

ЭРИК МЭТИЗ

ИЗУЧАЕМ PYTHON

ПРОГРАММИРОВАНИЕ ИГР,
ВИЗУАЛИЗАЦИЯ ДАННЫХ, ВЕБ-ПРИЛОЖЕНИЯ

3-Е ИЗДАНИЕ, ДОПОЛНЕННОЕ И ПЕРЕРАБОТАННОЕ

ПРОДАНО
СВЫШЕ
1 500 000
ЭКЗЕМПЛЯРОВ!



PYTHON CRASH COURSE

3RD EDITION

**A Hands-On, Project-Based
Introduction to Programming**

by Eric Matthes



**no starch
press**

San Francisco

ИЗУЧАЕМ PYTHON

3-Е ИЗДАНИЕ

дополненное и переработанное

**Программирование игр,
визуализация данных, веб-приложения**

Эрик Мэттиз



Санкт-Петербург • Москва • Минск

2025

ББК 32.973.2-018.1
УДК 004.43
М97

Мэттиз Эрик

М97 Изучаем Python: программирование игр, визуализация данных, веб-приложения. 3-е изд., доп. и перераб. — СПб.: Питер, 2025. — 560 с.: ил. — (Серия «Библиотека программиста»).
ISBN 978-5-4461-4112-8

«Изучаем Python» — это самое популярное в мире руководство по языку Python. Вы сможете не только максимально быстро его освоить, но и научитесь писать программы, устранять ошибки и создавать работающие приложения.

В первой части книги вы познакомитесь с основными концепциями программирования, такими как переменные, списки, классы и циклы, а простые упражнения приучат вас к шаблону чистого кода. Вы узнаете, как делать программы интерактивными и как протестировать код, прежде чем добавлять в проект. Во второй части вы примените новые знания на практике и создадите три проекта: аркадную игру в стиле Space Invaders, визуализацию данных с удобными библиотеками Python и простое веб-приложение, которое можно быстро развернуть онлайн.

Книга была переработана и дополнена, чтобы соответствовать последним практикам программирования на Python: приемы редактирования в VS Code, применение модуля pathlib для работы с файлами, тестирование с помощью pytest, а также Matplotlib, Plotty и Django.

Если вы подумываете «А не заняться ли мне программированием?», то эта книга — идеальный старт. Не нужно больше ждать! Погнали!

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

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с No Starch Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1593279288 англ.
ISBN 978-5-4461-4112-8

© 2023 by Eric Matthes
© Перевод на русский язык ООО «Прогресс книга», 2024
© Издание на русском языке, оформление ООО «Прогресс книга», 2024
© Серия «Библиотека программиста», 2024

Краткое содержание

Отзывы о книге.....	23
Об авторе	26
Предисловие к третьему изданию	27
Благодарности	29
Введение.....	31

ЧАСТЬ I. ОСНОВЫ

Глава 1. Первые шаги.....	37
Глава 2. Переменные и простые типы данных	49
Глава 3. Списки	67
Глава 4. Работа со списками	82
Глава 5. Оператор if.....	106
Глава 6. Словари	126
Глава 7. Ввод данных и циклы while	149
Глава 8. Функции.....	165
Глава 9. Классы.....	193
Глава 10. Файлы и исключения	220
Глава 11. Тестирование кода	246

ЧАСТЬ II. ПРОЕКТЫ

Глава 12. Атакующий корабль	265
Глава 13. Осторожно, пришельцы!	295
Глава 14. Игровой счет	319
Глава 15. Генерирование данных	344
Глава 16. Загрузка данных.....	376
Глава 17. Работа с API	404

Глава 18. Знакомство с Django.....	423
Глава 19. Учетные записи пользователей.....	455
Глава 20. Оформление и развертывание приложения.....	487

ПРИЛОЖЕНИЯ

Приложение А. Установка Python и диагностика проблем.....	522
Приложение Б. Редакторы кода и IDE.....	527
Приложение В. Помощь и поддержка	536
Приложение Г. Управление версиями с помощью Git.....	541
Приложение Д. Устранение неполадок при развертывании	551

Оглавление

Отзывы о книге.....	23
Об авторе	26
О научном редакторе.....	26
Предисловие к третьему изданию	27
Благодарности	29
Введение.....	31
Для кого написана эта книга.....	31
Чему вы научитесь	32
Онлайн-ресурсы	33
Почему именно Python.....	34
От издательства.....	34

ЧАСТЬ I. ОСНОВЫ

Глава 1. Первые шаги	37
Подготовка среды программирования.....	37
Версии Python	37
Выполнение фрагментов кода Python.....	37
О редакторе VS Code	38
Python в разных операционных системах.....	39
Python в системе Windows.....	39
Python в системе macOS	41
Python в системе Linux.....	43
Запуск программы Hello World.....	44
Установка расширения Python для VS Code	44
Запуск файла hello_world.py	44
Поиск и устранение ошибок.....	45
Запуск программ Python из терминала.....	46
В Windows	47
В macOS и Linux	47
Резюме	48

Глава 2. Переменные и простые типы данных	49
Что происходит при запуске <code>hello_world.py</code>	49
Переменные	50
Выбор имен и использование переменных	50
Предотвращение ошибок в именах при использовании переменных	51
Переменные как метки	52
Строки	53
Изменение регистра в строке с помощью методов	54
Использование переменных в строках	55
Добавление пробельных символов, таких как табуляции и разрывы строк	56
Удаление пробельных символов	56
Удаление префиксов	58
Предотвращение синтаксических ошибок в строках	58
Числа	60
Целые числа	60
Вещественные числа	61
Целые и вещественные числа	62
Символы подчеркивания в числах	62
Множественное присваивание	63
Константы	63
Комментарии	64
Как создаются комментарии	64
Какие комментарии следует писать	64
Дзен Python	65
Резюме	66
Глава 3. Списки	67
Что такое список	67
Обращение к элементам списка	68
Индексы начинаются с 0, а не с 1	68
Использование отдельных элементов из списка	69
Изменение, добавление и удаление элементов	70
Изменение элементов в списке	70
Добавление элементов в список	71
Удаление элементов из списка	72
Упорядочение списка	76
Постоянная сортировка списка с помощью метода <code>sort()</code>	76

Временная сортировка списка с помощью функции <code>sorted()</code>	77
Вывод списка в обратном порядке.....	78
Определение длины списка.....	78
Ошибки индексирования при работе со списками	80
Резюме	81
Глава 4. Работа со списками	82
Перебор всего списка	82
Подробнее о циклах.....	83
Более сложные действия в циклах <code>for</code>	84
Выполнение действий после цикла <code>for</code>	85
Предотвращение ошибок с отступами.....	86
Пропущенный отступ	86
Пропущенные отступы в других строках.....	87
Лишние отступы	87
Лишние отступы после цикла	88
Пропущенное двоеточие	89
Создание числовых списков	90
Функция <code>range()</code>	90
Использование функции <code>range()</code> для создания числового списка.....	91
Простая статистика с числовыми списками.....	92
Генераторы списков	93
Работа с частью списка	94
Нарезка списков.....	94
Перебор содержимого среза.....	96
Копирование списка.....	96
Кортежи	99
Определение кортежа.....	99
Перебор всех значений в кортеже	101
Замена кортежа.....	101
Форматирование кода.....	102
Рекомендации по стилю	102
Отступы	103
Длина строк	103
Пустые строки.....	104
Другие рекомендации	104
Резюме	105

Глава 5. Оператор if	106
Простой пример	106
Проверка условий	107
Проверка равенства	107
Проверка равенства без учета регистра	108
Проверка неравенства	109
Сравнения чисел	109
Проверка нескольких условий	110
Проверка вхождения значений в список	111
Проверка отсутствия значения в списке	112
Логические выражения	112
Использование операторов if	113
Простые операторы if	113
Операторы if-else	114
Цепочки if-elif-else	115
Серии блоков elif	117
Исключение блока else	117
Проверка нескольких условий	118
Использование операторов if со списками	121
Проверка специальных значений	121
Проверка наличия содержимого в списке	122
Множественные списки	123
Оформление операторов if	125
Резюме	125
Глава 6. Словари	126
Простой словарь	126
Работа со словарями	127
Обращение к значениям в словаре	127
Добавление новых пар «ключ — значение»	128
Создание пустого словаря	129
Изменение значений в словаре	130
Удаление пар «ключ — значение»	131
Словарь с однотипными объектами	132
Обращение к значениям методом get()	133
Перебор словаря	135
Перебор всех пар «ключ — значение»	135
Перебор всех ключей в словаре	137

Перебор ключей словаря в определенном порядке	138
Перебор всех значений в словаре	139
Вложение данных	141
Список словарей	141
Список в словаре	144
Вложение словарей	146
Резюме	148
Глава 7. Ввод данных и циклы while	149
Как работает функция input()	149
Содержательные подсказки	150
Использование функции int() для получения числового ввода	151
Оператор деления по модулю	152
Циклы while	153
Как работает цикл while	154
Пользователь решает прервать работу программы	154
Флаги	156
Оператор break и выход из цикла	157
Оператор continue и продолжение цикла	158
Предотвращение заикливания	159
Использование цикла while со списками и словарями	160
Перемещение элементов между списками	161
Удаление всех вхождений конкретного значения из списка	162
Заполнение словаря данными, введенными пользователем	162
Резюме	164
Глава 8. Функции	165
Определение функции	165
Передача данных функции	166
Аргументы и параметры	167
Передача аргументов	167
Позиционные аргументы	168
Именованные аргументы	169
Значения по умолчанию	170
Эквивалентные вызовы функций	171
Предотвращение ошибок в аргументах	172
Возвращаемое значение	173
Возвращение простого значения	174
Необязательные аргументы	174

Возвращение словаря	176
Использование функции в цикле while	177
Передача списка	179
Изменение списка в функции	180
Запрет изменения списка в функции	182
Передача произвольного набора аргументов	183
Позиционные аргументы с произвольными наборами аргументов	184
Использование произвольного набора именованных аргументов	185
Хранение функций в модулях	187
Импортирование модуля целиком	187
Импортирование конкретных функций из модуля	188
Назначение псевдонима для функции	188
Назначение псевдонима для модуля	189
Импортирование всех функций модуля	189
Форматирование функций	190
Резюме	192
Глава 9. Классы	193
Создание и использование класса	194
Создание класса Dog	194
Метод <code>__init__()</code>	195
Создание экземпляра класса	196
Работа с классами и экземплярами	198
Класс Car	198
Назначение атрибуту значения по умолчанию	199
Изменение значений атрибутов	200
Наследование	203
Метод <code>__init__()</code> класса-потомка	204
Определение атрибутов и методов класса-потомка	205
Переопределение методов класса-родителя	206
Экземпляры как атрибуты	207
Моделирование объектов реального мира	209
Импортирование классов	210
Импортирование одного класса	211
Хранение нескольких классов в модуле	212
Импортирование нескольких классов из модуля	213

Импортирование модуля целиком	214
Импортирование всех классов из модуля	214
Импортирование модуля в модуль	215
Использование псевдонимов	216
Выработка рабочего процесса	216
Стандартная библиотека Python	217
Оформление классов	218
Резюме	219
Глава 10. Файлы и исключения	220
Чтение из файла	220
Чтение всего файла	221
Относительные и абсолютные пути к файлам	223
Построчное считывание	224
Работа с содержимым файла	224
Большие файлы: миллион цифр	226
Проверка даты дня рождения	226
Запись в файл	228
Запись одной строки	228
Запись нескольких строк	228
Исключения	230
Обработка исключения ZeroDivisionError	230
Блоки try-excerpt	230
Использование исключений для предотвращения аварийного завершения программы	231
Блок else	232
Обработка исключения FileNotFoundError	233
Анализ текста	234
Работа с несколькими файлами	235
Ошибки без уведомления пользователя	237
О каких ошибках нужно сообщать	237
Сохранение данных	239
Функции json.dumps() и json.loads()	239
Сохранение и чтение пользовательских данных	240
Рефакторинг	242
Резюме	245

Глава 11. Тестирование кода	246
Установка pytest с помощью pip	246
Обновление pip	247
Установка pytest	248
Тестирование функции.....	248
Модульные тесты и тестовые сценарии	249
Прохождение теста	250
Выполнение тестирования.....	251
Сбой теста	252
Реакция на сбойный тест.....	253
Добавление новых тестов.....	254
Тестирование класса.....	255
Разные методы утверждений.....	256
Класс для тестирования	256
Тестирование класса AnonymousSurvey.....	258
Фикстуры	260
Резюме	262

ЧАСТЬ II. ПРОЕКТЫ

Программирование игры на языке Python.....	264
Визуализация данных	264
Веб-приложения	264
Глава 12. Атакующий корабль	265
Планирование проекта	266
Установка Pygame.....	266
Создание проекта игры.....	266
Создание окна Pygame и обработка ввода	267
Управление частотой кадров	268
Задание фонового цвета	269
Создание класса Settings	270
Добавление изображения корабля	271
Создание класса Ship	272
Вывод корабля на экран	274
Рефакторинг: методы <code>_check_events()</code> и <code>_update_screen()</code>	275
Метод <code>_check_events()</code>	276
Метод <code>_update_screen()</code>	276

Управление кораблем	277
Обработка нажатия клавиши	277
Непрерывное перемещение	278
Перемещение влево и вправо	280
Управление скоростью корабля	281
Ограничение перемещений корабля	283
Рефакторинг метода <code>_check_events()</code>	283
Нажатие клавиши Q для завершения	284
Запуск игры в полноэкранном режиме	284
Обобщим	285
Файл <code>alien_invasion.py</code>	285
Файл <code>settings.py</code>	286
Файл <code>ship.py</code>	286
Стрельба	287
Добавление настроек снарядов	287
Создание класса <code>Bullet</code>	287
Группировка снарядов	289
Обработка выстрелов	290
Удаление выпущенных снарядов	291
Ограничение количества снарядов	292
Создание метода <code>_update_bullets()</code>	293
Резюме	294
Глава 13. Осторожно, пришельцы!	295
Анализ проекта	295
Создание первого пришельца	296
Создание класса <code>Alien</code>	297
Создание экземпляра <code>Alien</code>	297
Создание флота	299
Создание ряда пришельцев	299
Рефакторинг функции <code>_create_fleet()</code>	301
Добавление рядов	302
Перемещение флота	304
Перемещение пришельцев вправо	304
Создание настроек для направления флота	305
Проверка достижения края	306
Снижение флота и смена направления	307

Уничтожение пришельцев	308
Выявление коллизий.....	308
Создание увеличенных снарядов для тестирования	309
Восстановление флота	310
Ускорение снарядов.....	311
Рефакторинг метода <code>_update_bullets()</code>	311
Завершение игры.....	312
Обнаружение коллизий между пришельцами и кораблем	312
Обработка столкновений с кораблем	313
Достижение нижнего края экрана	316
Конец игры.....	317
Определение исполняемых частей игры	317
Резюме	318
Глава 14. Игровой счет	319
Добавление кнопки Play.....	319
Создание класса <code>Button</code>	320
Вывод кнопки на экран.....	321
Запуск игры.....	322
Сброс игры.....	323
Блокировка кнопки запуска игры.....	324
Скрытие указателя мыши	324
Повышение сложности.....	325
Изменение настроек скорости	326
Сброс скорости.....	327
Подсчет очков	328
Вывод счета	328
Создание экземпляра <code>Scoreboard</code>	330
Обновление счета при уничтожении пришельцев	331
Сброс счета	332
Начисление очков за все попадания.....	332
Увеличение награды за пришельцев	333
Округление счета	334
Рекорды	335
Отображение уровня	337
Отображение количества кораблей	340
Резюме	343

Глава 15. Генерирование данных	344
Установка Matplotlib.....	345
Создание простого графика	345
Изменение типа надписей и толщины графика	347
Корректировка графика	348
Встроенные стили	349
Нанесение и оформление отдельных точек с помощью функции <code>scatter()</code>	350
Вывод серии точек с помощью функции <code>scatter()</code>	352
Автоматическое вычисление данных	352
Настройка меток на осях	354
Определение пользовательских цветов	354
Цветовые карты	354
Автоматическое сохранение диаграмм	356
Случайное блуждание	356
Создание класса <code>RandomWalk</code>	356
Выбор направления	357
Вывод случайного блуждания	358
Генерирование нескольких случайных блужданий	359
Форматирование блужданий	360
Моделирование бросков кубиков с помощью <code>Plotly</code>	365
Установка <code>Plotly</code>	365
Создание класса <code>Die</code>	366
Бросок кубика	366
Анализ результатов	367
Создание гистограммы	368
Настройка диаграммы	369
Бросок двух кубиков	370
Дальнейшая настройка диаграммы	372
Броски кубиков с разным количеством граней	372
Сохранение диаграммы в файл	374
Резюме	375
Глава 16. Загрузка данных	376
Формат CSV	376
Разбор заголовка файлов CSV	377
Вывод заголовков и их позиций	378
Извлечение и чтение данных	379
Нанесение данных на диаграмму	379

Модуль <code>datetime</code>	380
Представление дат на диаграмме	382
Расширение временного диапазона	383
Добавление в диаграмму второго набора данных	384
Цветовое выделение частей диаграммы	385
Проверка ошибок	386
Скачивание собственных данных	389
Создание карт с глобальными наборами данных: формат GeoJSON	391
Скачивание данных о землетрясениях	391
Знакомство с форматом GeoJSON	392
Создание списка всех землетрясений	394
Извлечение магнитуд	395
Извлечение данных о местоположении	395
Создание карты мира	396
Представление магнитуд землетрясений	397
Настройка цвета точек на карте	399
Другие цветовые шкалы	400
Добавление подсказки	400
Резюме	403
Глава 17. Работа с API	404
Использование API веб-приложений	404
Git и GitHub	404
Запрос данных с помощью вызовов API	405
Установка пакета <code>requests</code>	406
Обработка ответа API	406
Работа со словарем ответа	407
Сводка самых популярных репозиториях	410
Проверка ограничений частоты обращений API	411
Визуализация репозиториях с помощью Plotly	412
Форматирование диаграммы	413
Добавление подсказок	415
Добавление ссылок в диаграмму	416
Настройка цвета маркеров	417
Дополнительная информация о Plotly и GitHub API	418
API Hacker News	418
Резюме	422

Глава 18. Знакомство с Django	423
Подготовка к созданию проекта.....	423
Написание спецификации.....	424
Создание виртуальной среды.....	424
Активация виртуальной среды.....	425
Установка Django.....	425
Создание проекта в Django.....	426
Создание базы данных.....	427
Просмотр проекта.....	427
Начало работы над приложением.....	429
Определение моделей.....	430
Активизация моделей.....	431
Административный сайт Django.....	432
Определение модели Entry.....	435
Миграция модели Entry.....	436
Регистрация Entry на административном сайте.....	437
Интерактивная оболочка Django.....	438
Создание страниц: главная страница «Журнала обучения».....	440
Связывание URL.....	440
Написание представления.....	442
Написание шаблона.....	443
Создание других страниц.....	445
Наследование шаблонов.....	445
Страница со списком тем.....	447
Страницы отдельных тем.....	450
Резюме.....	454
Глава 19. Учетные записи пользователей	455
Введение данных пользователями.....	455
Добавление новых тем.....	456
Добавление новых записей.....	460
Редактирование записей.....	464
Создание учетных записей пользователей.....	468
Приложение accounts.....	468
Страница входа.....	469
Выход из учетной записи.....	472
Страница регистрации.....	473

Предоставление пользователям доступа к своим данным	476
Ограничение доступа с помощью @login_required	477
Связывание данных с конкретными пользователями	479
Ограничение доступа к темам	482
Защита тем пользователя	483
Защита страницы edit_entry	484
Связывание новых тем с текущим пользователем	484
Резюме	485
Глава 20. Оформление и развертывание приложения	487
Оформление «Журнала обучения»	487
Приложение django-bootstrap5	488
Использование Bootstrap для оформления «Журнала обучения»	488
Изменение файла base.html	489
Оформление главной страницы с помощью табло	495
Оформление страницы входа	497
Оформление страницы со списком тем	498
Оформление записей на странице темы	499
Развертывание «Журнала обучения»	501
Создание учетной записи Platform.sh	501
Установка инструментария Platform.sh CLI	502
Установка пакета platformshconfig	502
Создание файла requirements.txt	502
Дополнительные требования к развертыванию	503
Добавление файлов конфигурации	504
Изменение файла settings.py для Platform.sh	507
Использование Git для управления файлами проекта	508
Создание проекта на Platform.sh	511
Загрузка в Platform.sh	512
Просмотр проекта в реальном времени	513
Доработка развернутого приложения	513
Создание специализированных страниц ошибок	516
Текущая разработка	518
Удаление проекта с Platform.sh	519
Резюме	520

ПРИЛОЖЕНИЯ

Приложение А. Установка Python и диагностика проблем	522
Python в Windows	522
Выполнение команды ru вместо python	522
Переустановка Python	523
Python в macOS	523
Непреднамеренная установка версии Apple	523
Python в системе Linux	524
Использование стандартной установки Python	524
Установка последней версии Python	524
Проверка установленной версии Python	525
Ключевые слова и встроенные функции Python	525
Ключевые слова Python	526
Встроенные функции Python	526
 Приложение Б. Редакторы кода и IDE	 527
Эффективная работа в программе VS Code	528
Настройка программы VS Code	528
Горячие клавиши в VS Code	531
Другие редакторы кода и IDE	533
IDLE	533
Geany	533
Sublime Text	533
Emacs и Vim	534
PyCharm	534
Jupyter Notebook	534
 Приложение В. Помощь и поддержка	 536
Первые шаги	536
Попробуйте заново	537
Сделайте перерыв	537
Обратитесь к дополнительным материалам этой книги	537
Поиск в Интернете	538
Stack Overflow	538
Официальная документация Python	539

Официальная документация библиотек	539
Форум r/learnpython	539
Сообщения в блогах	539
Discord	539
Slack	540
Приложение Г. Управление версиями с помощью Git	541
Установка Git	541
Настройка Git	542
Создание проекта	542
Игнорирование файлов	542
Инициализация репозитория	543
Проверка статуса	543
Добавление файлов в репозиторий	544
Фиксация изменений	544
Просмотр журнала	545
Последующая фиксация	545
Откат изменений	546
Извлечение предыдущих фиксаций	548
Удаление репозитория	549
Приложение Д. Устранение неполадок при развертывании	551
Основы развертывания приложений	551
Устранение распространенных неполадок	552
Следуйте инструкциям на экране	552
Читайте журнал событий	554
Устранение неполадок, специфичных для ОС	555
Развертывание из Windows	556
Развертывание из macOS	557
Развертывание из Linux	558
Другие подходы к развертыванию	558

Отзывы о книге

Интересно наблюдать за тем, как No Starch Press создает будущую классику, которая по праву может занять место рядом с более традиционными книгами по программированию. Эта книга — одна из них.

