющий другими объектами. Это делается путем создания одной или нескольких переменных экземпляра, которые представляют собой списки или словари, состоящие из других объектов. Диспетчер объектов может вызывать методы любого конкретного объекта или всех управляемых объектов. Этот метод дает полный контроль над всеми управляемыми объектами только менеджеру объектов.

При возникновении ошибки в методе или функции можно создать исключение. Оператор `raise` возвращает управление вызывающему методу. Вызывающий может обнаружить потенциальную ошибку, поместив вызов в блок `try`, и отреагировать на любую такую ошибку с помощью блока `except`.

Интерфейс класса — это документирование всех методов и связанных с ними параметров в классе. Реализация — это фактический код класса. То, что вам нужно знать, зависит от вашей роли. Автор/сопровождающий класса должен понимать детали кода, в то время как любой, кто использует класс, должен понимать только интерфейс, который предоставляет класс.

ЧАСТЬ II

ГРАФИЧЕСКИЕ ПОЛЬЗОВАТЕЛЬСКИЕ ИНТЕРФЕЙСЫ С PYGAME

Эти главы познакомят вас с *pygame*, внешним пакетом, который добавляет функциональность, общую для GUI-программ. Pygame позволяет вам писать на языке Python программы, в которых есть окна, управляемые мышью и клавиатурой, и которые воспроизводят звуки и многое другое.

Глава 5 дает базовое понимание того, как *pygame* работает, и предоставляет стандартный шаблон для создания основанных на *pygame* программ. Для начала мы создадим несколько простых программ: приложение, которое управляет изображением с помощью клавиатуры, а затем приложение с прыгающими мячами.

Глава 6 объясняет, как лучше всего использовать *pygame* в объектно-ориентированной структуре. Вы увидите, как переписать программу с прыгающими мячами с помощью объектно-ориентированных методов и разработать простые кнопки и поля ввода текста.

Глава 7 описывает модуль *pygame.widgerts*, который содержит полные реализации многих стандартных виджетов пользовательского интерфейса, таких как кнопки, поля ввода и вывода, переключатели, флажки и прочее, каждый из которых задействует объектно-ориентированное программирование. Весь код доступен, чтобы вы могли использовать его для создания собственных приложений. Я приведу несколько примеров.

5

ВВЕДЕНИЕ В PYGAME



Язык Python был спроектирован для обработки ввода и вывода текста. Это обеспечивает возможность получать текст и отправлять его пользователю, файлу и интернету. Однако у базового языка нет возможности работать с более современными понятиями, такими как окна, щелчки мыши, звуки и так далее. А что если вы захотите использовать Python для более современной, а не для текстовой программы? В этой главе я познакомлю вас с *pygame*, бесплатным внешним пакетом с открытым исходным кодом, который был разработан для расширения возможностей Python, чтобы программисты могли создавать игровые программы. Также вы можете использовать *pygame* для создания других видов интерактивных программ с графическим интерфейсом пользователя (GUI). Он дает возможность создавать окна, показывать изображения, распознавать движения и щелчки мыши, воспроизводить звуки и многое другое. Если коротко, он позволяет программистам Python создавать такие виды игр и приложений, с которыми уже знакомы нынешние пользователи компьютеров.

В мои намерения не входит превратить вас в разработчиков игр, хотя это могло бы быть забавным результатом. Скорее,

я буду использовать окружение `pygame`, чтобы сделать конкретные объектно-ориентированные методы программирования более ясными и наглядными. Когда вы работаете с `pygame`, чтобы объекты стали видимыми в окне, и имеете дело с пользователем, взаимодействующим с этими объектами, следует глубже понять, как эффективно использовать методы ООП.

В этой главе представлено общее введение в `pygame`, поэтому большая часть информации и примеров этой главы будет использовать процедурное кодирование. Начиная со следующей главы, я объясню, как эффективно использовать ООП с помощью `pygame`.

Устанавливаем Pygame

`Pygame` — это бесплатный загружаемый пакет. Для установки пакетов Python мы будем использовать пакетный менеджер *pip* (сокращение от `pip installs packages` – *pip устанавливает пакеты*). Как уже упоминалось во введении, я предполагаю, что вы загрузили официальную версию Python с python.org. Программа *pip* включена туда, поэтому у вас она уже есть.

В отличие от стандартного приложения, вы должны запустить *pip* из командной строки. На Mac запустите приложение *Терминал* (расположенное в подпапке *Утилиты* внутри папки *Приложения*). В системе Windows щелкните по пиктограмме Windows, введите `cmd` и нажмите **ENTER**.

ПРИМЕЧАНИЕ Эта книга не проверялась на системах Linux. Тем не менее большая часть (если не все) содержимого должна работать с минимальной подстройкой. Чтобы установить `pygame` в дистрибутив Linux, откройте Терминал любым привычным для вас способом.

Введите следующие команды в командную строку:

```
python3 -m pip install -U pip --user
python3 -m pip install -U pygame --user
```

Первая команда гарантирует, что у вас последняя версия программы *pip*. Вторая строка устанавливает самую последнюю версию `pygame`.

Если у вас возникают какие-то проблемы с установкой `pygame`, посмотрите документацию `pygame` по адресу <https://www.pygame.org/wiki/GettingStarted>. Чтобы проверить, что `pygame` был установлен правильно, откройте IDLE (среда разработки, которая поставляется в комплекте с реализацией по умолчанию Python) и в окне оболочки введите:

```
import pygame
```

Если вы видите сообщение, в котором говорится что-то типа «Hello from the pygame community», или если сообщение вообще отсутствует, тогда `pygame` был установлен правильно. Отсутствие сообщения об ошибке свидетельствует о том, что Python смог найти и загрузить пакет `pygame` и он готов к использованию. Если вы хотите посмотреть образец игры, использующей `pygame`, введите следующую команду (которая запустит версию игры *Космические захватчики*):

```
python3 -m pygame.examples.aliens
```

Прежде чем мы перейдем к использованию `pygame`, я должен объяснить две важные концепции. Во-первых, я покажу, как отдельные пиксели адресуются в программах, которые используют GUI. Во-вторых, я рассмотрю управляемые событиями программы и как они отличаются от типичных текстовых. Затем мы напишем код для нескольких приложений, которые демонстрируют ключевые признаки `pygame`.

Детали окон

Экран компьютера состоит из большого количества строк и столбцов маленьких точек — *пикселей* (от слова *элемент изображения* (*picture element*)). Пользователь взаимодействует с GUI-программой с помощью одного или более окон, каждое из которых представляет собой прямоугольную часть экрана. Программы могут контролировать цвет любого отдельного пикселя в своих окнах. Если вы работаете с несколькими GUI-программами, каждая обычно отображается в собственном окне. В этом разделе я рассмотрю, как адресовать и изменять отдельные пиксели в окне. Данные концепции не зависят от Python; они общие для всех компьютеров и используются во всех языках программирования.

Система координат окна

Возможно, вы знакомы с декартовой системой координат в сетке, как на рис. 5.1.

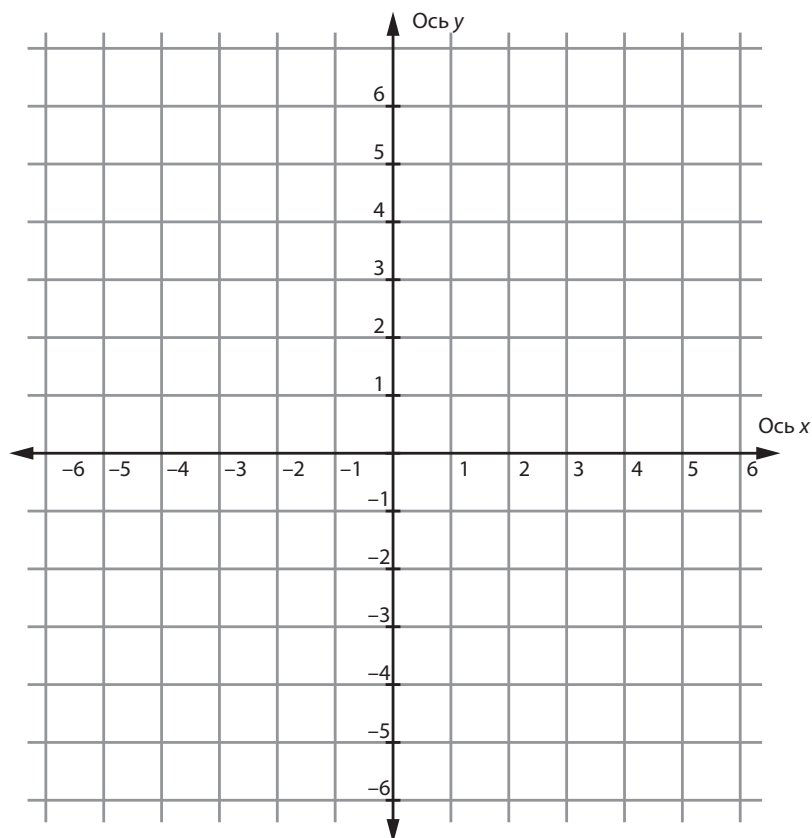


Рис. 5.1. Стандартная декартова система координат

Местоположение любой точки в декартовой сетке можно определить, указав ее координаты по осям x и y (в этом порядке). Началом считается точка, определяемая как $(0, 0)$ и находящаяся в центре сетки.

Координаты окна компьютера работают аналогичным образом (рис. 5.2).

Тем не менее есть несколько ключевых отличий.

1. Начальная точка $(0, 0)$ находится в верхнем левом углу окна.
2. Ось y перевернута так, что значения y начинаются с нуля в верхней части окна и увеличиваются по мере продвижения вниз.

3. Значения x и y — всегда целые числа. Каждая пара (x, y) определяет отдельный пиксель окна. Эти значения всегда указываются относительно верхнего левого угла окна, а не экрана. Таким образом, пользователь может перемещать окно в любую часть экрана, не влияя при этом на координаты элементов программы, отображаемой в окне.

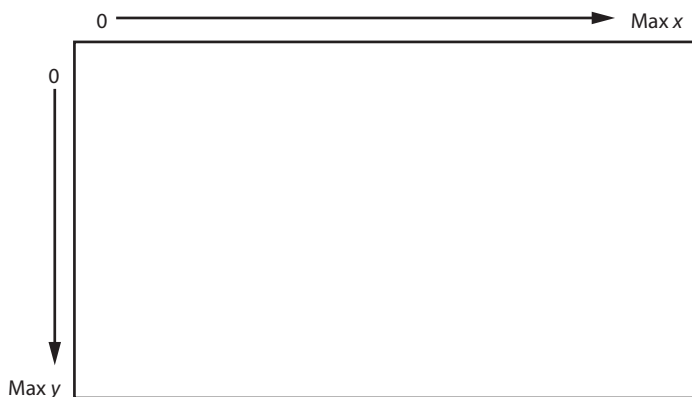


Рис. 5.2. Система координат окна компьютера

У полного экрана компьютера есть свой набор координат (x, y) для каждого пикселя, и он использует тот же тип системы координат, но программам редко, если вообще такое случается, приходится иметь дело с координатами экрана.

Когда мы пишем приложение `pygame`, необходимо указать ширину и высоту окна, которое хотим создать. Внутри окна мы можем адресовать любой пиксель, используя его координаты x и y , как показано на рис. 5.3.

На рис. 5.3 изображен черный пиксель на позиции $(3, 5)$. Это значение x , равное 3 (обратите внимание, что на самом деле это четвертый столбец, поскольку координаты начинаются со значения 0), и значение y , равное 5 (по факту шестая строка). Каждый пиксель в окне обычно называют *точкой*. Для ссылки на точку в окне вы, как правило, будете использовать кортеж Python. Например, у вас может быть подобный оператор присваивания, где значение x первое:

```
pixelLocation = (3, 5)
```

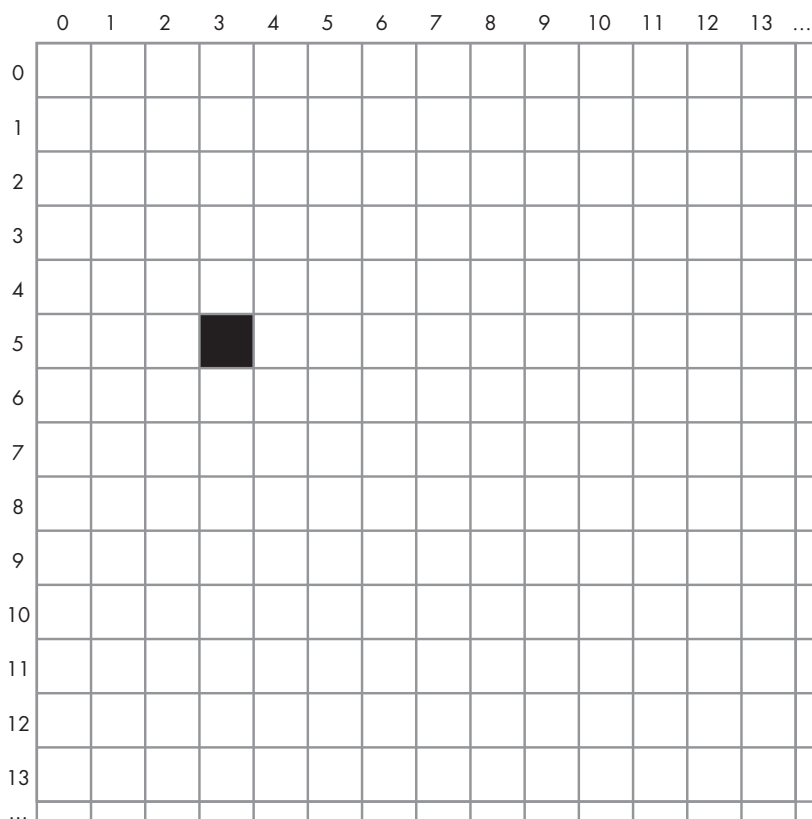


Рис. 5.3. Отдельная точка (отдельный пиксель) в окне компьютера

Чтобы представить изображение в окне, нам необходимо указать координаты его начальной точки — всегда верхний левый угол изображения — в виде пары (x, y) , как на рис. 5.4, где мы нарисовали изображение в местоположении $(3, 5)$.

При работе с изображением нам часто придется иметь дело с *ограничивающим прямоугольником* — самым маленьким прямоугольником, который можно создать и который полностью охватывает все пиксели изображения. В `pygame` прямоугольник представлен с помощью набора четырех значений: x , y , ширина, высота. Значения прямоугольника для изображения на рис. 5.4: 3, 5, 11, 7. Я покажу вам в следующем примере программы, как его использовать. Даже если изображение не является прямоугольником (например, если это круг или овал), все равно следует учитывать его ограничивающий прямоугольник для позиционирования и обнаружения коллизий.

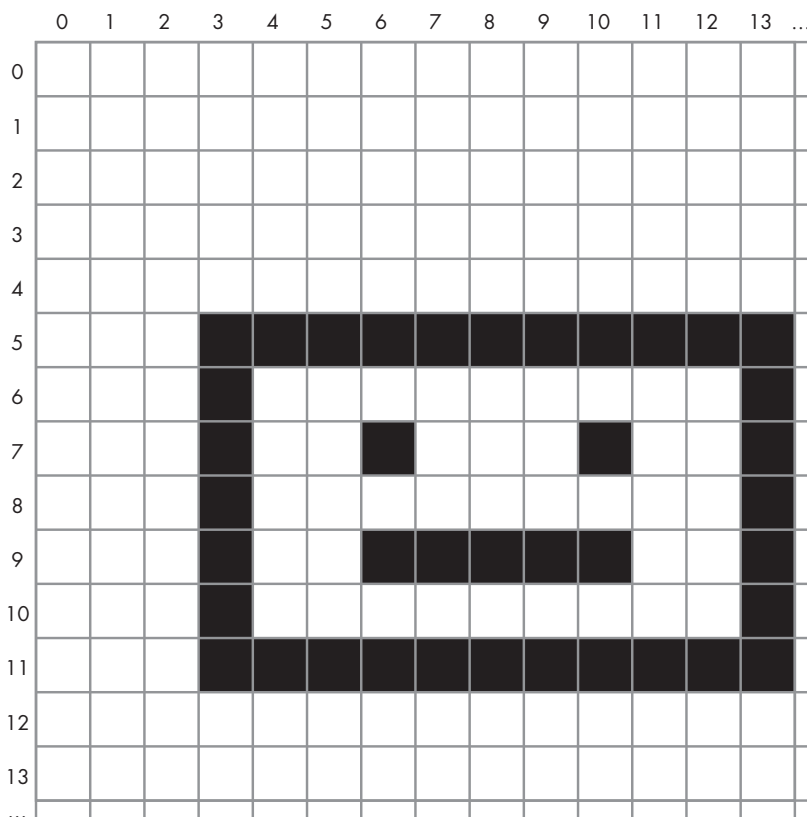


Рис. 5.4. Изображение в окне

Цвета пикселей

Давайте изучим, как представлены цвета на экране компьютера. Если у вас есть опыт работы с такими графическими программами, как Photoshop, вы, вероятно, уже знаете, как это работает, но, возможно, вы все равно захотите быстро освежить память.

Каждый пиксель на экране состоит из сочетания трех цветов: красного, зеленого и синего, что часто называется *RGB*. Цвет, отображаемый в каждом пикселе, состоит из некоторого количества красного, зеленого и синего, где количество каждого из них указывается как значение от 0 (что означает отсутствие цвета) до 255 (что означает полную интенсивность цвета). Таким образом, существует $256 \times 256 \times 256$ возможных сочетаний, или 16 777 216 (часто обозначается просто как «16 миллионов») цветов для каждого пикселя.

Цветам в `pygame` присваиваются RGB-значения, и мы записываем их как кортежи Python из трех чисел. Вот каким образом мы создаем константы для основных цветов:

```
RED = (255, 0, 0) # полный красный, нет зеленого, нет синего
GREEN = (0, 255, 0) # нет красного, полный зеленый, нет синего
BLUE = (0, 0, 255) # нет красного, нет зеленого, полный синий
```

Ниже представлены определения еще нескольких цветов. Вы можете создать оттенок, используя любое сочетание трех чисел от 0 до 255:

```
BLACK = (0, 0, 0) # нет красного, нет зеленого, нет синего
WHITE = (255, 255, 255) # полный красный, полный зеленый,
                        # полный синий
DARK_GRAY = (75, 75, 75)
MEDIUM_GRAY = (128, 128, 128)
LIGHT_GRAY = (175, 175, 175)
TEAL = (0, 128, 128) # нет красного, половинный зеленый,
                     # половинный синий
YELLOW = (255, 255, 0)
PURPLE = (128, 0, 128)
```

В `pygame` вам понадобится указывать цвета, когда вы хотите заполнить фон окна, нарисовать цветную форму или текст и так далее. Предварительное определение цветов как кортежей констант позволит вам легко обнаружить их позднее в коде.

Программы, управляемые событиями

В большинстве программ, которые были рассмотрены в книге до этого момента, основной код располагался в цикле `while`. Программа останавливается при вызове встроенной функции `input()` и ждет от пользователя входных данных для работы. Выходные данные программы, как правило, обрабатываются с помощью вызовов `print()`.

В интерактивных GUI-программах эта модель больше не работает. GUI представляют новую модель вычисления, известную как модель, *управляемая событиями*. Программы, управляемые событиями, не полагаются на `input()` и `print()`; вместо этого пользователь взаимодействует с элементами в окне, используя на свое усмотрение клавиатуру и/или мышь или другое

указывающее устройство. Они могут нажимать на различные кнопки и пиктограммы, выбирать из меню, вводить данные в текстовые поля или отдавать команды с помощью щелчков или нажатий клавиш, чтобы управлять каким-либо аватаром в окне.

ПРИМЕЧАНИЕ Вызов `print()` все еще может быть чрезвычайно полезен для отладки, когда используется для записи промежуточных результатов.

Центральным в программировании, управляемом событиями, является понятие *события*. События сложно определить, и их лучше всего описывать с помощью примеров, таких как щелчок мыши и нажатие клавиши (каждый из которых фактически состоит из двух событий: «мышь вниз» и «мышь вверх» и «клавиша вниз» и «клавиша вверх» соответственно). Ниже представлено мое рабочее определение.

Событие

Что-то, что происходит во время работы программы и на что она хочет или должна отреагировать. Большинство событий порождаются действиями пользователя.

Управляемая событиями GUI-программа постоянно работает в бесконечном цикле. Каждый раз проходя цикл, программа проверяет наличие каких-либо новых событий, на которые ей необходимо отреагировать, и исполняет соответствующий код для их обработки. Также во время каждого цикла программе необходимо перерисовывать все элементы в окне, чтобы обновить то, что видит пользователь.

Например, предположим, что у нас есть простая GUI-программа, которая отображает две кнопки: **Bark** и **Meow**. При щелчке по кнопке **Bark** она воспроизводит звук лающей собаки, а кнопка **Meow** воспроизводит звук мяукающей кошки (рис. 5.5).



Рис. 5.5. Простая программа с двумя кнопками

Пользователь может щелкать по этим кнопкам в любом порядке и в любое время. Чтобы обработать действия пользователя, программа работает циклически и постоянно проверяет, был ли щелчок по какой-либо из двух кнопок. Когда она получает событие «мышь вниз» на кнопке, программа понимает, что по кнопке щелкнули, и рисует изображение вдавленной кнопки. Когда она получает событие «мышь вверх» на кнопке, она запоминает новое состояние и перерисовывает кнопку в ее изначальный вид, а также воспроизводит соответствующий звук. Поскольку основной цикл выполняется очень быстро, пользователь воспринимает воспроизводимый звук сразу же после щелчка по кнопке. Каждый раз, проходя цикл, программа перерисовывает обе кнопки, показывая изображение, соответствующее текущему состоянию каждой кнопки.

Используем Rpygame

На первый взгляд rpygame может показаться чрезвычайно большим пакетом с различными доступными вызовами.

Но на самом деле вам необходимо понять не так много, чтобы приступить к работе с небольшой программой. Для знакомства с rpygame я сначала дам шаблон, который можно использовать для всех создаваемых вами программ rpygame. Затем я буду опираться на него, постепенно добавляя ключевые части функциональных возможностей.

В этом разделе я вам покажу, как:

- завести пустое окно;
- показать изображение;
- обнаружить щелчок мыши;
- обнаружить как одиночные, так и непрерывные нажатия клавиш;
- создать простую анимацию;
- воспроизвести звуковые эффекты и фоновые звуки;
- рисовать фигуры.

В следующей главе мы продолжим обсуждение rpygame и вы увидите, как:

- создавать анимацию для многих объектов;
- создавать кнопку и реагировать на нее;
- создавать поле отображения текста.

Создаем пустое окно

Как я уже упоминал ранее, программы `pygame` работают постоянно и циклически, проверяя наличие событий. Возможно, будет полезным воспринимать вашу программу как анимацию, в которой каждый проход основного цикла представляет собой один фрейм. Пользователь может щелкнуть на что-то во время любого фрейма, и ваша программа должна не только реагировать на эти входные данные, но также отслеживать все, что необходимо рисовать в окне. Например, в одном примере программы позднее в этой главе мы будем перемещать мяч по окну таким образом, чтобы в каждом фрейме мяч рисовался на немного другой позиции.

Листинг 5.1 представляет собой стандартный шаблон, который вы можете использовать как отправную точку для всех ваших программ `pygame`. Эта программа открывает окно и закрашивает все содержимое черным цветом. Единственное, что может сделать пользователь, — это щелкнуть по кнопке «закрыть», чтобы выйти из программы.

Файл: `PygameDemo0_WindowOnly/PygameWindowOnly.py`

```
# pygame демо 0 - только окно

#1 - Импортируем пакеты
import pygame
from pygame.locals import *
import sys

#2 - Определяем константы
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.

#5 - Инициализируем переменные

#6 - Бесконечный цикл
```

```

while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        # Нажата кнопка "закрыть"? Выходим из pygame и завершаем
        # программу
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    #8 - Выполняем действия "в рамках фрейма"

    #9 - Очищаем окно
    window.fill(BLACK)

    #10 - Рисуем все элементы окна

    #11 - Обновляем окно
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND)

```

Листинг 5.1. Шаблон для создания программ `pygame`

Давайте пройдемся по различным частям этого шаблона.

1. Импортируем пакеты.

Шаблон начинается с операторов `import`. Сначала мы импортируем сам пакет `pygame`, затем некоторые определенные внутри `pygame` константы, которые применим позднее. Последний импорт — это пакет `sys`, его мы будем использовать для выхода из программы.

2. Определяем константы.

Затем мы определяем любые константы для нашей программы. Сначала значение RGB для `BLACK`, которое мы будем использовать, чтобы нарисовать фон окна. Затем определим константы для ширины и высоты окна в пикселях и константы для частоты обновления программы. Это число определяет максимальное количество циклов (и, соответственно, перерисовки окна) программы в секунду. Наше значение 30 довольно типично. Если объем работы, выполняемой в вашем основном цикле, чрезмерен, программа может функционировать медленнее, чем это значение, но она никогда не будет работать

быстрее. Слишком высокая частота обновления может привести к слишком быстрой работе программы. В нашем примере с мячом это означает, что мяч может скакать в окне быстрее, чем предполагалось.

3. Инициализируем окружение `pygame`.

В этом разделе мы вызываем функцию, которая говорит `pygame` инициализировать себя самого. Затем просим `pygame` создать окно для нашей программы с помощью функции `pygame.display.set_mode()` и передаем желаемую ширину и высоту окна. И наконец, вызываем еще одну функцию `pygame`, чтобы создать объект часов, который будет использоваться в конце нашего основного цикла для поддержания максимальной частоты фреймов.

4. Загружаем элементы: изображения, звуки и так далее.

Это раздел-заполнитель, в который мы в итоге добавляем код, чтобы загружать внешние изображения, звуки и так далее из диска для использования в программе. Здесь мы не задействуем какие-либо внешние элементы, потому на данный момент этот раздел пустой.

5. Инициализируем переменные.

Здесь мы наконец инициализируем любые переменные, которые наша программа будет использовать. На текущий момент таких не имеется, поэтому кода тут нет.

6. Бесконечный цикл.

Здесь мы начинаем основной цикл. Это простой бесконечный цикл `while True`. И вновь вы можете воспринимать каждую итерацию основного цикла как фрейм в анимации.

7. Проверяем наличие событий и обрабатываем их; обычно называется *цикл событий*.

В этом разделе мы вызываем `pygame.event.get()`, чтобы получить список событий, которые произошли с момента последней проверки (с момента последнего запуска основного цикла), затем проводим итерацию списка событий. Каждое событие, о котором сообщается программе, является объектом, а у каждого объекта события есть тип. Если никаких событий не произошло, данный раздел пропускается.

В этой минимальной программе, где единственное действие, доступное пользователю, — закрытие окна, есть только один тип события, наличие которого мы проверяем, — это константа

`pygame.QUIT`, генерируемая `pygame`, когда пользователь щелкает по кнопке закрытия. Если мы нашли это событие, `pygame` завершает работу, освобождая ресурсы, которые она использовала. Затем мы выходим из программы.

8. Выполняем действия «в рамках фрейма».

В этом разделе мы наконец помещаем код, который необходимо выполнить, в каждом фрейме. Он может включать перемещение объектов в окне или проверку наличия коллизий между элементами. В этой конкретной программе нам здесь нечего делать.

9. Очищаем окно.

В каждой итерации основного цикла программа должна перерисовывать все в окне, так что для начала мы должны его очистить. Самый простой подход — просто заполнить окно цветом, что мы здесь и делаем, вызывая `window.fill()`, указав черный фон. Мы также нарисуем фоновую картинку, но пока отложим это.

10. Рисуем все элементы окна.

Здесь мы поместим код, чтобы рисовать все, что захотим отобразить в окне. В этом шаблоне программы рисовать нечего.

В реальных программах объекты рисуются в том порядке, в котором они появляются в коде, слоями от самого заднего к самому переднему. Например, предположим, что мы хотим нарисовать два частично пересекающихся круга, А и В. Если сначала нарисуем А, он появится за В и какие-то его части будут скрыты. Если мы сначала нарисуем В, а затем А, произойдет обратное и мы увидим А перед В. Это естественное отображение, эквивалентное слоям в графических программах, таких как Photoshop.

11. Обновляем окно.

Это строка говорит `pygame` взять все включенные нами рисунки и отобразить их в окне. `pygame` фактически выполняет все изображения на этапах 8, 9 и 10 во внеэкранный буфер. Когда вы говорите `pygame` обновиться, он берет содержимое этого внеэкранного буфера и помещает его в реальное окно.

12. Делаем короткую паузу.

Компьютеры очень быстры, и, если цикл будет переходить к следующей итерации без пауз, программа может работать быстрее, чем указанная частота фреймов. Строка в этом разделе говорит `pygame` подождать, пока пройдет заданное

количество времени, чтобы фреймы программы работали в рамках указанной нами частоты. Это важно, чтобы программа работала с постоянной скоростью независимо от скорости компьютера.

Когда вы запускаете эту программу, она просто выводит пустое окно, заполненное черным цветом. Чтобы завершить программу, щелкните по кнопке «закрыть» в строке заголовка.

Рисуем изображение

Давайте нарисуем что-нибудь в окне. Чтобы показать графическое изображение, необходимо выполнить два действия: сначала загружаем изображение в память компьютера, затем отображаем его в окне приложения.

При работе с `pygame` все изображения (и звуки) необходимо хранить в файлах, внешних по отношению к вашему коду. `pygame` поддерживает многие стандартные графические форматы файлов, включая `.png`, `.jpg` и `.gif`. В этой программе мы загрузим картинку мяча из файла `ball.png`. Напомню, что код и связанные со всеми основными листингами элементы в этой книге доступны к загрузке по адресу <https://addons.eksmo.ru/it/OOP-Code.zip>.



Рис. 5.6. Предложенная иерархия папки проекта

Поскольку нам нужен лишь один графический файл в этой программе, хорошей идеей будет использовать последовательный подход к обработке графических и звуковых файлов, поэтому я вам покажу один из них здесь. Сначала создайте папку проекта. Поместите в нее вашу основную программу вместе со всеми связанными файлами, содержащими классы и функции Python. Затем внутри папки проекта создайте папку `images`, куда вы поместите любые файлы изображений, которые хотите использовать в вашей программе. Также создайте папку `sounds` и поместите в нее любые звуковые файлы, которые вы хотите здесь использовать. На рис. 5.6 представлена предложенная

структура. Все примеры программ в этой книге будут использовать этот макет папки проекта.

Путь (pathname) — это строка, которая однозначно определяет местоположение файла или папки на компьютере. Чтобы загрузить графический или звуковой файл в программу, вы должны указать путь к файлу. Существует два типа путей: относительный и абсолютный.

Относительный путь — это путь относительно текущей папки, часто называемой *текущий рабочий каталог*. Когда вы выполняете программу, используя интегрированную среду разработки, такую как IDLE или PyCharm, она устанавливает текущую папку в ту, которая содержит вашу основную программу Python, чтобы вы могли с легкостью использовать относительные пути. В этой книге я подразумеваю, что вы используете интегрированную среду разработки, и буду представлять все пути как относительные.

Относительным путем для графического файла (например, *ball.png*) в той же папке, что и ваш основной файл Python, будет всего лишь имя файла в виде строки (например, «*ball.png*»). Если использовать предложенную структуру проекта, относительный путь для файла будет «*images/ball.png*».

Это говорит о том, что внутри папки проекта есть другая с именем *images*, а внутри нее находится файл под названием *ball.png*. В строках путей имена папок разделяются с помощью символов косой черты.

Тем не менее если вы собираетесь запускать вашу программу из командной строки, то вам необходимо создать абсолютные пути для всех файлов. *Абсолютный путь* — это тот путь, который начинается с корневого каталога файловой системы и включает полную иерархию папок для вашего файла. Чтобы выстроить абсолютный путь к любому файлу, вы можете использовать подобный код, который выстраивает строку абсолютного пути для файла *ball.png* в каталоге *images* внутри папки проекта:

```
from pathlib import Path

# помещаем в раздел #2, определяя константу
BASE_PATH = Path(__file__).resolve().parent

# выстраиваем путь к файлу в папке images
pathToBall = BASE_PATH + 'images/ball.png'
```

Теперь мы создадим код в программе мяча, начиная с ранее упомянутого 12-шагового шаблона и добавляя всего лишь две новые строки кода, как показано в листинге 5.2.

Файл: PygameDemo1_OneImage/PygameOneImage.py

```
# pygame демо 1 - рисуем одно изображение

--- пропуск ---

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.
❶ ballImage = pygame.image.load('images/ball.png')

#5 - Инициализируем переменные

--- пропуск ---

#10 - Рисуем все элементы окна
# рисуем мяч на позиции 100 вдоль (x) и 200 вниз по (y)
❷ window.blit(ballImage, (100, 200))

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) # ожидание pygame
```

Листинг 5.2. Загрузить одно изображение и нарисовать его в каждом фрейме

Сначала мы с помощью `pygame` ищем файл, содержащий изображение мяча, и загружаем его в память ❶. Переменная `ballImage` теперь относится к изображению мяча. Обратите внимание, что этот оператор присваивания выполняется только один раз перед началом основного цикла.

ПРИМЕЧАНИЕ В официальной документации `pygame` каждое изображение, включая окно приложения, известно как *поверхность*. Я буду использовать более специализированные термины: называть окно приложения просто *окном*, а любую картинку, загруженную из внешнего файла, — *изображением*. Термин *поверхность* приберегу для любой картинки, нарисованной на ходу.

Затем мы рисуем мяч ❷ каждый раз, когда проходим основной цикл. Мы указываем место, которое представляет собой позицию для размещения верхнего левого угла ограничивающего прямоугольника изображения, обычно в виде кортежа x и y координат.

Функция под названием `blit()` — это очень старая отсылка к словосочетанию *передача битового блока* (*bit block transfer*), но в данном контексте оно на самом деле обозначает лишь «нарисовать». Поскольку программа ранее загрузила изображение мяча, `pygame` знает размер изображения, поэтому нам необходимо лишь сообщить ему, где именно его рисовать. В листинге 5.2 мы присвоили оси x значение 100, а оси y — значение 200.

Когда вы запускаете программу, на каждой итерации цикла (30 раз в секунду) каждый пиксель в окне становится черным, затем мяч рисуется поверх фона. С точки зрения пользователя это выглядит так, будто ничего не происходит: мяч просто остается в одной точке с верхним левым углом его ограничивающего прямоугольника в местоположении (100, 200).

Обнаруживаем щелчок мыши

Далее мы позволим нашей программе обнаружить щелчок мыши и отреагировать на него. Пользователь сможет щелкнуть по мячу, чтобы он появился в другом месте окна. Когда программа обнаруживает щелчок мыши по мячу, она случайным образом выбирает новые координаты и рисует мяч в этом новом местоположении. Вместо того чтобы использовать жестко закодированные координаты (100, 200), мы создадим две переменные `ballX` и `ballY` и будем ссылаться на координаты мяча в окне в виде кортежа (`ballX`, `ballY`). В листинге 5.3 приведен код.

Файл: `PygameDemo2_ImageClickAndMove/PygameImageClickAndMove.py`

pygame демо 2 — одно изображение, щелчок и перемещение

```
#1 — Импортируем пакеты
import pygame
from pygame.locals import *
import sys
❶ import random
```

```

#2 - Определяем константы
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
❷ BALL_WIDTH_HEIGHT = 100
MAX_WIDTH = WINDOW_WIDTH - BALL_WIDTH_HEIGHT
MAX_HEIGHT = WINDOW_HEIGHT - BALL_WIDTH_HEIGHT

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.
ballImage = pygame.image.load('images/ball.png')

#5 - Инициализируем переменные
❸ ballX = random.randrange(MAX_WIDTH)
ballY = random.randrange(MAX_HEIGHT)
❹ ballRect = pygame.Rect(ballX, ballY, BALL_WIDTH_HEIGHT,
                          BALL_WIDTH_HEIGHT)

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        # Нажата кнопка закрытия? Выходим из pygame и завершаем
        # программу
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        # определяем, щелкнул ли пользователь
        ❺ if event.type == pygame.MOUSEBUTTONDOWN:
            # mouseX, mouseY = event.pos
            # Могли бы это сделать при необходимости

            # проверяем, был ли щелчок в пределах прямоугольника мяча
            # Если это так, выбираем случайным образом новое
            # местоположение
            ❻ if ballRect.collidepoint(event.pos):
                ballX = random.randrange(MAX_WIDTH)
                ballY = random.randrange(MAX_HEIGHT)
                ballRect = pygame.Rect(ballX, ballY,
                                       BALL_WIDTH_HEIGHT, BALL_WIDTH_HEIGHT)

```

```

#8 - Выполняем действия "в рамках фрейма"

#9 - Очищаем окно
window.fill(BLACK)

#10 - Рисуем все элементы окна
# рисуем мяч в произвольном местоположении
7 window.blit(ballImage, (ballX, ballY))

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 5.3. Обнаружение щелчка мыши и действие на основании этого

Поскольку нам необходимо сгенерировать случайные числа для координат мяча, мы импортируем пакет `random` ❶.

Затем добавляем новую константу, чтобы определить высоту и ширину нашего изображения как 100 пикселей ❷. Также мы создаем еще две константы, чтобы ограничить максимальную ширину и высоту координат. Используя их вместо размера окна, мы делаем так, чтобы наш мяч все время отображался полностью внутри окна (помните, что, когда ссылаемся на местоположение изображения, мы указываем позицию его верхнего левого угла). Используем эти константы, чтобы выбрать случайные значения для начальных координат x и y ❸.

Затем вызываем `pygame.Rect()`, чтобы создать прямоугольник ❹. Для определения прямоугольника необходимо четыре параметра: координаты x , координаты y , ширина и высота, — в следующем порядке:

```
<rectObject> = pygame.Rect(<x>, <y>, <width>, <height>)
```

Это возвращает прямоугольный объект `pygame` или `rect`. Мы будем использовать прямоугольник мяча в обработке событий.

Нам также необходимо добавить код, чтобы проверить, щелкнул ли пользователь мышью. Как было упомянуто, щелчок мыши фактически состоит из двух различных событий: события «мышь вниз» и события «мышь вверх». Поскольку событие «мышь вверх» обычно используется для подачи сигнала

об активации, мы здесь будем искать лишь это событие. О нем сигнализирует новое значение `event.type pygame.MOUSEBUTTONDOWN` ⑤.

Когда выявим, что событие «мышь вверх» произошло, мы проверим, находилось ли местоположение, по которому щелкнул пользователь, внутри текущего прямоугольника мяча.

Когда `pygame` обнаружит, что событие произошло, он создаст объект события, содержащий большое количество данных. В данном случае нас волнуют лишь координаты x и y того, где произошло событие. Мы извлекаем позицию (x, y) щелчка с помощью `event.pos`, которая предоставляет кортеж из двух значений.

ПРИМЕЧАНИЕ Если нам необходимо разделить координаты x и y щелчка, мы можем распаковать кортеж и сохранить значения в виде двух переменных следующим образом:

```
mouseX, mouseY = event.pos
```

Теперь проверим, произошло ли событие внутри прямоугольника мяча, с помощью `collidepoint()` ⑥, чей синтаксис следующий:

```
<booleanVariable> = <someRectangle>.collidepoint(<someXYLocation>)
```

Метод возвращает булево выражение `True`, если заданная точка находится внутри прямоугольника. Если пользователь щелкнул по мячу, мы случайным образом выбираем новые значения для `ballX` и `ballY`. Мы используем их, чтобы создать новый прямоугольник для мяча в новом произвольном местоположении.

Единственное изменение здесь заключается в том, что мы всегда рисуем мяч в местоположении, заданном кортежем `(ballX, ballY)` ⑦. В результате каждый раз, когда пользователь щелкает внутри прямоугольника мяча, мяч перемещается в какую-то новую, произвольно выбранную точку в окне.

Обрабатываем клавиатуру

Следующий шаг заключается в том, чтобы помочь пользователю управлять некоторыми аспектами программы с помощью клавиатуры. Существует два различных способа управления взаимодействиями пользователя с клавиатурой: с помощью одиночных нажатий и когда пользователь удерживает клавишу,

чтобы обозначить, что действие должно продолжаться, пока зажата клавиша (также известно как *непрерывная обработка*).

Распознаем одиночные нажатия клавиш

Как и в случае со щелчками мышью, каждое нажатие клавиши генерирует два события: клавиша вниз и клавиша вверх. У этих двух событий есть два различных типа событий: `pygame.KEYDOWN` и `pygame.KEYUP`.

В листинге 5.4 продемонстрирован небольшой пример программы, которая позволяет пользователю переместить изображение мяча в окне с помощью клавиатуры. Цель пользователя состоит в перемещении изображения мяча таким образом, чтобы оно перекрывалось целевым изображением.

Файл: PygameDemo3_MoveByKeyboard/ PygameMoveByKeyboardOncePerKey.py

```
# pygame демо 3 - одно изображение, управление клавиатурой

#1 - Импортируем пакеты
import pygame
from pygame.locals import *
import sys
import random

#2 - Определяем константы
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
BALL_WIDTH_HEIGHT = 100
MAX_WIDTH = WINDOW_WIDTH - BALL_WIDTH_HEIGHT
MAX_HEIGHT = WINDOW_HEIGHT - BALL_WIDTH_HEIGHT
❶ TARGET_X = 400
TARGET_Y = 320
TARGET_WIDTH_HEIGHT = 120
N_PIXELS_TO_MOVE = 3

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.
ballImage = pygame.image.load('images/ball.png')
❷ targetImage = pygame.image.load('images/target.jpg')
```

```

#5 - Инициализируем переменные
ballX = random.randrange(MAX_WIDTH)
ballY = random.randrange(MAX_HEIGHT)
targetRect = pygame.Rect(TARGET_X, TARGET_Y, TARGET_WIDTH_HEIGHT,
                          TARGET_WIDTH_HEIGHT)

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        # Нажата кнопка закрытия? Выходим из pygame и завершаем
        # программу
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        # определяем, нажал ли пользователь клавишу
    3 elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_LEFT:
            ballX = ballX - N_PIXELS_TO_MOVE
        elif event.key == pygame.K_RIGHT:
            ballX = ballX + N_PIXELS_TO_MOVE
        elif event.key == pygame.K_UP:
            ballY = ballY - N_PIXELS_TO_MOVE
        elif event.key == pygame.K_DOWN:
            ballY = ballY + N_PIXELS_TO_MOVE

    #8 - Выполняем действия "в рамках фрейма"
    # определяем, перекрывает ли мяч целевое изображение
    4 ballRect = pygame.Rect(ballX, ballY, BALL_WIDTH_HEIGHT,
                            BALL_WIDTH_HEIGHT)
    5 if ballRect.colliderect(targetRect):
        print('Ball is touching the target')

    #9 - Очищаем окно
    window.fill(BLACK)

    #10 - Рисуем все элементы окна
    6 window.blit(targetImage, (TARGET_X, TARGET_Y)) # рисуем
                                                    # целевое изображение
    window.blit(ballImage, (ballX, ballY)) # рисуем мяч

    #11 - Обновляем окно
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 5.4. Обнаружение одиночного нажатия клавиши и действие на основании этого

Сначала мы добавим несколько новых констант ❶, чтобы определить координаты x и y верхнего левого угла целевого прямоугольника и ширину и высоту целевого изображения. Затем перезагружаем изображение целевого прямоугольника ❷.

В цикле, где ищем события, мы добавляем проверку для нажатия клавиши на наличие типа события `pygame.KEYDOWN` ❸. Если обнаружено событие «клавиша вниз», мы просматриваем его, чтобы найти, какая клавиша была нажата. У каждой клавиши есть соответствующая ей константа в `pygame`, поэтому здесь мы проверяем, нажал ли пользователь клавиши стрелок влево, вверх, вниз или вправо. Для каждой из них мы соответствующим образом изменяем значение координат x и y мяча на небольшое количество пикселей.

Затем создаем объект `pygame.rect` для мяча на основании его координат x и y и его высоты и ширины ❹. Мы можем проверить, перекрывают ли друг друга два прямоугольника, с помощью следующего вызова:

```
<booleanVariable> = <rect1>.collidect (<rect2>)
```

Этот вызов сравнивает два прямоугольника и возвращает `True`, если они перекрывают друг друга, и `False`, если нет. Мы сравниваем прямоугольник мяча с целевым прямоугольником ❺, и, если они перекрывают друг друга, программа в окне оболочки выводит: «Мяч касается цели».

Последнее изменение касается того, где мы рисуем и целевое изображение, и мяч. Сначала рисуется целевое изображение, чтобы, если они пересекаются, мяч отображался поверх целевого изображения ❻.

При выполнении программы, если прямоугольник мяча перекрывает прямоугольник целевого изображения, выводится сообщение в окне оболочки. Если вы уберете мяч с целевого изображения, сообщение прекратит выводиться.

Работаем с непрерывным нажатием клавиш в непрерывном режиме

Вторым способом обработки взаимодействия с клавиатурой в `pygame` является *опрос* клавиатуры. Он включает запрос у `pygame` списка, представляющего, какие клавиши на текущий момент нажаты в каждом фрейме, используя следующий вызов:

```
<aTuple> = pygame.key.get_pressed()
```

Этот вызов возвращает кортеж из нулей и единиц, представляющий состояние каждой клавиши: 0 — если клавиша вверх, 1 — если клавиша вниз. Затем вы можете использовать константы, определенные в `pygame`, в качестве индекса в возвращенном кортеже, чтобы увидеть, нажата ли конкретная клавиша. Например, следующие строки можно использовать для определения состояния клавиши A:

```
keyPressedTuple = pygame.key.get_pressed()
# используем константу, чтобы получить соответствующий
# элемент кортежа
aIsDown = keyPressedTuple[pygame.K_a]
```

Полный список констант, представляющий все клавиши, определенные в `pygame`, вы можете найти по адресу <https://www.pygame.org/docs/ref/key.html>.

Код в листинге 5.5 показывает, как мы можем использовать этот метод для непрерывного перемещения изображения вместо одного перемещения за одно нажатие клавиши. В этой версии мы перемещаем обработку клавиатуры из раздела #7 в раздел #8. Остальная часть кода идентична предыдущей версии листинга 5.4.

Файл: `PygameDemo3_MoveByKeyboard/PygameMoveByKeyboardContinuous.py`

```
# pygame демо 3 - одно изображение, непрерывный режим, перемещать,
# пока зажата клавиша
```

```
--- пропуск ---
```

```
#7 - Проверяем наличие событий и обрабатываем их
for event in pygame.event.get():
    # Нажата кнопка закрытия? Выходим из pygame и завершаем
    # программу
    if event.type == pygame.QUIT:
        pygame.quit()
        sys.exit()
```

```
#8 - Выполняем действия "в рамках фрейма"
```

```
# Проверяем нажатия клавиш пользователем
```

```
❶ keyPressedTuple = pygame.key.get_pressed()
```

```
if keyPressedTuple[pygame.K_LEFT]: # перемещаемся влево
    ballX = ballX - N_PIXELS_TO_MOVE
```

```

if keyPressedTuple[pygame.K_RIGHT]: # перемещаемся вправо
    ballX = ballX + N_PIXELS_TO_MOVE

if keyPressedTuple[pygame.K_UP]: # перемещаемся вверх
    ballY = ballY - N_PIXELS_TO_MOVE

if keyPressedTuple[pygame.K_DOWN]: # перемещаемся вниз
    ballY = ballY + N_PIXELS_TO_MOVE

# определяем, перекрывает ли мяч целевое изображение
ballRect = pygame.Rect(ballX, ballY,
                        BALL_WIDTH_HEIGHT, BALL_WIDTH_HEIGHT)
if ballRect.colliderect(targetRect):
    print('Ball is touching the target')

--- пропуск ---

```

Листинг 5.5. Обработка зажатых клавиш

Код обработки клавиатуры в листинге 5.5 не полагается на события, поэтому мы помещаем новый код за пределами цикла `for`, который проводит итерацию всех событий, возвращенных `pygame` ❶.

Поскольку мы осуществляем эту проверку в каждом фрейме, движение мяча будет казаться непрерывным, пока пользователь удерживает клавишу. Например, если пользователь нажимает и удерживает клавишу правой стрелки, этот код будет добавлять 3 к значению координаты `ballX` в каждом фрейме, а пользователь будет видеть, что мяч плавно перемещается вправо. Когда он прекратит нажимать клавишу, перемещение остановится.

Другое изменение состоит в том, что этот подход позволяет вам проверять наличие нескольких клавиш, нажатых одновременно. Например, если пользователь нажимает и удерживает клавиши левой и нижней стрелок, мяч будет перемещаться по диагонали влево. Вы можете проверить наличие такого количества нажатых клавиш, какое пожелаете. Однако количество *одновременно* нажатых клавиш, которые можно обнаружить, ограничено операционной системой, компьютерной клавиатурой и многими другими факторами. Обычно лимит составляет четыре клавиши, но ваш лимит может отличаться.

Создаем анимацию, основанную на местоположении

Далее мы построим анимацию, основанную на местоположении. Этот код позволит нам перемещать изображение по диагонали, а затем сделать так, чтобы оно отскакивало от краев окна. Это был любимый метод заставок на старых ЭЛТ-мониторах, позволявший избегать выгорания статичного изображения.

Мы немного изменим местоположение нашего изображения в каждом фрейме. Также проверим, поместят ли эти перемещения какую-либо часть изображения за пределы одной из границ окна, и, если это так, изменим перемещение в этом направлении. Например, если изображение перемещается вниз и будет пересекать нижнюю часть окна, мы изменим направление и заставим изображение двигаться вверх.

И вновь мы воспользуемся тем же начальным шаблоном.

В листинге 5.6 представлен полный исходный код.

Файл: PygameDemo4_OneBallBounce/PygameOneBallBounceXY.py

```
# pygame демо 4 (a) - одно изображение, отскакивает от границ окна
# с использованием координат (x, y)
```

```
#1 - Импортируем пакеты
```

```
import pygame
from pygame.locals import *
import sys
import random
```

```
#2 - Определяем константы
```

```
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
BALL_WIDTH_HEIGHT = 100
N_PIXELS_PER_FRAME = 3
```

```
#3 - Инициализируем окружение pygame
```

```
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()
```

```
#4 - Загружаем элементы: изображения, звуки и т. д.
```

```
ballImage = pygame.image.load('images/ball.png')
```

```
#5 - Инициализируем переменные
```

```
MAX_WIDTH = WINDOW_WIDTH - BALL_WIDTH_HEIGHT
```

```

MAX_HEIGHT = WINDOW_HEIGHT - BALL_WIDTH_HEIGHT
❶ ballX = random.randrange(MAX_WIDTH)
ballY = random.randrange(MAX_HEIGHT)
xSpeed = N_PIXELS_PER_FRAME
ySpeed = N_PIXELS_PER_FRAME

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        # Нажата кнопка закрытия? Выходим из pygame и завершаем
        # программу
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    #8 - Выполняем действия "в рамках фрейма"
    ❷ if (ballX < 0) or (ballX >= MAX_WIDTH):
        xSpeed = -xSpeed # обращаем направление X

    if (ballY < 0) or (ballY >= MAX_HEIGHT):
        ySpeed = -ySpeed # обращаем направление Y

    # обновляем местоположение мяча, используя скорость в двух
    # направлениях
    ❸ ballX = ballX + xSpeed
    ballY = ballY + ySpeed

    #9 - Очищаем окно, прежде чем рисовать его заново
    window.fill(BLACK)

    #10 - Рисуем все элементы окна
    window.blit(ballImage, (ballX, ballY))

    #11 - Обновляем окно
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND)

```

Листинг 5.6. Основанная на местоположении анимация с мячом, отскакивающим от границ окна

Начинаем с создания и инициализации двух переменных `xSpeed` и `ySpeed` ❶, которые определяют, насколько далеко и в каком направлении должно перемещаться изображение

в каждом фрейме. Мы инициализируем обе переменные количеством перемещаемых во фрейме пикселей (3), чтобы изображение начало перемещаться на три пикселя вправо (положительное направление x) и на три пикселя вниз (положительное направление y).

В ключевой части программы мы обрабатываем координаты x и y отдельно ❷. Сначала мы проверяем, является ли значение координаты x мяча меньше нуля, что обозначает, что часть изображения выходит за левый край, или больше пикселя `MAX_WIDTH`, то есть фактически выходит за правый край. Если один из этих вариантов верен, мы изменяем знак скорости в направлении x , то есть изображение пойдет в противоположном направлении. Например, если мяч двигался вправо и зашел за правый край, мы изменим значение скорости `xSpeed` с 3 на -3, чтобы мяч стал двигаться влево, и наоборот.

Затем проводим аналогичную проверку координат y , чтобы мяч отскакивал от верхнего и нижнего края, если это необходимо.

И наконец, обновим позицию мяча, добавив `xSpeed` к координате `ballX` и `ySpeed` к координате `ballY` ❸. Это помещает его в новое местоположение по обеим осям.

В нижней части основного цикла мы рисуем мяч. Поскольку значения `ballX` и `ballY` обновляются в каждом фрейме, кажется, что мяч плавно анимируется. Попробуйте это. Каждый раз, когда мяч достигает края, кажется, будто он отскакивает.

Используем `rect` в `pygame`

Далее я представлю различные способы достижения одного и того же результата. Вместо того чтобы отслеживать текущие координаты x и y мяча в отдельных переменных, мы будем использовать `rect` мяча, обновлять `rect` в каждом фрейме и проверять, приведет ли обновление к выходу за границы окна какой-либо части `rect`. В результате мы получим меньше переменных, и, поскольку мы начнем с вызова для получения `rect` изображения, оно будет работать с изображениями любого размера.

Когда вы создаете объект `rect`, в дополнение к запоминанию `left`, `top`, `width` и `height` в качестве атрибутов этот объект также вычисляет и поддерживает для вас несколько других атрибутов. Вы можете получить доступ к любому из этих

атрибутов непосредственно по имени, используя *точечный синтаксис*, как показано в табл. 5.1. (Я представлю больше информации по этой теме в главе 8.)

Таблица 5.1. Прямой доступ к атрибутам `rect`

Атрибут	Описание
<code><rect>.x</code>	Координаты x левого края <code>rect</code>
<code><rect>.y</code>	Координаты y верхнего края <code>rect</code>
<code><rect>.left</code>	Координаты x левого края <code>rect</code> (то же, что и <code><rect>.x</code>)
<code><rect>.top</code>	Координаты y верхнего края <code>rect</code> (то же, что и <code><rect>.y</code>)
<code><rect>.right</code>	Координаты x правого края <code>rect</code>
<code><rect>.bottom</code>	Координаты y нижнего края <code>rect</code>
<code><rect>.topleft</code>	Кортеж из двух целых чисел: координаты верхнего левого угла <code>rect</code>
<code><rect>.bottomleft</code>	Кортеж из двух целых чисел: координаты нижнего левого угла <code>rect</code>
<code><rect>.topright</code>	Кортеж из двух целых чисел: координаты верхнего правого угла <code>rect</code>
<code><rect>.bottomright</code>	Кортеж из двух целых чисел: координаты нижнего правого угла <code>rect</code>
<code><rect>.midtop</code>	Кортеж из двух целых чисел: координаты средней точки верхнего края <code>rect</code>
<code><rect>.midleft</code>	Кортеж из двух целых чисел: координаты средней точки левого края <code>rect</code>
<code><rect>.midbottom</code>	Кортеж из двух целых чисел: координаты средней точки нижнего края <code>rect</code>
<code><rect>.midright</code>	Кортеж из двух целых чисел: координаты средней точки правого края <code>rect</code>
<code><rect>.center</code>	Кортеж из двух целых чисел: координаты центра <code>rect</code>
<code><rect>.centerx</code>	Координаты x центра ширины <code>rect</code>
<code><rect>.centery</code>	Координаты y центра высоты <code>rect</code>
<code><rect>.size</code>	Кортеж из двух целых чисел: (ширина, высота) <code>rect</code>
<code><rect>.width</code>	Ширина <code>rect</code>
<code><rect>.height</code>	Высота <code>rect</code>
<code><rect>.w</code>	Ширина <code>rect</code> (то же, что и <code><rect>.width</code>)
<code><rect>.h</code>	Высота <code>rect</code> (то же, что и <code><rect>.height</code>)

Объект `rect` `pygame` также допустимо представить в виде списка из четырех элементов и получить к нему доступ. В частности, вы можете использовать индекс, чтобы получить или установить любую отдельную часть `rect`. Например, используя `ballRect`, можно получить доступ к отдельным элементам следующим образом:

- `ballRect[0]` — это значение x (но вы также можете использовать `ballRect.left`);
- `ballRect[1]` — это значение y (но вы также можете использовать `ballRect.top`);
- `ballRect[2]` — это ширина (но вы также можете использовать `ballRect.width`);
- `ballRect[3]` — это высота (но вы также можете использовать `ballRect.height`).

Листинг 5.7 является альтернативной версией нашей программы с прыгающими мячами, которая поддерживает всю информацию о мяче в прямоугольном объекте.

Файл: PygameDemo4_OneBallBounce/ PygameOneBallBounceRects.py

```
# pygame демо 4(b) - одно изображение, отскакивает от границ окна
# с помощью rect
```

```
#1 - Импортируем пакеты
import pygame
from pygame.locals import *
import sys
import random
```

```
#2 - Определяем константы
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
N_PIXELS_PER_FRAME = 3
```

```
#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()
```

```
#4 - Загружаем элементы: изображения, звуки и т. д.
ballImage = pygame.image.load('images/ball.png')
```



```

#5 - Инициализируем переменные
❶ ballRect = ballImage.get_rect()
MAX_WIDTH = WINDOW_WIDTH - ballRect.width
MAX_HEIGHT = WINDOW_HEIGHT - ballRect.height
ballRect.left = random.randrange(MAX_WIDTH)
ballRect.top = random.randrange(MAX_HEIGHT)
xSpeed = N_PIXELS_PER_FRAME
ySpeed = N_PIXELS_PER_FRAME

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        # Нажата кнопка закрытия? Выходим из pygame и завершаем
        # программу
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    #8 - Выполняем действия "в рамках фрейма"
    ❷ if (ballRect.left < 0) or (ballRect.right >= WINDOW_WIDTH):
        xSpeed = -xSpeed # обращаем направление X

    if (ballRect.top < 0) or (ballRect.bottom >= WINDOW_HEIGHT):
        ySpeed = -ySpeed # обращаем направление Y

    # обновляем местоположение мяча, используя скорость в двух
    # направлениях
    ballRect.left = ballRect.left + xSpeed
    ballRect.top = ballRect.top + ySpeed

    #9 - Очищаем окно, прежде чем рисовать его заново
    window.fill(BLACK)

    #10 - Рисуем все элементы окна
    ❸ window.blit(ballImage, ballRect)

    #11 - Обновляем окно
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND)

```

Листинг 5.7. Основанная на местоположении анимация с мячом, отскакивающим от границ окна, с использованием `rect`

Этот подход использования объекта `rect` не лучше и не хуже применения отдельных переменных. Итоговая программа работает точно так же, как и исходная версия. Важный урок здесь в том, как вы можете использовать атрибуты объекта `rect` и манипулировать ими.

После загрузки изображения мяча мы вызываем метод `get_rect()` ❶, чтобы получить ограничивающий прямоугольник изображения. Этот вызов возвращает объект `rect`, который мы сохраняем в переменную под названием `ballRect`. Мы используем `ballRect.width` и `ballRect.height`, чтобы получить прямой доступ к ширине и высоте изображения мяча. (В предыдущей версии мы использовали константу 100 для ширины и высоты.) Извлечение этих значений из загруженного изображения делает наш код гораздо более адаптивным, поскольку это означает, что мы можем использовать графику любого размера.

Код также задействует атрибуты прямоугольника вместо использования отдельных переменных для проверки выхода за края любой части прямоугольника мяча. Мы применим `ballRect.left` и `ballRect.right`, чтобы увидеть, выходит ли `ballRect` за правый или левый край ❷. Проводим ту же проверку с помощью `ballRect.top` и `ballRect.bottom`. Вместо обновления отдельных координат `x` и `y` мы обновляем значения `left` и `right` `ballRect`.

Другое едва различимое, но важное изменение заключается в выполнении вызова, чтобы нарисовать мяч ❸. Вторым аргументом в вызове `blit()` может быть либо кортеж `(x, y)`, либо `rect`. Код внутри `blit()` использует левую верхнюю позицию в `rect` в качестве координат `x` и `y`.

Воспроизводим звуки

Существует два типа звуков, которые вы, возможно, захотите воспроизвести в вашей программе: короткие звуковые эффекты и фоновая музыка.

Воспроизводим звуковые эффекты

Все звуковые эффекты обязаны находиться во внешних файлах и должны быть либо `.wav`, либо `.ogg` формата. Воспроизведение относительно короткого звукового эффекта состоит из двух

шагов: один раз загрузить звук из внешнего звукового файла; затем в подходящее время воспроизвести его.

Чтобы загрузить звуковой эффект в память, используйте строку, подобную этой:

```
<soundVariable> = pygame.mixer.Sound(<путь к звуковому файлу>)
```

Чтобы воспроизвести звуковой эффект, вам лишь нужно вызвать метод `play()`:

```
<soundVariable>.play()
```

Мы изменим листинг 5.7, чтобы добавить звуковой эффект «бум», который станет воспроизводиться каждый раз, когда мяч будет отскакивать от стороны окна. В папке проекта есть папка *Sounds* на том же уровне, что и основная программа. Сразу же после загрузки изображения мяча мы загружаем звуковой файл, добавляя следующий код:

```
#4 - Загружаем элементы: изображения, звуки и т. д.  
ballImage = pygame.image.load('images/ball.png')  
bounceSound = pygame.mixer.Sound('sounds/boing.wav')
```

Чтобы воспроизводить звуковой эффект «бум» каждый раз, когда меняем либо горизонтальное, либо вертикальное направление мяча, мы изменяем раздел #8, чтобы он выглядел следующим образом:

```
#8 - Выполняем действия "в рамках фрейма"  
if (ballRect.left < 0) or (ballRect.right >= WINDOW_WIDTH):  
    xSpeed = -xSpeed # обращаем направление X  
    bounceSound.play()  
  
if (ballRect.top < 0) or (ballRect.bottom >= WINDOW_HEIGHT):  
    ySpeed = -ySpeed # обращаем направление Y  
    bounceSound.play()
```

Когда выполняется условие, при котором надо воспроизводить звуковой эффект, вы добавляете метод звука `play()`. Существует еще множество вариантов управления звуковыми эффектами; подробности вы найдете в официальной документации по адресу <https://www.pygame.org/docs/ref/mixer.html>.

Воспроизведение фоновой музыки

Воспроизведение фоновой музыки включает две строки кода, использующие вызовы модуля `pygame.mixer.music`. Во-первых, вам необходимо загрузить звуковой файл в память:

```
pygame.mixer.music.load (<путь к звуковому файлу>)
```

<путь к звуковому файлу> — это строка пути, по которому можно найти звуковой файл. Вы можете использовать файлы *.mp3*, которые подходят лучше всего, а также файлы формата *.wav* или *.ogg*. Когда вы хотите начать воспроизведение музыки, вам необходимо осуществить этот вызов:

```
pygame.mixer.music.play (<число циклов> <начальная позиция>)
```

Чтобы неоднократно воспроизводить какую-то фоновую музыку, вы можете передать `-1` в <число циклов>, чтобы музыка проигрывалась непрерывно. <начальная позиция>, как правило, устанавливается в значение `0`, чтобы обозначить, что вы хотите проигрывать звук с самого начала.

Вы можете скачать измененную версию программы прыгающего мяча, которая должным образом загружает файлы звуковых эффектов и фоновой музыки и запускает воспроизведение фонового звука. Единственные изменения внесены в раздел #4, как показано здесь.

```
#4 - Загружаем элементы: изображения, звуки и т. д.  
ballImage = pygame.image.load('images/ball.png')  
bounceSound = pygame.mixer.Sound('sounds/boing.wav')  
pygame.mixer.music.load('sounds/background.mp3')  
pygame.mixer.music.play(-1, 0.0)
```

Pygame позволяет гораздо более сложно обрабатывать фоновые звуки. Вы можете найти полную документацию по адресу <https://www.pygame.org/docs/ref/music.html#module-pygame.mixer.music>.

ПРИМЕЧАНИЕ Чтобы будущие примеры были более четко сосредоточены на ООП, я опушу вызовы для воспроизведения звуковых эффектов и фоновой музыки. Но добавление звуков значительно улучшает пользовательский игровой опыт, и я настоятельно рекомендую включать их.

Рисуем фигуры

Pygame предлагает некоторое количество встроенных функций, которые позволяют вам рисовать конкретные фигуры, известные как *примитивы*, которые включают линии, круги, овалы, дуги, многоугольники и прямоугольники. В табл. 5.2 представлен список этих функций. Обратите внимание, что существует два вызова, которые рисуют *сглаженные* линии, то есть те, которые содержат смешанные цвета по краям, чтобы они выглядели плавными и менее неровными. Есть два ключевых преимущества при использовании этих функций рисования: они выполняются невероятно быстро и позволяют вам рисовать простые фигуры без необходимости создания или загрузки изображений из внешних файлов.

Таблица 5.2. Функции для рисования фигур

Функция	Описание
<code>pygame.draw.aaline()</code>	Рисует сглаженные линии
<code>pygame.draw.aalines()</code>	Рисует серию сглаженных линий
<code>pygame.draw.arc()</code>	Рисует дугу
<code>pygame.draw.circle()</code>	Рисует круг
<code>pygame.draw.ellipse()</code>	Рисует овал
<code>pygame.draw.line()</code>	Рисует линию
<code>pygame.draw.lines()</code>	Рисует серию линий
<code>pygame.draw.polygon()</code>	Рисует многоугольник
<code>pygame.draw.rect()</code>	Рисует прямоугольник

На рис. 5.7 показаны выходные данные примера программы, который демонстрирует вызовы этих функций графических примитивов.

В листинге 5.8 показан код примера программы с использованием 12-шагового шаблона, который привел к выходным данным на рис. 5.7.

Файл: PygameDemo5_DrawingShapes.py

```
# pygame демо 5 - рисунок

--- пропуск ---
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():

        # Нажата кнопка закрытия? Выходим из pygame и завершаем
        # программу
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    #8 - Выполняем действия "в рамках фрейма"

    #9 - Очищаем окно
    window.fill(GRAY)

    ❶ #10 - Рисуем все элементы окна
    # рисуем прямоугольник
    pygame.draw.line(window, BLUE, (20, 20), (60, 20), 4) # верх
    pygame.draw.line(window, BLUE, (20, 20), (20, 60), 4) # лево
    pygame.draw.line(window, BLUE, (20, 60), (60, 60), 4) # право
    pygame.draw.line(window, BLUE, (60, 20), (60, 60), 4) # низ

    # рисуем X в прямоугольнике
    pygame.draw.line(window, BLUE, (20, 20), (60, 60), 1)
    pygame.draw.line(window, BLUE, (20, 60), (60, 20), 1)

    # рисуем закрашенный круг и пустой круг
    pygame.draw.circle(window, GREEN, (250, 50), 30, 0) # закрашенный
    pygame.draw.circle(window, GREEN, (400, 50), 30, 2) # контур
                                     # в 2 пикселя

    # рисуем закрашенный прямоугольник и пустой прямоугольник
    pygame.draw.rect(window, RED, (250, 150, 100, 50), 0) # закрашенный
    pygame.draw.rect(window, RED, (400, 150, 100, 50), 1) # контур
                                     # в 1 пиксель

    # рисуем закрашенный овал и пустой овал
    pygame.draw.ellipse(window, YELLOW, (250, 250, 80, 40), 0)
                                     # закрашенный
    pygame.draw.ellipse(window, YELLOW, (400, 250, 80, 40), 2)
                                     # контур в 2 пикселя
```

```

# рисуем шестисторонний многоугольник
pygame.draw.polygon(window, TEAL, ((240, 350), (350, 350),
                                     (410, 410), (350, 470),
                                     (240, 470), (170, 410)))

# рисуем дугу
pygame.draw.arc(window, BLUE, (20, 400, 100, 100), 0, 2, 5)

# рисуем сглаженные линии: одну линию, затем список точек
pygame.draw.aaline(window, RED, (500, 400), (540, 470), 1)
pygame.draw.aalines(window, BLUE, True,
                    ((580, 400), (587, 450),
                     (595, 460), (600, 444)), 1)

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 5.8. Программа для демонстрации вызовов функций графических примитивов в pygame

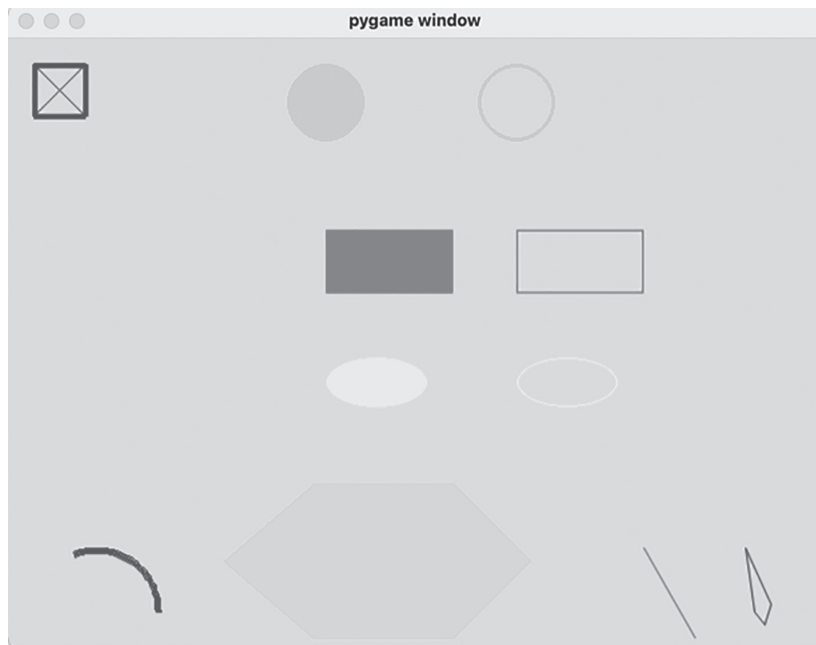


Рис. 5.7. Пример программы, который демонстрирует использование вызовов для графических примитивных фигур

Рисунок всех примитивов приведен в разделе #10 ❶. Мы вызываем функции pygame, чтобы нарисовать прямоугольник

с двумя диагоналями, закрашенные и пустые круги, закрашенные и пустые прямоугольники, закрашенные и пустые овалы, шестисторонний многоугольник, дугу и две сглаженные линии.

Справка по примитивным фигурам

Вам для справки представлена документация по методам `pygame` для изображения этих примитивов. Во всех следующих случаях аргумент `color` ожидает, что вы ему передадите кортеж значений RGB.

Сглаженная линия

```
pygame.draw.aaline(window, color, startpos, endpos, blend=True)
```

Рисует в окне сглаженную линию. Если значение `blend` равно `True`, оттенки будут смешаны с существующими цветами пикселей вместо того, чтобы переписывать пиксели.

Сглаженные линии

```
pygame.draw.aalines(window, color, closed, points, blend=True)
```

Рисует в окне последовательность сглаженных линий. Аргумент `closed` — это простое булево выражение; если он принимает значение `True`, между первой и последней точками будет нарисована линия, чтобы замкнуть фигуру. Аргумент `points` — это список кортежей координат (x, y) , которые должны быть соединены отрезками линий (должно быть как минимум две точки). Если значение булева аргумента `blend` равно `True`, то он станет смешивать оттенки с существующими оттенками пикселей вместо того, чтобы переписывать их.

Дуга

```
pygame.draw.arc(window, color, rect, angle_start, angle_stop, width=0)
```

Рисует в окне дугу. Дуга будет помещена внутри заданного `rect`. Два аргумента `angle` — это исходный и окончательный углы (в радианах, с нулем справа). Аргумент `width` — это толщина для рисования контура.

Круг

```
pygame.draw.circle(window, color, pos, radius, width=0)
```

Рисует в окне круг. `pos` — это центр круга, а `radius` — это радиус. Аргумент `width` — это толщина для рисования контура. Если `width` равен 0, тогда круг будет закрашен.

Овал

```
pygame.draw.ellipse(window, color, rect, width=0)
```

Рисует в окне овал. Заданный `rect` — это область, которую будет заполнять овал. Аргумент `width` — это толщина для рисования контура. Если `width` равен 0, тогда овал будет закрашен.

Линия

```
pygame.draw.line(window, color, startpos, endpos, width=1)
```

Рисует в окне линию. Аргумент `width` — это толщина линии.

Линии

```
pygame.draw.lines(window, color, closed, points, width=1)
```

Рисует в окне последовательность линий. Аргумент `closed` — это простое булево выражение; если он принимает значение `True`, между первой и последней точками нарисованная линия, чтобы замкнуть фигуру. Аргумент `points` — это список кортежей координат (x , y), которые должны быть соединены отрезками линий (как минимум двух). Аргумент `width` — это толщина линии. Обратите внимание, что указание ширины линии больше 1 не заполняет пробелы между линиями. Таким образом, широкие линии и острые углы не смогут плавно соединяться.

Многоугольник

```
pygame.draw.polygon(window, color, pointslist, width=0)
```

Рисует в окне многоугольник. `pointslist` указывает вершины многоугольника. Аргумент `width` — это толщина для

рисования контура. Если `width` равен 0, тогда многоугольник будет закрашен.

Прямоугольник

```
pygame.draw.rect(window, color, rect, width=0)
```

Рисует в окне прямоугольник. `rect` — это область прямоугольника. Аргумент `width` — это толщина для рисования контура. Если `width` равен 0, тогда прямоугольник будет закрашен.

ПРИМЕЧАНИЕ Чтобы получить дополнительную информацию, пройдите по ссылке <http://www.pygame.org/docs/ref/draw.html>.

Набор вызовов примитивов позволяет гибко рисовать любые фигуры, которые вы пожелаете. И вновь напоминаю: порядок осуществляемых вами вызовов важен. Думайте о порядке вызовов как о слоях; нарисованные раньше элементы могут быть перекрыты более поздними вызовами любой другой функции графического примитива.

Выводы

В этой главе я познакомил вас с основами `pygame`. Вы установили `pygame` на ваш компьютер, затем узнали о модели управляемого событиями программирования и об использовании этих событий, что сильно отличается от кодирования текстовых программ. Я объяснил систему координат пикселей в окне и способ представления цветов в коде.

Чтобы начать с `pygame` с самого начала, я представил 12-шаговый шаблон, который лишь выводит окно и может быть использован для создания любой основанной на `pygame` программы. Используя эту структуру, мы затем построили примеры программ, которые показали, как рисовать изображение в окне (с помощью `blit()`), обнаруживать события мыши и обрабатывать входные данные клавиатуры. Следующая демонстрация объяснила, как создать основанную на местоположении анимацию.

Прямоугольники невероятно важны в `pygame`, поэтому я рассказал о том, как можно использовать атрибуты объекта `rect`. Я также предоставил некоторые примеры кода, чтобы

показать, как воспроизводить звуковые эффекты и фоновую музыку для увеличения удовольствия пользователя от взаимодействия с программой. И наконец, я показал, как использовать методы `pygame` для изображения примитивных фигур в окне.

Несмотря на то что я представил много понятий в `pygame`, практически все, что я показал в этой главе, было, по сути, процедурным. Объект `rect` является примером объектно-ориентированного кода, встроенного непосредственно в `pygame`. В следующей главе я покажу, как использовать ООП в коде, чтобы применять `pygame` более эффективно.

6

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ PYGAME



В этой главе я продемонстрирую, как вы можете эффективно использовать методы ООП с фреймворком `pygame`. Мы начнем с примера процедурного кода, затем разобьем его на отдельный класс и основной код, который вызывает методы этого класса. После чего мы создадим два класса `SimpleButton` и `SimpleText`, которые реализуют базовые виджеты интерфейса пользователя: кнопку и поле для отображения текста. Я также познакомлю вас с понятием обратного вызова.

Создаем заставку мяча с помощью Pygame

В главе 5 мы создали старомодную заставку, в которой мяч прыгал внутри окна (листинг 5.6, если вам необходимо освежить память).

Код работает, но данные для мяча и код для управления мячом переплетены, то есть существует много кода инициализации, а код для обновления и рисования мяча вложен в 12-шаговую структуру.

Более модульный подход заключается в разбитии кода на класс `Ball` и основную программу, которая создает экземпляр объекта `Ball` и осуществляет вызовы его методов. Здесь мы разделим это, и я покажу вам, как создавать несколько мячей из класса `Ball`.

Создаем класс `Ball`

Начнем с извлечения всего кода, относящегося к мячу, из основной программы и перемещения его в отдельный класс `Ball`. Просматривая исходный код, мы можем увидеть, что разделы, относящиеся к мячу, включают:

- раздел #4, который загружает изображение мяча;
- раздел #5, который создает и инициализирует все переменные, имеющие какое-либо отношение к мячу;
- раздел #8, который включает код для перемещения мяча, обнаруживая края для отскока и изменяя скорость и направление;
- раздел #10, который рисует мяч.

Из этого мы можем сделать вывод, что нашему классу `Ball` потребуются следующие методы:

- **`create()`** — загружает изображение, устанавливает местоположение и инициализирует все переменные экземпляра;
- **`update()`** — изменяет местоположение мяча в каждом фрейме на основании скорости x и y мяча;
- **`draw()`** — рисует мяч в окне.

Первый шаг состоит в создании папки проекта, в которой вам понадобятся файл *Ball.py* для класса `Ball`, файл основного кода *Main_BallBounce.py* и папка *images*, содержащая файл изображения *ball.png*.

В листинге 6.1 показан код класса `Ball`.

Файл: `PygameDemo6_BallBounceObjectOriented/Ball.py`

```
import pygame
from pygame.locals import *
import random

# класс Ball
class Ball():
```

```

❶ def __init__(self, window, windowWidth, windowHeight):
    self.window = window # запоминаем окно, чтобы мы смогли
                          # нарисовать позднее
    self.windowWidth = windowWidth
    self.windowHeight = windowHeight
❷ self.image = pygame.image.load('images/ball.png')

    # прямоугольник состоит из [x, y, ширина, высота]
    ballRect = self.image.get_rect()
    self.width = ballRect.width
    self.height = ballRect.height
    self.maxWidth = windowWidth - self.width
    self.maxHeight = windowHeight - self.height

    # выбираем произвольную начальную позицию
❸ self.x = random.randrange(0, self.maxWidth)
    self.y = random.randrange(0, self.maxHeight)

    # выбираем произвольную скорость между -4 и 4, но не ноль
    # в обоих направлениях x и y
❹ speedsList = [-4, -3, -2, -1, 1, 2, 3, 4]
    self.xSpeed = random.choice(speedsList)
    self.ySpeed = random.choice(speedsList)

❺ def update(self):
    # проверяем наличие ударов о стену. Если они есть, изменяем
    # направление
    if (self.x < 0) or (self.x >= self.maxWidth):
        self.xSpeed = -self.xSpeed

    if (self.y < 0) or (self.y >= self.maxHeight):
        self.ySpeed = -self.ySpeed

    # обновляем x и y Ball, используя скорость в двух направлениях
    self.x = self.x + self.xSpeed
    self.y = self.y + self.ySpeed

❻ def draw(self):
    self.window.blit(self.image, (self.x, self.y))

```

Листинг 6.1. Новый класс Ball

Когда мы создаем экземпляр объекта Ball, метод `__init__()` получает три части данных: окно, в котором рисовать, ширину окна и высоту окна ❶. Мы сохраняем переменную `window` в переменную экземпляра `self.window`, чтобы можно было ее

использовать в дальнейшем в методе `draw()`, и делаем то же самое с переменными экземпляра `self.windowHeight` и `self.windowWidth`. Затем загружаем изображение мяча, используя путь к файлу, и получаем `rect` изображения мяча ❷. Нам необходимо, чтобы `rect` пересчитал максимальные значения для `x` и `y`, чтобы мяч всегда полностью отображался в окне. Далее мы выбираем случайное начальное местоположение для мяча ❸. И наконец, устанавливаем скорость направлений `x` и `y` в произвольное значение от -4 до 4 (но не 0), представляющее количество пикселей для перемещения в каждом фрейме ❹. Из-за всех этих чисел мяч может перемещаться по-разному каждый раз, когда мы запускаем программу. Эти значения сохраняются в переменных экземпляра, чтобы их могли использовать другие методы.

В основной программе мы вызываем метод `update()` в каждом фрейме основного цикла и именно сюда помещаем код, который проверяет, ударился ли мяч о границу окна ❺. Если он это делает, мы меняем скорость в этом направлении и изменяем координаты `x` и `y` (`self.x` и `self.y`) на текущую скорость в направлениях `x` и `y`.

Также вызываем метод `draw()`, который просто вызывает `blit()`, чтобы нарисовать мяч в его текущих координатах `x` и `y` ❻ в каждом фрейме основного цикла.

Используем класс `Ball`

Теперь вся связанная с мячом функциональность была помещена в код класса `Ball`. Все, что необходимо сделать основной программе, — это создать мяч, затем в каждом фрейме вызвать методы `update()` и `draw()`. В листинге 6.2 показан значительно упрощенный основной программный код.

Файл: `PygameDemo6_BallBounceObjectOriented/Main_BallBounce.py`

```
# pygame демо 6(a) - используя класс Ball, отбивать один мяч

#1 - Импортируем пакеты
import pygame
from pygame.locals import *
import sys
import random
❶ from Ball import * # вводим код класса Ball
```

```

#2 - Определяем константы
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.