Грег Лейден (Greg Laden), ScienceBlogs

В книге рассматриваются довольно сложные проекты; материал излагается в последовательной, логичной и приятной манере, которая привлекает внимание читателя к теме.

Full Circle Magazine

Хорошая подача материала с доступными объяснениями фрагментов кода. Книга помогает читателю двигаться вперед шаг за шагом, представляя все более сложный код и объясняя все происходящее.

FlickThrough Reviews

Процесс изучения Python с этой книгой оставил чрезвычайно хорошее впечатление! Отличный вариант, если вы только начинаете изучать Python.

Mikke Goes Coding

Внутри вы найдете все перечисленное на обложке, причем изложенное лучшим способом... В книге представлено множество полезных упражнений, а также три сложных, но увлекательных проекта.

RealPython.com

Задающее быстрый темп, но охватывающее все темы руководство по программированию на Python. У вас в руках еще одна превосходная книга, которая дополнит вашу библиотеку и поможет наконец-то освоить Python.

TutorialEdge.net

Отличный вариант для начинающих без опыта программирования. Если вы ищете проверенное, несложное введение в этот очень глубокий язык, то рекомендуем эту книгу.

WhatPixel.com

Книга содержит буквально все, что вам нужно знать о Python, и даже чуточку больше.

FireBearStudio.com

Хотя книга и посвящена основам программирования именно на Python, она также учит навыкам чистого программирования, которые применимы к большинству других языков.

Great Lakes Geek

*Моему отцу, который никогда не жалел
времени, чтобы ответить на мои вопросы
по программированию, и Эверу, который только
начинает задавать мне свои вопросы.*

Об авторе

Эрик Мэттиз (Eric Matthes) в течение 25 лет работал учителем физики и математики в средней школе и преподавал вводные курсы по Python всякий раз, когда мог найти способ внести их в учебный план. В настоящее время Эрик — профессиональный писатель и программист, а также участник ряда проектов с открытым исходным кодом. Его проекты преследуют самые разные цели: от помощи в прогнозировании оползней в горных районах до упрощения процесса развертывания проектов Django. В свободное время Эрик занимается альпинизмом и проводит время с семьей.

О научном редакторе

Кеннет Лав (Kenneth Love) живет на Тихоокеанском Северо-Западе со своей семьей и кошками. Он опытный программист на Python, участник различных проектов с открытым исходным кодом, преподаватель и докладчик на конференциях.

Предисловие к третьему изданию

Предыдущие два издания книги получили множество положительных отзывов. Тираж составил более миллиона экземпляров, включая переводы на более чем десять языков. Я получал письма и сообщения как от десятилетних читателей, так и от пенсионеров, желающих освоить программирование для собственного удовольствия. Материал книги используется для обучения в школах и колледжах. Студенты, которым приходится иметь дело с более сложными учебниками, используют мою книгу как дополнительный источник для своих занятий; по их мнению, книга является хорошим подспорьем в учебе. Разные люди пользуются ею для повышения квалификации и работы над параллельными проектами. Короче говоря, книга используется для самых разнообразных целей — как я и надеялся.

Я был очень рад тому, что мне представилась возможность написать следующее издание. Python — устоявшийся язык, но продолжает развиваться, как и любой другой. При переработке материала книги я старался сделать его более доступным и компактным. Прочитав ее, вы получите все знания, которые позволят вам начать работу над собственными проектами, а также заложите прочный фундамент для дальнейшего обучения. Я обновил часть разделов и представил новые, более простые средства решения некоторых задач на языке Python. Кроме того, я доработал разделы, в которых те или иные аспекты языка были представлены недостаточно точно. Все проекты были полностью обновлены; в них задействуются только популярные библиотеки, имеющие качественное сопровождение, которыми вы можете пользоваться при создании собственных проектов.

Вот краткое перечисление изменений, внесенных в третье издание.

- В главе 1 описывается редактор VS Code, популярный среди начинающих программистов и профессионалов, полностью поддерживаемый всеми операционными системами.
- В главе 2 описываются новые методы `removeprefix()` и `removesuffix()`, которые пригодятся при работе с файлами и URL. В этой главе также представлен улучшенный функционал системы обработки ошибок Python: теперь она выводит гораздо больше конкретной информации, помогающей устранить неполадки в коде в случае сбоя.
- В главе 10 для работы с файлами используется модуль `pathlib`. Так реализуется гораздо более простой подход к чтению и записи файлов.

- В главе 11 для разработки автоматизированных тестов создаваемого кода мы воспользуемся библиотекой `pytest`. Она стала стандартным инструментом для написания тестов на Python. Ее интерфейс достаточно удобен для начинающих, а если вы продолжите карьеру программиста на Python, то будете использовать ее и в профессиональной среде.
- В проекте «Инопланетное вторжение» в главах 12–14 добавлена настройка, которая позволяет управлять частотой кадров и обеспечивать стабильную работу игры в разных операционных системах. Для создания флота пришельцев выбран более простой подход, а общая организация проекта существенно улучшена.
- Проекты по визуализации данных в главах 15–17 используют самые последние возможности Matplotlib и Plotly. В работе с Matplotlib описаны обновленные настройки стилизации. В проект случайного блуждания внесено небольшое улучшение, благодаря которому повышена точность вывода графиков, поэтому вы увидите больше закономерностей, когда будете создавать новое блуждание. Во всех проектах с Plotly теперь используется модуль Plotly Express, позволяющий генерировать первичные визуализации с помощью лишь нескольких строк кода. Вы сможете быстро просмотреть множество визуализаций и выбрать конкретный вид графика, а затем сосредоточиться на доработке его отдельных элементов.
- Проект «Журнал обучения», рассматриваемый в главах 18–20, создается с применением последних версий Django и Bootstrap. Некоторые компоненты проекта переименованы в целях оптимизации его структуры. Теперь проект развернут на Platform.sh — современном хостинге для проектов Django. Конфигурационные файлы YAML позволяют тонко настроить процесс развертывания. Такой подход профессиональные программисты используют при развертывании современных проектов Django.
- Приложение А полностью обновлено и содержит рекомендации по установке Python во всех распространенных операционных системах. В приложении Б приводятся подробные инструкции по настройке VS Code, а также представлен краткий обзор популярных современных редакторов кода и IDE. Приложение В содержит ссылки на наиболее популярные справочные онлайн-ресурсы. В приложении Г по-прежнему представлен вводный мини-курс по использованию системы управления версиями Git. Приложение Д написано специально для этого издания. Даже при наличии прекрасных руководств по развертыванию программ многое может пойти не так. В этом приложении вы найдете подробное руководство по устранению неполадок, которое пригодится, если процесс развертывания не удастся с первой попытки.

Спасибо за то, что читаете эту книгу! Если у вас появятся вопросы или вы захотите поделиться своим мнением, не стесняйтесь и пишите мне в соцсетях.

Благодарности

Эта книга никогда не появилась бы на свет без великолепных, чрезвычайно профессиональных сотрудников издательства No Starch Press. Билл Поллок (Bill Pollock) предложил мне написать вводную часть книги, и я глубоко признателен ему за это предложение. Лиз Чедвик (Liz Chadwick) работала над всеми тремя изданиями, и книга стала, несомненно, лучше благодаря ее непосредственному участию. Идеи Евы Морроу (Eva Morrow) освежили и улучшили это издание. Кроме того, я ценю советы Дага Макнейра (Doug McNair) по использованию правильной грамматики без излишней формальности. Дженнифер Кеплер (Jennifer Kepler) руководила предпечатной подготовкой, превращая мои многочисленные файлы в безупречный макет.

Со многими сотрудниками No Starch Press, помогавшими добиться успеха с этой книгой, мне не довелось общаться. В издательстве трудится фантастическая команда маркетологов, которые не просто продают книги, а заботятся о том, чтобы читатели находили литературу, которая потенциально поможет достичь их целей. В издательстве также есть профессиональный отдел по работе с иностранными правами. Благодаря стараниям этой команды книга была переведена на многие языки и добралась до читателей по всему миру. Всем этим людям, с которыми я не работал напрямую, спасибо за то, что помогли книге найти свою аудиторию.

Я хотел бы отдельно поблагодарить Кеннета Лава, научного редактора всех трех изданий книги. Я познакомился с ним на конференции PyCon, и с тех пор его энтузиазм по отношению к языку и сообществу Python служит постоянным источником профессионального вдохновения. Кеннет, как всегда, не ограничился простой проверкой фактов и редактировал книгу с целью помочь начинающим программистам освоить язык Python и программирование в целом. Он также обратил внимание на моменты, давшие хороший результат в предыдущих изданиях, но которые можно было бы улучшить, если бы имелась возможность полностью их переписать. Таким образом, все найденные вами неточности остаются только на моей совести.

Я также хотел бы выразить признательность всем читателям, которые поделились своим опытом работы с книгой. Изучая основы программирования, можно

изменить свой взгляд на мир, и иногда это глубоко влияет на людей. Слышать такие истории очень трогательно, и я благодарен всем, кто открыто делится своим опытом.

Я хочу поблагодарить своего отца, который познакомил меня с программированием, когда я был еще ребенком, и не побоялся, что я сломаю его технику. Спасибо моей жене Эрин за поддержку и помощь во время работы над книгой и моему сыну Эверу, любознательность которого постоянно вдохновляет меня.

Введение



У каждого программиста своя история о том, как он написал первую программу. Я начал изучать программирование еще в детстве, когда мой отец работал в Digital Equipment Corporation, одной из ведущих компаний современной компьютерной эры. Я написал свою первую программу на компьютере, который мой отец собрал в подвале нашего дома из различных компонентов. Компьютер представлял собой материнскую плату, подключенную к клавиатуре без корпуса, а в качестве монитора использовалась простейшая электронно-лучевая трубка. Моей первой программой стала игра по отгадыванию чисел, которая выглядела примерно так:

Я загадал число! Попробуйте отгадать мое число: **25**
Слишком мало! Следующая попытка: **50**
Слишком много! Следующая попытка: **42**
Верно! Хотите сыграть снова? (да/нет) **нет**
Спасибо за игру!

Никогда не забуду, как это было здорово: моя семья играла в написанную мной игру, и все работало точно так, как я задумал.

Очень приятно создавать нечто предназначенное для конкретной цели и успешно решающее свою задачу. Программы, которые я пишу сейчас, намного серьезнее моих детских попыток, но чувство удовлетворения, которое я получаю, создавая работающую программу, остается практически тем же.

Для кого написана эта книга

Цель книги — помочь читателю как можно быстрее освоить Python, чтобы он начал писать работающие программы (игры, визуализации данных и веб-приложения), и одновременно заложить основу в области программирования, которая будет полезна для дальнейших изысканий. Книга написана для людей любого возраста, которые прежде никогда не программировали на Python или вообще никогда не программировали. Если вы хотите быстро изучить азы программирования,

чтобы сосредоточиться на интересных проектах, а также любите проверять свое понимание новых концепций, решая сложные задачи, — эта книга для вас. Она также прекрасно подходит для преподавателей, желающих предложить своим ученикам вводный курс программирования на основе проектов. Если вы студент, которому хочется иметь более доступное руководство по Python, чем предложенный вам учебник, то данная книга упростит ваше обучение. Она же поможет освоить программирование, если вы хотите сменить профессию. Эта книга хорошо зарекомендовала себя среди широкого круга читателей, имеющих самые разные цели.

Чему вы научитесь

Цель книги — сделать вас хорошим программистом вообще и хорошим программистом на Python в частности. Процесс обучения будет эффективным, и вы приобретете много полезных привычек, осваивая общие концепции программирования. Перевернув последнюю страницу книги, вы будете готовы к знакомству с более серьезными возможностями Python, а изучение вашего следующего языка программирования тоже упростится.

В части I будут представлены базовые концепции программирования, которые необходимо знать для написания программ на Python. Эти концепции ничем не отличаются от тех, которые рассматриваются в начале изучения почти любого языка программирования. Вы познакомитесь с разными видами данных и способами их хранения в своих программах. Будете создавать коллекции данных, такие как списки и словари, и работать с ними. Научитесь использовать циклы `while` и операторы `if` для проверки определенных условий и выполнения тех или иных разделов кода в зависимости от того, истинно условие или ложно, — это метод, который помогает автоматизировать многие процессы.

Вы научитесь получать входные данные от пользователя, чтобы ваши программы стали интерактивными, и выполнять их до тех пор, пока пользователь остается активным. Кроме того, вы узнаете, как написать функции, которые позволяют многократно использовать части ваших программ, чтобы вы могли написать блок кода для некоего действия один раз, а потом задействовать его многократно по мере необходимости. Затем вы примените эту концепцию к более сложному поведению классов, что позволит даже относительно простым программам реагировать на множество разнообразных ситуаций. Вы научитесь писать программы, корректно обрабатывающие многие типичные ошибки. Освоив каждую из этих базовых концепций, вы напишете ряд все более сложных программ, используя полученные знания. Наконец, вы сделаете первые шаги на пути к программированию среднего уровня: научитесь писать тесты для своего кода, чтобы вы могли продолжать разработку программ, не беспокоясь о возможном внесении багов. Весь материал части I подготовит вас к более сложным и масштабным проектам.

В части II полученные знания помогут вам создать три проекта. Вы можете взяться за любые из этих проектов в том порядке, который лучше подходит для вас. В первом проекте (главы 12–14) будет создан шутер «Инопланетное вторжение» в стиле классического хита Space Invaders, состоящий из многих уровней с нарастающей сложностью. Завершив этот проект, вы значительно продвинетесь в разработке собственных 2D-игр. Даже если вы не планируете стать разработчиком игр, работа над этим проектом — приятный способ связать воедино многое из того, что вы узнаете в части I.

Второй проект (главы 15–17) познакомит вас с визуализацией данных. Чтобы разобраться в огромных объемах доступной информации, специалисты по анализу данных применяют различные средства визуализации. Вы будете работать с разными наборами данных: теми, что генерируются в программах, которые вы скачиваете с онлайн-источников и которые ваши программы загружают автоматически. Завершив этот проект, вы сможете писать программы, обрабатывающие большие наборы данных и создающие визуальное представление самых разных видов информации.

В третьем проекте (главы 18–20) мы напишем небольшое веб-приложение «Журнал обучения». Этот проект позволяет фиксировать информацию, которую вы узнали в ходе изучения конкретной темы. Пользователь приложения сможет вести разные журналы по разным темам, позволять другим создавать учетные записи и вести собственные журналы. Вы также узнаете, как развернуть свой проект в Интернете, чтобы любой желающий мог получить к нему онлайн-доступ из любой точки мира.

Онлайн-ресурсы

Я публикую множество дополнительных материалов к книге на сайте https://ehmatthes.github.io/pcc_3e. В них содержится следующая информация.

- **Инструкции по настройке.** Онлайн-инструкции идентичны инструкциям, приведенным в книге, но содержат активные ссылки, с помощью которых можно переходить к различным фазам настройки. Если у вас возникнут какие-либо проблемы с настройкой, то обращайтесь к этому ресурсу.
- **Обновления.** Python, как и все языки, постоянно развивается. Я отслеживаю наборы обновлений, поэтому, если что-то не работает — обратитесь к этому ресурсу и проверьте, не было ли каких-либо изменений в инструкциях.
- **Решения к упражнениям.** Не жалейте времени на самостоятельное решение задач из разделов «Упражнения». Но если вы оказались в тупике и не знаете, что делать, ответы к большинству упражнений доступны на сайте.
- **Шпаргалки.** Полный набор шпаргалок с краткой информацией по основным концепциям также доступен для скачивания на сайте.

Почему именно Python

Каждый год я задумываюсь над тем, продолжать ли мне работать на Python или же перейти на другой язык — вероятно, более новый в мире программирования. И все же я продолжаю использовать Python по многим причинам. Этот язык невероятно эффективен: программы, написанные на нем, делают больше при меньшем объеме кода, чем требуется во многих других языках. Вдобавок синтаксис Python позволяет писать «чистый» код. Такой код легче читать и отлаживать, а также расширять и развивать по сравнению с другими языками.

Python используется для разных целей: создания игр и веб-приложений, решения бизнес-задач и разработки внутренних инструментов для разных интересных проектов. Кроме того, язык широко применяется в научной области для теоретических исследований и решения прикладных задач.

Впрочем, одной из самых важных причин использования этого языка для меня остается сообщество Python, состоящее из невероятно разных и благожелательных людей. Сообщество играет исключительно важную роль для программистов, поскольку программирование — не индивидуальное занятие. Многим из нас, даже самым опытным специалистам, приходится обращаться за советом к коллегам, которые уже решали похожие задачи. Доброжелательное сообщество, пронизанное связями между людьми, помогает решать задачи, и сообщество Python готово прийти на помощь людям, для которых это первый язык программирования или которые стали изучать Python, имея опыт работы на других языках.

Python — замечательный язык, так давайте браться за дело!

От издательства

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

Мы будем рады узнать ваше мнение!

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

ЧАСТЬ I

ОСНОВЫ

В части I представлены базовые концепции, необходимые для написания программ на языке Python. Многие из этих концепций встречаются во всех языках программирования, поэтому пригодятся вам на протяжении всей профессиональной карьеры.

В главе 1 вы установите Python на свой компьютер и запустите первую программу, которая выводит на экран сообщение `Hello world!`.

В главе 2 вы научитесь хранить информацию в переменных, работать с текстовыми и числовыми данными.

В главах 3 и 4 вы познакомитесь со списками. Они позволяют хранить любой объем информации в одной переменной, что повышает эффективность работы с данными. Вы сможете работать с сотнями, тысячами и даже миллионами значений всего в нескольких строках кода.

В главе 5 будут представлены операторы `if`. С их помощью вы сможете написать код, который делает что-то одно, если некое условие истинно, и что-то другое, если оно ложно.

В главе 6 будет показано, как использовать словари Python, связывающие разные виды информации. Как и списки, словари могут содержать столько информации, сколько вы захотите в них поместить.

В главе 7 вы научитесь получать данные от пользователей, чтобы ваши программы стали интерактивными. Там же описаны циклы `while`, многократно выполняющие блоки кода, пока некое условие остается истинным.

В главе 8 вы займетесь написанием функций — именованных блоков кода, которые решают конкретную задачу и запускаются по мере необходимости.

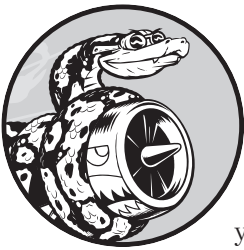
В главе 9 представлены классы, предназначенные для моделирования объектов реального мира: собак, кошек, людей, машин, ракет и т. д.

Благодаря главе 10 вы научитесь работать с файлами и обрабатывать ошибки, что поможет защитить ваши программы от неожиданного сбоя. Вы сохраните данные перед закрытием программы и снова прочитаете их, когда она запустится повторно. Вы узнаете об исключениях Python, которые позволяют предвидеть ошибки и организовать их корректную обработку в программах.

В главе 11 вы научитесь писать тесты для своего кода. Они проверяют, что ваша программа работает так, как было задумано. В результате вы сможете дорабатывать свои программы, не беспокоясь о возможном внесении новых багов. Тестирование — один из первых навыков, отличающий новичка от программиста среднего уровня.

1

Первые шаги



В этой главе вы запустите свою первую программу на языке Python — `hello_world.py`. Сначала проверите, есть ли на вашем компьютере последняя версия Python; если это не так, то установите ее. Вдобавок вы установите редактор кода для работы с вашими программами. Текстовые редакторы распознают код Python и выделяют синтаксические конструкции во время работы, упрощая понимание структуры кода разработчиком.

Подготовка среды программирования

Python слегка отличается в разных операционных системах, поэтому вы должны учитывать некоторые аспекты. В этой главе мы проверим, правильно ли установлен Python в вашей системе.

Версии Python

Каждый язык программирования развивается по мере появления новых идей и технологий, и разработчики Python постоянно расширяют возможности и гибкость языка. На момент написания книги новейшей была версия 3.11, но все представленные здесь программы должны нормально работать в версии 3.9 или более поздней. Этот раздел поможет вам понять, установлен ли Python в вашей системе и нужно ли установить более новую версию. В приложении А содержится дополнительная информация по установке новейшей версии Python во всех основных операционных системах.

Выполнение фрагментов кода Python

Вы можете запустить интерпретатор Python в терминальном окне, что позволяет опробовать фрагменты кода Python, не сохраняя и не запуская всю программу.

В книге встречаются фрагменты кода следующего вида:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
```

Три символа угловых скобок (`>>>`), называемых *приглашением Python* (Python prompt), означают, что вам следует использовать окно терминала. Жирным шрифтом выделен текст, который вам нужно ввести, а затем выполнить, нажав клавишу **Enter**. Большинство примеров в книге представляют собой небольшие независимые программы, которые вы будете запускать не из терминала, а из редактора, поскольку именно в нем вы будете писать большую часть кода. Но некоторые базовые концепции в целях более эффективной демонстрации будут показаны с помощью серии фрагментов кода, запускаемых в терминальном сеансе Python. Таким образом, три угловые скобки в листинге означают, что вы просматриваете код и выходные данные терминального сеанса. Чуть позже мы попробуем написать код в интерпретаторе для вашей системы.

Кроме того, редактор кода будет использоваться для создания простой программы **Hello World!**, что стало основой обучения программированию. В мире программирования издавна принято начинать освоение нового языка с программы, выводящей на экран сообщение **Hello world!**, — считается, что это принесет удачу. Даже такая простая программа выполняет вполне конкретную функцию. Если она корректно запускается в вашей системе, то и любая программа, которую вы напишете на Python, тоже должна работать нормально.

О редакторе VS Code

VS Code — мощный, профессиональный редактор кода, бесплатный и удобный для начинающих. Он отлично подходит для создания как простых, так и сложных проектов, поэтому, освоив его в процессе изучения Python, вы сможете использовать его и в более крупных проектах. VS Code можно установить на все современные операционные системы, и он поддерживает большинство языков программирования, в том числе Python.

В приложении Б приведена информация о других редакторах. Если вам они интересны, то на данном этапе вы можете бегло просмотреть его. Если же вы хотите быстро перейти к программированию, то на первых порах работайте с VS Code, а возможность использования других редакторов рассмотрите позже, когда получите некий опыт программирования. В этой главе я опишу процесс установки VS Code для вашей операционной системы.

ПРИМЕЧАНИЕ

Если у вас уже установлен редактор кода и вы знаете, как его настроить для запуска программ на Python, то можете использовать его.

Python в разных операционных системах

Python — кросс-платформенный язык программирования; это означает, что он работает во всех основных операционных системах. Написанная вами программа на языке Python должна выполняться на любом современном компьютере, на котором установлен Python. Однако способы настройки Python для разных операционных систем слегка различаются.

В этом разделе вы узнаете, как подготовить Python к работе в вашей системе. Сначала вы проверите, имеется ли новая версия Python в вашей системе, и если нет — установите ее. Затем вы установите VS Code. Процедура состоит всего из двух шагов, различающихся в разных операционных системах.

Затем вы запустите программу `hello_world.py` и устраните любые неполадки. Этот процесс будет описан для всех операционных систем, так что в итоге в вашем распоряжении появится простая и удобная среда программирования на Python.

Python в системе Windows

Операционная система Windows по умолчанию не содержит интерпретатора Python. Скорее всего, вам придется установить Python, а затем и редактор кода VS Code.

Установка Python

Для начала проверьте, установлен ли Python в вашей системе. Откройте окно командной строки: введите `command` в меню Пуск и щелкните на приложении **Command Prompt** (Командная строка). В окне терминала введите команду `python` в нижнем регистре. Если на экране появится приглашение `>>>`, значит, Python установлен в вашей системе. Если же вы видите сообщение об ошибке, в котором говорится, что команда `python` не опознана системой, или открывается окно магазина Microsoft Store, то Python не установлен. Закройте магазин, если он открылся; рекомендуется загрузить официальный дистрибутив, а не версию, распространяемую корпорацией Microsoft.

Если Python не установлен в вашей системе или вы видите версию ниже 3.9, то скачайте программу установки Python для Windows. Откройте страницу <https://python.org/> и наведите указатель мыши на ссылку **Downloads** (Загрузки). Появится кнопка для скачивания новейшей версии Python. Нажмите кнопку, которая запускает автоматическое скачивание правильного установочного пакета для вашей системы. По завершении скачивания запустите программу установки. Не забудьте установить флажок **Add Python to PATH** (Добавить Python в PATH) — это упростит правильную настройку системы. На рис. 1.1 изображено окно мастера установки.



Рис. 1.1. Не забудьте установить флажок Add Python to PATH
(Добавить Python в PATH)

Запуск Python в терминальном сеансе

Откройте окно командной строки и введите команду `python` в нижнем регистре. Если на экране появится приглашение Python (`>>>`), значит, система Windows обнаружила установленную версию Python:

```
C:\> python
Python 3.x.x (main, Jun . . . , 13:29:14) [MSC v.1932 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

ПРИМЕЧАНИЕ

Если вы не увидите этот (или похожий) вывод, то обращайтесь к более подробным инструкциям по установке, приведенным в приложении А.

Введите в сеансе Python следующую строку:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Вы должны увидеть вывод `Hello Python interpreter!`. Каждый раз, когда вы захотите выполнить фрагмент кода Python, откройте окно командной строки и запустите терминальный сеанс Python. Чтобы закрыть сеанс, нажмите `Ctrl+Z` и `Enter` или введите команду `exit()`.

Установка VS Code

Программу установки для VS Code можно скачать по адресу <https://code.visualstudio.com>. Нажмите кнопку **Download for Windows** (Скачать для Windows) и запустите программу установки. Пропустите следующие разделы о macOS и Linux и выполните действия, описанные в разделе «Запуск программы Hello World» далее в этой главе.

Python в системе macOS

В последних версиях macOS интерпретатора Python, как правило, нет, поэтому вам нужно будет установить его, если вы еще не сделали этого. В этом подразделе мы установим новейшую версию Python, а затем редактор кода VS Code и убедимся в том, что он настроен правильно.

ПРИМЕЧАНИЕ

В старых версиях операционной системы macOS предустановлен интерпретатор Python 2. Это устаревшая версия, которую не рекомендуется использовать.

Проверка наличия Python 3 в системе

Откройте терминальное окно (команда меню **Applications ▶ Utilities ▶ Terminal** (Приложения ▶ Утилиты ▶ Терминал)). Можно также нажать **⌘+Пробел**, ввести `terminal` и нажать **Enter**. Чтобы проверить, установлена ли у вас последняя версия Python, введите команду `python3`. Скорее всего, вы увидите сообщение с предложением установки *консольных инструментов разработчика* (command line developer tools). Их рекомендуется устанавливать только после Python, поэтому закройте сообщение, если оно появится.

Если появилось сообщение, что у вас установлен Python 3.9 или более поздняя версия, то вы можете пропустить следующий пункт и перейти к пункту «Запуск Python в терминальном сеансе» далее в этой главе. Если вы видите версию, предшествующую Python 3.9, то следуйте инструкциям, описанным ниже, чтобы установить последнюю версию.

Обратите внимание: если вы выполняете примеры из этой книги в системе macOS, то вам нужно вводить команду не `python`, а `python3`, чтобы использовать версию Python 3. В большинстве систем macOS команда `python` либо ссылается на устаревшую версию Python, которая должна использоваться только внутренними системными инструментами, либо выдает сообщение об ошибке.

Установка новейшей версии Python

Программа установки Python доступна на сайте <https://python.org/>. Наведите указатель мыши на ссылку **Downloads** (Загрузки). Появится кнопка для скачивания новейшей версии Python. Нажмите кнопку, которая запускает автоматическое

скачивание правильного установочного пакета для вашей системы. По завершении скачивания запустите программу установки.

Когда установка будет завершена, должно открыться окно программы Finder. Дважды щелкните на файле `Install Certificates.command`. Запустив его, вы сможете легко устанавливать дополнительные библиотеки, которые понадобятся вам в реальных проектах, в том числе тех, которые описаны в части II.

Запуск Python в терминальном сеансе

Для выполнения фрагментов кода Python можно открыть терминальное окно и ввести команду `python3`:

```
$ python3
Python 3.x.x (v3.11.0:eb0004c271, Jun . . . , 10:03:01)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Эта команда запускает сеанс Python. Вы должны увидеть приглашение Python (`>>>`), означающее, что система macOS обнаружила версию Python, которую вы только что установили.

Введите в терминальном сеансе следующую строку:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Вы должны увидеть сообщение `Hello Python interpreter!`, которое выводится прямо в текущем терминальном окне. Закрыть интерпретатор Python можно, нажав сочетание клавиш `Ctrl+D` или введя команду `exit()`.

ПРИМЕЧАНИЕ

В новых версиях операционной системы macOS в приглашении в терминале вместо знака доллара (\$) вы увидите знак процента (%).

Установка VS Code

Чтобы установить редактор VS Code, необходимо скачать программу установки, доступную на <https://code.visualstudio.com>. Щелкните на ссылке **Download** (Скачать) и скачайте программу установки для macOS. После того как программа установки будет загружена, откройте окно Finder и перейдите в папку **Downloads** (Загрузки). Перетащите программу установки Visual Studio Code в папку **Applications** (Приложения), а затем дважды щелкните на программе, чтобы запустить ее.

Пропустите следующий подраздел о Python в Linux и выполните действия, описанные в разделе «Запуск программы Hello World» далее в этой главе.

Python в системе Linux

Системы семейства Linux ориентированы на программистов, поэтому Python уже установлен на большинстве компьютеров Linux. Люди, которые занимаются разработкой и сопровождением Linux, ожидают, что в какой-то момент вы займетесь программированием, и всячески способствуют этому. Так что если вы захотите сделать это, вам почти ничего не придется устанавливать, а количество необходимых настроек будет минимальным.

Проверка версии Python

Откройте терминальное окно, запустив приложение **Terminal** (Терминал) в вашей системе (в Ubuntu нажмите **Ctrl+Alt+T**). Чтобы проверить, какая версия Python установлена в вашей системе, введите команду `python3` (со строчной буквы `p`). Если Python в системе есть, эта команда запустит интерпретатор Python. На экране появится информация о том, какая версия Python установлена, и приглашение `>>>`, после которого можно вводить команды Python:

```
$ python3
Python 3.10.4 (main, Apr . . . , 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Этот вывод сообщает, что Python 3.10.4 в настоящее время является версией Python по умолчанию, установленной на данном компьютере. Нажмите **Ctrl+D** или введите `exit()`, чтобы выйти из приглашения Python и вернуться к приглашению терминала. Каждый раз, когда в книге встречается команда `python`, вводите вместо нее команду `python3`.

Для запуска кода из книги необходима версия не ниже Python 3.9. Если в системе установлена более ранняя версия или вам требуется обновить текущую версию более новой, то обратитесь к приложению А.

Запуск Python в терминальном сеансе

Для выполнения фрагментов кода Python можно открыть терминальное окно и ввести команду `python3`, как мы поступили при проверке версии. Сделайте то же самое и, когда Python будет запущен, введите в терминальном сеансе следующую строку:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Сообщение выводится прямо в текущем терминальном окне. Помните, что закрыть интерпретатор Python можно, нажав **Ctrl+D** или введя команду `exit()`.

Установка VS Code

В системе Ubuntu Linux редактор VS Code устанавливается из Ubuntu Software Center. Щелкните на значке **Ubuntu Software** (Программное обеспечение Ubuntu) в меню и найдите вариант **vscode**. Щелкните на приложении **Visual Studio Code** (иногда просто **code**), а затем нажмите кнопку **Install** (Установить). После установки найдите в операционной системе программу VS Code и запустите ее.

Запуск программы Hello World

После того как в вашей системе будут установлены последние версии Python и VS Code, все почти готово к запуску вашей первой программы Python, написанной в редакторе кода. Но перед этим необходимо установить расширение Python для VS Code.

Установка расширения Python для VS Code

VS Code работает со многими языками программирования; чтобы программисту на Python использовать возможности редактора максимально эффективно, нужно установить расширение Python. Оно добавляет поддержку разработки, редактирования и запуска программ на языке Python.

Чтобы установить расширение Python, щелкните на значке **Manage** (Управление) в виде шестеренки в левом нижнем углу приложения VS Code. В появившемся меню выберите пункт **Extensions** (Расширения). Введите слово **python** в строке поиска и выберите расширение **Python**. (Если видите несколько расширений с таким названием, выберите разработанное корпорацией Microsoft.) Нажмите кнопку **Install** (Установить) и установите все дополнительные инструменты, необходимые системе для завершения установки. Если появится сообщение о том, что требуется установить Python, но вы уже сделали это, то можете проигнорировать его.

ПРИМЕЧАНИЕ

Если вы используете macOS и во всплывающем окне вам предлагается установить консольные инструменты разработчика, нажмите кнопку **Install** (Установить). Может появиться сообщение о том, что установка займет длительное время, но не обращайтесь внимания, она должна занять 10–20 минут при высокоскоростном доступе к Интернету.

Запуск файла `hello_world.py`

Прежде чем писать первую программу, создайте на рабочем столе в своей системе папку **python_work** для своих проектов. В именах файлов и папок лучше использовать строчные буквы и символы подчеркивания вместо пробелов в соответствии с правилами именования, принятыми в Python. Вы можете создать эту папку не на рабочем столе, а в любом другом каталоге, но вам будет проще выполнять последующие шаги, если вы сохраните ее прямо на рабочем столе.

Запустите программу VS Code и закройте вкладку **Get Started** (Начало работы), если она открылась. Создайте новый файл, выбрав команду меню **File ▶ New File** (Файл ▶ Создать файл) или нажмите **Ctrl+N** (**⌘+N** в macOS). Сохраните созданный файл `Python hello_world.py` в папке `python_work`. Расширение `.py` сообщает VS Code, что код в файле написан на языке Python; эта информация помогает редактору запустить программу и правильно выделить цветом элементы синтаксиса.

После того как файл будет сохранен, введите в редакторе кода следующую строку:

```
hello_world.py  
print("Hello Python world!")
```

Программу можно запустить с помощью команды меню **Run ▶ Run Without Debugging** (Запуск ▶ Запуск без отладки) или сочетания клавиш **Ctrl+F5** (**⌘+B** в macOS). В нижней части окна VS Code должно отображаться терминальное окно со следующим текстом:

```
Hello Python world!
```

Скорее всего, вы увидите дополнительный вывод, информирующий о версии интерпретатора Python, который использовался для запуска вашей программы. Если вы хотите упростить отображаемую информацию, чтобы видеть только вывод вашей программы, то обратитесь к приложению Б. В нем же вы найдете полезные советы о том, как использовать VS Code более эффективно.

Если вы не увидели этот вывод, то проверьте каждый символ во введенной строке. Может, вы случайно набрали `print` с прописной буквы? Пропустили одну или обе кавычки или круглые скобки? В языках программирования используется предельно конкретный синтаксис, и при малейшем его нарушении произойдет ошибка. Если программа так и не заработала, то прочитайте следующий раздел.

Поиск и устранение ошибок

Если вам так и не удалось запустить программу `hello_world.py`, то, возможно, помогут следующие полезные советы (кстати, они могут пригодиться для решения любых проблем в программах).

- Если программа содержит серьезную ошибку, то Python выводит данные *трассировки* (traceback). Python анализирует содержимое файла, пытается выявить проблему и составить отчет. Проверьте данные трассировки. Возможно, они подскажут, что именно мешает выполнению программы.
- Отойдите от компьютера, отдохните и попробуйте снова. Помните, что синтаксис в программировании очень важен; даже пропущенное двоеточие, неверно расположенная кавычка или непарная скобка могут помешать нормальной работе программы. Перечитайте соответствующие части главы, еще раз проанализируйте ваш код и попробуйте найти ошибку.

- Начните заново. Вероятно, ничего переустанавливать не придется, но попробуйте удалить файл `hello_world.py` и создать его с нуля.
- Попросите кого-нибудь повторить действия, описанные в этой главе, на вашем (или на другом) компьютере. Внимательно понаблюдайте за происходящим. Возможно, вы упустили какую-нибудь мелочь, которую заметят другие.
- Ознакомьтесь с дополнительными инструкциями по установке, приведенными в приложении А; некоторые из указанных там подробностей могут помочь вам решить проблему.
- Найдите специалиста, хорошо знающего Python, и попросите его помочь вам. Вполне может оказаться, что такой специалист есть среди ваших знакомых.
- Инструкции по настройке среды программирования, приведенные в этой главе, также доступны по адресу https://ehmatthes.github.io/pcc_3e. Онлайн-версия этих инструкций может быть более удобной для вас, поскольку вы можете просто вырезать и вставлять код и нажимать ссылки, ведущие на нужные вам ресурсы.
- Обратитесь за помощью в Интернет. В приложении В перечислены некоторые ресурсы (форумы, чаты и т. д.), где вы сможете проконсультироваться у людей, уже сталкивавшихся с вашей проблемой.

Не стесняйтесь обращаться к опытным программистам. Любой программист в какой-то момент своей жизни заходил в тупик, и многие программисты охотно помогут вам правильно настроить вашу систему. Если вы сможете четко объяснить, что хотите сделать, какие действия уже пытались выполнить и какие результаты получили, то, скорее всего, кто-нибудь вам поможет. Как упоминалось во введении, сообщество Python доброжелательно относится к новичкам.

Python должен нормально работать на любом современном компьютере. На первых порах проблемы могут быть весьма неприятными, но с ними стоит разобраться. Когда программа `hello_world.py` заработает, вы сможете приступить к изучению Python, а ваша работа станет намного более интересной и принесет больше удовольствия.

Запуск программ Python из терминала

Большинство программ, написанных вами в редакторе кода, будут запускаться прямо из него. Тем не менее иногда бывает полезно запускать программы из терминала — например, если вы хотите просто выполнить готовую программу, не открывая ее для редактирования.

Это можно сделать в любой системе с установленным Python; необходимо лишь знать путь к каталогу, в котором хранится файл программы. Приведенные ниже примеры предполагают, что вы сохранили файл `hello_world.py` в папке `python_work` на рабочем столе.

В Windows

Команда `cd` (от *change directory* — «изменить каталог») используется для перемещения по файловой системе в окне командной строки. Команда `dir` (от *directory* — «каталог») выводит список всех файлов в текущем каталоге.

Откройте новое терминальное окно и введите следующие команды для запуска программы `hello_world.py`:

```
C:\> cd Desktop\python_work
C:\Desktop\python_work> dir
hello_world.py
C:\Desktop\python_work> python hello_world.py
Hello Python world!
```

Команда `cd` используется для перехода к папке `python_work`, находящейся в папке `Desktop`. Затем команда `dir` проверяет, что файл `hello_world.py` действительно находится в этой папке. Далее файл запускается командой `python hello_world.py`.

Большинство программ будет нормально запускаться из редактора. Но со временем ваша работа станет более сложной, и, возможно, вы предпочтете запускать некоторые из своих программ из терминала.

В macOS и Linux

Запуск программы Python в терминальном сеансе в системах Linux и macOS осуществляется одинаково. Команда `cd` используется для перемещения по файловой системе в терминальном сеансе. Команда `ls` (от *list* — «список») выводит список всех не скрытых файлов в текущем каталоге.

Откройте новое терминальное окно и введите следующие команды для запуска программы `hello_world.py`:

```
~$ cd Desktop/python_work/
~/Desktop/python_work$ ls
hello_world.py
~/Desktop/python_work$ python3 hello_world.py
Hello Python world!
```

Команда `cd` используется для перехода к папке `python_work`, находящейся в каталоге `Desktop`. Затем команда `ls` проверяет, что файл `hello_world.py` действительно находится в этом каталоге. Далее файл запускается командой `python3 hello_world.py`.

Большинство программ будет нормально запускаться из редактора. Но со временем ваша работа станет более сложной, и, возможно, вы предпочтете запускать некоторые из своих программ из терминала.

УПРАЖНЕНИЯ

Упражнения этой главы в основном направлены на самостоятельный поиск информации. Начиная с главы 2, упражнения будут ориентированы на решение задач, основанных на изложенном материале.

1.1. python.org. Изучите главную страницу Python (<https://python.org/>) и найдите темы, которые вас заинтересуют. Со временем вы лучше узнаете Python, и другие разделы этого сайта покажутся вам более полезными.

1.2. Опечатки в программе Hello World. Откройте только что созданный файл `hello_world.py`. Сделайте где-нибудь намеренную опечатку и снова запустите программу. Удастся ли вам сделать опечатку, которая приводит к ошибке? Поймете ли вы смысл сообщения об ошибке? Удастся ли вам сделать опечатку, которая не приводит к ошибке? Как вы думаете, почему на этот раз выполнение обходится без ошибки?