#5 - Инициализируем переменные
❷ oBall = Ball(window, WINDOW_WIDTH, WINDOW_HEIGHT)

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    #8 - Выполняем действия "в рамках фрейма"
    ❸ oBall.update() # обновляем Ball

    #9 - Очищаем окно, прежде чем рисовать его заново
    window.fill(BLACK)

    #10 - Рисуем все элементы окна
    ❹ oBall.draw() # рисуем Ball

    #11 - Обновляем окно
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND)

```

Листинг 6.2. Новая основная программа, которая создает экземпляр Ball и вызывает его методы

Если вы сравните эту новую основную программу с исходным кодом в листинге 5.6, то увидите, что она значительно проще и яснее. Мы используем оператор `import`, чтобы ввести код класса `Ball` ❶. Создаем объект `Ball`, передавая в него уже

готовое окно, а также его ширину и высоту ❷, сохраняем итоговый объект Ball в переменную под названием oBall.

Ответственность за передвижение мяча теперь лежит на коде класса Ball, поэтому здесь нам только необходимо вызвать метод update() объекта oBall ❸. Поскольку объект Ball знает величину окна, величину изображения мяча плюс его местоположение и скорость, он может выполнять все необходимые вычисления, чтобы перемещать мяч и отбивать его от стен.

Основной код вызывает метод draw() объекта oBall ❹, а фактически рисунок выполняется объектом oBall.

Создаем много объектов Ball

Теперь давайте внесем мелкое, но важное изменение в основную программу, чтобы создать несколько объектов Ball. Это одно из реальных преимуществ ООП: чтобы создать три мяча, нам необходимо лишь создать экземпляры трех объектов Ball из класса Ball. Здесь воспользуемся базовым подходом и создадим список объектов Ball. В каждом фрейме мы будем проводить итерацию списка объектов Ball, говорить каждому из них обновлять его местоположение, затем снова проводить итерацию, чтобы сказать каждому из них нарисовать себя.

В листинге 6.3 продемонстрирована измененная основная программа, которая создает и обновляет три объекта Ball.

Файл: PygameDemo6_BallBounceObjectOriented/ Main_BallBounceManyBalls.py

```
# pygame демо 6(b) - используя класс Ball, отбивать много мячей

--- пропуск ---
N_BALLS = 3
--- пропуск ---

#5 - Инициализируем переменные
❶ ballList = []
for oBall in range(0, N_BALLS):
    # Каждый раз, проходя цикл, создаем объект Ball
    oBall = Ball(window, WINDOW_WIDTH, WINDOW_HEIGHT)
    ballList.append(oBall) # добавляем новый Ball в список мячей

#6 - Бесконечный цикл
while True:
```

```

--- пропуск ---

#8 - Выполняем действия "в рамках фрейма"
❷ for oBall in ballList:
    oBall.update() # обновляем Ball

#9 - Очищаем окно, прежде чем рисовать его заново
window.fill(BLACK)

#10 - Рисуем все элементы окна
❸ for oBall in ballList:
    oBall.draw() # рисуем Ball

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND)

```

Листинг 6.3. Создание, перемещение и отображение трех мячей

Мы начинаем с пустого списка объектов Ball ❶. Затем наш цикл создает три объекта Ball, каждый из которых добавляем в список объектов Ball, ballList. Каждый объект Ball выбирает и запоминает случайное начальное местоположение и случайную скорость как для направления x , так и для направления y .

Внутри основного цикла мы проводим итерацию всех объектов Ball и обновляем их ❷, изменяя координаты x и y каждого объекта Ball на новое местоположение. Затем мы снова проводим итерацию списка, вызывая метод draw() для каждого объекта Ball ❸.

Когда запускаем программу, мы видим три мяча, каждый из которых стартует из произвольного местоположения и перемещается с произвольной скоростью x и y . Каждый мяч безошибочно отскакивает от границ окна.

Используя объектно-ориентированный подход, мы не вносим никакие изменения в класс Ball, просто меняем основную программу, чтобы она управляла списком объектов Ball вместо одного объекта Ball. Это распространенный и очень положительный побочный эффект кода ООП: хорошо написанные классы можно часто повторно использовать без изменений.

Создаем много, много объектов Ball

Мы можем изменить значение константы `N_BALLS` с 3 на некоторое гораздо большее число, например 300, чтобы быстро создать множество мячей (рис. 6.1). Изменяя лишь одну константу, мы вносим огромные изменения в поведение программы. Каждый мяч поддерживает собственную скорость и местоположение и рисует сам себя.

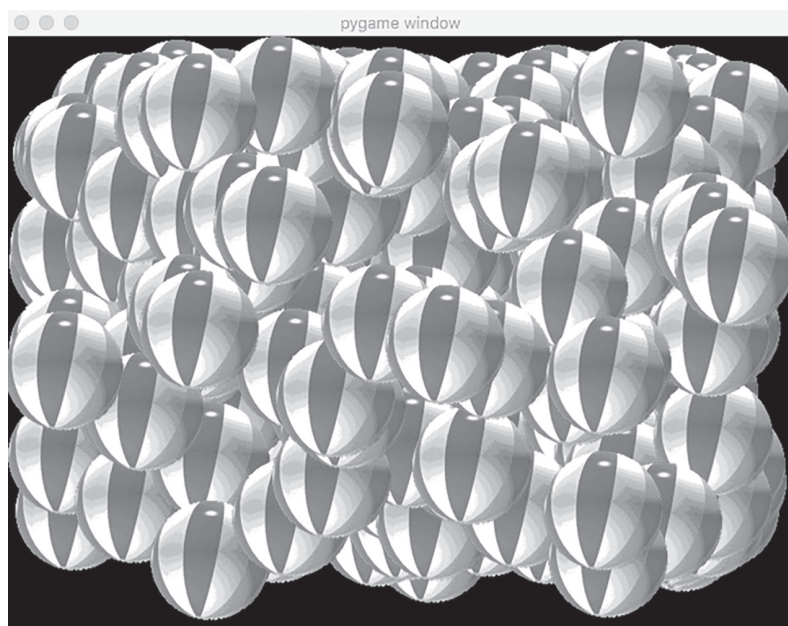


Рис. 6.1. Создание, перемещение и отображение 300 объектов `Ball`

Тот факт, что мы можем создавать экземпляры любого количества объектов из одного скрипта, будет важен не только для определения объектов игры, таких как космические корабли, зомби, пути, сокровища и так далее, но также при создании элементов управления GUI: кнопок, флажков, полей ввода текста и текстовых выходных данных.

Создаем многократно используемую объектно-ориентированную кнопку

Простая кнопка — один из наиболее узнаваемых элементов графического интерфейса пользователя. Ее стандартное поведение — реагировать на щелчок мыши. Пользователь нажимает на картинку кнопки, а затем отпускает.

Кнопки обычно состоят из как минимум двух изображений: одно для представления состояния *вверх* (или нормального состояния кнопки) и другое для представления состояния *вниз* (или нажатого состояния кнопки). Последовательность щелчков можно разбить на следующие шаги.

1. Пользователь перемещает указатель мыши на кнопку.
2. Пользователь нажимает кнопку мыши.
3. Программа реагирует, изменяя изображение в состояние вниз.
4. Пользователь отпускает кнопку мыши.
5. Программа реагирует, отображая изображение кнопки вверх.
6. Программа осуществляет некоторое действие на основании нажатия кнопки.

Хорошие GUI позволяют пользователю нажимать на кнопку, временно уйти с нее, изменяя рисунок на состояние вверх, а затем при все еще зажатой кнопке мыши вернуться к изображению, так что кнопка изменяет свой вид обратно на изображение вниз. Если пользователь щелкает по кнопке, но затем уводит мышь в сторону и отпускает кнопку мыши, это не считается щелчком. Это означает, что программа принимает действие, только когда пользователь нажимает и отпускает, в то время как курсор помещен на изображение кнопки.

Создаем класс кнопки

Поведение кнопки должно быть общим и последовательным для всех кнопок, используемых в GUI, поэтому мы создадим класс, который позаботится об особенностях поведения. Создав простой класс кнопки, мы можем создавать экземпляры любого количества кнопок, и они будут работать точно таким же образом.

Давайте рассмотрим, какие действия классов кнопок необходимо поддерживать. Нам нужны методы, чтобы:

- 1) загружать изображения состояний «вверх» и «вниз», затем инициализировать любые переменные экземпляра, чтобы отслеживать состояние кнопки;
- 2) сообщать кнопке о событиях, которые обнаружила основная программа, и проверять, есть ли какие-либо кнопки, на которые необходимо отреагировать;
- 3) рисовать текущее изображение, представляющее кнопку.

В листинге 6.4 представлен код класса SimpleButton. (Мы создадим более сложный класс кнопки в главе 7.) У этого класса есть три метода: `__init__()`, `handleEvent()` и `draw()`, которые реализуют упомянутые действия. Код метода `handleEvent()` действительно немного сложнее, но, как только вы заставите его работать, он окажется невероятно прост в использовании. Не стесняйтесь работать над ним, но имейте в виду, что реализация кода не так важна. Важным моментом здесь является понимание и использование различных методов.

Файл: PygameDemo7_SimpleButton/SimpleButton.py

```
# Класс SimpleButton
#
# Используем подход "конечного автомата"
#

import pygame
from pygame.locals import *

class SimpleButton():
    # Используется для отслеживания состояния кнопки
    STATE_IDLE = 'idle' # кнопка вверх, мышь не на кнопке
    STATE_ARMED = 'armed' # кнопка вниз, мышь на кнопке
    STATE_DISARMED = 'disarmed' # щелчок по кнопке, откат

    def __init__(self, window, loc, up, down): ❶
        self.window = window
        self.loc = loc
        self.surfaceUp = pygame.image.load(up)
        self.surfaceDown = pygame.image.load(down)

        # получаем rect кнопки (используется, чтобы увидеть,
        # находится ли мышь на кнопке)
        self.rect = self.surfaceUp.get_rect()
        self.rect[0] = loc[0]
        self.rect[1] = loc[1]

        self.state = SimpleButton.STATE_IDLE

    def handleEvent(self, eventObj): ❷
        # Это метод вернет значение True, если пользователь щелкнет
        # по кнопке.
        # Обычно возвращает False

        if eventObj.type not in (MOUSEMOTION, MOUSEBUTTONUP,
                                MOUSEBUTTONDOWN): ❸
```

```

        # Кнопка реагирует только на относящиеся к мыши события
        return False

    eventPointInButtonRect = self.rect.collidepoint(eventObj.pos)

    if self.state == SimpleButton.STATE_IDLE:
        if (eventObj.type == MOUSEBUTTONDOWN) and
                                eventPointInButtonRect:
            self.state = SimpleButton.STATE_ARMED

    elif self.state == SimpleButton.STATE_ARMED:
        if (eventObj.type == MOUSEBUTTONUP) and
                                eventPointInButtonRect:
            self.state = SimpleButton.STATE_IDLE
            return True # был щелчок!

        if (eventObj.type == MOUSEMOTION) and
                                (not eventPointInButtonRect):
            self.state = SimpleButton.STATE_DISARMED

    elif self.state == SimpleButton.STATE_DISARMED:
        if eventPointInButtonRect:
            self.state = SimpleButton.STATE_ARMED
        elif eventObj.type == MOUSEBUTTONUP:
            self.state = SimpleButton.STATE_IDLE

    return False

def draw(self):
    # рисуем текущий вид кнопки в окне
    if self.state == SimpleButton.STATE_ARMED:
        self.window.blit(self.surfaceDown, self.loc)

    else: # IDLE или DISARMED
        self.window.blit(self.surfaceUp, self.loc)

```

Листинг 6.4. Класс SimpleButton

Метод `__init__()` начинается с сохранения всех значений, переданных переменным экземпляра ❶, чтобы использовать другие методы. Затем он инициализирует еще несколько переменных экземпляра.

Каждый раз, когда программа обнаруживает какое-либо событие, она вызывает метод `handleEvent()` ❷. Сначала он проверяет, является ли событие одним из: `MOUSEMOTION`, `MOUSEBUTTONUP` или `MOUSEBUTTONDOWN` ❸.

Остальная часть реализуется в качестве *конечного автомата*, метода, который я рассмотрю более подробно в главе 15. Код немного сложен, и не стесняйтесь изучить, как он работает, но на данный момент обратите внимание, что он использует переменные экземпляра `self.state` (в течение нескольких вызовов), чтобы обнаружить, щелкнул ли пользователь по кнопке. Метод `handleEvent()` возвращает значение `True`, когда пользователь завершает щелчок мыши, нажимая кнопку, а затем позднее отпуская ее на той же кнопке. Во всех остальных случаях `handleEvent()` возвращает `False`.

И наконец, метод `draw()` использует состояние переменной экземпляра объекта `self.state`, чтобы решить, какое изображение (вверх или вниз) рисовать ❹.

Основной код, использующий SimpleButton

Чтобы использовать `SimpleButton` в основном коде, мы сначала создаем экземпляр из класса `SimpleButton` перед началом основного цикла с помощью строки, подобной этой:

```
oButton = SimpleButton(window, (150, 30),  
                        'images/buttonUp.png',  
                        'images/buttonDown.png')
```

Она создает объект `SimpleButton`, указывая местоположение для его отображения (как обычно, координаты представлены для верхнего левого угла ограничивающего прямоугольника) и предоставляя пути к изображениям кнопки вверх и вниз. В основном цикле каждый раз, когда происходит событие, нам необходимо вызывать метод `handleEvent()`, чтобы увидеть, щелкнул ли пользователь по кнопке. Если пользователь щелкает по кнопке, программа должна выполнить какое-то действие. Также в основном цикле необходимо вызвать метод `draw()`, чтобы отобразить кнопку в окне.

Мы создадим небольшую тестовую программу, которая будет генерировать интерфейс пользователя, как на рис. 6.2, чтобы включить один экземпляр в `SimpleButton`.

Каждый раз, когда пользователь завершает щелчок по кнопке, программа выводит строку текста в оболочке, где написано, что кнопка была нажата. В листинге 6.5 содержится основной программный код.



Рис. 6.2. Интерфейс пользователя программы с одним экземпляром SimpleButton

Файл: PygameDemo7_SimpleButton/Main_SimpleButton.py

```
# Pygame демо 7 - тест SimpleButton

--- пропуск ---

#5 - Инициализируем переменные
# создаем экземпляр SimpleButton
❶ oButton = SimpleButton(window, (150, 30),
                        'images/buttonUp.png',
                        'images/buttonDown.png')

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # передаем событие кнопке, смотрим, была ли она нажата
    ❷ if oButton.handleEvent(event):
    ❸     print('User has clicked the button')

    #8 - Выполняем действия "в рамках фрейма"

    #9 - Очищаем окно
    window.fill(GRAY)

    #10 - Рисуем все элементы окна
    ❹ oButton.draw() # рисуем кнопку

    #11 - Обновляем окно
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND)
```

Листинг 6.5. Основная программа, которая создает SimpleButton и реагирует на него

И вновь мы начинаем со стандартного шаблона `pygame` из главы 5. Перед началом основного цикла мы создаем экземпляр `SimpleButton` ❶, указывая окно, в котором нужно рисовать, местоположение, путь к изображению вверх и путь к изображению вниз.

Каждый раз, проходя основной цикл, нам необходимо реагировать на обнаруженные в основной программе события. Чтобы реализовать это, мы вызываем метод `handleEvent()` класса `SimpleButton` ❷ и передаем `event` из основной программы.

Метод `handleEvent()` отслеживает все действия пользователя с кнопкой (нажатие, отпускание, откат, возврат). Когда `handleEvent()` принимает значение `True`, указывая, что произошел щелчок, мы осуществляем действие, связанное с нажатием кнопки. Здесь мы просто выводим сообщение ❸.

И наконец, вызываем метод кнопки `draw()` ❹, чтобы нарисовать изображение, представляющее подходящее состояние кнопки (вверх или вниз).

Создаем программу с несколькими кнопками

С помощью класса `SimpleButton` мы способны создать экземпляры необходимого нам количества кнопок. Например, можем изменить нашу основную программу, чтобы она включала три экземпляра `SimpleButton`, как показано на рис. 6.3.



Рис. 6.3. Основная программа с тремя объектами `SimpleButton`

Чтобы сделать это, нам не нужно вносить какие-либо изменения в файл класса `SimpleButton`. Мы просто изменяем основной код, чтобы создать экземпляры трех объектов `SimpleButton` вместо одного.

```
oButtonA = SimpleButton(window, (25, 30),
                        'images/buttonAUp.png',
                        'images/buttonADown.png')
oButtonB = SimpleButton(window, (150, 30),
                        'images/buttonBUp.png',
                        'images/buttonBDown.png')
oButtonC = SimpleButton(window, (275, 30),
```

```
'images/buttonCUp.png',  
'images/buttonCDown.png')
```

Теперь нам нужно вызвать метод `handleEvent()` для всех трех кнопок:

```
# передаем событие каждой кнопке, смотрим, была ли  
# она нажата  
if oButtonA.handleEvent(event):  
    print('User clicked button A.')
```

```
elif oButtonB.handleEvent(event):  
    print('User clicked button B.')
```

```
elif oButtonC.handleEvent(event):  
    print('User clicked button C.')
```

И наконец, рисуем кнопки:

```
oButtonA.draw()  
oButtonB.draw()  
oButtonC.draw()
```

Когда запустите программу, вы увидите окно с тремя кнопками. Щелчок по любой из них выводит сообщение, содержащее имя кнопки, которая была нажата.

Ключевая идея здесь заключается в том, что, поскольку мы используем три экземпляра одного и того же класса `SimpleButton`, поведение всех кнопок будет идентичным. Важное преимущество этого подхода — любое изменение кода в классе `SimpleButton` повлияет на все кнопки класса, для которых были созданы экземпляры. Основной программе не стоит беспокоиться ни о каких деталях внутренней работы кода кнопки, ей просто необходимо вызывать метод `handleEvent()` для каждой кнопки основного цикла. Каждая кнопка будет возвращать значение `True` или `False`, обозначая, была ли нажата кнопка.

Создаем многократно используемое отображение текста

В программе `pygame` есть два различных типа текста: отображаемый и вводимый. Отображаемый текст — это выходные данные вашей программы, эквивалентные вызову функции `print()`, за исключением того, что они отображаются в окне

pygame. Вводимый текст — это строка входных данных от пользователя, эквивалентная вызову `input()`. В этом разделе я буду рассматривать отображаемый текст. А в следующей главе мы посмотрим, как работать с вводом текста.

Шаги для отображения текста

Отображение текста в окне представляет собой достаточно сложный процесс в pygame, потому что надо не просто показать его в виде строки в оболочке, но также выбрать местоположение, шрифты, размеры шрифтов и прочие атрибуты. Например, вы можете использовать подобный код:

```
pygame.font.init()

myFont = pygame.font.SysFont('Comic Sans MS', 30)
textSurface = myFont.render('Some text', True, (0, 0, 0))
window.blit(textSurface, (10, 10))
```

Мы начнем с инициализации системы шрифтов в pygame; сделаем это перед началом основного цикла. Затем скажем pygame загрузить конкретный шрифт из системы, указав его имя. Здесь мы запросим шрифт Comic Sans размером 30 пунктов.

Следующий шаг — ключевой: мы используем этот шрифт, чтобы *визуализировать* наш текст, что создаст графическое изображение текста, в pygame называемое *поверхностью*. Мы предоставим текст, который хотим вывести, булево выражение, сообщающее, хотим ли мы сгладить его, и цвет в формате RGB. Здесь (0, 0, 0) указывает, что мы хотим, чтобы текст был черным. И наконец, используя `blit()`, мы нарисуем изображение текста в окне в некотором местоположении (x, y).

Этот код хорошо работает для отображения в окне предоставленного текста в заданном местоположении. Однако, если текст не меняется, вы потратите много времени впустую, вновь создавая `textSurface` при каждой итерации основного цикла. Также необходимо помнить о множестве деталей, и вы должны все их выполнить правильно, чтобы нарисовать текст должным образом. Большую часть этих сложностей мы можем скрыть, создав класс.

Создаем класс SimpleText

Идея заключается в том, чтобы создать набор методов, который позаботится о загрузке шрифта и отображении текста в pygame, что обозначает, что нам больше не нужно запоминать детали реализации. В листинге 6.6 содержится новый класс под названием SimpleText, который выполняет эту работу.

Файл: PygameDemo8_SimpleTextDisplay/SimpleText.py

```
# Класс SimpleText

import pygame
from pygame.locals import *

class SimpleText():

    ❶ def __init__(self, window, loc, value, textColor):
    ❷     pygame.font.init()
        self.window = window
        self.loc = loc
    ❸     self.font = pygame.font.SysFont(None, 30)
        self.textColor = textColor
        self.text = None # так что вызов setText ниже
                        # приведет к созданию изображения текста
        self.setValue(value) # Настраиваем исходный текст для
                        # отображения

    ❹     def setValue(self, newText):
        if self.text == newText:
            return # ничего не менять

        self.text = newText # сохраняем новый текст
        self.textSurface = self.font.render(self.text, True,
                                            self.textColor)

    ❺ def draw(self):
        self.window.blit(self.textSurface, self.loc)
```

Листинг 6.6. Класс SimpleText для отображения текста

Вы можете воспринимать объект SimpleText как поле в окне, где хотите, чтобы отображался текст. Вы можете его использовать для отображения неизменного текста метки или для отображения текста, который меняется в процессе работы программы.

У класса `SimpleText` есть лишь три метода. Метод `__init__()` ❶ ожидает окно, где можно рисовать, место в нем, где должен быть текст, любой исходный текст, который вы хотите отобразить в поле, и цвет текста. Вызов `pygame.font.init()` ❷ запускает систему шрифтов `pygame`. Вызов в первом созданном экземпляре объекта `SimpleText` фактически выполняет инициализацию. Любые дополнительные объекты `SimpleText` также будут осуществлять этот вызов, но, поскольку шрифты уже были инициализированы, он завершится моментально. Мы создадим новый объект `Font` с помощью `pygame.font.SysFont()` ❸. Указав `None` вместо конкретного имени шрифта, будем использовать любой стандартный системный шрифт.

Метод `setValue()` визуализирует изображение текста, чтобы показать его, и сохраняет это изображение в переменной экземпляра `self.textSurface` ❹. Во время выполнения программы каждый раз, когда хотите изменить показываемый текст, вы вызываете метод `setValue()`, передавая новый текст для отображения. У метода `setValue()` есть также оптимизация: он запоминает последний визуализированный текст и, прежде чем сделать что-то еще, проверяет, является ли новый текст тем же самым, что и предыдущий. Если текст не изменился, ничего не надо делать, и метод просто возвращает его. Если есть новый текст, он визуализирует его на поверхность, чтобы нарисовать.

Метод `draw()` ❺ рисует в окне в заданном местоположении изображение, содержащее переменную экземпляра `self.textSurface`. Этот метод должен вызываться в каждом фрейме.

У данного подхода существует множество преимуществ.

- Класс скрывает все детали визуализации текста `pygame`, так что пользователю этого класса нет необходимости знать, какие специфичные для `pygame` вызовы необходимы для отображения текста.
- Каждый объект `SimpleText` запоминает окно, в котором он рисует, местоположение, куда текст должен быть помещен, и цвет текста. Следовательно, вам необходимо указать эти значения лишь один раз, когда вы создаете экземпляр `SimpleText`, обычно перед началом основного цикла.

- Каждый объект `SimpleText` также оптимизирован, чтобы запомнить как текст, который ему было сказано нарисовать в прошлый раз, так и изображение (`self.textSurface`), созданное им из текущего текста. Ему всего лишь необходимо визуализировать новую поверхность, когда текст изменяется.
- Чтобы отобразить несколько частей текста в окне, вам необходимо лишь создать экземпляры нескольких объектов `SimpleText`. Это ключевое понятие объектно-ориентированного программирования.

Демоверсия **Ball** с **SimpleText** и **SimpleButton**

Теперь мы изменим листинг 6.2, чтобы использовать классы `SimpleText` и `SimpleButton`. Обновленная программа в листинге 6.7 отслеживает количество прохождений основного цикла и сообщает эту информацию в верхней части окна. Щелчок по кнопке перезагрузки приводит к сбросу счетчика.

Файл: **PygameDemo8_SimpleTextDisplay/Main_BallTextAndButton.py**

```
# pygame демо 8 - SimpleText, SimpleButton и Ball

#1 - Импортируем пакеты
import pygame
from pygame.locals import *
import sys
import random

❶ from Ball import * # вводим код класса Ball
from SimpleText import *
from SimpleButton import *

#2 - Определяем константы
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.
```

```

#5 - Инициализируем переменные
❷ oBall = Ball(window, WINDOW_WIDTH, WINDOW_HEIGHT)
oFrameCountLabel = SimpleText(window, (60, 20),
                                'Program has run through this many loops: ', WHITE)
oFrameCountDisplay = SimpleText(window, (500, 20), '', WHITE)
oRestartButton = SimpleButton(window, (280, 60),
                                'images/restartUp.png', 'images/restartDown.png')

frameCounter = 0

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    ❸ if oRestartButton.handleEvent(event):
        frameCounter = 0 # кнопка нажата, сбрасываем счетчик

    #8 - Выполняем действия "в рамках фрейма"
    ❹ oBall.update() # обновляем ball
    frameCounter = frameCounter + 1 # увеличиваем каждый фрейм
    ❺ oFrameCountDisplay.setValue(str(frameCounter))

    #9 - Очищаем окно, прежде чем рисовать его заново
    window.fill(BLACK)

    #10 - Рисуем все элементы окна
    ❻ oBall.draw() # рисуем ball
    oFrameCountLabel.draw()
    oFrameCountDisplay.draw()
    oRestartButton.draw()

    #11 - Обновляем окно
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND)

```

Листинг 6.7. Пример основной программы, чтобы отобразить Ball, SimpleText и SimpleButton

В верхней части программы мы импортируем код классов Ball, SimpleText и SimpleButton ❶. Перед началом основного цикла создаем экземпляр Ball ❷, два экземпляра класса

`SimpleText` (`oFrameCountLabel` для неизменной метки сообщения и `oFrameCountDisplay` для изменяющихся отображений фреймов) и экземпляр класса `SimpleButton`, который храним в `oRestartButton`. Мы также инициализируем переменную `frameCounter` в значении ноль, которую будем увеличивать каждый раз во время прохождения основного цикла.

В основном цикле проверяем, нажал ли пользователь кнопку перезагрузки ❸. Если `True`, сбрасываем счетчик фреймов.

Обновляем позицию мяча ❹. Увеличиваем счетчик фреймов, затем вызываем метод `setValue()` текстового поля, чтобы отобразить новое число фреймов ❺. И наконец, рисуем мяч, вызывая метод `draw()` для каждого объекта ❻.

При создании экземпляров объектов `SimpleText` последний аргумент — цвет текста, и мы указали, что объекты должны визуализироваться в цвете `WHITE`, чтобы они были видны на фоне `BLACK`. В следующей главе я покажу, как расширить класс `SimpleText`, чтобы он включал больше атрибутов, не усложняя при этом интерфейс класса. Мы создадим более полнофункциональный текстовый объект с подходящим количеством значений по умолчанию для каждого из этих атрибутов, но при этом позволяющий переопределять их по умолчанию.

Сравнение интерфейса и реализации

Примеры `SimpleButton` и `SimpleText` поднимают важную тему сравнения интерфейса и реализации. Как было упомянуто в главе 4, интерфейс относится к тому, как что-то используется, в то время как реализация относится к тому, как что-то работает (внутренне).

В среде ООП интерфейс представляет собой набор методов в классе и связанных с ними параметров, также известный как *программный интерфейс приложения (API)*. Реализация — это фактический код всех методов класса.

Внешний пакет, такой как `pygame`, скорее всего будет полагаться с документацией API, объясняющей доступные вызовы и аргументы, которые вы должны передавать при каждом вызове. Полная документация `pygame` API доступна по адресу <https://www.pygame.org/docs/>.

Когда пишете код, который вызывает `pygame`, вам не нужно беспокоиться о реализации используемых вами методов.

Например, когда вызываете `blit()` для рисования изображения, вы на самом деле не в курсе, *как* `blit()` делает свою работу; вам лишь необходимо знать, *что* делает вызов и какие аргументы необходимо передать. С другой стороны, вы можете быть уверены, что конструктор, который написал метод `blit()`, тщательно продумал, как сделать работу `blit()` максимально эффективной.

В мире программирования мы часто примеряем на себя две роли — как конструктора, так и разработчика приложения, поэтому нам необходимо приложить усилия для разработки таких API, которые не только применимы в текущей ситуации, но также окажутся достаточно общими, чтобы их могли использовать наши будущие программы и программы, написанные другими людьми. Наши классы `SimpleButton` и `SimpleText` являются хорошими примерами, так как они написаны в таком общем виде, что их можно с легкостью многократно применять. Я расскажу подробнее о преимуществах интерфейса по сравнению с реализацией в главе 8, когда мы будем изучать инкапсуляцию.

Обратные вызовы

При использовании объекта `SimpleButton` мы управляем проверкой факта нажатия кнопки и реакцией на нее следующим образом:

```
if oButton.handleEvent(event):  
    print('The button was clicked')
```

Этот подход к обработке событий хорошо работает с классом `SimpleButton`. Однако некоторые другие пакеты Python и многие другие языки программирования обрабатывают события иным способом: с помощью *обратного вызова*.

Обратный вызов (callback)

Функция или метод объекта, который вызывается, когда происходит конкретное действие, событие или условие.

Простым способом понять это будет вспомнить популярное кино 1984 года «Охотники за привидениями». Слоган фильма — «Кому вы собираетесь звонить?». В кинофильме «Охотники

за привидениями» запускали рекламу на телевидении, которая сообщала людям, что если они увидели привидение (это событие, которое необходимо искать), то должны позвонить Охотникам за привидениями (обратный вызов), чтобы избавиться от него. После получения вызова Охотники за привидениями предпринимали соответствующие действия, чтобы избавиться от привидения.

В качестве примера рассмотрим объект кнопки, который инициализирован для обратного вызова. Когда пользователь щелкает по кнопке, она вызывает функцию или метод обратного вызова. Функция или метод исполняются каждый раз, когда код должен отреагировать на нажатие кнопки.

Создаем обратный вызов

Чтобы установить обратный вызов, когда создаете объект или вызываете один из методов объекта, вы передаете имя функции или метода объекта для вызова. В качестве примера возьмем стандартный пакет GUI для Python под названием `tkinter`. Код, необходимый для создания кнопки с помощью этого пакета, сильно отличается от того, что я показывал; ниже представлен пример:

```
import tkinter

def myFunction():
    print('myCallBackFunction was called')

oButton = tkinter.Button(text='Click me',
                           command=myFunction)
```

Когда создаете кнопку `tkinter`, вы должны передать функцию (или метод объекта), который будет вызван обратно, когда пользователь щелкнет по ней. Здесь мы передаем `myFunction` в качестве функции для обратного вызова. (Он использует параметры ключевых слов, которые мы более подробно обсудим в главе 7.) Кнопка `tkinter` запоминает функцию в качестве обратного вызова, и, когда пользователь щелкает по итоговой кнопке, он вызывает функцию `myFunction()`.

Вы также можете использовать обратный вызов, когда иницилируете действие, которое может занять некоторое время. Вместо того чтобы ждать окончания действия, что приведет

к заморозке программы на некий период времени, вы предоставляете обратный вызов, который будет вызван по завершении действия. Например, представьте, что вы хотите сделать запрос через сеть Интернет. Вместо того чтобы совершать вызов и ждать, когда он вернет данные, что может занять продолжительное время, лучше использовать пакеты, которые позволят вам осуществить вызов и установить обратный вызов. Таким образом, программа может продолжить работу, а пользователь не окажется заблокирован. Это задействует многопоточный Python и выходит за рамки данной книги, но метод использования обратного вызова представляет собой общий способ для подобных случаев.

Используем обратный вызов с SimpleButton

Для демонстрации этого понятия внесем незначительные изменения в класс SimpleButton, чтоб он мог принять обратный вызов. В качестве дополнительного необязательного параметра вызывающий может предоставить функцию или метод объекта для обратного вызова, когда происходит нажатие объекта SimpleButton. Каждый экземпляр SimpleButton запоминает обратный вызов в переменной экземпляра. Когда пользователь завершает щелчок, экземпляр SimpleButton вызывает обратный вызов.

Основная программа в листинге 6.8 создает три экземпляра класса SimpleButton, каждый из которых обрабатывает нажатие кнопки по-разному. Первая кнопка oButtonA не обеспечивает обратный вызов; oButtonB обеспечивает обратный вызов функции; а oButtonC определяет обратный вызов метода объекта.

Файл: PygameDemo9_SimpleButtonWithCallback/ Main_SimpleButtonCallback.py

```
# pygame демо 9 - 3-кнопочный тест с обратными вызовами
```

```
#1 - Импортируем пакеты
import pygame
from pygame.locals import *
from SimpleButton import *
import sys
```

```
#2 - Определяем константы
GRAY = (200, 200, 200)
```

```

WINDOW_WIDTH = 400
WINDOW_HEIGHT = 100
FRAMES_PER_SECOND = 30

# Определяем функцию, которую необходимо использовать в качестве
# "обратного вызова"
def myCallBackFunction(): ❶
    print('User pressed Button B, called myCallBackFunction')

# Определяем класс с методом, который необходимо использовать
# в качестве "обратного вызова"
class CallBackTest(): ❷
    --- любые другие методы в этом классе ---

    def myMethod(self):
        print('User pressed ButtonC, called myMethod of the
              CallBackTest object')

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.

#5 - Инициализируем переменные
oCallBackTest = CallBackTest() ❸
# создаем экземпляры SimpleButton
# Нет обратного вызова
oButtonA = SimpleButton(window, (25, 30), ❹
                        'images/buttonAUp.png',
                        'images/buttonADown.png')
# Указываем функцию для "обратного вызова"
oButtonB = SimpleButton(window, (150, 30),
                        'images/buttonBUp.png',
                        'images/buttonBDown.png',
                        callBack=myCallBackFunction)
# Указываем метод объекта для "обратного вызова"
oButtonC = SimpleButton(window, (275, 30),
                        'images/buttonCUp.png',
                        'images/buttonCDown.png',
                        callBack=oCallBackTest.myMethod)

counter = 0

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их

```

```

for event in pygame.event.get():
    if event.type == pygame.QUIT:
        pygame.quit()
        sys.exit()

    # передаем событие кнопке, смотрим, была ли она нажата
    if oButtonA.handleEvent(event): ❸
        print('User pressed button A, handled in the main loop')

    # У oButtonB и oButtonC есть обратные вызовы,
    # не нужно проверять результаты этих вызовов
    oButtonB.handleEvent(event) ❹

    oButtonC.handleEvent(event) ❺

#8 - Выполняем действия "в рамках фрейма"
counter = counter + 1

#9 - Очищаем окно, прежде чем рисовать его заново
window.fill(GRAY)

#10 - Рисуем все элементы окна
oButtonA.draw()
oButtonB.draw()
oButtonC.draw()

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 6.8. Версия основной программы, которая обрабатывает нажатия кнопок тремя разными способами

Мы начинаем с примитивной функции `myCallBackFunction()` ❶, которая просто выводит сообщение о том, что она была вызвана. Далее у нас есть класс `CallBackTest`, содержащий метод `myMethod()` ❷, который выводит собственное сообщение о том, что он был вызван. Создаем объект `oCallBackTest` из класса `CallBackTest` ❸. Нам необходим этот объект, чтобы мы могли установить обратный вызов `oCallBack.myMethod()`.

Затем создаем три объекта `SimpleButton`, каждый из которых использует свой подход ❹. У первого, `oButtonA`, нет обратного вызова. Вторым, `oButtonB`, устанавливает обратный вызов

функции `myCallBackFunction()`. Третий, `oButtonC`, устанавливает обратный вызов `oCallBack.myMethod()`.

В основном цикле мы проверяем, щелкнул ли пользователь по какой-то из этих кнопок, с помощью вызова метода `handleEvent()` для каждой кнопки. Поскольку у `oButtonA` нет обратного вызова, мы должны проверить, равно ли возвращаемое значение `True` ❸, и, если это так, осуществить действие. При нажатии на кнопку `oButtonB` ❹ будет вызвана функция `myCallBackFunction()`, которая выведет свое сообщение. При нажатии на кнопку `oButtonC` ❺ будет вызван метод `myMethod()` объекта `oCallBackTest`, который выведет собственное сообщение.

Некоторые программисты предпочитают использовать обратный вызов, потому что цель вызова устанавливается при создании объекта. Важно понимать этот метод, особенно если вы используете пакет, требующий его применения. Однако во всем своем демонстрационном коде я буду использовать изначальный подход проверки возвращаемого значения с помощью вызова `handleEvent()`.

Выводы

В этой главе я показал, как начать работу с процедурной программой и извлечь соответствующий код для создания класса. Чтобы продемонстрировать это, мы создали класс `Ball`, затем изменили основной код нашей демопрограммы из предыдущей главы, чтобы вызвать методы класса и сообщить объекту `Ball`, что делать, не заботясь о том, как он достигает результата. Имея весь соответствующий код в отдельном классе, легко создать список объектов и экземпляры и управлять необходимым количеством объектов.

Затем мы создали класс `SimpleButton` и класс `SimpleText`, которые скрывают сложность внутри своей реализации и создают код, подходящий для многократного использования. В следующей главе я создам на основании этих классов «профессиональные» кнопки и отображения текста.

И наконец, я представил понятие обратного вызова, в котором вы передаете функцию или метод для вызова в объект. Обратный вызов вызывается позднее, когда происходит событие или завершается действие.

7

ВИДЖЕТЫ PYGAME GUI



Pygame позволяет программистам брать текстовый язык Python и использовать его для основанных на GUI программах. Окна, указывающие устройства, щелчки, перетаскивания и звуки стали стандартными составляющими в нашем опыте взаимодействия с компьютерами. С сожалением, пакет `pygame` не поставляется вместе со встроенными базовыми элементами пользовательского интерфейса, поэтому нам необходимо создавать их самостоятельно. Делаем мы это с помощью `pygame_widgets`, библиотеки виджетов GUI.

В этой главе объясняется, как стандартные виджеты, например изображения, кнопки и поля ввода или вывода, можно создать в качестве классов и как клиентский код использует их. Создание каждого элемента в качестве класса позволяет включать несколько экземпляров каждого элемента при создании GUI. Прежде чем мы начнем знакомство с виджетами GUI, я все же сначала должен рассказать еще об одной функции Python: передаче данных в вызов функции или метода.

Передаем аргументы функции или методу

Отношения между аргументами в вызове функции и определяемыми в ней параметрами один к одному, так что значение первого аргумента присваивается первому параметру, значение следующего — второму и так далее.

Рис. 7.1, взятый из главы 3, показывает, что то же самое верно при вызове метода объекта. Как мы можем видеть, первый параметр, который всегда `self`, устанавливается для объекта в вызове.

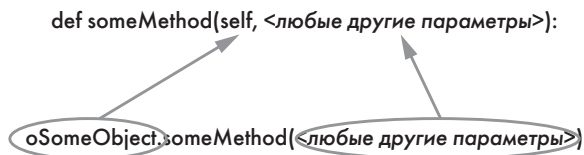


Рис. 7.1. Как аргументы, передаваемые в метод, сопоставляются со своими параметрами

Тем не менее Python (и некоторые другие языки) позволяет сделать некоторые аргументы необязательными. Если необязательный аргумент не указан в вызове, мы можем вместо него предоставить значение по умолчанию для использования в функции или методе. Я объясню с помощью аналогии из реального мира.

Если вы покупаете гамбургер в ресторане «Бургер Кинг», то получите его с кетчупом, горчицей и солеными огурчиками. Но «Бургер Кинг» известен своим высказыванием: «Вы можете сделать это по-своему». Если хотите какое-то другое сочетание приправ, вы должны сообщить об этом, когда делаете заказ.

Мы начнем с написания функции `orderBurgers()`, имитирующей заказ бургера обычным способом, которым мы определяем функции без реализации значений по умолчанию:

```
def orderBurgers(nBurgers, ketchup, mustard, pickles):
```

Вы должны указать количество гамбургеров, которое хотите заказать, но в идеале, если вы хотите использовать значения по умолчанию `True` для добавления кетчупа, горчицы и соленых огурчиков, вам не нужно передавать больше никаких аргументов. Таким образом, чтобы заказать два гамбургера со стандартными дополнениями, ваш вызов может выглядеть следующим образом:

```
orderBurgers(2) # с кетчупом, горчицей и солеными огурчиками
```

Однако в Python это приведет к ошибке, поскольку количество аргументов в вызове и количество указанных в функции параметров не совпадает:

```
TypeError: orderBurgers() missing 3 required positional arguments: 'ketchup', 'mustard', and 'pickles'
```

Давайте посмотрим, как Python дает нам возможность устанавливать необязательные параметры, которым можно присвоить значения по умолчанию, если ничего не указано.

Позиционные параметры и параметры ключевых слов

В Python есть два разных типа параметров: позиционные и параметры ключевых слов. *Позиционные параметры* представляют собой уже знакомый нам тип, в котором у каждого аргумента в вызове есть сопоставимый параметр в определении функции или метода.

Параметр ключевого слова позволяет указывать значение по умолчанию. Вы пишете параметр ключевого слова в виде имени переменной, знака равенства и значения по умолчанию следующим образом:

```
def someFunction (<параметр ключевого слова>=  
                  <значение по умолчанию>):
```

У вас может быть несколько параметров ключевых слов, каждый со своим именем и значением по умолчанию.

У функции или метода могут быть как позиционные параметры, так и параметры ключевых слов, в таком случае вы должны указать все позиционные параметры *перед* любыми параметрами ключевых слов:

```
def someOtherFunction(positionalParam1, positionalParam2, ...  
    <параметр ключевого слова1>=<значение по умолчанию1>,  
    <параметр ключевого слова2>=<значение по умолчанию2>, ...):
```

Давайте перепишем `orderBurgers()`, чтобы она использовала один позиционный параметр и три параметра ключевых слов со значениями по умолчанию, следующим образом:

```
def orderBurgers(nBurgers, ketchup=True, mustard=True, pickles=True):
```

Когда мы вызываем эту функцию, `nBurgers` является позиционным параметром и, следовательно, должен указываться в качестве аргумента при каждом вызове. Другие три — это параметры ключевых слов. Если никакие значения не будут переданы для `ketchup`, `mustard` и `pickles`, функция станет использовать значение по умолчанию `True` для каждой переменной этих параметров. Теперь мы можем заказать два бургера со всеми приправами следующим образом:

```
orderBurgers(2)
```

Если мы хотим нечто отличающееся от значения по умолчанию, то можем указать в нашем вызове имя параметра ключевого слова и другое значение. Например, если хотим в наши два бургера добавить лишь кетчуп, мы можем выполнить вызов следующим образом:

```
orderBurgers(2, mustard=False, pickles=False)
```

Когда функция выполняется, значения переменных `mustard` и `pickles` установлены равными `False`. Поскольку мы не указали значение для `ketchup`, ему по умолчанию присваивается значение `True`.

Вы также можете выполнить вызов, указывая все аргументы позиционно, включая те, что записаны в качестве параметров ключевых слов. Python будет использовать порядок ваших аргументов, чтобы присвоить каждому параметру верное значение:

```
orderBurgers(2, True, False, False)
```

В этом вызове мы снова указываем два бургера с кетчупом без горчицы и соленых огурчиков.

Дополнительные примечания к параметрам ключевых слов

Давайте быстренько пробежимся по некоторым соглашениям и советам для использования параметров ключевых слов. Исходя из соглашения Python, когда вы используете параметры ключевых слов и ключевые слова с аргументами, знак равенства между ключевым словом и значением *не* должен содержать пробелов перед и после него, чтобы показать, что это

не является типичным оператором присваивания. Здесь строки отформатированы правильно:

```
def orderBurgers(nBurgers, ketchup=True, mustard=True,
                pickles=True):

orderBurgers(2, mustard=False)
```

Эти строки тоже будут хорошо работать, но они не следуют соглашению форматирования и хуже читаются:

```
def orderBurgers(nBurgers, ketchup = True, mustard = True,
                pickles = True):

orderBurgers(2, mustard = False)
```

При вызове функции, у которой есть и позиционные параметры, и параметры ключевых слов, вы сначала должны представить значения для всех позиционных параметров до любого необязательного параметра ключевого слова.

Аргументы ключевых слов в вызовах могут указываться в любом порядке. Вызовы для нашей функции `orderBurgers()` можно осуществить различными способами, например таким:

```
orderBurgers(2, mustard=False, pickles=False) # только кетчуп
```

или:

```
orderBurgers(2, pickles=False, mustard=False, ketchup=False) # пустой
```

Всем параметрам ключевых слов будут присвоены соответствующие значения, вне зависимости от порядка аргументов.

Хотя все значения по умолчанию в примере `orderBurgers()` были булевыми выражениями, у параметра ключевого слова может быть значение по умолчанию любого типа данных. Например, мы могли написать функцию, чтобы позволить покупателю заказать мороженое следующим образом:

```
def orderIceCream(flavor, nScoops=1, coneOrCup='cone', sprinkles=False):
```

Вызывающий должен указать вкус, но по умолчанию он получил один шарик в рожке без присыпки. Вызывающий может переопределить эти значения по умолчанию другими значениями ключевых слов.

Используем None в качестве значения по умолчанию

Иногда полезно знать, передал ли вызывающий значение для параметра ключевого слова. В этом примере вызывающий заказывает пиццу. Он должен как минимум указать размер. Вторым параметром будет вид, который по умолчанию "regular", но может быть и "deepdish". В качестве третьего параметра вызывающий может дополнительно передать одну желаемую начинку. Если он хочет начинку, нам следует взимать дополнительную плату.

В листинге 7.1 мы будем использовать позиционные параметры для size и параметры ключевых слов style и topping. Значение по умолчанию для style — строка "regular". Поскольку выбор начинки необязателен, мы будем использовать специальное значение Python None в качестве значения по умолчанию, но вызывающий может передать начинку по своему выбору.

Файл: OrderPizzaWithNone.py

```
def orderPizza(size, style='regular', topping=None):
    # Произвести некоторые расчеты на основании размера и вида
    # проверяем, была ли указана начинка
    PRICE_OF_TOPPING = 1.50 # цена на любую начинку

    if size == 'small':
        price = 10.00
    elif size == 'medium':
        price = 14.00
    else: # большой
        price = 18.00

    if style == 'deepdish':
        price = price + 2.00 # берем дополнительную плату за высокие
                            # бортики

    line = 'You have ordered a ' + size + ' ' + style + ' pizza with '
    ❶ if topping is None: # проверяем, была ли передана начинка
        print(line + 'no topping')
    else:
        print(line + topping)
        price = price + PRICE_OF_TOPPING

    print('The price is $', price)
    print()
```

```
# Вы можете заказать пиццу следующими способами:
❷ orderPizza('large') # большая, обычная по умолчанию, без начинки

orderPizza('large', style='regular') # то же, что и выше

❸ orderPizza('medium', style='deepdish', topping='mushrooms')

orderPizza('small', topping='mushrooms') # вид по умолчанию обычный
```

Листинг 7.1. Версия основной программы, которая обрабатывает нажатия кнопок тремя разными способами

Первый и второй вызовы будут рассматриваться как одно и то же со значением переменной `topping`, равной `None` ❷. В третьем и четвертом вызовах значения `topping` установлены на `"mushrooms"` ❸. Поскольку `"mushrooms"` — это не `None`, в этих вызовах код будет добавлять дополнительную плату за начинку пиццы ❶.

Используя `None` в качестве значения по умолчанию для параметра ключевого слова, вы можете увидеть, предоставил ли вызывающий значение в вызов. Это очень тонкое использование параметров ключевых слов, но оно будет очень полезно в предстоящем обсуждении.

Выбираем ключевые слова и значения по умолчанию

Использование значений по умолчанию упрощает вызовы функций и методов, но есть и обратная сторона. Ваш выбор каждого ключевого слова для его параметра очень важен. Как только программисты начинают выполнять вызовы, которые переопределяют значения по умолчанию, становится очень сложно изменить имя параметра ключевого слова, потому что оно должно быть изменено во *всех* вызовах функции или метода в жесткой конфигурации. В противном случае работающий код сломается. Для более широко распределенного кода это потенциально может привести к значительным неудобствам для использующих его программистов. Итог: не меняйте имя параметра ключевого слова без острой необходимости. То есть выбирайте с умом!

Также очень важно использовать значения по умолчанию, которые подойдут широкому диапазону пользователей. (Лично я ненавижу горчицу! Каждый раз, когда я иду в «Бургер Кинг», мне приходится помнить, что я должен уточнить, чтобы

не клали горчицу, или я получу то, что считаю несъедобным бургером. На мой взгляд, они сделали неправильный выбор по умолчанию.)

Значения по умолчанию в виджетах GUI

В следующем разделе я представлю набор классов, которые вы можете использовать, чтобы легко создавать элементы GUI в `pygame`, такие как кнопки и текстовые поля. Эти классы инициализируются с помощью нескольких позиционных параметров, но в них также будут соответствующие необязательные параметры ключевых слов, все с разумными значениями по умолчанию, чтобы позволить программистам создавать виджеты GUI, указывая лишь несколько позиционных аргументов. Более точное управление можно получить за счет указания значений для перезаписи значений по умолчанию параметров ключевых слов.

В качестве углубленного примера мы рассмотрим виджет для отображения текста в окне приложения. Он может отображаться различными шрифтами, размерами шрифтов, цветами, фоновыми цветами и так далее. Создадим класс `DisplayText`, у которого будут значения по умолчанию для всех этих атрибутов, но который предоставит коду клиента возможность указывать разные значения.

Пакет `pygame`widgets

В оставшейся части главы мы сосредоточимся на пакете `pygame`widgets, который был написан с двумя целями.

1. Чтобы продемонстрировать множество объектно-ориентированных методов программирования.
2. Чтобы позволить программистам с легкостью создавать и использовать виджеты GUI в программах `pygame`.

Пакет `pygame`widgets содержит следующие классы.

`TextButton`

Кнопка, созданная со стандартным оформлением с помощью текстовой строки.

`CustomButton`

Кнопка с пользовательской графикой.

TextCheckBox

Флажок со стандартным оформлением, созданный из текстовой строки.

CustomCheckBox

Флажок с пользовательской графикой.

TextRadioButton

Переключатели со стандартным оформлением, созданные из текстовой строки.

CustomRadioButton

Переключатели с пользовательской графикой.

DisplayText

Поле, использованное для отображения выводимого текста.

InputText

Поле, где пользователь может вводить текст.

Dragger

Позволяет пользователю перетаскивать изображение.

Image

Отображает изображение в указанном месте.

ImageCollection

Отображает набор изображений в указанном месте.

Animation

Отображает последовательность изображений.

SpriteSheetAnimation

Отображает последовательность изображений из одного более крупного изображения.

Установка

Чтобы установить `pygwidgets`, откройте командную строку и введите следующее:

```
python3 -m pip install -U pip --user
python3 -m pip install -U pygwidgets --user
```

Эти команды загружают и устанавливают последнюю версию `pygwidgets` из каталога пакетов Python (PyPI). Она помещается в папку (с именем *site-packages*), которая доступна всем вашим программам Python. Как только пакет был установлен, вы можете использовать `pygwidgets`, включая следующего оператора в начало вашей программы:

```
import pygwidgets
```

Это импортирует весь пакет. После завершения импорта вы можете создать экземпляры объектов из их классов и вызвать методы этих объектов.

Самая актуальная документация `pygwidgets` находится по адресу <https://pygwidgets.readthedocs.io/en/latest/>. Если хотите просмотреть исходный код пакета, он доступен в репозитории GitHub по адресу <https://github.com/IrvKalb/pygwidgets/>.

Общий подход к разработке

Как показано в главе 5, одно из первых действий, которые вы выполняете в каждой программе `pygame`, — определение окна приложения. Следующие строки создают окно приложения и сохраняют ссылки на него в переменной с именем `window`:

```
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
```

Как вскоре увидим, каждый раз, когда мы создаем экземпляр любого виджета, нам будет необходимо передавать переменную `window`, чтобы виджет мог нарисовать себя в окне приложения.

Большинство виджетов в `pygwidgets` работают похожим образом, обычно включая следующие три шага.

1. Перед началом основного цикла `while` создайте экземпляр виджета с некоторыми аргументами инициализации.
2. В основном цикле каждый раз, когда происходит какое-либо событие, вызывайте метод виджета `handleEvent()` (передавая в объект события).
3. В нижней части основного цикла вызовите метод виджета `draw()`.

Шаг 1 при использовании любого виджета заключается в создании его экземпляра с помощью строки, подобной этой:

```
oWidget = pygwidgets.<SomeWidgetClass>(window, loc,  
    <прочие необходимые аргументы>)
```

Первым аргументом всегда является окно приложения. Вторым аргумент — это всегда координаты в окне для отображения виджета в виде кортежа: `(x, y)`.

Шаг 2 заключается в обработке любого события, которое могло повлиять на виджет, и вызове метода объекта `handleEvent()` внутри каждого цикла. Если происходит

какое-либо событие (например, щелчок мышью или нажатие кнопки) и виджет обрабатывает событие, этот вызов вернет значение `True`. Код в верхней части основного цикла `while` обычно выглядит следующим образом:

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    if oWidget.handleEvent(event):
        # пользователь сделал что-то с oWidget, на что
        # мы должны отреагировать; добавляем здесь код
```

Шаг 3 состоит в добавлении строки в нижней части цикла `while` для вызова метода виджета `draw()`, чтобы он появился в окне:

```
oWidget.draw()
```

Поскольку в шаге 1 мы указали окно для рисования, местоположение и все детали, которые влияют на появление виджета, нам не нужно ничего передавать в вызов `draw()`.

Добавляем изображение

Первым примером будет простейший виджет: мы будем использовать класс `Image`, чтобы отобразить изображение в окне. Когда вы создаете экземпляр объекта `Image`, единственными необходимыми аргументами являются окно, местоположение в окне для рисования изображения и путь к файлу изображения. Создайте объект `Image` перед началом основного цикла следующим образом:

```
oImage = pygwidgets.Image(window, (100, 200), 'images/SomeImage.png')
```

Используемый здесь путь предполагает, что папка проекта, содержащая основную программу, также содержит папку с именем *images*, внутри которой находится файл *SomeImage.png*. Затем в основном цикле вам лишь необходимо вызвать метод объекта `draw()`:

```
oImage.draw()
```

Метод `draw()` класса `Image` содержит вызов `blit()` для фактического рисования изображения, поэтому вам никогда не нужно напрямую вызывать `blit()`. Чтобы переместить изображение, вы можете вызвать метод `setLoc()` (сокращение от «установка местоположения»), указывая координаты `x` и `y` в виде кортежа:

```
oImage.setLoc((newX, newY))
```

В следующий раз, когда изображение будет нарисовано, оно появится в новых координатах. В документации перечислено множество дополнительных методов, которые вы можете вызывать, чтобы зеркально отобразить, развернуть, получить местоположение и прямоугольник изображения и так далее.

МОДУЛЬ СПРАЙТ

В `pygame` есть встроенный модуль `sprite` для отображения изображений в окне. Они называются *спрайтами*. Модуль `спрайт` предоставляет класс `Sprite` для обработки отдельных спрайтов и класс `Group` для обработки нескольких объектов `Sprite`. Вместе эти классы обеспечивают великолепные функциональные возможности, и, если вы намереваетесь программировать `pygame` в тяжелом режиме, возможно, стоит потратить время на знакомство с ними. Тем не менее, чтобы объяснить основополагающие понятия ООП, я решил не использовать эти классы. Взамен я перейду к общим элементам GUI, чтобы их можно было применять в любой среде и на любом языке. Если вы хотите больше узнать о модуле `sprite`, ознакомьтесь с руководством по адресу <https://www.pygame.org/docs/tut/SpriteIntro.html>.

Добавляем кнопки, флажки и переключатели

Когда создаете экземпляр кнопки, флажка или переключателя виджета в `pygame.widgets`, у вас есть два варианта: создать экземпляр текстовой версии, которая рисует собственную графику и добавляет текстовую метку на основе переданной вами строки, или создать экземпляр пользовательской версии, в которой вы предоставляете изображение. В табл. 7.1 продемонстрированы различные доступные классы кнопок.

Таблица 7.1. Текстовые и пользовательские классы кнопок в pygwidgets

	Текстовая версия (создает графику по ходу действия)	Пользовательская версия (использует вашу графику)
Кнопка	TextButton	CustomButton
Флажок	TextCheckBox	CustomCheckBox
Переключатель	TextRadioButton	CustomRadioButton

Разница между текстовой и пользовательской версиями этих классов имеет значение только во время создания экземпляра. Как только вы создали объект из текстового или пользовательского класса кнопки, все остальные методы пары классов идентичны. Чтобы прояснить это, давайте рассмотрим классы `TextButton` и `CustomButton`.

TextButton

Ниже представлено фактическое определение метода `__init__()` класса `TextButton` в `pygwidgets`:

```
def __init__(self, window, loc, text,
             width=None,
             height=40,
             textColor=PYGWIDGETS_BLACK,
             upColor=PYGWIDGETS_NORMAL_GRAY,
             overColor=PYGWIDGETS_OVER_GRAY,
             downColor=PYGWIDGETS_DOWN_GRAY,
             fontName=DEFAULT_FONT_NAME,
             fontSize=DEFAULT_FONT_SIZE,
             soundOnClick=None,
             enterToActivate=False,
             callback=None,
             nickname=None):
```

Однако вместо того, чтобы читать код класса, программист скорее всего обратится к его документации. Как упоминалось ранее, полная документация `pygwidgets` находится по *адресу* <https://pygwidgets.readthedocs.io/en/latest/>.

Также вы можете посмотреть документацию класса, вызвав встроенную функцию `help()` в оболочке Python следующим образом:

```
>>> help(pygwidgets.TextButton)
```

Когда создаете экземпляр `TextButton`, вам необходимо лишь передать окно, местоположение в окне и текст для отображения на кнопке. Если укажете только эти позиционные параметры, ваша кнопка будет использовать разумные значения по умолчанию для ширины и высоты, фоновых цветов для четырех состояний кнопки (различные оттенки серого), типа и размера шрифта. По умолчанию, когда пользователь щелкает по кнопке, никакие звуковые эффекты не воспроизводятся.

Код для создания `TextButton` с помощью этих значений по умолчанию выглядит следующим образом:

```
oButton = pygwidgets.TextButton(window, (50, 50), 'Text Button')
```

Код в методе `__init__()` класса `TextButton` использует методы рисования `pygame`, чтобы сконструировать собственную графику для всех четырех состояний (вверх, вниз, мышь наведена и недоступна для нажатия). Следующая строка создает версию кнопки «вверх», которая выглядит как на рис. 7.2.



Рис. 7.2. Кнопка `TextButton`, использующая значения по умолчанию

Вы можете переопределить любой или все параметры по умолчанию с помощью значений ключевых слов следующим образом:

```
oButton = pygwidgets.TextButton(window, (50, 50),
                                'Text Button',
                                width=200,
                                height=30,
                                textColor=(255, 255, 128),
                                upColor=(128, 0, 0),
                                fontName='Courier',
                                fontSize=14,
                                soundOnClick='sounds/blip.wav',
                                enterToActivate=True)
```

Эта установка создаст кнопку, которая будет выглядеть как на рис. 7.3.

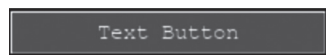


Рис. 7.3. Кнопка `TextButton`, использующая аргументы ключевых слов для шрифта, размера, цвета и так далее

Поведение переключения изображения этих двух кнопок работает абсолютно так же; единственное отличие заключается во внешнем виде изображений.

CustomButton

Класс CustomButton позволяет вам использовать собственную графику для кнопки. Чтобы создать экземпляр CustomButton, вам необходимо лишь передать окно, координаты и путь к изображению состояния «вверх» кнопки. Ниже приведен пример:

```
restartButton = pygamewidgets.CustomButton(window, (100, 430),
                                             'images/RestartButtonUp.png')
```

Состояния down, over и disabled — необязательные аргументы ключевых слов, и, если для любого из них не будет передано значение, CustomButton будет использовать копию изображения up. Более типично (и настоятельно рекомендуется) передавать путь к дополнительным изображениям следующим образом:

```
restartButton = pygamewidgets.CustomButton(window, (100, 430),
                                             'images/RestartButtonUp.png',
                                             down='images/RestartButtonDown.png',
                                             over='images/RestartButtonOver.png',
                                             disabled='images/RestartButtonDisabled.png',
                                             soundOnClick='sounds/blip.wav',
                                             nickname='restart')
```

Здесь мы также указали звуковой эффект, который должен воспроизводиться, когда пользователь щелкает по кнопке, и мы предоставили внутренний псевдоним, который мы сможем использовать позднее.

Используем кнопки

После создания экземпляра вы можете применить некоторый типичный код, чтобы использовать объект кнопки oButton независимо от того, является ли он TextButton или CustomButton:

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
```

```

        sys.exit()

    if oButton.handleEvent(event):
        # Пользователь щелкнул по кнопке
        <Код, выполняемый при нажатии кнопки>
--- пропуск ---
oButton.draw() # внизу цикла while сказать ему нарисовать

```

Каждый раз, когда обнаруживаем событие, нам необходимо вызвать метод кнопки `handleEvent()`, чтобы разрешить ей отреагировать на действия пользователя. Обычно этот вызов возвращает `False`, но вернет `True`, когда пользователь завершит щелчок по кнопке. В нижней части основного цикла `while` нам необходимо вызвать метод кнопки `draw()`, чтобы разрешить ей нарисовать себя.

Вывод и ввод текста

Как мы видели в главе 6, обработка входных и выходных данных текста в `pygame` достаточно сложна, но я здесь познакомлю вас с новыми классами поля отображения текста и поля ввода текста. У них обоих минимальное количество необходимых (позиционных) параметров и разумные значения по умолчанию для других атрибутов (шрифт, размер шрифта, цвет и так далее), которые легко переопределяются.

Вывод текста

Пакет `pygame` содержит класс `DisplayText` для отображения текста, который является более полнофункциональной версией класса `SimpleText` из главы 6. Когда вы создаете экземпляр поля `DisplayText`, единственные обязательные аргументы — окно и местоположение. Первый параметр ключевого слова — `value`, который можно указать с помощью строки в качестве начального текста для отображения в поле. Обычно это используется для значения по умолчанию конечного пользователя или для никогда не меняющегося текста, такого как метка или инструкции. Поскольку `value` — первый параметр ключевого слова, он может быть задан и как позиционный аргумент, и как аргумент ключевого слова. Например, вот это:

```
oTextField = pygamewidgets.DisplayText(window, (10, 400), 'Hello World')
```

будет работать так же, как и вот это:

```
oTextField = pygwidgets.DisplayText(window, (10, 400),  
                                     value='Hello World')
```

Вы также можете настроить вид выводимого текста, указывая любой или все необязательные параметры ключевых слов. Например:

```
oTextField = pygwidgets.DisplayText(window, (10, 400),  
                                     value='Some title text',  
                                     fontName='Courier',  
                                     fontSize=40,  
                                     width=150,  
                                     justified='center',  
                                     textColor=(255, 255, 0))
```

У класса `DisplayText` есть определенное количество дополнительных методов. Наиболее важный из них — `setValue()`, который вы вызываете для изменения нарисованного в поле текста:

```
oTextField.setValue('Any new text you want to see')
```

В нижней части основного цикла `while` вам необходимо вызывать метод объекта `draw()`:

```
oTextField.draw()
```

И, конечно же, вы можете создать столько объектов `DisplayText`, сколько вам необходимо.

Ввод текста

В типичной текстовой программе Python, чтобы получить входные данные от пользователя, вы вызвали бы функцию `input()`, которая останавливает программу, пока пользователь вводит текст в окне оболочки. Но в мире управляемых событиями GUI-программ основной цикл не останавливается никогда. Следовательно, мы должны использовать иной подход.

Для вводимого пользователем текста GUI-программа обычно представляет поле, в котором пользователь может печатать. Поле ввода должно иметь дело со всеми клавишами клавиатуры, некоторые из них отображаются, в то время как другие используются для редактирования или передвижений курсора внутри поля. Оно также должно позволять пользователю,

удерживающему клавишу, повторить ее. Класс `pygame.InputText` обеспечивает все эти функциональные возможности.

Единственными обязательными аргументами для создания экземпляра объекта `InputText` являются окно и местоположение:

```
oInputField = pygame.InputText(window, (10, 100))
```

Однако вы можете настроить атрибуты текста объекта `InputText`, указав необязательные параметры ключевых слов:

```
oInputField = pygame.InputText(window, (10, 400),
                                value='Starting Text',
                                fontName='Helvetica',
                                fontSize=40,
                                width=150,
                                textColor=(255, 255, 0))
```

После создания экземпляра поля `InputText` типичный код основного цикла будет выглядеть следующим образом:

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        if oInputField.handleEvent(event):
            # Пользователь нажал Enter или Return
            userText = oInputField.getValue() # получаем
                                                # введенный пользователем текст
            <Код, выполняемый после ввода пользователя>
    --- пропуск ---
    oInputField.draw() # внизу основного цикла while
```

Для каждого события нам необходимо вызвать метод `handleEvent()` поля `InputText`, чтобы позволить ему реагировать на нажатия клавиш и щелчки мыши. Обычно этот вызов возвращает значение `False`, но, когда пользователь нажимает клавишу **Enter** или **Return**, он возвращает `True`. Затем мы можем извлечь текст, который пользователь ввел, вызвав метод объекта `getValue()`.

В нижней части основного цикла `while` нам необходимо вызвать метод `draw()`, чтобы разрешить полю нарисовать себя.

Если окно содержит несколько полей ввода, нажатия клавиш обрабатываются полем с текущим фокусом клавиатуры, которое изменяется, когда пользователь щелкает по другому полю. Если вы хотите разрешить полю иметь начальный фокус клавиатуры, то можете установить параметр ключевого слова `initialFocus` в значение `True` в выбранном вами объекте `InputText`, когда создаете его. Далее, если у вас в окне есть несколько полей `InputText`, типичный подход к дизайну интерфейса пользователя предполагает наличие кнопки **ОК** или **Submit**. Когда щелкаете по ней, вы можете затем вызвать метод `getValue()` для каждого поля.

ПРИМЕЧАНИЕ На момент написания книги класс `InputText` не обеспечивал выделение нескольких символов с помощью перетаскивания мыши. Если эта функциональная возможность была добавлена в более поздней версии, для программ, использующих `InputText`, не потребуются никаких изменений, поскольку код будет полностью находиться внутри этого класса. Любое новое поведение станет автоматически поддерживаться во всех объектах `InputText`.

Другие классы `pygame`

Как вы видели в начале этого раздела, `pygame` содержит некоторое количество других классов.

Класс `ImageCollection` позволяет показывать любое отдельное изображение из набора. Например, предположим, у вас есть изображение персонажа, стоящего лицом, спиной и повернутым влево и вправо. Чтобы представить все потенциальные изображения, вы можете создать словарь следующим образом:

```
imageDict = {'front': 'images/front.png', 'left': 'images/left.png',
             'back': 'images/back.png', 'right': 'images/right.png'}
```

Затем создайте объект `ImageCollection`, указывая этот словарь и ключ изображения, с которого хотите начать. Чтобы перейти к другому изображению, вы вызываете метод `replace()` и передаете другой ключ. Вызов метода `draw()` в нижней части цикла всегда показывает текущее изображение.

Класс `Dragger` показывает одно изображение, но позволяет пользователю перетаскивать его в любое место в окне. Вы должны вызвать его метод `handleEvent()` в цикле события. Когда пользователь завершит перетаскивание, `handleEvent()` вернет значение `True` и вы сможете вызвать метод `getMouseUpLoc()` объекта `Dragger`, чтобы получить местоположение, в котором пользователь отпустил кнопку мыши.

Классы `Animation` и `SpriteSheetAnimation` обрабатывают создание и отображение анимации. Оба они требуют набора изображений для итерации. Класс `Animation` получает картинки из отдельных файлов, в то время как классу `SpriteSheetAnimation` требуется одно изображение с равномерно расположенными внутренними картинками. Мы более подробно изучим эти классы в главе 14.

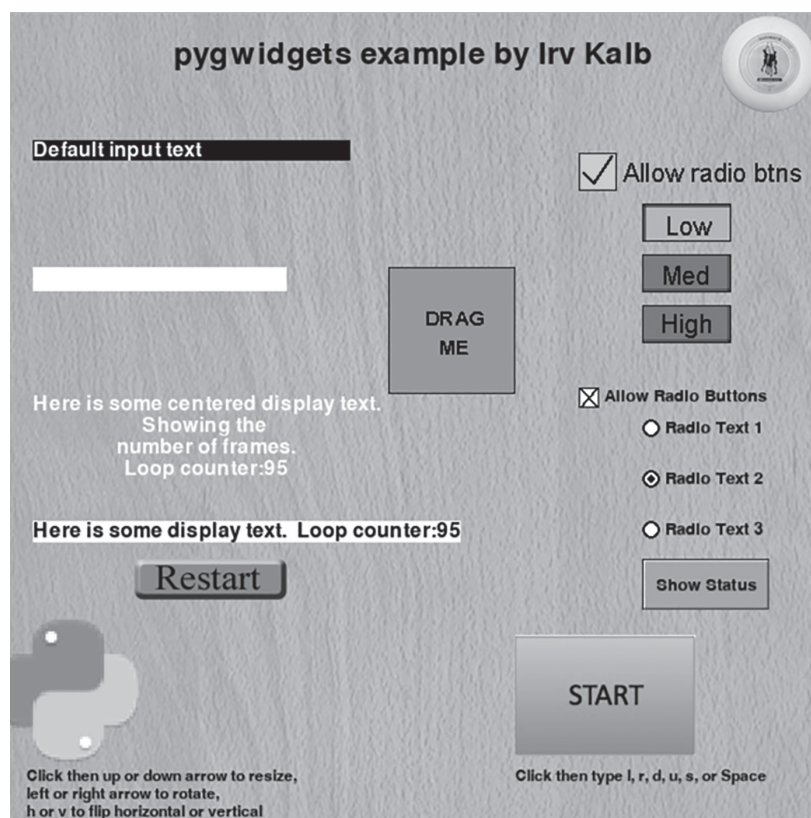


Рис. 7.4. Окно программы, показывающее объекты, для которых были созданы экземпляры из различных классов в `pygamewidgets`

pygwidgets в примере программы

На рис. 7.4 показан скриншот примера программы, демонстрирующей объекты, для которых были созданы экземпляры из множества классов в `pygwidgets`, включая `Image`, `DisplayText`, `InputText`, `TextButton`, `CustomButton`, `TextRadioButton`, `CustomRadioButton`, `TextCheckBox`, `CustomCheckBox`, `ImageCollection` и `Dragger`.

Исходный код этого примера программы можно найти в папке `pygwidgets_test` в репозитории GitHub по адресу <https://github.com/IrvKalb/pygwidgets/>.

Важность последовательного API

Одно последнее замечание по поводу создания API для набора классов: когда это возможно, прекрасной идеей будет обеспечение согласованности параметров методов в различных, но похожих классах. Хорошими примерами являются первые два параметра метода `__init__()` каждого класса в `pygwidgets` — `window` и `loc`, именно в этом порядке. Если в некоторых вызовах порядок будет иной, может оказаться сложнее использовать пакет как единое целое.

Кроме того, если различные классы реализуют одни и те же функциональные возможности, стоит использовать одинаковые имена методов. Например, у многих классов в `pygwidgets` есть метод с именем `setValue()` и другой с именем `getValue()`. В следующих двух главах я подробнее расскажу, почему этот тип согласованности так важен.

Выводы

Эта глава познакомила вас с объектно-ориентированным пакетом `pygwidgets` виджетов графического интерфейса пользователя. Мы начали с обсуждения значений по умолчанию для параметров в методах, и я объяснил, что параметры ключевых слов позволяют использовать значения по умолчанию, если в вызове не было указано соответствующее значение аргумента.

Затем я познакомил вас с модулем `pygwidgets`, содержащим некоторое количество предустановленных классов виджетов GUI, и показал вам, как использовать какие-то из них.

И наконец, я показал программу, предоставляющую примеры большинства этих виджетов.

Существует два ключевых преимущества при написании подобных классов в `pygame`. Во-первых, классы могут скрыть сложность методов. Как только вы заставите ваш класс работать правильно, вам никогда не придется снова беспокоиться о внутренних деталях. Во-вторых, вы можете повторно использовать код, создавая столько экземпляров класса, сколько вам необходимо. Ваш класс может обеспечивать основные функциональные возможности, включая параметры ключевых слов с тщательно подобранными значениями по умолчанию. Однако значения по умолчанию можно легко переопределить, чтобы обеспечить возможность настройки.

Вы можете опубликовать интерфейсы ваших классов для других программистов (и себя), чтобы использовать их в различных проектах. Хорошая документация и совместимость идут рука об руку, чтобы сделать эти типы классов очень удобными для использования.

ЧАСТЬ III

ИНКАПСУЛЯЦИЯ, ПОЛИМОРФИЗМ И НАСЛЕДОВАНИЕ

Три основных принципа объектно-ориентированного программирования — инкапсуляция, полиморфизм и наследование. Следующие три главы объяснят каждое из этих понятий по очереди, описывая основополагающие концепции и демонстрируя примеры их реализации в Python. Чтобы язык программирования мог называть себя языком ООП, он должен поддерживать все три требования.

В главе 8 объясняется инкапсуляция: сокрытие деталей и размещение всего в одном месте.

В главе 9 обсуждается полиморфизм: как у нескольких классов могут быть методы с одинаковыми именами.

Глава 10 охватывает наследование: создание на основании уже существующего кода.

И наконец, глава 11 подробно рассматривает несколько тем (в основном связанных с управлением памятью), которые логически не вписываются в предыдущие три главы, но полезны и важны для ООП.

8

ИНКАПСУЛЯЦИЯ



Первый из трех основных принципов объектно-ориентированного программирования — *инкапсуляция*. Это слово может вызвать в воображении образ космической капсулы, клеточной

оболочки или медицинской гелевой капсулы, в которых находящийся внутри ценный груз защищен от воздействий внешней среды. В программировании инкапсуляция имеет схожее, но даже более детализированное значение — сокрытие внутренних деталей состояния и поведения от внешнего кода и нахождение кода в одном месте.

В этой главе мы посмотрим, как работает инкапсуляция с помощью функций и с помощью методов объектов. Я рассмотрю различные интерпретации инкапсуляции: использование прямого доступа по сравнению с применением геттеров и сеттеров. Я покажу, как Python позволяет помечать переменную экземпляра как закрытую, указывая, что она не должна быть доступна коду, внешнему по отношению к классу, и коснусь особенности Python под названием декоратор. И наконец, я рассмотрю понятие абстракции при проектировании классов.

Инкапсуляция с помощью функций

Функции — яркий пример инкапсуляции, потому что, когда вы вызываете функцию, вас обычно не волнует, *как* она работает внутри. Грамотно написанная функция содержит набор шагов, которые составляют одну более крупную задачу, которая вам и важна. Имя функции должно описывать действие, реализующее ее код. Возьмем встроенную функцию `len()` из стандартной библиотеки Python, которая используется для поиска количества символов в строке или элементов в списке. Вы передаете строку или список, а она возвращает число. Когда пишете код, который вызывает эту функцию, вас не волнует, как `len()` выполняет свою работу. Вы не задумываетесь о том, содержит код функции две или две тысячи строк, использует ли он одну локальную переменную или сотню. Вам лишь необходимо знать, какой аргумент передать и как использовать возвращенный результат.

То же самое верно и для написанных вами функций, как, например, эта, вычисляющая и возвращающая среднее значение списка чисел:

```
def calculateAverage(numbersList):
    total = 0.0
    for number in numbersList:
        total = total + number
    nElements = len(numbersList)
    average = total / nElements
    return average
```

Как только вы протестировали подобную функцию и обнаружили, что она работает, больше не стоит беспокоиться о деталях реализации. Вам лишь необходимо знать, какой аргумент(ы) отправить функции и что она возвращает.

Однако если однажды вы обнаружите, что существует гораздо более простой и быстрый алгоритм для вычисления среднего значения, то можете переписать функцию новым способом. Пока интерфейс (входные и выходные данные) не меняется, нет необходимости менять какие-либо вызовы функции. Этот тип модуляризации облегчает обслуживание кода.

Инкапсуляция с помощью объектов

В отличие от используемых в регулярных функциях переменных, переменные экземпляра в объектах сохраняются при вызовах различных методов. Чтобы дальнейшее обсуждение было понятным, я введу новый термин: *клиент*. (Я не хочу здесь использовать термин *пользователь*, поскольку он обычно относится к человеку — пользователю конечной программы.)

Клиент

Любое программное обеспечение, которое создает объект из класса и вызывает методы этого объекта.

Мы должны также учитывать, что существует разница в подходе *внутри* и *снаружи* объекта или класса. Когда вы работаете внутри класса (пишете код методов в классе), необходимо позаботиться о том, как различные его методы совместно используют переменные экземпляра. Вы учитываете эффективность ваших алгоритмов. Думаете о том, как должен выглядеть интерфейс: какие методы следует предоставить, каковы параметры для каждого из них и что должно быть использовано в качестве значений по умолчанию. Если коротко, вы обязаны позаботиться о проектировании и реализации методов.

С внешней стороны, как программист клиента, вы должны знать интерфейс класса. Вы заботитесь о том, что делают методы класса, какие аргументы должны быть переданы и какие данные передаются обратно от каждого метода.

Таким образом, класс обеспечивает инкапсуляцию за счет:

- сокрытия деталей реализации в своих методах и переменных экземпляра;
- обеспечения всех функциональных возможностей, необходимых клиенту от объекта через его интерфейс (методы, определенные в классе).

Объекты владеют своими данными

В объектно-ориентированном программировании данные внутри объекта *принадлежат* ему. Программисты ООП обычно согласны, что хороший принцип проектирования состоит в том, чтобы клиентский код касался лишь интерфейса класса

и не заботился о реализации методов. В листинге 8.1 рассмотрен пример простого класса `Person`.

```
class Person():

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
```

Листинг 8.1. Владение данными в классе `Person`

Значения переменных экземпляра `self.name` и `self.salary` устанавливаются каждый раз, когда мы создаем экземпляры новых объектов `Person`, следующим образом:

```
oPerson1 = Person('Joe Schmoe', 90000)
oPerson2 = Person('Jane Smith', 99000)
```

Каждый объект `Person` владеет собственным набором двух переменных экземпляра.

Интерпретации инкапсуляции

Именно здесь возникают небольшие противоречия. Мнения программистов по поводу доступности переменной экземпляра могут отличаться. Python представляет свободную интерпретацию инкапсуляции, разрешая прямой доступ к переменным экземпляра с использованием синтаксиса простых точек. Клиентский код способен получить законный доступ к переменной экземпляра объекта по имени, используя синтаксис `<объект>.<имя переменной экземпляра>`.

Тем не менее *строгая* интерпретация инкапсуляции говорит, что клиентское программное обеспечение никогда не должно извлекать или изменять значение переменной экземпляра напрямую. Взамен единственный способ, которым клиент может извлечь содержащееся в объекте значение, заключается в использовании метода, предоставленного для этой цели классом.

Давайте рассмотрим оба подхода.

Прямой доступ и почему его следует избегать

Как уже упоминалось, Python разрешает прямой доступ к переменным экземпляра. В листинге 8.2 создаются экземпляры тех же двух объектов из класса `Person` листинга 8.1, что

и в предыдущем разделе, но затем осуществляется прямой доступ к их переменным экземпляра `self.salary`.

Файл: **PersonGettersSettersAndDirectAccess/** **Main_PersonDirectAccess.py**

```
# Пример основной программы Person с использованием прямого доступа

from Person import *

oPerson1 = Person('Joe Schmoe', 90000)
oPerson2 = Person('Jane Smith', 99000)

# получаем значения переменных зарплаты
❶ print(oPerson1.salary)
print(oPerson2.salary)

# меняем переменную зарплаты
❷ oPerson1.salary = 100000
oPerson2.salary = 111111

# обновляем зарплаты и выводим снова
print(oPerson1.salary)
print(oPerson2.salary)
```

Листинг 8.2. Пример основного кода, использующий прямой доступ к переменной экземпляра

Python разрешает писать подобный код, который обращается к объектам, чтобы напрямую получить ❶ и установить ❷ новую переменную экземпляра с помощью стандартного синтаксиса точки. Большинство программистов Python считают эту технику абсолютно приемлемой. Гвидо ван Россум (создатель Python) однажды высказался относительно этого вопроса: «Мы все здесь взрослые люди», — имея в виду, что программисты должны знать, что они делают и чем рискуют, когда пытаются напрямую получить доступ к переменным экземпляра.

Тем не менее я твердо убежден, что прямой доступ к переменной экземпляра объекта представляет собой невероятно опасную практику, поскольку нарушает основную идею инкапсуляции. Чтобы проиллюстрировать этот случай, давайте рассмотрим несколько примеров сценариев, в которых прямой доступ может вызвать проблемы.

Меняем имя переменной экземпляра

Первая проблема прямого доступа состоит в том, что изменение имени переменной экземпляра разрушит любой клиентский код, который напрямую использует исходное имя. Это может произойти, когда разработчик класса решает, что первоначальный выбор имени переменной не был оптимальным по следующим причинам.

- Имя недостаточно четко описывает данные, которые оно представляет.
- Переменная является булевым выражением, и он хочет поменять местами значения True и False, переименовав переменную (например, `closed` в `open`, `allowed` в `disallowed`, `active` в `disabled`).
- В исходном имени была ошибка в написании или заглавной букве.
- Переменная была изначально булевым выражением, но позднее он осознал, что ему нужно представить более двух значений.

В любом из этих случаев, если разработчик меняет имя переменной экземпляра в классе с `self.<исходноеИмя>` на `self.<новоеИмя>`, клиентское программное обеспечение, которое использует исходное имя напрямую, выйдет из строя.

Изменение переменной экземпляра на вычисление

Вторая ситуация, когда прямой доступ представляет проблему, — если коду класса необходимо измениться, чтобы соответствовать новым требованиям. Предположим, что при написании класса вы используете переменную экземпляра, чтобы представить фрагмент данных, но функциональные возможности меняются таким образом, что вместо этого вам требуется алгоритм для вычисления значения. Возьмем в качестве примера наш класс `Account` из главы 4. Чтобы сделать банковские счета более реалистичными, мы могли бы добавить процентную ставку. Вы можете подумать, что это простой вопрос добавления переменной экземпляра для процентной ставки с именем `self.interestRate`. Затем, используя прямой доступ, клиентское программное обеспечение может получить доступ к этому значению объекта `Account` с помощью:

`oAccount.interestRate`

Это сработает, но лишь на какое-то время. Позднее банк может принять новую политику: допустим, процентная ставка будет зависеть от количества денег на счету. Процентную ставку можно вычислить следующим образом:

```
def calculateInterestRate(self):
    # Предполагается, что self.balance был установлен
    # в другом методе
    if self.balance < 1000:
        self.interestRate = 1.0
    elif self.balance < 5000:
        self.interestRate = 1.5
    else:
        self.interestRate = 2.0
```

Вместо того чтобы просто полагаться на единственное значение процентной ставки в `self.interestRate`, метод `calculateInterestRate()` определяет текущую ставку на основании баланса счета.

Любое клиентское программное обеспечение, которое напрямую получает доступ к `oAccount.interestRate` и использует значение переменной экземпляра, может затем получить устаревшее значение в зависимости от времени последнего вызова `calculateInterestRate()`. И любое клиентское программное обеспечение, которое *устанавливает* новую `interestRate`, может обнаружить, что новое значение было загадочным образом изменено неким другим кодом, который вызывает `calculateInterestRate()`, или когда владелец счета вносит средства на счет или выводит с него деньги.

Однако, если метод вычисления процентной ставки был назван `getInterestRate()` и клиентская программа вызывает его, процентная ставка всегда будет вычисляться в процессе работы и не появится риск возникновения потенциальной ошибки.

Проверяем данные

Третья причина для избегания прямого доступа при установке значения состоит в том, что клиентский код может слишком легко установить переменную экземпляра в неверное значение. Лучший подход — вызов метода в классе, чья работа состоит

в установке значения. Как разработчик вы можете включить в этот метод проверочный код, чтобы убедиться, что значение устанавливается должным образом. Рассмотрим код из листинга 8.3, чья цель заключается в управлении членами клуба.

Файл: ValidatingData_ClubExample/Club.py

```
# Класс Club

class Club():

    def __init__(self, clubName, maxMembers):
        self.clubName = clubName ❶
        self.maxMembers = maxMembers
        self.membersList = []

    def addMember(self, name): ❷
        # проверяем, что осталось достаточно места
        if len(self.membersList) < self.maxMembers:
            self.membersList.append(name)
            print('OK.', name, 'has been added to the',
                  self.clubName, 'club')
        else:
            print('Sorry, but we cannot add', name, 'to the',
                  self.clubName, 'club.')
            print('This club already has the maximum of',
                  self.maxMembers, 'members.')

    def report(self): ❸
        print()
        print('Here are the', len(self.membersList),
              'members of the', self.clubName, 'club:')
        for name in self.membersList:
            print('    ' + name)
        print()
```

Листинг 8.3. Пример класса Club

Код Club отслеживает имя клуба, список его членов и их максимальное количество в переменных экземпляра ❶. Как только экземпляры созданы, вы можете вызывать методы, чтобы добавлять членов в клуб ❷ и сообщать о них ❸. (Несложно добавить больше методов: чтобы удалять членов, изменять имена и так далее, но этих двух достаточно, чтобы понять суть вопроса.)

Ниже представлен тестовый код, который использует класс Club.

Файл: ValidatingData_ClubExample/Main_Club.py

```
# Пример основной программы Club

from Club import *

# создаем клуб с максимум 5 членами
oProgrammingClub = Club('Programming', 5)

oProgrammingClub.addMember('Joe Schmoe')
oProgrammingClub.addMember('Cindy Lou Hoo')
oProgrammingClub.addMember('Dino Richmond')
oProgrammingClub.addMember('Susie Sweetness')
oProgrammingClub.addMember('Fred Farkle')
oProgrammingClub.report()
```

Мы создаем клуб программирования, который допускает максимум пять членов, и затем добавляем их. Код работает хорошо и сообщает о добавленных в клуб членах:

```
OK. Joe Schmoe has been added to the Programming club
OK. Cindy Lou Hoo has been added to the Programming club
OK. Dino Richmond has been added to the Programming club
OK. Susie Sweetness has been added to the Programming club
OK. Fred Farkle has been added to the Programming club
```

Теперь давайте добавим шестого члена:

```
# Попытка добавить дополнительного члена
oProgrammingClub.addMember('Iwanna Join')
```

Эта попытка была отклонена, и мы видим соответствующее сообщение об ошибке:

```
Sorry, but we cannot add Iwanna Join to the Programming club.
This club already has the maximum of 5 members.
```

Код `addMember()` осуществляет всю необходимую проверку, чтобы убедиться, что вызов для добавления нового члена работает правильно или генерирует сообщение об ошибке. Однако с помощью прямого доступа клиент может изменить фундаментальную природу класса `Club`. Например, клиент может

умышленно или случайно изменить максимальное количество членов:

```
oProgrammingClub.maxMembers = 300
```

Далее, предположим, вы в курсе, что класс `Club` представляет членов в виде списка, и знаете имя переменной экземпляра, которое представляет членов. В таком случае вы можете написать клиентский код, чтобы добавлять напрямую в список членов, не вызывая метод, следующим образом:

```
oProgrammingClub.memberList.append('Iwanna Join')
```

Эта строка приведет к превышению предполагаемого предела количества членов, поскольку она избегает кода, который гарантирует, что запрос на добавление нового участника правомерен.

Клиентский код, использующий прямой доступ, может даже привести к ошибке внутри объекта `Club`. Например, переменная экземпляра `self.maxMembers` должна быть целым числом. Используя прямой доступ, клиентский код может изменить свое значение на строку. Любой последующий вызов `addMember()` приведет к сбою в первой строке этого метода, где он пытается сравнить длину списка с максимальным количеством членов, потому что Python не в состоянии сравнивать целое число со строкой.

Разрешение прямого доступа к переменным экземпляра извне объекта может быть рискованным, так как оно обходит меры безопасности, спроектированные для защиты данных объекта.

Строгая интерпретация с помощью геттеров и сеттеров

Строгий подход к инкапсуляции утверждает, что клиентский код *никогда* не получает доступ к переменной экземпляра напрямую. Если класс хочет разрешить клиентскому программному обеспечению доступ к информации, находящейся внутри объекта, стандартным подходом будет включить методы *геттера* и *сеттера* в класс.

Геттер

Метод, который извлекает данные из объекта, экземпляр которого был создан из класса.

Сеттер

Метод, который присваивает данные объекту, экземпляр которого был создан из класса.

Методы геттера и сеттера спроектированы, чтобы позволить пишущим клиентское программное обеспечение получать данные из объекта и устанавливать их в объект, не обладая явными знаниями реализации класса, в частности не обязательно зная или используя имя какой-либо переменной экземпляра. У кода класса `Person` в листинге 8.1 есть переменная экземпляра `self.salary`. В листинге 8.4 мы добавляем геттер и сеттер в класс `Person`, что позволит вызывающему получить и установить зарплату, не предоставляя прямого доступа к переменной экземпляра `self.salary` класса `Person`.

Файл: `PersonGettersSettersAndDirectAccess/Person.py`

```
class Person():
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    # Позволяем вызывающему извлечь зарплату
    ❶ def getSalary(self):
        return self.salary

    # Позволяем вызывающему установить новую зарплату
    ❷ def setSalary(self, salary):
        self.salary = salary
```

Листинг 8.4. Пример класса `Person` с геттером и сеттером

Части имен этих методов `get` ❶ и `set` ❷ необязательны, но используются по соглашению. Обычно вы сопровождаете такие слова описанием данных, к которым представляется доступ, — в данном случае `Salary`. Хотя типично использовать имя переменной экземпляра, к которой предоставляется доступ, это также необязательно.

В листинге 8.5 продемонстрирован некий тестовый код, создающий экземпляры двух объектов `Person`, затем получающий и устанавливающий новые зарплаты с помощью этих методов геттера и сеттера.

Файл: PersonGettersSettersAndDirectAccess/ Main_PersonGetterSetter.py

```
# Пример основной программы Person, использующей геттеры и сеттеры

from Person import *

❶ oPerson1 = Person('Joe Schmoe', 90000)
oPerson2 = Person('Jane Smith', 99000)

# получаем зарплаты, используя геттер, и выводим на экран
❷ print(oPerson1.getSalary())
print(oPerson2.getSalary())

# меняем зарплаты с помощью сеттера
❸ oPerson1.setSalary(100000)
oPerson2.setSalary(111111)

# получаем зарплаты и снова выводим на экран
print(oPerson1.getSalary())
print(oPerson2.getSalary())
```

Листинг 8.5. Пример основного кода, использующего методы геттера и сеттера

Сначала мы создаем два объекта `Person` из класса `Person` ❶. Затем используем методы геттера и сеттера для извлечения ❷ и изменения ❸ зарплат в объектах `Person`.

Геттеры и сеттеры предоставляют формальный способ получать и устанавливать значения в объекте. Они усиливают слои защиты, которые позволяют получить доступ к переменным экземпляра лишь в том случае, если написавший класс хочет это разрешить.

ПРИМЕЧАНИЕ Литература Python использует термины *ассессор* для метода геттера и *мутатор* для метода сеттера. Это лишь другие названия тех же самых вещей. Я буду использовать более общие термины *геттер* и *сеттер*.

Безопасный прямой доступ

Существуют определенные обстоятельства, в которых кажется разумным получить доступ к переменной экземпляра напрямую: когда абсолютно ясно, что означает переменная экземпляра,

нужна незначительная проверка данных или она вообще не требуется и отсутствует вероятность, что имя когда-то будет изменено. Хорошим примером этого является класс `Rect` (прямоугольник) в пакете `pygame`. Прямоугольник в `pygame` определяется с помощью четырех значений: `x`, `y`, ширины и высоты — следующим образом:

```
oRectangle = pygame.Rect(10, 20, 300, 300)
```

После создания этого прямоугольного объекта использование `oRectangle.x`, `oRectangle.y`, `oRectangle.width` и `oRectangle.height` напрямую в качестве переменных кажется приемлемым.

Делаем переменные экземпляра более закрытыми

В Python все переменные экземпляра общедоступны (то есть к ним можно получить доступ с помощью внешнего по отношению к классу кода). А если вы хотите разрешить доступ только к отдельным переменным экземпляра вашего класса, а не ко всем? Некоторые языки ООП позволяют явным образом отмечать определенные переменные экземпляра как `public` или `private`, но в Python нет этих ключевых слов. Тем не менее существует два способа, с помощью которых программисты, разрабатывающие классы в Python, могут указать, что их переменные экземпляра и методы должны быть закрытыми.

Неявно закрытый

Чтобы отметить переменную экземпляра как ту, к которой никогда не должен быть разрешен доступ извне, по соглашению вы начинаете ее имя с одного подчеркивания:

```
self._name  
self._socialSecurityNumber  
self._dontTouchThis
```

Переменные экземпляра с подобными именами обязаны представлять закрытые данные, а клиентское программное обеспечение никогда не должно пытаться получить к ним

доступ напрямую. Код все еще может работать, если осуществляется доступ к переменным, но это не гарантируется.

Аналогичное соглашение используется для имен методов:

```
def _internalMethod(self):  
  
def _dontCallMeFromClientSoftware(self):
```

Повторю: существует лишь соглашение; нет никакого принуждения. Если какое-то клиентское программное обеспечение вызывает метод с именем, начинающимся с подчеркивания, Python разрешит это, но существует высокая вероятность, что это действие приведет к непредвиденным ошибкам.

Более явно закрытый

Python разрешает более явный уровень приватизации. Чтобы запретить клиентскому программному обеспечению напрямую осуществлять доступ к данным, нужно создать имя переменной экземпляра, которое начинается с двух подчеркиваний.

Предположим, вы создаете класс с именем `PrivatePerson` с переменной экземпляра `self.__privateData`, к которой никогда не должен осуществляться доступ извне объекта:

```
# Класс PrivatePerson  
  
class PrivatePerson():  
  
    def __init__(self, name, privateData):  
        self.name = name  
        ❶ self.__privateData = privateData  
  
    def getName(self):  
        return self.name  
  
    def setName(self, name):  
        self.name = name
```

Затем мы можем создать объект `PrivatePerson`, передавая некоторые данные, которые хотим сохранить закрытыми ❶. Попытки получить доступ к переменной экземпляра `__privateData` напрямую из клиентского программного обеспечения подобным образом:

```
usersPrivateData = oPrivatePerson.__privateData
```

сгенерируют ошибку:

```
AttributeError: 'PrivatePerson' object has no attribute  
'__privateData'
```

Аналогичным образом, если вы создаете имя метода, которое начинается с двух подчеркиваний, любая попытка клиентского программного обеспечения вызвать метод сгенерирует ошибку.

Python предоставляет эту возможность, выполняя *декорирование имени*. За кулисами Python изменяет любое имя, начинающееся с двух подчеркиваний, добавляя в его начало подчеркивание и имя класса, так что `__<имя>` становится `__<имякласса>__<имя>`. Например, в классе `PrivatePerson` Python поменяет `self.__privateData` на `self._PrivatePerson__privateData`. Таким образом, если клиент попытается использовать имя `oPrivatePerson.__privateData`, это имя не будет распознано.

Это едва уловимое изменение разработано в качестве защиты от прямого доступа, но вы должны обратить внимание, что оно не гарантирует абсолютную закрытость. Если программист клиента знает, как это работает, он все еще может получить доступ к переменной экземпляра с помощью `<объект>._<имякласса>__<имя>` (или в нашем примере `oPrivatePerson._PrivatePerson__privateData`).

Декораторы и @property

На высоком уровне декоратор является методом, который принимает другой метод в качестве аргумента и расширяет способ работы исходного метода. (Декораторы также могут быть функциями, декорирующими функции или методы, но я сосредоточусь на методах.) Декораторы — это сложная тема, и они в целом выходят за рамки данной книги. Однако существует набор встроенных декораторов, которые представляют компромисс между прямым доступом и использованием геттеров и сеттеров в классе.

Декоратор записывается в виде строки, которая начинается с символа `@`, сопровождаемого именем декоратора, и помещается непосредственно перед оператором `def` метода. Это применяет декоратор к методу, добавляя его поведение:

```
@<decorator>
def <someMethod>(self, <parameters>)
```

Мы будем использовать два встроенных декоратора и применим их к двум методам в классе, чтобы реализовать *свойство*.

Свойство

Атрибут класса, который для клиентского кода представляется как переменная экземпляра, но вместо этого приводит к вызову метода, когда к нему осуществляется доступ.

Свойство позволяет разработчикам класса применять косвенный метод, благодаря которому фокусник отвлекает внимание зрителей: аудитория думает, что видит одно, в то время как за кулисами происходит нечто совсем иное. При написании класса для использования декораторов разработчик пишет методы геттера и сеттера и добавляет отдельный встроенный декоратор к каждому из них. Первым методом является геттер, и он сопровождается встроенным декоратором `@property`. Имя метода определяет имя свойства, которое будет использоваться клиентским кодом. Вторым методом является сеттер, и он сопровождается встроенным декоратором `@<ИМЯ СВОЙСТВА>.setter`. Ниже представлен минимальный образец класса:

```
class Example():
    def __init__(self, startingValue):
        self._x = startingValue

    @property
    def x(self): # декорированный метод геттер
        return self._x

    @x.setter
    def x(self, value): # декорированный метод сеттер
        self._x = value
```

В классе `Example` `x` — это имя свойства. После стандартного метода `__init__()` необычным является то, что у нас есть два метода, оба с одинаковым именем: именем свойства. Первый является геттером, в то время как второй — сеттером. Метод сеттера необязателен, и, если он не будет представлен, свойство будет доступно только для чтения.

Для класса `Example` ниже представлен образец клиентского кода:

```
oExample = Example(10)
print(oExample.x)
oExample.x = 20
```

В этом коде мы создаем экземпляр класса `Example`, вызываем `print()` и выполняем простое присваивание. С точки зрения клиента, код хорошо читается. Когда мы пишем `oExample.x`, это выглядит, как будто мы используем прямой доступ к переменной экземпляра. Однако, когда клиентский код осуществляет доступ к значению свойства объекта (в правой части оператора присваивания или в качестве аргумента в вызове функции или метода), Python преобразует это в вызов метода геттера объекта. Когда в левой части оператора присваивания появляется объект точка свойство, Python вызывает метод сеттера. Методы геттера и сеттера воздействуют на реальную переменную экземпляра `self._x`.

Ниже приведен более реалистичный пример, который должен помочь прояснить это. В листинге 8.6 продемонстрирован класс `Student`, который включает свойство `grade`. Свойство декорировано с помощью методов геттера и сеттера, закрытой переменной экземпляра является `__grade`.

Файл: `PropertyDecorator/Student.py`

```
# Использование свойства для получения (непрямого) доступа к данным
# объекта
```

```
class Student():

    def __init__(self, name, startingGrade=0):
        self.__name = name
        self.grade = startingGrade ❶

    @property ❷
    def grade(self): ❸
        return self.__grade

    @grade.setter ❹
    def grade(self, newGrade): ❺
        try:
            newGrade = int(newGrade)
```



```

except (TypeError, ValueError) as e:
    raise type(e)('New grade: ' + str(newGrade) +
                  ', is an invalid type.')
if (newGrade < 0) or (newGrade > 100):
    raise ValueError('New grade: ' + str(newGrade) +
                     ', must be between 0 and 100.')
self.__grade = newGrade

```

Листинг 8.6. Класс Student с декораторами свойства

У метода `__init__()` есть небольшая хитрость, поэтому давайте сначала изучим другие методы. Обратите внимание, что у нас есть два метода с именем `grade()`. Перед определением первого метода `grade()` мы добавляем декоратор `@property` ❷. Это определяет имя `grade` в качестве свойства любого объекта, созданного из этого класса. Первый метод ❸ — это геттер, который просто возвращает значение текущей оценки, хранящееся в закрытой переменной экземпляра `self.__grade`, но может включать любой код, который может потребоваться для вычисления значения и его возврата.

Второму методу `grade()` предшествует декоратор `@grade.setter` ❹. Этот второй метод ❺ принимает новое значение в качестве параметра, выполняет ряд проверок, чтобы убедиться, что значение допустимо, затем устанавливает новое значение в `self.__grade`.

Метод `__init__()` сначала сохраняет имя студента в переменной экземпляра. Следующая строка ❶ кажется простой, но она немного необычна. Как мы уже видели, мы обычно храним значения параметров в переменных экземпляра. Следовательно, у нас может возникнуть соблазн написать эту строку как:

```
self.__grade = startingGrade
```

Но вместо этого мы сохраняем начальную оценку в свойство `grade`. Поскольку `grade` является свойством, Python преобразует этот оператор присваивания в вызов метода сеттера ❺, преимущество которого заключается в проверке входных данных перед сохранением значения в переменной экземпляра `self.__grade`.

В листинге 8.7 представлен текстовый код, который использует класс Student.

Файл: PropertyDecorator/Main_Property.py

```
# Основной пример свойства Student
❶ oStudent1= Student('Joe Schmoe')
oStudent2= Student ('Jane Smith')

# получаем оценки студентов с помощью свойства "grade" и выводим
# на экран
❷ print(oStudent1.grade)
print(oStudent2.grade)
print()

# Настраиваем новые значения с помощью свойства "grade"
❸ oStudent1.grade = 85
oStudent2.grade = 92

❹ print(oStudent1.grade)
print(oStudent2.grade)
```

Листинг 8.7. Класс Student с декораторами свойства

В тестовом коде мы сначала создаем два объекта Student ❶ и выводим на экран оценки каждого из них ❷. Выглядит, как будто мы напрямую обращаемся к каждому объекту, чтобы получить значения оценок, но, поскольку `grade` является свойством, Python возвращает эти строки в вызовы метода геттера и возвращает значение закрытой переменной экземпляра `self.__grade` для каждого объекта.

Затем мы устанавливаем новые значения оценок для каждого объекта Student ❸. Здесь это выглядит, будто мы устанавливаем значения напрямую в данные объекта, но, повторюсь, поскольку `grade` является свойством, Python возвращает эти строки в вызовы метода сеттера. Этот метод проверяет каждое значение, прежде чем выполнить предписанное. Тестовый код заканчивается выводом новых значений оценок на экран ❹.

Когда возвращаем тестовый код, мы получаем эти выходные данные, как и ожидали:

```
0
0

85
92
```

Используя декораторы `@property` и `@<имя декоратора>`. `setter`, вы получаете лучшее из миров прямого доступа и геттера-и-сеттера. Клиентское программное обеспечение может быть написано таким образом, что *кажется*, будто оно имеет прямой доступ к переменным экземпляра, но, как программирующие класс, ваши декорированные методы получают и управляют фактическими переменными экземпляра, которыми владеет объект, и даже разрешают проверку входных данных. Этот подход поддерживает инкапсуляцию, поскольку клиентский код не получает прямой доступ к переменной экземпляра.

Хотя этот метод используется многими профессиональными разработчиками Python, лично мне он кажется немного сомнительным, потому что, когда я читаю коды других разработчиков, применяющих такой подход, не сразу вижу, используют ли они прямые доступы к переменным экземпляра или задействуют свойства, которые Python преобразовывает в вызовы декорированных методов. Я предпочитаю использовать стандартные методы геттера и сеттера, что и буду делать в остальной части этой книги.

Инкапсуляция в классах `pygame`

Определение инкапсуляции в начале этой главы сосредоточилось на двух областях: сокрытии внутренних деталей и размещении соответствующего кода в одном месте. В `pygame` все классы спроектированы с учетом этих соображений. В качестве примеров возьмем классы `TextButton` и `CustomButton`.

Методы этих двух классов инкапсулируют все функциональные возможности кнопок GUI. Хотя исходный код этих классов доступен, программисту клиента нет необходимости просматривать его, чтобы эффективно их использовать. Также клиентскому коду не нужно пытаться получить доступ к любой из переменных экземпляра: вся функциональность кнопок доступна через вызов методов этих классов. Такой подход соответствует строгой интерпретации инкапсуляции, то есть для клиентского программного обеспечения *единственным* способом получить доступ к данным объекта будет вызов методов этого объекта. Программист клиента может воспринимать эти классы как черные ящики, поскольку нет никаких причин смотреть на то, как они выполняют свои задачи.

ПРИМЕЧАНИЕ Вся отрасль *тестирования черного ящика* была разработана вокруг идеи о том, что программисту предоставляется класс для тестирования, но не разрешается видеть его код. Дается лишь документация интерфейсов, и, используя ее, программист пишет код, который тестирует все интерфейсы во многих разнообразных случаях, чтобы убедиться, что все методы работают как описано. Набор тестов не только гарантирует соответствие кода и документации, но и используется снова каждый раз, когда добавляется код или изменяется класс, чтобы убедиться, что изменения ничего не нарушили.

История из реального мира

Несколько лет назад я принимал участие в проектировании и разработке очень крупного образовательного проекта, который был выстроен в среде под названием *Director* в Macromedia (позднее Adobe) с помощью объектно-ориентированного языка Lingo.

Director был спроектирован для расширения с помощью XTRAs, которые могли добавить функциональные возможности аналогично тому, как плагины добавляются в браузеры. Эти XTRAs были разработаны и проданы рядом сторонних поставщиков. При проектировании мы планировали хранить навигационную и прочую связанную с дисциплинами информацию в базе данных. Я просмотрел все разнообразие XTRAs, которые были доступны, и приобрел конкретную XTRA, которую я буду называть XTRA1.

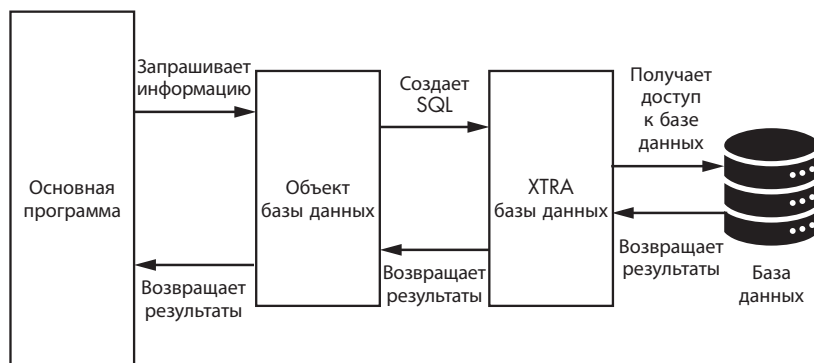


Рис. 8.1. Архитектура доступа к базе данных с помощью объекта и XTRA

Каждая XTRA поставлялась с документацией ее API, которая показывала, как выполнять запросы к базе данных с помощью языка структурированных запросов (SQL). Я решил создать класс Database, который включал все функциональные возможности доступа к базе данных с помощью XTRA1 API. Таким образом весь код, который напрямую взаимодействовал с XTRA, находился в классе Database. На рис. 8.1 показана общая архитектура.

При запуске программа создала один экземпляр класса Database. Основной код был клиентом объекта Database. Каждый раз, когда основной код хотел получить информацию из базы данных, вместо того чтобы самостоятельно форматировать запрос SQL, он вызывал метод объекта Database, предоставляя детали по поводу необходимой ему информации. Методы объекта Database преобразовывали каждый запрос в запрос SQL, выполняемый в XTRA1, чтобы получить данные из базы данных. Таким образом, только код объекта Database знал, как получить доступ к XTRA, используя ее API.

Программа работала хорошо, и клиенты с удовольствием использовали продукт. Но время от времени мы сталкивались с ошибками в данных, которые получали из базы. Я связался с разработчиком XTRA1 и предоставил ему множество воспроизводимых примеров проблем. К сожалению, разработчик не стал этим заниматься.

Из-за отсутствия ответа мы в итоге решили приобрести другую базу данных XTRA, XTRA2, для этой цели. XTRA2 работала аналогичным образом, но были едва уловимые отличия в том, как она была инициализирована, и это потребовало некоторых незначительных изменений в способе построения SQL-запросов.

Поскольку класс Database инкапсулировал все детали взаимодействия с XTRA, мы смогли выполнить все необходимые изменения, чтобы работать с XTRA2, только в классе Database. Мы не меняли ни единой строки в основной программе (клиентском коде).

В данном случае я был как разработчиком класса Database, так и разработчиком клиентского программного обеспечения. Если бы мой клиентский код использовал имена переменных экземпляра в классе, пришлось бы просматривать программу, изменяя каждую строку соответствующего кода. Использование инкапсуляции с классом сэкономило мне бесчисленное количество часов доработки и тестирования.

В продолжении этой истории, хотя XTRA2 работала хорошо, эта компания в итоге вышла из бизнеса, и мне пришлось заново пройти через тот же процесс. И вновь из-за инкапсуляции для работы с XTRA3 был изменен лишь код класса Database.

Абстракция

Абстракция — это еще одно понятие ООП, тесно связанное с инкапсуляцией; многие разработчики считают ее четвертым принципом ООП.

В то время как инкапсуляция связана с реализацией, сокрытием деталей кода и данных, которые составляют класс, *абстракция* связана с представлением клиента о классе. Речь идет о восприятии класса извне.

Абстракция

Управление сложностью с помощью сокрытия ненужных деталей.

По сути, абстракция призывает убедиться, что представление пользователя о системе максимально простое.

Абстракция невероятно широко распространена среди потребительских продуктов. Многие люди используют телевизоры, компьютеры, микроволновые печи, машины и так далее каждый день. Мы привыкаем к распространяемому на нас пользовательскому интерфейсу этих продуктов. С помощью своих элементов управления они предоставляют абстракцию собственной функциональности. Вы жмете на педаль газа, чтобы машина двигалась вперед. В микроволновой печи вы устанавливаете количество времени и нажимаете «Старт», чтобы разогреть какую-то еду. Но лишь некоторые из нас знают, как эти продукты устроены внутри.

Вот пример абстракции из мира компьютерной науки. В программировании *стек* — это механизм для запоминания данных в порядке *последний вошел первый вышел* (LIFO). Подумайте о стопке тарелок, в которой чистые тарелки добавляются наверх, и пользователи могут взять одну сверху, когда им понадобится тарелка. В стеке есть две стандартные операции: *push* добавляет элемент в верхнюю часть стека, а *pop* удаляет самый верхний элемент из стека.

Стек особенно полезен, когда ваша программа осуществляет навигацию, потому что с его помощью можно оставить след

с подсказками, чтобы найти путь назад. Именно так языки программирования отслеживают исполнение функции и вызовы методов в коде: когда вы вызываете функцию или метод, точка возврата проталкивается в стек, а когда функция или метод возвращается, место для возврата обнаруживается с помощью выталкивания самой последней информации из верхней части стека. Таким образом, код может выполнять столько уровней вызовов, сколько вам необходимо, и он всегда разворачивается правильно.

Что касается абстракции, предположим, что клиентская программа требует функциональных возможностей стека, который был бы прост в создании и предоставлял возможность проталкивать и выталкивать информацию. Если бы это было написано как класс, клиентский код создавал бы стек следующим образом:

```
oStack = Stack()
```

Клиент добавлял бы информацию, вызывая метод `push()` следующим образом:

```
oStack.push(<someData>)
```

И он извлекал бы самые последние данные, вызывая метод `pop()` следующим образом:

```
<someVariable> = oStack.pop()
```

Клиенту не нужно знать или беспокоиться о том, как эти методы реализованы или как были сохранены данные. Реализация `Stack` будет полностью обрабатываться методами `Stack`.

Хотя клиент может представлять класс `Stack` как черный ящик, написание такого класса в Python довольно тривиально. В листинге 8.8 показано, как его реализовать.

Файл: `Stack/Stack.py`

```
# Класс Stack

class Stack():
    '''Класс Stack реализует алгоритм последний вошел первый вышел LIFO'''
    def __init__(self, startingStackAsList=None):
        if startingStackAsList is None:
```

```

❶         self.dataList = [ ]
    else:
        self.dataList = startingStackAsList[:] # создаем копию

❷ def push(self, item):
    self.dataList.append(item)

❸ def pop(self):
    if len(self.dataList) == 0:
        raise IndexError
    element = self.dataList.pop()
    return element

❹ def peek(self):
    # извлекаем верхний элемент, не удаляя его
    item = self.dataList[-1]
    return item

❺ def getSize(self):
    nElements = len(self.dataList)
    return nElements

❻ def show(self):
    # отображаем стек в вертикальной ориентации
    print('Stack is:')
    for value in reversed(self.dataList):
        print(' ', value)

```

Листинг 8.8. Стек в качестве класса Python

Класс `Stack` отслеживает все данные, использующие переменную экземпляра списка с именем `self.dataList` ❶. Клиенту не нужно знать этот уровень детализации, `push()` ❷ просто добавляет элемент во внутренний список, используя операцию Python `append()`, в то время как `pop()` ❸ выталкивает последний элемент из внутреннего списка. Поскольку это легко сделать, такая реализация класса `Stack` также реализует три дополнительных метода:

- `peek()` ❹ позволяет вызывающему получать данные из верхней части стека, не удаляя их из него;
- `getSize()` ❺ возвращает число элементов в стеке;
- `show()` ❻ выводит на экран содержимое стека в таком виде, как клиент его себе представляет: данные отображаются вертикально, при этом помещенные последними отображаются сверху. Это может оказаться полезным при отладке клиентского кода, который включает несколько вызовов `push()` и `pop()`.

Приведенный пример невероятно прост, но, когда вы наберетесь больше опыта в написании классов, они будут становиться сложнее. В ходе работы вы можете найти более явные и более эффективные способы написания некоторых методов и, возможно, перепишите их. Поскольку объекты обеспечивают как инкапсуляцию, так и абстракцию, вам как автору класса не стоит стесняться изменять его код и данные, если при этом опубликованные интерфейсы не меняются. Изменения реализации методов не должны оказывать негативного воздействия на клиентское программное обеспечение, а скорее, обязаны позволять вам вносить улучшения, не воздействуя на какой-либо клиентский код. По сути, если вы найдете способы сделать ваш код более эффективным и опубликуете новую версию, может показаться, что клиентский код ускоряется, при этом не требуется вносить в него никаких изменений.

Свойство — великолепный пример абстракции. Как вы видели ранее, со свойствами программист клиента может использовать синтаксис, который проясняет его намерения (получить и установить значение объекта). Реализация в методах, которые в результате вызываются, может быть гораздо сложнее, но она полностью скрыта от клиентского кода.

Выводы

Инкапсуляция — первый из основных принципов объектно-ориентированного программирования, позволяющий классам скрывать их реализации и данные от клиентского кода и гарантирующий, что класс предоставляет все функциональные возможности, которые необходимы клиенту, в одном месте.

Ключевая концепция ООП состоит в том, что объекты владеют своими данными, и именно поэтому я рекомендую вам предоставлять методы геттера и сеттера, если вы хотите, чтобы клиентский код получил доступ к данным, содержащимся в переменной экземпляра. Python разрешает прямой доступ к переменным экземпляра с помощью синтаксиса точки, но я настоятельно рекомендую избегать его по причинам, которые изложил в этой главе.

Существуют соглашения для обозначения переменных экземпляра и методов как закрытых, использующие начальное подчеркивание или двойное подчеркивание, в зависимости

от уровня необходимой вам приватизации. В качестве компромисса Python позволяет использовать декоратор `@property`. Он создает впечатление, будто клиентский код может напрямую получить доступ к переменной экземпляра, в то время как за кулисами Python превращает такие ссылки в вызовы декорированных методов геттера и сеттера в классе.

Пакет `pygwidgets` предоставляет множество прекрасных примеров инкапсуляции. Как программист клиента вы видите класс со стороны и работаете с интерфейсами, которые он предоставляет. Абстракция — управление сложностью за счет сокрытия деталей — помогает проектировщику класса создавать хороший интерфейс, рассматривая его с точки зрения клиента. Однако в Python часто доступен исходный код, поэтому при желании вы можете посмотреть реализацию.

9

ПОЛИМОРФИЗМ



Эта глава посвящена второму основополагающему принципу ООП: *полиморфизму*. Термин происходит из греческого языка: приставка *поли* обозначает «много», а *морфизм* — «форму» или «структуру».

Итак, *полиморфизм* по сути обозначит *много форм*. Я не имею в виду меняющего форму инопланетянина из «Звездного пути» — на самом деле это нечто совсем противоположное. Полиморфизм в ООП заключается не в том, чтобы одна вещь принимала множество форм, а в том, что у нескольких классов могут быть методы с абсолютно идентичными именами. В итоге это даст нам интуитивно понятный способ для осуществления действий с коллекцией объектов вне зависимости от того, из какого класса был получен каждый из них.

Программисты ООП часто используют термин «отправить сообщение», когда говорят о клиентском коде, вызывающем метод объекта. Что должен делать объект, когда он получает сообщение, решает он сам. С помощью полиморфизма мы можем отправлять одно и то же сообщение нескольким объектам, и каждый будет реагировать по-своему, в зависимости от того, для чего он был спроектирован, и от доступных ему данных.

В этой главе я рассмотрю, как эта возможность позволяет создавать пакеты классов, которые легко расширяемы и предсказуемы. Мы также будем использовать полиморфизм с операторами, чтобы заставить одни и те же операторы выполнять различные действия в зависимости от типов данных, с которыми они работают. И наконец, я покажу вам, как использовать функцию `print()`, чтобы получить ценную отладочную информацию от объектов.

Отправляем сообщения объектам реального мира

Давайте рассмотрим полиморфизм в реальном мире, используя пример автомобилей. У них у всех есть педаль газа. Когда водитель нажимает на нее, педаль отправляет машине сообщение «ускориться». В автомобиле может быть двигатель внутреннего сгорания или электрический мотор, или он может быть гибридным. У каждого из этих типов машин есть своя реализация того, что происходит, когда она получает сообщение об ускорении, и каждая из них ведет себя соответствующим образом.

Полиморфизм позволяет облегчить адаптацию новой технологии. Если кто-то собирается разработать атомный автомобиль, пользовательский интерфейс машины останется таким же: водитель все так же будет нажимать на педаль газа, чтобы отправить сообщение, но уже совсем другой механизм будет заставлять атомный автомобиль двигаться быстрее.

В качестве другого примера из реального мира представьте, что вы входите в большую комнату с группой переключателей, управляющих множеством различных источников света. Некоторые из них представляют собой лампы накаливания старого образца, другие — флуоресцентные или же новейшие LED-лампы. Когда переводите все переключатели в положение вверх, вы отправляете сообщение «включить» всем лампам. Базовые механизмы, которые заставляют лампы накаливания, флуоресцентные и LED излучать свет, сильно различаются, но каждый из них достигает поставленной пользователем цели.

Классический пример полиморфизма в программировании

С точки зрения ООП полиморфизм заключается в том, как клиентский код может вызывать метод с точно таким же именем в различных объектах, и каждый объект будет делать все, что необходимо, для реализации значения этого метода для этого объекта.

В качестве классического примера полиморфизма рассмотрим код, который представляет различные виды домашних животных. Предположим, у вас есть коллекция собак, кошек и птиц, и каждая из них понимает некоторый набор базовых команд. Если вы попросите этих животных заговорить (то есть пошлете сообщение «говорить» каждому из них), собака скажет «гав», кошка скажет «мяу», а птица — «чик-чирик». В листинге 9.1 показано, как мы можем реализовать этот код.

Файл: `PetsPolymorphism.py`

```
# Полиморфизм Pets
# Три класса, все с различными методами "говорить"

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

    ❶ def speak(self):
        print(self.name, 'says bark, bark, bark!')

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

    ❷ def speak(self):
        print(self.name, 'says meeeooooow')

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

    ❸ def speak(self):
        print(self.name, 'says tweet')

oDog1 = Dog('Rover')
oDog2 = Dog('Fido')
```

```

oCat1 = Cat('Fluffy')
oCat2 = Cat('Spike')
oBird = Bird('Big Bird')

4 petsList = [oDog1, oDog2, oCat1, oCat2, oBird]

# Отправляем одно и то же сообщение (вызываем один и тот же метод)
# всем домашним животным
for oPet in petsList:
5     oPet.speak()

```

Листинг 9.1. Отправка сообщения «говорить» объектам, для которых были созданы экземпляры из различных классов

У каждого класса есть метод `speak()`, но содержимое у них разное ❶ ❷ ❸. Каждый класс делает все, что необходимо, чтобы выполнить свою версию метода; имя метода одинаковое, но реализации его различаются.

Для упрощения работы помещаем все объекты домашних животных в список ❹. Чтобы заставить их говорить, мы далее перебираем все объекты и отправляем одно и то же сообщение, вызывая метод с точно таким же именем в каждом объекте ❺, не беспокоясь о типе объекта.

Пример, использующий фигуры `pygame`

Далее рассмотрим демонстрацию полиморфизма с использованием `pygame`. В главе 5 я применил `pygame`, чтобы нарисовать примитивные фигуры, такие как прямоугольники, круги, многоугольники, овалы и линии. Здесь мы создадим демонстрационную программу, которая будет случайным образом создавать и рисовать различные фигуры в окне. Затем пользователь может щелкнуть по любой фигуре, и программа сообщит тип и площадь фигуры, по которой щелкнули. Поскольку фигуры создаются произвольным образом, каждый раз при выполнении программы размеры, местоположение, количество и позиции фигур будут разными. На рис. 9.1 показан образец выходных данных демонстрационной программы.

Мы реализуем программу с классом для каждой из трех различных фигур: `Square`, `Circle` и `Triangle`. Ключевой момент заключается в том, что здесь все классы фигур содержат методы с одинаковыми именами: `__init__()`, `draw()`, `getType()`, `getArea()` и `clickedInside()`, которые выполняют

одинаковые задачи. Однако реализация каждого метода различна, потому что каждый класс имеет дело со своей фигурой.

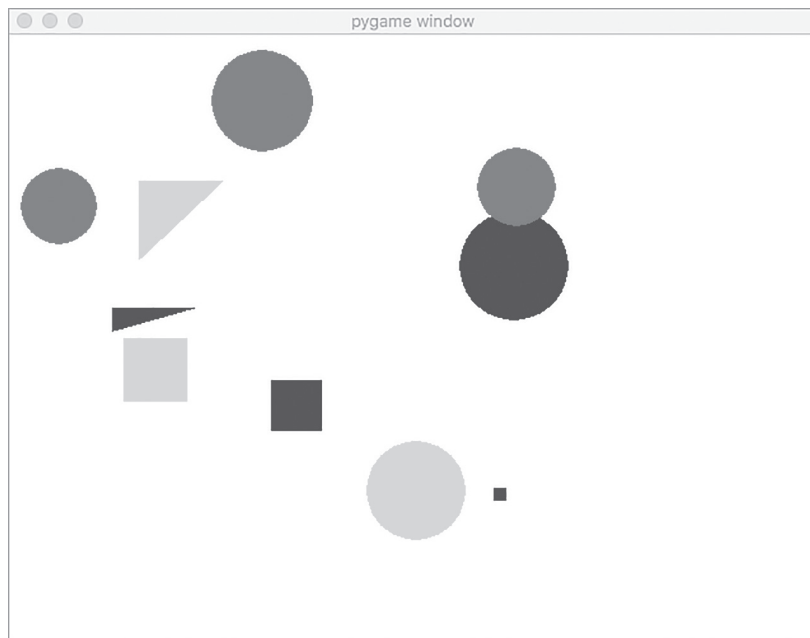


Рис. 9.1. Основанный на `pygame` пример использования полиморфизма для изображения различных фигур

Класс квадратной формы

Я начну с самой простой фигуры. В листинге 9.2 показан код класса `Square`.

Файл: `Shapes/Square.py`

```
# Класс Square

import pygame
import random

# Настраиваем цвета
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

class Square():
    ❶ def __init__(self, window, maxWidth, maxHeight):
        self.window = window
```



```

self.widthAndHeight = random.randrange(10, 100)
self.color = random.choice((RED, GREEN, BLUE))
self.x = random.randrange(1, maxWidth - 100)
self.y = random.randrange(25, maxHeight - 100)
self.rect = pygame.Rect(self.x, self.y, self.widthAndHeight,
                        self.widthAndHeight)
self.shapeType = 'Square'

❷ def clickedInside(self, mousePoint):
    clicked = self.rect.collidepoint(mousePoint)
    return clicked

❸ def getType(self):
    return self.shapeType

❹ def getArea(self):
    theArea = self.widthAndHeight * self.widthAndHeight
    return theArea

❺ def draw(self):
    pygame.draw.rect(self.window, self.color,
                    (self.x, self.y, self.widthAndHeight,
                     self.widthAndHeight))

```

Листинг 9.2. Класс Square

В методе `__init__()` ❶ мы установили несколько переменных экземпляра для использования в методах класса. Это позволит нам оставить код методов очень простым. Поскольку метод `__init__()` сохранил прямоугольник как `Square`, метод `clickedInside()` ❷ просто проверяет, был ли выполнен щелчок мыши внутри этого прямоугольника, возвращая `True` или `False`.

Метод `getType()` ❸ просто возвращает информацию о том, что нажатый элемент является квадратом. Метод `getArea()` ❹ умножает ширину на высоту и возвращает итоговую площадь. Метод `draw()` ❺ использует `draw.rect()` `pygame`, чтобы нарисовать фигуру в произвольно выбранном цвете.

Класс круглой и треугольной формы

Далее давайте посмотрим на код классов `Circle` и `Triangle`. Важная вещь, которую следует отметить, состоит в том, что имена методов этих классов такие же, как и у класса `Square`, но код в них (особенно в `clickedInside()` и `getArea()`) сильно отличается. В листинге 9.3 показан класс `Circle`. В листинге 9.4

показан класс `Triangle`, который создает прямоугольные треугольники произвольного размера, края которых параллельны осям x и y , а прямой угол находится в верхнем левом углу.

Файл: `Shapes/Circle.py`

```
# Класс Circle

import pygame
import random
import math

# Настраиваем цвета
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

class Circle():

    def __init__(self, window, maxWidth, maxHeight):
        self.window = window

        self.color = random.choice((RED, GREEN, BLUE))
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)
        self.radius = random.randrange(10, 50)
        self.centerX = self.x + self.radius
        self.centerY = self.y + self.radius
        self.rect = pygame.Rect(self.x, self.y,
                                self.radius * 2, self.radius * 2)
        self.shapeType = 'Circle'

    ❶ def clickedInside(self, mousePoint):
        distance = math.sqrt(((mousePoint[0] - self.centerX) ** 2) +
                              ((mousePoint[1] - self.centerY) ** 2))
        if distance <= self.radius:
            return True
        else:
            return False

    ❷ def getArea(self):
        theArea = math.pi * (self.radius ** 2)
        return theArea

    def getType(self):
        return self.shapeType
```

```
3 def draw(self):
    pygame.draw.circle(self.window, self.color,
                        (self.centerX, self.centerY),
                        self.radius, 0)
```

Листинг 9.3. Класс Circle

Файл: Shapes/Triangle.py

```
# Класс Triangle
import pygame
import random

# Настраиваем цвета
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

class Triangle():

    def __init__(self, window, maxWidth, maxHeight):
        self.window = window
        self.width = random.randrange(10, 100)
        self.height = random.randrange(10, 100)
        self.triangleSlope = -1 * (self.height / self.width)

        self.color = random.choice((RED, GREEN, BLUE))
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)
        self.rect = pygame.Rect(self.x, self.y,
                                self.width, self.height)
        self.shapeType = 'Triangle'

4 def clickedInside(self, mousePoint):
    inRect = self.rect.collidepoint(mousePoint)
    if not inRect:
        return False

    # выполняем некоторые вычисления, чтобы увидеть,
    # находится ли точка внутри треугольника
    xOffset = mousePoint[0] - self.x
    yOffset = mousePoint[1] - self.y
    if xOffset == 0:
        return True

    # рассчитываем наклон (подъем по ходу движения)
    pointSlopeFromYIntercept = (yOffset - self.height) / xOffset
```

```

        if pointSlopeFromYIntercept < self.triangleSlope:
            return True
        else:
            return False

    def getType(self):
        return self.shapeType

5   def getArea(self):
        theArea = .5 * self.width * self.height
        return theArea

6   def draw(self):
        pygame.draw.polygon(self.window, self.color,
                             ((self.x, self.y + self.height),
                              (self.x, self.y),
                              (self.x + self.width, self.y))

```

Листинг 9.4. Класс Triangle

Чтобы понять работу полиморфизма здесь, давайте рассмотрим код метода `clickedInside()` для каждой фигуры. Метод `clickedInside()` класса `Square` был очень простым: проверить, произошел ли щелчок мыши внутри прямоугольника `Square`. Детали вычислений в классах `Circle` и `Triangle` не особенно важны, но очевидно, что они выполняют другие подсчеты. Метод `clickedInside()` класса `Circle` ❶ лишь информирует о щелчке, если пользователь щелкнул по цветному пикселю фигуры. То есть он обнаруживает щелчок в пределах ограничивающего прямоугольника круга, но тот также обязан случиться внутри радиуса круга, чтобы быть засчитанным. Метод `clickedInside()` класса `Triangle` ❷ должен определять, щелкнул ли пользователь по пикселю внутри закрашенной треугольной части прямоугольника. Метод всех трех классов принимает щелчок мыши в качестве параметра и возвращает либо `True`, либо `False` в качестве результата.

У методов `getArea()` ❸ ❹ и `draw()` ❺ ❻ этих классов имена идентичны методам класса `Square`, но они осуществляют другую внутреннюю работу. Существуют различия в вычислении площади, и они рисуют разные фигуры.

Основная программа, создающая фигуры

В листинге 9.5 показан исходный код основной программы, которая создает список произвольно выбранных объектов фигуры.

Файл: Shapes/Main_ShapesExample.py

```
import pygame
import sys
from pygame.locals import *
from Square import *
from Circle import *
from Triangle import *
import pygamewidgets

# Настраиваем константы
WHITE = (255, 255, 255)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
N_SHAPES = 10

# Настраиваем окно
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT), 0, 32)
clock = pygame.time.Clock()

shapesList = []
shapeClassesTuple = (Square, Circle, Triangle)
for i in range(0, N_SHAPES): ❶
    randomlyChosenClass = random.choice(shapeClassesTuple)
    oShape = randomlyChosenClass (window, WINDOW_WIDTH,
                                  WINDOW_HEIGHT)
    shapesList.append(oShape)

oStatusLine = pygamewidgets.DisplayText(window, (4,4),
                                          'Click on shapes', fontSize=28)

# Основной цикл
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        if event.type == MOUSEBUTTONDOWN: ❷
            # Обратный порядок, чтобы проверить последнюю
            # нарисованную фигуру первой
```

```

for oShape in reversed(shapesList): ❸
    if oShape.clickedInside(event.pos): ❹
        area = oShape.getArea() ❺
        area = str(area)
        theType = oShape.getType()
        newText = 'Clicked on a ' + theType +
                  ' whose area is' + area
        oStatusLine.setValue(newText)
        break # работаем лишь с самой верхней фигурой

# рисуем каждую фигуру
window.fill(WHITE)
for oShape in shapesList:
    oShape.draw()
oStatusLine.draw()

pygame.display.update()
clock.tick(FRAMES_PER_SECOND)

```

Листинг 9.5. Основная программа, которая создает произвольные фигуры из трех классов

Как мы видели в главе 4, каждый раз, когда у нас есть большое количество объектов для управления, типичным подходом будет создать список объектов. Поэтому перед началом основного цикла программа сначала создает список фигур ❶, выбирая произвольным образом из круга, квадрата и треугольника, создавая объект этого типа и добавляя его в список. Используя такой подход, мы можем затем провести итерацию списка и вызывать методы с одинаковым именем в каждом объекте.

Внутри основного цикла программа проверяет наличие события «мышь вниз» ❷, которое происходит, когда пользователь выполняет щелчок. Каждый раз, когда событие обнаруживается, код проводит итерацию `shapesList` ❸ и вызывает метод `clickedInside()` ❹ для каждой фигуры. Благодаря полиморфизму не имеет значения, из какого класса был создан экземпляр объекта. И вновь ключевым моментом является то, что реализация метода `clickedInside()` может быть различна для разных классов.

Когда метод `clickedInside()` возвращает `True` ❺, мы можем вызвать метод `getArea()`, а затем `getType()` этого объекта, не беспокоясь о том, по какому типу объекта был выполнен щелчок.

Ниже представлены выходные данные типичного запуска после нажатия на несколько разных фигур:

```
Clicked on a Circle whose area is 5026.544
Clicked on a Square whose area is 1600
Clicked on a Triangle whose area is 1982.5
Clicked on a Square whose area is 1600
Clicked on a Square whose area is 100
Clicked on a Triangle whose area is 576.0
Clicked on a Circle whose area is 3019.06799
```

Расширяем шаблон

Построение классов с общими названиями методов создает последовательный шаблон, который позволяет вам с легкостью расширять программу. Например, чтобы добавить в нее возможность включать овалы, мы создадим класс `Ellipse`, который реализует методы `getArea()`, `clickedInside()`, `draw()` и `getType()`. (Код метода `clickedInside()` для овала может оказаться математически сложным!)

Как только мы написали код класса `Ellipse`, единственное необходимое нам изменение в коде установки состоит в добавлении `Ellipse` в кортеж классов фигур на выбор. Код в основном цикле, который проверяет наличие щелчков, получает площадь фигуры и так далее, нет необходимости менять.

Этот пример демонстрирует две важные особенности полиморфизма.

- Полиморфизм расширяет понятие абстракции, которое обсуждалось в главе 8, до коллекции классов. Если у нескольких классов одинаковые интерфейсы для их методов, программист клиента может игнорировать реализацию этих методов во всех классах.
- Полиморфизм может упростить клиентское программирование. Если программист клиента уже знаком с интерфейсами, которые предоставляются одним или несколькими классами, тогда вызов методов другого полиморфного класса должен быть таким же простым, как следование шаблону.

pygamewidgets проявляет полиморфизм

Все классы в `pygamewidgets` были спроектированы для использования полиморфизма, и все они реализуют два общих метода. Первый метод `handleEvent()` мы сначала использовали в главе 6, и он принимает объект события в качестве параметра. Каждый класс должен содержать собственный код в этом методе, чтобы обрабатывать любое событие, которое может сгенерировать `pygame`. Каждый раз во время прохождения основного цикла клиентским программам необходимо вызывать метод `handleEvent()` для каждого экземпляра каждого объекта, для которого был создан экземпляр из `pygamewidgets`.

Вторым является метод `draw()`, который рисует изображения в окне. Типичный фрагмент рисования программы, который использует `pygamewidgets`, может выглядеть следующим образом:

```
inputTextA.draw()
inputTextB.draw()
displayTextA.draw()
displayTextB.draw()
restartButton.draw()
checkBoxA.draw()
checkBoxB.draw()
radioCustom1.draw()
radioCustom2.draw()
radioCustom3.draw()
checkBoxC.draw()
radioDefault1.draw()
radioDefault2.draw()
radioDefault3.draw()
statusButton.draw()
```

С точки зрения клиента, каждая строка просто вызывает метод `draw()` и ничего не передает. С внутренней точки зрения код для реализации каждого из этих методов сильно различается. Например, метод `draw()` класса `TextButton` абсолютно не похож на тот же метод класса `InputText`.

Кроме того, все виджеты, которые управляют значением, содержат метод `setValue()` и необязательный метод `getValue()`. Например, чтобы получить вводимый пользователем текст в виджет `InputText`, вы вызываете метод геттера `getValue()`. У виджетов переключателя и флажка также есть метод

`getValue()` для получения их текущих значений. Чтобы поместить новый текст в виджет `DisplayText`, вы вызываете метод сеттера `setValue()`, передавая новый текст. Виджеты переключателя и флажка можно установить с помощью вызова метода `setValue()`.

Полиморфизм позволяет программисту клиента чувствовать себя комфортно при работе с коллекцией классов. Когда клиенты видят шаблон, такой как использование методов с именами `handleEvent()` и `draw()`, им легко предугадать, как использовать новый класс в той же коллекции.

На момент написания этой книги пакет `pygame` не предоставлял ни горизонтальные, ни вертикальные виджеты класса `Slider`, чтобы позволить пользователю с легкостью выбрать из диапазона чисел. Если бы я собирался добавить эти виджеты, они, естественно, содержали бы следующее: метод `handleEvent()`, в котором осуществлялись бы все взаимодействия пользователей; метод `getValue()` и метод `setValue()`, чтобы получать и устанавливать текущее значение для `Slider`; и метод `draw()`.

Полиморфизм для операторов

Python также демонстрирует полиморфизм с операторами. Рассмотрим следующий пример с оператором `+`:

```
value1 = 4
value2 = 5
result = value1 + value2
print(result)
```

который выводит:

```
9
```

Здесь оператор `+` ясно означает «прибавить» в математическом смысле, поскольку обе переменные содержат целые значения. Но давайте теперь рассмотрим вот такой второй пример:

```
value1 = 'Joe'
value2 = 'Schmoe'
result = value1 + value2
print(result)
```

который выводит:

JoeSchmoe

Строка `result = value1 + value2` абсолютно такая же, как и в первом примере, но она выполняет совершенно другую операцию. Со строковыми значениями оператор `+` выполняет конкатенацию строк. Был использован тот же оператор, но выполнено было другое действие.

Этот метод множественных значений оператора широко известен как *перегрузка операторов*. Для некоторых классов возможность перегрузки операторов добавляет невероятно полезные функции и значительно облегчает читаемость клиентского кода.

Магические методы

Python сохраняет для конкретной цели имена методов с необычной формой в виде двух подчеркиваний, имени и двух подчеркиваний:

`__<someName>__()`

Это официально называется *специальными методами*, но программисты Python их обычно называют *магическими методами*. Многие из них уже определены, такие как `__init__()`, который вызывается каждый раз, когда вы создаете экземпляр объекта из класса, но все другие имена такого вида доступны для дальнейшего расширения. Они известны как «магические» методы, потому что Python вызывает их за кулисами каждый раз, когда он обнаруживает оператор, специальный вызов функции или другое особое обстоятельство. Они не предназначены для вызова напрямую клиентским кодом.

ПРИМЕЧАНИЕ Поскольку имена этих магических методов сложно произносить, например, `__init__()` читается как «нижнее подчеркивание нижнее подчеркивание `init` нижнее подчеркивание нижнее подчеркивание», программисты Python часто называют их методами *dunder* (сокращенная версия от «двойное подчеркивание» (double underscore)). Этот метод будет называться *dunder init*.

Продолжая предыдущие примеры, посмотрим, как это работает с оператором `+`. Встроенные типы данных (целое число, число с плавающей точкой, строка, булево выражение и так далее) фактически реализуются в Python как классы. Мы можем увидеть это, протестировав функцию `isinstance()`, которая принимает объект и класс и возвращает значение `True`, если экземпляр объекта был создан из класса, и значение `False` — если нет. Обе эти строки будут сообщать значение `True`:

```
print(isinstance(123, int))
print(isinstance('some string', str))
```

Классы для встроенных типов данных содержат набор магических методов, включая методы для базовых математических операторов. Когда Python обнаруживает оператор `+` с целыми числами, он вызывает магический метод с именем `__add__()` во встроенном классе целого числа, который выполняет сложение. Когда Python видит тот же оператор, используемый со строками, он вызывает метод `__add__()` в классе строки, что выполняет конкатенацию строк.

Это общий механизм, поэтому, когда Python сталкивается с оператором `+` при работе с объектами, экземпляры которых были созданы из вашего класса, он будет вызывать метод `__add__()`, если тот присутствует в вашем классе. Следовательно, как разработчик класса вы можете написать код, чтобы создать новое значение для этого оператора.

Каждый оператор сопоставим с конкретным именем магического метода. Хотя существует множество типов магических методов, давайте начнем с относящихся к операторам сравнения.

Магические методы оператора сравнения

Возьмем наш класс `Square` из листинга 9.2. Вы хотите, чтобы клиентская программа смогла сравнить два объекта `Square`, чтобы увидеть, равны ли они. Именно вы решаете, что значит «равны» при сравнении объектов. Например, вы можете определить это как два объекта одинакового цвета в одном и том же местоположении и одинакового размера. В качестве простого примера мы определим два объекта `Square` как равные, если у них одинакова лишь длина стороны. Это легко реализовать, сравнивая переменные экземпляра `self.heightAndWidth` двух объектов и возвращая булево выражение. Вы можете написать

собственный метод `equals()`, а клиентская программа вызовет его следующим образом:

```
if oSquare1.equals(oSquare2):
```

Это будет хорошо работать. Однако было бы естественным для клиентской программы использовать стандартный оператор сравнения `==`:

```
if oSquare1 == oSquare2:
```

Написанный таким образом, Python преобразует оператор `==` в вызов магического метода первого объекта. В этом случае Python попытается вызвать магический метод с именем `__eq__()` в классе `Square`. В табл. 9.1 показаны магические методы для всех операторов сравнения.

Таблица 9.1. Символы, значения и имена магических методов операторов сравнения

Символ	Значение	Имя магического метода
<code>==</code>	Равно	<code>__eq__()</code>
<code>!=</code>	Не равно	<code>__ne__()</code>
<code><</code>	Меньше	<code>__lt__()</code>
<code>></code>	Больше	<code>__gt__()</code>
<code><=</code>	Меньше или равно	<code>__le__()</code>
<code>>=</code>	Больше или равно	<code>__ge__()</code>

Чтобы разрешить оператору сравнения `==` проверить наличие равенства между двумя объектами `Square`, вам необходимо будет написать метод в классе `Square`, подобный этому:

```
def __eq__(self, oOtherSquare):
    if not isinstance(oOtherSquare, Square):
        raise TypeError('Second object was not a Square')
    if self.heightAndWidth == oOtherSquare.heightAndWidth:
        return True # сопоставимо
    else:
        return False # не сопоставимо
```

Когда Python обнаруживает сравнение `==`, где первым объектом является `Square`, он вызывает этот метод в классе `Square`.

Поскольку Python — это свободно типизированный язык (он не требует от вас определения типов переменных), тип данных второго параметра может быть любой. Тем не менее, чтобы сравнение работало правильно, второй параметр также должен быть объектом `Square`. Мы осуществим проверку, используя функцию `isinstance()`, которая работает с определенными программистом классами таким же образом, как она работает со встроенными классами. Если второй объект не `Square`, мы вызываем исключение.

Затем мы сравниваем `heightAndWidth` текущего объекта (`self`) с `heightAndWidth` второго объекта (`oOtherSquare`). В этом случае использование прямого доступа к переменным экземпляра двух объектов абсолютно допустимо, потому что оба объекта одного типа и, следовательно, они должны содержать одинаковые переменные экземпляра.

Магические методы в классе `Rectangle`

Для расширения мы создадим программу, которая рисует несколько прямоугольных форм с помощью класса `Rectangle`. Пользователь сможет щелкнуть по любым двум прямоугольникам, а программа сообщит, одинакова ли площадь этих прямоугольников или площадь первого больше или меньше площади второго прямоугольника. Мы используем операторы `==`, `<` и `>` и будем ожидать, что результаты для каждого сравнения окажутся представлены в виде булева выражения `True` или `False`. В листинге 9.6 содержится код класса `Rectangle`, который реализует магические методы для этих операторов.

Файл: `MagicMethods/Rectangle/Rectangle.py`

```
# Класс Rectangle
import pygame
import random

# Настраиваем цвета
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

class Rectangle():

    def __init__(self, window):
        self.window = window
```

```

        self.width = random.choice((20, 30, 40))
        self.height = random.choice((20, 30, 40))
        self.color = random.choice((RED, GREEN, BLUE))
        self.x = random.randrange(0, 400)
        self.y = random.randrange(0, 400)
        self.rect = pygame.Rect(self.x, self.y, self.width, self.height)
        self.area = self.width * self.height

def clickedInside(self, mousePoint):
    clicked = self.rect.collidepoint(mousePoint)
    return clicked

# Вызываем магический метод, когда сравниваются
# два объекта Rectangle с помощью оператора ==
def __eq__(self, oOtherRectangle): ❶
    if not isinstance(oOtherRectangle, Rectangle):
        raise TypeError('Second object was not a Rectangle')
    if self.area == oOtherRectangle.area:
        return True
    else:
        return False

# Вызываем магический метод, когда сравниваются
# два объекта Rectangle с помощью оператора <
def __lt__(self, oOtherRectangle): ❷
    if not isinstance(oOtherRectangle, Rectangle):
        raise TypeError('Second object was not a Rectangle')
    if self.area < oOtherRectangle.area:
        return True
    else:
        return False

# Вызываем магический метод, когда сравниваются
# два объекта Rectangle с помощью оператора >
def __gt__(self, oOtherRectangle): ❸
    if not isinstance(oOtherRectangle, Rectangle):
        raise TypeError('Second object was not a Rectangle')
    if self.area > oOtherRectangle.area:
        return True
    else:
        return False

def getArea(self):
    return self.area

def draw(self):
    pygame.draw.rect(self.window, self.color, (self.x, self.y,
                                                self.width, self.height))

```

Листинг 9.6. Класс Rectangle

Методы `__eq__()` ❶, `__lt__()` ❷ и `__gt__()` ❸ позволяют клиентскому коду использовать стандартные операторы сравнения между объектами `Rectangle`. Чтобы сравнить два прямоугольника для выявления равенства, вы напишете:

```
if oRectangle1 == oRectangle2:
```

Когда эта строка выполняется, вызывается метод `__eq__()` первого объекта, а второй объект передается в качестве второго параметра. Функция возвращает либо `True`, либо `False`. Аналогичным образом для сравнения «меньше чем» вы напишете строку, подобную этой:

```
if oRectangle1 < oRectangle2:
```

Затем метод `__lt__()` проверит, будет ли площадь первого прямоугольника меньше площади второго прямоугольника. Если клиентский код использует оператор `>` для сравнения двух прямоугольников, то вызывается метод `__gt__()`.

Использование магических методов основной программой

В листинге 9.7 показан код основной программы, который тестирует магические методы.

Файл: MagicMethods/Rectangle/Main_RectangleExample.py

```
import pygame
import sys
from pygame.locals import *
from Rectangle import *

# Настраиваем константы
WHITE = (255, 255, 255)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
N_RECTANGLES = 10
FIRST_RECTANGLE = 'first'
SECOND_RECTANGLE = 'second'

# Настраиваем окно
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT), 0, 32)
clock = pygame.time.Clock()
```

```

rectanglesList = []
for i in range(0, N_RECTANGLES):
    oRectangle = Rectangle(window)
    rectanglesList.append(oRectangle)

whichRectangle = FIRST_RECTANGLE

# Основной цикл
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        if event.type == MOUSEBUTTONDOWN:
            for oRectangle in rectanglesList:
                if oRectangle.clickedInside(event.pos):
                    print('Clicked on', whichRectangle, 'rectangle.')