1.3. Неограниченные возможности. Если бы вы были программистом с неограниченными возможностями, то за какой проект взялись бы? Сейчас вы учитесь программировать. Если вы ясно представляете конечную цель, то сможете немедленно применить новые навыки на практике; попробуйте набросать общие описания программ, над которыми вам хотелось бы поработать. Заведите «блокнот идей», к которому сможете обращаться каждый раз, когда соберетесь начать новый проект. Выделите пару минут и составьте описания трех программ, которые вам хотелось бы создать.

Резюме

В этой главе вы познакомились с языком Python и узнали, как установить его в своей системе. Кроме того, вы установили редактор кода, упрощающий работу над кодом Python. Вы научились выполнять фрагменты кода Python в терминальном сеансе и запустили свою первую настоящую программу `hello_world.py`. Скорее всего, попутно вы кое-что узнали о поиске и исправлении ошибок.

В следующей главе рассматриваются структуры данных, с которыми вы будете работать в программах Python. Кроме того, вы научитесь пользоваться переменными Python.

2

Переменные и простые типы данных



В этой главе представлены разные виды данных, с которыми вы будете работать в своих программах Python. Вы также научитесь использовать переменные для представления данных в своих программах.

Что происходит при запуске `hello_world.py`

Давайте подробнее рассмотрим, что же делает Python при запуске `hello_world.py`. Оказывается, он проделывает достаточно серьезную работу даже для такой простой программы:

`hello_world.py`

```
print("Hello Python world!")
```

При выполнении этого кода выводится следующий текст:

```
Hello Python world!
```

Суффикс `.py` в имени файла `hello_world.py` указывает, что файл является программой Python. Редактор запускает файл в *интерпретаторе Python*, который читает программу и определяет, что означает каждое слово в программе. Например, обнаружив слово `print`, за которым следуют круглые скобки, интерпретатор выводит на экран текст, находящийся внутри них.

Когда вы пишете программу, редактор выделяет цветом разные ее части. Например, он понимает, что `print()` — имя функции, и выделяет это слово одним цветом. А текст `Hello Python world!` не является кодом Python, поэтому выделяется другим цветом. Этот механизм, называемый *цветовым выделением синтаксических элементов* (syntax highlighting), очень поможет вам, когда вы начнете писать собственные программы.

Переменные

Попробуем использовать переменную в программе `hello_world.py`. Добавьте новую строку в начало файла и измените вторую строку:

hello_world.py

```
message = "Hello Python world!"  
print(message)
```

Запустите программу и посмотрите, что получится. Программа выводит уже знакомый вам результат:

```
Hello Python world!
```

Вы добавили в программу *переменную* `message`. В каждой переменной хранится *значение*, то есть связанные с ней данные. В нашем случае значением является текст `"Hello Python world!"`.

Добавление переменной немного усложняет задачу интерпретатора Python. Во время обработки первой строки он связывает текст `"Hello Python world!"` с переменной `message`. А добравшись до второй строки, выводит на экран значение, связанное с именем `message`.

Немного расширим программу `hello_world.py`, чтобы она выводила второе сообщение. Добавьте в `hello_world.py` пустую строку, а после нее еще две строки кода:

```
message = "Hello Python world!"  
print(message)  
  
message = "Hello Python Crash Course world!"  
print(message)
```

Теперь при выполнении `hello_world.py` на экране должны появляться две строки:

```
Hello Python world!  
Hello Python Crash Course world!
```

Вы можете в любой момент изменить значение переменной в своей программе; Python постоянно отслеживает его текущее состояние.

Выбор имен и использование переменных

При работе с переменными в языке Python необходимо соблюдать некоторые правила и рекомендации. Нарушение правил приведет к ошибке; рекомендации всего лишь помогают писать более понятный и удобочитаемый код. Работая с переменными, помните о следующем.

- Имена переменных могут состоять только из букв, цифр и символов подчеркивания. Они могут начинаться с буквы или символа подчеркивания,

но не с цифры. Например, переменной можно присвоить имя `message_1`, но не `1_message`.

- Пробелы в именах переменных запрещены, а для разделения слов в именах переменных используются символы подчеркивания. Например, имя `greeting_message` допустимо, а имя `greeting message` вызовет ошибку.
- Не используйте в качестве имен переменных названия функций и ключевые слова Python; иначе говоря, не используйте слова, которые зарезервированы в Python для конкретной цели, — например, слово `print` (см. раздел «Ключевые слова и встроенные функции Python» приложения А).
- Имена переменных должны быть короткими, но содержательными. Например, имя `name` лучше `n`, имя `student_name` лучше `s_n`, а имя `name_length` лучше `length_of_persons_name`.
- Будьте внимательны при использовании строчной буквы `l` и прописной буквы `O`, поскольку они похожи на цифры `1` и `0`.

Вероятно, вы не сразу научитесь создавать хорошие имена переменных, особенно когда программы станут сложнее и интереснее. Но когда вы начнете писать больше программ и читать код, созданный другими разработчиками, ваши имена переменных станут более содержательными.

ПРИМЕЧАНИЕ

На данном этапе ограничьтесь именами переменных, записанными в нижнем регистре. Использование символов верхнего регистра не приведет к ошибке, но они имеют специальное значение в именах переменных, о котором мы поговорим в других главах.

Предотвращение ошибок в именах при использовании переменных

Каждый программист совершает ошибки, а большинство программистов делают это ежедневно. И хотя даже опытный программист не застрахован от ошибок, он знает, как эффективно реагировать на них. Рассмотрим типичную ошибку, которую вы довольно часто будете совершать на первых порах, и выясним, как ее исправить.

Для начала напишем код с намеренно внесенной ошибкой. Введите следующий фрагмент (неправильно написанное слово `mesage` выделено жирным шрифтом):

```
message = "Hello Python Crash Course reader!"  
print(mesage)
```

Когда в программе происходит ошибка, интерпретатор Python всеми силами старается помочь вам найти ее причину. Если программа не выполняется нормально, интерпретатор предоставляет данные *трассировки* — информацию о том, в каком месте кода находился интерпретатор при возникновении проблемы. Ниже приведен

пример трассировки, которую Python выдает после того, как вы случайно сделали опечатку в имени переменной:

Traceback (most recent call last):

- ```
❶ File "hello_world.py", line 2, in <module>
❷ print(message)
 ^^^^^
❸ NameError: name 'message' is not defined. Did you mean: 'message'?
```

В строке ❶ сообщается, что ошибка произошла в строке 2 файла `hello_world.py`. Интерпретатор выводит номер строки, чтобы вам было проще найти ошибку ❷, и сообщает тип обнаруженной ошибки ❸. В данном случае была обнаружена *ошибка в имени*, и сообщается, что переменная с указанным именем (`message`) не определена. Другими словами, Python не распознает имя переменной. Обычно такие ошибки возникают в том случае, если вы забыли присвоить значение переменной перед ее использованием или ошиблись при вводе имени. Если Python обнаружит имя переменной, похожее на нераспознанное, то появится запрос, не это ли имя вы хотели использовать.

Конечно, в данном примере в имени переменной во второй строке пропущена буква `s`. Интерпретатор Python не проверяет код на наличие опечаток, но следит за тем, чтобы имена переменных записывались одинаково. Например, вот что происходит, если имя `message` будет неправильно записано еще в одном месте кода:

```
message = "Hello Python Crash Course reader!"
print(message)
```

На этот раз программа выполняется успешно!

```
Hello Python Crash Course reader!
```

Имена переменных совпадают, поэтому Python не видит проблему. Языки программирования строги, но орфография их совершенно не волнует. Как следствие, при создании имен переменных и написании кода вам не нужно жестко соблюдать правила орфографии и грамматики английского языка.

Многие ошибки программирования сводятся к простым опечаткам — случайной замене одного символа в одной строке программы. Если вы потратили много времени на поиск одной из таких ошибок, то знайте, что вы не одиноки. Многие опытные и талантливые программисты проводят долгие часы в поисках подобных мелких ошибок. Нечто подобное будет часто происходить и с вами — просто посмейтесь и продолжайте работать.

## Переменные как метки

Переменные часто описывают как «ящики» для хранения значений. Такое сравнение может быть полезным на первых порах работы с переменными, но оно неточно описывает внутреннее представление переменных в Python. Намного правильнее

представлять переменные как метки, которые можно назначать переменным. Можно также сказать, что переменная *содержит ссылку* на некоторое значение.

Вероятно, это различие ни на что не повлияет в ваших первых программах. И все же лучше узнать о нем раньше, чем позже. В какой-то момент вы столкнетесь с неожиданным поведением переменных, и более точное понимание их работы поможет вам разобраться в том, что же происходит в вашем коде.

### ПРИМЕЧАНИЕ

Как лучше всего освоить новые концепции программирования? Попробуйте использовать их в своей программе. Если в ходе работы над упражнением вы зайдете в тупик, то попробуйте на какое-то время заняться чем-нибудь другим. Если это не поможет, перечитайте соответствующую часть этой главы. Если и это не помогло, то следуйте рекомендациям из приложения В.

### УПРАЖНЕНИЯ

Напишите отдельную программу для выполнения каждого из следующих упражнений. Сохраните каждую программу в файле, имя которого подчиняется стандартным правилам Python по использованию строчных букв и символов подчеркивания — например, `simple_message.py` и `simple_messages.py`.

**2.1. Простое сообщение.** Сохраните текстовое сообщение в переменной и выведите его на экран.

**2.2. Простые сообщения.** Сохраните сообщение в переменной и выведите его. Затем замените значение переменной другим сообщением и выведите это новое сообщение.

## Строки

Многие программы определяют и собирают некие данные, а затем делают с ними что-то полезное, поэтому желательно выделить основные разновидности данных. Начнем со строк. На первый взгляд, они достаточно просты, но с ними можно работать разными способами.

*Строка* представляет собой простую последовательность символов. Любая последовательность символов, заключенная в кавычки, в Python считается строкой; при этом строки могут быть заключены как в одиночные, так и в двойные кавычки:

```
"This is a string."
'This is also a string.'
```

Это правило позволяет использовать внутренние кавычки и апострофы в строках:

```
'I told my friend, "Python is my favorite language!'"
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Рассмотрим некоторые типичные операции со строками.

### **Изменение регистра в строке с помощью методов**

Одна из простейших операций, выполняемых со строками, — изменение регистра символов. Взгляните на следующий фрагмент кода и попробуйте определить, что в нем происходит:

#### ***name.py***

```
name = "ada lovelace"
print(name.title())
```

Сохраните файл как `name.py` и запустите его. Вывод программы должен выглядеть так:

```
Ada Lovelace
```

В этом примере в переменной `name` сохраняется строка, состоящая из букв нижнего регистра `"ada lovelace"`. За именем переменной в функции `print()` следует вызов метода `title()`. Метод представляет собой действие, которое Python выполняет с данными. Благодаря точке `(.)` после `name` в конструкции `name.title()` Python получает указание применить метод `title()` к переменной `name`. За именем метода всегда следует пара круглых скобок, поскольку методам для выполнения их работы часто требуется дополнительная информация. Она указывается в скобках. Функции `title()` дополнительная информация не нужна, поэтому в круглых скобках ничего нет.

Метод `title()` выполняет капитализацию начальных букв каждого слова (переводит их в верхний регистр), тогда как все остальные символы выводятся в нижнем. Например, данная возможность может быть полезна, если в вашей программе входные значения `Ada`, `ADA` и `ada` должны рассматриваться как одно и то же имя и все они должны отображаться в виде `Ada`.

Для работы с регистром существуют и другие полезные методы. Так, все символы строки можно преобразовать в верхний или нижний регистр:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

Программа выводит следующий результат:

```
ADA LOVELACE
ada lovelace
```

Метод `lower()` особенно полезен для хранения данных. Нередко программист не может рассчитывать на то, что пользователи введут все данные, точно соблюдая регистр, поэтому строки перед сохранением преобразуются в нижний регистр. Затем, когда потребуется вывести информацию, используется регистр, наиболее подходящий для каждой строки.

## Использование переменных в строках

В некоторых ситуациях требуется использовать значения переменных внутри строки. Представьте, что имя и фамилия хранятся в разных переменных и вы хотите объединить их для вывода полного имени:

### *full\_name.py*

```
first_name = "ada"
last_name = "lovelace"
❶ full_name = f"{first_name} {last_name}"
print(full_name)
```

Чтобы вставить значение переменной в строку, поставьте букву `f` непосредственно перед открывающей кавычкой ❶. Заключите имя (или имена) переменных, которые должны использоваться внутри строки, в фигурные скобки. Python заменит каждую переменную ее значением при выводе строки.

Такие строки называются *f-строками*. Буква *f* происходит от слова *format*, поскольку Python форматирует строку, заменяя имена переменных в фигурных скобках их значениями. Приведенный выше код выводит следующий результат:

```
ada lovelace
```

С помощью *f*-строк можно выполнять много интересных действий. Например, составлять сложные сообщения с информацией, хранящейся в переменных. Рассмотрим пример:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ print(f"Hello, {full_name.title()}!")
```

Полное имя используется для вывода приветственного сообщения ❶, а метод `title()` выполняет капитализацию начальных букв каждого слова (переводит в верхний регистр). Этот фрагмент возвращает простое, хорошо отформатированное сообщение:

```
Hello, Ada Lovelace!
```

Кроме того, *f*-строки можно использовать для составления сообщения, которое затем сохраняется в переменной:

```
first_name = "ada"
last_name = "lovelace"
```

```
full_name = f"{first_name} {last_name}"
❶ message = f"Hello, {full_name.title()}!"
❷ print(message)
```

Этот код тоже выводит сообщение `Hello, Ada Lovelace!`, но сохранение текста сообщения в переменной ❶ существенно упрощает финальный вызов функции `print()` ❷.

## Добавление пробельных символов, таких как табуляции и разрывы строк

В программировании термином «*пробельный символ*» (whitespace) называются такие непечатаемые символы, как пробелы, табуляции и символы конца строки. Пробельные символы структурируют текст, чтобы пользователю было удобнее читать его.

Для добавления в текст табуляции используется комбинация символов `\t`:

```
>>> print("Python")
Python
>>> print("\tPython")
 Python
```

Разрывы строк добавляются с помощью комбинации символов `\n`:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

Табуляции и разрывы строк могут стоять в одной строке. Так, благодаря последовательности `"\n\t"` Python получает указание начать текст с новой строки, в начале которой располагается табуляция. В следующем примере показано, как использовать одну строку кода для создания четырех строк вывода:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
 Python
 C
 JavaScript
```

Разрывы строк и табуляции часто будут встречаться в двух следующих главах, когда программы начнут выдавать много строк вывода, полученных всего из нескольких строк кода.

## Удаление пробельных символов

Лишние пробельные символы могут вызвать путаницу в программах. Для человека строки `'python'` и `'python '` внешне неотличимы, но для программы это совершенно разные строки. Python видит лишний пробел в `'python '` и считает, что он действительно важен, — до тех пор пока вы не сообщите о противоположном.



Обращайте внимание на пробельные символы, поскольку в программах часто приходится сравнивать строки, чтобы проверить их содержимое на совпадение. Типичный пример — проверка имен пользователей при входе на сайт. Лишние пробельные символы могут создавать путаницу и в более простых ситуациях. К счастью, Python позволяет легко удалить лишние пробельные символы из данных, введенных пользователем.

Python может искать лишние пробельные символы у левого и правого краев строки. Чтобы убедиться в том, что у правого края (в конце) строки нет пробельных символов, вызовите метод `rstrip()`:

```
❶ >>> favorite_language = 'python '
❷ >>> favorite_language
'python '
❸ >>> favorite_language.rstrip()
'python'
❹ >>> favorite_language
'python '
```

Значение, хранящееся в переменной `favorite_language` ❶, содержит лишний пробел в конце строки. Выводя это значение в терминальном сеансе Python, вы видите пробел в конце значения ❷. Когда метод `rstrip()` работает с переменной `favorite_language` ❸, этот лишний символ удаляется. Впрочем, удаление временное — если вы снова запросите значение `favorite_language`, то увидите, что строка совпадает с исходной, включая лишний пробельный символ ❹.

Чтобы навсегда удалить пробельный символ из строки, следует записать исправленное значение обратно в переменную:

```
>>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
>>> favorite_language
'python'
```

Сначала пробельные символы удаляются в конце строки, а потом значение записывается в исходную переменную ❶. Операция изменения значения переменной с последующим его сохранением в исходной переменной выполняется в программировании часто. Так, значение переменной может изменяться в ходе выполнения программы или в ответ на действия пользователя.

Пробельные символы также можно удалить у левого края (в начале) строки с помощью метода `lstrip()`, а метод `strip()` удаляет пробельные символы с обоих концов:

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
' python '
❸ >>> favorite_language.lstrip()
'python '
❹ >>> favorite_language.strip()
'python'
```

В этом примере исходное значение содержит пробельные символы в начале и конце ❶. Затем пробельные символы удаляются у правого края ❷, у левого края ❸ и с обоих концов строки ❹. Поэкспериментируйте с функциями удаления пробельных символов, это поможет вам освоить работу со строками. На практике эти функции чаще всего применяются для очистки пользовательского ввода перед его сохранением в программе.

## Удаление префиксов

Помимо пробельных символов, при работе со строками часто требуется удалять и префиксы. В качестве примера рассмотрим URL с префиксом <https://>. Попробуем удалить этот префикс, чтобы работать только с той частью URL, которую пользователи должны ввести в адресной строке. Вот как это сделать:

```
>>> nostarch_url = 'https://nostarch.com'
>>> nostarch_url.removeprefix('https://')
'nostarch.com'
```

Укажите имя переменной, затем точку и метод `removeprefix()`. В круглых скобках укажите префикс, который нужно удалить из исходной строки.

Как и в случае удаления пробельных символов, метод `removeprefix()` не изменяет исходную строку. Если вы хотите сохранить в переменной новое значение с удаленным префиксом, то присвойте его либо исходной, либо новой переменной:

```
>>> simple_url = nostarch_url.removeprefix('https://')
```

Если в адресной строке браузера URL отображается без префикса <https://>, то, вероятно, «за кадром» используется метод, подобный `removeprefix()`.

## Предотвращение синтаксических ошибок в строках

*Синтаксические ошибки* (syntax error) встречаются в программах относительно регулярно. Они возникают тогда, когда Python не распознает часть вашей программы как действительный код. Например, если заключить апостроф в одиночные кавычки, то произойдет ошибка. Это происходит из-за того, что Python интерпретирует все символы от первой одиночной кавычки до апострофа как строку. После этого он пытается интерпретировать остаток текста строки как код Python, что порождает ошибки.

Разберемся, как же правильно использовать одиночные или двойные кавычки. Сохраните следующую программу как файл `apostrophe.py` и запустите ее:

### *apostrophe.py*

```
message = "One of Python's strengths is its diverse community."
print(message)
```

Апостроф находится в строке, заключенной в двойные кавычки, так что у интерпретатора Python не возникает проблем с правильным пониманием следующей строки:

```
One of Python's strengths is its diverse community.
```

Однако при использовании одиночных кавычек Python не сможет определить, где должна заканчиваться строка:

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

Программа выводит следующий результат:

```
File "apostrophe.py", line 1
 message = 'One of Python's strengths is its diverse community.'
 ❶ ^
```

SyntaxError: unterminated string literal (detected at line 1)

Из выходных данных видно, что ошибка происходит сразу же после второй одиночной кавычки ❶. Эта синтаксическая ошибка указывает на то, что интерпретатор не распознает какую-то конструкцию как действительный код Python. Ошибки могут возникать по разным причинам; я буду выделять наиболее распространенные по мере того, как они станут нам встречаться. Вы будете часто сталкиваться с синтаксическими ошибками, учась писать правильный код Python. Кроме того, ошибки этой категории являются наиболее неконкретными, поэтому их очень трудно находить и исправлять. Если вы зайдете в тупик из-за особенно сложной ошибки, то обратитесь к рекомендациям в приложении В.

## ПРИМЕЧАНИЕ

Функция цветового выделения синтаксических элементов ускоряет выявление некоторых синтаксических ошибок прямо во время написания программы. Если вы увидите, что код Python выделяется как обычный текст (или обычный текст выделяется как код Python), то, скорее всего, в вашем файле где-то пропущена кавычка.

## УПРАЖНЕНИЯ

Сохраните код каждого из следующих упражнений в отдельном файле вида `name_cases.py`. Если у вас возникнут проблемы, то сделайте перерыв или обратитесь к рекомендациям в приложении В.

**2.3. Личное сообщение.** Сохраните имя пользователя в переменной и выведите сообщение, предназначенное для конкретного человека. Сообщение должно быть простым, например: «Здравствуйте, Эрик, не хотите ли вы изучить Python сегодня?»

**2.4. Регистр символов в именах.** Сохраните имя пользователя в переменной и выведите его в нижнем регистре, верхнем регистре и с капитализацией начальных букв каждого слова.

**2.5. Знаменитая цитата.** Найдите известное высказывание, которое вам понравилось. Выведите текст цитаты с именем автора. Результат должен выглядеть примерно так (включая кавычки):

*Альберт Эйнштейн однажды сказал: «Тот, кто никогда не совершал ошибок, никогда не пробовал ничего нового».*

**2.6. Знаменитая цитата 2.** Повторите упражнение 2.5, однако на этот раз сохраните имя автора цитаты в переменной `famous_person`. Затем составьте сообщение и сохраните его в новой переменной `message`. Выведите свое сообщение.