                    if whichRectangle == FIRST_RECTANGLE:
                        oFirstRectangle = oRectangle ❶
                        whichRectangle = SECOND_RECTANGLE

                    elif whichRectangle == SECOND_RECTANGLE:
                        oSecondRectangle2 = oRectangle ❷
                        # Пользователь выбрал 2 прямоугольника,
                        # сравниваем их
                        if oFirstRectangle == oSecondRectangle: ❸
                            print('Rectangles are the same size.')
                        elif oFirstRectangle < oSecondRectangle: ❹
                            print('First rectangle is smaller than
                                second rectangle.')
                        else: # должен быть больше ❺
                            print('First rectangle is larger than
                                second rectangle.')
                        whichRectangle = FIRST_RECTANGLE

    # Очищаем окно и рисуем все прямоугольники
    window.fill(WHITE)
    for oRectangle in rectanglesList: ❻
        oRectangle.draw()

    pygame.display.update()

    clock.tick(FRAMES_PER_SECOND)

```

Листинг 9.7. Основная программа, которая рисует и затем сравнивает объекты

Пользователь программы щелкает по паре прямоугольников, чтобы сравнить их размеры. Мы сохраняем выбранные прямоугольники в двух переменных ❶ ❷.

Проверяем наличие равенства с помощью оператора `==` ❸, который решает вызвать метод `__eq__()` класса `Rectangle`. Если размер прямоугольников одинаковый, мы выводим соответствующее сообщение. Если они не равны, мы проверяем, будет ли первый прямоугольник меньше второго, с помощью оператора `<` ❹, что приводит к вызову метода `__lt__()`. Если это сравнение также не дает `True`, мы выводим сообщение, что первый прямоугольник больше, чем второй ❺. В этой программе нам не понадобилось использовать оператор `>`; однако, поскольку другой клиентский код может реализовать сравнения размеров по-другому, мы включили для полноты картины метод `__gt__()`.

И наконец, рисуем все прямоугольники в нашем списке ❻.

Поскольку мы включили методы `__eq__()`, `__lt__()` и `__gt__()` в класс `Rectangle`, то смогли использовать стандартные операторы сравнения интуитивно понятным и легко читаемым способом.

Ниже приведены выходные данные щелчков по нескольким различным прямоугольникам:

```
Clicked on first rectangle.
Clicked on second rectangle.
Rectangles are the same size.
Clicked on first rectangle.
Clicked on second rectangle.
First rectangle is smaller than second rectangle.
Clicked on first rectangle.
Clicked on second rectangle.
First rectangle is larger than second rectangle.
```

Магические методы математических операторов

Вы можете написать дополнительные магические методы, чтобы определить, что происходит, когда клиентский код использует другие арифметические операторы между объектами, экземпляры которых были созданы из вашего класса.

В табл. 9.2 показаны методы, вызываемые для базовых арифметических операторов.

Таблица 9.2. Символы, значения и имена магических методов математических операторов

Символ	Значение	Имя магического метода
+	Сложение	<code>__add__()</code>
−	Вычитание	<code>__sub__()</code>
*	Умножение	<code>__mul__()</code>
/	Деление (результат с плавающей точкой)	<code>__truediv__()</code>
//	Деление целого	<code>__floordiv__()</code>
%	Остаток	<code>__mod__()</code>
Abs	Абсолютное значение	<code>__abs__()</code>

Например, чтобы обработать оператор `+`, вам необходимо будет реализовать метод в классе следующим образом:

```
def __add__(self, oOther):
    # Определение, что происходит при добавлении двух
    # из числа объектов
```

Полный список магических и dunder-методов можно найти в официальной документации по адресу <https://docs.python.org/3/reference/datamodel.html>.

Векторный пример

В математике *вектор* — это упорядоченная пара значений *x* и *y*, которая часто представляется на графике в виде направленного отрезка. В этом разделе мы создадим класс, который использует магические методы математических операторов для векторов. Существует ряд математических операций, которые можно выполнять с векторами. На рис. 9.2 показан пример сложения двух векторов.

Сложение двух векторов приводит к появлению нового вектора, чье значение *x* равно сумме значений *x* двух суммированных векторов и чье значение *y* представляет сумму значений *y* двух суммированных векторов. На рис. 9.2 мы суммируем вектор (3, 2) и вектор (1, 3), чтобы получить вектор (4, 5).

Два вектора считаются равными, если их значения *x* и *y* одинаковы. Размер вектора вычисляется как гипотенуза прямоугольного треугольника, одна сторона которого представляет

собой длину x , а вторая сторона — длину y . Мы можем применить теорему Пифагора, чтобы вычислить длину, и использовать длины для сравнения размеров двух векторов.

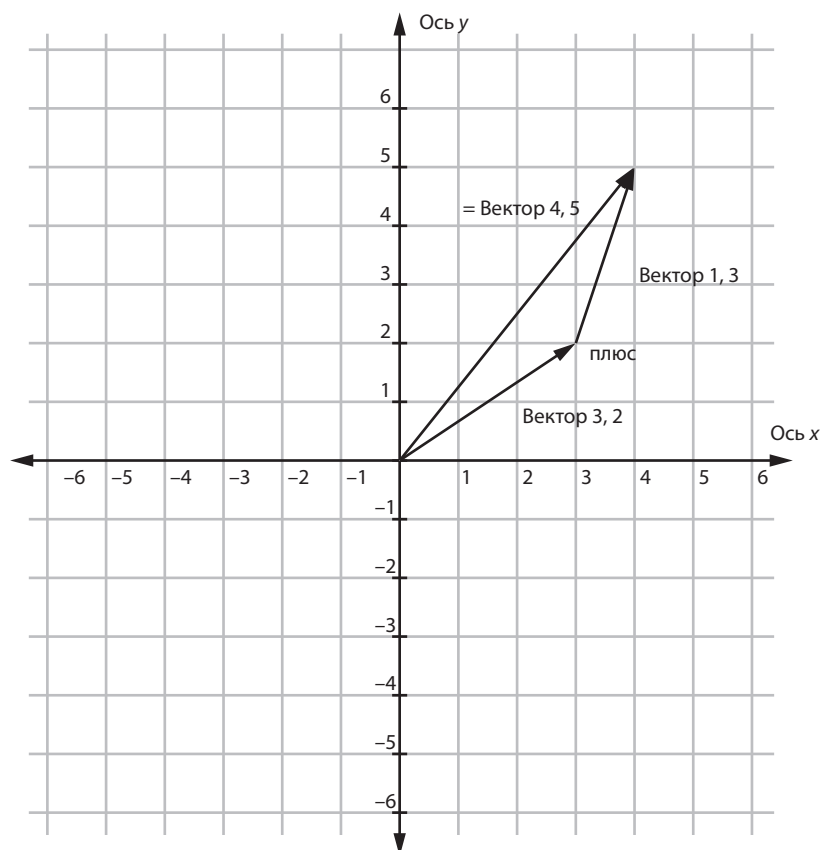


Рис. 9.2. Сложение векторов в декартовой системе координат

В листинге 9.8 класс `Vector` демонстрирует соответствующие магические методы для выполнения вычислений и сравнений между двумя объектами `Vector`. (У каждого из этих методов есть дополнительный код, использующий вызов `isinstance()`, чтобы удостовериться, что второй объект является `Vector`. Эти проверки включены в загружаемый файл, но я опустил их здесь, чтобы сэкономить место.)

Файл: MagicMethods/Vectors/Vector.py

```
# Класс Vector

import math

class Vector():
    '''Класс Vector представляет два значения в качестве вектора,
    разрешая многие математические вычисления'''
    def __init__(self, x, y):
        self.x = x
        self.y = y
    ❶ def __add__(self, oOther): # вызываем оператор +
        return Vector(self.x + oOther.x, self.y + oOther.y)

    def __sub__(self, oOther): # вызываем оператор -
        return Vector(self.x - oOther.x, self.y - oOther.y)

    ❷ def __mul__(self, oOther): # вызываем оператор *
        # Специальный код допускает умножение на вектор или скаляр
        if isinstance(oOther, Vector): # умножаем два вектора
            return Vector((self.x * oOther.x), (self.y * oOther.y))
        elif isinstance(oOther, (int, float)): # умножаем на скаляр
            return Vector((self.x * oOther), (self.y * oOther))
        else:
            raise TypeError('Second value must be a vector or scalar')

    def __abs__(self):
        return math.sqrt((self.x ** 2) + (self.y ** 2))

    def __eq__(self, oOther): # вызываем оператор ==
        return (self.x == oOther.x) and (self.y == oOther.y)

    def __ne__(self, oOther): # вызываем оператор !=
        return not (self == oOther) # вызываем метод __eq__

    def __lt__(self, oOther): # вызываем оператор <
        if abs(self) < abs(oOther): # вызываем метод __abs__
            return True
        else:
            return False

    def __gt__(self, oOther): # вызываем оператор >
        if abs(self) > abs(oOther): # вызываем метод __abs__
            return True
        else:
            return False
```

Листинг 9.8. Класс Vector, который реализует ряд магических методов

Этот класс реализует арифметические операторы и операторы сравнения в качестве магических методов. Клиентский код будет использовать стандартные символы для вычислений и сравнения между двумя объектами `Vector`. Например, сложение векторов на рис. 9.2 можно обработать следующим образом:

```
oVector1 = Vector(3, 2)
oVector2 = Vector(1, 3)
oNewVector = oVector1 + oVector2 # используем оператор +
                                   # для сложения векторов
```

Когда выполняется третья строка, вызывается метод `__add__()` ❶ для сложения двух объектов `Vector`, что приводит к созданию нового объекта `Vector`. В методе `__mul__()` ❷ проводится специальная проверка, которая позволяет оператору `*` либо умножить два `Vector`, либо умножить один `Vector` на скалярную величину (в зависимости от типа второго значения).

Создаем строковое представление значений в объекте

Стандартный подход к отладке заключается в добавлении вызовов `print()` для вывода значений переменных в определенных точках вашей программы:

```
print('My variable is', myVariable)
```

Однако, если вы попытаетесь использовать `print()` для помощи в отладке содержимого объекта, результаты не будут особо полезными. Например, здесь мы создаем объект `Vector` и выводим его на экран:

```
oVector = Vector(3, 4)
print('My vector is', oVector)
```

Вот что выводится:

```
<Vector object at 0x10361b518>
```

Это говорит нам, что есть объект, экземпляр которого был создан из класса `Vector`, и показывает адрес памяти этого объекта. Однако в большинстве случаев мы на самом деле хотим знать значения переменных экземпляра в объекте на данный

момент. К счастью, можно использовать для этого магические методы.

Существует два магических метода, которые полезны для получения информации (в виде строк) из объекта.

- Метод `__str__()` используется для создания строкового представления объекта, которое может легко прочитать человек. Если клиентский код вызывает встроенную функцию `str()` и передает объект, Python вызовет магический метод `__str__()`, если он есть в этом классе.
- Метод `__repr__()` используется для создания однозначных, машиночитаемых строковых представлений объекта. Если клиентский код вызывает встроенную функцию `repr()` и передает объект, Python попытается вызвать магический метод `__repr__()` в этом классе, если он есть.

Я покажу метод `__str__()`, так как он более широко используется для простой отладки. Когда вы вызываете функцию `print()`, Python вызывает встроенную функцию `str()`, чтобы преобразовать каждый аргумент в строку. Для аргумента, у которого нет метода `__str__()`, эта функция форматирует строку, которая содержит тип объекта, слова «object at» и адрес памяти, затем возвращает итоговую строку. Вот почему мы видим выше выходные данные, содержащие адрес памяти.

Вместо этого вы можете написать собственную версию `__str__()` и заставить ее воспроизводить любую строку, которая необходима для отладки кода вашего класса. Общий подход заключается в создании строки, содержащей значения всех переменных экземпляра, которые вы хотите видеть, и возвращающей эту строку для вывода на экран. Например, мы можем добавить следующий метод в класс `Vector` из листинга 9.8, чтобы получить информацию о любом объекте `Vector`:

```
class Vector():
    --- пропущены все предыдущие методы ---
    def __str__(self):
        return 'This vector has the value (' + str(self.x) + ', ' +
                str(self.y) + ')
```

Если создаете экземпляр `Vector`, вы затем можете вызвать функцию `print()` и передать объект `Vector`:

```
oVector = Vector(10, 7)
print(oVector)
```

Вместо того чтобы просто выводить на экран адрес памяти объекта `Vector`, вы получите хорошо отформатированный отчет о значениях двух переменных экземпляра, содержащихся в объекте:

```
This vector has the value (10, 7)
```

Основной код в листинге 9.9 создает несколько объектов `Vector`, проводит некоторые векторные вычисления и выводит результаты некоторых вычислений `Vector`.

Файл: `MagicMethods/Vectors/Vector.py`

```
# Тестовый код Vector

from Vector import *

v1 = Vector(3, 4)
v2 = Vector(2, 2)
v3 = Vector(3, 4)

# Эти строки выводят булево выражение или числовые значения
print(v1 == v2)
print(v1 == v3)
print(v1 < v2)
print(v1 > v2)
print(abs(v1))
print(abs(v2))
print()

# Эти строки выводят объекты Vector (вызов метода __str__())
print('Vector 1:', v1)
print('Vector 2:', v2)
print('Vector 1 + Vector 2:', v1 + v2)
print('Vector 1 - Vector 2:', v1 - v2)
print('Vector 1 times Vector 2:', v1 * v2)
print('Vector 2 times 5:', v1 * 5)
```

Листинг 9.9. Образец основного кода, который создает и сравнивает `Vectors`, производит математические действия и выводит `Vectors` на экран

Код генерирует следующие выходные данные:

```
False
True
False
True
5.0
2.8284271247461903
Vector 1: This vector has the value (3, 4)
Vector 2: This vector has the value (2, 2)
Vector 1 + Vector 2: This vector has the value (5, 6)
Vector 1 - Vector 2: This vector has the value (1, 2)
Vector 1 times Vector 2: This vector has the value (6, 8)
Vector 2 times 5: This vector has the value (15, 20)
```

Первый набор вызовов `print()` выводит булево выражение и числовые значения, которые являются результатом вызова магических методов математического оператора и оператора сравнения. Во втором наборе мы выводим два объекта `Vector`, затем вычисляем и выводим на экран некоторые новые `Vectors`. Внутренне функция `print()` сначала вызывает функцию Python `str()` для каждого выводимого на экран элемента; это приводит к вызову магического метода `Vector.__str__()`, который создает отформатированную строку с соответствующей информацией.

Класс `Fraction` с магическими методами

Давайте объединим некоторые из этих магических методов в более сложном примере. В листинге 9.10 показан код класса `Fraction`. Каждый объект `Fraction` — дробь, которая состоит из числителя (верхняя часть) и знаменателя (нижняя часть). Класс отслеживает дроби, сохраняя отдельные части в переменных экземпляра наряду с примерным десятичным значением дроби. Методы позволяют вызывающему получить сокращенное значение дроби, вывести на экран дробь вместе с ее значением с плавающей точкой, сравнить две дроби для выявления равенства и добавить два объекта `Fraction`.

Файл: MagicMethods/Fraction.py

```
# Класс Fraction

import math

class Fraction():
    def __init__(self, numerator, denominator): ❶
        if not isinstance(numerator, int):
            raise TypeError('Numerator', numerator,
                            'must be an integer')
        if not isinstance(denominator, int):
            raise TypeError('Denominator', denominator,
                            'must be an integer')
        self.numerator = numerator
        self.denominator = denominator

        # Используем математический пакет, чтобы найти наибольший
        # общий делитель
        greatestCommonDivisor = math.gcd(self.numerator,
                                          self.denominator)

        if greatestCommonDivisor > 1:
            self.numerator = self.numerator // greatestCommonDivisor
            self.denominator = self.denominator // greatestCommonDivisor
        self.value = self.numerator / self.denominator

        # Нормализуем знак числителя и знаменателя
        self.numerator = int(math.copysign(1.0, self.value)) *
            abs(self.numerator)
        self.denominator = abs(self.denominator)

    def getValue(self): ❷
        return self.value

    def __str__(self): ❸
        '''Создаем строковое представление дроби'''
        output = ' Fraction: ' + str(self.numerator) + '/' + \
            str(self.denominator) + '\n' + \
            ' Value: ' + str(self.value) + '\n'
        return output

    def __add__(self, oOtherFraction): ❹
        '''Складываем два объекта Fraction'''
        if not isinstance(oOtherFraction, Fraction):
            raise TypeError('Second value in attempt to add is not
                            a Fraction')

        # Используем математический пакет, чтобы найти наименьшее
        # общее кратное
```

```

newDenominator = math.lcm(self.denominator,
                           oOtherFraction.denominator)

multiplicationFactor = newDenominator // self.denominator
equivalentNumerator = self.numerator * multiplicationFactor

otherMultiplicationFactor = newDenominator //
                           oOtherFraction.denominator
oOtherFractionEquivalentNumerator =
    oOtherFraction.numerator * otherMultiplicationFactor

newNumerator = equivalentNumerator +
               oOtherFractionEquivalentNumerator

oAddedFraction = Fraction(newNumerator, newDenominator)
return oAddedFraction

def __eq__(self, oOtherFraction): ❸
    '''Проверяем на равенство'''
    if not isinstance(oOtherFraction, Fraction):
        return False # не сравнимо с дробью
    if (self.numerator == oOtherFraction.numerator) and \
        (self.denominator == oOtherFraction.denominator):
        return True
    else:
        return False

```

Листинг 9.10. Класс `Fraction`, который реализует некоторые магические методы

Когда создаете объект `Fraction`, вы передаете числитель и знаменатель ❶, и метод `__init__()` сразу же вычисляет сокращенную дробь и ее значение с плавающей точкой. В любой момент клиентский код может вызвать `getValue()`, чтобы извлечь это значение ❷. Клиентский код может также вызвать `print()`, чтобы вывести объект на экран, и Python вызовет метод `__str__()`, чтобы отформатировать строку для вывода на экран ❸.

Клиент может добавить два различных объекта `Fraction` вместе с помощью оператора `+`. Когда это происходит, вызывается метод `__add__()` ❹. Он использует метод `math.lcd()` (наименьший общий знаменатель), чтобы убедиться, что у итогового объекта `Fraction` наименьший общий знаменатель.

И наконец, клиентский код может использовать оператор `==`, чтобы проверить, равны ли два объекта `Fraction`. Когда вы действуете этот оператор, вызывается метод `__eq__()` ❺,

который проверяет значения двух Fraction и возвращает True или False.

Ниже представлен код, который создает экземпляры объектов Fraction и тестирует различные магические методы:

```
# Тестовый код

oFraction1 = Fraction(1, 3) # создаем объект Fraction
oFraction2 = Fraction(2, 5)
print('Fraction1\n', oFraction1) # выводим на экран объект...
                                # вызываем __str__
print('Fraction2\n', oFraction2)

oSumFraction = oFraction1 + oFraction2 # вызываем __add__
print('Sum is\n', oSumFraction)

print('Are fractions 1 and 2 equal?', (oFraction1 == oFraction2))
# ожидаем False
print()

oFraction3 = Fraction(-20, 80)
oFraction4 = Fraction(4, -16)
print('Fraction3\n', oFraction3)
print('Fraction4\n', oFraction4)
print('Are fractions 3 and 4 equal?', (oFraction3 == oFraction4))
# ожидаем True
print()

oFraction5 = Fraction(5, 2)
oFraction6 = Fraction(500, 200)
print('Sum of 5/2 and 500/200\n', oFraction5 + oFraction6)
```

При запуске этот код выдает:

```
Fraction1
  Fraction: 1/3
  Value: 0.3333333333333333

Fraction2
  Fraction: 2/5
  Value: 0.4

Sum is
  Fraction: 11/15
  Value: 0.7333333333333333

Are fractions 1 and 2 equal? False
```

```
Fraction3
  Fraction: -1/4
  Value: -0.25

Fraction4
  Fraction: -1/4
  Value: -0.25

Are fractions 3 and 4 equal? True

Sum of 5/2 and 500/200
  Fraction: 5/1
  Value: 5.0
```

Выводы

Эта глава была посвящена ключевому понятию ООП — полиморфизму. Проще говоря, полиморфизм — это возможность нескольких классов реализовывать методы с одинаковыми именами. Каждый класс содержит конкретный код, который делает все необходимое, чтобы были созданы экземпляры объектов из этого класса. В качестве демонстрационной программы я показал вам, как создать некоторое количество различных классов фигур, каждый из которых содержит методы `__init__()`, `getArea()`, `clickedInside()` и `draw()`. Код этих методов был разным, потому что специфичен для типа фигуры.

Как вы видели, существует два ключевых преимущества при использовании полиморфизма. Во-первых, он расширяет понятие абстракции до коллекции классов, давая программисту клиента возможность игнорировать реализацию. Во-вторых, он позволяет создать систему классов, которые работают аналогичным образом, что делает систему предсказуемой для программистов клиента.

Я также рассмотрел идею полиморфизма в операторах, объяснив, как один и тот же оператор может выполнять различные операции с различными типами данных. Я продемонстрировал, как для достижения этого используются магические методы Python и как вы можете создать методы, чтобы реализовать эти операторы в собственных классах. Чтобы

продемонстрировать магические методы арифметических операторов и операторов сравнения, я показал класс `Vector` и класс `Fraction`, а также продемонстрировал, как вы можете использовать метод `__str__()` для помощи в отладке содержимого объекта.

10

НАСЛЕДОВАНИЕ



Третий принцип ООП — *наследование*, которое представляет собой механизм для получения нового класса из существующего. Вместо того чтобы начинать с нуля и потенциально дублировать код, наследование позволяет программисту

писать код для нового класса, который расширяет возможности или отличает его от существующего класса.

Давайте начнем с примера из реального мира, который демонстрирует, что такое наследование. Вы посещаете кулинарную школу. Один из уроков включает исчерпывающую демонстрацию процесса приготовления гамбургеров. Вы узнаете все, что можно узнать, о различных кусках мяса, измельчении мяса, лучших типах булочек, лучших листьях салата, помидорах и приправах — практически обо всем, что только можно было себе представить. Вы также можете узнать о лучшем способе приготовления гамбургера: насколько долго его готовить, когда и как часто переворачивать и так далее.

Следующий урок в учебном плане касается чизбургеров. Преподаватель мог бы начать с нуля и пройти весь материал по приготовлению гамбургеров снова. Но вместо этого он предполагает, что вы усвоили знания, полученные на предыдущих

уроках, и уже знаете все, что необходимо знать для создания великолепного гамбургера. Следовательно, этот урок сосредоточится на типах сыра, которые необходимо использовать, когда этот сыр добавлять, в каком количестве и так далее.

Смысл истории в том, что нет необходимости изобретать велосипед; вместо этого вы можете просто добавить новое к тому, что уже знаете.

Наследование в объектно-ориентированном программировании

Наследование в ООП представляет собой возможность создавать класс, который выстраивается на основании существующего класса (*расширяет* его). При создании больших программ вы часто будете использовать классы, которые предоставляют очень полезные общие возможности. Иногда вам захочется создать класс, похожий на уже существующий, но делающий некоторые вещи немного по-другому. Наследование позволяет вам именно это: создать класс, который включает все методы и переменные экземпляра существующего класса, но добавив новые и отличающиеся функциональные возможности.

Наследование — невероятно мощная концепция. Когда классы спроектированы правильно, использование наследования *кажется* простым. Однако возможность проектировать классы, чтобы использовать их понятным образом, — это навык, которым сложно овладеть. Как средство реализации наследование требует много практики, чтобы использовать его правильно и эффективно.

Используя наследование, мы поговорим о взаимоотношениях между двумя классами, которые обычно называются *базовый класс* и *подкласс*.

Базовый класс

Класс, от которого наследуют; он служит отправной точкой для подкласса.

Подкласс

Класс, который наследует; он улучшает базовый класс.

Хотя это наиболее распространенные термины для описания двух классов в Python, вы также можете услышать другие их названия, такие как:

- *суперкласс* и *подкласс*;
- *базовый класс* и *производный класс*;
- *родительский класс* и *дочерний класс*.

На рис. 10.1 показана стандартная диаграмма, демонстрирующая эти взаимоотношения.



Рис. 10.1. Подкласс наследует от базового класса

Подкласс наследует все методы и переменные экземпляра, определенные в базовом классе.

На рис. 10.2 представлен другой, возможно, более точный способ восприятия взаимоотношений между этими двумя классами.

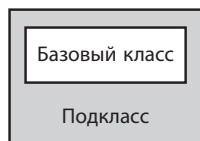


Рис. 10.2. Базовый класс включен в подкласс

Как конструктор вы можете думать о базовом классе как о включенном в подкласс. Таким образом, базовый класс фактически становится частью более крупного подкласса. Как клиент подкласса вы думаете о подклассе как об отдельном структурном элементе, и вам не нужно знать, что существует еще базовый класс.

При обсуждении наследования мы часто говорим, что существуют отношения *is a* между подклассом и базовым классом. Например, студент — это человек, апельсин — это фрукт, машина — это транспорт и так далее. Подкласс — это специализированная версия базового класса, которая наследует все свойства

и поведение базового класса, но также предоставляет дополнительные детали и функциональные возможности.

И самое важное — подкласс расширяет базовый класс одним или обоими указанными ниже способами (которые будут вскоре объяснены).

- Подкласс *переопределяет* метод, который определен в базовом классе. То есть подкласс может предоставить метод с тем же именем, что и в базовом классе, но с другой функциональностью. Это называется *переопределением* метода. Когда клиентский код вызывает переопределенный метод, вызывается метод в подклассе. (Однако код метода в подклассе все еще может вызвать метод с тем же именем в базовом классе.)
- Подкласс может добавлять новые методы и переменные экземпляра, которых нет в базовом классе.

Один из способов представить подкласс — с помощью фразы *кодирование по разнице*. Поскольку подкласс наследует все переменные экземпляра и методы базового класса, ему не нужно повторять весь этот код; следовательно, подклассу необходимо лишь содержать код, который отличает его от базового класса. Поэтому код подкласса содержит лишь новые переменные экземпляра (и инициализацию), переопределяющие методы, и/или новые методы, которых нет в базовом классе.

Реализуем наследование

Синтаксис наследования в Python прост и элегантен. Базовому классу не нужно *знать*, что он используется как базовый. Только подкласс должен указать, что он хочет наследовать от базового класса. Ниже представлен общий синтаксис:

```
class <ИмяБазовогоКласса>():  
    # Методы базового класса  
  
class <ИмяПодкласса>(<ИмяБазовогоКласса>):  
    # Методы подкласса
```

В операторе подкласса `class` внутри скобок вы указываете имя базового класса, от которого он должен наследовать. В данном случае мы хотим, чтобы `<ИмяПодкласса>` наследовал

от базового класса *<ИмяБазовогоКласса>*. Ниже приведен пример с реальными именами классов:

```
class Widget():
    # Методы Widget

class WidgetWithFrills(Widget):
    # Методы WidgetWithFrills
```

Класс `Widget` обеспечит общие функциональные возможности. Класс `WidgetWithFrills` (виджет с оборочками) включит все из класса `Widget` и определит любые дополнительные методы и переменные экземпляра, которые ему необходимы, с более специфичными возможностями.

Пример работника и менеджера

Чтобы прояснить ключевые понятия, начну с очень простого, а затем перейду к более сложным практическим примерам.

Базовый класс: работник

В листинге 10.1 определен базовый класс под названием `Employee`.

Файл: `EmployeeManagerInheritance/EmployeeManagerInheritance.py`

```
# Наследование Employee Manager
#
# Определяем класс Employee, который мы будем использовать как
# базовый класс

class Employee():
    def __init__(self, name, title, ratePerHour=None):
        self.name = name
        self.title = title
        if ratePerHour is not None:
            ratePerHour = float(ratePerHour)
        self.ratePerHour = ratePerHour

    def getName(self):
        return self.name

    def getTitle(self):
        return self.title
```

```
def payPerYear(self):
    # 52 недели * 5 дней в неделю * 8 часов в день
    pay = 52 * 5 * 8 * self.ratePerHour
    return pay
```

Листинг 10.1. Класс Employee, который будет использоваться как базовый класс

В классе Employee есть методы `__init__()`, `getName()`, `getTitle()` и `payPerYear()`. В нем также есть три переменные экземпляра `self.name`, `self.title` и `self.ratePerHour`, которые устанавливаются в методе `__init__()`. Мы извлекаем имя и название с помощью методов геттера. У этих работников почасовая оплата, поэтому `self.payPerYear()` выполняет вычисления, чтобы определить годовую оплату на основе почасовой ставки. В этом классе все должно быть вам знакомо; здесь нет ничего нового. Вы можете создать экземпляр объекта Employee сам по себе, и он будет работать прекрасно.

Подкласс: менеджер

Для класса Manager мы рассмотрим разницу между менеджером и работником: менеджер — наемный работник с некоторым количеством подчиненных. Если он хорошо выполняет свою работу, то получает 10-процентный бонус за год. Класс Manager может расширить класс Employee, поскольку менеджер является работником, но обладает дополнительными возможностями и обязанностями.

В листинге 10.2 показан код нашего класса Manager. Он должен содержать лишь код, который отличается от класса Employee, поэтому вы не увидите в нем методов `getName()` и `getTitle()`. Любые вызовы этих методов с объектом Manager будут обрабатываться методами класса Employee.

Файл: EmployeeManagerInheritance/ EmployeeManagerInheritance.py

```
# Определяем подкласс Manager, который наследует от Employee

❶ class Manager(Employee):
    def __init__(self, name, title, salary, reportsList=None):
❷         self.salary = float(salary)
            if reportsList is None:
                reportsList = []
            self.reportsList = reportsList
❸         super().__init__(name, title)
```

```

❹ def getReports(self):
    return self.reportsList

❺ def payPerYear(self, giveBonus=False):
    pay = self.salary
    if giveBonus:
        pay = pay + (.10 * self.salary) # добавляем бонус 10%
❻ print(self.name, 'gets a bonus for good work')
    return pay

```

Листинг 10.2. Класс `Manager`, реализованный как подкласс класса `Employee`

В операторе `class` ❶ вы можете увидеть, что этот класс наследует от класса `Employee`, потому что `Employee` находится внутри скобок после имени `Manager`.

Метод `__init__()` класса `Employee` ожидает имя, название и дополнительную ставку за час. Менеджер является наемным работником и управляет некоторым количеством работников, поэтому метод `__init__()` класса `Manager` ожидает имя, название, зарплату и список работников. Придерживаясь принципа кодирования по разнице, метод `__init__()` начинает с инициализации всего, что метод `__init__()` класса `Employee` не делает. Следовательно, мы сохраняем `salary` и `reportsList` в переменных экземпляра с одинаковыми именами ❷.

Далее мы хотим вызвать метод `__init__()` базового класса `Employee` ❸. Здесь я вызываю встроенную функцию `super()`, которая просит Python выяснить, какой класс является базовым (часто упоминается как *суперкласс*), и вызывает метод `__init__()` этого класса. Она также настраивает аргументы, чтобы они включали `self` в качестве первого аргумента в этом вызове. Следовательно, вы можете рассматривать эту строку как преобразование в:

```
Employee.__init__(self, name, title)
```

По сути, кодирование этой строки подобным образом будет работать отлично; но использование вызова `super()` — гораздо более чистый способ записи вызова без необходимости указания имени базового класса.

В результате новый метод `__init__()` класса `Manager` инициализирует две переменные экземпляра (`self.salary` и `self.reportsList`), которые отличаются от находящихся

в классе `Employee`, а метод `__init__()` класса `Employee` инициализирует переменные экземпляра `self.name` и `self.title`, являющиеся общими для любого создаваемого объекта `Employee` или `Manager`. Для `Manager`, у которого есть зарплата, `self.ratePerHour` устанавливается в значение `None`.

ПРИМЕЧАНИЕ Предыдущие версии Python требуют от вас написания этого кода третьим способом, поэтому в старых программах и документации вы можете увидеть это:

```
super(Employee, self).__init__(name, salary)
```

Этот код выполняет абсолютно то же самое. Однако более новый синтаксис с простым вызовом `super()` гораздо проще запомнить. Использование `super()` также делает его менее подверженным ошибкам, если вы решите изменить имя вашего базового класса.

В классе `Manager` есть добавленный метод геттера `getReports()` ④, который позволяет клиентскому коду извлекать список `Employee`, подотчетных `Manager`. Метод `payPerYear()` ⑤ вычисляет и возвращает оплату `Manager`.

Обратите внимание, что в обоих классах `Employee` и `Manager` есть метод с именем `payPerYear()`. Если вы вызовете метод `payPerYear()`, используя экземпляр `Employee`, будет выполняться метод класса `Employee`, который вычисляет оплату на основании почасовой ставки. Если вы вызовете метод `payPerYear()` для экземпляра `Manager`, задействуется метод класса `Manager`, осуществляющий другое вычисление. Метод `payPerYear()` в классе `Manager` *переопределяет* метод с тем же именем в базовом классе. Переопределение метода в подклассе указывает подкласс, чтобы отличать его от базового. Имя переопределяющего метода должно быть абсолютно таким же, как имя переопределяемого метода (хотя у него может быть другой список параметров). В переопределяющем методе вы можете:

- полностью заменить переопределяемый метод в базовом классе. Мы видим это в методе `payPerYear()` класса `Manager`;
- выполнить некоторую работу самостоятельно и вызвать наследуемый и переопределяемый метод с тем же именем в базовом классе. Мы видим это в методе `__init__()` класса `Manager`.

Фактическое содержимое переопределяющего метода зависит от ситуации. Если клиент вызывает метод, которого не существует в подклассе, вызов метода будет отправлен в базовый класс. Например, обратите внимание, что в классе `Manager` нет метода с именем `getName()`, но он существует в базовом классе `Employee`. Если клиент вызывает `getName()` для экземпляра `Manager`, этот вызов обрабатывается базовым классом `Employee`.

Метод `payPerYear()` класса `Manager` содержит следующий код:

```
if giveBonus:
    pay = pay + (.10 * self.salary) # добавляем бонус 10%
    print(self.name, 'gets a bonus for good work')
```

Переменная экземпляра `self.name` была определена в классе `Employee`, но в классе `Manager` она до этого не упоминалась. Это демонстрирует, что определенные в базовом классе переменные экземпляра доступны к использованию в методах подкласса. Здесь мы вычисляем оплату менеджера, которая работает правильно, потому что у `payPerYear()` есть доступ к переменным экземпляра, определенным внутри его собственного класса (`self.salary`), и к переменным экземпляра, определенным внутри базового класса (вывод на экран с использованием `self.name` ❹).

Тестовый код

Давайте протестируем наши объекты `Employee` и `Manager` и вызовем методы каждого из них.

Файл: `EmployeeManagerInheritance/EmployeeManagerInheritance.py`

```
# создаем объекты
oEmployee1 = Employee('Joe Schmoe', 'Pizza Maker', 16)
oEmployee2 = Employee('Chris Smith', 'Cashier', 14)
oManager = Manager('Sue Jones', 'Pizza Restaurant Manager',
                    55000, [oEmployee1, oEmployee2])

# вызываем методы объектов Employee
print('Employee name:', oEmployee1.getName())
print('Employee salary:', '{:,.2f}'.format(oEmployee1.payPerYear()))
print('Employee name:', oEmployee2.getName())
print('Employee salary:', '{:,.2f}'.format(oEmployee2.payPerYear()))
```

```

print()

# вызываем методы объекта Manager
managerName = oManager.getName()
print('Manager name:', managerName)

# Даем менеджеру бонус
print('Manager salary:', '{:,.2f}'.format(oManager.payPerYear(True)))
print(managerName, '(' + oManager.getTitle() + ')', 'direct reports:')
reportsList = oManager.getReports()
for oEmployee in reportsList:
    print(' ', oEmployee.getName(),
          '(' + oEmployee.getTitle() + ')')

```

Когда выполняем этот код, мы видим следующие выходные данные, как мы и ожидали:

```

Employee name: Joe Schmoe
Employee salary: 33,280.00
Employee name: Chris Smith
Employee salary: 29,120.00

Manager name: Sue Jones
Sue Jones gets a bonus for good work
Manager salary: 60,500.00
Sue Jones (Pizza Restaurant Manager) direct reports:
    Joe Schmoe (Pizza Maker)
    Chris Smith (Cashier)

```

Представление клиента о подклассе

До настоящего момента обсуждение было сосредоточено на деталях реализации. Но класс может выглядеть по-разному в зависимости от того, являетесь ли вы разработчиком класса или пишете код для его использования. Давайте изменим фокус и посмотрим на наследование с точки зрения клиента. Что касается клиентского кода, подкласс обладает всеми функциональными возможностями базового класса плюс все, что определено самим подклассом. Может оказаться полезным представить итоговую коллекцию методов в виде слоев краски на стене. Когда клиент просматривает класс `Employee`, он видит все определенные в этом классе методы (рис. 10.3).

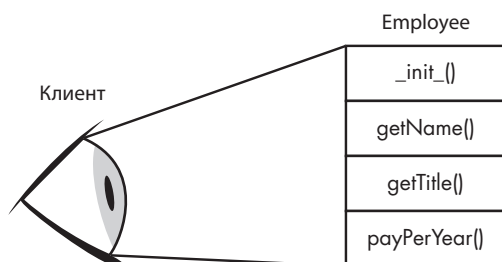


Рис. 10.3. Что увидит клиент, просматривая интерфейс класса `Employee`

Когда мы вводим класс `Manager`, который наследует от класса `Employee`, это похоже на добавление краски, чтобы подправить места, где хотим добавить или изменить методы. Для методов, которые не нужно менять, мы просто оставляем старые слои краски (рис. 10.4).

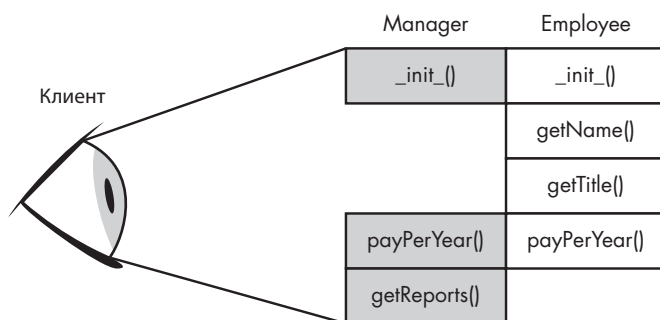


Рис. 10.4. Что увидит клиент, просматривая интерфейс класса `Manager`

Как разработчики мы знаем, что класс `Manager` наследует от класса `Employee` и переопределяет некоторые методы. Как клиент мы лишь видим пять методов. Клиенту не нужно знать, что некоторые методы реализуются в классе `Manager`, а источник других находится в классе `Employee`.

Примеры наследования из реального мира

Давайте рассмотрим два примера наследования из реального мира. Сначала я покажу вам, как создать поле ввода, которое разрешает вам вносить только числа. Затем создам поле вывода, которое форматирует денежные значения.

InputNumber

В этом первом примере мы создадим поле ввода, которое разрешает пользователю вводить только числовые данные. В качестве общего принципа проектирования пользовательского интерфейса гораздо лучше ограничить ввод, чтобы разрешить пользователю вносить только правильно отформатированные данные, вместо того чтобы разрешать любые входные данные и проверять позднее их правильность. Ввод в это поле букв или других символов, десятичных знаков и отрицательных чисел не должен быть разрешен.

Пакет `pygame` содержит класс `InputText`, который позволяет пользователю вводить любые символы. Мы напишем класс `InputNumber`, чтобы в качестве входных данных разрешить лишь допустимые числа. Новый класс `InputNumber` унаследует большую часть своего кода от `InputText`. Нам только понадобится переопределить три метода `InputText`: `__init__()`, `handleEvent()` и `getValue()`. В листинге 10.3 продемонстрирован класс `InputNumber`, который переопределяет эти методы.

Файл: MoneyExamples/InputNumber.py

```
# Класс InputNumber - разрешает пользователю вводить только числа
#
# Демонстрация наследования

import pygame
from pygame.locals import *
import pygamewidgets

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
# Кортеж допустимых клавиш редактирования
LEGAL_KEYS_TUPLE = (pygame.K_RIGHT, pygame.K_LEFT, pygame.K_HOME,
                    pygame.K_END, pygame.K_DELETE, pygame.K_BACKSPACE,
                    pygame.K_RETURN, pygame.K_KP_ENTER)
# Допустимые клавиши для ввода
LEGAL_UNICODE_CHARS = ('0123456789.-')

#
# InputNumber наследует от InputText
#
class InputNumber(pygamewidgets.InputText):

    def __init__(self, window, loc, value='', fontName=None, ❶
                 fontSize=24, width=200, textColor=BLACK,
```

```

        backgroundColor=WHITE, focusColor=BLACK,
        initialFocus=False, nickName=None, callback=None,
        mask=None, keepFocusOnSubmit=False,
        allowFloatingNumber=True, allowNegativeNumber=True):
self.allowFloatingNumber = allowFloatingNumber
self.allowNegativeNumber = allowNegativeNumber

# Вызываем метод __init__() нашего базового класса
super().__init__(window, loc, value, fontName, fontSize, ❷
                 width, textColor, backgroundColor,
                 focusColor, initialFocus, nickName, callback,
                 mask, keepFocusOnSubmit)

# Переопределяем handleEvent, чтобы фильтровать подходящие клавиши
def handleEvent(self, event): ❸
    if (event.type == pygame.KEYDOWN):
        # Если это не клавиша редактирования или числовая клавиша,
        # игнорируем ее
        # Значение Юникода присутствует только при нажатии клавиши
        # вниз
        allowableKey = (event.key in LEGAL_KEYS_TUPLE) or
                        (event.unicode in LEGAL_UNICODE_CHARS)
        if not allowableKey:
            return False

    if event.unicode == '-': # пользователь ввел знак минус
        if not self.allowNegativeNumber:
            # Если нет отрицательных величин, не надо передавать
            return False
        if self.cursorPosition > 0:
            return False # нельзя поместить знак минус после
                        # 1-го символа
        if '-' in self.text:
            return False # нельзя ввести второй знак минус

    if event.unicode == '.':
        if not self.allowFloatingNumber:
            # Если нет плавающих точек, не передаем точку
            return False
        if '.' in self.text:
            return False # нельзя ввести вторую точку

    # разрешаем клавише перейти к базовому классу
    result = super().handleEvent(event)
    return result

def getValue(self): ❹
    userString = super().getValue()

```

```

try:
    if self.allowFloatingNumber:
        returnValue = float(userString)
    else:
        returnValue = int(userString)
except ValueError:
    raise ValueError('Entry is not a number, needs to have at
                      least one digit.')

return returnValue

```

Листинг 10.3. InputNumber разрешает пользователю вводить только числовые данные

Метод `__init__()` разрешает те же параметры, что находятся в базовом классе `InputText`, плюс еще несколько ❶. Он добавляет булевы выражения `allowFloatingNumber`, чтобы определить, можно ли позволить пользователю вводить числа с плавающей точкой, и `allowNegativeNumber`, чтобы определить, можно ли вводить числа, начинающиеся со знака минус. Оба по умолчанию принимают значение `True`, поэтому случай по умолчанию разрешает пользователю вводить числа с плавающей точкой и как положительные, так и отрицательные числа. Вы могли бы их использовать, чтобы ограничить пользователя, например, вводом только положительных целых чисел, установив оба значения на `False`. Метод `__init__()` сохраняет значения этих необязательных параметров в переменных экземпляра, затем вызывает метод `__init__()` базового класса с помощью вызова `super()` ❷.

Значимый код находится в методе `handleEvent()` ❸, который ограничивает допустимые клавиши до небольшого подмножества: числа от нуля до девяти, знак минус, точка (десятичная точка), **Enter** и несколько клавиш редактирования. Когда пользователь нажимает клавишу, вызывается этот метод и передается событие `KEYDOWN` или `KEYUP`. Сначала код удостоверяется, что нажатая клавиша принадлежит ограниченному набору. Если пользователь нажимает клавишу не из этого набора (например, любую букву), мы возвращаем значение `False`, чтобы обозначить, что не произошло ничего важного в этом виджете и что клавиша игнорируется.

Метод `handleEvent()` проводит еще несколько проверок, чтобы убедиться, что вводимые числа допустимы (например, не содержат двух точек, содержат лишь один знак минус и так далее). Каждый раз, когда обнаруживается нажатие допустимой

клавиши, код вызывает метод `handleEvent()` базового класса `InputText`, чтобы выполнить все необходимое с этой клавишей (отобразить или отредактировать поле).

Когда пользователь нажимает **Return** или **Enter**, клиентский код вызывает метод `getValue()` ❹, чтобы получить ввод пользователя. Метод `getValue()` в этом классе вызывает `getValue()` в классе `InputText`, чтобы получить строку из поля, затем пытается преобразовать эту строку в число. Если это преобразование не удастся, он вызывает исключение.

Переопределяя методы, мы создали очень эффективный новый класс многократного использования, который расширяет функциональные возможности класса `InputText`, не меняя при этом ни единой строки в базовом классе. `InputText` продолжит функционировать как класс сам по себе без каких-либо изменений его функциональных возможностей.

DisplayMoney

В качестве второго примера из реального мира мы создадим поле для отображения суммы денег. Для универсальности будем отображать сумму с выбранным символом валюты, поместим его слева или справа от текста (в зависимости от обстоятельств) и отформатируем число, добавив запятые между каждыми тремя цифрами, за которыми следует точка и затем две десятичные цифры. Например, мы хотели бы отобразить 1234,56 доллара США в виде \$1234,56.

В пакете `pygwidgets` уже есть класс `DisplayText`. Мы можем создать экземпляр объекта из этого класса, используя следующий интерфейс:

```
def __init__(self, window, loc=(0, 0), value='',
             fontName=None, fontSize=18, width=None, height=None,
             textColor=PYGWIDGETS_BLACK, backgroundColor=None,
             justified='left', nickname=None):
```

Давайте предположим, что у нас некоторый код, который создает объект `DisplayText` с именем `oSomeDisplayText`, используя соответствующие аргументы. Каждый раз, когда хотим обновить текст в объекте `DisplayText`, мы должны вызвать метод `setValue()` следующим образом:

```
oSomeDisplayText.setValue('1234.56')
```

Функциональные возможности отображения текста (в виде строки) с помощью объекта `DisplayText` уже существуют. Мы хотим создать новый класс с именем `DisplayMoney`, который аналогичен `DisplayText`, но добавляет функциональность, поэтому мы наследуем от `DisplayText`.

В классе `DisplayMoney` будет улучшенная версия метода `setValue()`, которая переопределяет метод базового класса `setValue()`. Версия `DisplayMoney` внесет необходимое форматирование, добавляя символ валюты, запятые, дополнительно сокращая до двух десятичных цифр, и так далее. В конце метод вызовет унаследованный метод `setValue()` базового класса `DisplayText` и передаст версию строки отформатированного текста для отображения в окне.

Мы также добавим некоторые дополнительные параметры установки в метод `__init__()`, чтобы клиентский код:

- выбирал символ валюты (по умолчанию \$);
- помещал символ валюты слева или справа (по умолчанию слева);
- отображал или скрывал два десятичных разряда (по умолчанию отображает).

В листинге 10.4 продемонстрирован код нашего нового класса `DisplayMoney`.

Файл: `MoneyExamples/DisplayMoney.py`

```
# Класс DisplayMoney - отображает число в виде денежной суммы
#
# Демонстрация наследования

import pygwidgets

BLACK = (0, 0, 0)

#
# Класс DisplayMoney наследует от класса DisplayText
#

❶ class DisplayMoney(pygwidgets.DisplayText):

    ❷     def __init__(self, window, loc, value=None,
                    fontName=None, fontSize=24, width=150, height=None,
                    textColor=BLACK, backgroundColor=None,
```

```

        justified='left', value=None, currencySymbol='$',
        currencySymbolOnLeft=True, showCents=True):

3   self.currencySymbol = currencySymbol
    self.currencySymbolOnLeft = currencySymbolOnLeft
    self.showCents = showCents
    if value is None:
        value = 0.00

    # вызываем метод __init__() нашего базового класса
4   super().__init__(window, loc, value,
                    fontName, fontSize, width, height,
                    textColor, backgroundColor, justified)

5   def setValue(self, money):
        if money == '':
            money = 0.00

        money = float(money)

        if self.showCents:
            money = '{:,.2f}'.format(money)
        else:
            money = '{:,.0f}'.format(money)

        if self.currencySymbolOnLeft:
            theText = self.currencySymbol + money
        else:
            theText = money + self.currencySymbol

        # вызываем метод setValue нашего базового класса
6   super().setValue(theText)

```

Листинг 10.4. DisplayMoney отображает число, отформатированное в виде денежного значения

В определении класса мы явно наследуем от `pygame.widgets.DisplayText` ❶. Класс `DisplayMoney` содержит лишь два метода: `__init__()` и `setValue()`. Они переопределяют методы с аналогичными именами в базовом классе.

Клиент создает экземпляр объекта `DisplayMoney` следующим образом:

```
oDisplayMoney = DisplayMoney(window, (100, 100), 1234.56)
```

С помощью этой строки метод `__init__()` в `DisplayMoney` ❷ выполнит и переопределит метод `__init__()` в базовом классе.

Он выполняет некоторую инициализацию, включая сохранение любых клиентских предпочтений символов валюты, стороны отображения символа и отображения с точностью до сотых, все в переменных экземпляра ❸. Метод заканчивается вызовом метода `__init__()` базового класса `DisplayText` ❹ (который он находит, вызывая `super()`) и передает данные, требуемые этим методом.

Позднее клиент, чтобы отобразить значение, выполняет вызов следующим образом:

```
oDisplayMoney.setValue(12233.44)
```

Метод `setValue()` ❺ в классе `DisplayMoney` выполняется, чтобы создать версию суммы денег, отформатированную в виде значения валюты. Метод завершается вызовом унаследованного метода `setValue()` в классе `DisplayText` ❻, чтобы установить новый текст для отображения.

Когда идет вызов любого другого метода с экземпляром `DisplayMoney`, выполняется версия, находящаяся в `DisplayText`. И самое главное, каждый раз во время прохождения цикла клиентский код должен вызывать `oDisplayMoney.draw()`, который рисует поле в окне. Поскольку в `DisplayMoney` нет метода `draw()`, этот вызов будет отправляться базовому классу `DisplayText`, в котором есть метод `draw()`.

Пример использования

На рис. 10.5 показаны выходные данные примера программы, в которой используется как класс `InputNumber`, так и класс `DisplayMoney`. Пользователь вводит число в поле `InputNumber`. Когда он нажмет **ОК** или **Enter**, это значение будет отображаться в двух полях `DisplayMoney`. Первое показывает число с десятичными разрядами, второе округляет до ближайшего доллара, используя различные исходные настройки.

В листинге 10.5 содержится весь код основной программы. Обратите внимание, что код создает один объект `InputNumber` и два объекта `DisplayMoney`.

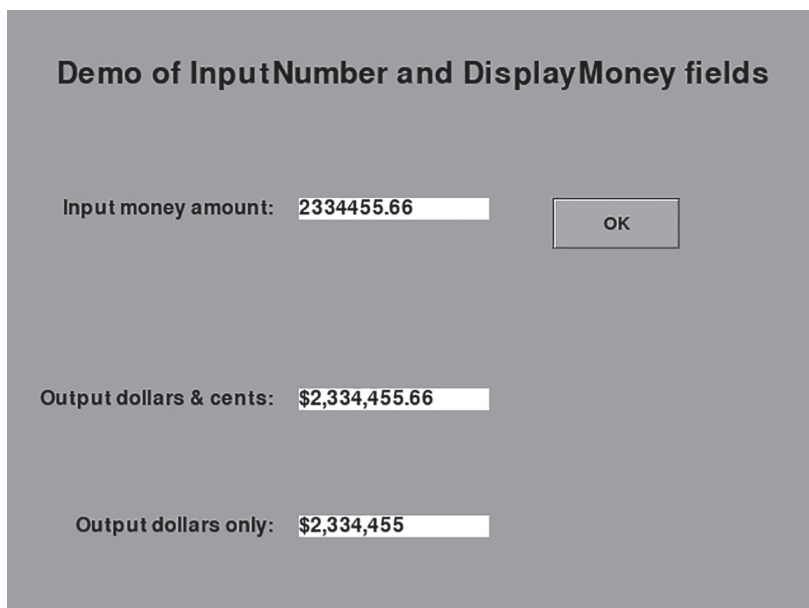


Рис. 10.5. Клиентская программа, в которой пользователь вводит сумму в поле `InputNumber` и сумма отображается в двух полях `DisplayMoney`

Файл: `MoneyExamples/Main_MoneyExample.py`

```
# Денежный пример
#
# Демонстрирует переопределение унаследованных методов DisplayText
# и InputText

#1 - Импортируем пакеты
import pygame
from pygame.locals import *
import sys
import pygwidgets
from DisplayMoney import *
from InputNumber import *

#2 - Определяем константы
BLACK = (0, 0, 0)
BLACKISH = (10, 10, 10)
GRAY = (128, 128, 128)
WHITE = (255, 255, 255)
BACKGROUND_COLOR = (0, 180, 180)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
```



```

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode([WINDOW_WIDTH, WINDOW_HEIGHT])
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.

#5 - Инициализируем переменные
title = pygwidgets.DisplayText(window, (0, 40),
                                'Demo of InputNumber and DisplayMoney'
                                'fields', fontSize=36, width=WINDOW_WIDTH,
                                justified='center')

inputCaption = pygwidgets.DisplayText(window, (20, 150),
                                       'Input money amount:', fontSize=24,
                                       width=190, justified='right')
inputField = InputNumber(window, (230, 150), '', width=150)
okButton = pygwidgets.TextButton(window, (430, 150), 'OK')

outputCaption1 = pygwidgets.DisplayText(window, (20, 300),
                                       'Output dollars & cents: ', fontSize=24,
                                       width=190, justified='right')
moneyField1 = DisplayMoney(window, (230, 300), '', textColor=BLACK,
                           backgroundColor=WHITE, width=150)

outputCaption2 = pygwidgets.DisplayText(window, (20, 400),
                                       'Output dollars only: ', fontSize=24,
                                       width=190, justified='right')
moneyField2 = DisplayMoney(window, (230, 400), '', textColor=BLACK,
                           backgroundColor=WHITE, width=150,
                           showCents=False)

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        # Если событием был щелчок по кнопке закрытия, выходим
        # из pygame и программы
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        # Нажатие Return/Enter или щелчок по ОК приводит к действию
        if inputField.handleEvent(event) or okButton.handleEvent(event): ❶
            try:
                theValue = inputField.getValue()
            except ValueError: # любая оставшаяся ошибка
                inputField.setValue('(not a number)')

```

```

        else: # ввод был ОК
            theText = str(theValue)
            moneyField1.setValue(theText)
            moneyField2.setValue(theText)

#8 - Выполняем действия "в рамках фрейма"

#9 - Очищаем окно
window.fill(BACKGROUND_COLOR)

#10 - Рисуем все элементы окна
title.draw()
inputCaption.draw()
inputField.draw()
okButton.draw()
outputCaption1.draw()
moneyField1.draw()
outputCaption2.draw()
moneyField2.draw()

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) #ожидание pygame

```

Листинг 10.5. Основная программа для демонстрации классов `InputNumber` и `DisplayMoney`

Пользователь вводит число в поле `InputNumber`. Когда он печатает, любые неподходящие символы отфильтровываются и игнорируются методом `handleEvent()`. Когда пользователь щелкает по ОК **1**, код считывает входные данные и передает их в два поля `DisplayMoney`. Первое отображает сумму в долларах и центах (с двумя десятичными цифрами), в то время как второе показывает значение только в долларах. Оба добавляют \$ в качестве символа валюты и разделяют запятыми каждые три цифры.

Наследование нескольких классов от одного базового класса

Несколько различных классов способны наследовать от одного и того же базового класса. Вы можете создать самый общий базовый класс, затем выстроить любое количество подклассов, которые наследуют от него. Рис. 10.6 является представлением этих взаимоотношений.

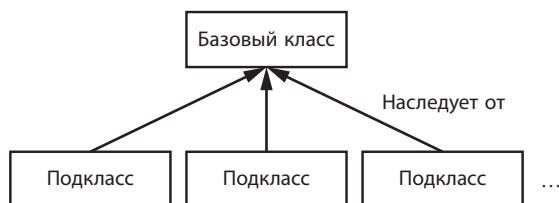


Рис. 10.6. Три и более различных подклассов, наследующих от общего базового класса

Каждый из различных подклассов может быть затем вариантом (более специфичной версией) общего базового класса. Каждый подкласс переопределяет любые желаемые или необходимые методы в базовом классе, независимо от других подклассов.

Давайте разберем пример, используя программу `Shapes` из главы 9, которая создавала и рисовала круги, квадраты и треугольники. Код также разрешал пользователю щелкать по любой фигуре в окне, чтобы увидеть площадь этой фигуры.

Программа была реализована с тремя различными классами фигур: `Circle`, `Square` и `Triangle`. Если мы вернемся к этим трем классам, то обнаружим, что у каждого из них есть абсолютно одинаковый метод:

```
def getType(self):  
    return self.shapeType
```

Далее, рассматривая методы `__init__()` трех классов, мы обнаружили некий общий код, который запоминает окно, выбирает произвольный цвет и произвольное местоположение:

```
self.window = window  
self.color = random.choice((RED, GREEN, BLUE))  
self.x = random.randrange(1, maxWidth - 100)  
self.y = random.randrange(1, maxHeight - 100)
```

И наконец, каждый класс устанавливает переменную экземпляра `self.shapeType` в соответствующую строку.

Всякий раз, находя набор классов, который реализует абсолютно одинаковый метод и/или содержит общий код в методе с общим именем, мы должны признать его хорошим кандидатом для наследования.

Давайте извлечем общий код из трех классов и создадим общий базовый класс с именем `Shape`, продемонстрированный в листинге 10.6.

Файл: `InheritedShapes/ShapeBasic.py`

```
# Класс Shape - базовый

import random

# Настраиваем цвета
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

class Shape():

    ❶ def __init__(self, window, shapeType, maxWidth, maxHeight):
        self.window = window
        self.shapeType = shapeType
        self.color = random.choice((RED, GREEN, BLUE))
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)

    ❷ def getType(self):
        return self.shapeType
```

Листинг 10.6. Класс `Shape` для использования в качестве базового класса

Класс состоит только из двух методов: `__init__()` и `getType()`. Метод `__init__()` ❶ запоминает данные, передаваемые в переменные экземпляров, затем произвольным образом выбирает цвет и начальное местоположение (`self.x` и `self.y`). Метод `getType()` ❷ лишь возвращает тип фигуры, заданной инициализацией.

Теперь мы можем написать любое число подклассов, которые наследуют от `Shape`. Создадим три подкласса, которые будут вызывать метод `__init__()` класса `Shape`, передавая строку, определяющую его тип и размер окна. Метод `getType()` появится лишь в классе `Shape`, поэтому любой клиентский вызов `getType()` будет обрабатываться этим методом в унаследованном классе `Shape`. Мы начнем с кода класса `Square`, который продемонстрирован в листинге 10.7.

Файл: InheritedShapes/Square.py

```
# Класс Square

import pygame
from Shape import *

class Square(Shape): ❶

    def __init__(self, window, maxWidth, maxHeight):
        super().__init__(window, 'Square', maxWidth, maxHeight) ❷
        self.widthAndHeight = random.randrange(10, 100)
        self.rect = pygame.Rect(self.x, self.y, self.widthAndHeight,
                                self.widthAndHeight)

    def clickedInside(self, mousePoint): ❸
        clicked = self.rect.collidepoint(mousePoint)
        return clicked

    def getArea(self): ❹
        theArea = self.widthAndHeight * self.widthAndHeight
        return theArea

    def draw(self): ❺
        pygame.draw.rect(self.window, self.color,
                          (self.x, self.y, self.widthAndHeight,
                           self.widthAndHeight))
```

Листинг 10.7. Класс Square, который наследует от класса Shape

Класс Square начинается с наследования от класса Shape ❶. Метод `__init__()` вызывает метод `__init__()` его базового класса (или суперкласса) ❷, определяя фигуру как квадрат и произвольно выбирая ее размер.

Далее у нас идут три метода, реализация которых специфична для квадрата. Методу `clickedInside()` всего лишь требуется вызвать `rect.collidepoint()`, чтобы определить, был ли щелчок внутри прямоугольника ❸. Метод `getArea()` просто умножает `widthAndHeight` на `widthAndHeight` ❹. И наконец, метод `draw()` рисует прямоугольник с помощью значения `widthAndHeight` ❺.

В листинге 10.8 продемонстрирован класс `Circle`, который также был изменен, чтобы наследовать от класса `Shape`.

Файл: InheritedShapes/Circle.py

```
# Класс Circle

import pygame
from Shape import *
import math

class Circle(Shape):

    def __init__(self, window, maxWidth, maxHeight):
        super().__init__(window, 'Circle', maxWidth, maxHeight)
        self.radius = random.randrange(10, 50)
        self.centerX = self.x + self.radius
        self.centerY = self.y + self.radius
        self.rect = pygame.Rect(self.x, self.y, self.radius * 2,
                                self.radius * 2)

    def clickedInside(self, mousePoint):
        theDistance = math.sqrt(((mousePoint[0] - self.centerX) ** 2) +
                                ((mousePoint[1] - self.centerY) ** 2))
        if theDistance <= self.radius:
            return True
        else:
            return False

    def getArea(self):
        theArea = math.pi * (self.radius ** 2)
        return theArea

    def draw(self):
        pygame.draw.circle(self.window, self.color,
                            (self.centerX, self.centerY), self.radius, 0)
```

Листинг 10.8. Класс Circle, который наследует от класса Shape

Класс Circle также содержит методы `clickedInside()`, `getArea()` и `draw()`, чья реализация специфична для круга.

И наконец, в листинге 10.9 продемонстрирован код класса `Triangle`.

Файл: InheritedShapes/Triangle.py

```
# Класс Triangle
import pygame
from Shape import *
class Triangle(Shape):

    def __init__(self, window, maxWidth, maxHeight):
        super().__init__(window, 'Triangle', maxWidth, maxHeight)
        self.width = random.randrange(10, 100)
        self.height = random.randrange(10, 100)
        self.triangleSlope = -1 * (self.height / self.width)
        self.rect = pygame.Rect(self.x, self.y, self.width,
                                self.height)

    def clickedInside(self, mousePoint):
        inRect = self.rect.collidepoint(mousePoint)
        if not inRect:
            return False

        # выполняем некоторые вычисления, чтобы увидеть,
        # находится ли точка внутри треугольника
        xOffset = mousePoint[0] - self.x
        yOffset = mousePoint[1] - self.y
        if xOffset == 0:
            return True

        pointSlopeFromYIntercept = (yOffset - self.height) / xOffset
        # подъем по ходу движения
        if pointSlopeFromYIntercept < 1:
            return True
        else:
            return False

    def getArea(self):
        theArea = .5 * self.width * self.height
        return theArea

    def draw(self):
        pygame.draw.polygon(self.window, self.color, (
            (self.x, self.y + self.height),
            (self.x, self.y),
            (self.x + self.width, self.y)))
```

Листинг 10.9. Класс Triangle, который наследует от класса Shape

Основной код, который мы использовали для тестирования в главе 9, вообще не нужно менять. Как клиент новых классов,

он создает экземпляры объектов `Square`, `Circle` и `Triangle`, не беспокоясь о реализации этих классов. Ему не нужно знать, что каждый из них является подклассом общего класса `Shape`.

Абстрактные классы и методы

К сожалению, у нашего класса `Shape` есть потенциальная ошибка. На данный момент клиент мог бы создать экземпляр общего объекта `Shape`, но он слишком общий, чтобы в нем был собственный метод `getArea()`. Далее все объекты класса, которые наследуют от класса `Shape` (такие как `Square`, `Circle` и `Triangle`), *должны* реализовывать `clickedInside()`, `getArea()` и `draw()`. Чтобы решить обе эти проблемы, я познакомлю вас с понятиями *абстрактного класса* и *абстрактного метода*.

Абстрактный класс

Класс, который *не* предназначен для создания экземпляра напрямую, а только для использования в качестве базового класса одним или несколькими подклассами. (В некоторых других языках абстрактный класс называется *виртуальным классом*.)

Абстрактный метод

Метод, который *должен* быть переопределен в каждом подклассе.

Часто базовый класс не в силах правильно реализовать абстрактный метод, потому что он не знает подробные данные, с которыми должен работать, или не может реализовать универсальный алгоритм. Вместо этого всем подклассам необходимо реализовать их собственные версии абстрактного метода.

В нашем примере с фигурами мы хотим, чтобы класс `Shape` был абстрактным, чтобы никакой клиентский код не мог создать экземпляр объекта `Shape`. Далее наш класс `Shape` должен указать, что всем его подклассам необходимо реализовать методы `clickedInside()`, `getArea()` и `draw()`.

В Python нет ключевого слова для обозначения класса или метода в качестве абстрактного. Тем не менее стандартная библиотека Python содержит модуль `abc`, сокращение от *абстрактный базовый класс*, который спроектирован для помощи разработчикам в создании абстрактных базовых классов и методов.

Давайте рассмотрим, что нам необходимо сделать, чтобы создать абстрактный класс с абстрактными методами. Для начала необходимо импортировать две вещи из модуля `abc`:

```
from abc import ABC, abstractmethod
```

Далее мы должны указать, что именно класс, который мы хотим использовать как абстрактный базовый класс, должен унаследовать от класса `ABC`, что мы и делаем, помещая `ABC` внутрь скобок после имени класса:

```
class <класс, который мы хотим назначить абстрактным>(ABC):
```

Затем нужно использовать специальный декоратор `@abstractmethod` перед любыми методами, которые должны быть переопределены во всех подклассах:

```
@abstractmethod
```

```
def <некий метод, который должен быть переопределен>(self, ...):
```

В листинге 10.10 продемонстрировано, как мы можем отметить класс `Shape` как абстрактный базовый класс и указать его абстрактные методы.

Файл: `InheritedShapes/Shape.py`

```
# Класс Shape
#
# Должен использоваться как базовый класс для других классов

import random
from abc import ABC, abstractmethod

# Настраиваем цвета
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

❶ class Shape(ABC): # определяем абстрактный базовый класс

    ❷ def __init__(self, window, shapeType, maxWidth, maxHeight):
        self.window = window
        self.shapeType = shapeType
        self.color = random.choice((RED, GREEN, BLUE))
```

```

        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)

❸ def getType(self):
    return self.shapeType

❹ @abstractmethod
def clickedInside(self, mousePoint):
    raise NotImplementedError

❺ @abstractmethod
def getArea(self):
    raise NotImplementedError

❻ @abstractmethod
def draw(self):
    raise NotImplementedError

```

Листинг 10.10. Базовый класс Shape, который наследует от ABC с абстрактными методами

Класс Shape наследует от класса ABC ❶, говоря Python предотвращать создание экземпляров объекта Shape клиентским кодом напрямую. Любая подобная попытка приведет к следующему сообщению об ошибке:

```

TypeError: Can't instantiate abstract class Shape with abstract
methods clickedInside, draw, getArea

```

Методы `__init__()` ❷ и `getType()` ❸ содержат код, который будет общим для всех подклассов Shape.

Методам `clickedInside()` ❹, `getArea()` ❺ и `draw()` ❻ предшествует декоратор `@abstractmethod`. Он указывает, что эти методы *должны* быть переопределены всеми подклассами Shape. Поскольку эти методы никогда не будут выполнены в абстрактном классе, реализация здесь состоит только в `raise NotImplementedError`, чтобы еще раз подчеркнуть, что метод ничего не делает.

Давайте расширим демонстрацию фигуры, добавив новый класс `Rectangle`, как показано в листинге 10.11. Класс `Rectangle` наследует от абстрактного класса `Shape` и, следовательно, должен реализовывать методы `clickedInside()`, `getArea()` и `draw()`. Я сделаю преднамеренную ошибку в этом подклассе, чтобы показать, что произойдет.

Файл: InheritedShapes/Rectangle.py

```
# Класс Rectangle

import pygame
from Shape import *

class Rectangle(Shape):

    def __init__(self, window, maxWidth, maxHeight):
        super().__init__(window, 'Rectangle', maxWidth, maxHeight)
        self.width = random.randrange(10, 100)
        self.height = random.randrange(10, 100)
        self.rect = pygame.Rect(self.x, self.y, self.width,
                                self.height)

    def clickedInside(self, mousePoint):
        clicked = self.rect.collidepoint(mousePoint)
        return clicked

    def getArea(self):
        theArea = self.width * self.height
        return theArea
```

Листинг 10.11. Класс `Rectangle`, который реализует методы `clickedInside()` и `getArea()`, но не `draw()`.

В качестве демонстрации этот класс по ошибке не содержит метод `draw()`. В листинге 10.12 продемонстрирована измененная версия основного кода, которая включает создание объектов `Rectangle`.

Файл: InheritedShapes/Main_ShapesWithRectangle.py

```
shapesList = []
shapeClassesTuple = ('Square', 'Circle', 'Triangle', 'Rectangle')
for I in range(0, N_SHAPES):
    randomlyChosenClass = random.choice(shapeClassesTuple)
    oShape = randomlyChosenClass(window, WINDOW_WIDTH, WINDOW_HEIGHT)
    shapesList.append(oShape)
```

Листинг 10.12. Основной код, который произвольным образом создает `Squares`, `Circles`, `Triangles` и `Rectangles`

Когда этот код попытается создать объект `Rectangle`, Python сгенерирует следующее сообщение об ошибке:

```
TypeError: Can't instantiate abstract class Rectangle with  
abstract method draw
```

Это говорит нам, что нельзя создать экземпляр объекта `Rectangle`, потому что мы не написали метод `draw()` в классе `Rectangle`. Добавление метода `draw()` в класс `Rectangle` (с соответствующим кодом для изображения прямоугольника) исправит ошибку.

Как `pygame` применяет наследование

Модуль `pygame` применяет наследование для совместного использования общего кода. Например, возьмем два класса кнопок, которые мы обсуждали в главе 7: `TextButton` и `CustomButton`. Классу `TextButton` требуется строка для использования в качестве метки на кнопке, в то время как класс `CustomButton` требует, чтобы вы представили собственную графику. Способ, которым вы создадите экземпляр каждого из этих классов, отличается: вам необходимо указать разный набор аргументов. Однако, как только он создан, все остальные методы обоих объектов будут абсолютно одинаковыми. Это происходит потому, что два класса наследуют от общего базового класса под названием `PygWidgetsButton` (рис. 10.7).

`PygWidgetsButton` является абстрактным классом. Не предполагается, что клиентский код будет создавать его экземпляр, и попытка сделать это сгенерирует ошибку.

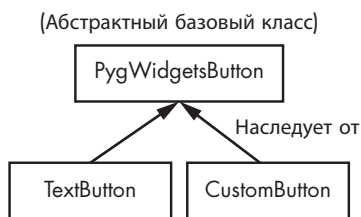


Рис. 10.7. Классы `pygame` `TextButton` и `CustomButton`, оба наследующие от `PygWidgetsButton`

Вместо этого для `PygWidgetsButton` создают подклассы классов `TextButton` и `CustomButton`. Каждый из них предоставляет один метод `__init__()`, который будет делать все необходимое для инициализации их типа кнопки. Затем каждый

из них будет передавать идентичные аргументы в метод `__init__()` базового класса `PygWidgetsButton`.

Класс `TextButton` используется для создания текстовой кнопки с минимальной графикой. Это полезно при попытке быстро запустить и выполнить программу. Ниже представлен интерфейс для создания объекта `TextButton`:

```
def __init__(self, window, loc, text, width=None, height=40,
            textColor=PYGWIDGETS_BLACK,
            upColor= PYGWIDGETS_NORMAL_GRAY,
            overColor= PYGWIDGETS_OVER_GRAY,
            downColor=PYGWIDGETS_DOWN_GRAY,
            fontName=None, fontSize=20, soundOnClick=None,
            enterToActivate=False, callBack=None,
            nickname=None)
```

Хотя многие параметры имеют разумные значения по умолчанию, вызывающий должен предоставить значение для `text`, которое появится на кнопке. Метод `__init__()` сам по себе создает «поверхности» (изображения) для кнопки, которые используются в отображении стандартной кнопки. Код для создания типичного объекта `TextButton` выглядит следующим образом:

```
oButton = pygwidgets.TextButton(window, (50, 50), 'Text Button')
```

Когда кнопка нарисована, пользователи видят ее как показано на рис. 10.8.



Рис. 10.8. Пример типичной `TextButton`

Класс `CustomButton` используется для создания кнопки с помощью графики, которую предоставляет клиент. Ниже показан интерфейс для создания объекта `CustomButton`:

```
def __init__(self, window, loc, up, down=None, over=None,
            disabled=None, soundOnClick=None,
            nickname=None, enterToActivate=False):
```

Ключевая разница заключается в том, что эта версия метода `__init__()` требует, чтобы вызывающий предоставил значения для параметра `up` (помните, что у кнопки есть четыре изображения: «вверх», «вниз», «недоступна для нажатия» и «мышь

наведена»). Дополнительно вы можете также предоставить изображения «вверх», «мышь наведена» и «недоступна для нажатия». Для не предоставленного изображения CustomButton делает копию изображения кнопки «вверх» и использует ее.

Последняя строка метода `__init__()` для *обоих* классов `TextButton` и `CustomButton` представляет собой вызов метода `__init__()` общего базового класса `PygWidgetsButton`. Оба вызова передают четыре изображения для кнопки наряду с другими аргументами:

```
super().__init__(window, loc, surfaceUp, surfaceOver,
                 surfaceDown, surfaceDisabled, buttonRect,
                 soundOnClick, nickname, enterToActivate,
                 callBack)
```

С точки зрения клиента, вы видите два абсолютно разных класса со множеством методов (большая часть которых идентична). Но, с точки зрения конструктора, вы теперь можете видеть, как наследование позволяет переопределять метод `__init__()` в базовом классе, чтобы предоставить программистам клиента два похожих и одинаково полезных способа создания кнопок. У двух классов общее все, кроме метода `__init__()`. Следовательно, способ функционирования кнопок и другие доступные вызовы методов (`handleEvent()`, `draw()`, `disable()`, `enable()` и так далее) должны быть идентичны.

Существует ряд преимуществ у подобного типа наследования. Во-первых, оно обеспечивает согласованность как для клиентского кода, так и для конечного пользователя: объекты `TextButton` и `CustomButton` работают одинаково. Оно также упрощает исправление ошибок: исправление ошибки в базовом классе означает, что вы также исправили ошибки во всех подклассах, которые наследуют от него. И наконец, если вы добавляете функциональные возможности в базовый класс, они сразу же становятся доступными во всех классах, которые наследуют от базового.

Иерархия классов

Любой класс можно использовать в качестве базового, даже подкласс, который уже наследует от другого базового класса. Этот вид взаимоотношений, известный как *иерархия классов*, изображен на рис. 10.9.

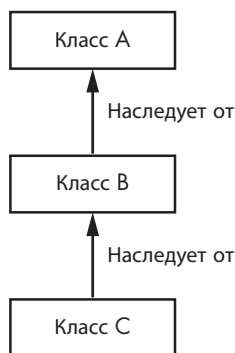


Рис. 10.9. Иерархия классов

На этом рисунке класс С наследует от класса В, который наследует от класса А. Следовательно, класс С является подклассом, а класс В — базовый класс, но класс В также является подклассом класса А. Таким образом, класс В исполняет обе роли. В подобных случаях класс С наследует не только все методы и переменные экземпляра в классе В, но также все методы и переменные экземпляра в классе А. Этот тип иерархии полезен при создании все более специфичных классов. Класс А может быть очень общим, класс В — более детализированным, а класс С — еще более специфичным.

На рис. 10.10 представлен другой способ восприятия взаимоотношений в иерархии классов.

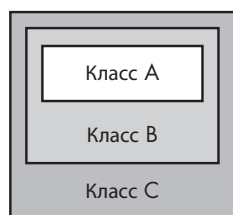


Рис. 10.10. Другой способ изображения иерархии классов

Здесь клиент видит только класс С, но этот класс состоит из всех методов и переменных экземпляра, определенных совместно в классах С, В и А.

Пакет `pygame.widgets` использует иерархию классов для всех виджетов. Первый класс в `pygame.widgets` — это абстрактный класс `PygWidget`, который предоставляет базовые функциональные возможности всем виджетам в пакете. Его код состоит из методов, которые позволяют отображать и скрывать, включать

и выключать, получать и устанавливать местоположение и получать псевдоним (внутреннее имя) любого виджета.

В `pygwidgets` существуют другие классы, которые используются в качестве абстрактных классов, включая вышеупомянутый `PygWidgetsButton`, который является базовым классом для `TextButton` и `CustomButton`. Рис. 10.11 должен помочь прояснить эти взаимоотношения.

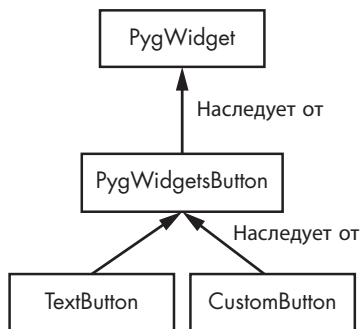


Рис. 10.11. Иерархия классов в `pygwidgets`

Как видите, класс `PygWidgetsButton` является подклассом `PygWidget` и базовым классом для `TextButton` и `CustomButton`.

Сложность программирования с наследованием

При проектировании с использованием наследования может оказаться сложным понять, что и куда помещать. Вы постоянно задаете себе вопросы. Должна ли эта переменная экземпляра быть в базовом классе? Достаточно ли общего кода в подклассах, чтобы создать метод в базовом классе? Каковы подходящие параметры для метода в подклассе? Каковы подходящие параметры и значения по умолчанию для использования в базовом классе, который будет переопределен или вызван из подкласса?

Попытка понять взаимодействие всех переменных и методов в иерархии классов может оказаться невероятно сложной, мудреной и удручающей задачей. Это особенно верно при прочтении кода иерархии класса, которая была разработана другим программистом. Чтобы полностью понять, что происходит, вам, как правило, необходимо ознакомиться с кодом в базовом классе, пройдя весь путь вверх по иерархии.

Например, представим иерархию, где класс D будет подклассом A, являющимся подклассом B, который в свою очередь будет подклассом базового класса A. В классе D вы можете встретить код, чьи ветви основаны на значении переменной экземпляра, но эта переменная могла быть не установлена в коде класса D. В подобных случаях вы должны поискать переменную экземпляра в коде класса C. Если ее там нет, следует поискать в коде класса B и так далее.

При проектировании иерархии класса, вероятно, лучший способ избежать этой проблемы — вызывать только те методы и использовать только те переменные экземпляра, которые наследуются из предыдущего уровня иерархии. В нашем примере код в классе D должен вызывать лишь методы в классе C, в то время как класс C обязан выполнять вызовы только методов класса B и так далее. Это упрощенная версия *Закона Деметры*. Говоря по-простому, вы (подразумевая объекты) должны говорить лишь с вашими близкими друзьями (близлежащие объекты) и никогда не разговаривать с незнакомцами (удаленные объекты). Подробное обсуждение выходит за рамки данной книги, но в интернете доступно множество ссылок на эту тему.

Другой подход, о котором мы впервые упоминали в главе 4, заключается в использовании *композиции*, в которой объект создает экземпляр одного или более объектов. Ключевая разница состоит в том, что наследование применяется для моделирования отношения «is a», в то время как композиция использует отношение «has a». Например, если нам нужен виджет spinbox (редактируемое текстовое числовое поле со стрелками вверх и вниз), мы должны написать класс SpinBox, создающий экземпляры объекта DisplayNumber и двух объектов CustomButton для стрелок. Каждый из этих объектов уже знает, как обрабатывать свои пользовательские взаимодействия.

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

Вы видели, как класс может наследовать от другого класса. По сути, Python (как некоторые другие языки программирования) позволяет классу наследовать от более чем одного класса. Это называется *множественное наследование*. Синтаксис Python для наследования от более чем одного класса довольно прост:

```
class Некий_класс(<Базовый_класс1>, <Базовый_класс2>, ...):
```

Однако важно осознавать, что множественное наследование способно привести к конфликту, когда базовые классы, от которых вы наследуете, содержат одинаково названные методы и/или переменные экземпляра. В Python есть правила (известные как *порядок разрешения методов* или *MRO*) для решения подобных потенциальных проблем. Я считаю это продвинутой темой и не буду здесь рассматривать, но, если вы хотите углубиться в нее, вы можете найти подробное обсуждение по адресу <https://www.python.org/download/releases/2.3/mro>.

Выводы

Это была очень объемная глава по теме наследования: искусство «программирования по разнице». Базовая идея наследования состоит в создании класса (подкласса), который включает все методы и переменные экземпляра другого класса (базового класса), таким образом позволяя вам повторно использовать существующий код. Ваши новые подклассы могут выбрать использование или переопределение методов базового класса, а также определение собственных методов. Метод в подклассе может найти базовый класс с помощью вызова `super()`.

Мы создали два класса `InputNumber` и `DisplayMoney`, которые предоставляют многократно используемые функциональные возможности. Эти классы реализуются как подклассы, использующие классы в пакете `pygwidgets` в качестве базовых.

Любой клиентский код, использующий подкласс, увидит интерфейс, который включает методы, определенные как

в подклассе, так и в базовом классе. С помощью одного и того же базового класса можно создать любое число подклассов. Абстрактный класс — это класс, не предназначенный для создания экземпляров клиентским кодом, а скорее предназначенный для наследования подклассами. Абстрактный метод — это базовый класс, который *должен* быть переопределен в каждом подклассе.

Мы проработали несколько примеров, чтобы продемонстрировать наследование в пакете `pygame.widgets`, включая наследование обоих классов `TextButton` и `CustomButton` от общего базового класса `PygWidgetsButton`.

Я продемонстрировал вам, как выстроить иерархию классов, где класс наследует от другого класса, который в свою очередь наследует от третьего класса и так далее.

Наследование бывает сложным: чтение чужого кода способно привести к путанице, но, как мы убедились, наследование может быть невероятно эффективным.

11

УПРАВЛЕНИЕ ПАМЯТЬЮ, ИСПОЛЪЗУЕМОЙ ОБЪЕКТАМИ



В этой главе я объясню несколько важных концепций Python и ООП, таких как жизненный цикл объекта (включая удаление объектов) и переменные класса, которые не вписывались

в предыдущие главы этой части. Чтобы связать все это воедино, мы создадим небольшую игру. Также я познакомлю вас со слотами, методом управления памятью для объектов. Эта глава должна обеспечить вам лучшее понимание того, как код может повлиять на способ использования памяти объектами.

Жизненный цикл объекта

В главе 2 я определил объект как «данные плюс код, который действует на них с течением времени». Я довольно много говорил о данных (переменные экземпляра) и о коде, который воздействует на них (методы), но особо не объяснил временной аспект. Именно на этом я здесь и сосредоточусь.

Вы уже знаете, что программа может создавать объект в любое время. Часто она будет создавать один или более объектов на начальном этапе и использовать их во время своих

операций. Однако во многих случаях программа захочет создать объект, когда он ей нужен, а потом удалить, когда закончит его использовать, чтобы высвободить используемые объектом ресурсы (память, файлы, сетевые подключения и так далее). Ниже представлено несколько примеров.

- Объект «транзакции», который используется, когда клиент совершает электронную покупку. Когда покупка завершена, объект уничтожается.
- Объект для обработки взаимодействия в сети Интернет, который высвобождается, когда взаимодействие завершено.
- Временные объекты в игре. Программа может создать экземпляры множества копий плохих парней, инопланетян, космических кораблей и так далее; когда игрок уничтожает каждого из них, программа может устранить базовый объект.

Временной период, начиная с создания экземпляра объекта и до момента его уничтожения, называется *жизненный цикл*. Чтобы понять жизненный цикл объекта, в первую очередь вам необходимо знать о связанной базовой концепции, которая имеет отношение к реализации объектов в Python (и некоторых других языках ООП): о подсчете ссылок.

Подсчет ссылок

Существуют различные реализации Python. Дальнейшее обсуждение подсчета ссылок применимо к официальной версии, выпущенной Python Software Foundation, — версии, загружаемой с сайта **python.org**, — которая широко известна как *CPython*. Другие реализации Python могут использовать другой подход.

Часть философии Python состоит в том, что программисты никогда не должны беспокоиться о деталях управления памятью. Python заботится об этом за вас. Однако наличие базового представления о том, как Python управляет памятью, будет полезным для понимания того, как и когда объекты возвращаются в систему.

Когда программа создает экземпляр объекта из класса, Python выделяет память для хранения переменных экземпляра, определенных в классе. Каждый объект также содержит

дополнительное внутреннее поле под названием *подсчет ссылок*, которое отслеживает, какое количество различных переменных ссылается на этот объект. В листинге 11.1 я показываю, как это работает.

Файл: ReferenceCount.py

```
# Пример подсчета ссылок
❶ class Square():

    def __init__(self, width, color):
        self.width = width
        self.color = color

# создаем экземпляр объекта
❷ oSquare1 = Square(5, 'red')
print(oSquare1)
# Количество ссылок объекта Square равно 1

# настраиваем еще одну переменную для того же объекта
❸ oSquare2 = oSquare1
print(oSquare2)
# Количество ссылок объекта Square равно 2
```

Листинг 11.1. Одна переменная (oSquare1), ссылающаяся на объект

Python 3.6
(known limitations)

```
1 # Reference count example
2
3 class Square():
4     def __init__(self, width, color):
5         self.width = width
6         self.color = color
7
8 # Instantiate an object
9 oSquare1 = Square(5, 'red')
10 print(oSquare1)
11 # Reference count of the Square object is 1
12
13 # Now set another variable to the same object
14 oSquare2 = oSquare1
15 print(oSquare2)
16 # Reference count of the Square object is 2
```

[Edit this code](#)

⇒ line that just executed
⇒ next line to execute

Print output (drag lower right corner to resize)

<__main__.Square object at 0x7fea99026780>

Frames

Objects

Global frame

Square

oSquare1

Square class

__init__

function

__init__(self, width, color)

Square instance

color

"red"

width

5

Рис. 11.1. Одна переменная (oSquare1), ссылающаяся на объект

Мы можем использовать Python Tutor (<http://pythontutor.com/>) для прохождения нашего кода. Начинаем с простого класса Square ❶, содержащего несколько переменных экземпляра. Затем создаем экземпляр объекта и присваиваем его переменной

oSquare1 ❷. На рис. 11.1 показано, что мы видим после создания экземпляра первого объекта: переменная oSquare1 ссылается на экземпляр класса Square.

Далее мы устанавливаем вторую переменную для ссылки на тот объект Square ❸ и вывода значения новой переменной. Обратите внимание, что оператор oSquare2 = oSquare1 не создает новую копию объекта Square! На рис. 11.2 показано, что мы видим после выполнения этих двух строк.

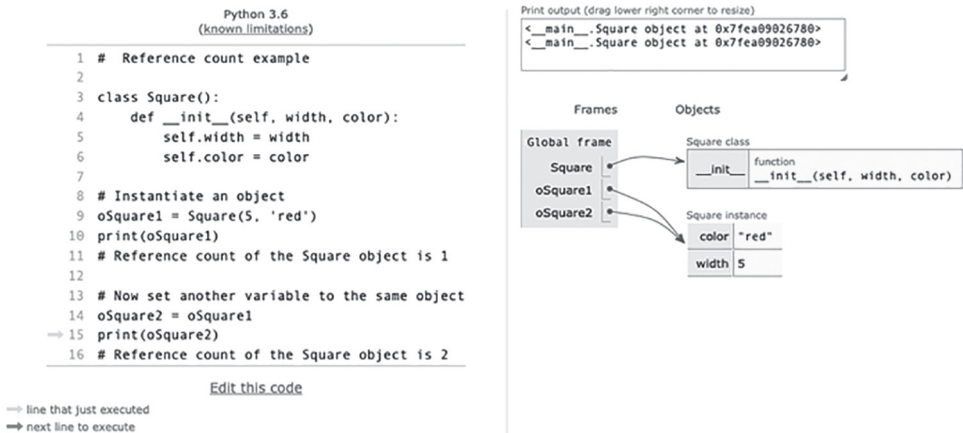


Рис. 11.2. Две переменные, ссылающиеся на один и тот же объект

Обе переменные oSquare1 и oSquare2 ссылаются на один и тот же объект Square. В верхнем блоке вы также можете видеть, что два вызова print () отображают один и тот же адрес памяти. Следовательно, количество ссылок объекта теперь равно 2. Если бы мы собирались назначить еще одну переменную:

```
oSquare3 = oSquare2 # или oSquare1
```

количество ссылок возросло бы до 3 (потому что все три переменные ссылались бы на один и тот же объект) и так далее.

Подсчет ссылок объекта важен, потому что, когда количество достигает нуля, Python отмечает соответствующую память как больше не используемую программой. Это называется *быть помеченным как мусор*. В Python есть сборщик мусора, который запускается для восстановления любых блоков памяти, помеченных как мусор; я рассмотрю это чуть позднее.

Стандартная библиотека Python содержит функцию getrefcount (), которая возвращает число переменных,

ссылающихся на объект. Здесь мы используем ее, чтобы увидеть подсчет ссылок после создания первого экземпляра объекта Square из класса Square:

```
oSquare1 = Square(5, 'red')
print('Reference count is', sys.getrefcount(oSquare1))
```

На экран выводится число 2. Это может оказаться неожиданным: скорее всего, вы ожидали, что число будет равно 1. Однако, как объясняет документация функции: «Возвращаемое число обычно на один выше, чем вы могли бы ожидать, потому что оно включает (временную) ссылку в качестве аргумента `getrefcount()`».

Увеличение количества ссылок

Существует несколько способов, которыми можно увеличить количество ссылок объекта.

1. Когда присваивается дополнительная переменная для ссылки на тот же самый объект:

```
oSquare2 = oSquare1
```

2. Когда объект передается в функцию и, следовательно, локальная параметр-переменная устанавливается для ссылки на объект:

```
def myFunctionOrMethod(oLocalSquareParam):
    # oLocalSquareParam теперь ссылается на то же, что и аргумент
    <body of myFunctionOrMethod>
```

```
myFunctionOrMethod(oSquare1) # вызываем функцию и передаем объект
```

3. Когда объект помещается в контейнер, такой как список или словарь:

```
myList = [oSquare1, someValue, someOtherValue]
```

Если `oSquare1` уже ссылается на объект, то после исполнения этой строки список содержит дополнительную ссылку на тот же объект Square.

Уменьшение количества ссылок

Для уменьшения количества ссылок также есть несколько способов. Чтобы их продемонстрировать, давайте создадим объект и увеличим его количество ссылок:

```
oSquare1 = Square(20, BLACK)
oSquare2 = oSquare1
myList = [oSquare1]
myFunctionOrMethod(oSquare1) # вызываем функцию и передаем объект
```

Когда запускается `myFunctionOrMethod()`, количество ссылок объекта копируется в локальную параметр-переменную для использования внутри функции. Количество ссылок этого объекта `Square` на текущий момент равно 4: все переменные объекта, одна копия внутри списка плюс параметр-переменная внутри функции. Это количество ссылок может быть уменьшено.

1. Когда переназначается любая переменная, которая ссылается на объект. Например:

```
oSquare2 = 5
```

2. Когда локальная переменная, ссылающаяся на объект, выходит из области видимости. Когда переменная создается внутри функции или метода, область видимости этой переменной ограничена данной функцией или методом. Когда заканчивается выполнение текущей функции или метода, эта переменная буквально уходит. В текущем примере, когда `myFunctionOrMethod()` завершается, локальная переменная, ссылающаяся на объект, исключается.
3. Когда объект удаляется из контейнера, такого как список, кортеж или словарь, например, с помощью:

```
myList.pop()
```

Вызов метода `remove()` списка также уменьшит количество ссылок.

4. Когда вы используете оператор `del`, чтобы явно удалить переменную, которая ссылается на объект. Это исключает переменную и сокращает количество ссылок объекта:

```
del oSquare3 # удаляем переменную
```

5. Если количество ссылок контейнера объекта (в данном случае `myList`) сойдется к нулю:

```
del myList # у myList есть элемент, который ссылается на объект
```

Если у вас есть переменная, ссылающаяся на объект, и вы хотите сохранить ее, но избавиться от ссылки на объект, можете выполнить оператор, подобный этому:

```
oSquare1 = None
```

Это сохранит имя переменной, но снизит количество ссылок объекта.

Некролог

Когда количество ссылок объекта сойдется к нулю, Python знает, что объект можно безопасно удалить. Прямо перед уничтожением объекта Python вызывает магический метод этого объекта под названием `__del__()`, чтобы сообщить объекту о его грядущей кончине.

В любом классе вы можете написать собственную версию метода `__del__()`. В нее допустимо включить любой код, который вы хотите, чтобы объект выполнял, прежде чем исчезнет навсегда. Например, ваш объект может захотеть закрыть файл, закрыть сетевое соединение и так далее.

Когда объект удаляется, Python проверяет, ссылаются ли какие-либо его переменные экземпляра на другие объекты. Если это так, количество ссылок этих объектов также уменьшается. Если в результате количество ссылок другого объекта сойдется к нулю, тогда этот объект также удаляется. Такой тип цепного или *каскадного* удаления может погрузиться на столько слоев в глубину, на сколько это необходимо. В листинге 11.2 представлен пример.

Файл: DeleteExample_Teacher_Student.py

```
# Класс Student

class Student():
    def __init__(self, name):
        self.name = name
        print('Creating Student object', self.name)
```

```

❶ def __del__(self):
    print('In the __del__ method for student:', self.name)

# Класс Teacher
class Teacher():
    def __init__(self):
        print('Creating the Teacher object')
❷ self.oStudent1 = Student('Joe')
    self.oStudent2 = Student('Sue')
    self.oStudent3 = Student('Chris')

❸ def __del__(self):
    print('In the __del__ method for Teacher')

# создаем экземпляр объекта Teacher (и, таким образом, объекты Student)
❹ oTeacher = Teacher()

# удаляем объект Teacher
❺ del oTeacher

```

Листинг 11.2. Классы, демонстрирующие методы `__del__()`

Здесь у нас есть два класса: `Student` и `Teacher`. Основной код создает экземпляр одного объекта `Teacher` ❹, а его метод `__init__()` создает три переменные экземпляра класса `Student` ❷, по одной для `Joe`, `Sue` и `Chris`. Следовательно, после запуска у объекта `Teacher` есть три переменные экземпляра объектов `Student`. Выходные данные этой первой части следующие:

```

Creating the Teacher object
Creating Student object Joe
Creating Student object Sue
Creating Student object Chris

```

Далее основной код использует оператор `del`, чтобы удалить объект `Teacher` ❺. Поскольку мы написали метод `__del__()` в классе `Teacher` ❸, вызывается этот метод *объекта* `Teacher`, который (в целях демонстрации) лишь выводит сообщение.

Когда удаляется объект `Teacher`, Python видит, что он содержит три других объекта (три объекта `Student`). Поэтому Python снижает количество ссылок в каждом из этих объектов с 1 до 0.

Как только это произошло, вызывается метод `__del__()` объектов `Student` ❶, и каждый из них выводит сообщение. Затем память, используемая всеми тремя объектами `Student`,

помечается как мусор. Выходные данные в конце программы следующие:

```
In the __del__ method for Teacher
In the __del__ method for student: Joe
In the __del__ method for student: Sue
In the __del__ method for student: Chris
```

Поскольку Python отслеживает подсчет ссылок для всех объектов, вам редко, если вообще когда-либо придется беспокоиться об управлении памятью в Python и нечасто включать метод `__del__()`. Однако вы можете использовать оператор `del`, чтобы явно указать Python удалить объекты, которые задействуют очень большие объемы памяти, когда вы уже их не используете. Например, вы можете захотеть удалить объект, загружающий большое количество записей из базы данных или много изображений, после того как вы закончили использовать его. Также нет гарантии, что Python будет вызывать метод `__del__()`, когда программа завершает работу, поэтому вам следует избегать помещения в него любого критически важного завершающего программу кода.

Сбор мусора

Когда объект удаляется либо за счет количества ссылок, сходящихся к нулю, либо с помощью явного использования оператора `del`, как программист вы должны считать объект недоступным.

Однако конкретная реализация сборщика мусора полностью зависит от Python. Для вас не важны детали алгоритма, который решает, когда выполнять текущий код сбора мусора. Он может запускаться, когда ваша программа создает экземпляры объекта и Python необходимо выделить память, либо в произвольно выбранные моменты времени, либо в конкретно установленные сроки. Алгоритм может меняться в различных версиях Python. Какой бы он ни был, Python позаботится о сборе мусора, и вам нет нужды беспокоиться о специфике.

Переменные класса

Я подробно рассказал о том, как переменные экземпляра определяются в классе и как каждый объект, для которого создан экземпляр из класса, получает собственный набор всех

переменных экземпляра. Префикс `self.` используется для идентификации каждой переменной экземпляра. Однако вы также можете создать *переменные класса* на уровне класса.

Переменная класса

Переменная, которая определяется в классе и которая принадлежит ему. У каждого класса есть только одна переменная, независимо от того, сколько экземпляров этого класса было создано.

Вы создаете переменную класса с помощью оператора присваивания, который по соглашению помещается между оператором `class` и первым оператором `def` следующим образом:

```
class MyDemoClass():
    myClassVariable = 0 # создаем переменную класса и присваиваем ей 0

    def __init__(self, <otherParameters>):
        # Прочий код
```

Поскольку переменная класса принадлежит классу, в методах класса вы будете ссылаться на нее в виде `MyDemoClass.myClassVariable`. У каждого объекта, для которого создан экземпляр из класса, есть доступ ко всем переменным класса, определенным в классе.

Существует два типичных варианта использования переменных класса: определение константы и создание счетчика.

Константы переменных класса

Вы можете создать переменную класса для использования в качестве константы следующим образом:

```
class MyClass():
    DEGREES_IN_CIRCLE = 360 # создаем константу переменной класса
```

Чтобы получить доступ к этой константе в методах класса, вы должны написать `MyClass.DEGREES_IN_CIRCLE`.

Напомним, что на самом деле в Python нет констант. Вместо этого существует соглашение среди программистов Python, что любая переменная, чье имя состоит только из букв верхнего регистра, а слова разделены нижними подчеркиваниями, должна рассматриваться как константа. То есть подобный тип переменной никогда не должен переназначаться.

Мы также можем использовать константы переменной класса, чтобы сэкономить ресурсы (память и время). Представьте, что вы пишете игру, в которой создается много экземпляров класса `SpaceShip`. Мы воплощаем картинку космического корабля и помещаем файл в папку с именем *images*. Прежде чем рассматривать переменные класса, метод `__init__()` нашего класса `SpaceShip` должен начать с создания экземпляра объекта `Image` следующим образом:

```
class SpaceShip():
    def __init__(self, window, ...):
        self.image = pygwidgets.Image(window, (0, 0),
                                       'images/ship.png')
```

Этот метод хорошо работает. Однако кодирование подобным образом означает, что не только каждому объекту, экземпляру которого был создан из класса `SpaceShip`, потребуется время для загрузки изображения, но также каждый объект занимает всю память, необходимую для представления копии того же изображения. Вместо этого мы можем сделать так, чтобы класс загружал изображение один раз, а затем каждый объект `SpaceShip` использовал одно изображение, хранящееся в классе, следующим образом:

```
class SpaceShip():
    SPACE_SHIP_IMAGE = pygame.image.load('images/ship.png')
    def __init__(self, window, ...):
        self.image = pygwidgets.Image(window, (0, 0),
                                       SpaceShip.SPACE_SHIP_IMAGE)
```

Объект `Image` (в `pygwidgets`, как применяется здесь) может использовать любой путь к изображению или уже загруженную картинку. Разрешение классу загружать изображение только *один раз* ускоряет запуск и приводит к снижению объема используемой памяти.

Переменные класса для подсчета

Второй способ использовать переменную класса — отслеживание количества объектов, для которых были созданы экземпляры из класса. В листинге 11.3 показан пример.

Файл: ClassVariable.py

```
# Класс Sample
class Sample():
    ❶ nObjects = 0 # переменная класса Sample
    def __init__(self, name):
        self.name = name
    ❷ Sample.nObjects = Sample.nObjects + 1

    def howManyObjects(self):
    ❸ print('There are', Sample.nObjects, 'Sample objects')

    def __del__(self):
    ❹ Sample.nObjects = Sample.nObjects - 1

# создаем экземпляры 4 объектов
oSample1 = Sample('A')
oSample2 = Sample('B')
oSample3 = Sample('C')
oSample4 = Sample('D')

# удаляем 1 объект
del oSample3

# определяем, сколько у нас есть
oSample1.howManyObjects()
```

Листинг 11.3. Использование переменной класса для подсчета объектов, для которых были созданы экземпляры из класса

В классе `Sample` `nObjects` — это переменная класса, потому что она определена в области видимости класса, обычно между оператором `class` и первым оператором `def` ❶. Она используется для подсчета числа существующих объектов `Sample` и инициализируется нулем. Все методы ссылаются на эту переменную, используя имя `Sample.nObjects`. Каждый раз, когда создается экземпляр объекта `Sample`, число увеличивается ❷. Когда какой-то экземпляр удаляется, число уменьшается ❹. Метод `howManyObjects()` сообщает текущее число ❸.

Основной код создает четыре объекта, затем удаляет один. При выполнении эта программа выводит:

```
There are 3 Sample objects
```

Собираем все воедино: пример программы «Шары»

В этом разделе мы возьмем некоторое количество различных концепций, которые уже рассмотрели, и совместим их в относительно простой игре — по крайней мере простой с точки зрения пользователя. Игра будет представлять некое количество шаров трех размеров, которые перемещаются в окне вверх. Цель пользователя — заставить лопнуть как можно большее количество шаров, прежде чем они вылетят за пределы верхней части окна. За маленькие шары присваивается 30 очков, за средние — 20 очков, а большие шары стоят 10 очков.

Игру можно расширить, чтобы она включала много уровней с шарами, движение которых ускоряется, но пока есть только один уровень. Размер и местоположение шара выбираются произвольно. Перед каждым раундом становится доступна кнопка **Start**, которая позволяет пользователю сыграть еще раз. На рис. 11.3 представлен скриншот игры в действии.

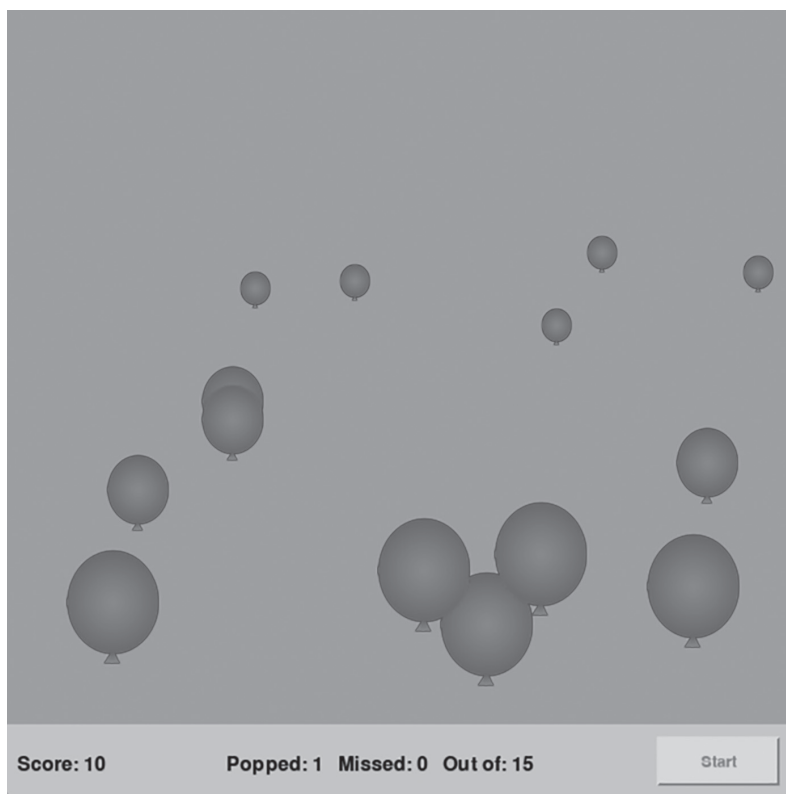


Рис. 11.3. Скриншот игры «Шары»

На рис. 11.4 показан каталог проекта для игры.

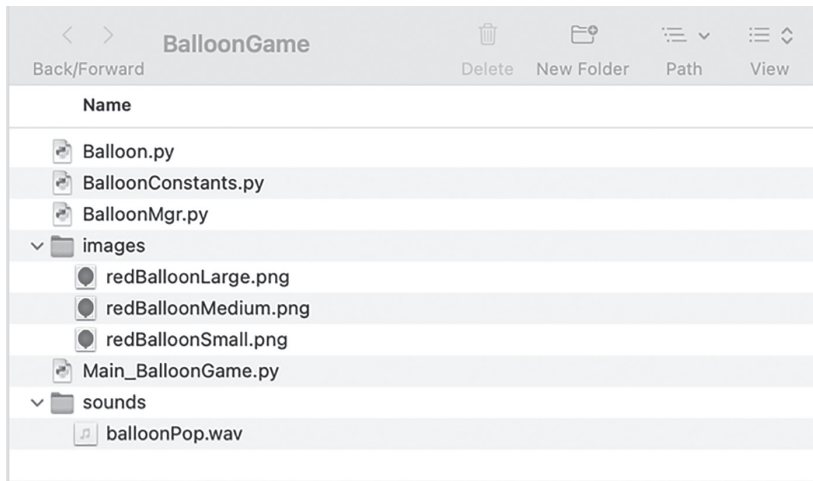


Рис. 11.4. Каталог проекта игры «Шары»

Игра реализуется четырьмя исходными файлами Python.

- ***Main_BalloonGame.py*** Основной код, выполняет основной цикл.
- ***BalloonMgr.py*** Содержит класс `BalloonMgr`, который обрабатывает все объекты `Balloon`.
- ***Balloon.py*** Содержит класс `Balloon` и подклассы `BalloonSmall`, `BalloonMedium` и `BalloonLarge`.
- ***BalloonConstants.py*** Содержит константы, используемые более чем одним файлом.

На рис. 11.5 показан схема реализации объекта.

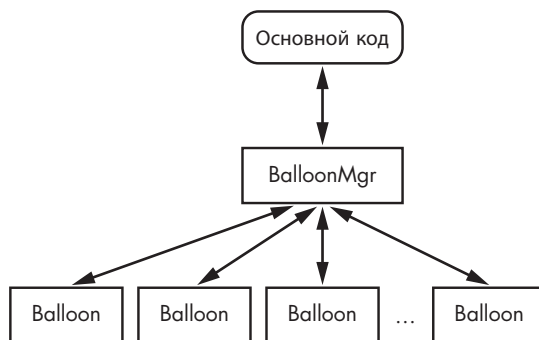


Рис. 11.5. Схема объекта игры «Шары»

Основной код (находящийся в *Main_BalloonGame.py*) создает один экземпляр объекта менеджера шаров (*oBalloonMgr*) из класса *BalloonMgr*. Менеджер шаров затем создает некоторое количество экземпляров шаров, каждый из которых выбирается произвольным образом из классов *BalloonSmall*, *BalloonMedium* и *BalloonLarge*, и сохраняет этот список объектов в переменной экземпляра. Каждый объект *Balloon* устанавливает собственную скорость, очки и произвольную начальную позицию внизу окна.

Согласно данной структуре, основной код отвечает за представление всего пользовательского интерфейса. Он взаимодействует лишь с *oBalloonMgr*. *oBalloonMgr* взаимодействует со всеми объектами *Balloon*. Следовательно, основной код даже не знает о существовании объектов *Balloon*. Он доверяет менеджеру шаров заботу о них. Давайте пройдемся по различным частям программы и посмотрим, как работает каждая.

Модуль констант

Эта структура представляет новый метод для работы с несколькими файлами Python, каждый из которых общепринято называть *модуль*. Если вы оказались в ситуации, где нескольким модулям Python требуется доступ к одним и тем же константам, хорошим решением будет создать модуль констант и импортировать его во все модули, которые используют константы. В листинге 11.4 продемонстрированы некоторые константы, определенные в *BalloonConstants.py*.

Файл: *BalloonGame/BalloonConstants.py*

```
# Константы используются более чем одним модулем Python

N_BALLOONS = 15 # количество шаров в раунде игры
BALLOON_MISSED = 'Missed' # шар вышел за пределы окна
BALLOON_MOVING = 'Balloon Moving' # шар движется
```

Листинг 11.4. Модуль констант, который импортируется другими модулями

Это всего лишь простой файл Python, который содержит константы, общие для более чем одного модуля. Основному коду нужно знать, сколько имеется шаров, чтобы отобразить это количество. Менеджеру шаров необходимо знать количество, чтобы он мог создать экземпляры правильного

количества объектов `Balloon`. Этот подход невероятно упрощает процесс изменения количества объектов `Balloon`. Если бы мы добавили уровни с различным количеством шаров, то могли бы создать список или словарь только в данном файле, а у всех остальных файлов был бы доступ к этой информации.

Другие две константы используются в каждом объекте `Balloon` в качестве индикаторов состояния по мере продвижения шара в окне вверх. Когда я перейду к обсуждению игрового процесса, вы увидите, что менеджер шаров (`oBalloonMgr`) запрашивает у каждого объекта `Balloon` его состояние и каждый объект отвечает одной из этих двух констант. Размещение общих констант в модуле и импорт этого модуля в модули, использующие константы, — простой и эффективный метод для обеспечения того, чтобы различные части программы использовали согласованные значения. Это хороший пример применения принципа «Не повторяйся» (Don't Repeat Yourself, DRY) путем определения значений в одном месте.

Код основной программы

Основной код нашего примера программы, продемонстрированный в листинге 11.5, следует 12-шаговому шаблону, который я использовал на протяжении этой книги. Он отображает очки пользователя, состояние игры и кнопку **Start** в нижней части окна, а также реагирует на щелчок пользователя по кнопке **Start**.

Файл: `BalloonGame/Main_BalloonGame.py`

```
# Основной код игры "Шары"

#1 - Импортируем пакеты
from pygame.locals import *
import pygamewidgets
import sys
import pygame
from BalloonMgr import *

#2 - Определяем константы
BLACK = (0, 0, 0)
GRAY = (200, 200, 200)
BACKGROUND_COLOR = (0, 180, 180)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 640
```

```

PANEL_HEIGHT = 60
USABLE_WINDOW_HEIGHT = WINDOW_HEIGHT - PANEL_HEIGHT
FRAMES_PER_SECOND = 30

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.
oScoreDisplay = pygame_widgets.DisplayText(window,
                                             (10, USABLE_WINDOW_HEIGHT + 25),
                                             'Score: 0', textColor=BLACK,
                                             backgroundColor=None, width=140,
                                             fontSize=24)
oStatusDisplay = pygame_widgets.DisplayText(window, (180,
                                             USABLE_WINDOW_HEIGHT + 25), '', textColor=BLACK,
                                             backgroundColor=None, width=300, fontSize=24)
oStartButton = pygame_widgets.TextButton(window, (WINDOW_WIDTH - 110,
                                             USABLE_WINDOW_HEIGHT + 10), 'Start')

#5 - Инициализируем переменные
oBalloonMgr = BalloonMgr(window, WINDOW_WIDTH, USABLE_WINDOW_HEIGHT)
playing = False ❶ # ждем, пока пользователь не нажмет кнопку Start

#6 - Бесконечный цикл
while True:
    #7 - Проверяем наличие событий и обрабатываем их
    nPointsEarned = 0
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        if playing: ❷
            oBalloonMgr.handleEvent(event)
            theScore = oBalloonMgr.getScore()
            oScoreDisplay.setValue('Score: ' + str(theScore))
        elif oStartButton.handleEvent(event): ❸
            oBalloonMgr.start()
            oScoreDisplay.setValue('Score: 0')
            playing = True
            oStartButton.disable()

    #8 - Выполняем действия "в рамках фрейма"
    if playing: ❹
        oBalloonMgr.update()
        nPopped = oBalloonMgr.getCountPopped()

```

```

nMissed = oBalloonMgr.getCountMissed()
oStatusDisplay.setValue('Popped: ' + str(nPopped) +
                        ' Missed: ' + str(nMissed) +
                        ' Out of: ' + str(N_BALLOONS))

if (nPopped + nMissed) == N_BALLOONS: ❸
    playing = False
    oStartButton.enable()

#9 - Очищаем окно
window.fill(BACKGROUND_COLOR)

#10 - Рисуем все элементы окна
if playing: ❹
    oBalloonMgr.draw()

pygame.draw.rect(window, GRAY, pygame.Rect(0,
                                             USABLE_WINDOW_HEIGHT, WINDOW_WIDTH, PANEL_HEIGHT))
oScoreDisplay.draw()
oStatusDisplay.draw()
oStartButton.draw()

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 11.5. Основной код игры «Шары»

Код основан на одной булевой переменной `playing`, установленной по умолчанию в значение `False`, чтобы позволить пользователю начать игру, нажав **Start** ❶.

Когда `playing` равно `True`, основной код вызывает метод `handleEvent()` ❷ менеджера шаров `oBalloonMgr`, чтобы обработать все события. Мы вызываем метод менеджера шаров `getScore()`, чтобы получить очки, и обновляем текст поля очков.

По завершении игры программа ждет, чтобы пользователь нажал на кнопку **Start** ❸. Когда щелчок по кнопке произошел, менеджеру шаров предлагается начать игру, а пользовательский интерфейс обновляется.

В каждом фрейме, если игра запущена, мы отправляем сообщение `update()` менеджеру шаров ❹, иницилируя его передать сообщение `update()` всем шарам. Затем запрашиваем у менеджера шаров количество оставшихся и лопнувших шаров. Мы

используем эту информацию для обновления пользовательского интерфейса.

Когда пользователь заставил лопнуть все шары или последний шар вылетает за пределы окна, мы устанавливаем переменную `playing` в значение `False` и активируем кнопку **Start** ⑤.

Код рисования очень прост ⑥. С помощью менеджера мы рисуем шары. Затем изображаем нижнюю панель с данными состояния и кнопкой **Start**.

Менеджер шаров

Менеджер шаров отвечает за отслеживание всех шаров, включая создание объектов `Balloon`, указание каждому из них нарисовать себя, указание каждому перемещаться и отслеживание количества лопнувших и пропущенных шаров. В листинге 11.6 содержится код класса `BalloonMgr`.

Файл: `BalloonGame/BalloonMgr.py`

```
# Класс BalloonMgr

import pygame
import random
from pygame.locals import *
import pygwidgets
from BalloonConstants import *
from Balloon import *

# BalloonMgr управляет списком объектов Balloon
class BalloonMgr():
    ①    def __init__(self, window, maxWidth, maxHeight):
        self.window = window
        self.maxWidth = maxWidth
        self.maxHeight = maxHeight

    ②    def start(self):
        self.balloonList = []
        self.nPopped = 0
        self.nMissed = 0

    ③    for balloonNum in range(0, N_BALLOONS):
        randomBalloonClass = random.choice((BalloonSmall,
                                             BalloonMedium,
                                             BalloonLarge))
        oBalloon = randomBalloonClass(self.window, self.maxWidth,
                                      self.maxHeight, balloonNum)
```

```

        self.balloonList.append(oBalloon)

def handleEvent(self, event):
    4 if event.type == MOUSEBUTTONDOWN:
        # Двигаемся "в обратном направлении", чтобы самый
        # верхний шар лопнул
        for oBalloon in reversed(self.balloonList):
            wasHit, nPoints = oBalloon.clickedInside(event.pos)
            if wasHit:
                if nPoints > 0: # удаляем этот шар
                    self.balloonList.remove(oBalloon)
                    self.nPopped = self.nPopped + 1
                    self.score = self.score + nPoints
                return # не нужно проверять другие

    5 def update(self):
        for oBalloon in self.balloonList:
            status = oBalloon.update()
            if status == BALLOON_MISSED:
                # Шар вышел за пределы окна, удаляем его
                self.balloonList.remove(oBalloon)
                self.nMissed = self.nMissed + 1

    6 def getScore(self):
        return self.score

    7 def getCountPopped(self):
        return self.nPopped

    8 def getCountMissed(self):
        return self.nMissed

    9 def draw(self):
        for oBalloon in self.balloonList:
            oBalloon.draw()

```

Листинг 11.6. Класс BalloonMgr

При создании экземпляра менеджеру шаров сообщается ширина и высота окна ❶, и он сохраняет эту информацию в переменных экземпляра.

Важна концепция, лежащая основе метода `start()` ❷. Его цель состоит в инициализации любой переменной экземпляра, необходимой для раунда игры, поэтому он вызывается каждый раз, когда пользователь начинает играть. Здесь `start()`

сбрасывает количество лопнувших и пропущенных шаров. Затем он проходит цикл, который создает все объекты `Balloon` (произвольно выбранные среди трех различных размеров с использованием трех различных классов), и сохраняет их в списке ❸. Каждый раз, когда метод создает объект `Balloon`, он передает в окно его ширину и высоту. (Для дальнейшего расширения всем объектам `Balloon` присваивается уникальный номер.)

При каждом прохождении основного цикла основной код вызывает метод `handleEvent()` менеджера шаров ❹. Здесь мы проверяем, щелкнул ли пользователь по какому-либо объекту `Balloon`. Если обнаруженное событие было `MOUSEDOWNEVENT`, код перебирает все объекты `Balloon`, спрашивая у каждого из них, произошел ли щелчок внутри шара. Каждый `Balloon` возвращает булево выражение, указывая, заставили ли его лопнуть, и если это так, то сообщает количество очков, которое должен получить за это пользователь. (Код настроен подобным образом для дальнейшего расширения, что обсуждается в примечании в конце этого раздела.) Затем менеджер шаров использует метод `remove()`, чтобы исключить этот `Balloon` из своего списка, увеличивает количество лопнувших шаров и обновляет очки.

В каждой итерации основного цикла основной код также вызывает метод менеджера шаров `update()` ❺, который передает этот вызов всем шарам, сообщая им: обновить себя. Каждый шар перемещается вверх по экрану в соответствии со своей настройкой скорости и возвращает свое состояние: либо он все еще движется (`BALLOON_MOVING`), либо он вышел за верхнюю часть окна (`BALLOON_MISSED`). Если шар пропустили, менеджер шаров удаляет его из своего списка и увеличивает количество пропущенных шаров.

Менеджер шаров предоставляет три метода геттера, которые позволяют основному коду получить очки ❻, количество лопнувших ❼ и пропущенных шаров ❽.

Каждый раз, проходя основной цикл, основной код вызывает метод менеджера шаров `draw()` ❾. Менеджер шаров не должен ничего рисовать сам, но он перебирает объекты `Balloon` и вызывает для каждого метод `draw()`. (Обратите внимание здесь на полиморфизм. У менеджера шаров есть метод `draw()`, и у каждого объекта `Balloon` есть метод `draw()`.)

ПРИМЕЧАНИЕ В качестве испытания попробуйте расширить эту игру, чтобы она включала новый тип (подкласс) `Balloon` — `MegaBalloon`. Пусть для того, чтобы заставить лопнуть `MegaBalloon`, потребуется три щелчка. Графика включена в загрузку этой игры.

Класс шаров и объекты

И наконец, у нас есть классы шаров. Чтобы усилить концепцию наследования из главы 10, модуль *Balloon.py* включает абстрактный базовый класс под названием `Balloon` и три подкласса: `BalloonSmall`, `BalloonMedium` и `BalloonLarge`. Менеджер шаров создает экземпляры объектов `Balloon` из этих подклассов. Каждый подкласс включает лишь метод `__init__()`, который переопределяет и затем вызывает абстрактный метод `__init__()` в классе `Balloon`. Каждое изображение мяча появляется в произвольном месте (под нижней частью окна) и будет перемещаться вверх на несколько пикселей в каждом фрейме. В листинге 11.7 показан код класса `Balloon` и его подклассов.

Файл: `BalloonGame/Balloon.py`

```
# Базовый класс Balloon и 3 подкласса

import pygame
import random
from pygame.locals import *
import pygamewidgets
from BalloonConstants import *
from abc import ABC, abstractmethod

❶ class Balloon(ABC):

    popSoundLoaded = False
    popSound = None # загружаем, когда первый шар создан

    @abstractmethod
    ❷ def __init__(self, window, maxWidth, maxHeight, ID,
                  oImage, size, nPoints, speedY):
        self.window = window
        self.ID = ID
        self.balloonImage = oImage
        self.size = size
        self.nPoints = nPoints
        self.speedY = speedY
```

```

if not Balloon.popSoundLoaded: # загружаем только в первый раз
    Balloon.popSoundLoaded = True
    Balloon.popSound = pygame.mixer.Sound('sounds/
                                                balloonPop.wav')

balloonRect = self.balloonImage.getRect()
self.width = balloonRect.width
self.height = balloonRect.height
# помещаем шар в пределах ширины окна,
# но под нижней границей
self.x = random.randrange(maxWidth - self.width)
self.y = maxHeight + random.randrange(75)
self.balloonImage.setLoc((self.x, self.y))

❸ def clickedInside(self, mousePoint):
    myRect = pygame.Rect(self.x, self.y, self.width, self.height)
    if myRect.collidepoint(mousePoint):
        Balloon.popSound.play()
        return True, self.nPoints # True здесь обозначает, что
                                   # его лопнули
    else:
        return False, 0 # не лопнули, нет очков

❹ def update(self):
    self.y = self.y - self.speedY # обновляем позицию
                                   # у по скорости
    self.balloonImage.setLoc((self.x, self.y))
    if self.y < -self.height: # вышел за верхнюю часть окна
        return BALLOON_MISSED
    else:
        return BALLOON_MOVING

❺ def draw(self):
    self.balloonImage.draw()

❻ def __del__(self):
    print(self.size, 'Balloon', self.ID, 'is going away')

❼ class BalloonSmall(Balloon):
    balloonImage = pygame.image.load('images/redBalloonSmall.png')
    def __init__(self, window, maxWidth, maxHeight, ID):
        oImage = pygwidgets.Image(window, (0, 0),
                                    BalloonSmall.balloonImage)
        super().__init__(window, maxWidth, maxHeight, ID,
                          oImage, 'Small', 30, 3.1)

❽ class BalloonMedium(Balloon):
    balloonImage = pygame.image.load('images/redBalloonMedium.png')

```

```

def __init__(self, window, maxWidth, maxHeight, ID):
    oImage = pygame.image.load('images/redBalloonMedium.png')
    BalloonMedium.balloonImage)
    super().__init__(window, maxWidth, maxHeight, ID,
                     oImage, 'Medium', 20, 2.2)

9 class BalloonLarge(Balloon):
    balloonImage = pygame.image.load('images/redBalloonLarge.png')
    def __init__(self, window, maxWidth, maxHeight, ID):
        oImage = pygame.image.load('images/redBalloonLarge.png')
        BalloonLarge.balloonImage)
        super().__init__(window, maxWidth, maxHeight, ID,
                         oImage, 'Large', 10, 1.5)

```

Листинг 11.7. Классы Balloon

Класс Balloon — это абстрактный класс ❶, поэтому BalloonMgr создает экземпляры объектов (произвольным образом) из классов BalloonSmall ❷, BalloonMedium ❸ и BalloonLarge ❹. Каждый из этих классов создает объект pygame.image.load, затем вызывает метод __init__() в базовом классе Balloon. Мы различаем шары с помощью аргументов, представляющих изображение, размер, количество очков и скорость.

Метод __init__() в классе Balloon ❷ сохраняет информацию о каждом шаре в переменных экземпляра. Мы получаем прямоугольное изображение шара и запоминаем его ширину и высоту. Устанавливаем произвольную горизонтальную позицию, которая обеспечит полное отображение картинки шара внутри окна.

Всякий раз, когда происходит MOUSEBUTTONDOWN, менеджер шаров перебирает объекты Balloon и вызывает для каждого метод clickedInside() ❸. Здесь код проверяет, было ли обнаружено MOUSEBUTTONDOWN внутри текущего шара. Если это так, Balloon воспроизводит лопающийся звук и возвращает булево выражение, чтобы сообщить, что по шару щелкнули, а также количество очков, которое дает этот шар. Если щелчка не было, он возвращает False и ноль.

В каждом фрейме менеджер шаров вызывает метод update() для каждого Balloon ❹, который обновляет позицию у этого Balloon путем вычитания его собственной скорости, чтобы переместиться выше в окне. После изменения позиции

метод `update()` возвращает либо `BALLOON_MISSED` (если он полностью вышел за пределы верхнего края окна), либо `BALLOON_MOVING` (чтобы указать, что он все еще в игре).

Метод `draw()` просто рисует изображение шара в соответствующем местоположении (x, y) ❸. Хотя позиция y сохраняется как значение с плавающей точкой, `pygame` автоматически преобразовывает его в целое число для размещения пикселей в окне.

Последний метод `__del__()` ❹ был добавлен для отладки и будущей разработки. Каждый раз, когда менеджер шаров удаляет шар, вызывается метод `__del__()` этого `Balloon`. Для демонстрации он пока просто выводит сообщение, которое отображает размер шара и идентификационный номер.

Когда программа запущена и пользователь начинает щелкать по шарам, мы видим следующие выходные данные в окне оболочки или консоли:

```
Small Balloon 2 is going away
Small Balloon 8 is going away
Small Balloon 3 is going away
Small Balloon 7 is going away
Small Balloon 9 is going away
Small Balloon 12 is going away
Small Balloon 11 is going away
Small Balloon 6 is going away
Medium Balloon 14 is going away
Large Balloon 1 is going away
Medium Balloon 10 is going away
Medium Balloon 13 is going away
Medium Balloon 0 is going away
Medium Balloon 4 is going away
Large Balloon 5 is going away
```

Когда игра завершена, программа ждет, пока пользователь щелкнет по кнопке **Start**. После щелчка по кнопке менеджер шаров воссоздает список объектов `Balloon` и сбрасывает его переменные экземпляра, и игра начинается заново.

Управляем памятью: слоты

Как мы уже обсуждали, когда вы создаете экземпляр объекта, Python должен выделить место для переменных экземпляра, определенных в классе. По умолчанию он делает это с помощью словаря со специальным именем `__dict__`. Чтобы увидеть это

в действии, вы можете добавить такую строку в конце метода `__init__()` любого класса:

```
print(self.__dict__)
```

Словарь — прекрасный способ представить все переменные экземпляра, потому что он динамичен, то есть может увеличиваться каждый раз, когда Python встречает переменную экземпляра, которую он до этого не видел в классе. Хотя я рекомендую инициализировать все переменные экземпляра в вашем методе `__init__()`, фактически вы можете определить переменные экземпляра в любом методе, и они будут добавлены, когда метод выполняется в первый раз. Несмотря на то что лично я считаю следующее плохой идеей, это демонстрирует возможность динамически добавлять переменные экземпляра в объект:

```
myObject = MyClass()
myObject.someInstanceVariable = 5
```

Чтобы обеспечить такую возможность, словари обычно реализуются начиная с достаточного количества свободного места для представления некоторого количества переменных экземпляра (точное число — это внутренняя деталь Python). Когда встречается новая переменная экземпляра, она добавляется в словарь. Если в нем заканчивается место, Python добавляет еще. Обычно это работает хорошо, и программисты не сталкиваются с проблемами реализации.

Однако представим, что у вас есть следующий класс с двумя переменными экземпляра, созданными в методе `__init__()`, и вы знаете, что вам не потребуется добавлять еще какие-либо переменные экземпляра:

```
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y
    # Дополнительные методы
```

Теперь предположим, что вам нужно создать экземпляры очень большого количества (сотен тысяч или даже миллионов) объектов из этого класса. В подобном случае это может в общей сложности привести к большому количеству потраченного впустую пространства памяти (RAM).

Чтобы справиться с этой потенциальной напрасной тратой, Python предоставляет другой подход, известный как *слоты*, для представления переменных экземпляра. Идея состоит в том, что вы можете сообщить Python имена всех переменных экземпляра заранее и Python будет использовать структуру данных, которая выделяет ровно столько места, сколько необходимо именно для этих переменных экземпляра. Чтобы использовать слоты, вам необходимо включить специальную переменную класса `__slots__`, чтобы определить список переменных:

```
__slots__ = [<instanceVar1>, <instanceVar2>, ... <instanceVarN>]
```

Вот как будет выглядеть измененная версия примера нашего класса:

```
class PointWithSlots():
    # Определяем слоты только для двух переменных
    # экземпляра
    __slots__ = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y
        print(x, y)
```

Эти два класса будут работать идентично, но объекты, для которых созданы экземпляры из `PointWithSlots`, займут значительно меньший объем памяти. Чтобы продемонстрировать разницу, мы добавим эту строку в конец метода `__init__()` обоих классов:

```
# Попробовать создать дополнительную переменную экземпляра
self.color = 'black'
```

Теперь, когда мы пытаемся создать экземпляр объекта из обоих классов, у класса `Point` не возникнет проблем с добавлением еще одной переменной экземпляра, но класс `PointWithSlots` завершается сбоем со следующей ошибкой:

```
AttributeError: 'PointWithSlots' object has no attribute 'color'
```

Использование слотов невероятно эффективно с точки зрения памяти, но при этом утрачиваются динамические переменные экземпляра. Если вы имеете дело с очень большим

количеством объектов класса, такой компромисс может оказаться вполне стоящим.

Выводы

Эта глава сосредоточена на некоторых концепциях, которые не вписались в предыдущие главы. Во-первых, я рассмотрел обстоятельства, при которых вы можете захотеть удалить объект. Мы изучили подсчет ссылок и то, как они отслеживают, сколько переменных ссылается на один и тот же объект, что привело к обсуждению жизненного цикла объекта и сбора мусора. Когда количество ссылок сходится к нулю, объект становится доступным для сбора мусора. Если у класса есть метод `__del__()`, тогда любой созданный из класса объект может использовать метод `__del__()` для очистки, которую он, возможно, захочет выполнить.

Далее я рассмотрел, чем переменные класса отличаются от переменных экземпляра. Каждый объект, для которого создается экземпляр из класса, получает собственный набор всех переменных экземпляра в классе. Однако каждая переменная класса существует лишь в единственном экземпляре, и к ней есть доступ у всех объектов, созданных из класса. Переменные класса часто используются как константы или счетчики или для загрузки чего-то большого и предоставления доступа ко всем объектам, экземпляры которых созданы из класса.

Чтобы свести воедино методы и концепции, мы создали игру с лопающимися шарами и организовали ее очень эффективно. У нас был один файл, содержащий только константы, используемые другими файлами. Основной код состоял из основного цикла и отображения состояния, а менеджер шаров хорошо выполнял свою работу по управлению объектами. Подобное разделение труда позволяет разбить игру на более мелкие логические фрагменты. Роль каждой части хорошо определена, что делает всю программу легче управляемой.

И наконец, я объяснил, как методы вызова слотов обеспечивают эффективное с точки зрения памяти представление переменных экземпляра.

ЧАСТЬ IV

ИСПОЛЬЗОВАНИЕ ООП В РАЗРАБОТКЕ ИГР

В этой части книги мы создадим несколько образцов игр с помощью `pygame`. Также я познакомлю вас с модулем `pygame`, который включает в себя некоторое количество классов и функций, полезных при создании игровых программ.

Глава 12 возвращается к игре «Больше-меньше» из главы 1. Мы создадим версию игры с графическим интерфейсом пользователя, и я представлю карточные классы `Deck` и `Card`, которые можно многократно использовать в любой программе карточной игры.

Глава 13 сосредоточена на таймерах. Мы создадим несколько различных классов таймеров, которые позволят вашей программе продолжать работать, одновременно проверяя определенный лимит времени.

В главе 14 обсуждаются различные классы анимации, которые можно использовать для отображения последовательности изображений. Это позволит вам с легкостью создавать более художественные игры и программы.

Глава 15 знакомит с походом к созданию программы, которая содержит много сцен, таких как начальная, игровая и сцена окончания игры. Я продемонстрирую класс `SceneManager`, спроектированный для управления любым количеством созданных программистом сцен, и мы используем его для построения игры «камень-ножницы-бумага».

Глава 16 демонстрирует, как отображать различные типы диалоговых окон и реагировать на них. Затем вы используете все, что узнали, для построения полнофункциональной анимированной игры.

Глава 17 знакомит с концепцией шаблонов проектирования, используя в качестве примера модель, представление и шаблон контроллера. Затем идет краткое резюме книги.

12

КАРТОЧНЫЕ ИГРЫ



В оставшихся главах мы создадим несколько демоверсий программ, используя `pygame` и `pygamewidgets`. Каждая программа будет представлять один или более многократно используемых классов и показывать, как они могут быть использованы в типовом проекте.

В главе 1 я представил текстовую карточную игру «Больше-меньше». Здесь же мы создадим GUI-версию игры, как показано на рис. 12.1.

Кратко повторим правила игры: мы начинаем с семи карт рубашкой вниз и одной карты лицевой стороной вверх. Игрок угадывает, будет ли следующая открытая карта больше или меньше, чем последняя видимая карта, нажимая **Lower** или **Higher**. Когда игра завершена, пользователь щелкает по кнопке **New Game**, чтобы начать новый раунд игры. Игрок начинает со 100 очков, получает 15 очков за правильный ответ и теряет 10 — за неверный.



Рис. 12.1. Интерфейс пользователя для игры «Больше-меньше»

Класс Card

В исходной текстовой версии игры код, связанный с колодой карт, нельзя было с легкостью многократно использовать в других проектах. Для решения этой проблемы здесь мы создадим класс `Deck` с высокой степенью многократного использования, который управляет картами из класса `Card`.

Чтобы представить карту в `pygame`, нам необходимо сохранить следующие данные в переменных экземпляра для каждого объекта `Card`:

- ранг (туз, 2, 3, ...10, валет, королева, король);
- масть (трефы, черви, бубны и пики);
- значение (1, 2, 3, ...12, 13);
- имя (создается с использованием ранга и масти, например `7 of clubs`);
- изображение рубашки карты (одно изображение, общее для всех объектов `Card`);
- изображение лицевой стороны карты (уникальное изображение для каждого объекта `Card`).

Каждая карта должна выполнять следующие действия, для которых мы создадим методы:

- отметить себя как скрытую (рубашкой вверх);
- отметить себя как открытую (лицом вверх);

- вернуть свое имя;
- вернуть свое значение;
- установить и получить свои координаты в окне;
- нарисовать себя (либо открытое изображение, либо скрытое изображение).

Хотя следующие действия карты не используются в игре «Больше-меньше», мы добавим и их тоже на случай, если они понадобятся в какой-то другой игре:

- вернуть свой ранг;
- вернуть свою масть.

В листинге 12.1 показан код класса `Card`.

Файл: `HigherOrLower/Card.py`

```
# Класс Card

import pygame
import pygwidgets

class Card():

    ❶ BACK_OF_CARD_IMAGE = pygame.image.load('images/BackOfCard.png')

    ❷ def __init__(self, window, rank, suit, value):
        self.window = window
        self.rank = rank
        self.suit = suit
        self.cardName = rank + ' of ' + suit
        self.value = value
    ❸ fileName = 'images/' + self.cardName + '.png'
        # Настраиваем некоторое начальное местоположение; для
        # изменения использовать setLoc ниже
    ❹ self.images = pygwidgets.ImageCollection(window, (0, 0),
        {'front': fileName,
         'back': Card.BACK_OF_CARD_IMAGE}, 'back')

    ❺ def conceal(self):
        self.images.replace('back')

    ❻ def reveal(self):
        self.images.replace('front')

    ❼ def getName(self):
        return self.cardName
```

```

def getValue(self):
    return self.value

def getSuit(self):
    return self.suit

def getRank(self):
    return self.rank

8 def setLoc(self, loc): # вызываем метод setLoc ImageCollection
    self.images.setLoc(loc)

9 def getLoc(self): # получаем местоположение от ImageCollection
    loc = self.images.getLoc()
    return loc

10 def draw(self):
    self.images.draw()

```

Листинг 12.1. Класс Card

Класс Card предполагает, что файлы изображений для всех 52 карт плюс изображения рубашки всех карт доступны в папке с именем *images* внутри каталога проекта. Если вы загрузите файлы, относящиеся к этой главе, то увидите, что папка *images* содержит полный набор *.png*-файлов. Файлы доступны в репозитории GitHub по ссылке <https://github.com/IrvKalb/Object-Oriented-Python-Code/>.

Класс загружает изображение рубашки карт один раз и сохраняет его в переменной класса ❶. Это изображение доступно всем объектам Card.

Вызываемый для каждой карты метод `__init__()` ❷ начинается с сохранения окна, создания и сохранения имен карт и сохранения их ранга, значения и масти в переменных экземпляра. Затем он выстраивает путь к файлу в папке *images*, которая содержит изображение для этой конкретной карты ❸. Например, если рангом карты является туз, а мастью — пики, мы выстраиваем путь *images/Ace of Spades.png*. Мы используем объект *ImageCollection*, чтобы запомнить путь к лицевому и обратному изображению ❹; будем использовать 'back', чтобы сообщить, что хотим отобразить рубашку карты в качестве стартового изображения.

Метод `conceal()` ❺ говорит *ImageCollection* установить рубашку карты в качестве текущего изображения. Метод

`reveal()` ⑥ говорит `ImageCollection` установить лицевую сторону карты в качестве текущего изображения.

Методы `getName()`, `getValue()`, `getSuit()` и `getRank()` ⑦ — это методы геттера, которые позволяют вызывающему извлекать имя, значение, масть и ранг заданной карты.

Метод `setLoc()` устанавливает новые координаты для карты ⑧, а `getLoc()` извлекает текущие ⑨. Координаты хранятся в `ImageCollection`. И наконец, `draw()` ⑩ рисует изображение карты в окне. Если точнее, он приказывает `ImageCollection` нарисовать текущее указанное изображение в сохраненных координатах.

Класс Deck

Объект `Deck` является классическим примером объекта диспетчера объектов. Его работа заключается в создании и управлении 52 объектами `Card`. В листинге 12.2 содержится код нашего класса `Deck`.

Файл: `HigherOrLower/Deck.py`

```
# Класс Deck

import random
from Card import *

class Deck():
    ❶ SUIT_TUPLE = ('Diamonds', 'Clubs', 'Hearts', 'Spades')
    # Этот словарь сопоставляет каждый ранг карты со значением
    # для стандартной колоды
    STANDARD_DICT = {'Ace':1, '2':2, '3':3, '4':4, '5':5,
                     '6':6, '7':7, '8':8, '9':9, '10':10,
                     'Jack':11, 'Queen':12, 'King':13}

    ❷ def __init__(self, window, rankValueDict=STANDARD_DICT):
        # по умолчанию значение rankValueDict равно STANDARD_DICT,
        # но вы можете вызвать его с другим словарем, например
        # для блек-джека
        self.startingDeckList = []
        self.playingDeckList = []
        for suit in Deck.SUIT_TUPLE:
            ❸ for rank, value in rankValueDict.items():
                oCard = Card(window, rank, suit, value)
                self.startingDeckList.append(oCard)
```

```

        self.shuffle()

❷ def shuffle(self):
    # Копировать начальную колоду и сохранить ее в списке
    # игровой колоды
    self.playingDeckList = self.startingDeckList.copy()
    for oCard in self.playingDeckList:
        oCard.conceal()
    random.shuffle(self.playingDeckList)

❸ def getCard(self):
    if len(self.playingDeckList) == 0:
        raise IndexError('No more cards')
    # Вытолкнуть одну карту из колоды и вернуть ее
    oCard = self.playingDeckList.pop()
    return oCard

❹ def returnCardToDeck(self, oCard):
    # Вернуть карту обратно в колоду
    self.deckList.insert(0, oCard)

```

Листинг 12.2. Класс Card

Мы начинаем класс Deck с создания нескольких переменных класса ❶, которые будем использовать для создания 52 карт с подходящими мастями и значениями. Используется только четыре метода.

Методу `__init__()` ❷ передаем ссылку на окно и дополнительный словарь, который сопоставляет ранги с их значениями. Если ничего не передается, мы используем словарь для значений стандартной колоды. Создаем колоды из 52 карт, сохраненных в `self.startingDeckList`, перебирая все масти, затем перебирая все ранги и значения карт. Во внутреннем цикле `for` ❸ используем вызов метода словаря `items()`, который позволяет нам с легкостью получить ключ и значение (здесь `rank` и `value`) в одном операторе. Каждый раз во время прохождения внутреннего цикла мы создаем экземпляр объекта Card, передаем ранг, масть и значение новой карты. Добавляем каждый объект Card в список `self.startingDeckList`, чтобы создать полную колоду карт.

Завершающий шаг состоит в вызове метода `shuffle()` ❹ для рандомизации колоды. Цель этого метода может показаться очевидной: перетасовать колоду. Однако он выполняет дополнительный небольшой трюк. Метод `__init__()` создает

`self.startingDeckList`, и эту работу необходимо выполнить лишь раз. Таким образом, каждый раз, когда мы перетасовываем колоду, вместо воссоздания всех объектов `Card` мы делаем копию начального списка колоды, сохраняем его в `self.playingDeckList` и перетасовываем ее. Копия — это именно то, что будет использоваться и чем мы будем управлять во время игры. С таким подходом мы можем удалять карты из `self.playingDeckList` и не беспокоиться о добавлении их в дальнейшем обратно в колоду или о перезагрузке карт. У двух списков `self.startingDeckList` и `self.playingDeckList` общие ссылки на одни и те же 52 объекта `Card`.

Обратите внимание, что, когда мы вызываем `shuffle()` для последующих запусков игры, некоторые объекты `Card` могут быть в «открытом» состоянии. Поэтому, прежде чем продолжить, мы перебираем всю колоду и вызываем метод `conceal()` для каждой карты, чтобы все карты изначально появлялись рубашкой вверх. Метод `shuffle()` завершается рандомизацией карт в игровой колоде с помощью `random.shuffle()`.

Метод `getCard()` ❶ извлекает карту из колоды. Он сначала проверяет, пуста ли колода, и, если это так, вызывает исключение. В противном случае, поскольку колода уже перетасована, он вытаскивает карту из колоды и возвращает ее вызывающему.

Вместе `Deck` и `Card` представляют комбинацию классов с высокой степенью многократного использования, которую можно применять в большинстве карточных игр. Игра «Больше-меньше» использует только восемь карт в раунде и перетасовывает всю колоду в начале каждой игры. Следовательно, в этой игре у объекта `Deck` не могут закончиться карты. Для игры, в которой необходимо знать, закончились ли карты в колоде, вы можете создать блок `try` вокруг вызова `getCard()` и использовать выражение `except`, чтобы обработать исключение. Здесь выбор остается за вами.

Хотя в этой игре он не используется, метод `returnCardToDeck()` ❷ позволяет вернуть карту в колоду.

Игра «Больше-меньше»

Код для реальной игры достаточно прост: основной код реализует основной цикл, а объект `Game` содержит логику для самой игры.

Основная программа

В листинге 12.3 представлена основная программа, которая делает начальные установки и содержит основной цикл. Она также создает объект `Game`, который выполняет игру.

Файл: `HigherOrLower/Main_HigherOrLower.py`

```
# "Больше-меньше" - версия pygame
# Основная программа

--- пропуск ---
#4 - Загружаем элементы: изображения, звуки и т. д.
❶ background = pygame.image.load('images/background.png')
newGameButton = pygame.Button(window, (20, 530),
                                'New Game', width=100, height=45)
higherButton = pygame.Button(window, (540, 520),
                               'Higher', width=120, height=55)
lowerButton = pygame.Button(window, (340, 520),
                              'Lower', width=120, height=55)
quitButton = pygame.Button(window, (880, 530),
                             'Quit', width=100, height=45)

#5 - Инициализируем переменные
❷ oGame = Game(window)

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        if ((event.type == QUIT) or
            ((event.type == KEYDOWN) and (event.key == K_ESCAPE)) or
            (quitButton.handleClick(event))):
            pygame.quit()
            sys.exit()

    ❸ if newGameButton.handleClick(event):
        oGame.reset()
        lowerButton.enable()
        higherButton.enable()

    if higherButton.handleClick(event):
        gameOver = oGame.hitHigherOrLower(HIGHER)
        if gameOver:
            higherButton.disable()
            lowerButton.disable()
```

```

        if lowerButton.handleEvent(event):
            gameOver = oGame.hitHigherOrLower(LOWER)
            if gameOver:
                higherButton.disable()
                lowerButton.disable()

#8 - Выполняем действия "в рамках фрейма"

#9 - Очищаем окно, прежде чем нарисовать его заново
4 background.draw()

#10 - Рисуем элементы окна
# рисуем игру
5 oGame.draw()
# рисуем остальные компоненты интерфейса пользователя
newGameButton.draw()
higherButton.draw()
lowerButton.draw()
quitButton.draw()

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND)

```

Листинг 12.3. Основной код игры «Больше-меньше»

Основная программа загружает фоновое изображение и рисует четыре кнопки ❶, затем создает экземпляры объекта Game ❷.

В основном цикле мы смотрим, была ли нажата какая-либо из кнопок ❸, и, когда нажатие произошло, вызываем соответствующий метод в объекте Game.

В нижней части цикла рисуем элементы окна ❹, начиная с фона. Самое главное, мы вызываем метод draw() объекта Game ❺. Как вы увидите, объект Game передает это сообщение каждому объекту Card. И наконец, рисуем все четыре кнопки.

Объект Game

Объект Game обрабатывает логику реальной игры. Листинг 12.4 содержит код класса Game.

Файл: HigherOrLower/Game.py

```
# Класс Game

import pygwidgets
from Constants import *
from Deck import *
from Card import *

class Game():
    CARD_OFFSET = 110
    CARDS_TOP = 300
    CARDS_LEFT = 75
    NCARDS = 8
    POINTS_CORRECT = 15
    POINTS_INCORRECT = 10

    def __init__(self, window): ❶
        self.window = window
        self.oDeck = Deck(self.window)
        self.score = 100
        self.scoreText = pygwidgets.DisplayText(window, (450, 164),
                                                    'Score: ' + str(self.score),
                                                    fontSize=36, textColor=WHITE,
                                                    justified='right')

        self.messageText = pygwidgets.DisplayText(window, (50, 460), '',
                                                    width=900, justified='center',
                                                    fontSize=36, textColor=WHITE)

        self.loserSound = pygame.mixer.Sound("sounds/loser.wav")
        self.winnerSound = pygame.mixer.Sound("sounds/ding.wav")
        self.cardShuffleSound = pygame.mixer.Sound("sounds/
                                                    cardShuffle.wav")

        self.cardXPositionsList = []
        thisLeft = Game.CARDS_LEFT
        # вычисляем координаты x для всех карт, один раз
        for cardNum in range(Game.NCARDS):
            self.cardXPositionsList.append(thisLeft)
            thisLeft = thisLeft + Game.CARD_OFFSET

        self.reset() # начинаем раунд игры

    def reset(self): ❷ # этот метод вызывается, когда начинается
                        # новый раунд
        self.cardShuffleSound.play()
        self.cardList = []
```

```

self.oDeck.shuffle()
for cardIndex in range(0, Game.NCARDS): # раздаем карты
    oCard = self.oDeck.getCard()
    self.cardList.append(oCard)
    thisXPosition = self.cardXPositionsList[cardIndex]
    oCard.setLoc((thisXPosition, Game.CARDS_TOP))

self.showCard(0)
self.cardNumber = 0
self.currentCardName, self.currentCardValue = \
    self.getCardNameAndValue(self.cardNumber)

self.messageText.setValue('Starting card is
                            ' + self.currentCardName +
                            '. Will the next card be higher or lower?')

def getCardNameAndValue(self, index):
    oCard = self.cardList[index]
    theName = oCard.getName()
    theValue = oCard.getValue()
    return theName, theValue

def showCard(self, index):
    oCard = self.cardList[index]
    oCard.reveal()

def hitHigherOrLower(self, higherOrLower): ❸
    self.cardNumber = self.cardNumber + 1
    self.showCard(self.cardNumber)
    nextCardName, nextCardValue =
        self.getCardNameAndValue(self.cardNumber)

    if higherOrLower == HIGHER:
        if nextCardValue > self.currentCardValue:
            self.score = self.score + Game.POINTS_CORRECT
            self.messageText.setValue('Yes, the ' + nextCardName +
                                      ' was higher')
            self.winnerSound.play()
        else:
            self.score = self.score - Game.POINTS_INCORRECT
            self.messageText.setValue('No, the ' + nextCardName +
                                      ' was not higher')
            self.loserSound.play()
    else: # пользователь нажал на кнопку Lower
        if nextCardValue < self.currentCardValue:
            self.score = self.score + Game.POINTS_CORRECT
            self.messageText.setValue('Yes, the ' + nextCardName +
                                      ' was lower')

```

```

        self.winnerSound.play()
    else:
        self.score = self.score - Game.POINTS_INCORRECT
        self.messageText.setValue('No, the ' + nextCardName +
                                   ' was not lower')

        self.loserSound.play()

self.scoreText.setValue('Score: ' + str(self.score))

self.currentCardValue = nextCardValue # настраиваем для
                                       # следующей карты

done = (self.cardNumber == (Game.NCARDS - 1)) # мы добрались
                                              # до последней карты?

return done

def draw(self): ❹
    # рисуем карты
    for oCard in self.cardList:
        oCard.draw()

    self.scoreText.draw()
    self.messageText.draw()

```

Листинг 12.4. Объект Game, который выполняет игру

В методе `__init__()` ❶ мы инициализируем несколько переменных экземпляра, которые необходимо установить только один раз. Создаем объект `Deck`, устанавливаем начальные очки и создаем объект `DisplayText` для отображения очков и результата каждого хода. Мы также загружаем ряд звуковых файлов для использования во время игры. И наконец, вызываем метод `reset()` ❷, который содержит любой необходимый для одной игры код, то есть: перетасовать колоду, воспроизвести звук перетасовки, раздать восемь карт, отобразить их на предварительно рассчитанных позициях и отобразить первую карту лицевой стороной вверх.

Когда пользователь нажимает на кнопку **Higher** или **Lower**, основной код вызывает `hitHigherOrLower()` ❸, который переворачивает следующую карту, сравнивает значение с предыдущей открытой картой и начисляет или вычитает очки.

Метод `draw()` ❹ перебирает все карты текущей игры, говоря каждой нарисовать себя (вызывая метод `draw()` каждого объекта `Card`). Затем он рисует текст очков и отклик для текущего хода.

Тестируем с помощью `__name__`

Когда вы пишете класс, всегда полезно сочинить некоторый тестовый код, чтобы убедиться, что создаваемый из класса объект будет работать верно. Напоминаю, что любой файл, содержащий код Python, называется *модулем*. Стандартная практика — написать один или более классов в модуле, затем использовать оператор `import`, чтобы перенести этот модуль в какой-либо другой. Когда вы пишете модуль, который содержит класс (или классы), можете добавить некоторый тестовый код, который должен выполняться, *только* когда выполняется модуль в качестве основной программы, и не выполняться в типичном случае, когда модуль импортируется другим файлом Python.

В проекте с несколькими модулями Python обычно есть один основной модуль и несколько других. Когда программа выполняется, Python создает специальную переменную с именем `__name__` в каждом модуле. В любом модуле, которому было в первую очередь предоставлено управление, Python устанавливает значение `__name__` в строку `'__main__'`. Следовательно, вы можете написать код, чтобы проверить значение `__name__` и запустить некоторый тестовый код, только если модуль выполняется как основная программа.

В качестве примера я буду использовать класс `Deck`. В конце *Deck.py*, после кода класса, я добавил этот код, чтобы создать экземпляр класса `Deck` и вывести карты, которые он создает:

```
--- пропущенный код класса Deck ---
if __name__ == '__main__':
    # Основной код для тестирования класса Deck

    import pygame

    # Константы
    WINDOW_WIDTH = 100
    WINDOW_HEIGHT = 100

    pygame.init()
    window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))

    oDeck = Deck(window)
    for i in range(1, 53):
        oCard = oDeck.getCard()
        print('Name: ', oCard.getName(), ' Value:', oCard.getValue())
```

Так можно проверить, выполняется ли файл *Deck.py* как основная программа. В типичном случае, когда класс *Deck* импортируется каким-то другим модулем, значение `__name__` будет `'Deck'`, потому этот код не делает ничего. Но если мы выполняем *Deck.py* в качестве основной программы, только в целях тестирования, Python устанавливает значение `__name__` в `'__main__'`, и выполняется этот тестовый код.