**2.7. Удаление пробельных символов.** Сохраните имя пользователя в переменной. Добавьте в начале и конце имени несколько пробельных символов. Проследите за тем, чтобы каждая последовательность символов `"\t"` и `"\n"` встречалась по крайней мере один раз.

Выведите имя, чтобы были видны пробельные символы в начале и конце строки. Затем выведите его снова, используя каждую из функций удаления пробельных символов: `lstrip()`, `rstrip()` и `strip()`.

**2.8. Расширения файлов.** В Python доступен метод `removesuffix()`, функционирующий точно так же, как `removeprefix()`. Присвойте переменной `filename` значение `'python_notes.txt'`. Затем используйте метод `removesuffix()`, чтобы отобразить имя файла без расширения, как это делают некоторые файловые браузеры.

## Числа

Числа очень часто применяются в программировании для ведения счета в играх, представления данных в визуализациях, хранения информации в веб-приложениях и т. д. В Python числовые данные делятся на несколько категорий в соответствии со способом их использования. Для начала посмотрим, как Python работает с целыми числами, поскольку с ними возникает меньше всего проблем.

### Целые числа

В Python с целыми числами можно выполнять операции сложения (+), вычитания (-), умножения (\*) и деления (/).

```
>>> 2 + 3
5
>>> 3 - 2
1
```

```
>>> 2 * 3
6
>>> 3 / 2
1.5
```

В терминальном сеансе Python просто возвращает результат операции. Для представления операции возведения в степень в Python используется сдвоенный знак умножения:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

Кроме того, в Python существует определенный порядок операций, что позволяет использовать несколько операций в одном выражении. Круглые скобки применяются для изменения порядка операций, чтобы выражение могло вычисляться в нужном порядке. Пример:

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

Пробелы в этих примерах не влияют на то, как Python вычисляет выражения; они просто помогают быстрее найти приоритетные операции при чтении кода.

## Вещественные числа

В Python числа, имеющие дробную часть, называются *вещественными* (или числами с плавающей точкой). Этот термин используется в большинстве языков программирования, и точка «плавает» потому, что может появиться в любой позиции числа. Во всех языках программирования механизмы обработки вещественных чисел тщательно прорабатываются, чтобы числа вели себя предсказуемым образом, независимо от позиции точки.

В большинстве случаев вы можете использовать вещественные числа, не особенно задумываясь об их поведении. Просто введите нужные числа, а Python, скорее всего, сделает именно то, что вы от него хотите:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

Однако в некоторых ситуациях вдруг оказывается, что результат содержит неожиданно большое количество разрядов в дробной части:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

Нечто подобное может произойти в любом языке; для беспокойства нет причин. Python пытается подобрать как можно более точное представление результата, что иногда бывает нелегко из-за особенностей внутреннего представления чисел в компьютерах. Пока просто не обращайте внимания на «лишние» разряды; вы узнаете, как поступать в подобных ситуациях, когда эта проблема станет актуальной для вас в проектах части II.

## **Целые и вещественные числа**

При делении двух любых чисел — даже если это целые числа, частным от деления которых является целое число, — вы всегда получаете вещественное число:

```
>>> 4/2
2.0
```

При смешении целого и вещественного числа в любой другой операции вы также получаете вещественное число:

```
>>> 1 + 2.0
3.0
>>> 2 * 3.0
6.0
>>> 3.0 ** 2
9.0
```

Python по умолчанию использует вещественный тип для результата любой операции, в которой задействовано вещественное число, даже если результат является целым числом.

## **Символы подчеркивания в числах**

В записи целых чисел можно группировать цифры, используя символы подчеркивания, чтобы числа лучше читались:

```
>>> universe_age = 14_000_000_000
```

При выводе числа, определяемого с помощью символов подчеркивания, Python выводит только цифры:

```
>>> print(universe_age)
14000000000
```

Python игнорирует символы подчеркивания при хранении таких значений. Даже если цифры не группируются в тройках, это никак не повлияет на значение. С точки зрения Python `1000` ничем не отличается от записи `1_000`, которая эквивалентна `10_00`. Этот вариант записи работает как для целых, так и для вещественных чисел.

### Множественное присваивание

В одной строке программы можно присвоить значения сразу нескольким переменным. Этот синтаксис сократит длину программы и упростит ее чтение; чаще всего он применяется при инициализации наборов чисел.

Например, следующая строка инициализирует переменные `x`, `y` и `z` нулями:

```
>>> x, y, z = 0, 0, 0
```

Имена переменных, как и значения, должны разделяться запятыми. Python присваивает каждое значение переменной в конкретной позиции. Если количество значений соответствует количеству переменных, то Python правильно сопоставит их друг с другом.

### Константы

*Константа* представляет собой переменную, значение которой остается неизменным на протяжении всего срока жизни программы. В Python нет встроенных типов констант, но программисты Python для записи имен переменных, которые должны рассматриваться как константы и оставаться неизменными, используют буквы верхнего регистра:

```
MAX_CONNECTIONS = 5000
```

Если вы собираетесь работать с переменной в коде как с константой, то не забудьте записать ее имя буквами верхнего регистра.

## УПРАЖНЕНИЯ

**2.9. Число 8.** Напишите операции сложения, вычитания, умножения и деления, результатом которых является число 8. Не забудьте заключить операции в функции `print()`, чтобы проверить результат. Вы должны написать четыре строки кода, которые выглядят примерно так:

```
print(5 + 3)
```

Результатом должны быть четыре строки, в каждой из которых выводится число 8.

**2.10. Любимое число.** Сохраните свое любимое число в переменной. Затем с помощью переменной создайте сообщение для вывода этого числа. Выведите данное сообщение.

## Комментарии

Комментарии чрезвычайно полезны в любом языке программирования. До сих пор ваши программы состояли только из кода Python. По мере роста объема и сложности кода в программы следует добавлять *комментарии*, описывающие общий подход к решаемой задаче, — своего рода заметки на понятном языке.

### Как создаются комментарии

В языке Python признаком комментария является символ «решетка» (#). Интерпретатор Python игнорирует все символы, следующие в коде после # до конца строки. Пример:

#### *comment.py*

```
Поздороваться со всеми.
print("Hello Python people!")
```

Python игнорирует первую строку и выполняет вторую.

```
Hello Python people!
```

### Какие комментарии следует писать

Главная задача комментария — объяснить, что должен делать ваш код и как он работает. В разгаре работы над проектом вы понимаете, как работают все его компоненты. Но если вернуться к нему спустя время, то, скорее всего, некоторые подробности будут забыты. Конечно, всегда можно изучить код и разобраться в том, как должны работать его части, но хорошие комментарии, в которых доступно излагаются общие принципы работы кода, сэкономят немало времени.

Если вы хотите стать профессиональным программистом или участвовать в совместной работе с другими программистами, то научитесь писать содержательные комментарии. В наши дни почти все программы разрабатываются коллективно: либо группами работников одной компании, либо группами энтузиастов, совместно работающих над проектом с открытым кодом. Опытные программисты ожидают увидеть комментарии в коде, поэтому лучше привыкайте добавлять содержательные комментарии прямо сейчас. Написание простых, лаконичных комментариев — одна из самых полезных привычек, необходимых начинающему программисту.

Принимая решение о том, нужно ли писать комментарий, спросите себя, пришлось ли вам перебрать несколько вариантов в поисках разумного решения для некой задачи; если ответ положительный, то напишите комментарий по поводу вашего решения. Удалить лишние комментарии позднее намного проще, чем возвращаться и добавлять комментарии в программу. С этого момента я буду использовать комментарии в примерах, чтобы пояснить смысл некоторых частей кода.



## УПРАЖНЕНИЯ

**2.11. Добавление комментариев.** Выберите две программы из написанных вами и добавьте в каждую хотя бы один комментарий. Если вы не найдете, что написать в комментариях, поскольку программы были слишком просты, то укажите свое имя и текущую дату в начале кода. Затем добавьте одно предложение с описанием того, что делает программа.

## Дзен Python

Опытные программисты Python рекомендуют избегать лишних сложностей и по возможности выбирать простые решения там, где это возможно. Философия сообщества Python выражена в очерке Тима Питерса *The Zen of Python* («Дзен Python»). Чтобы получить доступ к этому краткому набору принципов написания хорошего кода Python, введите команду `import this` в интерпретаторе. Я не стану цитировать здесь все принципы, но приведу несколько строк, чтобы вы поняли, почему они важны для вас как начинающего программиста Python.

```
>>> import this
The Zen of Python, by Tim Peters
Красивое лучше, чем уродливое.
```

Программисты Python считают, что код может быть красивым и элегантным. В программировании люди занимаются решением задач. Программисты всегда ценили хорошо спроектированные, эффективные и даже красивые решения. Со временем вы больше узнаете о Python, начнете писать больше кода — и когда-нибудь ваш коллега посмотрит на экран вашего компьютера и скажет: «Ого, какой красивый код!»

Простое лучше, чем сложное.

Если у вас есть выбор между простым и сложным решениями и оба работают, то используйте простое. Код будет более легко сопровождать, а вы и другие разработчики столкнетесь с меньшим количеством проблем при обновлении этого кода в будущем.

Сложное лучше, чем запутанное.

Реальность создает свои сложности; иногда простое решение задачи невозможно. В таком случае используйте самое простое решение, которое работает.

Удобочитаемость имеет значение.

Даже если ваш код сложен, постарайтесь сделать так, чтобы он был читабельным. Работая над проектом, требующим создания сложного кода, постарайтесь написать содержательные комментарии для этого кода.

Должен существовать один — и желательно, только один — очевидный способ сделать это.

Если предложить двум программистам Python решить одну и ту же задачу, то они должны выработать похожие решения. Это не значит, что в программировании нет места для творчества. Наоборот, его предостаточно! Но чаще всего работа программиста заключается в применении небольших стандартных решений для простых ситуаций в контексте большого, более творческого проекта. Внутренняя организация ваших программ должна выглядеть логично с точки зрения других программистов Python.

Сейчас лучше, чем никогда.

Вы можете потратить весь остаток жизни на изучение всех тонкостей Python и программирования в целом, но в таком случае никогда не закончите ни один проект. Не пытайтесь написать идеальный код; напишите код, который работает, а потом решите, стоит ли доработать его для текущего проекта или заняться чем-то другим.

Когда вы перейдете к следующей главе и займетесь изучением более сложных тем, постарайтесь не забывать об этой философии простоты и ясности. Опытные программисты будут с большим уважением относиться к вашему коду и будут рады поделиться своим мнением и сотрудничать с вами в интересных проектах.

## УПРАЖНЕНИЯ

**2.12. Дзен Python.** Введите команду `import this` в терминальном сеансе Python и просмотрите другие принципы.

## Резюме

В этой главе вы научились работать с переменными. Вы узнали, как задавать содержательные имена переменным и исправлять ошибки в именах и синтаксические ошибки в случае их возникновения. Вы разобрались, что такое строки и как вывести их в нижнем/верхнем регистре и с капитализацией начальных букв всех слов. Кроме того, вы узнали о способах аккуратного оформления вывода с применением пробельных символов, а также о том, как удалять лишние элементы из разных частей строки. Вы начали работать с целыми и вещественными числами и узнали о некоторых неожиданностях, встречающихся при работе с числовыми данными. Вы научились писать содержательные комментарии, которые упрощают написание кода для вас и его чтение для других разработчиков. В завершение главы вы познакомились с философией максимальной простоты кода.

В главах 3 и 4 мы поговорим о хранении наборов данных в переменных, называемых *списками*. Вы узнаете, как перебрать содержимое списка и обработать хранящуюся в нем информацию.

# 3

## Списки



В этой и следующей главе вы узнаете, что такое списки и как начать работать с их элементами. Списки позволяют хранить в одном месте взаимосвязанные данные, сколько бы ни было элементов — несколько или миллионы. Работа со списками — одна из самых выдающихся возможностей Python, доступных для начинающего программиста. Операции со списками связывают воедино многие важные концепции в программировании.

### Что такое список

*Список* представляет собой набор элементов, следующих в определенном порядке. Вы можете создать список для хранения букв алфавита, цифр от 0 до 9 или имен всех членов вашей семьи. В список можно поместить любую информацию, причем данные в списке даже не обязаны быть как-то связаны друг с другом. Список обычно содержит более одного элемента, поэтому рекомендуется присваивать спискам имена во множественном числе: `letters`, `digits`, `names` и т. д.

В языке Python список обозначается квадратными скобками (`[]`), а отдельные его элементы разделяются запятыми. Вот простой пример списка, содержащего названия моделей велосипедов:

#### *bicycles.py*

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

Если вы прикажете Python вывести список, то на экране появится перечисление элементов списка в квадратных скобках:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Конечно, вашим пользователям такое представление не подойдет; разберемся, как обратиться к отдельным элементам в списке.

## Обращение к элементам списка

Списки представляют собой упорядоченные наборы данных, поэтому для обращения к любому элементу списка следует сообщить Python позицию (*индекс*) нужного элемента. Чтобы обратиться к элементу, укажите имя списка, за которым следует индекс элемента в квадратных скобках.

Например, название первого велосипеда в списке `bicycles` выводится следующим образом:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])
```

Когда мы запрашиваем один элемент из списка, Python возвращает только этот элемент без квадратных скобок или кавычек:

```
trek
```

Именно такой результат должны увидеть пользователи — чистый, отформатированный вывод.

Кроме того, можно использовать и строковые методы из главы 2 для любого элемента списка. Например, элемент `'trek'` можно отформатировать с помощью метода `title()`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

Этот пример выдает такой же результат, как и предыдущий, только название `'Trek'` выводится с прописной буквы.

## Индексы начинаются с 0, а не с 1

Python считает, что первый элемент списка находится в позиции 0, а не в позиции 1. Этот принцип встречается в большинстве языков программирования и объясняется особенностями низкоуровневой реализации операций со списками. Если вы получаете неожиданные результаты, то определите, не допустили ли простую ошибку «смещения на 1».

Второму элементу списка соответствует индекс 1. В этой простой схеме вы можете получить любой элемент, вычитая единицу из его позиции в списке. Например, чтобы обратиться к четвертому элементу списка, следует запросить элемент с индексом 3.

В следующем примере выводятся названия велосипедов с индексами 1 и 3:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

При этом выводится второй и четвертый элементы списка:

```
cannondale
specialized
```

Кроме того, в Python существует специальный синтаксис для обращения к последнему элементу списка. Если запросить элемент с индексом  $-1$ , то Python всегда возвращает последний элемент в списке:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

Фрагмент вернет значение `'specialized'`. Этот синтаксис весьма полезен, поскольку при работе со списками часто требуется обратиться к последним элементам, не зная точного количества элементов в списке. Синтаксис распространяется и на другие отрицательные значения индексов. По индексу  $-2$  возвращается второй элемент от конца списка, по индексу  $-3$  — третий элемент от конца и т. д.

## Использование отдельных элементов из списка

Отдельные значения из списка используются так же, как и любые другие переменные. Например, с помощью *f*-строк вы можете создать сообщение, содержащее значение из списка.

Попробуем извлечь название первого велосипеда из списка и составить сообщение, используя это значение.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
message = f"My first bicycle was a {bicycles[0].title()}."

print(message)
```

Программа создает сообщение, содержащее значение из `bicycles[0]`, и сохраняет его в переменной `message`. Так получается простое предложение с упоминанием первого велосипеда из списка:

```
My first bicycle was a Trek.
```

## УПРАЖНЕНИЯ

Попробуйте написать несколько коротких программ, чтобы получить предварительное представление о списках Python. Возможно, для упражнений каждой главы стоит создать отдельную папку, чтобы избежать неразберихи.

**3.1. Имена.** Сохраните имена нескольких своих друзей в списке `names`. Выведите имя каждого друга, обратившись к каждому элементу списка (по одному за раз).

**3.2. Сообщения.** Начните со списка, использованного в упражнении 3.1, но вместо вывода имени каждого человека выведите сообщение. Основной текст всех сообщений должен быть одинаковым, но каждое сообщение должно содержать имя адресата.

**3.3. Собственный список.** Выберите ваш любимый вид транспорта (например, мотоциклы или машины) и создайте список с примерами. Используйте список для вывода утверждений об элементах, например: «Я хотел бы купить мотоцикл Honda».

## Изменение, добавление и удаление элементов

Чаще всего вы будете создавать *динамические* списки; это означает, что во время выполнения программы в таком списке будут добавляться и удаляться элементы. Например, вы можете создать игру, в которой игрок должен стрелять по кораблям космических захватчиков. Исходный набор кораблей сохраняется в списке; каждый раз, когда вы сбиваете корабль, он удаляется из списка. Каждый раз, когда на экране появляется новый враг, он добавляется в список. Длина списка кораблей будет уменьшаться и увеличиваться по ходу игры.

### Изменение элементов в списке

Синтаксис изменения элемента напоминает синтаксис обращения к элементу списка. Чтобы изменить элемент, укажите имя списка и индекс изменяемого элемента в квадратных скобках; далее задайте новое значение, которое должно быть присвоено элементу.

Допустим, имеется список мотоциклов, и первый его элемент — строка 'honda'. Мы можем изменить значение этого элемента после того, как список будет создан:

#### *motorcycles.py*

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles[0] = 'ducati'
print(motorcycles)
```

В коде определяется список `motorcycles`, первый элемент которого содержит строку 'honda'. Значение первого элемента заменяется строкой 'ducati'. Из вывода видно, что первый элемент действительно изменился, тогда как остальные элементы списка сохранили прежние значения:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

Изменить можно значение любого элемента в списке, а не только первого.

## Добавление элементов в список

Новые элементы могут добавляться в списки по разным причинам: например, для ввода новых космических кораблей в игру, новых данных в визуализацию или новых зарегистрированных пользователей созданного вами сайта. Python предоставляет несколько способов добавить новые данные в существующие списки.

## Присоединение элементов в конец списка

Простейший способ добавления новых элементов в список — *присоединение* элемента в конец списка. Используя список из предыдущего примера, добавим новый элемент 'ducati' в конец списка:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles.append('ducati')
print(motorcycles)
```

Метод `append()` добавляет строку 'ducati' в конец списка, при этом другие элементы в нем остаются неизменными:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

Метод `append()` упрощает динамическое создание списков. Например, вы можете начать с пустого списка и добавлять в него элементы, используя серию команд `append()`. В следующем примере в пустой список добавляются элементы 'honda', 'yamaha' и 'suzuki':

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

Полученный список выглядит точно так же, как и списки из предыдущих примеров:

```
['honda', 'yamaha', 'suzuki']
```

Такой способ создания списков встречается очень часто, поскольку данные, которые пользователь захочет сохранить в программе, часто становятся известны только после запуска программы. Чтобы пользователь мог управлять содержимым списка, начните с определения пустого списка, а затем добавляйте в него каждое новое значение.

### ***Вставка элементов в список***

Метод `insert()` позволяет добавить новый элемент в произвольную позицию списка. Для этого следует указать индекс и значение нового элемента.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

motorcycles.insert(0, 'ducati')
print(motorcycles)
```

В этом примере значение `'ducati'` вставляется в начало списка. Метод `insert()` выделяет свободное место в позиции `0` и сохраняет в нем значение `'ducati'`:

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

Эта операция сдвигает все остальные значения в списке на одну позицию вправо.

### ***Удаление элементов из списка***

Нередко возникает необходимость в удалении одного или нескольких элементов из списка. Например, сбитый в игре корабль пришельца стоит удалить из списка активных врагов. Или пользователь решает удалить свою учетную запись в созданном вами веб-приложении, и тогда этого пользователя следует удалить из списка активных пользователей. Элементы удаляются из списка по позиции или по значению.

### ***Удаление элемента с помощью оператора `del`***

Если вам известна позиция элемента, который должен быть удален из списка, воспользуйтесь оператором `del`:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[0]
print(motorcycles)
```

Оператор `del` удаляет первый элемент, `'honda'`, из списка `motorcycles`:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

С помощью оператора `del` вы можете удалить элемент из любой позиции списка, если вам известен его индекс. Например, вот как из списка удаляется второй элемент `'yamaha'`:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```



Второй элемент исчез из списка:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

В обоих примерах значение, удаленное из списка после использования оператора `del`, становится недоступным.

### Удаление элемента с помощью метода `pop()`

Иногда значение, удаляемое из списка, должно как-то использоваться. Допустим, вы хотите получить координаты *x* и *y* только что сбитого корабля пришельцев, чтобы изобразить взрыв в этой позиции. В веб-приложении пользователя, удаленного из списка активных участников, можно добавить в список неактивных и т. д.

Метод `pop()` удаляет последний элемент из списка, но позволяет работать с ним после удаления. Термин *pop* («выталкивание») возник из-за того, что список представляет собой стопку элементов и один элемент выталкивается из вершины стопки. В этой аналогии вершина стека соответствует концу списка.