В нем мы выстраиваем минимальную программу *pygame*, которая создает экземпляр класса *Deck*, затем выводит имена и значения всех 52 карт. Выходные данные выполняемого в качестве основной программы *Deck.py* выглядят следующим образом в окне оболочки или консоли:

```
Name: 4 of Spades    Value: 4
Name: 4 of Diamonds Value: 4
Name: Jack of Hearts Value: 11
Name: 8 of Spades   Value: 8
Name: 10 of Diamonds Value: 10
Name: 3 of Clubs    Value: 3
Name: Jack of Diamonds Value: 11
Name: 9 of Spades   Value: 9
Name: Ace of Diamonds Value: 1
Name: 2 of Clubs    Value: 2
Name: 7 of Clubs    Value: 7
Name: 4 of Clubs    Value: 4
Name: 8 of Hearts   Value: 8
Name: 3 of Diamonds Value: 3
Name: 7 of Spades   Value: 7
Name: 7 of Diamonds Value: 7
Name: King of Diamonds Value: 13
Name: 10 of Spades  Value: 10
Name: Ace of Hearts Value: 1
Name: 8 of Diamonds Value: 8
Name: Queen of Diamonds Value: 12
...
```

Подобный код полезен для проверки того, работает ли класс в целом так, как мы ожидали, без необходимости иметь дело с более объемной основной программой для создания его экземпляра. Он предоставляет быстрый способ убедиться, что класс не нарушен. В зависимости от наших потребностей мы можем пойти дальше и добавить какой-то пример кода для демонстрации типичных вызовов методов класса.

Другие карточные игры

Существует много карточных игр, которые задействуют стандартную колоду из 52 карт. Мы могли бы использовать классы `Deck` и `Card`, чтобы создать такие игры, как бридж, черви, джингамми и многие пасьянсы. Однако встречаются игры, которые используют другие карточные значения или другое количество карт. Давайте рассмотрим несколько примеров и поглядим, как можно адаптировать наши классы для этих случаев.

Колода для блек-джека

Хотя в блек-джеке, также известном как 21, используются те же карты, что и в стандартной колоде, их значения другие: десятка, валет, дама и король — все стоят по 10 очков. Метод `__init__()` класса `Deck` начинается следующим образом:

```
def __init__(self, window, rankValueDict=STANDARD_DICT):
```

Чтобы создать колоду для блек-джека, необходимо предоставить словарь для `rankValueDict` следующим образом:

```
blackJackDict = {'Ace':1, '2':2, '3':3, '4':4, '5':5,
                 '6':6, '7':7, '8': 8, '9':9, '10':10,
                 'Jack':10, 'Queen':10, 'King':10}
oBlackjackDeck = Deck(window, rankValueDict=blackJackDict)
```

Как только вы создали `oBlackjackDeck`, можете вызвать без изменений существующие методы `shuffle()` и `getCard()`. В реализации блек-джека вам также придется иметь дело с тем фактом, что значение туза может быть 1 или 11. Но это упражнение для самостоятельного выполнения.

Игры с необычными колодами

Существует несколько карточных игр, которые не используют стандартную колоду из 52 карт. В канасте требуется как минимум две колоды с джокерами, в общей сложности 108 карт. Колода пиночля состоит из двух девяток, десятков, валетов, дам, королей и тузов каждой масти, итого 48 карт.

Для подобных игр вы все еще можете использовать класс `Deck`, но понадобится создать подкласс с `Deck` в качестве базового класса. Новому классу `CanastaDeck` или `PinochleDeck`

понадобится собственный метод `__init__()`, который создает колоду в виде списка, состоящего из соответствующих объектов `Card`. Однако методы `shuffle()` и `getCard()` могут быть унаследованы из класса `Deck`. Следовательно, класс `CanastaDeck` или `PinochleDeck` станет подклассом класса `Deck` и будет включать только метод `__init__()`.

Выводы

В этой главе мы создали GUI-версию карточной игры «Большее-меньшее» из главы 1, используя хорошо воспроизводимые классы `Deck` и `Card`. Основной код создает экземпляр объекта `Game`, он в свою очередь создает объект `Deck`, который создает экземпляры 52 объектов `Card`, по одному для каждой карты в итоговой колоде. Каждый объект `Card` отвечает за рисование соответствующего изображения в окне и может отвечать на запросы по поводу его имени, ранга, масти и значения. Класс `Game`, который содержит логику игры, отделен от основного кода, выполняющего основной цикл.

Я продемонстрировал, как Python создает специальную переменную с именем `__name__` и присваивает ей различные значения в зависимости от того, выполняется ли файл в качестве основной программы или нет. Вы можете использовать эту функцию для добавления некоторого тестового кода, выполняющегося, когда вы запускаете файл в качестве основной программы (для тестирования кода в модуле), но бездействующего в типичном случае, когда файл импортируется другим модулем.

И наконец, я показал, как можно создавать различные типы колод карт в зависимости от того, насколько они отличаются от класса `Deck`.

13

ТАЙМЕРЫ



Эта глава посвящена таймерам. *Таймер* позволяет программе отсчитывать и ждать заданное количество времени, прежде чем перейти к выполнению какого-то другого действия. В мире текстовых программ Python это легко достижимо с помощью `time.sleep()` путем указания количества секунд для режима ожидания. Чтобы сделать паузу на две с половиной секунды, вы могли бы написать:

```
import time
time.sleep(2.5)
```

Тем не менее в мире `pygame` и управляемого событиями программирования в целом у пользователя всегда должна быть возможность взаимодействия с программой, потому подобные паузы неуместны. Вызов `time.sleep()` сделает программу инертной во время режима ожидания.

Вместо этого основной цикл должен продолжать выполняться с любой выбранной вами частотой фреймов. Нужно, чтобы программа продолжала цикл, но также отсчитывала время от заданной начальной точки до некоторого времени в будущем. Это может быть достигнуто тремя различными способами:

- измерением времени путем подсчета фреймов;
- использованием `pygame` для создания события, которое будет выпущено в будущем;
- запоминанием времени начала и постоянной проверки прошедшего времени.

Я кратко рассмотрю первые два, но сосредоточусь на третьем способе, поскольку он предоставляет наиболее четкий и точный подход.

Демонстрационная программа таймера

Чтобы продемонстрировать различные подходы, я буду использовать различные реализации тестовой программы, показанной на рис. 13.1.

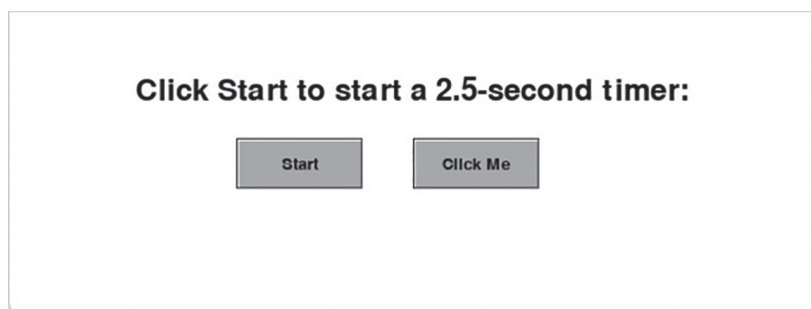


Рис. 13.1. Демонстрационная программа таймера

Когда пользователь щелкает по кнопке **Start**, запускается 2,5-секундный таймер и вид окна изменяется, как показано на рис. 13.2.

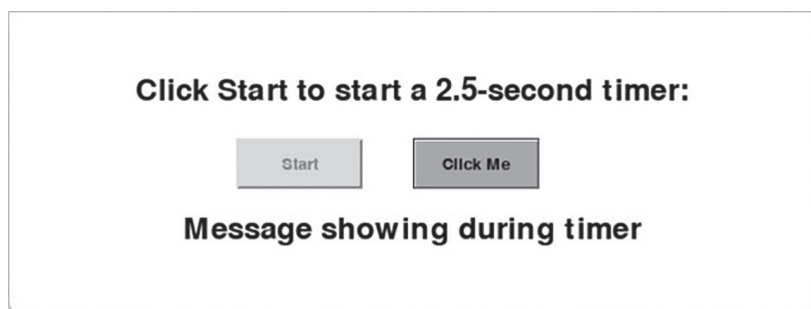


Рис. 13.2. Отображенное сообщение во время работы таймера

На две с половиной секунды кнопка **Start** становится неактивной и под кнопками отображается сообщение. По истечении времени оно исчезает и кнопка **Start** вновь становится активной. Независимо от запущенного таймера все остальное, что пользователь хочет сделать в программе, по-прежнему должно быть активным. В этом примере щелчок по кнопке **Click me** выводит сообщение в окне оболочки вне зависимости от работы таймера.

Три подхода к реализации таймеров

В этом разделе я рассмотрю три различных подхода к реализации таймеров: подсчет фреймов, генерацию события `pygame` и проверку прошедшего времени. Для прояснения этих концепций следующие примеры кода встроены непосредственно в основной цикл.

Подсчет фреймов

Простой подход к созданию таймера заключается в подсчете количества выполняемых фреймов. Один фрейм — это то же самое, что одна итерация цикла. Если знаете частоту фреймов программы, вы можете вычислить, сколько времени необходимо подождать, умножив время ожидания на частоту фреймов. Следующий код демонстрирует ключевые моменты реализации:

Файл: `InLineTimerExamples/CountingFrames.py`

```
FRAMES_PER_SECOND = 30 # каждый фрейм занимает 1/30 секунды
TIMER_LENGTH = 2.5

--- пропуск ---
timerRunning = False
```

Этот код показывает, что происходит, когда пользователь щелкает по кнопке **Start**:

```
if startButton.handleEvent(event):
    timerRunning = True
    nFramesElapsed = 0 # инициализируем счетчик
    nFramesToWait = int(FRAMES_PER_SECOND * TIMER_LENGTH)
    startButton.disable()
    timerMessage.show()
```

Программа вычисляет, что она должна подождать 75 фреймов (2,5 секунды \times 30 фреймов в секунду), и мы устанавливаем `timerRunning` в значение `True`, чтобы указать, что таймер запущен. Внутри основного цикла используем следующий код для проверки завершения таймера:

```
if timerRunning:
    nFramesElapsed = nFramesElapsed + 1 # увеличиваем
                                         # счетчик
    if nFramesElapsed >= nFramesToWait:
        startButton.enable()
        timerMessage.hide()
        print('Timer ended by counting frames')
        timerRunning = False
```

Когда таймер завершается, мы вновь активируем кнопку **Start**, убираем сообщение и сбрасываем переменную `timerRunning`. (Если хотите, вы можете установить счетчик на количество фреймов для ожидания и выполнять вместо этого обратный отсчет до нуля.) Этот подход хорошо работает, но он привязан к частоте фреймов программы.

Таймер событий

Во втором подходе мы воспользуемся встроенным в `pygame` таймером. `pygame` позволяет добавлять новое событие в очередь — это называется *регистрация* события. Если точнее, мы просим `pygame` создать и зарегистрировать событие таймера. Нам всего лишь необходимо указать, насколько далеко в будущем должно произойти событие. После заданного промежутка времени `pygame` запустит событие таймера в основном цикле тем же способом, которым он запускает другие стандартные события, такие как `KEYUP`, `KEYDOWN`, `MOUSEBUTTONUP`, `MOUSEBUTTONDOWN` и так далее. Ваш код должен будет искать этот тип события и реагировать на него.

Следующая документация взята с <https://www.pygame.org/docs/ref/time.html>:

```
pygame.time.set_timer()
```

Множественно создавать событие в очереди событий

```
set_timer(eventid, milliseconds) -> None
```

```
set_timer(eventid, milliseconds, once) -> None
```

Устанавливаем тип события для отображения в очереди событий каждое заданное количество миллисекунд. Первое событие не появится до тех пор, пока не пройдет заданное количество времени.

К каждому типу события может быть прикреплен отдельный таймер. Лучше всего использовать значение между `pygame.USEREVENT` и `pygame.NUMEVENTS`.

Чтобы отключить таймер для события, установите аргумент `milliseconds` в значение 0.

Если аргумент `once` имеет значение `True`, таймер устанавливается только один раз.

Каждый тип события в `pygame` представлен уникальным идентификатором. В `pygame 2.0` вы можете теперь вызвать `pygame.event.custom_type()`, чтобы получить идентификатор для пользовательского события.

Файл: `InLineTimerExamples/TimerEvent.py`

```
TIMER_EVENT_ID = pygame.event.custom_type() # новое в pygame 2.0
TIMER_LENGTH = 2.5 # секунды
```

Когда пользователь щелкает по кнопке **Start**, код создает и регистрирует событие таймера:

```
if startButton.handleEvent(event):
    pygame.time.set_timer(TIMER_EVENT_ID,
                          int(TIMER_LENGTH * 1000), True)
    --- пропуск деактивации кнопки, отображения сообщения ---
```

Рассчитанное значение равно 2500 миллисекундам. `True` означает, что таймер должен выполняться только один раз (сгенерировать только одно событие). Теперь нам нужен код в цикле события, который проверяет его наступление:

```
if event.type == TIMER_EVENT_ID:
    --- пропуск активации кнопки, закрытия сообщения ---
```

Так как мы указали `True` в вызове для установки таймера, это событие запускается только один раз. Если хотим повторять события каждые 2500 миллисекунд, мы можем установить последний аргумент в исходном вызове в значение `False` (или

просто разрешить значение по умолчанию False). Чтобы завершить повторяемые события таймера, мы вызвали бы `set_timer()` и передали бы 0 (ноль) в качестве второго аргумента.

Создаем таймер путем вычисления прошедшего времени

Третий подход к реализации таймера использует текущее время в качестве отправной точки. Затем мы можем многократно запрашивать текущее время и выполнять простое вычитание, чтобы вычислить прошедшее время. Код, показанный в этом примере, выполняется в основном цикле; далее мы извлечем относящийся к таймеру код и создадим многократно используемый класс `Timer`.

В модуле `time` стандартной библиотеки Python есть следующая функция:

```
time.time()
```

Вызов этой функции возвращает текущее время в секундах в виде числа с плавающей точкой. Возвращаемое значение представляет собой количество секунд, которое прошло с момента «начала отсчета времени», которое определено как 00:00:00 UTC 1 января 1970 года.

Код в листинге 13.1 создает таймер, запоминая время, когда пользователь щелкнул по кнопке **Start**. Во время работы таймера мы проверяем каждый фрейм, чтобы увидеть, прошло ли необходимое количество времени. Вы уже видели интерфейс пользователя, поэтому для краткости я опущу эти детали и некоторые части установочного кода.

Файл: `InLineTimerExamples/ElapsedTime.py`

```
# Timer в основном цикле

--- пропуск ---

TIMER_LENGTH = 2.5 # секунды
--- пропуск ---
timerRunning = False

#6 - Бесконечный цикл
while True:
    #7 - Проверяем наличие событий и обрабатываем их
```

```

for event in pygame.event.get():
    if event.type == pygame.QUIT:
        pygame.quit()
        sys.exit()

❶    if startButton.handleEvent(event):
        timeStarted = time.time() # запоминаем время начала
        startButton.disable()
        timerMessage.show()
        print('Starting timer')
        timerRunning = True
        if clickMeButton.handleEvent(event):
            print('Other button was clicked')

#8 - Выполняем действия "в рамках фрейма"
❷    if timerRunning: # если таймер запущен
        elapsed = time.time() - timeStarted
❸    if elapsed >= TIMER_LENGTH: # True здесь обозначает
                                    # завершение таймера
        startButton.enable()
        timerMessage.hide()
        print('Timer ended')
        timerRunning = False

#9 - Очищаем окно
window.fill(WHITE)

#10 - Рисуем все элементы окна
headerMessage.draw()
startButton.draw()
clickMeButton.draw()
timerMessage.draw()

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 13.1. Таймер, встроенный в основной цикл

Важными переменными, на которые стоит обратить внимание в этой программе, являются следующие.

`TIMER_LENGTH` Константа, которая сообщает, как долго мы хотим, чтобы работал наш таймер.

`timerRunning` Булево выражение, которое сообщает нам, запущен ли таймер.

`timeStarted` Время, когда пользователь нажал кнопку **Start**.

Когда пользователь щелкает по кнопке **Start**, `timerRunning` устанавливается в значение `True` ❶. Мы инициализируем переменную `startTime` текущим временем. Затем деактивируем кнопку **Start** и отображаем сообщение под кнопками.

Во время прохождения каждого цикла, если таймер запущен ❷, мы вычитаем время начала из текущего, чтобы увидеть, сколько прошло с момента запуска таймера. Когда количество прошедшего времени превосходит или равно `TIMER_LENGTH`, может произойти любое действие, которое мы запланируем. В этом примере программы активируем кнопку **Start**, удаляем нижнее сообщение, выводим краткий текстовый вывод и сбрасываем переменную `timerRunning` в значение `False` ❸.

Код в листинге 13.1 работает хорошо... для одного таймера. Однако эта книга посвящена объектно-ориентированному программированию, поэтому давайте попробуем масштабировать то, что у нас уже получилось. Чтобы обобщить функциональные возможности, преобразуем код тайминга в класс. Возьмем важные переменные, преобразуем их в переменные экземпляра и разобьем код на методы. Таким способом мы можем определить и использовать любое количество таймеров в программе. Класс `Timer` наряду с другими классами, используемыми для отображения тайминга в программах `pygame`, доступен в модуле под названием `pyghelpers`.

Устанавливаем `pyghelpers`

Чтобы установить `pyghelpers`, откройте командную строку и введите следующие две команды:

```
python3 -m pip install -U pip --user
```

```
python3 -m pip install -U pyghelpers --user
```

Они загрузят и установят `pyghelpers` из PyPi в папку, доступную всем вашим программам Python. Как только установка завершена, вы можете использовать `pyghelpers`, включая следующий оператор в начале ваших программ:

```
import pyghelpers
```

Затем можете создать экземпляры объектов из классов в модуле и вызвать методы этих объектов. Самая актуальная

документация pyghelpers находится по адресу <https://pyghelpers.readthedocs.io/en/latest/>, а исходный код доступен в моем репозитории GitHub по адресу <https://github.com/IrvKalb/pyghelpers/>.

Класс Timer

В листинге 13.2 содержится код очень простого таймера в виде класса. Этот код встроен в пакет pyghelpers в качестве класса Timer (для краткости я опустил здесь некоторую документацию).

Файл: (Доступен как часть модуля pyghelpers)

```
# Класс Timer

class Timer():
    --- пропуск ---
❶ def __init__(self, timeInSeconds, nickname=None, callBack=None):
    self.timeInSeconds = timeInSeconds
    self.nickname = nickname
    self.callBack = callBack
    self.savedSecondsElapsed = 0.0
    self.running = False

❷ def start(self, newTimeInSeconds=None):

    --- пропуск ---

    if newTimeInSeconds != None:
        self.timeInSeconds = newTimeInSeconds
    self.running = True
    self.startTime = time.time()

❸ def update(self):

    --- пропуск ---

    if not self.running:
        return False
    self.savedSecondsElapsed = time.time() - self.startTime
    if self.savedSecondsElapsed < self.timeInSeconds:
        return False # выполняется, но не достиг граничного
                     # условия

    else: # таймер завершен
        self.running = False
        if self.callBack is not None:
```

```

        self.callBack(self.nickname)

        return True # True здесь обозначает завершение таймера

❷ def getTime(self):

    --- пропуск ---

    if self.running:
        self.savedSecondsElapsed = time.time() - self.startTime

    return self.savedSecondsElapsed

❸ def stop(self):
    """Останавливаем таймер"""
    self.getTime() # запоминаем окончательную self.
savedSecondsElapsed
    self.running = False

```

Листинг 13.2. Простой класс Timer

Когда вы создаете объект `Timer`, единственным обязательным аргументом является количество секунд, в течение которых вы хотите, чтобы таймер работал ❶. Можете дополнительно предоставить псевдоним для таймера и функцию или метод для обратного вызова, когда пройдет заданное время. Если вы указываете обратный вызов, псевдоним будет передан, когда произойдет обратный вызов.

Вы вызываете метод `start()` ❷ для запуска таймера. Объект `Timer` запоминает время начала в переменной экземпляра `self.startTime`.

Метод `update()` ❸ должен вызываться каждый раз во время прохождения основного цикла. Если таймер запущен и прошло соответствующее количество времени, этот метод возвращает `True`. В любом другом вызове метод возвращает `False`.

Если таймер `Timer` выполняется, вызов `getTime()` ❹ возвращает количество прошедшего времени для этого `Timer`. Вы можете вызвать метод `stop()` ❺, чтобы немедленно остановить `Timer`.

Теперь мы можем переписать демонстрационную программу таймера, показанную с листинге 13.1, чтобы использовать класс `Timer` из пакета `pyghelpers`. В листинге 13.3 видно, как мы используем объект `Timer` в коде.

Файл: TimerObjectExamples/SimpleTimerExample.py

```
# Пример простого таймера

--- пропуск ---

❶ oTimer = pyghelpers.Timer(TIMER_LENGTH) # создаем объект Timer

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        if startButton.handleEvent(event):
            ❷ oTimer.start() # запускаем таймер
            startButton.disable()
            timerMessage.show()
            print('Starting timer')

        if clickMeButton.handleEvent(event):
            print('Other button was clicked')

    #8 - Выполняем действия "в рамках фрейма"
    ❸ if oTimer.update(): # True здесь обозначает завершение таймера
        startButton.enable()
        timerMessage.hide()
        print('Timer ended')

    #9 - Очистить экран
    window.fill(WHITE)

    #10 - Нарисовать все элементы экрана
    headerMessage.draw()
    startButton.draw()
    clickMeButton.draw()
    timerMessage.draw()

    #11 - Обновить экран
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND) # ожидание pygame
```

Листинг 13.3. Основная программа, в которой используется экземпляр класса `Timer`

И вновь я вырезал установочный код. Перед началом основного цикла мы создаем объект `Timer` ❶. Когда пользователь щелкает по кнопке **Start**, вызываем `oTimer.start()` ❷, чтобы запустить таймер.

Во время каждого прохождения цикла мы вызываем метод `update()` объекта `Timer` ❸. Существует два способа узнать, что таймер завершен. Первый заключается в проверке того, возвращает ли этот вызов значение `True`. Пример кода в листинге 13.3 использует такой подход. В качестве альтернативы, если мы указали значение для `callBack` в вызове `__init__()`, то, когда таймер завершается, будет обратно вызвана функция или метод, указанные в `callBack`. В большинстве случаев я рекомендовал бы использовать первый подход.

Существует два преимущества при использовании класса `Timer`. Во-первых, он скрывает детали кода тайминга; вы только создаете объект `Timer`, когда хотите, и вызываете методы этого объекта. Во-вторых, вы можете создать столько объектов `Timer`, сколько пожелаете, и каждый будет выполняться независимо.

Отображаем время

Многим программам требуется подсчитывать и отображать время для пользователя. Например, в игре прошедшее время должно постоянно отображаться и обновляться, или у пользователя может быть определенное количество времени для завершения задачи, для чего требуется таймер обратного отсчета. Я продемонстрирую, как выполнить и то и другое, используя игру пятнашки, изображенную на рис. 13.3.

Когда вы запускаете игру, плитки размещаются произвольным образом и остается одно пустое черное поле. Цель игры — перемещать плитки по одной, чтобы разместить их в порядке от 1 до 15. Разрешается щелкать только по плитке, которая по горизонтали или по вертикали примыкает к пустому квадрату. Щелчок по подходящей плитке меняет ее местами с пустым полем. Я не буду вдаваться в подробности полной реализации игры (хотя исходный код доступен онлайн вместе с остальными ресурсами книги). Вместо этого я сосредоточусь на том, как интегрировать таймер.

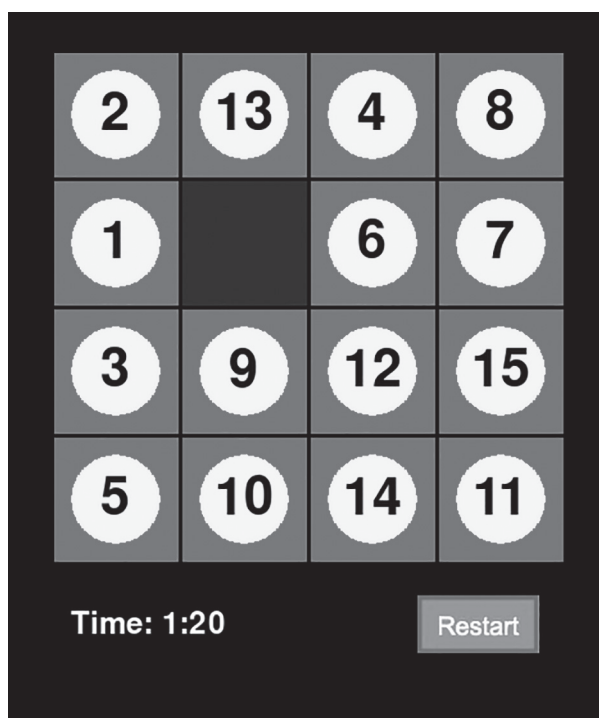


Рис. 13.3. Пользовательский интерфейс игры пятнашки

Пакет `pyghelpers` содержит два класса, которые позволяют программистам отслеживать время. Первый — это `CountUpTimer`, который начинается с нуля и считает в прямом направлении до бесконечности или до тех пор, пока вы не скажете ему остановиться. Второй — это `CountDownTimer`, который начинается с заданного количества времени и считает в обратном направлении до нуля. Я создал версии игры для каждого из них. Первая позволяет пользователю видеть, сколько времени у него занимает решение головоломки. Во второй пользователю в начале игры выделено определенное количество времени, и если он не завершил задачу, когда таймер достиг нуля, то проиграл.

CountUpTimer

С помощью класса `CountUpTimer` вы создаете объект таймера и сообщаете ему, когда начать. Затем в каждом фрейме можете вызвать один из трех различных методов, чтобы получить прошедшее время в разных форматах.

Листинг 13.4 содержит реализацию класса `CountUpTimer` из `pyghelpers`. Этот код является хорошим примером того, как различные методы класса совместно используют переменные экземпляра.

Файл: (Доступен как часть модуля `pyghelpers`)

```
# Класс CountUpTimer

class CountUpTimer():

    --- пропуск ---

    def __init__(self): ❶
        self.running = False
        self.savedSecondsElapsed = 0.0
        self.secondsStart = 0 # защитная мера

    def start(self): ❷
        --- пропуск ---
        self.secondsStart = time.time() # получаем текущее время
                                         # в секундах и сохраняем значение

        self.running = True
        self.savedSecondsElapsed = 0.0

    def getTime(self): ❸
        """Возвращает прошедшее время в виде значения с плавающей
        точкой"""
        if not self.running:
            return self.savedSecondsElapsed # ничего не делаем

        self.savedSecondsElapsed = time.time() - self.secondsStart
        return self.savedSecondsElapsed # возвращаем значение
                                         # с плавающей точкой

    def getTimeInSeconds(self): ❹
        """Возвращает прошедшее время в виде целого числа в секундах"""
        nSeconds = int(self.getTime())
        return nSeconds

    # Обновленная версия, использующая fStrings
    def getTimeInHHMMSS(self, nMillisecondsDigits=0): ❺
        --- пропуск ---
        nSeconds = self.getTime()
        mins, secs = divmod(nSeconds, 60)
        hours, mins = divmod(int(mins), 60)
```

```

if nMillisecondsDigits > 0:
    secondsWidth = nMillisecondsDigits + 3
else:
    secondsWidth = 2

if hours > 0:
    output =
        f'{hours:d}:{mins:02d}:{secs:0{secondsWidth}.
                                     {nMillisecondsDigits}f}'
elif mins > 0:
    output = f'{mins:d}:{secs:0{secondsWidth}.
                                     {nMillisecondsDigits}f}'
else:
    output = f'{secs:.{nMillisecondsDigits}f}'

return output

def stop(self): ❹
    """Останавливает таймер"""
    self.getTime() # запоминаем окончательное значение self.
savedSecondsElapsed
    self.running = False

```

Листинг 13.4. Класс CountUpTimer

Реализация зависит от трех ключевых переменных экземпляра ❶:

- `self.running` — это булево выражение, которое указывает, запущен ли таймер;
- `self.savedSecondsElapsed` — это значение с плавающей точкой, которое представляет прошедшее время таймера;
- `self.secondsStart` — это время запуска таймера.

Клиент вызывает метод `start()` ❷, чтобы запустить таймер. В результате метод вызывает `time.time()`, сохраняет время начала в `self.secondsStart` и устанавливает `self.running` в значение `True`, чтобы указать, что таймер запущен.

Клиент может вызвать любой из этих трех методов, чтобы получить прошедшее время, связанное с таймером, в трех различных форматах:

- `getTime()` ❸ возвращает прошедшее время в виде числа с плавающей точкой;
- `getTimeInSeconds()` ❹ возвращает прошедшее время в виде целого числа секунд;

- `getTimeInHHMMSS()` ❸ возвращает прошедшее время в виде отформатированной строки.

Метод `getTime()` вызывает `time.time()`, чтобы получить текущее время, и вычитает время начала, чтобы получить прошедшее время. Другие два метода вызывают метод `getTime()` этого класса, чтобы вычислить прошедшее время, затем выполняют разную обработку выходных данных:

`getTimeInSeconds()` преобразует время в целое число секунд, а `getTimeInHHMMSS()` форматирует его в строку формата *часы:минуты:секунды*. Выходные данные каждого из этих методов предназначены для отправки объекту `DisplayText` (определенному в пакете `pygwidgets`) для отображения в окне.

Метод `stop()` ❹ может быть вызван для остановки таймера (например, когда пользователь завершает головоломку).

Основной файл для этой версии пятнашек доступен вместе с остальными ресурсами книги в каталоге *SliderPuzzles/Main_SliderPuzzleCountUp.py*. Он создает экземпляр объекта `CountUpTimer` перед началом основного цикла и сохраняет его в переменной `oCountUpTimer`. Затем сразу же вызывает метод `start()`. Также он создает поле `DisplayText` для отображения времени. Каждый раз во время прохождения основного цикла основной код вызывает метод `getTimeInHHMMSS()` и отображает результаты в поле:

```
timeToShow = oCountUpTimer.getTimeInHHMMSS() # запрашиваем у объекта
                                              # Timer прошедшее время
oTimerDisplay.setValue('Time: ' + timeToShow) # помещаем в текстовое поле
```

Переменная `oTimerDisplay` является экземпляром класса `pygwidgets.DisplayText`. Метод `setValue()` класса `DisplayText` оптимизирован для проверки того, окажется ли новый для отображения текст тем же, что и предыдущий. Следовательно, даже несмотря на то что мы определяем в поле количество времени 30 раз в секунду, не так много работы выполняется, пока время не изменится один раз в секунду.

Код игры проверяет, решена ли головоломка, и, когда она решена, вызывается метод `stop()` для остановки времени. Если пользователь щелкает по кнопке **Restart**, чтобы начать новую игру, программа вызывает `start()`, чтобы вновь запустить объект таймера.

CountDownTimer

В классе `CountDownTimer` есть небольшое отличие. Вместо подсчета в прямом направлении от нуля вы инициализируете `CountDownTimer`, задавая начальное количество секунд, и он считает в обратном направлении от этого значения. Интерфейс для создания `CountDownTimer` выглядит следующим образом:

```
CountDownTimer(nStartingSeconds, stopAtZero=True, nickname=None,
               callBack=None):
```

Существует второй необязательный параметр `stopAtZero`, который по умолчанию равен `True`, когда предполагается, что вы хотите, чтобы таймер остановился по достижении нуля. Также допустимо дополнительно указать функцию или метод в качестве обратного вызова, когда таймер достигает нуля. И наконец, вы можете предоставить псевдоним для использования при выполнении обратного вызова.

Клиент вызывает метод `start()`, чтобы начать обратный отсчет.

С точки зрения клиента, `getTime()`, `getTimeInSeconds()`, `getTimeInHHMMSS()` и метод `stop()` выглядят идентично их аналогам в классе `CountUpTimer`.

В `CountDownTimer` есть дополнительный метод с именем `ended()`. Приложение должно вызывать метод `ended()` каждый раз во время прохождения своего основного цикла. Он возвращает `False`, если таймер активен, но возвращает `True`, когда тот завершается (то есть достигает нуля).

Версия обратного отсчета основного файла игры в пятнашки доступна вместе с остальными ресурсами книги в каталоге *SliderPuzzles/Main_SliderPuzzleCountDown.py*.

Код очень похож на предыдущую версию, которая выполняет прямой отсчет, но эта версия создает экземпляры `CountDownTimer` и предоставляет количество секунд, которое выделено на решение головоломки. Она также вызывает `getTimeInHHMMSS(2)` в каждом фрейме и обновляет время с двумя десятичными знаками. И наконец, она включает метод `ended()` в каждый фрейм, чтобы увидеть, закончилось ли время. Если таймер завершается до того, как пользователь решил головоломку, он воспроизводит звук и отображает сообщение, говорящее пользователю, что его время вышло.

Выводы

Эта глава предоставила вам несколько способов обработки тайминга в программах. Я рассмотрел три различных подхода: первый — путем подсчета фреймов, второй — путем создания пользовательского события и последний — путем запоминания времени начала и вычитания его из текущего времени для выяснения прошедшего времени.

Используя третий подход, мы создали универсальный, многократно используемый класс `Timer` (который вы можете найти в пакете `pygamehelpers`). Я также продемонстрировал два дополнительных класса из этого пакета: `CountUpTimer` и `CountDownTimer`, — которые можно использовать для обработки тайминга в тех программах, где вы хотите отображать таймер для пользователя.

14

АНИМАЦИЯ



Данная глава посвящена анимации, в частности традиционной анимации изображений.

На очень простом уровне вы можете воспринимать это как мультфильм: серию изображений,

каждое из которых немного отличается от предыдущего, и они отображаются последовательно. Пользователь видит каждое изображение в течение короткого периода времени и испытывает иллюзию движения. Анимация обеспечивает хорошую возможность для создания класса, потому что механика демонстрации изображений с течением временем хорошо понятна и легко кодируется.

Чтобы продемонстрировать общие принципы, мы начнем с реализации двух классов анимации: класса `SimpleAnimation`, основанного на серии отдельных файлов изображений, и класса `SimpleSpriteSheetAnimation`, выстраиваемого с использованием одного файла, содержащего последовательность множества изображений. Затем я продемонстрирую два наиболее надежных анимационных класса в пакете `pygwidgets` — `Animation` и `SpriteSheetAnimation` — и объясню, как создавать их с помощью общего базового класса.

Создаем классы анимации

Основная идея, лежащая в основе класса анимации, относительно проста. Клиент будет предоставлять упорядоченный набор изображений и количество времени. Клиентский код станет сообщать анимации, когда начинать воспроизведение, и периодически говорить ей обновить себя. Изображения в анимации будут отображаться по порядку, каждое в течение заданного количества времени.

Класс SimpleAnimation

Общий метод состоит в том, чтобы начать с загрузки полного набора изображений, сохранения их в списке и отображения первого изображения. Когда клиент говорит анимации запуститься, та начинает отслеживать время. Каждый раз, когда объекту предлагается обновить себя, наш код проверяет, прошло ли указанное количество времени, и, если это так, показывает следующее изображение в последовательности. Когда анимация завершается, мы вновь видим первое изображение.

Создаем класс

В листинге 14.1 содержится код класса `SimpleAnimation`, который обрабатывает анимацию, составленную из отдельных файлов изображений. Чтобы все было четко организовано, я настоятельно рекомендую поместить все файлы изображений, связанные с анимацией, в подпапку внутри папки *images* внутри папки вашего проекта. Приведенные здесь примеры будут использовать эту структуру, а связанная графика и основной код доступны вместе с остальными ресурсами книги.

Файл: SimpleAnimation/SimpleAnimation.py

```
# Класс SimpleAnimation

import pygame
import time

class SimpleAnimation():
    def __init__(self, window, loc, picPaths, durationPerImage): ❶
        self.window = window
        self.loc = loc
        self.imagesList = []
```

```

for picPath in picPaths:
    image = pygame.image.load(picPath) # загружаем
                                         # изображение
    image = pygame.Surface.convert_alpha(image) ❷
                                         # оптимизируем блиттинг
    self.imagesList.append(image)

self.playing = False
self.durationPerImage = durationPerImage
self.nImages = len(self.imagesList)
self.index = 0

def play(self): ❸
    if self.playing:
        return
    self.playing = True
    self.imageStartTime = time.time()
    self.index = 0

def update(self): ❹
    if not self.playing:
        return

    # Сколько времени прошло с момента начала демонстрации этого
    # изображения
    self.elapsed = time.time() - self.imageStartTime

    # Если прошло достаточно времени, переходим к следующему
    # изображению
    if self.elapsed > self.durationPerImage:
        self.index = self.index + 1

    if self.index < self.nImages: # переходим к следующему
                                   # изображению
        self.imageStartTime = time.time()
    else: # анимация завершена
        self.playing = False
        self.index = 0 # сбрасываем на начало

def draw(self): ❺
    # Предполагается, что self.index была установлена ранее -
    # в методе update().
    # Используется в качестве индекса в imagesList для поиска
    # текущего изображения.
    theImage = self.imagesList[self.index] # выбираем
                                             # изображение для отображения

    self.window.blit(theImage, self.loc) # отображаем его

```

Листинг 14.1. Класс SimpleAnimation

Когда клиент создает экземпляр класса `SimpleAnimation`, он должен передать следующее:

window — окно, в котором рисовать;

loc — местоположение окна для рисования изображений;

picPaths — список или кортеж путей к изображениям. Изображения будут отображаться в заданном здесь порядке;

durationPerImage — как долго (в секундах) демонстрировать изображение.

В методе `__init__()` ❶ мы сохраняем эти параметры-переменные в переменные экземпляра с аналогичными именами. Метод перебирает список путей, загружает каждое изображение и сохраняет итоговое изображение в список. Список — это великолепный способ представить упорядоченный набор изображений. Класс будет использовать переменную `self.index` для отслеживания текущего изображения в списке.

Формат изображения в файле отличается от формата изображения, когда оно показывается на экране. Вызов `convert_alpha()` ❷ преобразует файловый формат в экранный, чтобы оптимизировать работу при демонстрации изображений в окне. Фактическое рисование осуществляется позднее в методе `draw()`.

Метод `play()` ❸ запускает анимацию. Сначала он проверяет, не выполняется ли уже анимация, и, если это так, метод просто завершает работу. В противном случае он устанавливает `self.playing` в значение `True`, чтобы обозначить, что анимация теперь выполняется.

Когда создается `SimpleAnimation`, вызывающий указывает количество времени, в течение которого должно демонстрироваться каждое изображение, и это сохраняется в `self.durationPerImage`. Следовательно, нам надо отслеживать время по мере выполнения `SimpleAnimation`, чтобы знать, когда переключаться на следующее изображение. Вызываем `time.time()`, чтобы получить текущее время (в миллисекундах), и сохраняем его в переменной экземпляра. Создание основанного на времени класса означает, что любой объект `SimpleAnimation`, созданный из этого класса, будет работать правильно вне зависимости от частоты фреймов, которая используется в основном цикле.

И наконец, мы устанавливаем переменную `self.index()`, чтобы указать, что нужно показывать первое изображение.

Метод `update()` ❹ необходимо вызывать в каждом фрейме основного цикла. Если анимация не воспроизводится, `update()` ничего не делает и просто завершает работу. В противном случае `update()` вычисляет, сколько времени прошло с момента начала демонстрации первого изображения, получая текущее время с помощью системной функции `time.time()` и вычитая из него время, когда текущее изображение начало демонстрацию.

Если прошедшее время превосходит количество времени, в течение которого должно демонстрироваться каждое изображение, пора перейти к следующему. В данном случае мы увеличиваем `self.index`, чтобы предстоящий вызов метода `draw()` рисовал соответствующее изображение. Затем мы проверяем, завершилась ли анимация. Если нет, сохраняем время начала для нового изображения. Если анимация завершена, устанавливаем `self.playing` обратно в значение `False` (чтобы указать, что мы больше не воспроизводим анимацию) и сбрасываем `self.index` до 0, чтобы метод `draw()` вновь продемонстрировал первое изображение.

Наконец, вызываем `draw()` в каждом фрейме ❺, чтобы рисовать текущее изображение анимации. Метод `draw()` предполагает, что поле `self.index` было установлено правильно предыдущим методом, и использует его для индексирования в списке изображений. Затем он рисует это изображение в окне в указанных координатах.

Пример основной программы

В листинге 14.2 продемонстрирована основная программа, которая создает и использует объект `SimpleAnimation`. Она создаст анимацию динозавра, едущего на велосипеде.

Файл: SimpleAnimation/Main_SimpleAnimation.py

```
# Пример анимации
# Демонстрируется пример объекта SimpleAnimation

#1 - Импортируем библиотеку
import pygame
from pygame.locals import *
import sys
```



```

import pygame
from SimpleAnimation import *

#2 - Определяем константы
SCREEN_WIDTH = 640
SCREEN_HEIGHT = 480
FRAMES_PER_SECOND = 30
BGCOLOR = (0, 128, 128)

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode([SCREEN_WIDTH, SCREEN_HEIGHT])
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.
❶ dinosaurAnimTuple = ('images/Dinobike/f1.gif',
                        'images/Dinobike/f2.gif',
                        'images/Dinobike/f3.gif',
                        'images/Dinobike/f4.gif',
                        'images/Dinobike/f5.gif',
                        'images/Dinobike/f6.gif',
                        'images/Dinobike/f7.gif',
                        'images/Dinobike/f8.gif',
                        'images/Dinobike/f9.gif',
                        'images/Dinobike/f10.gif')

#5 - Инициализируем переменные
oDinosaurAnimation = SimpleAnimation(window, (22, 140),
                                     dinosaurAnimTuple, .1)
oPlayButton = pygame_widgets.TextButton(window, (20, 240), "Play")

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
        ❷ if oPlayButton.handleEvent(event):
            oDinosaurAnimation.play()

    #8 - Выполняем действия "в рамках фрейма"
    ❸ oDinosaurAnimation.update()

    #9 - Очищаем окно
    window.fill(BGCOLOR)

```

```

#10 - Рисуем все элементы окна
4 oDinosaurAnimation.draw()
oPlayButton.draw()

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 14.2. Основная программа, которая создает экземпляр SimpleAnimation и воспроизводит его

Все изображения для анимированного динозавра находятся в папке *images/DinoBike/*. Сначала мы создаем кортеж изображений ❶. Затем, используя его, создаем объект SimpleAnimation и указываем, что каждое изображение должно демонстрироваться десятую долю секунды. Также мы создаем экземпляр кнопки **Play**.

В основном цикле вызываем методы update() и draw() объекта oDinosaurAnimation. Программа заикливается, непрерывно рисуя текущее изображение анимации и кнопки **Play**. Когда анимация не выполняется, пользователь просто видит первое изображение.

Когда пользователь щелкает по кнопке **Play** ❷, программа вызывает метод play() объекта oDinosaurAnimation, чтобы запустить анимацию.

В основном цикле мы вызываем метод play() объекта oDinosaurAnimation ❸, который определяет, достаточно ли времени прошло, чтобы анимация перешла к следующему изображению.

И наконец, мы вызываем draw() ❹, и объект рисует соответствующее изображение.

Класс SimpleSpriteSheetAnimation

Второй тип анимации реализуется в классе SimpleSpriteSheetAnimation. *Лист спрайта* — это одно изображение, состоящее из некоторого количества меньших изображений одинакового размера, предназначенных для появления в определенном порядке, чтобы создать анимацию. С точки зрения разработчика, у листа спрайта есть три преимущества. Во-первых, все изображения находятся в одном файле, поэтому не нужно беспокоиться о создании имени для каждого отдельного файла. Во-вторых, можно увидеть ход анимации в одном

файле, вместо того чтобы пролистывать последовательность изображений. И, в-третьих, загрузка одного файла выполняется быстрее, чем загрузка списка файлов, составляющих анимацию.

На рис. 14.1 продемонстрирован пример листа спрайта.



Рис. 14.1. Лист спрайта изображения, состоящий из 18 изображений меньшего размера

Пример предназначен для отображения чисел от 0 до 17. Исходный файл содержит изображение размером 384×192 пикселей. Быстрое деление показывает, что размер каждого отдельного числового изображения — 64×64 пикселя. Ключевая идея заключается в том, что мы используем `pygame` для создания *подизображений* более крупного изображения, чтобы получить набор из 18 картинок размером 64×64 пикселя. Изображения меньшего размера затем можно показать при помощи того же метода, который мы использовали в классе `SimpleAnimation`.

Создаем класс

Листинг 14.3 содержит класс `SimpleSpriteSheetAnimation` для обработки анимации, основанной на листе спрайта.

Во время инициализации содержимое отдельного изображения листа спрайта разбивается на список меньших картинок, которые затем отображаются другими методами.

Файл: `SimpleSpriteSheetAnimation/SimpleSpriteSheetAnimation.py`

```
# Класс SimpleSpriteSheetAnimation

import pygame
import time

class SimpleSpriteSheetAnimation():
    def __init__(self, window, loc, imagePath, nImages, width, height,
                 durationPerImage): ❶
```

```

self.window = window
self.loc = loc
self.nImages = nImages
self.imagesList = []

# загружаем лист спрайта
spriteSheetImage = pygame.image.load(imagePath)
# оптимизируем блиттинг
spriteSheetImage = pygame.Surface.convert_alpha(spriteSheetImage)

# считаем количество столбцов в начальном изображении
nCols = spriteSheetImage.get_width() // width

# разбиваем начальное изображение на подизображения
row = 0
col = 0
for imageNumber in range(nImages):
    x = col * width
    y = row * height

    # создаем подповерхность из большего spriteSheet
    subsurfaceRect = pygame.Rect(x, y, width, height)
    image = spriteSheetImage.subsurface(subsurfaceRect)
    self.imagesList.append(image)

    col = col + 1
    if col == nCols:
        col = 0
        row = row + 1

self.durationPerImage = durationPerImage
self.playing = False
self.index = 0

def play(self):
    if self.playing:
        return
    self.playing = True
    self.imageStartTime = time.time()
    self.index = 0

def update(self):
    if not self.playing:
        return

    # Сколько времени прошло с момента начала демонстрации этого
    # изображения
    self.elapsed = time.time() - self.imageStartTime

```

```

# Если прошло достаточно времени, переходим к следующему
# изображению
if self.elapsed > self.durationPerImage:
    self.index = self.index + 1

    if self.index < self.nImages: # переходим к следующему
                                    # изображению
        self.imageStartTime = time.time()

    else: # анимация завершена
        self.playing = False
        self.index = 0 # сбрасываем на начало

def draw(self):
    # Предполагается, что self.index была установлена ранее -
    # в методе update().
    # Используем в качестве индекса в imagesList для поиска
    # текущего изображения.
    theImage = self.imagesList[self.index] # выбираем изображение
                                            # для отображения

    self.window.blit(theImage, self.loc) # отображаем его

```

Листинг 14.3. Класс SimpleSpriteSheetAnimation

Этот класс очень похож на SimpleAnimation, но, поскольку данная анимация основана на листе спрайта, методу `__init__()` должна передаваться другая информация ❶. Метод требует стандартных параметров `window` и `loc`, а также тех, что перечислены ниже:

imagePath — путь к изображению листа спрайта (один файл);

nImages — количество изображений в листе спрайта;

width — ширина каждого подизображения;

height — высота каждого подизображения;

durationPerImage — сколько (в секундах) демонстрировать каждое изображение.

Учитывая эти значения, метод `__init__()` загружает файл листа спрайта и использует цикл для разбивки более крупного изображения на список подизображений меньшего размера путем вызова метода `pygame.subsurface()`. Затем картинки меньшего размера добавляются в список `self.imagesList` для

использования другими методами. Метод `__init__()` использует счетчик для подсчета количества подизображений вплоть до числа, указанного пользователем; следовательно, последний ряд изображений не должен быть полным. Например, мы могли бы использовать изображение листа спрайта, в котором были бы только цифры от 0 до 14, вместо того чтобы заполнять ряд вплоть до 17. Параметр `nImages` является ключевым для выполнения этой работы.

У остальной части класса методы в точности такие же, как в предыдущем классе `SimpleAnimation`: `play()`, `update()` и `draw()`.

Пример основной программы

В листинге 14.4 представлена простая основная программа, создающая и отображающая объект `SimpleSpriteSheetAnimation`, который демонстрирует анимированную каплю воды: как она приземляется и растекается. Если вы загрузите все из папки *SpriteSheetAnimation* ресурсов этой книги, то получите код и соответствующую графику.

Файл: SimpleSpriteSheetAnimation/ Main_SimpleSpriteSheetAnimation.py

```
# Демонстрируется пример объекта SimpleSpriteSheetAnimation

#1 - Импортируем библиотеку
import pygame
from pygame.locals import *
import sys
import pygamewidgets
from SimpleSpriteSheetAnimation import *

#2 - Определяем константы
SCREEN_WIDTH = 640
SCREEN_HEIGHT = 480
FRAMES_PER_SECOND = 30
BGCOLOR = (0, 128, 128)

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode([SCREEN_WIDTH, SCREEN_HEIGHT])
clock = pygame.time.Clock()

#4 - Загружаем элементы: изображения, звуки и т. д.
```

```

#5 - Инициализируем переменные
❶ oWaterAnimation = SimpleSpriteSheetAnimation(window, (22, 140),
                                                'images/water_003.png',
                                                5, 50, 192, 192, .05)
oPlayButton = pygame_widgets.TextButton(window, (60, 320), "Play")

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        if oPlayButton.handleEvent(event):
            oWaterAnimation.play()

    #8 - Выполняем действия "в рамках фрейма"
    oWaterAnimation.update()

    #9 - Очищаем окно
    window.fill(BG_COLOR)

    #10 - Рисуем все элементы окна
    oWaterAnimation.draw()
    oPlayButton.draw()

    #11 - Обновляем окно
    pygame.display.update()

    #12 - Делаем паузу
    clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 14.4. Пример основной программы, которая создает и использует объект `SimpleSpriteSheetAnimation`

Единственное существенное отличие этого примера заключается в том, что он создает экземпляр объекта `SimpleSpriteSheetAnimation` ❶ вместо объекта `SimpleAnimation`.

Объединяем два класса

В методах `__init__()` в `SimpleAnimation` и `SimpleSpriteSheetAnimation` есть различные параметры, но другие три метода (`play()`, `update()` и `draw()`) идентичны.

Как только вы создали экземпляр какого-либо из этих двух классов, способ доступа к итоговым объектам становится абсолютно одинаковым. Согласно принципу «Не повторяйся» (DRY), наличие этих повторяющихся методов — плохая идея, поскольку любые исправления ошибок и/или улучшения нужно будет применять к обоим копиям методов.

Вместо этого лучше объединить классы. Мы можем создать общий абстрактный базовый класс, чтобы они наследовали от него. У базового класса появится собственный метод `__init__()`, который включает любой общий код из методов `__init__()` обоих изначальных классов, и он будет содержать методы `play()`, `update()` и `draw()`.

Каждый изначальный класс станет наследовать от нового базового класса и реализовывать собственный метод `__init__()`, используя соответствующие параметры. Каждый будет выполнять собственную работу по созданию `self.imagesList`, который затем используется в других трех методах в новом базовом классе.

Вместо демонстрации результата объединения этих двух «простых» классов я вам покажу результат такого объединения в «профессиональных» классах `Animation` и `SpriteSheetAnimation`, которые являются частью пакета `pygamewidgets`.

Классы анимации в `pygamewidgets`

Модуль `pygamewidgets` содержит следующие три класса анимации:

`PygAnimation` — абстрактный базовый класс для классов `Animation` и `SpriteSheetAnimation`;

`Animation` — класс для основанных на изображении анимаций (отдельных файлов изображений);

`SpriteSheetAnimation` — класс для основанных на листе спрайта анимаций (одно большое изображение).

Мы рассмотрим каждый по очереди. Классы `Animation` и `SpriteSheetAnimation` используют те же базовые концепции, что уже рассматривались, но у них также имеется больше опций, доступных посредством инициализации параметров.

Класс Animation

Класс `pygame Animation` предназначен для создания анимации из множества различных файлов изображений. Интерфейс следующий:

```
Animation(window, loc, animTuplesList, autoStart=False, loop=False,
          nickname=None, callBack=None, nIterations=1):
```

К обязательным параметрам относятся:

window — окно для рисования;

loc — верхний левый угол, в котором должны быть нарисованы изображения;

animTuplesList — список (или кортеж) кортежей, описывающий последовательность анимации.

Каждый внутренний кортеж содержит:

pathToImage — относительный путь к файлу изображения;

duration — продолжительность демонстрации этого изображения (в секундах, с плавающей точкой);

offset (необязательный) — если присутствует, кортеж (x, y) используется в качестве сдвига от основного loc для демонстрации изображения.

Следующие параметры необязательны:

autoStart — равен `True`, если вы хотите, чтобы анимация запускалась сразу; по умолчанию равен `False`;

loop — равен `True`, если вы хотите, чтобы анимация выполнялась непрерывно; по умолчанию равен `False`;

showFirstImageAtEnd — когда анимация завершается, отображает первое изображение снова; по умолчанию равно `True`;

nickname — внутреннее имя, присваиваемое этой анимации, используемое в качестве аргумента, когда указан `callback`;

callback — функция или метод объекта для вызова после завершения анимации;

nIterations — количество циклов анимации; по умолчанию равно 1.

В отличие от `SimpleAnimation`, который использует одну длительность для всех изображений, класс `Animation` позволяет указать длительность для *каждого*, обеспечивая большую гибкость в тайминге демонстрации картинок. Вы также можете указать сдвиг *x*, *y* при рисовании каждой, но обычно это не нужно. Ниже представлен пример кода, который создает объект `Animation`, демонстрирующий бегущего тираннозавра:

```
TRexAnimationList = [('images/TRex/f1.gif', .1),
                     ('images/TRex/f2.gif', .1),
                     ('images/TRex/f3.gif', .1),
                     ('images/TRex/f4.gif', .1),
                     ('images/TRex/f5.gif', .1),
                     ('images/TRex/f6.gif', .1),
                     ('images/TRex/f7.gif', .1),
                     ('images/TRex/f8.gif', .1),
                     ('images/TRex/f9.gif', .1),
                     ('images/TRex/f10.gif', .4)]
```

#5 — Инициализируем переменные

```
oDinosaurAnimation = pygwidgets.Animation(window, (22, 145),
TRexAnimationList, callBack=myFunction, nickname='Dinosaur')
```

Так создается объект `Animation`, который будет демонстрировать 10 различных картинок. Каждое из первых девяти изображений показывается одну десятую долю секунды, но последнее — четыре десятые доли секунды. Анимация будет воспроизводиться только один раз и не станет запускаться автоматически. Когда анимация завершена, будет вызвана `myFunction()` с аргументом `'Dinosaur'`.

Класс `SpriteSheetAnimation`

Для `SpriteSheetAnimation` вы передаете путь к одному файлу листа спрайта. Чтобы `SpriteSheetAnimation` разбил большое изображение на много маленьких, вы должны указать ширину и высоту подизображений. Для длительности у вас есть два варианта: можете указать одно значение, чтобы все изображения демонстрировались одинаковое количество времени, или указать список или кортеж длительностей, по одной для каждого изображения. Ниже представлен интерфейс:

```
SpriteSheetAnimation(window, loc, imagePath, nImages, width, height,  
                    durationOrDurationsList, autoStart=False,  
                    loop=False, nickname=None, callBack=None,  
                    nIterations=1):
```

К обязательным параметрам относятся:

Window — окно для рисования;

loc — верхний левый угол, в котором должны быть нарисованы изображения;

imagePath — относительный путь к файлу изображения листа спрайта;

nImages — общее количество подизображений в подизображении листа спрайта;

Width — ширина каждого отдельного итогового подизображения;

Height — высота каждого отдельного итогового подизображения;

durationOrDurationsList — количество времени для демонстрации каждого подизображения во время анимации или список длительностей, по одной на подизображение (длина должна быть `nImages`).

Следующие параметры необязательны:

autoStart — равен `True`, если вы хотите, чтобы анимация запускалась сразу; по умолчанию равен `False`;

loop — равен `True`, если вы хотите, чтобы анимация выполнялась непрерывно; по умолчанию равен `False`;

showFirstImageAtEnd — когда анимация завершается, отображает первое изображение снова; по умолчанию равно `True`;

nickname — внутреннее имя, присваиваемое этой анимации, используемое в качестве аргумента, когда указан `callback`;

callback — функция или метод объекта для вызова после завершения анимации;

nIterations — количество циклов анимации; по умолчанию равно 1.

Ниже представлен типичный оператор для создания объекта `SpriteSheetAnimation`:

```
oEffectAnimation = pygwidgets.SpriteSheetAnimation(window, (400, 150),  
                                                    'images/effect.png', 35, 192, 192, .1,  
                                                    autoStart=True, loop=True)
```

Это создает объект `SpriteSheetAnimation`, используя один файл изображения, найденный по заданному пути. Исходная картинка содержит 35 подизображений, каждое — 192×192 пикселя, и будет демонстрироваться в течение одной десятой доли секунды. Анимация станет запускаться автоматически и выполняться непрерывно.

Общий базовый класс: `PygAnimation`

Каждый класс `Animation` и `SpriteSheetAnimation` состоит только из метода `__init__()` и наследует от общего абстрактного базового класса `PygAnimation`. Методы `__init__()` в обоих классах вызывают унаследованный метод `__init__()` базового класса `PygAnimation`. Следовательно, методы `__init__()` классов `Animation` и `SpriteSheetAnimation` инициализируют только уникальные данные в своих классах.

После создания объекта `Animation` или `SpriteSheetAnimation` клиентский код должен включить вызовы `update()` и `draw()` в каждый фрейм. Ниже представлен список методов, доступных в обоих классах через базовый.

`handleEvent()` (*событие*)

Должен вызываться в каждом фрейме, чтобы проверить, щелкнул ли пользователь по анимации. Если это так, вы передаете событие, предоставляемое `pygame`. Этот метод в большинстве случаев возвращает `False`, но возвращает `True`, когда пользователь щелкает по изображению, и в этом случае вы, как правило, будете вызывать `play()`.

`play()`

Запускает анимацию.

`stop()`

Останавливает анимацию, где бы она ни находилась, и возвращается к демонстрации первого изображения.

pause()

Приводит к временной остановке анимации на текущем изображении. Вы можете продолжить воспроизведение, вызвав `play()`.

update()

Должен вызываться в каждом фрейме. Когда анимация запущена, метод высчитывает время для перехода к следующему изображению. Обычно он возвращает `False`, но возвращает `True`, когда анимация завершается (и не установлена на циклическое воспроизведение).

draw()

Должен вызываться в каждом фрейме. Он рисует текущее изображение анимации.

setLoop(*trueИлиFalse*)

Передать `True` или `False`, чтобы указать, должна ли анимация выполняться циклически.

getLoop()

Возвращает `True`, если анимация установлена на циклическое выполнение, или `False`, если нет.

ПРИМЕЧАНИЕ Местоположение анимации в окне определяется исходным значением `loc`, которое передается в `__init__()`. И `Animation`, и `SpriteSheetAnimation` наследуют от общего класса `PygAnimation`, а он наследует от `PygWidget`. Поскольку все методы, доступные в `PygWidget`, доступны также и в обоих классах анимации, несложно создать анимацию, которая также меняет свое местоположение во время воспроизведения. Вы можете выполнить любое перемещение анимации, вызвав `setLoc()`, унаследованный от `PygWidget`, и предоставив любые координаты `x` и `y`, которые вам нравятся, для каждого изображения.

Пример программы анимации

На рис. 14.2 показан скриншот примера программы, которая демонстрирует несколько анимаций, созданных из классов `Animation` и `SpriteSheetAnimation`.

Маленький динозавр слева — это объект `Animation`. Для него установлен `autoStart`, чтобы анимация

воспроизводилась при запуске программы, но только один раз. Нажатие кнопок под маленьким динозавром выполняет соответствующие вызовы объекта `Animation`. Если вы щелкнете по кнопке **Play (Играть)**, анимация запустится снова. Щелчок по кнопке **Pause (Пауза)** во время воспроизведения приостановит анимацию до тех пор, пока вы снова не щелкнете по кнопке **Play**. Если вы воспроизводите анимацию и затем щелкаете по кнопке **Stop (Стоп)**, анимация остановится и покажет первое изображение. Под этими кнопками находится два флажка. По умолчанию данная анимация не за циклируется. Если вы установите флажок **Loop (Цикл)**, затем нажмете **Play**, она будет воспроизводиться, пока вы не уберете флажок цикла. Флажок **Show (Показать)** делает анимацию видимой или невидимой.

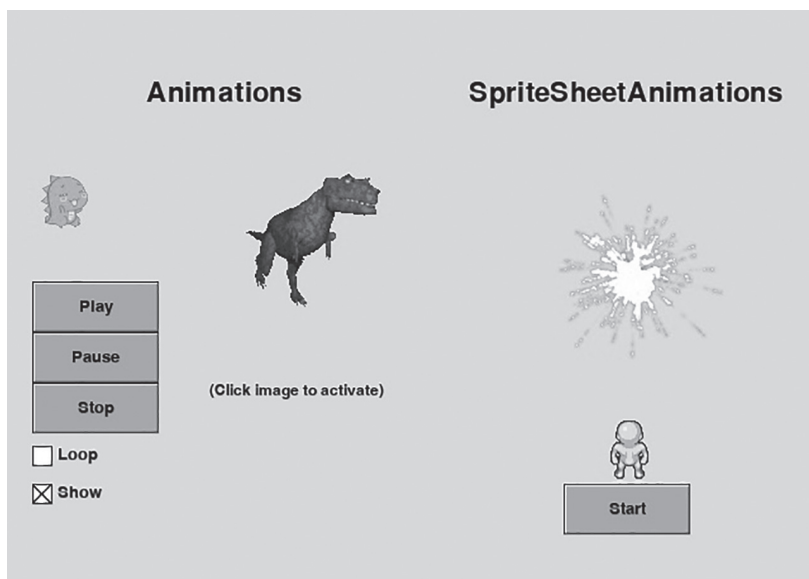


Рис. 14.2. Пример программы анимации с использованием классов `Animation` и `SpriteSheetAnimation`

Второй объект `Animation` (тираннозавр) не установлен на `autoStart`, поэтому вы видите только первое изображение анимации. Если щелкнете по изображению, анимация воспроизведет все свои картинки три раза (три цикла), прежде чем остановиться.

В верхнем правом углу находится фейерверк — объект `SpriteSheetAnimation`, который создан из одного изображения, содержащего 35 подизображений. Эта анимация

установлена на заикливание, поэтому вы видите ее непрерывное воспроизведение.

Внизу справа находится идущий человек `SpriteSheetAnimation` из одного изображения с 36 подизображениями. Когда вы щелкаете по кнопке **Start**, анимация воспроизводит все изображения один раз.

Весь исходный код этой программы доступен в файле *AnimationExample/Main_AnimationExample.py* вместе с остальными ресурсами книги.

Программа создает экземпляры двух объектов `Animation` (маленький динозавр и тираннозавр) и двух объектов `SpriteSheetAnimation` (фейерверк и идущий человек). Когда нажимается кнопка под маленьким динозавром, мы вызываем соответствующий метод анимационного объекта динозавра. Щелчок по маленькому динозавру или тираннозавру приводит к вызову метода `play()` этой анимации.

Программа демонстрирует, что одновременно может выполнять несколько анимаций. Это работает, поскольку основным циклом вызывает методы `update()` и `draw()` каждой анимации в каждом фрейме в основном цикле и каждая анимация принимает собственное решение по поводу сохранения текущего изображения или демонстрации следующего.

Выводы

В этой главе мы изучили механизмы, необходимые для класса анимации, создав наши собственные классы `SimpleAnimation` и `SimpleSpriteSheetAnimation`. Первый состоит из нескольких изображений, в то время как последний использует одно большое изображение, которое содержит несколько подизображений.

У этих двух классов различные инициализации, но остальные методы идентичны. Я объяснил процесс объединения двух классов путем создания общего абстрактного базового класса.

Затем познакомил вас с классом `Animation` и классом `SpriteSheetAnimation` в `pygame`, а также объяснил, что они только реализуют собственные версии метода `__init__()`, наследуя остальные от общего базового класса `PygAnimation`. Я завершил главу, показав демонстрационную программу, которая предоставляет примеры анимаций и анимаций листов спрайта.

15

СЦЕНЫ



Играм и программам часто требуется показывать пользователю различные сцены. В этом обсуждении я определю *сцену* как любую компоновку окон и связанные с ней взаимодействия пользователя, которые существенно отличаются от любых других. Например, в такой игре, как «Космические захватчики», может быть начальная сцена или сцена *приветствия*, главная сцена игры, сцена высоких результатов и, вероятно, завершающая или прощальная сцена.

В этой главе я рассмотрю два различных подхода к написанию программ, в которых есть несколько сцен. Для начала я познакомлю вас с методом конечного автомата, который хорошо работает в относительно небольших программах. Затем покажу полноценный объектно-ориентированный подход, в котором каждая сцена реализуется как объект под управлением менеджера сцен. Последний гораздо лучше подходит для крупных программ.

Концепция конечного автомата

В начале этой книги мы разработали программу имитации выключателя света. В главе 1 мы сначала реализовали выключатель света, используя процедурный код, а затем переписали его с использованием класса. В обоих случаях положение (или состояние) выключателя было представлено одной булевой переменной: `True` означало «включено», а `False` — «выключено».

Существует множество ситуаций, в которых программа или объект могут находиться в одном из нескольких различных состояний, и на основании текущего состояния должен выполняться соответствующий код. Например, рассмотрим ряд шагов, связанных с использованием банкомата. Существует начальное состояние (приветствие), затем вам надо вставить карту, после чего вам предлагают ввести пин-код, выбрать необходимое действие и так далее. В любой момент вам может понадобиться вернуться на шаг назад или даже начать заново. Общий подход к реализации состоит в использовании *конечного автомата*.

Конечный автомат

Модель, которая представляет поток выполнения и управляет им через ряд состояний.

Реализация конечного автомата включает в себя:

- сбор предопределенных состояний, обычно выраженных в виде констант, чьи значения являются строками, состоящими из слова или короткой фразы, которые описывают, что происходит в этом состоянии;
- одну переменную для отслеживания текущего состояния;
- начальное состояние (из набора предопределенных состояний);
- набор четко определенных переходов между состояниями.

Конечный автомат может находиться только в одном состоянии в любое заданное время, но способен переходить в новое состояние, обычно на основании конкретных входных данных от пользователя.

В главе 7 мы рассмотрели классы кнопок GUI в пакете `pygame.widgets`. При наведении мыши на кнопку и нажатии на нее пользователь видит различные изображения: «вверх», «мышь наведена» и «вниз», что соответствует различным состояниям кнопки.

Переключение изображения осуществляется в методе `handleEvent()` (который вызывается каждый раз, когда происходит событие). Давайте подробнее рассмотрим, как это реализуется. Метод `handleEvent()` создается в качестве конечного автомата. Состояние хранится в переменной экземпляра `self.state`. Каждая кнопка начинает работу в состоянии «вверх», показывая картинку «вверх». Когда пользователь помещает курсор на кнопку, мы демонстрируем изображение «мышь наведена», и код переходит в состояние «мышь наведена». Когда пользователь щелкает по кнопке, он видит картинку «вниз», и код переходит в состояние «вниз» (внутренне называется состоянием *готовности*). Когда пользователь отпускает кнопку мыши (щелкает вверх), мы вновь демонстрируем изображение «мышь наведена», и код переключается обратно на состояние «мышь наведена» (а `handleEvent()` возвращает `True`, чтобы обозначить, что был выполнен щелчок). Если затем пользователь уводит курсор с кнопки, мы вновь демонстрируем изображение «вверх» и переходим обратно в состояние «вверх».

Далее я покажу вам, как мы можем использовать конечный автомат для представления различных сцен, с которыми сталкивается пользователь в более крупной программе. В качестве общего примера у нас будут следующие сцены: *Приветствие* (начальная), *Игра* и *Конец*. Мы создадим набор констант, которые представляют различные состояния, создадим переменную с именем `state` и присвоим ей значение начального состояния:

```
STATE_SPLASH = 'splash'
STATE_PLAY = 'play'
STATE_END = 'end'
state = STATE_SPLASH # инициализируем до начального
                      # состояния
```

Чтобы выполнять различные действия в различных состояниях, в основном цикле программы мы используем структурный компонент `if/elif/elif/.../else`, который выполняет ветвление на основе текущего состояния переменной `state`:

```
while True:
    if state == STATE_SPLASH:
        # Делайте здесь все, что вы хотите сделать в состоянии
        # Приветствие
    elif state == STATE_PLAY:
        # Делайте здесь все, что вы хотите сделать в состоянии Игра
    elif state == STATE_END:
        # Делайте здесь все, что вы хотите сделать в состоянии Конец
    else:
        raise ValueError('Unknown value for state: ' + state)
```

Поскольку `state` изначально установлен в значение `STATE_SPLASH`, будет выполняться только первая ветвь оператора `if`.

Идея конечного автомата заключается в том, что при определенных обстоятельствах, обычно запускаемых каким-то событием, программа изменяет свое состояние, присваивая различные значения переменной `state`. Например, начальная сцена Приветствия может просто показывать заставку игры с кнопкой **Start**. Когда пользователь щелкает по кнопке **Start**, игра выполняет оператор присваивания, который изменяет значение переменной `state` для перехода в состояние Игра:

```
state = STATE_PLAY
```

Как только эта строка запущена, активируется лишь код в первом `elif`, и будет выполняться абсолютно другой код: код для отображения и реагирования на состояние Игра.

Аналогичным образом, каждый раз, когда программа достигает конечного состояния игры, она будет выполнять следующую строку для перехода в состояние Конец:

```
state = STATE_END
```

С этого момента каждый раз, когда программа проходит цикл `while`, будет выполняться код второй ветви `elif`.

Таким образом, у конечного автомата есть набор состояний, одна переменная для отслеживания состояния, в котором находится программа, и набор событий, которые приводят к переходу программы из одного состояния в другое. Поскольку существует одна переменная, которая отслеживает состояние, программа может находиться только в одном из состояний в каждый момент времени. Различные действия, которые выполняет

пользователь (щелчок по кнопке, нажатие клавиши, перетаскивание элемента и так далее), или другие события (такие как завершение времени таймера) могут привести к переходу программы из одного состояния в другое. В зависимости от состояния, в котором находится программа, она может ожидать различные события и, как правило, будет выполнять разный код.

Пример ругате с конечным автоматом

Далее мы создадим игру «Камень-ножницы-бумага», в которой используется конечный автомат. Пользователь выбирает камень, бумагу или ножницы; затем компьютер случайным образом выбирает из трех. Если человек и компьютер выбирают один и тот же элемент, это ничья. В противном случае одно очко присуждается игроку или компьютеру, согласно следующим правилам:

- камень затупляет ножницы;
- ножницы режут бумагу;
- бумага заворачивает камень.

Пользователь увидит игру в виде трех сцен: открывающая сцена Приветствия (рис. 15.1), сцена Игры (рис. 15.2) и сцена Результатов (рис. 15.3).



Рис. 15.1. Сцена Приветствия игры «Камень-ножницы-бумага»

Сцена Приветствия ждет, когда пользователь щелкнет по кнопке **Start**.

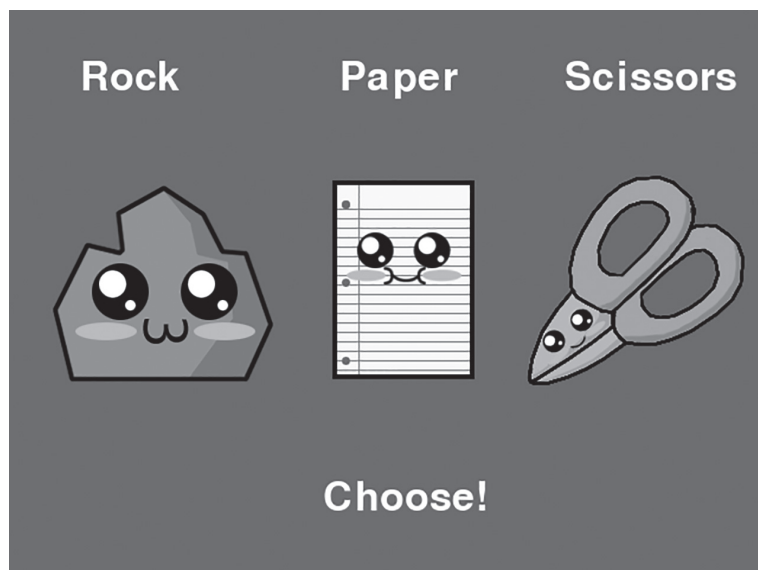


Рис. 15.2. Сцена Игры «Камень-ножницы-бумага»

Именно в сцене Игры пользователь делает выбор. После того как пользователь щелкнул по пиктограмме, обозначив свой выбор, компьютер делает собственный произвольный выбор.

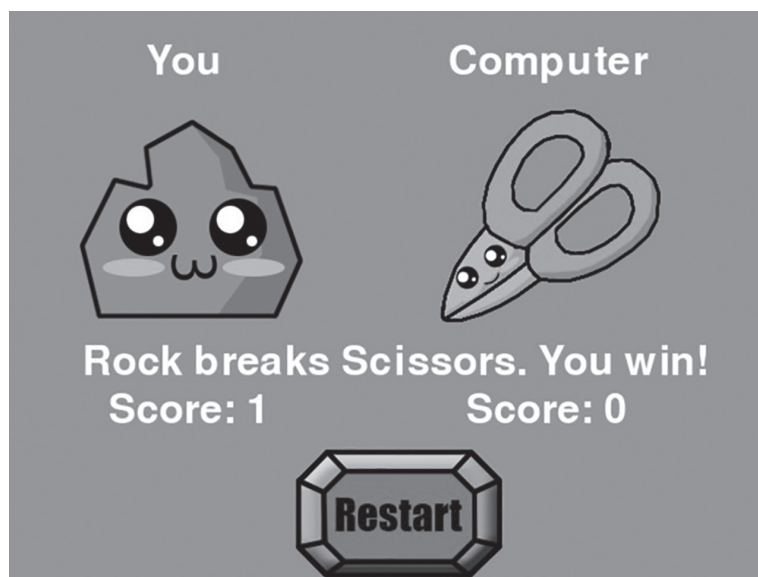


Рис. 15.3. Сцена Результатов игры «Камень-ножницы-бумага»

Сцена результатов демонстрирует результат раунда и очки. Она ждет, когда пользователь щелкнет по кнопке **Restart**, чтобы сыграть еще один раунд.

В этой игре каждое значение `state` соответствует различным сценам. На рис. 15.4 изображена *диаграмма состояний*, которая демонстрирует состояния и переходы (действия или события, которые приводят к переходу программы из одного состояния в другое).

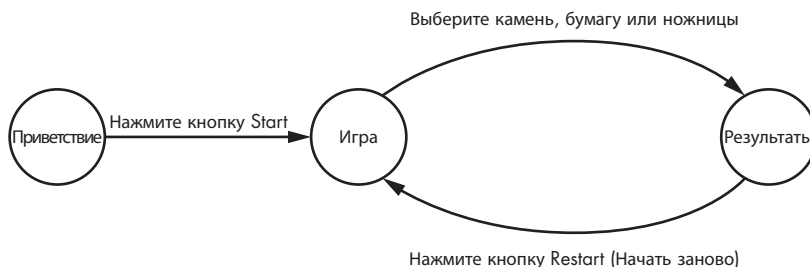


Рис. 15.4. Диаграмма состояний игры «Камень-ножницы-бумага»

При бездействии (ожидании пользователя) текущая сцена будет, как правило, оставаться неизменной. То есть внутри цикла основного события программа обычно не меняет значение переменной `state`. (Состояние может измениться, когда завершается таймер, но это редкое событие.) Данная игра начинается со сцены Приветствия и, когда пользователь нажимает на кнопку **Start**, переходит к сцене Игры. Затем ход игры чередуется между сценами Игры и Результаты. Хотя это простой пример, диаграмма состояний может оказаться очень полезной для понимания потока более сложных программ.

В листинге 15.1 представлен код программы «Камень-ножницы-бумага», при этом шаблонный код был опущен для экономии места.

Файл: `RockPaperScissorsStateMachine/RockPaperScissors.py`

```
# Камень-ножницы-бумага в ругате
# Демонстрация конечного автомата

--- пропуск ---

ROCK = 'Rock'
PAPER = 'Paper'
SCISSORS = 'Scissors'
```

```

# Настраиваем константы для каждого из трех состояний
STATE_SPLASH = 'Splash' ❶
STATE_PLAYER_CHOICE = 'PlayerChoice'
STATE_SHOW_RESULTS = 'ShowResults'

#3 - Инициализируем окружение pygame

--- пропуск ---

#4 - Загружаем элементы: изображения, звуки и т. д.