Удалим мотоцикл из списка:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
 print(motorcycles)

❷ popped_motorcycle = motorcycles.pop()
❸ print(motorcycles)
❹ print(popped_motorcycle)
```

Сначала определяется и выводится содержимое списка `motorcycles` ❶. Затем значение извлекается из списка и сохраняется в переменной `popped_motorcycle` ❷. Вывод измененного списка ❸ показывает, что значение было удалено из списка. Затем мы выводим извлеченное значение ❹, демонстрируя, что удаленное из списка значение остается доступным в программе.

Из вывода видно, что значение `'suzuki'`, удаленное в конце списка, теперь хранится в переменной `popped_motorcycle`:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

Для чего может понадобиться метод `pop()`? Представьте, что информация о мотоциклах хранится в списке в хронологическом порядке, соответствующем дате их покупки. В таком случае команда `pop()` может использоваться для вывода сообщения о последнем купленном мотоцикле:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f"The last motorcycle I owned was a {last_owned.title()}.")
```

Программа выводит простое сообщение:

```
The last motorcycle I owned was a Suzuki.
```

### **Удаление элементов из произвольной позиции списка**

Вызов `pop()` может использоваться для удаления элемента в произвольной позиции списка; для этого следует указать индекс удаляемого элемента в круглых скобках:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

first_owned = motorcycles.pop(0)
print(f"The first motorcycle I owned was a {first_owned.title()}.")
```

Сначала первый элемент извлекается из списка, а затем выводится сообщение об этом мотоцикле. Программа выдает простое сообщение о мотоцикле, который был куплен первым:

```
The first motorcycle I owned was a Honda.
```

Помните, что после каждого вызова `pop()` элемент, с которым вы работаете, уже не находится в списке.

Если вы не уверены в том, что выбрать: оператор `del` или метод `pop()`, — вам поможет простое правило: если вы собираетесь просто удалить элемент из списка, никак не используя его после удаления, то выбирайте оператор `del`; в противном случае выбирайте метод `pop()`.

### **Удаление элементов по значению**

Иногда позиция удаляемого элемента неизвестна. Если вы знаете только значение элемента, то используйте метод `remove()`.

Допустим, из списка нужно удалить значение `'ducati'`:

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

motorcycles.remove('ducati')
print(motorcycles)
```

Благодаря методу `remove()` Python получает указание выяснить, в какой позиции списка находится значение `'ducati'`, и удалить этот элемент:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

Кроме того, метод `remove()` может использоваться для работы со значением, которое удаляется из списка. Следующая программа удаляет значение `'ducati'` и выводит причину удаления:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

❷ too_expensive = 'ducati'
❸ motorcycles.remove(too_expensive)
print(motorcycles)
❹ print(f"\nA {too_expensive.title()} is too expensive for me.")
```

После определения списка ❶ значение 'ducati' сохраняется в переменной `too_expensive` ❷. Затем эта переменная сообщает Python, какое значение должно быть удалено из списка ❸. Значение 'ducati' было удалено из списка ❹, но продолжает храниться в переменной `too_expensive`, что позволяет вывести сообщение, в котором указывается причина удаления 'ducati' из списка мотоциклов:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

```
A Ducati is too expensive for me.
```

### ПРИМЕЧАНИЕ

Метод `remove()` удаляет только первое вхождение заданного значения. Если существует вероятность того, что значение встречается в списке несколько раз, то используйте цикл, чтобы определить, были ли удалены все вхождения данного значения. О том, как это делать, рассказано в главе 7.

## УПРАЖНЕНИЯ

Следующие упражнения немного сложнее упражнений из главы 2, но дают возможность попрактиковаться в выполнении всех описанных операций со списками.

**3.4. Список гостей.** Если бы вы могли пригласить на обед кого угодно (из живых или умерших), то кто бы это был? Создайте список минимум из трех человек. Затем используйте его для вывода пригласительного сообщения каждому участнику.

**3.5. Изменение списка гостей.** Вы только что узнали, что один из гостей не сможет прийти, поэтому вам придется разослать новые приглашения. Отсутствующего гостя нужно заменить кем-то другим.

- Начните с программы из упражнения 3.4. Добавьте в конец кода вызов функции `print()` для вывода имени гостя, который не сможет прийти.
- Измените список и замените имя гостя, который прийти не сможет, именем нового приглашенного.
- Выведите новый набор сообщений с приглашениями — по одному для каждого участника, входящего в список.

**3.6. Больше гостей.** Вы решили купить обеденный стол большего размера. Дополнительные места позволяют пригласить на обед еще трех гостей.

- Начните с программы из упражнения 3.4 или 3.5. Добавьте в конец кода вызов функции `print()` для вывода сообщения о том, что вы нашли стол большего размера.
- Добавьте вызов `insert()` для добавления одного нового гостя в начало списка.
- Добавьте вызов `insert()` для добавления одного нового гостя в середину списка.
- Добавьте вызов `append()` для добавления одного нового гостя в конец списка.
- Выведите новый набор сообщений с приглашениями — по одному для каждого участника, входящего в список.

**3.7. Сокращение списка гостей.** Только что выяснилось, что новый обеденный стол привезти вовремя не успеют и места хватит только для двух гостей.

- Начните с программы из упражнения 3.6. Добавьте команду для вывода сообщения о том, что на обед приглашаются всего два гостя.
- Используйте метод `pop()` для последовательного удаления гостей из списка до тех пор, пока в нем не останутся только два человека. Каждый раз, когда из списка удаляется очередное имя, выведите для этого человека сообщение о том, что вы сожалеете об отмене приглашения.
- Выведите сообщение для каждого из двух человек, остающихся в списке. Оно должно подтверждать, что более раннее приглашение остается в силе.
- Используйте оператор `del` для удаления двух последних имен, чтобы список остался пустым. Выведите список, чтобы убедиться в том, что в конце работы программы список действительно не содержит ни одного элемента.

## Упорядочение списка

Нередко список создается в непредсказуемом порядке, поскольку очередность получения данных от пользователя не всегда находится под вашим контролем. И хотя во многих случаях такое положение дел неизбежно, часто требуется вывести информацию в определенном порядке. В одних случаях требуется сохранить исходный порядок элементов в списке, в других этот порядок должен быть изменен. Python предоставляет несколько разных способов упорядочения списка в зависимости от ситуации.

### Постоянная сортировка списка с помощью метода `sort()`

Метод `sort()` позволяет относительно легко отсортировать список. Допустим, имеется список машин, и вы хотите изменить порядок, расставив элементы по алфавиту. Чтобы упростить задачу, предположим, что все значения в списке состоят из символов нижнего регистра.

***cars.py***

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort()
print(cars)
```

Метод `sort()` изменяет порядок элементов в списке навсегда. Названия машин располагаются в алфавитном порядке, и вернуться к исходному порядку уже не удастся:

```
['audi', 'bmw', 'subaru', 'toyota']
```

Список также можно отсортировать в обратном алфавитном порядке; для этого методу `sort()` следует передать аргумент `reverse=True`:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

И снова порядок элементов изменяется навсегда:

```
['toyota', 'subaru', 'bmw', 'audi']
```

### ***Временная сортировка списка с помощью функции `sorted()`***

Чтобы сохранить исходный порядок элементов списка, но временно представить их в отсортированном порядке, можно воспользоваться функцией `sorted()`. Она позволяет представить список в определенном порядке, но не изменяет фактический порядок элементов в списке.

Попробуем применить эту функцию к списку машин:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
```

- ```
❶ print("Here is the original list:")
   print(cars)

❷ print("\nHere is the sorted list:")
   print(sorted(cars))

❸ print("\nHere is the original list again:")
   print(cars)
```

Сначала список выводится в исходном порядке ❶, а затем в алфавитном ❷. После того как список будет выведен в новом порядке, мы убеждаемся в том, что список все еще хранится в исходном порядке ❸:

```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']
```

```
Here is the sorted list:  
['audi', 'bmw', 'subaru', 'toyota']
```

❶ Here is the original list again:
['bmw', 'audi', 'toyota', 'subaru']

Обратите внимание: после вызова функции `sorted()` список продолжает храниться в исходном порядке ❶. Функции также можно передать аргумент `reverse=True`, чтобы список был представлен в порядке, обратном алфавитному.

ПРИМЕЧАНИЕ

Сортировка списка по алфавиту немного усложняется, если не все значения записаны в нижнем регистре. При определении порядка сортировки появляются разные способы интерпретации прописных букв, и точное определение порядка может оказаться более сложной задачей, чем нам хотелось бы в текущий момент. Однако большинство подходов к сортировке строятся на принципах, описанных в этом подразделе.

Вывод списка в обратном порядке

Чтобы переставить элементы списка в обратном порядке, используйте метод `reverse()`. Например, если список машин первоначально хранился в хронологическом порядке, соответствующем дате приобретения, то элементы можно переставить в обратном хронологическом порядке:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']  
print(cars)  
  
cars.reverse()  
print(cars)
```

Обратите внимание: метод `reverse()` не сортирует элементы в обратном алфавитном порядке, а просто меняет порядок списка на обратный:

```
['bmw', 'audi', 'toyota', 'subaru']  
['subaru', 'toyota', 'audi', 'bmw']
```

Метод `reverse()` изменяет порядок элементов навсегда, но вы можете легко вернуться к исходному порядку, снова применив `reverse()` к обратному списку.

Определение длины списка

Быстро определить длину списка позволяет функция `len()`. Список в нашем примере состоит из четырех элементов, поэтому его длина равна 4:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']  
>>> len(cars)  
4
```

С помощью функции `len()` можно определять количество пришельцев, которых необходимо сбить в игре; устанавливать объем данных, которыми необходимо управлять в визуализации; вычислять количество зарегистрированных пользователей на сайте и т. д.

ПРИМЕЧАНИЕ

Python подсчитывает элементы списка, начиная с 1, поэтому при определении длины списка ошибок «смещения на 1» уже быть не должно.

УПРАЖНЕНИЯ

3.8. Повидать мир. Вспомните хотя бы пять стран, в которых вам хотелось бы побывать.

- Сохраните названия стран в списке. Проследите за тем, чтобы он не хранился в алфавитном порядке.
- Выведите список в исходном порядке. Не беспокойтесь об оформлении, просто выведите его как обычный список Python.
- Используйте функцию `sorted()` для вывода списка в алфавитном порядке без изменения списка.
- Снова выведите список, чтобы показать, что он по-прежнему хранится в исходном порядке.
- Используйте функцию `sorted()` для вывода списка в обратном алфавитном порядке без изменения порядка исходного списка.
- Снова выведите список, чтобы показать, что исходный порядок не изменился.
- Измените порядок элементов, вызвав метод `reverse()`. Выведите список, чтобы показать, что элементы следуют в другом порядке.
- Измените порядок элементов, повторно вызвав `reverse()`. Выведите список, чтобы показать, что список вернулся к исходному порядку.
- Отсортируйте список в алфавитном порядке, вызвав `sort()`. Выведите список, чтобы показать, что элементы следуют в другом порядке.
- Вызовите `sort()`, чтобы переставить элементы списка в обратном алфавитном порядке. Выведите список, чтобы показать, что порядок элементов изменился.

3.9. Количество гостей. В одной из программ из упражнений 3.4–3.7 используйте функцию `len()` для вывода сообщения, в котором указано количество людей, приглашенных на обед.

3.10. Все функции. Придумайте, какую информацию вы могли бы хранить в списке. Например, создайте список гор, рек, стран, городов, языков... словом, чего угодно. Напишите программу, которая создает список элементов, а затем вызывает каждую функцию, упоминавшуюся в этой главе, хотя бы один раз.

Ошибки индексирования при работе со списками

Когда программист только начинает работать со списками, он часто допускает одну характерную ошибку. Например, имеется список с тремя элементами, и программа запрашивает четвертый:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']  
print(motorcycles[3])
```

В этом случае происходит *ошибка индексирования*:

```
Traceback (most recent call last):  
  File "motorcycles.py", line 2, in <module>  
    print(motorcycles[3])  
            ~~~~~^  
IndexError: list index out of range
```

Python пытается вернуть элемент с индексом 3. Однако при поиске по списку ни у одного элемента `motorcycles` нет этого индекса. Из-за смещения индексов на 1 эта ошибка весьма распространена. Люди думают, что третьим является элемент с индексом 3, поскольку начинают отсчет с 1. Но для Python третьим является элемент с индексом 2, так как индексирование начинается с 0.

Ошибка индексирования означает, что Python не может понять, какой индекс запрашивается в программе. Если в вашей программе происходит ошибка индексирования, то попробуйте уменьшить запрашиваемый индекс на 1. Затем снова запустите программу и проверьте правильность результатов.

Помните, что для обращения к последнему элементу в списке используется индекс `-1`. Этот способ работает всегда, даже если размер списка изменился с момента последнего обращения к нему:

```
motorcycles = ['honda', 'yamaha', 'suzuki']  
print(motorcycles[-1])
```

Индекс `-1` всегда возвращает последний элемент списка, в данном случае значение `'suzuki'`:

```
'suzuki'
```

Этот синтаксис порождает ошибку только в одном случае — при попытке получить последний элемент пустого списка:

```
motorcycles = []  
print(motorcycles[-1])
```


В списке `motorcycles` нет ни одного элемента, поэтому Python снова выдает ошибку индексирования:

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[-1])
    ~~~~~^
IndexError: list index out of range
```

ПРИМЕЧАНИЕ

Если в вашей программе произошла ошибка индексирования и вы не знаете, как ее исправить, попробуйте вывести список или хотя бы его длину. Возможно, ваш список выглядит совсем не так, как вы думаете, особенно если его содержимое динамически определялось программой. Фактическое состояние списка или точное количество элементов в нем может выявить логические ошибки такого рода.

УПРАЖНЕНИЯ

3.11. Намеренная ошибка. Если ни в одной из предшествующих программ вы еще не сталкивались с ошибками индексирования, то попробуйте создать такую ошибку искусственно. Измените индекс в одной из программ, чтобы вызвать ошибку индексирования. Не забудьте исправить ошибку, прежде чем закрывать программу.

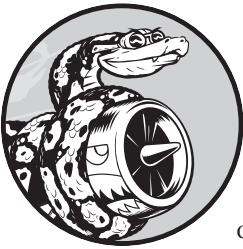
Резюме

В этой главе вы узнали, что такое списки и как работать с их отдельными элементами. Вы научились определять списки, добавлять и удалять элементы, выполнять сортировку (постоянную или временную в целях отображения). Кроме того, вы узнали, как определить длину списка и избежать ошибок индексирования при работе с ним.

В главе 4 рассматриваются приемы более эффективной работы со списками. Перебор всех элементов списка всего в нескольких строках кода, даже если список содержит тысячи или миллионы элементов, сокращает объем работы.

4

Работа со списками



В главе 3 вы научились создавать простые списки и работать с их отдельными элементами. В этой вы узнаете, как перебрать весь список, используя несколько строк кода (независимо от длины списка). Механизм *перебора* позволяет выполнить одно действие (или их набор) с каждым элементом в списке. С его помощью вы сможете эффективно работать со списками любой длины, даже состоящими из тысяч и миллионов элементов.

Перебор всего списка

Типичная задача из области программирования — перебрать все элементы списка и выполнить с каждым элементом одну и ту же операцию. Например, в компьютерной игре все экранные объекты могут смещаться на одинаковую величину, или в списке чисел к каждому элементу может применяться одна и та же статистическая операция. А может быть, вам потребовалось вывести все заголовки из списка статей на сайте. В ситуациях, требующих применения одного действия к каждому элементу списка, можно воспользоваться циклами `for`.

Допустим, имеется список с именами фокусников, и вы хотите вывести каждое имя из списка. Конечно, можно обратиться к каждому элементу по отдельности, но такой подход создает ряд проблем. Во-первых, для очень длинных списков все сведется к однообразным повторениям. Во-вторых, при любом изменении длины списка в программу придется вносить изменения. Цикл `for` решает обе проблемы: Python будет следить за всеми техническими деталями в своей внутренней реализации.

В следующем примере цикл `for` используется для вывода имен фокусников:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician)
```

Все начинается с определения списка, как и в главе 3. Затем определяется цикл `for`. С помощью этой строки Python получает указание — взять очередное имя из списка и сохранить его в переменной `magician`. Далее выводится имя, только что сохраненное в переменной `magician`. Затем строки повторяются для каждого имени в списке. Этот код можно описать так: «Для каждого фокусника в списке вывести его имя». Результат представляет собой простой перечень имен из списка:

```
alice
david
carolina
```

Подробнее о циклах

Концепция циклов очень важна, поскольку она представляет один из основных способов автоматизации повторяющихся задач компьютером. Например, в простом цикле, использованном в `magicians.py`, Python сначала читает первую строку цикла:

```
for magician in magicians:
```

Эта строка означает, что нужно взять первое значение из списка `magicians` и сохранить его в переменной `magician`. Первое значение в списке — `'alice'`. Затем Python читает следующую строку:

```
    print(magician)
```

Python выводит текущее значение `magician`, которое все еще равно `'alice'`. Так как в списке еще остались другие значения, Python возвращается к первой строке цикла:

```
for magician in magicians:
```

Python берет следующее значение из списка, `'david'`, и сохраняет его в `magician`. Затем выполняет строку:

```
    print(magician)
```

Python снова выводит текущее значение `magician`; теперь это строка `'david'`. Весь цикл повторяется еще раз с последним значением в списке, `'carolina'`. Так как других значений в списке не осталось, Python переходит к следующей строке в программе. В данном случае после цикла `for` ничего нет, поэтому программа просто завершается.

Используя циклы впервые, помните: все действия повторяются по одному разу для каждого элемента в списке независимо от их количества. Если список содержит миллион элементов, то Python повторит эти действия миллион раз — обычно это происходит очень быстро.

Кроме того, следует учесть, что при написании собственных циклов `for` временной переменной для текущего значения из списка можно присвоить любое имя. Однако на практике рекомендуется выбирать содержательное имя, описывающее отдельный

элемент списка. Например, вот хороший способ запустить цикл `for` для получения списка кошек, собак и общего списка предметов:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

Выполнение этого правила поможет вам проследить за тем, какие действия выполняются с каждым элементом в цикле `for`. В зависимости от формы числа имени (единственного или множественного) вы сможете понять, с чем работает данная часть кода — с отдельным элементом или всем списком.

Более сложные действия в циклах `for`

В цикле `for` с каждым элементом списка может выполняться практически любое действие. Дополним предыдущий пример, чтобы программа выводила для каждого фокусника отдельное сообщение:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
```

Этот код отличается от предыдущего лишь тем, что для каждого фокусника создается сообщение с его именем. При первом проходе цикла переменная `magician` содержит значение `'alice'`, поэтому Python начинает первое сообщение с имени `'Alice'`. При втором проходе сообщение будет начинаться с имени `'David'`, а при третьем — с имени `'Carolina'`.

В выводе вы увидите персональное сообщение для каждого фокусника из списка:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

Тело цикла `for` может содержать сколько угодно строк кода. Каждая строка с начальным отступом после строки `for magician in magicians` считается находящейся *в цикле* и выполняется по одному разу для каждого значения в списке. Таким образом, с каждым значением в списке можно выполнить любые операции на ваше усмотрение.

Добавим в сообщение для каждого фокусника вторую строку, в которой сообщаем каждому фокуснику, что с нетерпением ждем его следующего трюка:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\\n")
```

Поскольку оба вызова функции `print()` снабжены отступами, каждая строка будет выполнена по одному разу для каждого фокусника в списке. Символ новой строки

("\\n") во втором вызове `print()` вставляет пустую строку после каждого прохода цикла. В результате будет создан набор сообщений, аккуратно сгруппированных для каждого фокусника в списке:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.
```

```
David, that was a great trick!
I can't wait to see your next trick, David.
```

```
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Повторюсь: тело цикла `for` может содержать сколько угодно строк кода. На практике часто требуется выполнить в этом цикле несколько разных операций для каждого элемента списка.

Выполнение действий после цикла `for`

Что происходит после завершения цикла `for`? Обычно программа выводит некую сводную информацию или переходит к другим операциям.

Каждая строка кода после цикла `for`, не имеющая отступа, выполняется без повторения. Допустим, вы хотите вывести сообщение для всей группы фокусников и поблагодарить их за превосходное представление. Чтобы вывести общее сообщение после всех отдельных, поместите его после цикла `for`, не делая отступа:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

print("Thank you, everyone. That was a great magic show!")
```

Первые два вызова функции `print()` повторяются по одному разу для каждого фокусника в списке, как было показано ранее. Но поскольку последняя строка не имеет отступа, это сообщение выводится только раз:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.
```

```
David, that was a great trick!
I can't wait to see your next trick, David.
```

```
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

```
Thank you, everyone. That was a great magic show!
```

При обработке данных в циклах `for` завершающее сообщение позволяет подвести итог операции, выполненной со всем набором данных. Например, цикл `for` может

инициализировать игру, перебирая список персонажей и изображая каждого персонажа на экране. После цикла выполняется блок без отступа, который выводит кнопку Play Now (Начало игры) после того, как все персонажи появятся на экране.

Предотвращение ошибок с отступами

Python использует отступы, чтобы определить, как одна строка или группа строк связана с остальной частью программы. В предыдущих примерах строки, выводившие сообщения для отдельных фокусников, были частью цикла `for`, поскольку имели отступы. Наличие отступов в Python сильно упрощает чтение кода. Фактически отступы заставляют разработчика писать отформатированный код, имеющий четкую визуальную структуру. В более длинных программах Python могут встречаться блоки кода с отступами нескольких разных уровней. Эти уровни помогают вам понимать общую структуру программы.

Когда разработчики только начинают писать код, работа которого зависит от правильности отступов, в их коде нередко встречаются распространенные *ошибки, связанные с отступами* (indentation errors). Например, иногда программисты расставляют отступы в коде, не нуждающемся в отступах, или наоборот — забывают добавлять отступы в блоках, где это необходимо. Несколько примеров помогут вам избежать подобных ошибок в будущем и успешно исправлять их, когда они встретятся в ваших программах.

Итак, рассмотрим несколько типичных ошибок, связанных с использованием отступов.

Пропущенный отступ

Строка после оператора `for` в цикле всегда должна иметь отступ. Если вы забудете добавить его, то Python напомнит вам об этом:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶ print(magician)
```

Вызов функции `print()` ❶ должен иметь отступ, но здесь его нет. Когда Python ожидает увидеть блок с отступом, но не находит его, появляется сообщение с указанием номера строки:

```
File "magicians.py", line 3
    print(magician)
    ^
```

IndentationError: expected an indented block after 'for' statement on line 2

Обычно для устранения подобных ошибок достаточно поставить отступ в строке (или строках), следующей (-их) непосредственно после оператора `for`.

Пропущенные отступы в других строках

Иногда цикл выполняется без ошибок, но не выдает ожидаемых результатов. Такое часто происходит, когда вы пытаетесь выполнить несколько операций в цикле, но забываете добавить отступ в некоторые из строк.

Например, вот что происходит, если вы забудете добавить отступ во вторую строку в цикле:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶ print(f"I can't wait to see your next trick, {magician.title()}.\\n")
```

Второй вызов функции `print()` ❶ должен иметь отступ, но поскольку Python находит хотя бы одну строку с отступом после оператора `for`, сообщение об ошибке не выдается. В результате первый вызов `print()` будет выполнен для каждого элемента в списке, поскольку в нем есть отступ. Во втором вызове `print()` отступа нет, поэтому он будет выполнен только раз после завершения цикла. Так как последним значением `magician` является строка `'carolina'`, второе сообщение будет выведено только с этим именем:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Это пример *логической ошибки* (logical error). Синтаксис является допустимым, но код не приводит к желаемому результату, поскольку проблема кроется в его логике. Если какое-то действие должно повторяться для каждого элемента в списке, но выполняется только раз, то проверьте, не нужно ли добавить отступы в одной или нескольких строках кода.

Лишние отступы

Если вы случайно поставите отступ в строке, в которой он не нужен, то Python сообщит об этом:

hello_world.py

```
message = "Hello Python world!"
    print(message)
```

Отступ для вызова функции `print()` не нужен, поскольку эта строка не подчинена предшествующей; Python сообщает об ошибке:

```
File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

Чтобы избежать непредвиденных ошибок с отступами, используйте их только там, где для этого есть конкретные причины. В тех программах, которые вы пишете на данной стадии изучения Python, отступы нужны только в строках действий, повторяемых для каждого элемента в цикле `for`.

Лишние отступы после цикла

Если вы случайно добавите отступ в код, который должен выполняться *после* завершения цикла, этот код будет выполнен для каждого элемента. Иногда Python выводит сообщение об ошибке, но часто дело ограничивается простой логической ошибкой.

Например, посмотрим, что произойдет, если случайно добавить отступ в строку, в которой мы благодарим фокусников за хорошее представление:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}\n")
```

❶ `print("Thank you everyone, that was a great magic show!")`

Поскольку в последней строке ❶ отступ есть, сообщение будет продублировано для каждого фокусника в списке:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.
```

```
Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.
```

```
Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

```
Thank you everyone, that was a great magic show!
```

Это еще один пример логической ошибки, подобной той, которая описана в подразделе «Пропущенные отступы в других строках». Python не знает, что вы пытаетесь сделать с помощью кода, поэтому просто выполняет весь код, не нарушающий правил синтаксиса. Если действие, которое должно выполняться один раз, происходит многократно, то проверьте, нет ли лишнего отступа в соответствующей строке кода; если есть — удалите.

Пропущенное двоеточие

Двоеточие в конце оператора `for` сообщает Python, что следующая строка является началом цикла.

```
magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians:
    print(magician)
```

Если вы случайно забудете поставить двоеточие, как показано в примере ❶, то произойдет синтаксическая ошибка, поскольку Python не понимает, что именно вы пытаетесь сделать:

```
File "magicians.py", line 2
    for magician in magicians
                                ^
```

SyntaxError: expected ':'

Python не знает, забыли ли вы указать двоеточие или хотели дополнить код операторами, чтобы создать более сложный цикл. Если интерпретатор способен определить возможное исправление, то предложит его, например добавит двоеточие в конец строки, как это сделано в строке `expected ':'`. Одни ошибки легко исправляются благодаря подобным предложениям в трассировках Python. Другие исправить гораздо сложнее, даже если в конечном счете исправление касается лишь одного символа. Не расстраивайтесь, если на поиск небольшого исправления уходит много времени; не вы первый, не вы последний.

УПРАЖНЕНИЯ

4.1. Пицца. Вспомните по крайней мере три названия ваших любимых видов пиццы. Сохраните их в списке и используйте цикл `for` для вывода всех названий.

- Измените цикл `for` так, чтобы вместо простого названия пиццы выводилось сообщение, содержащее это название. Таким образом, для каждого элемента должна выводиться строка с простым текстом вида «Я люблю пиццу пеперони».
- Добавьте в конец программы (после цикла `for`) строку, в которой будет указано, насколько вы любите пиццу. Таким образом, вывод должен состоять из трех (и более) строк с названиями пиццы и дополнительного предложения — скажем, «Я очень люблю пиццу!»

4.2. Животные. Создайте список из трех (и более) животных, обладающих общей характеристикой. Используйте цикл `for` для вывода названий каждого животного.

- Измените программу так, чтобы она выводила сообщение о каждом животном, — например, «Собака — отличное домашнее животное».
- Добавьте в конец программы строку с описанием общего свойства. Например, можно вывести сообщение «Любое из этих животных — отличное домашнее животное!»

Создание числовых списков

Необходимость хранения наборов чисел возникает в программах по многим причинам. Например, в компьютерной игре могут храниться координаты каждого персонажа на экране, таблицы рекордов и т. д. В программах визуализации данных пользователь почти всегда работает с наборами чисел: температурой, расстоянием, численностью населения, широтой/долготой и другими числовыми данными.

Списки идеально подходят для хранения наборов чисел, а Python предоставляет специальные средства, позволяющие эффективно работать с числовыми списками. Достаточно один раз понять, как пользоваться этими средствами, — и ваш код будет хорошо работать даже в том случае, если список содержит миллионы элементов.

Функция `range()`

Функция `range()` упрощает создание числовых последовательностей. Например, с ее помощью можно легко вывести серию чисел:

first_numbers.py

```
for value in range(1,5):  
    print(value)
```

И хотя на первый взгляд может показаться, что код должен вывести числа от 1 до 5, на самом деле число 5 не выводится:

```
1  
2  
3  
4
```

В этом примере функция `range()` выводит только числа от 1 до 4. Перед вами еще один вариант явления «смещения на 1», часто встречающегося в языках программирования. При выполнении `range()` Python начинает отсчет от первого переданного значения и прекращает его при достижении второго. Так как на втором значении происходит остановка, конец интервала (в данном случае 5) не встречается в выводе.

Чтобы вывести числа от 1 до 5, используйте функцию `range(1, 6)`:

```
for value in range(1, 6):  
    print(value)
```

На этот раз вывод начинается с единицы и завершается цифрой 5:

```
1  
2  
3  
4  
5
```

Если ваша программа при использовании функции `range()` выводит не тот результат, на который вы рассчитывали, попробуйте увеличить конечное значение на 1.

Кроме того, функции `range()` можно передать только один аргумент; в этом случае последовательность чисел будет начинаться с 0. Например, `range(6)` вернет числа от 0 до 5.

Использование функции `range()` для создания числового списка

Если вы хотите создать числовой список, то преобразуйте результаты `range()` в список с помощью функции `list()`. Если заключить вызов `range()` в функцию `list()`, то результат будет представлять собой список с числовыми элементами.

В примере из предыдущего подраздела числовая последовательность просто выводилась на экран. Тот же набор чисел можно преобразовать в список с помощью функции `list()`:

```
numbers = list(range(1, 6))
print(numbers)
```

Результат:

```
[1, 2, 3, 4, 5]
```

Кроме того, функция `range()` может генерировать числовые последовательности, пропуская числа в заданном диапазоне. Если вы передадите в `range()` третий аргумент, то Python будет использовать это значение в качестве величины шага при генерации чисел.

Например, список четных чисел от 1 до 10 создается так:

even_numbers.py

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

В этом примере функция `range()` начинает со значения 2, а затем увеличивает его на 2. Приращение 2 последовательно применяется до тех пор, пока не будет достигнуто или пройдено конечное значение 11, после чего выводится результат:

```
[2, 4, 6, 8, 10]
```

С помощью функции `range()` можно создать практически любой диапазон чисел. Например, как бы вы создали список квадратов всех целых чисел от 1 до 10? В языке Python операция возведения в степень обозначается двумя звездочками (**). Один из возможных вариантов списка квадратов выглядит так:

square_numbers.py

```
squares = []
for value in range(1, 11):
    ❶ square = value ** 2
    ❷ squares.append(square)

print(squares)
```

Сначала создается пустой список `squares`. Затем Python перебирает все значения от 1 до 10 с помощью функции `range()`. В цикле текущее значение возводится во вторую степень, а результат сохраняется в переменной `square` ❶. Каждое новое значение `square` присоединяется к списку `squares` ❷. Наконец, после завершения цикла выводится список квадратов:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Чтобы сделать код более компактным, можно опустить временную переменную `square` и присоединять каждое новое значение прямо к списку:

```
squares = []
for value in range(1, 11):
    squares.append(value**2)
```

```
print(squares)
```

Эта строка выполняет ту же работу, что и код цикла `for`, показанный выше. Каждое значение в цикле возводится во вторую степень, а затем немедленно присоединяется к списку квадратов.

При создании более сложных списков можно применять любой из двух подходов. В одних случаях использование временной переменной упрощает чтение кода; в других — чрезмерно удлиняет код. Сначала сосредоточьтесь на написании четкого и понятного кода, который делает именно то, что нужно, и только потом переходите к анализу кода и поиску более эффективных решений.

Простая статистика с числовыми списками

Некоторые функции Python предназначены для работы с числовыми списками. Например, вы можете легко узнать минимум, максимум и сумму числового списка:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

ПРИМЕЧАНИЕ

В примерах этого раздела используются короткие списки чисел, но это делается только для того, чтобы данные помещались на странице. Примеры будут работать и в том случае, если список содержит миллионы чисел.

Генераторы списков

Описанный выше пример генерирования списка `squares` состоял из трех или четырех строк кода. *Генератор списка* (list comprehension) позволяет создать тот же список всего одной строкой, объединяя цикл `for` и создание новых элементов в одну строку и автоматически добавляя к списку все новые элементы. В учебниках для начинающих программистов не всегда рассказывается о генераторах списков, но я привожу этот материал, поскольку вы с большой вероятностью встретите данную конструкцию, как только начнете просматривать код других разработчиков.

В следующем примере знакомый вам список квадратов создается с помощью генератора списка:

`squares.py`

```
squares = [value**2 for value in range(1, 11)]  
print(squares)
```

Чтобы использовать этот синтаксис, начните с описательного имени списка — например, `squares`. Затем откройте квадратные скобки и определите выражение для значений, которые должны быть сохранены в новом списке. В данном примере это выражение `value**2`, которое возводит значение во вторую степень. Затем напишите цикл `for` для генерирования чисел, которые должны передаваться выражению, и закройте квадратные скобки. Цикл `for` в данном примере — `for value in range(1, 11)` — передает значения с 1 до 10 выражению `value**2`. Обратите внимание на отсутствие двоеточия в конце оператора `for`.

Результатом будет уже знакомый вам список квадратов:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Чтобы успешно писать собственные генераторы списков, необходим определенный опыт. Тем не менее, освоив создание обычных списков, вы оцените возможности генераторов. Когда после очередного трех-четырёхстрочного блока вам надоест создавать списки, подумайте о написании генераторов.

УПРАЖНЕНИЯ

4.3. Считаю до 20. Используйте цикл `for` для вывода чисел от 1 до 20 включительно.

4.4. Миллион. Создайте список чисел от 1 до 1 000 000, затем воспользуйтесь циклом `for` для вывода чисел. (Если вывод занимает слишком много времени, то остановите его, нажав `Ctrl+C` или закрыв окно вывода.)

4.5. Суммирование миллиона чисел. Создайте список чисел от 1 до 1 000 000, затем воспользуйтесь функциями `min()` и `max()` и убедитесь, что список действительно начинается с 1 и заканчивается 1 000 000. Вызовите функцию `sum()` и посмотрите, насколько быстро Python сможет суммировать миллион чисел.

4.6. Нечетные числа. Используйте третий аргумент функции `range()`, чтобы создать список нечетных чисел от 1 до 20. Выведите все числа в цикле `for`.

4.7. Тройки. Создайте список чисел, кратных 3, в диапазоне от 3 до 30. Выведите все числа списка в цикле `for`.

4.8. Кубы. Результат возведения числа в третью степень называется кубом. Например, куб 2 записывается в языке Python как `2**3`. Создайте список первых 10 кубов (то есть кубов всех целых чисел от 1 до 10) и с помощью цикла `for` выведите значения каждого куба.

4.9. Генератор кубов. Используйте конструкцию генератора списка для создания списка первых 10 кубов.

Работа с частью списка

В главе 3 вы узнали, как обращаться к отдельным элементам списка, а в этой мы занимались перебором всех его элементов. Можно работать и с конкретным подмножеством элементов списка; в Python такие подмножества называются *срезами* (slices).

Нарезка списков

Чтобы создать срез списка, следует задать индексы первого и последнего элементов, с которыми вы намереваетесь работать. Как и в случае с функцией `range()`, Python останавливается на элементе, предшествующем второму индексу. Например, чтобы вывести первые три элемента списка, запросите индексы с 0 по 3 и получите элементы 0, 1 и 2.

В следующем примере используется список игроков команды:

players.py

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[0:3])
```

Здесь выводится часть списка. Вывод сохраняет структуру списка, но содержит только первых трех игроков:

```
['charles', 'martina', 'michael']
```

Подмножество может содержать любую часть списка. Например, чтобы ограничиться вторым, третьим и четвертым элементами списка, создайте срез, который начинается с индекса 1 и заканчивается на индексе 4:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

На этот раз срез начинается с элемента `'martina'` и заканчивается элементом `'florence'`:

```
['martina', 'michael', 'florence']
```

Если первый индекс среза не указан, то Python автоматически начинает срез от начала списка:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Без начального индекса Python берет элементы от начала списка:

```
['charles', 'martina', 'michael', 'florence']
```

Аналогичный синтаксис работает и для срезов, содержащих конец списка. Например, если вам нужны все элементы с третьего до последнего, то начните с индекса 2 и не указывайте второй индекс:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python возвращает все элементы с третьего до конца списка:

```
['michael', 'florence', 'eli']
```

Этот синтаксис позволяет вывести все элементы от любой позиции до конца списка независимо от его длины. Вспомните, что отрицательный индекс возвращает элемент, находящийся на определенном расстоянии от конца списка; следовательно, вы можете получить любой срез от конца списка. Например, чтобы отобразить последних трех игроков из списка, используйте срез `players[-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

Программа выводит имена трех последних игроков, причем продолжает работать даже при изменении размера списка.

ПРИМЕЧАНИЕ

В квадратные скобки, определяющие срез, можно добавить третье значение. Если оно присутствует, то сообщает Python, сколько элементов следует пропускать при выборе элементов в заданном диапазоне.

Перебор содержимого среза

Если вы хотите перебрать элементы, входящие в подмножество элементов, используйте срез в цикле `for`. В следующем примере программа перебирает первых трех игроков и выводит их имена как часть простого списка:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

Вместо того чтобы перебирать весь список игроков, Python ограничивается первыми тремя именами ❶:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Срезы очень полезны во многих ситуациях. Например, при создании компьютерной игры итоговый счет игрока может добавляться в список после окончания текущей партии. После этого программа может получить три лучших результата игрока, отсортировав список по уменьшению и получив срез, содержащий только три элемента. При работе с данными срезы могут использоваться для обработки данных блоками заданного размера. Или при создании веб-приложения срезы можно использовать для постраничного вывода информации так, чтобы на каждой странице выводился соответствующий объем информации.

Копирование списка

Часто разработчик берет существующий список и на его основе создает совершенно новый. Посмотрим, как работает копирование списков, и разберем одну ситуацию, в которой копирование списка может принести пользу.

Чтобы скопировать список, создайте срез, содержащий весь исходный список без указания первого и второго индексов (`[ : ]`). Эта конструкция создает срез, который начинается с первого элемента и завершается последним; в результате создается копия всего списка.

Представьте, что вы создали список своих любимых блюд и теперь хотите создать отдельный список блюд, которые нравятся вашему другу. Пока вашему другу нравятся все блюда из нашего списка, поэтому вы можете создать другой список, просто скопировав наш:

foods.py

```
my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Сначала создается список блюд `my_foods`. Затем создается другой список `friend_foods`. Чтобы создать копию `my_foods`, программа запрашивает срез `my_foods` без указания индексов ❶ и сохраняет копию в `friend_foods`. При выводе обоих списков становится видно, что они содержат одинаковые данные:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

Чтобы доказать, что на самом деле речь идет о двух разных списках, добавим новое блюдо в каждый список и покажем, что каждый из них отслеживает любимые блюда человека:

```
my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

❷ my_foods.append('cannoli')
❸ friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Исходные элементы `my_foods` копируются в новый список `friend_foods` ❶. Затем в каждый список добавляется новый элемент: `'cannoli'` в `my_foods` ❷, а `'ice cream'` в `friend_foods` ❸. После этого вывод двух списков наглядно показывает, что каждое блюдо находится в соответствующем списке:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

Вывод показывает, что элемент 'cannoli' находится в списке `my_foods`, а элемент 'ice cream' — нет. Видно, что 'ice cream' входит в список `friend_foods`, а элемент 'cannoli' — нет. Если бы два этих списка просто совпадали, то их содержимое не различалось бы. Например, вот что происходит при попытке копирования списка без использования среза:

```
my_foods = ['pizza', 'falafel', 'carrot cake']
```

```
# Не работает:
```

```
friend_foods = my_foods
```

```
my_foods.append('cannoli')
```

```
friend_foods.append('ice cream')
```

```
print("My favorite foods are:")
```

```
print(my_foods)
```

```
print("\nMy friend's favorite foods are:")
```

```
print(friend_foods)
```

Вместо того чтобы сохранять копию `my_foods` в `friend_foods`, мы присваиваем переменной `friend_foods` значение переменной `my_foods`. На самом деле этот синтаксис сообщает Python, что новая переменная `friend_foods` должна быть связана со списком, уже хранящимся в `my_foods`, поэтому теперь обе переменные связаны с одним списком. В результате при добавлении элемента 'cannoli' в `my_foods` этот элемент также появляется в `friend_foods`. Аналогичным образом элемент 'ice cream' оказывается в обоих списках, хотя на первый взгляд был добавлен только в `friend_foods`.

Вывод показывает, что оба списка содержат одинаковые элементы, а это совсем не то, что требовалось:

```
My favorite foods are:
```

```
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

```
My friend's favorite foods are:
```

```
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

ПРИМЕЧАНИЕ

Если какие-то подробности в этом примере кажутся непонятными, то не огорчайтесь. Если при работе с копией списка происходит что-то непредвиденное, убедитесь в том, что копируете список с помощью среза, как мы делали в первом примере.

УПРАЖНЕНИЯ

4.10. Срезы. Добавьте в конец одной из программ, написанных в этой главе, фрагмент, который делает следующее:

- выводит сообщение «Первые три пункта в списке — это:», а затем использует срез для вывода первых трех элементов из списка;
- выводит сообщение «Три пункта из середины списка:», а затем использует срез для вывода первых трех элементов из середины списка;
- выводит сообщение «Последние три пункта в списке — это:», а затем использует срез для вывода последних трех элементов из списка.

4.11. Моя пицца, твоя пицца. Начните с программы из упражнения 4.1. Создайте копию списка с видами пиццы, присвойте ему имя `friend_pizzas`. Затем сделайте следующее:

- добавьте новую пиццу в исходный список;
- добавьте другую пиццу в список `friend_pizzas`;
- докажите, что в программе существуют два разных списка. Выведите сообщение «Мои любимые пиццы:», а затем первый список в цикле `for`. Выведите сообщение «Любимые пиццы моего друга:», а затем второй список в цикле `for`. Убедитесь в том, что каждая новая пицца находится в соответствующем списке.

4.12. Больше циклов. Во всех версиях `foods.py` из этого раздела мы избегали использования цикла `for` при выводе для экономии места. Выберите версию `foods.py` и напишите два цикла `for` для вывода каждого списка.

Кортежи

Списки позволяют хранить наборы элементов, которые могут изменяться на протяжении жизненного цикла программы. Например, возможность изменения списков необходима при работе со списками пользователей сайта или списками персонажей игры. Однако в некоторых ситуациях требуется создать список элементов, который не может изменяться. *Кортежи* (tuples) предоставляют именно такую возможность. В языке Python значения, которые не могут изменяться, называются *неизменяемыми* (immutable), а неизменяемый список называется *кортежем*.