--- пропуск ---

#5 - Инициализируем переменные
playerScore = 0
computerScore = 0
state = STATE_SPLASH ❷ # начальное состояние

#6 - Бесконечный цикл
while True:

    #7 - Проверяем наличие событий и обрабатываем их
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        if state == STATE_SPLASH: ❸
            if startButton.handleEvent(event):
                state = STATE_PLAYER_CHOICE

        elif state == STATE_PLAYER_CHOICE: ❹ # разрешаем пользователю
                                                # выбрать
            playerChoice = '' # указываем, что выбор пока не был сделан
            if rockButton.handleEvent(event):
                playerChoice = ROCK
                rpsCollectionPlayer.replace(ROCK)

            elif paperButton.handleEvent(event):
                playerChoice = PAPER
                rpsCollectionPlayer.replace(PAPER)

            elif scissorButton.handleEvent(event):
                playerChoice = SCISSORS
                rpsCollectionPlayer.replace(SCISSORS)

            if playerChoice != '': # игрок сделал выбор, делаем выбор
                                    # за компьютер

```

```

# Компьютер выбирает из кортежа ходов
rps = (ROCK, PAPER, SCISSORS)
computerChoice = random.choice(rps) # компьютер
                                         # делает выбор
rpsCollectionComputer.replace(computerChoice)

# Оценить игру
if playerChoice == computerChoice: # ничья
    resultsField.setValue('It is a tie!')
    tieSound.play()

    elif playerChoice == ROCK and computerChoice
                                         ==SCISSORS:
        resultsField.setValue('Rock breaks Scissors.
                                You win!')
        playerScore = playerScore + 1
        winnerSound.play()

elif playerChoice == ROCK and computerChoice == PAPER:
    resultsField.setValue('Rock is covered by Paper.
                            You lose.')
    computerScore = computerScore + 1
    loserSound.play()

elif playerChoice == SCISSORS and computerChoice ==
                                         PAPER:
    resultsField.setValue('Scissors cuts Paper.
                            You win!')
    playerScore = playerScore + 1
    winnerSound.play()

elif playerChoice == SCISSORS and computerChoice ==
                                         ROCK:
    resultsField.setValue('Scissors crushed by Rock.
                            You lose.')
    computerScore = computerScore + 1
    loserSound.play()

elif playerChoice == PAPER and computerChoice == ROCK:
    resultsField.setValue('Paper covers Rock.
                            You win!')
    playerScore = playerScore + 1
    winnerSound.play()

elif playerChoice == PAPER and computerChoice ==
                                         SCISSORS:
    resultsField.setValue('Paper is cut by Scissors.
                            You lose.')

```



```

        computerScore = computerScore + 1
        loserSound.play()

        # отображаем очки пользователя
        playerScoreCounter.setValue('Your Score: ' +
                                    str(playerScore))

        # отображаем очки компьютера
        computerScoreCounter.setValue('Computer Score: ' +
                                     str(computerScore))

        state = STATE_SHOW_RESULTS # меняем состояние

    elif state == STATE_SHOW_RESULTS: ❸
        if restartButton.handleEvent(event):
            state = STATE_PLAYER_CHOICE # меняем состояние

    else:
        raise ValueError('Unknown value for state:', state)

#8 - Выполняем действия "в рамках фрейма"
if state == STATE_PLAYER_CHOICE:
    messageField.setValue('      Rock      Paper      Scissors')
elif state == STATE_SHOW_RESULTS:
    messageField.setValue('You                        Computer')

#9 - Очищаем окно
window.fill(GRAY)

#10 - Рисуем все элементы окна
messageField.draw()

    if state == STATE_SPLASH: ❹
        rockImage.draw()
        paperImage.draw()
        scissorsImage.draw()
        startButton.draw()

# рисуем выборы игрока
elif state == STATE_PLAYER_CHOICE: ❺
    rockButton.draw()
    paperButton.draw()
    scissorButton.draw()
    chooseText.draw()

# рисуем результаты
elif state == STATE_SHOW_RESULTS: ❻
    resultsField.draw()
    rpsCollectionPlayer.draw()

```

```

rpsCollectionComputer.draw()
playerScoreCounter.draw()
computerScoreCounter.draw()
restartButton.draw()

#11 - Обновляем окно
pygame.display.update()

#12 - Делаем паузу
clock.tick(FRAMES_PER_SECOND) # ожидание pygame

```

Листинг 15.1. Игра «Камень-ножницы-бумага»

В этом листинге я вырезал код, который создает изображения, кнопки и текстовые поля для сцен Приветствия, Игры и Результатов. Загружаемые файлы для книги содержат весь исходный код и всю связанную с ним графику.

Прежде чем программа начнет основной цикл, мы определяем три состояния ❶, создаем экземпляры, загружаем все элементы экрана и устанавливаем начальное состояние ❷.

Выполняем проверки различных событий в зависимости от состояния, в котором находится программа. В состоянии Приветствия мы только проверяем наличие щелчка по кнопке **Start** ❸. В состоянии Игры проверяем наличие щелчка по кнопкам пиктограмм Камень, Бумага или Ножницы ❹. В состоянии Результатов проверяем только наличие щелчка по кнопке **Restart** ❺.

Нажатие кнопки или выбор в одной сцене меняет значение переменной `state` и, следовательно, перемещает игру в другую сцену. В нижней части основного цикла ❻ ❼ ❸ мы рисуем различные элементы экрана в зависимости от текущего состояния программы.

Этот метод хорошо работает для небольшого количества состояний/сцен. Однако в программе с более сложными правилами или в той, где много сцен и/или состояний, отслеживание того, что необходимо сделать, бывает чрезвычайно сложным. Вместо этого мы можем воспользоваться многими методами объектно-ориентированного программирования, представленными ранее в этой книге, и выстроить различные архитектуры на основе независимых сцен, которые управляются объектом диспетчера объектов.

ДемOVERсия программы, использующая менеджер сцен

В качестве демонстрации создадим программу Scene Demo, которая содержит три простые сцены: Сцену А, Сцену В и Сцену С. Идея заключается в том, что в любой сцене вы можете щелкнуть по кнопке, чтобы перейти в другую. На рис. 15.5–15.7 показаны скриншоты этих трех сцен.

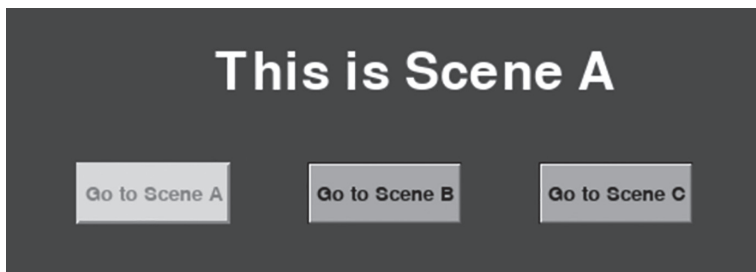


Рис. 15.5. Что видит пользователь в сцене А

Из Сцены А вы можете попасть в Сцену В или в Сцену С.

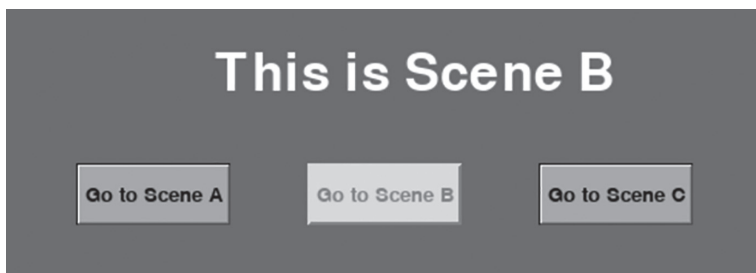


Рис. 15.6. Что видит пользователь в сцене В

Из Сцены В вы можете попасть в Сцену А или в Сцену С.



Рис. 15.7. Что видит пользователь в сцене С

Из Сцены С вы можете попасть в Сцену А или в Сцену В.

Структура папки проекта показана на рис. 15.8. Предполагается, что вы уже установили модули `pygwidgets` и `pyghelpers` в папке *site-packages*.







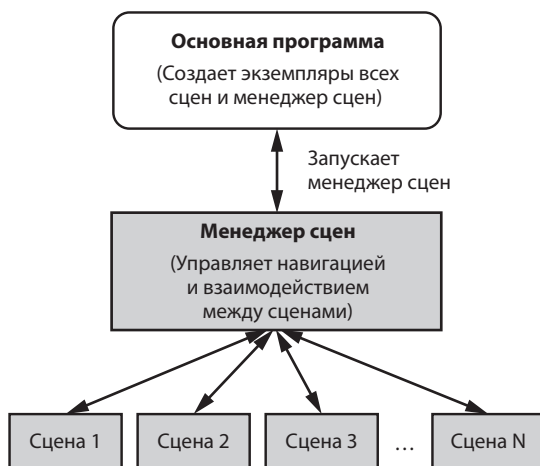
Name
 Constants.py
 Main_SceneDemo.py
 SceneA.py
 SceneB.py
 SceneC.py
 SceneExample.py

Рис. 15.8. Папка проекта, демонстрирующая основную программу и различные файлы сцен

Main_SceneDemo.py — это основная программа. *Constants.py* содержит несколько констант, которые совместно используются основной программой и всеми сценами. *Scene A.py*, *Scene B.py* и *Scene C.py* — это фактические сцены, каждая из которых содержит соответствующий класс сцены. *SceneExample.py* — файл шаблона, демонстрирующий, как может выглядеть файл типичной сцены. Он не используется в этой программе, но вы можете обратиться к нему, чтобы сформировать понимание основ написания типичной сцены.

На рис. 15.9 показаны взаимоотношения объектов в программе.



Все сцены наследуют от базового класса *Scene*

Рис. 15.9. Иерархия объектов в проекте

Давайте посмотрим, каким образом различные части программы, использующие менеджер сцен, совместно работают, начиная с основной программы.

Основная программа

Основная программа будет индивидуальна для каждого проекта. Ее цель — инициализировать среду pygame, создать экземпляры всех сцен, экземпляр SceneMgr, а затем передать управление менеджеру сцен oSceneMgr.

В листинге 15.2 представлен код демоверсии основной программы.

Файл: SceneDemo/Main_SceneDemo.py

```
# Scene Demo основной программы с тремя сценами

--- пропуск ---

#1 - Импортируем пакеты
import pygame
❶ import pyghelpers

from SceneA import *
from SceneB import *
from SceneC import *

#2 - Определяем константы
❷ WINDOW_WIDTH = 640
WINDOW_HEIGHT = 180
FRAMES_PER_SECOND = 30

#3 - Инициализируем окружение pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))

#4 - Загружаем элементы: изображения, звуки и т. д.

#5 - Инициализируем переменные
# создаем экземпляры всех сцен и сохраняем их в список
❸ scenesList = [SceneA(window),
                 SceneB(window),
                 SceneC(window)]

# создаем менеджер сцен, передавая список сцен и FPS
❹ oSceneMgr = pyghelpers.SceneMgr(scenesList, FRAMES_PER_SECOND)
```

```
# Просим менеджер сцен начать выполнение
5 oSceneMgr.run()
```

Листинг 15.2. Образец основной программы, использующей менеджер сцен

Код основной программы относительно короткий. Мы начинаем с импорта `pyhelpers`, затем всех сцен (в данном случае Сцены А, Сцены В и Сцены С) ❶. Потом мы определяем еще несколько констант, инициализируем `pygame` и создаем окно ❷. Далее создаем экземпляр каждой сцены и сохраняем все сцены в списке ❸. После выполнения этой строки у нас появляется инициализированный объект для каждой сцены.

Затем создаем экземпляр объекта менеджера сцен (`oSceneMgr`) ❹ из класса `SceneMgr`. При этом нам необходимо передать два значения:

- список сцен, чтобы менеджер сцен знал обо всех сценах. Первая сцена в списке используется в качестве начальной для программы;
- фреймы в секунду (частота фреймов), которые программа должна поддерживать.

И наконец, мы инструктируем менеджер сцен начать работу, вызывая его метод `run()` ❺. Менеджер сцен всегда обслуживает одну сцену в качестве текущей — ту, что видит пользователь и с которой он взаимодействует.

Обратите внимание, что при таком подходе основная программа реализует инициализацию типичной программы `pygame`, но *не* создает основной цикл. Вместо этого основной цикл встроен в сам менеджер сцен.

Создаем сцены

Чтобы понять взаимодействие между менеджером сцен и любой отдельной сценой, я объясню, как создается типичная сцена.

Каждый раз во время прохождения цикла менеджер сцен вызывает предопределенный набор методов в текущей сцене, которые предназначены для обработки событий, выполнения любых действий за фрейм и изображения всего, что должно быть нарисовано в этой сцене. Следовательно, код каждой сцены должен быть разбит на эти методы. Подход использует полиморфизм: каждая сцена должна реализовать общий набор методов.

Методы для реализации в каждой сцене

Каждая сцена реализуется как класс, который наследует от базового класса `Scene`, определенного в файле `pyghelpers.py`. Следовательно, каждая сцена должна импортировать `pyghelpers`. Как минимум сцены должны содержать метод `__init__()` и обязаны переопределять методы `getSceneKey()`, `handleInputs()` и `draw()` из базового класса.

В каждой сцене должен быть уникальный *ключ сцены* — строка, используемая менеджером сцен для идентификации каждой сцены. Я рекомендую вам создать файл с таким именем, как `Constants.py`, который содержит все ключи для всех сцен, и импортировать его в файл каждой сцены. Например, файл `Constants.py` для образца программы содержит:

```
# Ключи сцен (любые уникальные значения):
SCENE_A = 'scene A'
SCENE_B = 'scene B'
SCENE_C = 'scene C'
```

Во время его инициализации менеджер сцен вызывает метод `getSceneKey()` каждой сцены, который просто возвращает уникальный ключ сцены. Затем менеджер сцен создает внутренний словарь ключей и объектов сцен. Когда какая-либо сцена в программе хочет переключиться на другую, он вызывает `self.goToScene()` (описан в следующем разделе) и передает ключ сцены целевой сцене. Менеджер сцен использует этот ключ в словаре, чтобы найти связанный объект сцены; затем он устанавливает новый объект сцены в качестве текущей сцены и вызывает его методы.

Каждая сцена должна содержать свою версию `handleInputs()`, чтобы обрабатывать любые события, которые обычно обрабатываются в основном цикле, и собственную версию `draw()`, чтобы рисовать в окне все, что хочет. Если ваша сцена не переопределяет эти два метода, она не будет реагировать ни на какие события и не сможет ничего рисовать в окне.

Давайте поближе познакомимся с четырьмя методами, которые вам необходимо реализовать для каждой сцены.

```
def __init__(self, window):
```

Каждая сцена должна начинаться с метода `__init__()`.

Параметр `window` — это окно, в котором рисует программа.

Ваш метод нужно начинать со следующего оператора, чтобы сохранить параметр `window` для использования в методе `draw()`:

```
self.window = window
```

После этого вы можете включить любую другую необходимую вам инициализацию кода, например код для создания экземпляров кнопок и текстовых полей, загрузки изображений и звуков и так далее.

def getSceneKey(self):

Этот метод необходимо реализовывать в каждой написанной вами сцене. Он должен возвращать уникальный ключ сцены, связанный с этой сценой.

def handleInputs(self, events, keyPressedList):

Этот метод необходимо реализовывать в каждой написанной вами сцене. Он должен делать все необходимое для работы с событиями или ключами. Параметр `events` — список событий, произошедших с момента последнего фрейма, а `keyPressedList` — список булевых выражений, представляющих состояние всех клавиш клавиатуры (`True` обозначает нажатую клавишу). Чтобы определить, нажата ли конкретная клавиша, вы должны использовать константу вместо целочисленного индекса. Константы, представляющие все клавиши клавиатуры, доступны в документации pygame (<https://www.pygame.org/docs/ref/key.html>).

Ваша реализация этого метода обязана содержать цикл `for`, который перебирает все события в переданном списке. Если хотите, он также может содержать код для реализации непрерывного режима обработки клавиатуры, как было описано в главе 5.

def draw(self):

Этот метод необходимо реализовывать в каждой написанной вами сцене. Он должен рисовать все, что должно быть изображено в этой сцене.

Менеджер сцен также вызывает следующие методы в каждой сцене. В базовом классе `Scene` каждый из этих методов

содержит простой оператор `pass`, поэтому они ничего не делают. Вы можете переопределить любой из них или все, чтобы выполнить код, который необходим для конкретной сцены.

`def enter(self, data):`

Этот метод вызывается после того, как менеджер сцен выполнил переход в эту сцену. В нем есть один параметр `data` со значением по умолчанию `None`. Если `data` не равен `None`, тогда содержащаяся в нем информация была отправлена из предыдущей сцены, когда она вызывала `goToScene()` (описан в следующем разделе). Значение `data` может принимать любой формат, от одной строки или числового значения, списка или словаря до объекта, при условии, что сцена выхода и сцена входа согласовывают тип передаваемых данных. Метод `enter()` должен делать все необходимое, когда этой сцене собираются передать управление.

`def update(self):`

Этот метод вызывается в каждом фрейме. Здесь вы можете выполнить любые действия, которые вы бы выполнили в шаге 8 исходного 12-шагового шаблона, представленного в главе 5. Например, заставить этот метод перемещать изображения по экрану, проверять наличие коллизий и так далее.

`def leave(self):`

Этот метод вызывается менеджером сцен каждый раз, когда программа собирается перейти к другой сцене. Он должен выполнить любую очистку, которая необходима перед уходом, например записать информацию в файл.

Навигация между сценами

Менеджер сцен и базовый класс `Scene` предоставляют простой способ навигации между сценами. Когда программа хочет выполнить переход к другой сцене, текущая должна следующим образом вызвать собственный метод `goToScene()`, который находится в наследуемом базовом классе `Scene`:

```
self.goToScene(nextSceneKey, data)
```

Метод `goToScene()` сообщает менеджеру сцен, что вы хотите перейти к другой сцене, чей ключ сцены — это `nextSceneKey`. Вы должны сделать ключи всех сцен доступными через такой файл, как *Constants.py*. Параметр `data` — это любая дополнительная информация, которую вы хотите передать в следующую сцену. Если не нужно передавать никакие данные, можете исключить этот аргумент.

Типичные вызовы будут выглядеть следующим образом:

```
self.goToScene(SOME_SCENE_KEY) # не надо передавать данные

# или

self.goToScene(ANOTHER_SCENE_KEY, data=someValueOrValues)
# переходим к сцене и передаем данные
```

Значение `data` может принимать любую форму при условии, что сцены выхода и входа понимают формат. В ответ на этот вызов, прежде чем покинуть текущую сцену, менеджер сцен вызывает метод `leave()`. Перед активацией следующей сцены менеджер сцен вызывает метод `enter()` этой сцены и передает значение `data` новой сцене.

Выход из программы

Менеджер сцен предоставляет три различных способа, с помощью которых пользователь может выйти из выполняемой в данный момент программы:

- с помощью щелчка по кнопке закрытия в верхней части окна;
- с помощью нажатия клавиши **Esc**;
- посредством дополнительного механизма, такого как кнопка **Quit**. В этом случае выполните следующий вызов (который также встроен в базовый класс `Scene`):

```
self.quit() # выходим из программы
```

Типичная сцена

В листинге 15.3 продемонстрирован пример типичной сцены — это файл *Scene A.py*, который реализует Сцену А в демонстрационной программе, показанной на рис. 15.5. Помните, что основной цикл реализуется менеджером сцен. Внутри цикла менеджер сцен вызывает методы `handleInputs()`, `update()` и `draw()` для текущей сцены.

Файл: SceneDemo/SceneA.py

```
# Сцена А

import pygwidgets
import pyghelpers
import pygame
from pygame.locals import *
from Constants import *

class SceneA(pyghelpers.Scene):
❶   def __init__(self, window):

        self.window = window

        self.messageField = pygwidgets.DisplayText(self.window,
            (15, 25), 'This is Scene A', fontSize=50,
            textColor=WHITE, width=610, justified='center')

        self.gotoAButton = pygwidgets.TextButton(self.window,
            (250, 100), 'Go to Scene A')

        self.gotoBButton = pygwidgets.TextButton(self.window,
            (250, 100), 'Go to Scene B')

        self.gotoCButton = pygwidgets.TextButton(self.window,
            (400, 100), 'Go to Scene C')

        self.gotoAButton.disable()

❷   def getSceneKey(self):
        return SCENE_A

❸   def handleInputs(self, eventsList, keyPressedList):
        for event in eventsList:
            if self.gotoBButton.handleEvent(event):
❹                self.goToScene(SCENE_B)
            if self.gotoCButton.handleEvent(event):
❺                self.goToScene(SCENE_C)

        --- (тестовый код для отправки сообщений) ---

❻   def draw(self):
        self.window.fill(GRAYA)
        self.messageField.draw()
        self.gotoAButton.draw()
        self.gotoBButton.draw()
        self.gotoCButton.draw()

        --- (тестовый код для ответа на сообщения) ---
```

Листинг 15.3. Типичная сцена (Сцена А в программе Scene Demo)

В методе `__init__()` ❶ мы сохраняем параметр `window` в переменной экземпляра. Затем создаем поле `DisplayText`, чтобы отобразить название, и несколько `TextButton`, чтобы разрешить навигацию к другим сценам.

Метод `getSceneKey()` ❷ просто возвращает уникальный ключ сцены (находящийся в *Constants.py*) для этой сцены. В методе `handleInputs()` ❸, если пользователь щелкнул по кнопке для другой сцены, мы вызываем метод навигации `self.goToScene()` ❹ ❺ для передачи управления новой сцене. В методе `draw()` ❻ заполняем фон, рисуем поле сообщения и кнопки. Этот пример сцены делает очень мало, поэтому нам не нужно писать собственные методы `enter()`, `update()` и `leave()`. Их вызовы будут обрабатываться методами с теми же именами в базовом классе `Scene`, и этим методам не нужно ничего делать, они просто выполняют оператор `pass`.

К двум другим файлам сцен относятся *Scene B.py* и *Scene C.py*. Отличия заключаются в отображаемых заголовках, нарисованных кнопках и эффектах нажатия по кнопкам для перехода к подходящей новой сцене.

Игра «Камень-ножницы-бумага», использующая сцены

Давайте создадим альтернативную реализацию игры «Камень-ножницы-бумага», используя менеджер сцен. Для пользователя игра будет выглядеть абсолютно так же, как и предыдущая версия конечного автомата. Мы создадим сцену Приветствия, сцену Игры и сцену Результаты.

Весь исходный код доступен, поэтому я не стану рассматривать каждый файл Python. Сцена Приветствия — это всего лишь фоновая картинка с кнопкой **Start**. Когда пользователь нажимает кнопку **Start**, код выполняет `goToScene(SCENE_PLAY)`, чтобы перейти к сцене Игры. В ней пользователю представлен набор изображений (камень, бумага и ножницы), и его просят выбрать одно из них. Щелчок по изображению передает управление сцене Результаты. В листинге 15.4 содержится код сцены Игры.

Файл: RockPaperScissorsWithScenes/ScenePlay.py

```
# Сцена Игры
# Игрок выбирает камень, бумагу или ножницы

import pygwidgets
import pyghelpers
import pygame
from Constants import *
import random

class ScenePlay(pyghelpers.Scene):
    def __init__(self, window):

        self.window = window

        self.RPSTuple = (ROCK, PAPER, SCISSORS)

        --- пропуск ---

    def getSceneKey(self): ❶
        return SCENE_PLAY

    def handleInputs(self, eventsList, keyPressedList): ❷
        playerChoice = None

        for event in eventsList:
            if self.rockButton.handleEvent(event):
                playerChoice = ROCK

            if self.paperButton.handleEvent(event):
                playerChoice = PAPER

            if self.scissorButton.handleEvent(event):
                playerChoice = SCISSORS

        if playerChoice is not None: ❸ # пользователь сделал выбор
            computerChoice = random.choice(self.RPSTuple)
                                # компьютер делает выбор
            dataDict = {'player': playerChoice,
                        'computer': computerChoice} ❹
            self.goToScene(SCENE_RESULTS, dataDict) 5
                                # переходим к сцене Результаты

        # Не нужно включать метод update, по умолчанию используется
        # унаследованный, который ничего не делает
```

```

def draw(self):
    self.window.fill(GRAY)
    self.titleField.draw()
    self.rockButton.draw()
    self.paperButton.draw()
    self.scissorButton.draw()
    self.messageField.draw()

```

Листинг 15.4. Сцена Игры в игре «Камень-ножницы-бумага»

Я вырезал код для создания текстовых полей и кнопок камня, бумаги и ножниц. Метод `getSceneKey()` ❶ просто возвращает ключ сцены для этой сцены.

Самым важным методом является `handleInputs()` ❷, который вызывается в каждом фрейме. Если какая-то из кнопок была нажата, мы устанавливаем переменную с именем `playerChoice` в соответствующую константу ❸ и выполняем произвольный выбор для компьютера. Затем берем выбор пользователя и выбор компьютера и создаем простой словарь ❹, включающий оба выбора, чтобы мы могли передать эту информацию в качестве данных сцене Результаты. И наконец, для перехода к сцене Результаты вызываем `goToScene()` и передаем словарь ❺.

Менеджер сцен получает этот вызов, вызывает `leave()` для текущей сцены (Игра), меняет текущую сцену на новую сцену (Результаты) и вызывает `enter()` для новой сцены (Результаты). Он передает данные из сцены выхода в метод `enter()` новой сцены.

В листинге 15.5 содержится код сцены Результаты. Здесь много кода, но большая его часть имеет дело с отображением соответствующих пиктограмм и оценкой результатов раунда.

Файл: `RockPaperScissorsWithScenes/SceneResults.py`

```

# Сцена Результаты
# Игроку демонстрируются результаты текущего раунда

import pygwidgets
import pyghelpers
import pygame
from Constants import *

class SceneResults(pyghelpers.Scene):
    def __init__(self, window, sceneKey):
        self.window = window

```

```

self.playerScore = 0
self.computerScore = 0

```

❶

```

self.rpsCollectionPlayer = pygwidgets.ImageCollection(
    window, (50, 62),
    {ROCK: 'images/Rock.png',
     PAPER: 'images/Paper.png',
     SCISSORS: 'images/Scissors.png'}, '')

self.rpsCollectionComputer = pygwidgets.ImageCollection(
    window, (350, 62),
    {ROCK: 'images/Rock.png',
     PAPER: 'images/Paper.png',
     SCISSORS: 'images/Scissors.png'}, '')

self.youComputerField = pygwidgets.DisplayText(
    window, (22, 25),
    'You Computer',
    fontSize=50, textColor=WHITE,
    width=610, justified='center')

self.resultsField = pygwidgets.DisplayText(
    self.window, (20, 275), '',
    fontSize=50, textColor=WHITE,
    width=610, justified='center')

self.restartButton = pygwidgets.CustomButton(
    self.window, (220, 310),
    up='images/restartButtonUp.png',
    down='images/restartButtonDown.png'
    over='images/restartButtonHighlight.png')

self.playerScoreCounter = pygwidgets.DisplayText(
    self.window, (86, 315), 'Score:',
    fontSize=50, textColor=WHITE)

self.computerScoreCounter = pygwidgets.DisplayText(
    self.window, (384, 315), 'Score:',
    fontSize=50, textColor=WHITE)

# Звуки
self.winnerSound = pygame.mixer.Sound("sounds/ding.wav")
self.tieSound = pygame.mixer.Sound("sounds/push.wav")
self.loserSound = pygame.mixer.Sound("sounds/buzz.wav")

```

❷

```

def enter(self, data):
    # данные являются словарем (получен из сцены Игры), который
    # выглядит следующим образом:
    #     {'player': playerChoice, 'computer': computerChoice}

```

```

playerChoice = data['player']
computerChoice = data['computer']

# Настраиваем изображения игрока и компьютера
3 self.rpsCollectionPlayer.replace(playerChoice)
  self.rpsCollectionComputer.replace(computerChoice)

# оцениваем условия игры выиграл/проиграл/ничья
4 if playerChoice == computerChoice:
    self.resultsField.setValue("It's a tie!")
    self.tieSound.play()

elif playerChoice == ROCK and computerChoice == SCISSORS:
    self.resultsField.setValue("Rock breaks Scissors.
                                You win!")
    self.playerScore = self.playerScore + 1
    self.winnerSound.play()

--- пропуск ---

# отображаем очки игрока и компьютера
self.playerScoreCounter.setValue(
    'Score: ' + str(self.playerScore))
self.computerScoreCounter.setValue(
    'Score: ' + str(self.computerScore))

5 def handleInputs(self, eventsList, keyPressedList):
    for event in eventsList:
        if self.restartButton.handleEvent(event):
            self.goToScene(SCENE_PLAY)

# Не нужно включать метод update, по умолчанию используется
# унаследованный, который ничего не делает

6 def draw(self):
    self.window.fill(OTHER_GRAY)
    self.youComputerField.draw()
    self.resultsField.draw()
    self.rpsCollectionPlayer.draw()
    self.rpsCollectionComputer.draw()
    self.playerScoreCounter.draw()
    self.computerScoreCounter.draw()
    self.restartButton.draw()

```

Листинг 15.5. Сцена Результатов в игре «Камень-ножницы-бумага»

Здесь я вырезал некоторую логику оценки игры. Метод `enter()` ② — самый важный в этом классе. Когда пользователь делает выбор в сцене Игры, программа переходит к сцене

Результатов. Сначала мы извлекаем выборы пользователя и компьютера, переданные из сцены Игры в виде словаря, который выглядит следующим образом:

```
{'player': playerChoice, 'computer': computerChoice}
```

В методе `__init__()` ❶ создаем объекты `ImageCollection` для игрока и компьютера, каждый из которых содержит изображения камня, бумаги и ножниц. В методе `enter()` ❷ мы используем метод `replace()` объекта `ImageCollection` ❸, чтобы показать изображения, которые представляют выборы игрока и компьютера.

Далее оценка достаточно проста ❹. Если компьютер и игрок сделали одинаковый выбор, у нас ничья, воспроизводится соответствующий звук. Если игрок выигрывает, мы увеличиваем его очки и проигрываем веселый звук. Если выигрывает компьютер, мы увеличиваем очки компьютера и воспроизводим грустный звук. Обновляем оценки игрока или компьютера и отображаем очки в соответствующих полях отображения текста.

После выполнения метода `enter()` (один раз в каждом раунде) в каждом фрейме менеджер сцен вызывает метод `handleInputs()` ❺. Когда пользователь щелкает по кнопке **Restart**, мы вызываем унаследованный метод `goToScene()`, чтобы вернуться обратно к сцене Игры.

Метод `draw()` ❻ рисует в окне все для этой сцены.

Здесь не требуется выполнять какую-либо дополнительную работу в каждом фрейме, поэтому нам не нужно писать метод `update()`. Когда менеджер сцен вызывает `update()`, запускается унаследованный метод в базовом классе `Scene` и просто выполняется оператор `pass`.

Взаимодействие между сценами

Менеджер сцен предоставляет набор методов, которые позволяют сценам взаимодействовать друг с другом, отправляя или запрашивая информацию. Это понадобится не всем программам, но бывает невероятно полезным. Менеджер сцен дает возможность каждой сцене:

- запрашивать информацию у другой сцены;
- отправлять информацию другой сцене;
- отправлять информацию всем сценам.

В следующих разделах я буду называть сцену, которую видит пользователь, *текущей*. Она отправляет информацию или запрашивает данные у *целевой* сцены. Все используемые для перехода методы реализуются в базовом классе `Scene`. Следовательно, у всех сцен (которые мы должны наследовать от базового класса `Scene`) есть доступ к этим методам с помощью `self.<метод>()`.

Запрашиваем информацию у целевой сцены

Чтобы запросить информацию у любой другой сцены, сцена вызывает унаследованный метод `request()` следующим образом:

```
self.request(targetSceneKey, requestID)
```

Этот вызов разрешает текущей сцене запрашивать информацию у целевой, идентифицируемой по ее ключу сцены (`targetSceneKey`). `requestID` однозначно идентифицирует информацию, которую вы запрашиваете. Значение, используемое для `requestID`, обычно будет константой, определенной в таком файле, как *Constants.py*. Вызов возвращает запрашиваемую информацию. Типичный вызов будет выглядеть следующим образом:

```
someData = self.request(SOME_SCENE_KEY, SOME_INFO_CONSTANT)
```

Это значит: «Выполнить запрос к сцене `SOME_SCENE_KEY`, спрашивая информацию, идентифицированную в `SOME_INFO_CONSTANT`». Данные возвращаются и присваиваются переменной `someData`.

Менеджер сцен действует как посредник: получает вызов `request()` и возвращает его в вызов `respond()` в целевой сцене. Чтобы целевая сцена смогла предоставлять информацию, следует применить метод `respond()` в классе этой сцены. Метод должен начинаться следующим образом:

```
def respond(self, requestID):
```

Типичный код метода `respond()` проверяет значение параметра `requestID` и возвращает соответствующие данные. Они могут быть отформатированы любым способом, согласованным с текущей и целевой сценами.

Отправляем информацию целевой сцене

Чтобы отправить информацию целевой сцене, текущая сцена вызывает унаследованный метод `send()` следующим образом:

```
self.send(targetSceneKey, sendID, info)
```

Это разрешает ей отправлять информацию целевой сцене, идентифицируемой по ее ключу (`targetSceneKey`). `sendID` однозначно идентифицирует отправляемую вами информацию. Параметр `info` — та информация, которую вы хотите отправить целевой сцене.

Типичный вызов будет выглядеть следующим образом:

```
self.send(SOME_SCENE_KEY, SOME_INFO_CONSTANT, data)
```

Это значит: «Отправить информацию сцене `SOME_SCENE_KEY`. Информация идентифицирована `SOME_INFO_CONSTANT` и находится в значении переменной `data`».

Менеджер сцен получает вызов `send()` и возвращает его в вызов `receive()` в целевой сцене. Чтобы разрешить сцене отправлять информацию другой сцене, вы должны реализовать метод `receive()` в классе вашей целевой сцены следующим образом:

```
def receive(self, receiveID, info):
```

Метод `receive()` может содержать структурный компонент `if/elif/elif/.../else`, если ему необходимо обработать различные значения для `receiveID`. Передаваемая информация может быть отформатирована любым способом, согласованным с текущей и целевой сценами.

Отправляем информацию всем сценам

Для дополнительного удобства сцена может отправлять информацию всем другим сценам, используя один метод `sendAll()`:

```
self.sendAll(sendID, info)
```

Он разрешает текущей сцене отправлять сведения остальным сценам. `sendID` однозначно идентифицирует эту информацию. Параметр `info` — та информация, которую вы хотите отправить всем сценам.

Типичный вызов будет выглядеть следующим образом:

```
self.sendAll(SOME_INFO_CONSTANT, data)
```

Это значит: «Отправить информацию всем сценам. Информация идентифицирована *SOME_INFO_CONSTANT* и находится в значении переменной *data*».

Для выполнения такой работы все сцены, кроме текущей, должны реализовывать метод `receive()`, как было описано в предыдущем разделе. Менеджер сцен отправляет сообщение всем сценам (кроме текущей). Текущая сцена может содержать метод `receive()` для информации, отправляемой другими сценами.

Проверяем взаимодействие между сценами

Демонстрационная программа сцен (со Сценой А, Сценой В и Сценой С), которая обсуждалась ранее в листингах 15.2 и 15.3, в каждой сцене содержит код, демонстрирующий вызовы `send()`, `request()` и `sendAll()`. Кроме того, каждая сцена реализует простые версии методов `receive()` и `respond()`. В демо-версии программы можно отправить сообщение другой сцене, нажав А, В или С. Нажав Х, вы отправите сообщение всем сценам. Нажав 1, 2 или 3, вы отправите запрос на получение данных от целевой сцены. Она отвечает строкой.

Реализация менеджера сцен

Здесь мы рассмотрим, как реализуется менеджер сцен. Однако один из важных уроков ООП заключается в том, что разработчик клиентского кода должен понимать не реализацию класса, а только интерфейс. Что касается менеджера сцен, вам нет необходимости разбираться, как он работает, достаточно знать, какие методы вы должны реализовать в ваших сценах, когда они вызываются и какие методы вы можете вызвать. Следовательно, если вам не интересна внутренняя работа, можете сразу перейти к выводам. Однако на случай, если вы хотите вникнуть в подробности, я в этом разделе разбираю детали реализации и попутно знакоблю с интересными методами, позволяющими осуществлять двустороннее взаимодействие между объектами.

Менеджер сцен реализуется в классе с именем `SceneMgr` в модуле `pyghelpers`. Как объяснялось ранее, в основной программе вы создаете один экземпляр менеджера сцен следующим образом:

```
oSceneMgr = SceneMgr(scenesList, FRAMES_PER_SECOND)
```

Последняя строка основной программы должны быть следующей:

```
oSceneMgr.run()
```

В листинге 15.6 содержится код метода `__init__()` класса `SceneMgr`.

```
--- пропуск ---

def __init__(self, scenesList, fps):

    # создаем словарь, каждая запись которого представляет собой ключ
    # сцены: объект сцены
    ❶ self.scenesDict = {}
    ❷ for oScene in scenesList:
        key = oScene.getSceneKey()
        self.scenesDict[key] = oScene

    # Первый элемент в списке используется как начальная сцена
    ❸ self.oCurrentScene = scenesList[0]
    self.framesPerSecond = fps

    # даем каждой сцене обратную ссылку в SceneMgr
    # Это разрешает всем сценам выполнять goToScene, request,
    # send или sendAll, что перенаправляется менеджеру сцен.
    ❹ for key, oScene in self.scenesDict.items():
        oScene._setRefToSceneMgr(self)
```

Листинг 15.6. Метод `__init__()` класса `SceneMgr`

Метод `__init__()` отслеживает все сцены в словаре ❶. Он перебирает список сцен, запрашивая у каждой сцены ее ключ сцены, и создает словарь ❷. Первый объект сцены в списке сцен запускается как начальная сцена ❸.

Последняя часть метода `__init__()` выполняет интересную работу. Менеджер сцен содержит ссылку на каждую сцену, поэтому он способен отправить сообщение абсолютно любой

сцене. Но каждая сцена должна иметь возможность отправлять сообщения менеджеру сцен. Для этого последний цикл `for` в методе `__init__()` вызывает специальный метод `_setRefToSceneMgr()` ❹, который располагается в базовом классе каждой сцены, и он передает `self`, что является ссылкой на менеджер сцен. Весь код этого метода состоит из единственной строки:

```
def _setRefToSceneMgr(self, oSceneMgr):
    --- пропуск ---
    self.oSceneMgr = oSceneMgr
```

Метод просто сохраняет эту ссылку обратно в менеджер сцен в переменной экземпляра `self.oSceneMgr`. Каждая сцена может использовать данную переменную, чтобы вызывать менеджера сцен. Немного позднее в этом разделе я покажу вам, как сцены используют это.

Метод `run()`

Для каждого создаваемого проекта вам необходимо написать небольшую основную программу, которая создает экземпляры менеджера сцен. Последний шаг в программе — вызов метода `run()` менеджера сцен. Именно здесь находится основной цикл всей программы. В листинге 15.7 содержится код этого метода.

```
def run(self):

    --- пропуск ---

    clock = pygame.time.Clock()

    #6 - Бесконечный цикл
    while True:

        ❶ keysDownList = pygame.key.get_pressed()

        #7 - Проверяем наличие событий и обрабатываем их
        ❷ eventsList = []
        for event in pygame.event.get():
            if (event.type == pygame.QUIT) or \
                ((event.type == pygame.KEYDOWN) and
                 (event.key == pygame.K_ESCAPE)):
```

```

        # Сообщаем текущей сцене, что мы уходим
        self.oCurrentScene.leave()
        pygame.quit()
        sys.exit()

eventsList.append(event)

# Здесь мы позволяем текущей сцене обработать все события.
# Выполняем действия "в рамках фрейма" в ее методе update
# и рисуем все, что должно быть нарисовано
❸ self.oCurrentScene.handleInputs(eventsList, keysDownList)
❹ self.oCurrentScene.update()
❺ self.oCurrentScene.draw()

#11 - Обновляем окно
❻ pygame.display.update()

#12 - Делаем паузу
clock.tick(self.framesPerSecond)

```

Листинг 15.7. Метод `run()` класса `SceneMgr`

Метод `run()` — ключ к работе менеджера сцен. Помните, что все сцены должны быть полиморфны: как минимум каждая обязана реализовывать методы `handleInputs()` и `draw()`. Всякий раз во время прохождения цикла метод `run()` выполняет следующее.

- Получает список всех клавиш клавиатуры ❶ (`False` обозначает, что клавиша не нажата, `True` — клавиша нажата).
- Создает список событий ❷, которые произошли с момента последнего прохождения цикла.
- Вызывает полиморфные методы ❸ текущей сцены. Текущая сцена всегда хранится в переменной экземпляра с именем `self.oCurrentScene`. В вызове метода сцены `handleInputs()` менеджер сцен передает список произошедших событий и список клавиш. Каждая сцена отвечает за обработку событий и работу с состоянием клавиатуры.
- Вызывает метод `update()` ❹, чтобы разрешить сцене выполнять любые действия за фрейм. Базовый класс `Scene` реализует метод `update()`, который содержит только оператор `pass`, но сцена может переопределить этот метод с помощью любого кода, который она хочет выполнить.

- Вызывает метод `draw()` ❸, чтобы разрешить сцене рисовать в окне все, что ей необходимо.

В нижней части цикла (идентичной стандартному основному циклу без менеджера сцен) метод обновляет окно ❹ и ждет соответствующее количество времени.

Основные методы

Остальные методы класса `SceneMgr` реализуют навигацию и взаимодействие между сценами:

`_goToScene()` — вызывается для перехода к другой сцене;

`_request_respond()` — вызывается для запроса данных в другой сцене;

`_send_receive()` — вызывается для отправки информации от одной сцены другой;

`_sendAll_receive()` — вызывается для отправки информации от одной сцены всем остальным.

Код любой написанной вами сцены не должен вызывать эти методы напрямую, и они не должны быть переопределены. Нижнее подчеркивание перед их именами подразумевает, что это скрытые (внутренние) методы. Они вызываются базовым классом `Scene`, а не напрямую в самом менеджере сцен.

Чтобы объяснить, как эти методы работают, я начну с обзора шагов, выполняемых, когда сцена хочет перейти к другой. Для перехода к целевой сцене текущая сцена вызывает:

```
self.goToScene(SOME_SCENE_KEY)
```

Когда сцена выполняет этот вызов, он переходит к методу `goToScene()` в унаследованном базовом классе `Scene`. Код унаследованного метода состоит из одной строки:

```
def goToScene(self, nextSceneKey, data=None):
```

```
    --- пропуск ---
```

```
    self.oSceneMgr._goToScene(nextSceneKey, data)
```

Это вызывает скрытый метод `_goToScene()` в менеджере сцен. Внутри метода менеджера сцен мы должны предоставить текущей сцене возможность выполнить любую очистку, которая потребуется, а затем передать управление новой сцене. Ниже представлен код метода `_goToScene()` менеджера сцен:

```
def _goToScene(self, nextSceneKey, dataForNextScene):

    --- пропуск ---

    if nextSceneKey is None: # значение, выход
        pygame.quit()
        sys.exit()

    # Вызываем метод leave старой сцены, чтобы разрешить ей очистку.
    # Настраиваем новую сцену (на основе ключа)
    # и вызываем метод enter новой сцены
    ❶ self.oCurrentScene.leave()
    pygame.key.set_repeat(0) # отключаем повторяющиеся символы
    try:
        ❷ self.oCurrentScene = self.scenesDict[nextSceneKey]
    except KeyError:
        raise KeyError("Trying to go to scene '" + nextSceneKey +
                        "' but that key is not in the dictionary of scenes.")
    ❸ self.oCurrentScene.enter(dataForNextScene)
```

Метод `_goToScene()` выполняет ряд шагов для перехода из текущей сцены к целевой. Сначала он вызывает `leave()` в текущей сцене ❶, так что она может выполнить всю необходимую очистку. Затем, используя переданный ключ целевой сцены, он находит объект для целевой сцены ❷ и устанавливает в качестве текущей сцены. И наконец, он вызывает `enter()` для новой текущей сцены ❸, чтобы разрешить ей выполнить всю необходимую настройку.

С этого момента метод `run()` менеджера сцен закидывает и вызывает методы `handleInputs()`, `update()` и `draw()` текущей сцены. Они будут вызываться в текущей сцене до тех пор, пока программа не выполнит другой вызов `self._goToScene()` для перехода к еще одной сцене или пока пользователь не выйдет из программы.

Взаимодействие между сценами

Наконец, мы рассмотрим, как одна сцена взаимодействует с другой. Чтобы запросить информацию от другой сцены, ей нужно лишь выполнить вызов `self.request()`, который находится в базовом классе `Scene`, следующим образом:

```
dataRequested = self.request(SOME_SCENE_KEY, SOME_DATA_IDENTIFIER)
```

В целевой сцене должен быть метод `respond()`. Он определяется следующим образом:

```
def respond(self, requestID):
```

Метод использует значение `requestID`, чтобы однозначно определить, какие данные извлекать, и возвращает их. Запрашивающая и целевая сцены должны быть согласованы по поводу значения любого идентификатора. Весь процесс показан на рис. 15.10.

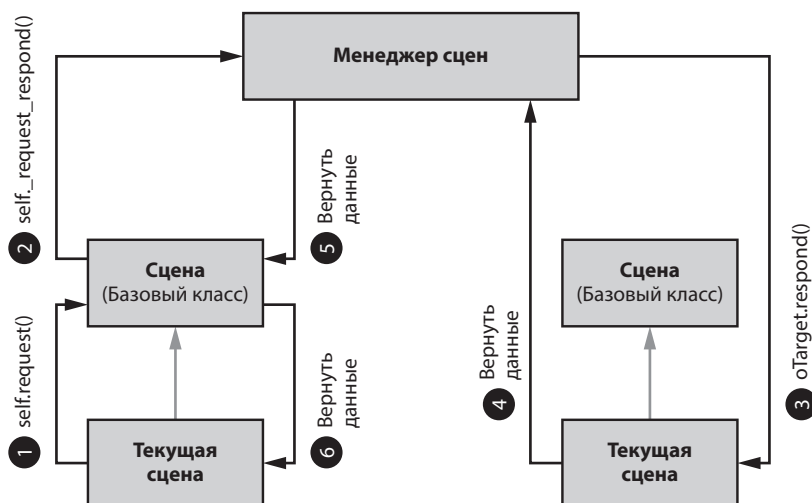


Рис. 15.10. Путь взаимодействия одной сцены, запрашивающей информацию у другой сцены

Текущая сцена не может получить информацию от другой сцены напрямую, поскольку в текущей сцене нет ссылок на какие-либо другие сцены. Вместо этого она использует менеджер сцен в качестве посредника. Вот как это работает.

1. Текущая сцена вызывает `self.request()`, который находится в базовом классе `Scene`.
2. В базовом классе `Scene` есть ссылка на менеджер сцен в его переменной экземпляра `self.oSceneMgr`, чтобы разрешить его методам вызывать методы менеджера сцен. Метод `self.request()` вызывает метод менеджера сцен `_request_respond()`, чтобы запросить информацию у целевой сцены.
3. В менеджере сцен есть словарь всех ключей сцен и связанных объектов, и он запускает передаваемые параметры, чтобы найти объект, связанный с целевой сценой. Затем он вызывает метод `respond()` в целевой сцене.
4. Метод `respond()` в целевой сцене (которую вы должны написать) выполняет все необходимое для генерирования запрашиваемых данных, затем возвращает их менеджеру сцен.
5. Менеджер сцен возвращает данные методу `request()` в базовом классе `Scene`, унаследованном текущей сценой.
6. И наконец, метод `request()` в базовом классе `Scene` возвращает данные исходному вызывающему.

Аналогичный механизм используется для реализации `send()` и `sendAll()`. Единственное отличие состоит в том, что при отправке сообщения одной или всем сценам исходному вызывающему не возвращаются никакие данные.

Выводы

В этой главе я познакомил вас с двумя разными способами реализации программы, которая включает несколько сцен. Конечный автомат — это метод представления и управления потоком выполнения через ряд состояний; вы можете использовать его для реализации программы с небольшим количеством сцен. Менеджер сцен спроектирован, чтобы помочь создать более крупные приложения со множеством сцен, обеспечивая навигацию и общий способ взаимодействия сцен друг с другом. Я также объяснил, каким образом менеджер сцен реализует все эти функциональные возможности.

Менеджер сцен и базовый класс `Scene` предоставляют понятные примеры трех основных принципов объектно-ориентированного программирования: инкапсуляции,

полиморфизма и наследования. Каждая сцена является прекрасным примером инкапсуляции, потому что весь код и данные сцены написаны в виде класса. Каждая сцена должна быть полиморфной в том смысле, что она обязана реализовывать общий набор методов, чтобы он мог работать с вызовами из менеджера сцен. И наконец, каждая сцена наследует от общего базового класса `Scene`. Двустороннее взаимодействие между менеджером сцен и базовым классом `Scene` реализуется каждой сценой с использованием унаследованных методов и переменных экземпляра в базовом классе.

16

ПОЛНОЦЕННАЯ ИГРА: DODGER



В этой главе мы создадим полноценную игру под названием Dodger, где используются многие методы и концепции, которые уже объяснялись в этой книге. Это полностью объектно-

ориентированная расширенная версия игры, которая изначально была разработана Элом Свейгартом в его книге «Учим Python, делая крутые игры»* (базовая концепция игры, графика и звуки используются с разрешения).

Прежде чем перейти к самой игре, я познакомлю вас с набором функций, представляющих модальные диалоговые окна, который мы будем использовать в игре. *Модальное диалоговое окно* — это диалоговое окно, которое заставляет пользователя взаимодействовать с ним, например, выбирая опцию, прежде чем он сможет продолжить работу с базовой программой. Эти диалоговые окна останавливают выполнение программы до тех пор, пока не будет выбрана опция.

* Русское издание: Свейгарт Э. Учим Python, делая крутые игры. М.: Бомбора, 2023.

Модальные диалоговые окна

В модуле `pyghelpers` есть два типа модальных диалоговых окон.

- *Диалоговые окна **Yes/No*** задают вопрос и ждут, пока пользователь щелкнет по одной из двух кнопок. По умолчанию надписи на этих кнопках — **Yes** и **No**, хотя вы можете использовать любой понравившийся вам текст (например, **OK** и **Cancel**). Если для кнопки **No** не указан текст, это диалоговое окно может использоваться как предупреждение с одной кнопкой **Yes** (или обычно **OK**).
- *Диалоговые окна с ответом* представляют вопрос, текстовое поле для ввода пользователем и набор кнопок с текстом по умолчанию **OK** и **Cancel**. Пользователь может ответить на вопрос и нажать **OK** или отменить (закрыть) диалоговое окно, нажав кнопку **Cancel**.

Вы представляете каждый тип диалогового окна пользователю, вызывая конкретную функцию в модуле `pyghelpers`. Любое диалоговое окно представлено в двух видах: простой, основанной на `TextButton` версии и более сложной пользовательской версии. Первая использует компоновку по умолчанию с двумя объектами `TextButton`, которые хороши для быстрого прототипирования. В пользовательской версии вы можете предоставить фон для диалогового окна, настроить текст вопроса и текст ответа (с диалоговым окном с ответом) и предоставить пользовательскую графику для кнопок.

Диалоговые окна **Yes/No** и Предупреждения

Сначала мы разберем диалоговое окно **Yes/No**, начиная с текстовой версии.

Текстовая версия

Ниже представлен интерфейс функции `textYesNoDialog()`:

```
textYesNoDialog(theWindow, theRect, prompt,  
                yesButtonText='Yes', noButtonText='No',  
                backgroundColor=DIALOG_BACKGROUND_COLOR,  
                textColor=DIALOG_BLACK)
```

Когда вы вызываете эту функцию, вам необходимо передать окно для рисования, прямоугольный объект или кортеж, представляющий местоположение и размер создаваемого диалогового окна, и текстовую подсказку для отображения. Дополнительно вы также можете указать текст двух кнопок, цвет фона и цвет текстовой подсказки. Текст кнопки по умолчанию будет **Yes** и **No**.

Вот как выглядит типичный вызов функции:

```
returnValue = pyghelpers.textYesNoDialog(window,  
    (75, 100, 500, 150),  
    'Do you want fries with that?')
```

Этот вызов демонстрирует диалоговое окно на рис. 16.1.

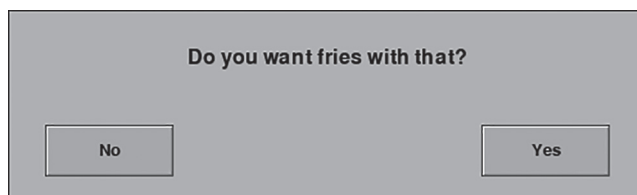


Рис. 16.1. Типичное диалоговое окно `textYesNoDialog`

Кнопки **Yes** и **No** — это экземпляры класса `TextButton` в `pygwidgets`. Основная программа останавливается во время демонстрации диалогового окна. Когда пользователь щелкает по кнопке, функция возвращает `True` для **Yes** и `False` для **No**. Ваш код выполняет все необходимое на основе возвращенного булевого значения; затем основная программа продолжает работу с того места, где она остановилась.

Вы можете также использовать эту функцию для создания простого диалогового окна предупреждения только с одной кнопкой. Если переданное значение для `noButtonText` равно `None`, тогда эта кнопка не будет отображаться. Например, вы можете выполнить следующий вызов, чтобы показать только одну кнопку:

```
ignore = pyghelpers.textYesNoDialog(window,  
    (75, 80, 500, 150),  
    'This is an alert!', 'OK', None)
```

На рис. 16.2 демонстрируется итоговое окно предупреждения.

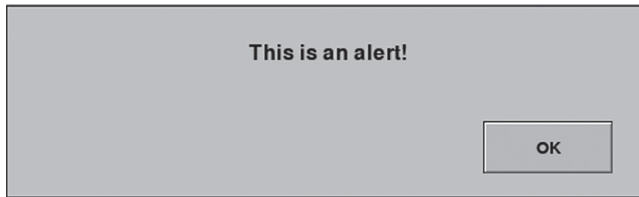


Рис. 16.2. `textYesNoDialog`, используемый в качестве диалогового окна

Пользовательская версия

Настройка пользовательского диалогового окна **Yes/No** более сложна, но обеспечивает больше контроля. Ниже представлен интерфейс функции `customYesNoDialog()`:

```
customYesNoDialog(theWindow, oDialogImage, oPromptText,  
                  oYesButton, oNoButton)
```

Перед вызовом этой функции вам необходимо создать объекты для фона диалогового окна, текстовую подсказку и кнопки **Yes** и **No**. Как правило, вы будете использовать объекты `Image`, `DisplayText` и `CustomButton` (или `TextButton`), созданные для этой цели из классов `pygwidgets`. Код `customYesNoDialog()` демонстрирует полиморфизм, вызывая методы кнопок `handleEvent()`, поэтому не имеет значения, используете вы `CustomButton` или `TextButton` или вызываете метод `draw()` всех объектов, составляющих диалоговое окно. Поскольку вы создаете все эти объекты, можете настроить вид любого из них или всех. Вам нужно будет предоставить собственную графику для любого объекта `Image` или `CustomButton` и поместить эти файлы в папку проекта *images*.

Реализуя пользовательское диалоговое окно **Yes/No**, вы, как правило, пишете промежуточную функцию, такую как `showCustomYesNoDialog()`, продемонстрированную в листинге 16.1. Затем в том месте кода, где хотите отобразить диалоговое окно, вы вместо вызова `customYesNoDialog()` напрямую вызываете промежуточную функцию, которая одновременно создает экземпляры виджетов и выполняет фактический вызов.

```
def showCustomYesNoDialog(theWindow, theText):  
❶    oDialogBackground = pygwidgets.Image(theWindow, (60, 120),  
                                           'images/dialog.png')
```

```

❷ oPromptDisplayText = pygwidgets.DisplayText(theWindow, (0, 170),
                                              theText, width=WINDOW_WIDTH,
                                              justified='center', fontSize=36)
❸ oNoButton = pygwidgets.CustomButton(theWindow, (95, 265),
                                       'images/noNormal.png',
                                       over='images/noOver.png',
                                       down='images/noDown.png',
                                       disabled='images/noDisabled.png')
oYesButton = pygwidgets.CustomButton(theWindow, (355, 265),
                                       'images/yesNormal.png',
                                       over='images/yesOver.png',
                                       down='images/yesDown.png',
                                       disabled='images/yesDisabled.png')
❹ userAnswer = pyghelpers.customYesNoDialog(theWindow,
                                             oDialogBackground,
                                             oPromptDisplayText,
                                             oYesButton, oNoButton)
❺ return userAnswer

```

Листинг 16.1. Промежуточная функция для создания пользовательского диалогового окна **Yes/No**

Внутри функции вы пишете код, чтобы создать объект `Image` для фона, используя указанное вами изображение ❶. Также вы создаете объект `DisplayText` для подсказки ❷, в которой вы указываете размещение, размер текста и так далее. Затем создаете кнопки либо в виде объектов `TextButton`, либо, что более вероятно, в виде объектов `CustomButton`, чтобы вы могли показать пользовательские изображения ❸. И наконец, эта функция вызывает `customYesNoDialog()`, передавая все только что созданные вами объекты ❹. Вызов `customYesNoDialog()` возвращает выбор пользователя этой промежуточной функции, а она в свою очередь возвращает выбор пользователя первоначальному вызывающему ❺. Этот подход хорошо работает, потому что все объекты виджетов (`oDialogBackground`, `oPromptDisplayText`, `oYesButton` и `oNoButton`), созданные внутри этой функции, являются локальными переменными и, следовательно, все они исчезнут, когда промежуточная функция завершится.

Когда вы вызываете данную функцию, вам необходимо только передать окно и текстовую подсказку для отображения. Например:

```

returnValue = showCustomYesNoDialog(window,
                                     'Do you want fries with that?')

```

На рис. 16.3 продемонстрировано итоговое диалоговое окно. Это всего лишь один пример; вы можете спроектировать любую компоновку, какая вам нравится.



Рис. 16.3. Типичное диалоговое окно `customYesNoDialog`

Как и в простой текстовой версии, если передаваемое значение `oNoButton` равно `None`, эта кнопка не будет отображаться, что полезно для создания и отображения диалогового окна предупреждения.

Внутренне каждая функция `textYesNoDialog()` и `customYesNoDialog()` выполняет собственный цикл `while`, который обрабатывает события и обновляет и рисует диалоговые окна. Таким образом, вызывающая программа приостанавливается (ее основной цикл не выполняется) до тех пор, пока пользователь не щелкнет по кнопке и модальное диалоговое окно не вернет выбранный ответ. (Исходный код обеих функций доступен в модуле `pyghelpers`.)

Диалоговые окна с ответом

Диалоговое окно с ответом добавляет поле ввода текста, в котором пользователь может напечатать ответ. Модуль `pyghelpers` также содержит функции `textAnswerDialog()` и `customAnswerDialog()` для обработки этих диалогов, которые работают аналогично диалоговому окну **Yes/No**.

Текстовая версия

Вот как выглядит интерфейс функции `textAnswerDialog()`:

```
textAnswerDialog(theWindow, theRect, prompt, okButtonText='OK'
                 cancelButtonText='Cancel',
                 backgroundColor=DIALOG_BACKGROUND_COLOR,
                 promptTextColor=DIALOG_BLACK,
                 inputTextColor=DIALOG_BLACK)
```

Если пользователь щелкает по кнопке **ОК**, функция возвращает весь введенный им текст. Если пользователь щелкает по кнопке **Cancel**, функция возвращает `None`. Ниже представлен типичный вызов:

```
userAnswer = pyghelpers.textAnswerDialog(window, (75, 100, 500, 200),
                                           'What is your favorite flavor of ice cream?')
if userAnswer is not None:
    # Пользователь нажал ОК, выполняем операции с переменной
    userAnswer
else:
    # Выполняем соответствующие операции, так как пользователь
    # нажал Cancel
```

Отобразится диалоговое окно, представленное на рис. 16.4.

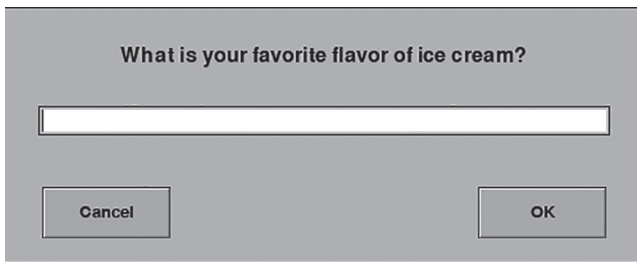


Рис. 16.4. Типичное диалоговое окно `textAnswerDialog`

Пользовательская версия

Чтобы реализовать пользовательское диалоговое окно с ответом, вы должны написать промежуточную функцию аналогично подходу, продемонстрированному с `customYesNoDialog()`. Ваш основной код вызывает промежуточную функцию, которая в свою очередь вызывает `customAnswerDialog()`. В листинге 16.2 продемонстрирован код типичной промежуточной функции.

```
def showCustomAnswerDialog(theWindow, theText):
    oDialogBackground = pygwidgets.Image(theWindow, (60, 80),
                                           'images/dialog.png')
    oPromptDisplayText = pygwidgets.DisplayText(theWindow, (0, 120),
                                                  theText, width=WINDOW_WIDTH,
                                                  justified='center', fontSize=36)
    oUserInputText = pygwidgets.InputText(theWindow, (225, 165), '',
                                           fontSize=36, initialFocus=True)
```

```

oNoButton = pygwidgets.CustomButton(theWindow, (105, 235),
                                     'images/cancelNormal.png',
                                     over='images/cancelOver.png',
                                     down='images/cancelDown.png',
                                     isabled='images/cancelDisabled.png')
oYesButton = pygwidgets.CustomButton(theWindow, (375, 235),
                                     'images/okNormal.png',
                                     over='images/okOver.png',
                                     down='images/okDown.png',
                                     disabled='images/okDisabled.png')
response = pyghelpers.customAnswerDialog(theWindow,
                                          oDialogBackground, oPromptDisplayText,
                                          oUserInputText,
                                          oYesButton, oNoButton)

return response

```

Листинг 16.2. Промежуточная функция для создания пользовательского диалогового окна с ответом

Вы можете настроить весь внешний вид диалогового окна: фон изображения, шрифты и их размеры, размещение полей отображения и ввода текста и двух кнопок. Чтобы показать пользовательское диалоговое окно, ваш основной код вызовет промежуточную функцию и передаст текстовую подсказку следующим образом:

```

userAnswer = showCustomAnswerDialog(window,
                                     'What is your favorite flavor of ice cream?')

```

Этот вызов отображает пользовательское диалоговое окно с ответом, подобно продемонстрированному на рис. 16.5.

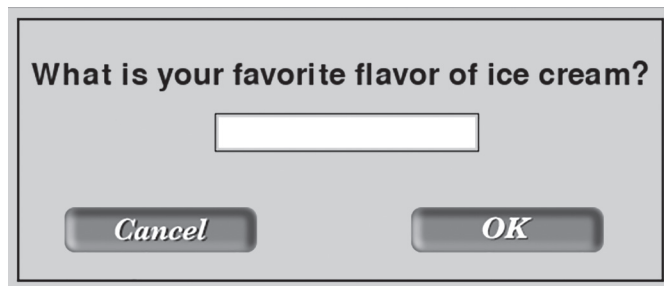


Рис. 16.5. Типичное диалоговое окно `customAnswerDialog`

Если пользователь нажимает **ОК**, функция возвращает введенный пользователем текст. Если пользователь щелкает по кнопке **Cancel**, функция возвращает `None`.

ДемOVERсия программы, которая демонстрирует все типы диалоговых окон, *DialogTester/Main_DialogTester.py*, доступна вместе с загружаемыми ресурсами этой книги.

Создаем полноценную игру: Dodger

В этом разделе мы сведем воедино весь материал этой книги в контексте игры под названием Dodger. С точки зрения пользователя, игра невероятно проста: надо заработать как можно больше очков, уворачиваясь от красных Злодеев и вступая в контакт с Героями.

Обзор игры

Красные Злодеи будут выпадать из верхней части окна, и пользователь должен избегать их. Любой Злодей, который добрался до нижней части игровой зоны, удаляется, и пользователь получает одно очко. Пользователь передвигает мышью, чтобы управлять пиктограммой Игрока. Если Игрок задевает какого-либо Злодея, он проиграл. Небольшое количество зеленых Героев отображается случайным образом и движется горизонтально, и пользователь получает 25 очков за касание любого Героя.

В игре есть три сцены: начальная, или сцена Приветствия, с инструкциями, сцена Игры, в которой вы играете, и сцена Лучших результатов, где вы можете увидеть 10 лучших результатов. Если ваши очки попадают в этот топ-10, вам предлагают ввести свое имя и информацию об очках в таблицу лучших результатов. На рис. 16.6 продемонстрированы эти три сцены.

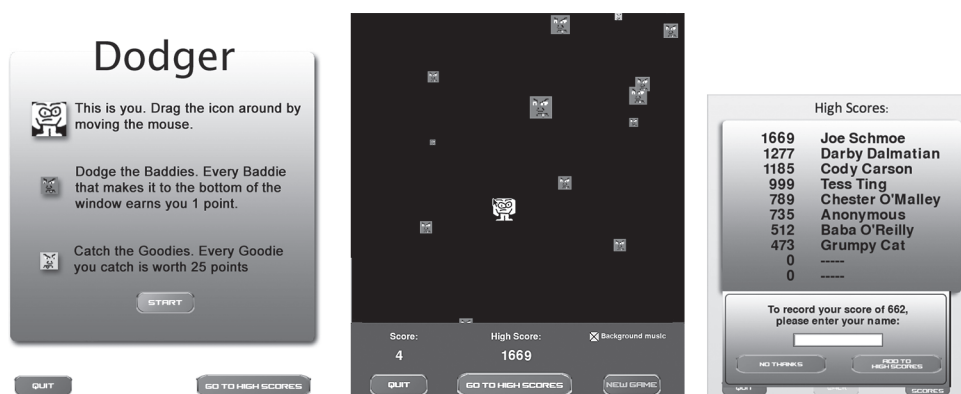


Рис. 16.6. Сцены Приветствия, Игры и Лучших результатов (слева направо)

Реализация

Содержимое папки проекта *Dodger* выглядит следующим образом (имена файлов выделены курсивом).

__init__.py Пустой файл, который указывает, что это пакет Python.

Baddies.py Содержит классы *Baddie* и *BaddieMgr*.

Constants.py Содержит константы, используемые несколькими сценами.

Goodies.py Содержит классы *Goodie* и *GoodieMgr*.

HighScoresData.py Содержит класс *HighScoresData*.

images Папка, которая содержит всю графику для игры.

Main_Dodger.py Основная программа.

Player.py Содержит класс *Player*.

SceneHighScores.py Сцена, которая отображает и записывает наивысшие очки.

ScenePlay.py Основная сцена Игры.

Scene.Splash.py Сцена Приветствия.

sounds Папка, которая содержит все звуковые файлы для игры.

Папка проекта включена в ресурсы этой книги. Я не буду подробно разбирать весь код, но пройдуся по исходным файлам и объясню, как работают их ключевые части.

Файл: *Dodger/Constants.py*

Этот файл содержит константы, которые могут быть использованы более чем одним исходным файлом. Наиболее важными константами являются следующие ключи сцен:

```
# Ключи сцен
SCENE_SPLASH = 'scene splash'
SCENE_PLAY = 'scene play'
SCENE_HIGH_SCORES = 'scene high scores'
```

Их значения — однозначные строки, которые идентифицируют различные сцены.

Файл: *Main_Dodger.py*

Основной файл выполняет необходимую инициализацию, затем передает управление менеджеру сцен. Наиболее важный код в файле следующий:

```
# создаем экземпляры всех сцен и сохраняем их в списке
scenesList = [SceneSplash(window)
               SceneHighScores(window)
               ScenePlay(window)]

# создаем менеджер сцен, передавая список сцен и FPS
oSceneMgr = pyghelpers.SceneMgr(scenesList, FRAMES_PER_SECOND)

# менеджер сцен начинает выполнение
oSceneMgr.run()
```

Здесь мы создаем экземпляр каждой сцены и экземпляр менеджера сцен, которому затем передаем управление. Метод менеджера сцен `run()` предоставляет управление первой сцене в списке. В этой игре он передает управление сцене Приветствия.

Как обсуждалось в предыдущей главе, каждая сцена класса наследует от базового класса `Scene`. В дополнение к предоставлению собственного метода `__init__()` каждый из этих классов должен переопределить методы `getSceneKey()`, `handleInputs()` и `draw()` из базового класса.

Файл: *Dodger/SceneSplash.py*

Сцена Приветствия демонстрирует пользователю графическое изображение с правилами игры и тремя кнопками: **Start**, **Exit** и **Go to High Scores**. Код для класса этой сцены содержит только необходимые методы. Все другие по умолчанию находятся в базовом классе `Scene`.

Метод `__init__()` создает объект `Image` для фонового изображения и три объекта `CustomButton` для пользовательских опций.

Метод `getSceneKey()` должен реализовываться во всех сценах; он просто возвращает уникальный ключ для сцены.

Метод `handleInputs()` проверяет, щелкнул ли пользователь по какой-либо из кнопок. Если пользователь щелкнул по кнопке **Start**, мы вызываем унаследованный метод `self.goToScene()`, чтобы попросить менеджера сцен передать управление сцене

Игры. Аналогичным образом, щелчок по кнопке **Go to High Scores** переместит пользователя в сцену Лучших результатов. Если пользователь щелкает по кнопке **Exit**, мы вызываем унаследованный метод сцены `self.quit()`, который завершает работу программы.

В методе `draw()` программа рисует фон и все три кнопки.

Файл: **Dodger/ScenePlay.py**

Сцена Игры управляет фактическим процессом игры: перемещением пользователем пиктограммы Игрока, генерированием и перемещением Злодеев и Героев и обнаружением коллизий. Она также управляет отображением элементов в нижней части окна, включая текущие очки игры и лучшие очки, и реагирует на нажатие кнопок **Exit**, **Go to High Scores** и **Start** и на установку флажка **Background Music**.

В сцене Игры довольно много кода, но я разобью его на более мелкие части (листинги 16.3–16.7), чтобы объяснить методы. Сцена придерживается правил проектирования, установленных в главе 15, путем реализации методов `__init__()`, `handleInputs()`, `update()` и `draw()`. Она также реализует метод `enter()` для обработки того, что сцена должна делать, когда она становится активной, и метод `leave()` для того, что сцена должна делать, когда пользователь покидает игру. И наконец, в ней есть метод `reset()` для сброса состояния перед началом нового раунда. В листинге 16.3 продемонстрирован код инициализации.

```
# Сцена Игры – основная сцена игры
--- пропущены операторы импорта и showCustomYesNoDialog ---

BOTTOM_RECT = (0, GAME_HEIGHT + 1, WINDOW_WIDTH,
               WINDOW_HEIGHT - GAME_HEIGHT)
STATE_WAITING = 'waiting'
STATE_PLAYING = 'playing'
STATE_GAME_OVER = 'game over'

class ScenePlay(pyghelpers.Scene):

    def __init__(self, window):
        self.window = window

        self.controlsBackground = pygwidgets.Image(self.window,
                                                    (0, GAME_HEIGHT),
                                                    'images/controlsBackground.jpg')
```

```

self.quitButton = pygwidgets.CustomButton(self.window,
                                           (30, GAME_HEIGHT + 90),
                                           up='images/quitNormal.png',
                                           down='images/quitDown.png',
                                           over='images/quitOver.png',
                                           disabled='images/quitDisabled.png')

self.highScoresButton = pygwidgets.CustomButton(self.window,
                                                  (190, GAME_HEIGHT + 90),
                                                  up='images/gotoHighScoresNormal.png',
                                                  down='images/gotoHighScoresDown.png',
                                                  over='images/gotoHighScoresOver.png',
                                                  disabled='images/
                                                  gotoHighScoresDisabled.png')

self.startButton = pygwidgets.CustomButton(self.window,
                                             (450, GAME_HEIGHT + 90),
                                             up='images/startNewNormal.png',
                                             down='images/startNewDown.png',
                                             over='images/startNewOver.png',
                                             disabled='images/startNewDisabled.png',
                                             enterToActivate=True)

self.soundCheckBox = pygwidgets.TextCheckBox(self.window,
                                              (430, GAME_HEIGHT + 17),
                                              'Background music',
                                              True, textColor=WHITE)

self.gameOverImage = pygwidgets.Image(self.window, (140, 180),
                                          'images/gameOver.png')

self.titleText = pygwidgets.DisplayText(self.window,
                                          (70, GAME_HEIGHT + 17),
                                          'Score:           High Score:',
                                          fontSize=24, textColor=WHITE)

self.scoreText = pygwidgets.DisplayText(self.window,
                                          (80, GAME_HEIGHT + 47), '0',
                                          fontSize=36, textColor=WHITE,
                                          justified='right')

self.highScoreText = pygwidgets.DisplayText(self.window,
                                              (270, GAME_HEIGHT + 47), '',
                                              fontSize=36, textColor=WHITE,
                                              justified='right')

pygame.mixer.music.load('sounds/background.mid')
self.dingSound = pygame.mixer.Sound('sounds/ding.wav')

```

```

self.gameOverSound = pygame.mixer.Sound('sounds/gameover.wav')

# создаем экземпляры объектов
2 self.oPlayer = Player(self.window)
  self.oBaddieMgr = BaddieMgr(self.window)
  self.oGoodieMgr = GoodieMgr(self.window)

self.highestHighScore = 0
self.lowestHighScore = 0
self.backgroundMusic = True
self.score = 0
3 self.playingState = STATE_WAITING

4 def getSceneKey(self):
  return SCENE_PLAY

```

Листинг 16.3. Методы `__init__()` и `getSceneKey()` класса `ScenePlay`

При выполнении основной код игры создает экземпляры всех сцен. В сцене Игры метод `__init__()` создает все кнопки и поля отображения текста для нижней части окна ❶, затем загружает звуки. Очень важно, что мы используем композицию, которая рассматривалась в главах 4 и 10, чтобы создать объект Игрока (`oPlayer`), менеджер объекта Злодея (`oBaddieMgr`) и менеджер объекта Героя (`oGoodieMgr`) ❷. Объект сцены Игры создает этих менеджеров и ожидает, что они будут создавать всех Злодеев и Героев и управлять ими. Метод `__init__()` выполняется, когда программа начинает работу, но, по сути, не запускает игру. Вместо этого он реализует конечный автомат (как обсуждалось в главе 15), который запускает состояние ожидания ❸. Раунд игры начинается, когда пользователь нажимает на кнопку **New Game**.

Во всех сценах должен быть метод `getSceneKey()` ❹, который возвращает строку, представляющую текущую сцену. Листинг 16.4 демонстрирует код, который извлекает очки и сбрасывает игру по запросу.

```

1 def enter(self, data):
  self.getHiAndLowScores()

2 def getHiAndLowScores(self):
  # запрашиваем у сцены Лучших результатов словарь очков,
  # который выглядит следующим образом:
  # {'вышие': высшиеОчки, 'низшие': низшиеОчки}
3 infoDict = self.request(SCENE_HIGH_SCORES, HIGH_SCORES_DATA)

```

```

self.highestHighScore = infoDict['highest']
self.highScoreText.setValue(self.highestHighScore)
self.lowestHighScore = infoDict['lowest']

❹ def reset(self): # начинаем новую игру
    self.score = 0
    self.scoreText.setValue(self.score)
    self.getHiAndLowScores()

    # сбрасываем менеджеры
❺ self.oBaddieMgr.reset()
    self.oGoodieMgr.reset()

    if self.backgroundMusic:
        pygame.mixer.music.play(-1, 0.0)
❻ self.startButton.disable()
    self.highScoresButton.disable()
    self.soundCheckBox.disable()
    self.quitButton.disable()
    pygame.mouse.set_visible(False)

```

Листинг 16.4. Методы `enter()`, `getHiAndLowScores()` и `reset()` класса `ScenePlay`

При переходе к сцене Игры менеджер сцен вызывает `enter()` ❶, который в свою очередь вызывает метод `getHiAndLowScores()` ❷. Последний отправляет запрос сцене Лучших результатов ❸, чтобы узнать самое большое и самое маленькое количество очков в таблице лучших результатов, тогда мы сможем нарисовать лучший результат в строке в нижней части окна. В конце каждой игры он сравнивает итоговые очки с низшими из топ-10, чтобы увидеть, входит ли эта игра в топ-10.

Когда пользователь щелкает по кнопке **New Game**, вызывается метод `reset()` ❹ для повторной инициализации всего, чтобы должно быть сброшено перед началом нового раунда игры. Метод `reset()` говорит менеджеру Злодеев и менеджеру Героев повторно инициализировать себя, вызвав собственные методы `reset()` ❺, отключает кнопки в нижней части экрана, чтобы их нельзя было нажать во время игры ❻, и скрывает указатель мыши. Во время игры пользователь передвигает мышь, чтобы управлять пиктограммой Игрока в окне.

Код в листинге 16.5 работает с входными данными пользователя.

```

❶ def handleInputs(self, eventsList, keyPressedList):
❷     if self.playingState == STATE_PLAYING:
        return # игнорируем события кнопок во время игры

        for event in eventsList:
❸             if self.startButton.handleEvent(event):
                    self.reset()
                    self.playingState = STATE_PLAYING

❹             if self.highScoresButton.handleEvent(event):
                    self.goToScene(SCENE_HIGH_SCORES)

❺             if self.soundCheckBox.handleEvent(event):
                    self.backgroundMusic = self.soundCheckBox.getValue()

❻             if self.quitButton.handleEvent(event):
                    self.quit()

```

Листинг 16.5. Метод `handleInputs()` класса `ScenePlay`

Метод `handleInputs()` ❶ отвечает за события щелчков. Если конечный автомат находится в состоянии воспроизведения, пользователь не может нажимать кнопки, поэтому мы не утруждаем себя проверкой событий ❷. Если пользователь нажимает кнопку **New Game** ❸, мы вызываем `reset()` для повторной инициализации переменных и перевода конечного автомата в состояние воспроизведения. Если пользователь нажимает **Go to High Scores** ❹, мы переходим к сцене Лучших результатов, используя унаследованный метод `self.goToScene()`. Если пользователь устанавливает флажок **Background Music** ❺, мы вызываем его метод `getValue()`, чтобы извлечь его новые настройки; метод `reset()` задействует свои настройки, чтобы решить, стоит ли воспроизводить фоновую музыку. Если пользователь нажимает **Exit** ❻, мы вызываем унаследованный метод `self.quit()` из базового класса. В листинге 16.6 продемонстрирован код для реальной игры.

```

❶ def update(self):
    if self.playingState != STATE_PLAYING:
        return # обновляем только во время игры

    # перемещаем Игрока на позицию мыши, возвращаем его rect
❷ mouseX, mouseY = pygame.mouse.get_pos()
    playerRect = self.oPlayer.update(mouseX, mouseY)

```

```

# инструктируем GoodieMgr переместить всех Героев, возвращаем
# число Героев, с которыми у пользователя был контакт
3 nGoodiesHit = self.oGoodieMgr.update(playerRect)
  if nGoodiesHit > 0:
      self.dingSound.play()
      self.score = self.score + (nGoodiesHit * POINTS_FOR_GOODIE)

# инструктируем BaddieMgr переместить всех Злодеев,
# возвращаем число Злодеев, которые выпали из окна
4 nBaddiesEvaded = self.oBaddieMgr.update()
  self.score = self.score + (nBaddiesEvaded *
                              POINTS_FOR_BADDIE_EVADED)
  self.scoreText.setValue(self.score)

# проверяем, задел ли Игрок какого-либо Злодея
5   if self.oBaddieMgr.hasPlayerHitBaddie(playerRect):
       pygame.mouse.set_visible(True)
       pygame.mixer.music.stop()

  self.gameOverSound.play()
  self.playingState = STATE_GAME_OVER
6  self.draw() # принудительный вывод сообщения
               # об окончании игры

7
      if self.score > self.lowestHighScore:
          scoreAsString = 'Your score: ' + str(self.score) + '\n'
          if self.score > self.highestHighScore:
              dialogText = (scoreString +
                              'is a new high score,
                              CONGRATULATIONS!')
          else:
              dialogText = (scoreString +
                              'gets you on the high scores list.')

          result = showCustomYesNoDialog(self.window,
                                         dialogText)

          if result: # переходим
              self.goToScene(SCENE_HIGH_SCORES, self.score)

  self.startButton.enable()
  self.highScoresButton.enable()
  self.soundCheckBox.enable()
  self.quitButton.enable()

```

Листинг 16.6. Метод `update()` класса `ScenePlay`

Менеджер сцен вызывает метод `update()` класса `ScenePlay` ❶ в каждом фрейме. Этот метод обрабатывает всё

происходящее в процессе игры. Сначала он говорит объекту Игрок переместить пиктограмму Игрока в позицию мыши. Затем вызывает метод Игрока `update()` ❷, который возвращает текущий прямоугольник пиктограммы в окно. Мы используем это, чтобы увидеть, был ли у пиктограммы Игрока контакт с какими-либо Героями или Злодеями.

Далее, он вызывает метод менеджера Героев `update()` ❸, чтобы переместить всех Героев. Этот метод возвращает число Героев, с которыми у Игрока был контакт, что мы используем для увеличения очков.

Затем следует вызов метода менеджера Злодеев `update()` ❹, чтобы переместить всех Злодеев. Этот метод возвращает число Злодеев, которые выпали за пределы нижней части игровой области.

Затем мы проверяем, был ли у Игрока контакт с каким-либо из Злодеев ❺. Если это так, игра проиграна и мы показываем графическое изображение Конец игры. Также мы выполняем специальный вызов метода `draw()` ❻, потому что можем вывести для пользователя диалоговое окно, и основной цикл игры не будет рисовать графическое изображение Конец игры, пока пользователь не щелкнет по одной из кнопок диалогового окна.

И наконец, когда игра завершается, если текущие результаты игры выше, чем десятый лучший результат ❼, мы выводим диалоговое окно, предлагая пользователю внести свои очки в список лучших результатов. Если результат текущей игры — это новый лучший результат всех игр, мы выводим специальное сообщение в диалоговом окне.

Код в листинге 16.7 рисует персонажей игры.

```
❶ def draw(self):
    self.window.fill(BLACK)

    # инструктируем менеджеры нарисовать всех Злодеев и Героев
    self.oBaddieMgr.draw()
    self.oGoodieMgr.draw()

    # инструктируем игрока нарисовать себя
    self.oPlayer.draw()

    # выводим всю информацию внизу окна
❷ self.controlsBackground.draw()
    self.titleText.draw()
    self.scoreText.draw()
```

```

self.highScoreText.draw()
self.soundCheckBox.draw()
self.quitButton.draw()
self.highScoresButton.draw()
self.startButton.draw()

❸ if self.playingState == STATE_GAME_OVER:
    self.gameOverImage.draw()

❹ def leave(self):
    pygame.mixer.music.stop()

```

Листинг 16.7. Методы draw() и leave() класса ScenePlay

Метод draw() говорит Игроку нарисовать себя, менеджеру Героев и Злодеев — нарисовать всех Героев и Злодеев ❶. Затем мы рисуем нижнюю часть окна ❷ со всеми кнопками и полями отображения текста. Если находимся в состоянии конца игры ❸, мы рисуем изображение Конец игры.

Когда пользователь уходит из этой сцены, менеджер сцен вызывает метод leave() ❹, и мы прекращаем воспроизведение какой-либо музыки.

Файл: Dodger/Baddies.py

Файл *Baddies.py* содержит два класса Baddie и BaddieMgr. Сцена игры создает один объект менеджера Злодеев, который формирует и поддерживает список Злодеев. Менеджер Злодеев создает экземпляры объектов из класса Baddie каждые несколько фреймов на основании таймера. В листинге 16.8 содержится код для класса Baddie.

```

# Класс Baddie
--- пропущенные операторы импорта ---

class Baddie():
    MIN_SIZE = 10
    MAX_SIZE = 40
    MIN_SPEED = 1
    MAX_SPEED = 8
    # загружаем изображение только один раз
❶ BADDIE_IMAGE = pygame.image.load('images/baddie.png')

    def __init__(self, window):
        self.window = window

```



```

# создаем объект изображения
size = random.randrange(Baddie.MIN_SIZE, Baddie.MAX_SIZE + 1)
self.x = random.randrange(0, WINDOW_WIDTH - size)
self.y = 0 - size # начинаем над окном
❷ self.image = pygame.image.load(self.window, (self.x, self.y),
                                Baddie.BADDIE_IMAGE)

# масштабируем
percent = (size * 100) / Baddie.MAX_SIZE
self.image.scale(percent, False)
self.speed = random.randrange(Baddie.MIN_SPEED,
                               Baddie.MAX_SPEED + 1)

❸ def update(self): # перемещаем Злодея вниз
    self.y = self.y + self.speed
    self.image.setLoc((self.x, self.y))
    if self.y > GAME_HEIGHT:
        return True # необходимо удалить
    else:
        return False # остается в окне

❹ def draw(self):
    self.image.draw()

❺ def collide(self, playerRect):
    collidedWithPlayer = self.image.overlaps(playerRect)
    return collidedWithPlayer

```

Листинг 16.8. Класс Baddie

Мы загружаем картинку Злодея в качестве переменной класса ❶, поэтому одно изображение совместно используется *всеми* Злодеями.

Метод `__init__()` ❷ выбирает произвольный размер для каждого нового Злодея, поэтому пользователь видит Злодеев разного размера. Он выбирает случайные координаты *x* и *y*, которые поместят картинку сразу над окном. Затем создает объект `Image` и уменьшает масштаб изображения до выбранного размера ❸. И наконец, он выбирает произвольную скорость.

Менеджер Злодеев, код для которого я сейчас продемонстрирую, вызывает метод `update()` ❹ в каждом фрейме: здесь код перемещает местоположение Злодея вниз на некоторое количество пикселей, представляющих его скорость. Если Злодей вышел за пределы нижней игровой области, мы возвращаем `True`, чтобы сообщить, что его можно удалить. В противном случае

возвращаем False, чтобы сообщить менеджеру Злодеев оставить этого Злодея в окне.

Метод draw() **4** рисует Злодея в его новом местоположении.

Метод collide() **5** проверяет, пересекаются ли Игрок и Злодей.

Класс BaddieMgr, продемонстрированный в листинге 16.9, создает список объектов Baddie и управляет им; это классический пример объекта диспетчера объектов.

```
# Класс BaddieMgr
class BaddieMgr():
    ADD_NEW_BADDIE_RATE = 8 # насколько часто добавлять нового Злодея

1    def __init__(self, window):
        self.window = window
        self.reset()

2    def reset(self): # Вызываем при начале новой игры
        self.baddiesList = []
        self.nFramesTilNextBaddie = BaddieMgr.ADD_NEW_BADDIE_RATE

3    def update(self):
        # обновляем каждого Злодея,
        # считаем, сколько Злодеев выпало за пределы нижней части окна
        nBaddiesRemoved = 0
4        baddiesListCopy = self.baddiesList.copy()
        for oBaddie in baddiesListCopy:
5            deleteMe = oBaddie.update()
            if deleteMe:
                self.baddiesList.remove(oBaddie)
                nBaddiesRemoved = nBaddiesRemoved + 1

        # проверяем, пришло ли время добавить нового Злодея
6        self.nFramesTilNextBaddie = self.nFramesTilNextBaddie - 1
        if self.nFramesTilNextBaddie == 0:
            oBaddie = Baddie(self.window)
            self.baddiesList.append(oBaddie)
            self.nFramesTilNextBaddie = BaddieMgr.ADD_NEW_BADDIE_RATE

        # возвращаем число убранных Злодеев
        return nBaddiesRemoved

7    def draw(self):
        for oBaddie in self.baddiesList:
            oBaddie.draw()
```

```

8 def hasPlayerHitBaddie(self, playerRect):
    for oBaddie in self.baddiesList:
        if oBaddie.collide(playerRect):
            return True
    return False

```

Листинг 16.9. Класс BaddieMgr

Метод `__init__()` вызывает ❶ собственный метод `BaddieMgr.reset()`, чтобы установить список объектов `Baddie` в пустой список. Мы применяем подсчет фреймов для относительно частого создания нового Злодея, чтобы игра оставалась интересной. Используем переменную экземпляра `self.nFramesTilNextBaddie` для подсчета фреймов.

Метод `reset()` ❷ вызывается при начале нового раунда игры. Он очищает список Злодеев и сбрасывает счетчик фреймов. В методе `update()` ❸ осуществляется реальное управление Злодеями. Нашей целью здесь является перебрать всех Злодеев, говоря каждому обновить его местоположение и удаляя тех, кто выпал за пределы нижней части окна. Однако существует потенциальная ошибка. Если вы просто перебираете список и убираете элемент согласно критерию удаления, список немедленно сжимается. Когда это происходит, элемент, расположенный непосредственно за удаленным, окажется пропущен; в этом цикле данному элементу не будет сказано обновить себя. Хотя я тогда не вдавался в подробности, мы столкнулись с аналогичной проблемой в главе 11 в игре «Шары», где нам нужно было исключать шары, которые улетали за пределы верхней части окна. Там я задействовал функцию `reversed()`, примененную к списку, для итерации в обратном порядке (см. листинг 11.6).

Здесь я реализовал более общее решение ❹. Подход, использованный в классе `BaddieMgr`, состоит в создании копии списка и итерации *скопированного* списка; затем, если мы нашли элемент, удовлетворяющий критерию удаления (в данном случае Злодея, который выпал за пределы нижней части окна), мы удаляем его (конкретного Злодея) из *исходного* списка. Используя этот подход, мы перебираем список, отличный от того, из которого мы удаляем элементы.

Когда мы проводим итерацию объектов Злодеев, вызов метода `update()` каждого объекта Злодея ❺ возвращает булево выражение: `False`, чтобы указать, что он все еще перемещается вниз в окне, или `True`, чтобы указать, что он выпал за пределы

нижней части. Мы считаем количество Злодеев, которые выпадают за пределы нижней части, и удаляем каждого из них из списка. В конце метода возвращаем подсчет в основной код, чтобы он мог обновить очки.

В каждом фрейме мы также проверяем, пришло ли время создать нового Злодея ⑥. Когда пройдем через константу `ADD_NEW_BADDIE_RATE` количества фреймов, мы создадим новый объект `Baddie` и добавим его в список Злодеев.

Метод `draw()` ⑦ перебирает список Злодеев и вызывает метод `draw()` каждого Злодея, чтобы он нарисовал себя в подходящем местоположении.

И наконец, метод `hasPlayerHitBaddie()` ⑧ проверяет, пересекается ли прямоугольник игрока с каким-либо Злодеем. Код перебирает список Злодеев и вызывает метод `collide()` каждого из них. Если произошло пересечение (перекрытие), тогда мы сообщаем об этом обратно в основной цикл, который завершает игру.

Файл: *Dodger/Goodies.py*

Классы `GoodieMgr` и `Goodie` очень похожи на классы `BaddieMgr` и `Baddie`. Менеджер Героев — это объект диспетчера объектов, который поддерживает список Героев. Отличие от менеджера Злодеев состоит в том, что он может помещать Героя случайным образом либо у левого края окна (в этом случае тот двигается вправо), либо у правого края (тогда тот двигается влево). Он также создает нового Героя спустя произвольное количество фреймов. Когда Игрок пересекается с Героем, пользователю присуждается 25 очков. Метод `update()` менеджера Героев использует метод, описанный в предыдущем разделе: он делает копию списка Героев и перебирает ее.

Файл: *Dodger/Player.py*

Класс `Player`, продемонстрированный в листинге 16.10, управляет пиктограммой Игрока и отслеживает, где она должна появиться в окне игры.

```
# Класс Player
--- пропущенные операторы импорта ---

class Player():
❶   def __init__(self, window):
```

```

self.window = window
self.image = pygame.image.load(window,
                                (-100, -100), 'images/player.png')
playerRect = self.image.getRect()
self.maxX = WINDOW_WIDTH - playerRect.width
self.maxY = GAME_HEIGHT - playerRect.height

# В каждом фрейме перемещаем пиктограмму Игрока на позицию мыши
# Ограничиваем координаты x и y игровой областью окна
❷ def update(self, x, y):
    if x < 0:
        x = 0
    elif x > self.maxX:
        x = self.maxX
    if y < 0:
        y = 0
    elif y > self.maxY:
        y = self.maxY

    self.image.setLoc((x, y))
    return self.image.getRect()

❸ def draw(self):
    self.image.draw()

```

Метод `__init__()` ❶ загружает изображение пиктограммы Игрока и устанавливает количество переменных экземпляра для дальнейшего использования.

Метод `update()` ❷ вызывается в каждом фрейме сценой Игры. Основная идея заключается в отображении пиктограммы Игрока на местоположении мыши, которое передается. Мы выполняем несколько проверок, чтобы убедиться, что пиктограмма остается внутри прямоугольника игровой области. В каждом фрейме метод `update()` возвращает обновленный прямоугольник пиктограммы Игрока, так что основной код `Play` в листинге 16.6 может проверить, пересекается ли тот с каким-либо Злодеем или Героем.

И наконец, метод `draw()` ❸ рисует пиктограмму Игрока в новом местоположении.

Использование менеджера Героев, менеджера Злодеев и объекта `Player` четко демонстрирует силу ООП. Мы можем просто отправлять сообщения этим объектам, прося их обновлять или сбрасывать себя, и в ответ они выполняют все необходимое.

Менеджеры Героев и Злодеев передают эти сообщения всем Героям и Злодеям, которыми они управляют.

Файл: *Dodger/SceneHighScores.py*

Сцена Лучших результатов отображает топ-10 высших очков (и имена игроков) в таблице и позволяет пользователю, результаты которого попали в топ-10, ввести по желанию свое имя и очки в таблицу. Она создает экземпляр объекта `HighScoresData`, чтобы управлять фактическими данными, включая чтение и запись файлов данных. Это позволяет сцене Лучших результатов обновлять таблицу и отвечать на запросы от сцены Игры по поводу текущих высших и низших очков в таблице.

В листингах 16.11–16.13 содержится код класса `SceneHighScores`. Мы начинаем с методов `__init__()` и `getSceneKey()` в листинге 16.11.

```
# Сцена Лучших результатов
--- пропущенные операторы импорта, showCustomAnswersDialog
и showCustomResetDialog ---

class SceneHighScores(pyghelpers.Scene):
    def __init__(self, window):
        self.window = window
        self.oHighScoresData = HighScoresData()

        self.backgroundImage = pygwidgets.Image(self.window, (0, 0),
            'images/highScoresBackground.jpg')

        self.namesField = pygwidgets.DisplayText(self.window,
            (260, 84), '', fontSize=48,
            textColor=BLACK,
            width=300, justified='left')

        self.scoresField = pygwidgets.DisplayText(self.window,
            (25, 84), '', fontSize=48,
            textColor=BLACK,
            width=175, justified='right')

        self.quitButton = pygwidgets.CustomButton(self.window,
            (30, 650),
            up='images/quitNormal.png',
            down='images/quitDown.png',
            over='images/quitOver.png',
            disabled='images/quitDisabled.png')
```

```

self.backButton = pygwidgets.CustomButton(self.window,
                                           (240, 650),
                                           up='images/backNormal.png',
                                           down='images/backDown.png',
                                           over='images/backOver.png',
                                           disabled='images/backDisabled.png')

self.resetScoresButton = pygwidgets.CustomButton(self.window,
                                                  (450, 650),
                                                  up='images/resetNormal.png',
                                                  down='images/resetDown.png',
                                                  over='images/resetOver.png',

❷      self.showHighScores()

❸      def getSceneKey(self):
        return SCENE_HIGH_SCORES

```

Листинг 16.11. Методы `__init__()` и `getSceneKey()` класса `SceneHighScores`

Метод `__init__()` ❶ создает экземпляр класса `HighScoresData`, который поддерживает все данные для сцены Лучших результатов. Затем мы создаем все изображения, поля и кнопки для этой сцены. В конце инициализации вызываем `self.showHighScores()` ❷ для заполнения полей имени и очков.

Метод `getSceneKey()` ❸ возвращает уникальный ключ для сцены и должен реализовываться во всех сценах.

В листинге 16.12 показан код для метода `enter()` класса `SceneHighScores`.

```

❶      def enter(self, newHighScoreValue=None):
        # Это может быть вызвано двумя различными способами:
        # 1. Если нет новых лучших результатов, newHighScoresData
        # будет равно None
        # 2. newHighScoresData - это очки текущей игры, попадают
        # в топ-10

❷      if newHighScoreValue is None:
        return # ничего не делаем

❸      self.draw() # рисуем перед демонстрацией диалогового окна
        # У нас есть новые лучшие результаты, отправленные
        # из сцены Игры
        dialogQuestion = ('To record your score of ' +
                           str(newHighScoreValue) + ',\n' +
                           'please enter your name:')

❹      playerName = showCustomAnswerDialog(self.window,
                                           dialogQuestion)

```

```

5         if playerName is None:
            return # пользователь нажал Cancel

        # добавляем пользователя и очки в лучшие результаты
        if playerName == '':
            playerName = 'Anonymous'
6     self.oHighScoresData.addHighScore(playerName,
                                         newHighScoreValue)

    # отображаем обновленную таблицу лучших результатов
    self.showHighScores()

```

Листинг 16.12. Метод enter() класса SceneHighScores

Менеджер сцен вызывает метод enter() сцены Лучших результатов ❶ при переходе к ней из сцены Игры. Если результаты завершенной пользователем игры не попадают в топ-10, этот метод просто завершает работу ❷. Но если игрок заработал очки из топ-10, метод enter() вызывается с дополнительным значением — результатом только что завершенной игры.

В этом случае мы вызываем draw() ❸, чтобы нарисовать содержимое сцены Лучших результатов, прежде чем отобразить диалоговое окно, предлагающее пользователю добавить свои очки в список. Затем вызываем промежуточную функцию showCustomAnswerDialog(), которая создает и отображает пользовательское диалоговое окно ❹, продемонстрированное на рис. 16.7.

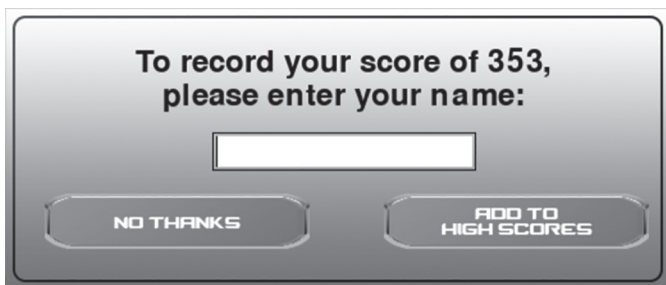


Рис. 16.7. Окно CustomAnswerDialog, позволяющее пользователю добавить свое имя в список лучших результатов

Если пользователь выбирает NO THANKS (Нет, спасибо), мы получаем возвращенное значение None и пропускаем остальную часть этого метода ❹. В противном случае берем возвращенное имя и добавляем имя и очки в таблицу ❺, вызывая метод объекта HighScoresData. И наконец, обновляем поля,

вызывая метод `showHighScores()`. Если в его вызове нет результатов ❷, ничего не нужно делать, поскольку текущий список уже отображен.

Листинг 16.13 содержит код для остальных методов этого класса.

```
def showHighScores(self): ❶
    # получаем очки и имена, отображаем их в двух полях
    scoresList, namesList =
        self.oHighScoresData.getScoresAndNames()
    self.namesField.setValue(namesList)
    self.scoresField.setValue(scoresList)

def handleInputs(self, eventsList, keyPressedList): ❷
    for event in eventsList:
        if self.quitButton.handleEvent(event):
            self.quit()

        elif self.backButton.handleEvent(event):
            self.goToScene(SCENE_PLAY)

        elif self.resetScoresButton.handleEvent(event):
            confirmed = showCustomResetDialog(self.window, ❸
                'Are you sure you want to \n
                RESET the high scores?')
            if confirmed:
                self.oHighScoresData.resetScores()
                self.showHighScores()

def draw(self): ❹
    self.backgroundImage.draw()
    self.scoresField.draw()
    self.namesField.draw()
    self.quitButton.draw()
    self.resetScoresButton.draw()
    self.backButton.draw()

def respond(self, requestID): ❺
    if requestID == HIGH_SCORES_DATA:
        # сцена Игры запросила высшие и низшие результаты
        # создаем словарь и возвращаем его в сцену Игры
        highestScore, lowestScore =
            self.oHighScoresData.getHighestAndLowest()
    return {'highest':highestScore, 'lowest':lowestScore}
```

Листинг 16.13. Методы `showHighScores()`, `handleInputs()`, `draw()` и `respond()` класса `SceneHighScores`

Метод `showHighScores()` ❶ начинается с запроса у объекта `HighScoresData` двух списков: топ-10 имен и результатов. Он берет возвращенные списки и отправляет их в два демонстрируемых поля отображения. Если вы передадите список в метод `setValue()` объекта `DisplayText`, он отобразит каждый элемент в виде отдельной строки. Мы используем объекты `DisplayText`, потому что `self.namesField` выровнено по левому краю, в то время как `self.scoresField` выровнено по правому краю.

Метод `handleInputs()` ❷ должен только проверить и отреагировать на нажатие пользователем кнопок **Quit**, **Back** и **Reset Scores**. Поскольку кнопка **Reset Scores** стирает данные, мы должны запросить подтверждение, прежде чем выполнить такое действие. Следовательно, когда пользователь щелкает по этой кнопке, мы вызываем промежуточную функцию `showCustomResetDialog()` ❸, чтобы вывести диалоговое окно, просящее пользователя подтвердить, что он действительно хочет очистить все текущие результаты.

Метод `draw()` ❹ рисует все элементы в окне.

И наконец, метод `respond()` ❺ разрешает другой сцене запрашивать информацию. Именно это позволяет сцене Игры запросить текущие высшие результаты и десятые высшие результаты — минимальные очки, чтобы определить, помещать ли игрока в список лучших результатов. Вызывающий отправляет значение, которое указывает, какую информацию он ищет. В данном случае запрашиваемой информацией является `HIGH_SCORES_DATA`, константа совместного использования из файла *Constants.py*. Этот метод выстраивает словарь из двух запрошенных значений и возвращает его вызывающей сцене.

Файл: *Dodger/HighScoresData.py*

Последний класс — `HighScoresData`, отвечающий за управление информацией лучших результатов. Он читает и записывает данные в виде файла формата JSON. Они всегда хранятся в определенном порядке, от большего количества очков к меньшему. Например, файл, представляющий 10 высших результатов, может выглядеть следующим образом:

```
[['Moe', 987], ['Larry', 812], ... ['Curly', 597]]
```

В листинге 16.14 продемонстрирован код класса `HighScoresData`.

```

# Класс HighScoresData
from Constants import *
from pathlib import Path
import json

class HighScoresData():
    """Файл данных сохранен в виде списка списков в формате JSON.
    Каждый список состоит из имени и очков:
        [[имя, очки], [имя, очки], [имя, очки] ...]
    В этом классе все очки хранятся в self.scoresList.
    Очки в списке упорядочены определенным образом, от высших к низшим.
    """
    ❶ def __init__(self):
        self.BLANK_SCORES_LIST = N_HIGH_SCORES * [['-----', 0]]
    ❷ self.oFilePath = Path('HighScores.json')

        # пробуем открыть и загрузить данные из файла данных
        try:
    ❸ data = self.oFilePath.read_text()
        except FileNotFoundError: # нет файла, ставим в значение
                                # "пустые очки" и сохраняем
    ❹ self.scoresList = self.BLANK_SCORES_LIST.copy()
            self.saveScores()
            return

        # Файл существует, загружаем очки JSON-файла
    ❺ self.scoresList = json.loads(data)

    ❻ def addHighScore(self, name, newHighScore):

        # ищем подходящее место для добавления новых высших очков
        placeFound = False
        for index, nameScoreList in enumerate(self.scoresList):
            thisScore = nameScoreList[1]
            if newHighScore > thisScore:
                # вставляем в соответствующее место, удаляем
                # последнюю запись
                self.scoresList.insert(index, [name, newHighScore])
                self.scoresList.pop(N_HIGH_SCORES)
                placeFound = True
                break
        if not placeFound:
            return # результат не принадлежит списку

        # сохраняем обновленные результаты
        self.saveScores()

```

```

7 def saveScores(self):
    scoresAsJson = json.dumps(self.scoresList)
    self.oFilePath.write_text(scoresAsJson)

8 def resetScores(self):
    self.scoresList = self.BLANK_SCORES_LIST.copy()
    self.saveScores()

9 def getScoresAndNames(self):
    namesList = []
    scoresList = []
    for nameAndScore in self.scoresList:
        thisName = nameAndScore[0]
        thisScore = nameAndScore[1]
        namesList.append(thisName)
        scoresList.append(thisScore)

    return scoresList, namesList

10 def getHighestAndLowest(self):
    # Элемент 0 - это высшая запись, элемент -1 - низшая
    highestEntry = self.scoresList[0]
    lowestEntry = self.scoresList[-1]
    # получаем результат (элемент 1) каждого подписка
    highestScore = highestEntry[1]
    lowestScore = lowestEntry[1]
    return highestScore, lowestScore

```

Листинг 16.14. Класс HighScoresData

В методе `__init__()` ❶ мы сначала создадим список всех пустых записей. Используем модуль `Path`, чтобы создать объект пути с местоположением файла данных ❷.

ПРИМЕЧАНИЕ Продемонстрированный в приведенном листинге путь находится в той же папке, что и код. Это хорошо подходит для изучения концепции файлового ввода и вывода. Однако, если вы собираетесь совместно использовать свою программу с другими людьми для игры на их компьютерах, будет лучше задействовать другой путь в домашней папке пользователя. Его можно построить следующим образом:

```

import os.path
DATA_FILE_PATH = os.path.expanduser('~ / DodgerHighScores.json')

или

from pathlib import Path
DATA_FILE_PATH = Path('~ / DodgerHighScores.json').expanduser()

```

Далее посмотрим, есть ли у нас уже сохраненные высшие очки, проверив наличие файла данных ❸. В противном случае ❹ мы читаем содержимое файла ❺ и преобразуем формат JSON в список списков.

Метод `addHighScores()` ❻ отвечает за добавление нового высшего результата в список. Поскольку данные всегда хранятся в определенном порядке, мы перебираем список очков, пока не найдем соответствующий индекс, и вставляем новое имя и очки. Поскольку эта операция расширит список, мы удаляем последний элемент, чтобы сохранить только топ-10. Мы также проверим, действительно ли новые очки должны быть добавлены в список. И наконец, вызываем `saveScores()`, чтобы сохранить результаты в файл данных.

Метод `saveScores()` ❼ сохраняет данные очков в файл формата JSON. Он вызывается из разных мест.

Метод `resetScores()` ❺ вызывается, когда пользователь сообщает, что хочет сбросить все имена и очки до начальной точки (все пустые имена и очки установлены в значение ноль). Мы вызываем `saveScores()`, чтобы перезаписать файл данных.

Метод `getScoresAndNames()` ❻ вызывается сценой Лучших результатов, чтобы получить топ-10 очков и имен. Мы перебираем список списков данных лучших результатов, чтобы создать один с очками и другой с именами; оба списка возвращаются.

И наконец, сцена Лучших результатов вызывает метод `getHighestAndLowest()` ❼, чтобы получить большие и меньшие очки в таблице. Он использует эти сведения, чтобы определить, подходит ли результат пользователя для добавления его имени и очков в таблицу лучших результатов.

Дополнения к игре

Общая архитектура является модульной, что обеспечивает простоту модификаций. Каждая сцена инкапсулирует собственные данные и методы, в то время как взаимодействие и навигация обрабатываются менеджером сцен. Дополнения могут обрабатываться в одной сцене, никак не влияя на другие.

Например, вы можете захотеть, чтобы пользователь начал игру с несколькими жизнями. Тогда, вместо того чтобы завершить игру, как только пиктограмма Игрока заденет Злодея, программа уменьшит количество жизней на единицу, а игра

завершится, когда они все закончатся. Подобное изменение относительно легко реализовать, и оно повлияет только на сцену Игры.

Другая идея заключается в том, что пользователь может начать с небольшим количеством бомб, которые может взорвать, оказавшись в безвыходном положении, тем самым уничтожив Злодеев в пределах заданного радиуса вокруг пиктограммы Игрока. Число бомб будет уменьшаться после каждого их применения, пока не достигнет нуля. Это изменение повлияет только на код сцены Игры и менеджера Злодеев.

Или, возможно, вы захотите отслеживать больше лучших результатов, скажем 20 вместо 10. Подобное изменение можно выполнить в сцене Лучших результатов, не повлияв на сцены Игры и Приветствия.

Выводы

В этой главе я продемонстрировал, как создавать и использовать диалоговые окна **Yes/No** и диалоговые окна с ответом: как текстовую, так и пользовательскую версии. Затем мы сосредоточились на создании полноценной объектно-ориентированной программы игры **Dodger**.

Мы применили модуль `pygame.widgets` для всех кнопок, отображений текста и полей ввода текста, а модуль `pygame.helpers` — для всех диалоговых окон. `SceneMgr` позволил нам разбить игру на более мелкие, лучше управляемые части (объекты `Scene`) и перемещаться между сценами.

Игра использовала и продемонстрировала следующие объектно-ориентированные концепции.

Инкапсуляция. Каждая сцена обрабатывает только то, что специфично для этой сцены.

Полиморфизм. Каждая сцена реализует одинаковые методы.

Наследование. Каждая сцена наследует от базового класса `Scene`.

Объект диспетчера объектов. Сцена Игры использует композицию для создания объекта менеджера Злодеев `self`.

oBaddieMgr и объекта менеджера Героев self.oGoodieMgr, каждый из которых управляет списком своих объектов.

Общие константы. Мы применяем отдельные модули для Героев и Злодеев, а файл *Constants.py* позволяет с легкостью совместно использовать константы в разных модулях.

17

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ И РЕЗЮМЕ



В завершающей главе я познакомлю вас с еще одной концепцией объектно-ориентированного программирования — с *шаблонами проектирования*, которые представляют собой многократно применяемые решения ООП для часто возникающих программных проблем. В этой книге мы уже видели один шаблон проектирования: использование объекта диспетчера объектов для управления списком или словарем объектов. Множество полноценных книг было написано по теме шаблонов проектирования, поэтому мы не будем рассматривать их все. В этой главе я сосредоточусь на шаблоне MVC (Model View Controller, модель-представление-контроллер), который используется для разбивки системы на более мелкие, более управляемые и легче изменяемые части. И в конце я подведу краткие итоги по ООП.

Модель Представление Контроллер

Шаблон проектирования *Модель Представление Контроллер* (MVC, *Model View Controller*) обеспечивает четкое разделение между набором данных и способом их представления пользователю. Он разделяет функциональность на три части: модель, представление и контроллер. У каждой есть четко определенные обязанности, и каждая реализуется одним или несколькими объектами.

Модель хранит данные. Представление отвечает за изображение информации модели одним из множества возможных способов. Контроллер обычно создает модель и просматривает объекты, обрабатывает все взаимодействия пользователей, сообщает модели об изменениях и говорит представлению отображать данные. Благодаря этому разделению система становится невероятно проста в обслуживании и модификации.

Пример отображения файла

В качестве хорошего примера шаблона MVC рассмотрим способы отображения файлов в macOS Finder или Windows Explorer. Допустим, у нас есть папка, содержащая четыре файла и подпапку. Конечный пользователь может решить отобразить эти элементы в виде списка, как на рис. 17.1.



Рис. 17.1. Файлы в папке, показанные в виде списка

В качестве альтернативы пользователь может решить отобразить эти же элементы в виде пиктограмм, как на рис. 17.2.

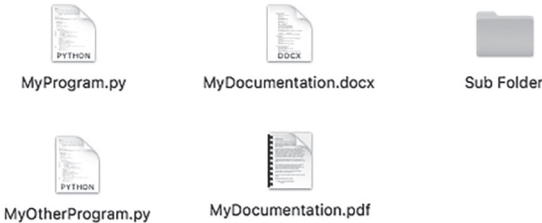


Рис. 17.2. Файлы в папке, показанные в виде пиктограмм

Исходные данные для обоих отображений идентичны, но представление информации пользователю отличается. В этом примере данные — список файлов и подпапок; они хранятся в объекте модели. Объект представления отображает данные любым выбранным пользователем способом: в виде списка, пиктограмм, в виде подробного списка и так далее. Контроллер говорит представлению отображать информацию в выбранной пользователем компоновке.

Пример статистического отображения

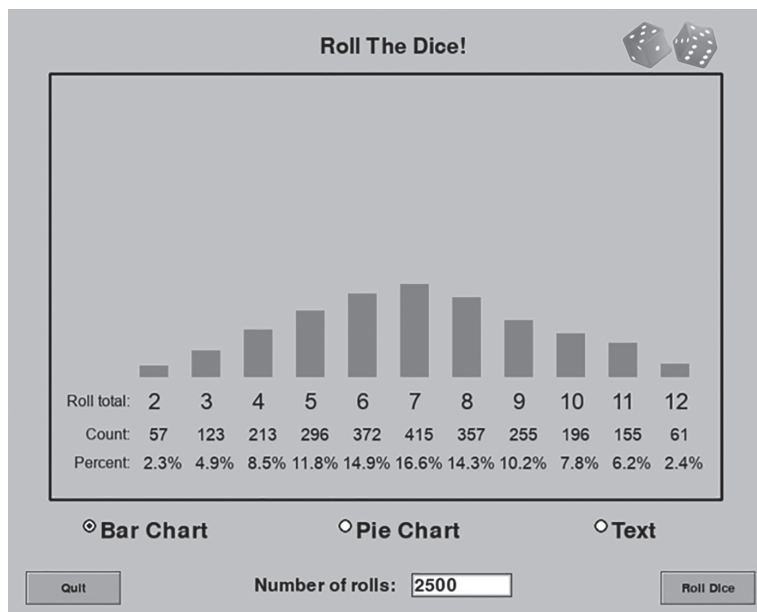


Рис. 17.3. Данные броска костей в виде гистограммы

В качестве более показательного примера шаблона MVC давайте рассмотрим программу, которая имитирует бросание пары костей много раз и отображает результаты. В каждом броске мы станем добавлять значения двух костей, поэтому сумма, которую назовем *результатом*, должна быть между 2 и 12. Данные состоят из подсчета количества бросков каждого результата и процента от общего количества бросков, которое составляет каждый результат. Программа может отображать их в трех различных представлениях: в виде гистограммы, диаграммы и текстовой таблицы. По умолчанию она использует гистограмму и отображает результаты после имитации броска пары костей 2500 раз. Поскольку эта программа предназначена

лишь для рабочей демонстрации шаблона MVC, мы сгенерируем выходные данные, используя `pygame` и `pygame_widgets`. Для знакомства с более профессиональными графиками и отображениями я предлагаю вам обратиться в библиотеки визуализации данных Python, такие как `Matplotlib`, `Seaborn`, `Plotly`, `Bokeh` и другие, которые спроектированы для этой цели.

Рис. 17.3 демонстрирует данные, отображенные в виде гистограммы.

Под каждым столбцом указан результат, количество бросков с этим результатом и его процент от общего числа бросков. Высота каждого столбца соответствует количеству (или проценту). Нажатие кнопки **Roll Dice** вновь запускает имитацию, используя число бросков, указанное в поле ввода. Пользователь может щелкать по разным переключателям, чтобы продемонстрировать разные представления тех же данных. Если он выбирает переключатель **Pie Chart**, информация отображается как на рис. 17.4.

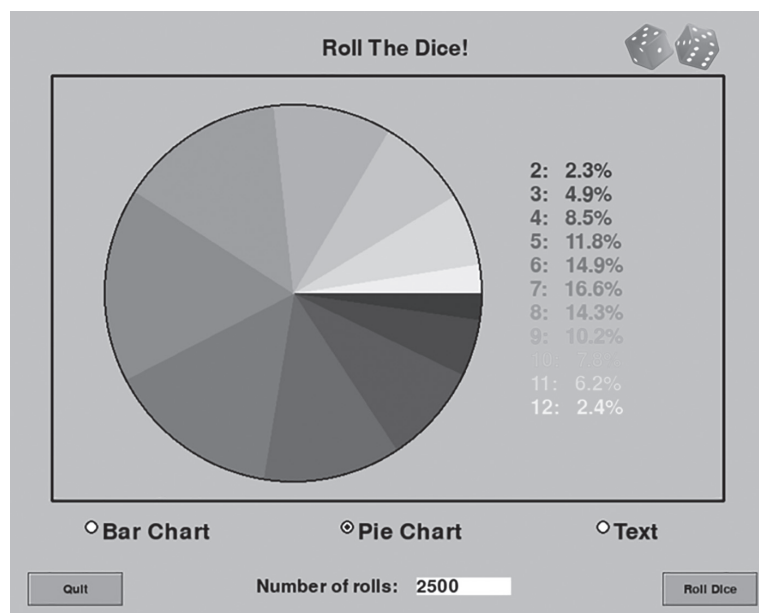


Рис. 17.4. Данные броска костей в виде диаграммы

Если пользователь выбирает переключатель **Text**, данные отображаются как на рис. 17.5.

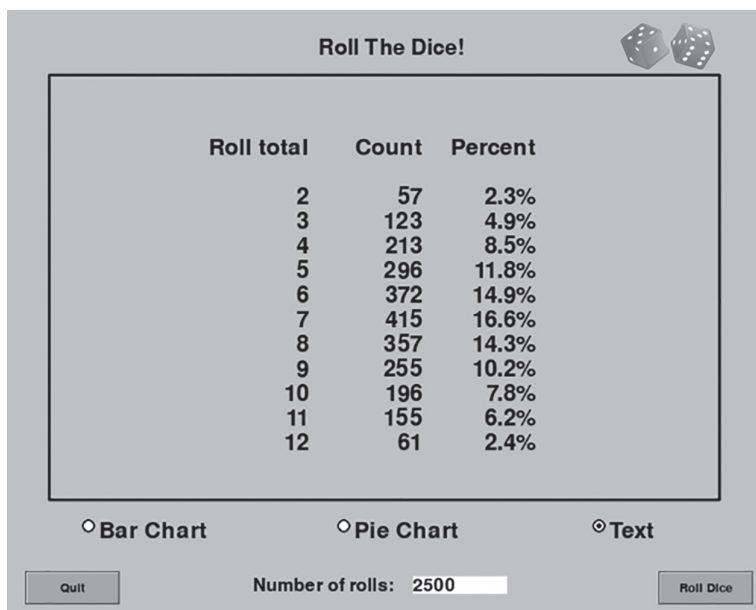


Рис. 17.5. Данные броска костей в виде текста

Пользователь может поменять значение поля **Number of rolls**, чтобы бросать кости столько раз, сколько он хочет. Эти данные в этой программе основаны на статистике и произвольности. Очевидно, что при других размерах выборки точное количество будет отличаться, но процентное соотношение всегда должно оставаться примерно одинаковым.

Я не буду показывать здесь весь листинг программы, но сосредоточусь на нескольких ключевых строках, которые демонстрируют установку и поток управления в шаблоне MVC. Полная программа доступна для загрузки вместе с остальными ресурсами этой книги в папке *MVC_RollTheDice*. Папка содержит следующие файлы:

Main _ MVC.py — основной файл Python;

Controller.py содержит класс *Controller*;

Model.py содержит класс *Model*;

BarView.py содержит класс *BarView*, который отображает гистограмму;

Bin.py содержит класс *Bin*, который рисует один столбец в гистограмме;

PieView.py содержит класс `PieView`, который отображает диаграмму;

TextView.py содержит класс `TextView`, который отображает текстовое представление;

Constants.py содержит константы, используемые несколькими модулями.

Основная программа создает экземпляры объекта `Controller` и выполняет основной цикл. Код в основном цикле перенаправляет все события (кроме события `pygame.QUIT`) для обработки контроллеру.

Контроллер

Контроллер является диспетчером всей программы. Он начинает с создания экземпляра объекта `Model`. Затем создает по экземпляру каждого из объектов различных представлений: `BarView`, `PieView` и `TextView`. Ниже представлен начальный код в методе `__init__()` класса `Controller`:

```
# создаем экземпляр модели
self.oModel = Model()
# создаем экземпляры объектов различных представлений
self.oBarView = BarView(self.window, self.oModel)
self.oPieView = PieView(self.window, self.oModel)
self.oTextView = TextView(self.window, self.oModel)
```

Когда объект `Controller` создает экземпляры этих объектов `View`, он передает объект `Model`, чтобы каждый объект `View` мог запросить информацию напрямую у модели. Различные реализации шаблона MVC могут по-разному обрабатывать взаимодействия между этими тремя элементами; например, контроллер может действовать как посредник, запрашивая данные у модели и перенаправляя их текущему представлению, вместо того чтобы разрешить модели и представлению взаимодействовать напрямую.

Контроллер реагирует на все, находящееся за пределами черного прямоугольника в окне, включая название, изображение костей и переключатели. Он рисует кнопки **Quit** и **Roll Dice** и реагирует, когда они нажимаются, а также обрабатывает любые изменения, которые вносит пользователь в число бросков.

Объект `Controller` хранит текущий объект `View`, который определяет, какое представление отображается в данный момент. По умолчанию мы устанавливаем его в объект `BarView` (гистограмма):

```
self.oView = self.oBarView
```

Когда пользователь щелкает по кнопке переключателя, `Controller` устанавливает свой текущий объект `View` в новое выбранное представление и говорит новому объекту `View` обновить себя, вызывая его метод `update()`:

```
if self.oBarButton.handleEvent(event):
    self.oView = self.oBarView
    self.oView.update()
elif self.oPieButton.handleEvent(event):
    self.oView = self.oPieView
    self.oView.update()
elif self.oTextButton.handleEvent(event):
    self.oView = self.oTextView
    self.oView.update()
```

При запуске и каждый раз, когда пользователь щелкает по кнопке **Roll Dice**, контроллер проверяет число бросков, указанное в поле **Number of rolls**, и говорит модели сгенерировать новые данные:

```
self.oModel.generateRolls(nRounds)
```

Все представления полиморфны, поэтому в каждом фрейме объект `Controller` вызывает метод `draw()` текущего объекта `View`:

```
self.oView.draw() # инструктируем текущее представление
                  # нарисовать себя
```

Модель

Модель ответственна за получение (и потенциальное обновление) информации. В этой программе объект `Model` прост: он имитирует бросок пары костей множество раз, сохраняет результаты в переменных экземпляра и сообщает данные, когда их запрашивает объект `View`.

Когда запрашивают сгенерированные данные, модель выполняет цикл, имитирующий бросок костей, и сохраняет его итог в двух словарях: `self.rollsDict`, который использует каждый результат в качестве ключа и количество в качестве значения, и `self.percentsDict`, который использует каждый результат в качестве ключа и процентное соотношение бросков в качестве значения.

В более сложных программах модель может получать свои данные из базы данных, интернета или других источников. Например, объект `Model` может обслуживать биржевую информацию, демографические данные, сведения о городском жилье, показатели температуры и так далее.

В этой модели метод `getRoundsRollsPercents()` вызывается объектами `View` для извлечения всех данных сразу. Однако модель в состоянии содержать больше информации, чем может понадобиться какому-либо представлению. Следовательно, различные объекты `View` способны вызывать различные методы в объекте `Model`, чтобы запрашивать различную информацию от одной и той же модели. Я включил в пример программы несколько дополнительных методов геттера (`getNumberOfRounds()`, `getRolls()` и `getPercents()`), программист может их использовать при создании нового объекта `View`, чтобы получить только те данные, которые захочет отобразить это новое представление.

Представление

Объект `View` отвечает за показ данных пользователю. В нашем примере программы у нас есть три различных объекта `View`, которые демонстрируют одну и ту же исходную информацию в трех различных формах; каждый отображает информацию внутри черного прямоугольника в окне. При запуске и когда пользователь щелкает по кнопке **Roll Dice**, контроллер вызывает метод `update()` текущего объекта `View`. Затем все объекты `View` выполняют один и тот же вызов объекта `Model`, чтобы получить текущие данные:

```
nRounds, resultsDict, percentsDict =  
    self.oModel.getRoundsRollsPercents()
```

Затем объект `View` форматирует информацию собственным способом и представляет ее пользователю.

Преимущества шаблона MVC

Шаблон проектирования MVC разбивает обязанности на различные классы, которые действуют независимо, но работают совместно. Выстраивание компонентов как отдельных классов и минимизация взаимодействий между полученными объектами позволяют каждому отдельному компоненту быть проще и менее подверженным ошибкам. Как только определен интерфейс каждого компонента, код классов могут писать даже разные программисты.

В подходе MVC каждый компонент демонстрирует основные концепции ООП: инкапсуляцию и абстракцию. Используя структуру объекта MVC, модель может изменить внутренний способ представления данных, не влияя на контроллер или представление. Как ранее упоминалось, модель может содержать больше данных, чем требуется любому отдельно взятому представлению. Пока контроллер не меняет способ своего взаимодействия с моделью и та продолжает возвращать запрашиваемую информацию представлению согласованным способом, модель может добавлять новые данные, не нарушая работу системы.

Модель MVC также упрощает добавление улучшений. Например, в нашей программе бросания костей она способна отслеживать количество различных сочетаний бросков двух костей, составляющих каждый результат, например получение 5 путем выброса 1 и 4 или 2 и 3. Затем мы можем изменить представление BarChart, чтобы получить эту дополнительную информацию от модели и показать каждый столбец, разделенный на более мелкие столбцы для отображения процентного соотношения каждого сочетания.

Любой объект View полностью настраиваемый. TextView может использовать другие шрифты и их размеры или другую компоновку. PieView способен раскрасить сектора различными цветами. Столбцы в BarView могут быть толще или выше, отображаться различными цветами или даже горизонтально. Любые подобные изменения будут выполняться только в соответствующем объекте View, полностью независимом от модели или контроллера.

Шаблон MVC также упрощает добавление нового способа представления данных путем написания нового класса View.

Единственные необходимые дополнительные изменения заключаются в том, чтобы заставить контроллер нарисовать еще один переключатель, создать экземпляр нового объекта View и вызвать метод `update()` нового объекта View, когда пользователь выбирает новое представление.

ПРИМЕЧАНИЕ MVC и другие шаблоны проектирования не зависят от какого-либо конкретного языка программирования и могут использоваться в любом языке, который поддерживает ООП. Если вы хотите узнать больше, я рекомендую вам поискать в интернете шаблоны проектирования ООП, такие как шаблоны Factory, Flyweight, Observer и Visitor; существует множество видео и текстовых обучающих руководств (а также книг), доступных по всем этим шаблонам. Для общего ознакомления книга «Приемы объектно-ориентированного проектирования. Паттерны проектирования»* Эрика Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса (Банда четырех) считается библией шаблонов проектирования.

Резюме

Размышляя об объектно-ориентированном программировании, не забывайте мое изначальное определение объекта: данные плюс код, который воздействует на них с течением времени.

ООП предоставляет вам новый подход к программированию, предлагая простой и удобный способ группировки данных и кода, который с ними работает. Вы пишете классы и создаете из них экземпляры объектов. Каждый получает набор переменных экземпляра, определенных в классе, но переменные экземпляра в различных объектах могут содержать различные данные и остаются независимыми друг от друга. Методы объектов способны действовать по-разному, поскольку они работают с разными данными. Экземпляры объектов могут создаваться и уничтожаться в любое время.

* Русское издание: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2019.

При создании экземпляров нескольких объектов из одного класса вы, как правило, создаете список или словарь объектов, затем позднее перебираете его, вызывая методы каждого объекта.

И в качестве последнего напоминания — к трем основным принципам ООП относятся:

- **инкапсуляция** — все в одном месте, объекты владеют своими данными;
- **полиморфизм** — разные объекты могут реализовывать одинаковые методы;
- **наследование** — класс может расширить или изменить поведение другого класса.

Объекты часто работают в иерархиях; они могут использовать композицию, чтобы создавать экземпляры других объектов, и способны вызывать методы объектов низшего уровня, чтобы попросить их выполнить работу и предоставить информацию.

Для того чтобы у вас сложилось четкое визуальное представление ООП в действии, большинство примеров этой книги были сосредоточены на виджетах и других объектах, которые могут оказаться полезными в игровой среде. Я разработал пакеты `pygamewidgets` и `pyghelpers` для демонстрации множества методов ООП, они также помогут вам с легкостью использовать виджеты GUI в программах `pygame`. Надеюсь, что эти пакеты оказались полезны и вы продолжите использовать их для разработки собственных развлекательных или полезных программ.

И самое главное, я надеюсь, что вы поймете: объектно-ориентированное программирование представляет собой универсальный подход, и его можно применять в самых разных обстоятельствах. Каждый раз, когда вы видите две или более функций, которые должны работать с общим набором данных, вам следует подумать о создании класса и экземпляра объекта. Вероятно, вы также захотите рассмотреть возможность создания объекта диспетчера объекта для управления группой объектов.

С учетом всего вышесказанного я хотел бы поздравить вас: вы дочитали эту книгу до конца! Хотя, по сути, это должно рассматриваться как начало вашего путешествия

по объектно-ориентированному программированию. Надеюсь, что описанные в этой книге концепции дали структуру, на которую вы сможете опираться; но единственный способ по-настоящему разобраться с тем, как работает ООП, состоит в написании большого количества кода. Со временем вы начнете замечать шаблоны, которые снова и снова используете в вашем коде. Понимание того, как структурировать классы, — сложный процесс. При наработке опыта вам будет все легче создавать подходящие методы и переменные экземпляра в правильных классах.

Практика, практика и еще раз практика!

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- @property, декоратор 250
- >= (больше или равно), магический метод 279
- > (больше), магический метод 279
- <= (меньше или равно), магический метод 279
- < (меньше), магический метод 279
- != (не равно), магический метод 279
- + оператор, полиморфизм 276
- == (равно), магический метод 279
- abc (абстрактный базовый класс), модуль 323
- Account, класс 102
 - с исключениями 128
- Animation, класс (pygwidgets) 219, 412
- Balloon, класс 356
- Ball, класс
 - использование 186
 - создание 184
 - создание множества объектов 188
- Bank, класс 120
 - оптимизированный 130
- Bokeh, библиотека 494
- Card, класс 368
- Circle, класс 268
- Class, оператор 62
- Club, класс 242
- CountDownTimer, класс 397
- CountUpTimer, класс 393
- create(), метод 184
- curved brackets. См. parentheses
- CustomButton, класс (pygwidgets) 218
- CustomCheckBox, класс (pygwidgets) 219
- CustomRadioButton, класс (pygwidgets) 219
- Deck, класс 366, 369
- DimmerSwitch, класс 71
- displaying numbers. См. print()
- displaying text. См. print()
- DisplayMoney, класс 312
- DisplayText, класс (pygwidgets) 219
- Dodger, игра
 - дополнения к 488
 - новая 471
 - обзор 465
 - реализация 466
 - результаты 481
 - события щелчков 472
 - сцены 467
- Dragger, класс (pygwidgets) 219
- draw(), метод 184
- Ellipse, класс 274
- Employee, класс (наследование) 301
- Fraction, класс 291
- Game, объект 373

- Group, класс 222
- ImageCollection, класс (pygwidgets) 219
- Image, класс (pygwidgets) 219
- InputText, класс (pygwidgets) 219, 308
- interactive mode. См. shell
- JSON, формат 485
- len(), функция 236
- LIFO, принцип 257
- Lingo, язык 255
- Manager, класс (наследование) 302
- Matplotlib, библиотека 494
- MVC, шаблон проектирования 492
 - преимущества 499
- None, специальное значение 103
 - в качестве значения по умолчанию 216
- Person, класс 238
- Pip, пакетный менеджер 142
- Plotly, библиотека 494
- Pygame, пакет 139
 - API 203
 - виджеты 211
 - документация 143
 - знакомство с 141
 - объект rect 169
 - поддерживаемые форматы файлов 155
 - установка 142
 - шаблон для создания программ 151
- PygAnimation, класс 415
- Pyghelpers, модуль
 - установка 388
- Pygwidgets, библиотека 211
 - документация 220
 - классы 218, 229
 - классы анимации в 411
 - наследование 327
 - установка 219
- Python
 - каталог пакетов (PyPI) 219
 - написание класса на 61
 - прямой доступ к переменным экземпляра 238
 - реализации 336
 - типы данных 69
 - управление памятью 336
 - установка 20
- Python Editor. См. IDLE
- Raise, инструкция
 - пользовательские исключения и 126
- Rect, класс (pygame) 247
- regular intervals. См. clock
- RGB, значения цветов 147
- SceneMgr, класс 448
 - методы 451
- Seaborn, библиотека 494
- Self
 - параметр 62
 - префикс 63
 - смысл слова 95
- SimpleAnimation, класс 400
- SimpleButton, класс 191
 - код, использующий 194
- SimpleSpriteSheetAnimation, класс 405
- SimpleText, класс 199
- SpinBox, класс 332
- SpriteSheetAnimation, класс (pygwidgets) 219, 413
- Sprite, класс 222
- Square, класс 267

- Student, класс 251
- text. См. string
- TextButton, класс (pygwidgets) 218
- TextCheckBox, класс (pygwidgets) 219
- TextRadioButton, класс (pygwidgets) 219
- Timer, класс 389
- Triangle, класс 268
- Try/except, конструкция 125
- update(), метод 184
- Vector, класс 286
- Абсолютный путь 156
- Абстрактный
 - класс 323
 - метод 323
- Абстракция 257
- Анимация 399
 - запуск 402
 - классы в pygwidgets 411
 - основанная на местоположении 167
 - создание классов 400
- Аргументы
 - необязательные 212
 - параметры и 212
 - передача методу 79
 - переупорядочивание при вызове метода 96
- Ассессор 246
- Банковский счет
 - данные 37
 - класс 101
 - операции 36
- «Больше-меньше», игра 32
 - GUI-версия 365
 - представление данных 32
 - реализация 33
 - тестирование 377
- Вектор 285
- Воспроизведение
 - звуковых эффектов 173
 - фоновой музыки 175
- Время
 - отображение 392
- Выключатель освещения, пример 56
- Выход из программы 437
- Геттер 244
- Графический интерфейс пользователя (GUI) 141
- Декоратор 249
- Декорирование имени 249
- Деметры, закон 332
- Диаграмма состояний 425
- Диалоговое окно
 - Да/Нет 458
 - модальное 457
 - настройка пользовательского 460
 - предупреждение 459
 - с ответом 462
 - типы 458
- Диспетчер объектов 118
- Дуга, рисование 179
- Жизненный цикл объекта 336
- Звуки
 - воспроизведение 173
 - форматы 173
- Значения по умолчанию
 - None в качестве 216
 - выбор 217
- Иерархия классов 329
- Изображение
 - добавление 221
 - под- 406

- Инициализация 39
 - метод 62
 - параметры 83
- Инкапсуляция 233, 235
 - в классах pygame 254
 - интерпретации 238
 - с помощью функций 236
- Инстанцирование 61
- Интегрированная среда разработки (IDLE) 26
- Интерфейс 136, 203
- Исключение
 - AbortTransaction 128
 - использование 128
 - пользовательское 126
- Камень-ножницы-бумага, игра
 - использующая сцены 439
 - создание 423
- Карточные игры 379
 - с необычными колодами 379
- Каскадное удаление 341
- Клавиатура
 - обработка событий 161
 - одиночные нажатия клавиш 162
 - опрос 164
- Класс 51, 60
 - анимации 400
 - базовый 298
 - дочерний 299
 - иерархия 329
 - импорт кода 105
 - использование 85
 - название 62
 - написание на Python 61
 - наследование 298
 - наследование нескольких 317
 - под- 298
 - представление физического объекта 73
 - производный 299
 - родительский 299
 - создание более сложного 70
 - создание нескольких экземпляров
 - из одного 67
 - создание объекта из 66
 - супер- 299
- Клиент 237
- Ключевые слова 213
 - выбор 217
- Ключ сцены 434
- Кнопка
 - графика для 225
 - добавление 222
 - класс 191
 - обратный вызов 206
 - создание 190
- Код
 - повторное использование 35
 - создание тестового 107
- Колода для блэк-джека 379
- Композиция 119, 332
- Конечный автомат 194
 - пример pygame с 423
 - реализация 420
- Константа 344
 - модуль 349
- Контроллер 496
- Круг, рисование 180
- Линия, рисование 180
- Магические методы 277
 - в классе Rectangle 280
 - для векторов 285
 - для получения информации из объекта 289

- класс Fraction 291
- математических операторов 284
- операторы сравнения 278
- Ментальные модели 89
 - глубокая 92
- Меню, интерактивное
 - создание 115
- Метод 62
 - магический 277
 - отличие от функции 64
 - передача аргументов 79
 - порядок разрешения 333
 - специальный 277
- Многоугольник, рисование 180
- Модель 497
- Модуль 377
- Мутатор 246
- Наследование 233, 297
 - в pygame 327
 - в ООП 298
 - множественное 333
 - нескольких классов от одного базового 317
 - примеры из реального мира 307
 - реализация 300
 - сложность программирования с 331
- Непрерывная обработка 162
- Область видимости
 - глобальная 63
 - локальная 63
 - объекта 63
- Обратный вызов 204
 - использование с классом SimpleButton 206
 - создание 205
- Объект 61
 - вызов методов 66
- диспетчер 118
- использование памяти 93
- композиция 119
- определение 70
- подсчет 345
- подсчет ссылок 336
- создание из класса 66
- строковое представление значений в 288
- с уникальным идентификатором 112
- удаление 341
- Объектно-ориентированное программирование (ООП) 19, 31
 - основные принципы 233
 - шаблоны проектирования 491
- Объектно-ориентированное решение
 - классы 51
- Овал, рисование 180
- Ограничивающий прямоугольник 146
- Окно 143
 - диалоговое 457
 - организация пустого 151
 - приложения 220
 - система координат 144
- Оператор
 - арифметический 284
 - перегрузка 277
 - полиморфизм для 276
- Относительный путь 156
- Память
 - слоты 361
 - управление 336, 359
 - экономия 345
- Параметры
 - аргументы и 212
 - инициализации 83
 - ключевого слова 213

- позиционные 213
- Перегрузка операторов 277
- Переключатель
 - добавление 222
- Переменная
 - временная 112
 - глобальная 42, 63
 - инициализация 39
 - локальная 63
 - экземпляра 63, 238
- Пиксел 143
 - точка 145
 - цвет 147
- Поверхность 198
- Повторное использование кода 35
- Подсчет ссылок 336
- Позиционные параметры 213
- Полиморфизм 233, 263
 - в реальном мире 264
 - для операторов 276
 - проявляемый `pygame` 275
 - с использованием `pygame` 266
- Порядок разрешения методов 333
- Представление 498
- Предупреждения, диалоговое окно 459
- Примитив 176
- Программный интерфейс приложения (API) 203, 231
- Процедурная реализация
 - проблемы с 50
- Процедурное программирование 19
- Прямой доступ 238
 - безопасный 246
- Прямоугольник, рисование 181
- Путь 156
 - абсолютный 156
 - относительный 156
- Реализация 136, 203
- Рисование
 - изображения 155
 - фигур 176
- Россум, Гвидо ван 239
- Сборка мусора 343
- Сборщик мусора 338
- Свейгарт, Эл 457
- Свойство 250
- Сглаженная линия, рисование 179
- Сеттер 245
- Словарь 32, 47, 360
- Слот 361
- Событие
 - очередь 384
 - программы, управляемые 149
 - регистрация 384
 - цикл 153
- Состояние
 - готовности 421
 - диаграмма 425
- Список 32, 44, 69, 402
- Спрайт 222
 - лист 405
- Среда разработки 26
- Стек 257
 - метод `pop()` 257
 - метод `push()` 257
- Сцена 419
 - взаимодействие между 444, 453
 - запрос информации у 445
 - ключ 434
 - метод `gun()` 449
 - навигация между 436

- отправка информации всем 446
- отправка информации целевой 446
- приветствия 421
- проверка взаимодействия между 447
- реализация 434
- реализация менеджера 447
- создание 433
- текущая 445
- целевая 445
- Таймер 381
 - демонстрационная программа 382
 - подсчет фреймов 383
 - создание путем вычисления прошедшего времени 386
- Текст
 - визуализация 198
 - вывод 226
 - цвет 198
 - шрифт 198
- Текущий рабочий каталог 156
- Тестирование черного ящика 255
- Типы данных
 - реализация 69
- Точечный синтаксис 170
- Угловые скобки (<>), значение в 47
- Учетная запись
 - несколько объектов в списке 110
 - создание нескольких 42
 - список 47
 - увеличение числа 44
- Физические объекты
 - атрибуты 56
 - представление в виде класса 73
- Флажок
 - добавление 222
- Фрейм 383
- Функция
 - инкапсуляция с помощью 236
 - отличие от метода 64
- Шаблон проектирования 491
 - Factory 500
 - Flyweight 500
 - MVC, модель-представление-контроллер 492
 - Observer 500
 - Visitor 500
- Шары, игра
 - код программы 350
 - менеджер шаров 353
- Щелчок кнопкой мыши
 - обнаружение 158
- Экземпляр 61
 - работа с несколькими 81
- Экран компьютера 143

Удивить своих клиентов, бизнес-партнеров, сделать памятный подарок сотрудникам и рассказать о своей компании читателям бизнес-литературы? Приглашаем стать партнерами выпуска актуальных и популярных книг. О вашей компании узнает наиболее активная аудитория.

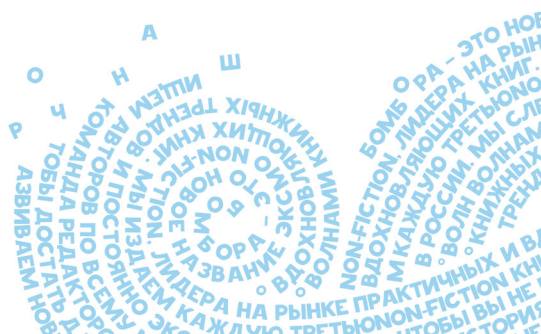
- Специальный тираж уже существующих книг с логотипом вашей компании.
- Размещение логотипа на супер-обложке для малых тиражей (от 30 штук).
- Поддержка выхода новинки, которая ранее не была доступна читателям (50 книг в подарок).

- Рекламная полоса о вашей компании внутри книги.
- Вступительное слово в книге от первых лиц компании-партнера.
- Обращение первых лиц на суперобложке.
- Отзыв на обороте обложки вложение информационных материалов о вашей компании (закладки, листовки, мини-буклеты).



Звоните:
+7 495 411 68 59, доб. 2261

Заходите на сайт:
eksmo.ru/b2b



Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Производственно-практическое издание
МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Кальб Ирв

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С ПОМОЩЬЮ PYTHON

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Продюсер *Н. Витько*
Научный редактор *Н. Белявская*
Литературный редактор *А. Воеводенко*
Младший редактор *П. Смирнов*
Художественный редактор *Е. Пуговкина*
Компьютерная верстка *Э. Брегис*
Корректоры *Н. Витько, Л. Крымова*

Страна происхождения: Российская Федерация
Шығарылған елі: Ресей Федерациясы

ООО «Издательство «Эксмо»
123308, Россия, г. Москва, ул. Зорге, д. 1, стр. 1, эт. 20, каб. 2013. Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «Издательство «Эксмо» ЖШҚ
123308, Ресей, Мәскеу қаласы, Зорге көшесі, 1-үйі, 1-құрылыс, 20 қабат, 2013-каб.
Тел.: 8 (495) 411-68-86. Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Tayap belgici: «Эксмо»

Интернет-магазин: www.book24.ru

Интернет-магазин: www.book24.kz

Интернет-дүкен: www.book24.kz

Импортер в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасына импорттаушы «РДЦ-Алматы» ЖШС.

Дистрибутор и представитель по приему претензий на продукцию
в Республике Казахстан: ТОО «РДЦ-Алматы»

Дистрибутор және Қазақстан Республикасында өнімге шағымдар
қабылдау жөніндегі өкіл: «РДЦ-Алматы» ЖШС.

Алматы қ., Домбровский көш., 3 «а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92. E-mail: RDC-Almaty@eksmo.kz

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»:
www.eksmo.ru/certification

Техникалық реттеу туралы РФ заңнамасына сай басылымның сәйкестігін растау
туралы мәліметтерді мына адрес бойынша алуға болады: <http://eksmo.ru/certification/>

Произведено в Российской Федерации
Ресей Федерациясында өндірілген

Сертификаттауға жатпайды

Дата изготовления / Подписано в печать 08.11.2023.
Формат 70х100^{1/16}. Печать офсетная. Усл. печ. л. 41,48.

Тираж экз. Заказ

ISBN 978-5-04-186627-3



9 785041 866273 >

12+

Москва. ООО «Торговый Дом «Эксмо»

Адрес: 123308, г. Москва, ул. Зорге, д.1, строение 1.
Телефон: +7 (495) 411-50-74. **E-mail:** reception@eksmo-sale.ru

По вопросам приобретения книг «Эксмо» зарубежными оптовыми
покупателями обращаться в отдел зарубежных продаж ТД «Эксмо»
E-mail: international@eksmo-sale.ru

*International Sales: International wholesale customers should contact
Foreign Sales Department of Trading House «Eksmo» for their orders.*
international@eksmo-sale.ru

По вопросам заказа книг корпоративным клиентам, в том числе в специальном
оформлении, обращаться по тел.: +7 (495) 411-68-59, доб. 2151.
E-mail: borodkin.da@eksmo.ru

Оптовая торговля бумажно-беловыми
и канцелярскими товарами для школы и офиса «Канц-Эксмо»:
Компания «Канц-Эксмо»: 142702, Московская обл., Ленинский р-н, г. Видное-2,
Белокаменное ш., д. 1, а/я 5. Тел./факс: +7 (495) 745-28-87 (многоканальный).
e-mail: kanc@eksmo-sale.ru, сайт: www.kanc-eksmo.ru

Филиал «Торгового Дома «Эксмо» в Нижнем Новгороде

Адрес: 603094, г. Нижний Новгород, улица Карпинского, д. 29, бизнес-парк «Грин Плаза»
Телефон: +7 (831) 216-15-91 (92, 93, 94). **E-mail:** reception@eksmonn.ru

Филиал ООО «Издательство «Эксмо» в г. Санкт-Петербурге

Адрес: 192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 84, лит. «Е»
Телефон: +7 (812) 365-46-03 / 04. **E-mail:** server@szko.ru

Филиал ООО «Издательство «Эксмо» в г. Екатеринбурге

Адрес: 620024, г. Екатеринбург, ул. Новинская, д. 2щ
Телефон: +7 (343) 272-72-01 (02/03/04/05/06/08)

Филиал ООО «Издательство «Эксмо» в г. Самаре

Адрес: 443052, г. Самара, пр-т Кирова, д. 75/1, лит. «Е»
Телефон: +7 (846) 207-55-50. **E-mail:** RDC-samara@mail.ru

Филиал ООО «Издательство «Эксмо» в г. Ростове-на-Дону

Адрес: 344023, г. Ростов-на-Дону, ул. Страны Советов, 44А
Телефон: +7(863) 303-62-10. **E-mail:** info@rnd.eksmo.ru

Филиал ООО «Издательство «Эксмо» в г. Новосибирске

Адрес: 630015, г. Новосибирск, Комбинатский пер., д. 3
Телефон: +7(383) 289-91-42. **E-mail:** eksmo-nsk@yandex.ru

Обособленное подразделение в г. Хабаровске

Фактический адрес: 680000, г. Хабаровск, ул. Фрунзе, 22, оф. 703
Почтовый адрес: 680020, г. Хабаровск, А/Я 1006
Телефон: (4212) 910-120, 910-211. **E-mail:** eksmo-khv@mail.ru

Республика Беларусь: ООО «ЭКМО АСТ Си энд Си»

Центр оптово-розничных продаж Cash&Carry в г. Минск
Адрес: 220014, Республика Беларусь, г. Минск, проспект Жукова, 44, пом. 1-17, ТЦ «Outleto»
Телефон: +375 17 251-40-23; +375 44 581-81-92
Режим работы: с 10.00 до 22.00. **E-mail:** exmoast@yandex.by

Казахстан: «РДЦ Алматы»

Адрес: 050039, г. Алматы, ул. Домбровского, 3А
Телефон: +7 (727) 251-58-12, 251-59-90 (91,92,99). **E-mail:** RDC-Almaty@eksmo.kz

**Полный ассортимент продукции ООО «Издательство «Эксмо» можно приобрести в книжных
магазинах «Читай-город» и заказать в интернет-магазине: www.chitai-gorod.ru.**

Телефон единой справочной службы: 8 (800) 444-8-444. Звонок по России бесплатный.

Интернет-магазин ООО «Издательство «Эксмо»

www.eksmo.ru

Розничная продажа книг с доставкой по всему миру.
Тел.: +7 (495) 745-89-14. **E-mail: imarket@eksmo-sale.ru**



В электронном виде книги издательства вы можете
купить на www.litres.ru

ЛитРес:
один клик до книг



Хочешь стать
автором «Эксмо»?



eksmo.ru

Официальный
интернет-магазин
издательства «Эксмо»



БОМБОРА – лидер на рынке полезных и вдохновляющих книг.
Мы любим книги и создаем их, чтобы вы могли творить, открывать
мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bombora.ru [bomborabooks](https://t.me/bomborabooks) [bombora](https://www.instagram.com/bombora)



ВСЕ, ЧТО НУЖНО, ЧТОБЫ ОСВОИТЬ ООП С ПОМОЩЬЮ PYTHON

Объектно-ориентированное программирование (ООП) — это метод, основанный на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования, что позволяет по-другому думать о вычислительных задачах и решать их с возможностью многократного использования.

«Объектно-ориентированное программирование с помощью Python» предназначено для программистов среднего уровня и представляет собой практическое руководство, которое глубоко изучает основные принципы ООП и показывает, как использовать инкапсуляцию, полиморфизм и наследование для написания игр и приложений с использованием Python. Книга начинается с рассказа о ключевых проблемах, присущих процедурному программированию, затем вы познакомитесь с основами создания классов и объектов в Python.

Затем вы научитесь создавать графические интерфейсы с помощью `pygame`, благодаря чему вы сможете писать интерактивные игры и приложения с виджетами графического пользовательского интерфейса (GUI), анимацией, различными сценами и многообразной игровой логикой.

В итоге у вас получится полнофункциональная видеоигра, включающая в себя многие методы ООП и элементы графического пользовательского интерфейса, описанные в книге.

ВЫ УЗНАЕТЕ:

- Как создавать несколько объектов и управлять ими с помощью объекта диспетчера объектов.
- Как использовать инкапсуляцию, чтобы скрыть внутренние детали объектов от клиентского кода.
- Как использовать полиморфизм для определения одного интерфейса и реализации его в нескольких классах.
- Как применять наследование для использования существующего кода.

«Объектно-ориентированное программирование с помощью Python» — это визуальное, интуитивно понятное руководство, позволяющее полностью понять, как работает ООП и как с его помощью сделать свой код более удобным в сопровождении, читаемым и эффективным, не жертвуя при этом функциональностью.


ОБ АВТОРЕ

Ирв Кальб — профессор UCSC Silicon Valley Extension и Университета Кремниевой долины, где он преподает вводные курсы и курсы объектно-ориентированного программирования на Python. Кроме того, он автор книги «Научитесь программировать на Python 3: пошаговое руководство по программированию».

ISBN 978-5-04-186627-3



9 785041 866273 >

 **БОМБОРА**
ИЗДАТЕЛЬСТВО

БОМБОРА — лидер на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

 bombora.ru  bomborabooks  bombora