Определение кортежа

Кортеж выглядит как список, за исключением того, что вместо квадратных скобок используются круглые. После определения кортежа вы можете обращаться к его отдельным элементам по индексам точно так же, как это делается при работе со списком.

Допустим, у нас в программе имеется прямоугольник, у которого всегда должны быть строго определенные размеры. Чтобы гарантировать их неизменность, можно объединить их в кортеж:

dimensions.py

```
dimensions = (200, 50)
print(dimensions[0])
print(dimensions[1])
```

В данном коде определяется кортеж `dimensions`, при этом вместо квадратных скобок ставятся круглые. Затем каждый элемент кортежа выводится по отдельности с помощью того же синтаксиса, который использовался для обращения к элементу списка:

```
200
50
```

Посмотрим, что произойдет при попытке изменения одного из элементов в кортеже `dimensions`:

```
dimensions = (200, 50)
dimensions[0] = 250
```

Код пытается изменить первое значение, но Python возвращает ошибку типа. По сути, мы пытаемся изменить кортеж, а эта операция недопустима для объектов данного типа, так что Python сообщает о невозможности присваивания нового значения элементу в кортеже:

```
Traceback (most recent call last):
  File "dimensions.py", line 2, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

И это хорошо, поскольку мы хотим, чтобы Python сообщал о попытке изменения размеров прямоугольника в программе, выдавая сообщение об ошибке.

ПРИМЕЧАНИЕ

Формально кортеж определяется наличием запятой; круглые скобки просто делают запись более аккуратной и понятной. Если вы хотите задать кортеж, состоящий из одного элемента, то добавьте завершающую запятую:

```
my_t = (3,)
```

Обычно создание кортежа из одного элемента не имеет особого смысла. Тем не менее это может произойти при автоматическом генерировании кортежей.

Перебор всех значений в кортеже

Для данной операции используется цикл `for`, как и при работе со списками:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python возвращает все элементы кортежа по аналогии с тем, как это делается со списком:

```
200
50
```

Замена кортежа

Элементы кортежа не могут изменяться, но вы можете присвоить новое значение переменной, в которой хранится кортеж. Таким образом, для изменения размеров прямоугольника следует переопределить весь кортеж:

```
dimensions = (200, 50)
print("Original dimensions:")
for dimension in dimensions:
    print(dimension)

dimensions = (400, 100)
print("\nModified dimensions:")
for dimension in dimensions:
    print(dimension)
```

Первые четыре строки кода определяют исходный кортеж и выводят исходные размеры. Затем в переменной `dimensions` сохраняется новый кортеж, после чего выводятся новые размеры. На этот раз Python не выдает сообщений об ошибках, поскольку замена значения переменной является допустимой операцией:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

По сравнению со списками структуры данных кортежи относительно просты. Используйте их для хранения наборов значений, которые не должны изменяться на протяжении жизненного цикла программы.

УПРАЖНЕНИЯ

4.13. Шведский стол. Меню шведского стола в ресторане состоит всего из пяти пунктов. Придумайте пять простых блюд и сохраните их в кортеже.

- Используйте цикл `for` для вывода всех блюд, предлагаемых рестораном.
- Попробуйте изменить один из элементов и убедитесь в том, что Python отказывается вносить изменения.
- Ресторан обновляет меню, заменяя два элемента другими блюдами. Добавьте блок кода, который заменяет кортеж, и используйте цикл `for` для вывода каждого элемента нового меню.

Форматирование кода

Итак, вы постепенно начинаете писать более длинные программы, и вам стоит познакомиться с некоторыми рекомендациями по форматированию кода. Не жалейте времени на то, чтобы сделать ваш код максимально читабельным. Понятный код помогает следить за тем, что делает программа, и упрощает его изучение другими разработчиками.

Программисты Python выработали ряд правил по стилю, чтобы код, написанный разными программистами, имел хотя бы отдаленно похожую структуру. Научившись писать «чистый» код Python, вы сможете понять общую структуру кода, написанного любым другим программистом, соблюдающим те же рекомендации. Если вы рассчитываете когда-нибудь стать профессиональным программистом, то стоит как можно скорее начать следовать этим рекомендациям, чтобы выработать полезную привычку.

Рекомендации по стилю

Когда кто-нибудь хочет внести изменения в язык Python, он пишет документ *PEP* (Python Enhancement Proposal, предложение по улучшению Python). Один из самых старых PEP — документ *PEP 8* с рекомендациями по форматированию кода. Он довольно длинный, но большая часть документа посвящена более сложным программным структурам, чем те, которые встречались вам до настоящего момента.

Руководство по стилю Python было написано на основании того факта, что код читается чаще, чем пишется. Вы пишете код один раз, а потом начинаете читать его, когда переходите к отладке. При расширении функциональности программы вы снова читаете код. А когда им начинают пользоваться другие программисты, они тоже читают его.

Выбирая между написанием кода, который проще пишется, и кодом, который проще читается, программисты Python почти всегда рекомендуют второй вариант. Следующие советы помогут вам с самого начала писать чистый, понятный код.

Отступы

PEP 8 рекомендует обозначать уровень отступа четырьмя пробелами. Их использование упрощает чтение программы и при этом оставляет достаточно места для нескольких уровней отступов в каждой строке.

В программах форматирования текста для создания отступов вместо пробелов часто используется табуляция. Такой способ хорошо работает в текстовых процессорах, но интерпретатор Python приходит в замешательство, когда знаки табуляции смешиваются с пробелами. В любом редакторе кода есть параметр конфигурации, который заменяет нажатие клавиши табуляции заданным количеством пробелов. Конечно, этой клавишей удобно пользоваться, но вы должны проследить за тем, чтобы редактор вставлял в документ пробелы вместо знака табуляции.

Смешение знаков табуляции и пробелов в файле может создать проблемы, сильно затрудняющие диагностику. Если вы думаете, что в программе знаки табуляции смешались с пробелами, то можете преобразовать все знаки табуляции в пробелы — в большинстве редакторов есть такая возможность.

Длина строк

Многие программисты Python рекомендуют ограничивать длину строк 80 символами. Исторически эта рекомендация появилась из-за того, что в большинстве компьютеров в одной строке терминального окна помещалось всего 79 символов. В настоящее время на экранах помещаются куда более длинные строки, но для использования стандартной длины строки в 79 символов существуют и другие причины.

Профессиональные программисты часто открывают на одном экране сразу несколько файлов; стандартная длина строки позволяет видеть все строки в двух или трех файлах, открытых на экране одновременно. Кроме того, PEP 8 рекомендует ограничивать комментарии 72 символами на строку, поскольку некоторые служебные программы, автоматически генерирующие документацию в больших проектах, добавляют символы форматирования в начале каждой строки комментария.

Рекомендации PEP 8 по выбору длины строки не являются незыблемыми, и некоторые программисты предпочитают ограничение в 99 символов. Пока вы учитесь, длина строки в коде не так важна, но учтите, что при совместной работе в группах почти всегда соблюдаются рекомендации PEP 8. В большинстве редакторов можно

установить визуальный ориентир (обычно вертикальную линию на экране), показывающий, где проходит граница.

ПРИМЕЧАНИЕ

В приложении Б описано, как настроить редактор кода, чтобы он всегда вставлял четыре пробела при нажатии клавиши табуляции и отображал вертикальную линию, позволяющую соблюдать ограничение длины строки в 79 символов.

Пустые строки

Пустые строки применяются для визуальной группировки частей программы. Используйте их для структурирования файлов, но не злоупотребляйте. Примеры, приведенные в книге, помогут вам выработать нужный баланс. Например, если в программе пять строк кода создают список, а затем следующие три строки что-то делают с этим списком, то два фрагмента уместно разделить пустой строкой. Однако между ними не стоит вставлять три или четыре пустые строки.

Пустые строки не влияют на работу кода, но отражаются на его удобочитаемости. Интерпретатор Python использует горизонтальные отступы для интерпретации смысла кода, но игнорирует межстрочные интервалы.

Другие рекомендации

PEP 8 содержит много других рекомендаций по стилю, но они в основном относятся к программам, более сложным, чем те, которые вы пишете на данный момент. По мере изучения более сложных элементов Python я буду приводить соответствующие фрагменты рекомендаций PEP 8.

УПРАЖНЕНИЯ

4.14. Просмотрите оригинальное руководство по стилю PEP 8 по адресу <https://python.org/dev/peps/pep-0008/>. На данном этапе вы будете пользоваться им относительно редко, но просмотреть его будет интересно.

4.15. Анализ кода. Выберите три программы, написанные в этой главе, и измените каждую в соответствии с рекомендациями PEP 8.

- Используйте четыре пробела для каждого уровня отступов. Настройте редактор кода так, чтобы он вставлял четыре пробела при каждом нажатии клавиши табуляции, если не сделали этого ранее (за инструкциями обращайтесь к приложению Б).
- Используйте менее 80 символов в каждой строке. Настройте редактор так, чтобы он отображал вертикальную линию на месте 80-го символа.
- Не злоупотребляйте пустыми строками в файлах программ.

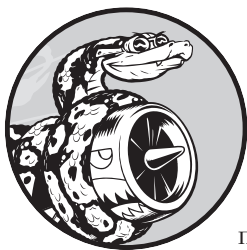
Резюме

В этой главе вы научились эффективно работать с элементами списка. Вы узнали, как работать со списком в цикле `for`, как Python использует отступы для определения структуры программы и как избежать некоторых типичных ошибок при использовании отступов. Вы научились создавать простые числовые списки, а также изучили некоторые операции с числовыми списками. Вы узнали, как создать срез списка, позволяющий работать с подмножеством элементов, и как правильно копировать списки с помощью среза. Кроме того, вы познакомились с кортежами, до определенной степени защищающими наборы значений, которые не должны изменяться, и изучили рекомендации по форматированию вашего кода (сложность которого со временем только возрастает), позволяющие упростить его чтение.

В главе 5 вы узнаете, как обрабатывать различные условия с помощью операторов `if`. Вы научитесь группировать относительно сложные наборы проверок, позволяющие обрабатывать именно ту ситуацию или информацию, которая вам нужна. Кроме того, вы узнаете, как использовать операторы `if` при переборе элементов списка и выполнять действия с элементами, отобранными по какому-то критерию.

5

Оператор if



Программисту часто приходится проверять наборы условий и принимать решения в зависимости от этих условий. Оператор `if` в языке Python позволяет проверить текущее состояние программы и выбрать дальнейшие действия в зависимости от результатов проверки.

В этой главе вы научитесь писать условные проверки для любых интересующих вас условий. Мы начнем с простых операторов `if`, а затем перейдем к более сложным сериям операторов `if` для проверки комбинированных условий. Затем эта концепция будет применена к спискам; вы узнаете, как написать цикл, который выполняет с большинством элементов списка одну операцию, но для некоторых элементов с конкретными значениями применяется особая обработка.

Простой пример

Следующий короткий пример показывает, как правильно организовать обработку специальных ситуаций с помощью оператора `if`. Допустим, у вас есть список машин и вы хотите вывести название каждой из них. Названия большинства машин должны записываться с капитализацией (первая буква в верхнем регистре, остальные в нижнем). Однако значение `'bmw'` должно записываться в верхнем регистре. Код ниже перебирает список названий машин и ищет в нем значение `'bmw'`. Для всех элементов, содержащих это значение, оно выводится в верхнем регистре:

cars.py

```
cars = ['audi', 'bmw', 'subaru', 'toyota']
```

```
for car in cars:
    ❶ if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

Цикл в этом примере сначала проверяет, содержит ли `car` значение `'bmw'` ❶. Если проверка дает положительный результат, то значение выводится в верхнем регистре. Если `car` содержит все что угодно, кроме `'bmw'`, то при выводе значения применяется капитализация:

```
Audi
BMW
Subaru
Toyota
```

В этом примере объединяются несколько концепций, о которых вы узнаете в данной главе. Для начала рассмотрим основные конструкции, применяемые для проверки условий в программах.

Проверка условий

В каждом операторе `if` центральное место занимает выражение, результатом которого является логическая истина (`True`) или логическая ложь (`False`); это выражение называется *проверкой условия* (conditional test). В зависимости от результата проверки Python решает, должен ли выполняться код в операторе `if`. Если результат условия равен `True`, то Python выполняет код, следующий за оператором `if`. Если же получен результат `False`, то Python игнорирует этот код.

Проверка равенства

Во многих условиях текущее значение переменной сравнивается с конкретным значением, интересующим вас. Простейшее условие проверяет, равно ли значение переменной конкретной величине:

```
>>> car = 'bmw'
>>> car == 'bmw'
True
```

В первой строке переменной `car` присваивается значение `'bmw'`; операция выполняется с помощью одного знака `=`, как вы уже неоднократно видели. Вторая строка проверяет, равно ли значение `car` строке `'bmw'`; для проверки используется двойной знак равенства (`==`). Это *оператор равенства* (equality operator), и он возвращает `True`, если значения слева и справа от оператора равны; если же значения не совпадают, то оператор возвращает `False`. В нашем примере значения совпадают, поэтому Python возвращает `True`.

Если `car` принимает любое другое значение вместо `'bmw'`, то проверка возвращает `False`:

```
>>> car = 'audi'
>>> car == 'bmw'
False
```

Одиночный знак равенства выполняет операцию; первую строку кода можно прочитать как «Присвоить `car` значение `'audi'`». С другой стороны, двойной знак равенства, как во второй строке кода, задает вопрос: «Значение `car` равно `'bmw'`?» Такое применение знаков равенства встречается во многих языках программирования.

Проверка равенства без учета регистра

В языке Python проверка равенства выполняется с учетом регистра. Например, два значения с разным регистром символов равными не считаются:

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

Если регистр символов важен, то такое поведение приносит пользу. Но если проверка должна выполняться на уровне символов без учета регистра, то преобразуйте значение переменной в нижний регистр перед выполнением сравнения:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Условие возвращает `True` независимо от регистра символов `'Audi'`, поскольку теперь проверка выполняется без учета регистра. Метод `lower()` не изменяет значения, которое изначально хранилось в `car`, так что сравнение не отражается на исходной переменной:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
>>> car
'Audi'
```

Сначала строка `'Audi'` сохраняется в переменной `car`. Затем значение `car` приводится к нижнему регистру и сравнивается со значением строки `'audi'`, также записанным в нижнем регистре. Две строки совпадают, поэтому Python возвращает `True`. Вывод показывает, что значение, хранящееся в `car`, не изменилось при вызове метода `lower()`.

Сайты устанавливают определенные правила для данных, вводимых пользователями подобным образом. Например, он может использовать проверку условия, чтобы убедиться в том, что имя каждого пользователя уникально (а не совпадает с именем другого пользователя, отличаясь от него только регистром символов). Когда кто-то указывает новое имя пользователя, оно преобразуется в нижний регистр и сравнивается с версиями всех существующих имен в нижнем регистре. Во время такой проверки имя `'John'` будет отклонено, если в системе уже используется любая разновидность `'john'`.

Проверка неравенства

Если вы хотите проверить, что два значения различны, используйте комбинацию из восклицательного знака и знака равенства (`!=`) — *оператор неравенства* (inequality operator). Чтобы познакомиться с ним, мы воспользуемся другим оператором `if`. В переменной хранится заказанная начинка (`topping`) к пицце; если клиент не заказал анчоусы (`anchovies`), то программа выводит сообщение:

toppings.py

```
requested_topping = 'mushrooms'

if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

Код сравнивает значение `requested_topping` со значением `'anchovies'`. Если эти два значения не равны, то Python возвращает `True` и выполняет код, следующий за оператором `if`. Если равны, то возвращает `False` и не выполняет этот код.

Значение `requested_topping` отличается от `'anchovies'`, поэтому функция `print()` будет выполнена:

```
Hold the anchovies!
```

В большинстве условных выражений, которые вы будете использовать в программах, будет проверяться равенство, но иногда проверка неравенства оказывается более эффективной.

Сравнения чисел

Проверка числовых значений достаточно проста. Например, следующий код проверяет, что переменная `age` равна 18:

```
>>> age = 18
>>> age == 18
True
```

Можно проверить условие неравенства двух чисел. Например, следующий код выводит сообщение, если значение переменной `answer` отличается от ожидаемого:

magic_number.py

```
answer = 17
if answer != 42:
    print("That is not the correct answer. Please try again!")
```

Условие выполняется, поскольку значение `answer` (17) не равно 42. Так как условие истинно, блок с отступом выполняется:

```
That is not the correct answer. Please try again!
```

В условные операторы также можно добавлять всевозможные математические сравнения: меньше, меньше или равно, больше, больше или равно:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Все эти математические сравнения могут использоваться в операторах `if`, что повышает точность формулировки интересующих вас условий.

Проверка нескольких условий

Иногда необходимо проверить несколько условий одновременно. Например, в одних случаях выполнение действия требует, чтобы истинными были сразу два условия; в других достаточно того, чтобы истинным было хотя бы одно из них. Ключевые слова `and` и `or` помогут вам в подобных ситуациях.

Ключевое слово *and* для проверки нескольких условий

Чтобы проверить, истинны ли два условия, объедините их ключевым словом `and`; если истинны, то и все выражение тоже истинно. Если хотя бы одно (или оба) условие ложно, то и результат всего выражения равен `False`.

Например, чтобы убедиться в том, что каждому из двух человек больше 21 года, используйте следующую проверку:

```
>>> age_0 = 22
>>> age_1 = 18
❶ >>> age_0 >= 21 and age_1 >= 21
False
❷ >>> age_1 = 22
>>> age_0 >= 21 and age_1 >= 21
True
```

В коде определяются две переменные, `age_0` и `age_1`. Затем программа проверяет, что оба значения равны 21 или более ❶. Левое условие выполняется, а правое нет, поэтому все условное выражение дает результат `False`. Затем переменной `age_1` присваивается значение 22 ❷. Теперь значение `age_1` больше 21; обе проверки проходят, а все условное выражение дает результат `True`.

Чтобы улучшить читабельность кода, отдельные условия можно заключить в круглые скобки, но это необязательно. При наличии круглых скобок проверка может выглядеть так:

```
(age_0 >= 21) and (age_1 >= 21)
```

Ключевое слово *or* для проверки нескольких условий

Ключевое слово *or* тоже позволяет проверить несколько условий, но результат общей проверки является истинным в том случае, когда истинно хотя бы одно или оба условия. Ложным результат становится только в том случае, если оба отдельных условия ложны.

Вернемся к примеру с возрастом, однако на этот раз проверим, что хотя бы одна из двух переменных больше 21:

```
>>> age_0 = 22
>>> age_1 = 18
❶ >>> age_0 >= 21 or age_1 >= 21
True
❷ >>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Как и в предыдущем случае, сначала определяются две переменные. Условие для *age_0* истинно ❶, поэтому все выражение дает результат *True*. Затем значение *age_0* уменьшается до 18. При проверке оба условия оказываются ложными, и общий результат всего выражения тоже *False* ❷.

Проверка вхождения значений в список

Иногда бывает важно проверить, содержит ли список некое значение, прежде чем выполнять действие. Например, перед завершением регистрации нового пользователя на сайте можно проверить, существует ли его имя в списке имен действующих пользователей. Или в картографическом проекте можно определить, входит ли передаваемое место в список известных мест на карте.

Чтобы узнать, присутствует ли заданное значение в списке, воспользуйтесь ключевым словом *in*. Допустим, вы пишете программу для пиццерии. Вы создали список начинок, заказанных клиентом, и хотите проверить, входят ли некоторые начинки в этот список.

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
>>> 'mushrooms' in requested_toppings
True
>>> 'pepperoni' in requested_toppings
False
```

Благодаря ключевому слову `in` Python получает указание проверить, входят ли значения `'mushrooms'` и `'pepperoni'` в список `requested_toppings`. Этот прием весьма полезен, поскольку вы можете создать список значений, важных для вашей программы, а затем легко проверить, есть ли проверяемое значение в списке.

Проверка отсутствия значения в списке

В других случаях программа должна убедиться в том, что значение не входит в список. Для этого используется ключевое слово `not`. Для примера рассмотрим список пользователей, которым запрещено писать комментарии на форуме. Прежде чем разрешить пользователю отправку комментария, можно проверить, не был ли этот человек добавлен в черный список:

banned_users.py

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'
```

```
if user not in banned_users:
    print(f"{user.title()}, you can post a response if you wish.")
```

Оператор `if` читается достаточно четко: если пользователь не входит в черный список `banned_users`, то Python возвращает `True` и выполняет строку с отступом.

Пользователь `'marie'` не входит в этот список, поэтому программа выводит соответствующее сообщение:

```
Marie, you can post a response if you wish.
```

Логические выражения

В процессе изучения программирования вы рано или поздно услышите термин «*логическое выражение*» (Boolean expression). По сути, это всего лишь другое название для проверки условия. *Логическое значение* (Boolean value) равно `True` или `False`, как и результат условного выражения после его вычисления.

Логические значения часто используются для проверки некоторых условий — например, запущена ли компьютерная игра или разрешено ли пользователю редактирование некой информации на сайте:

```
game_active = True
can_edit = False
```

Логические значения предоставляют эффективные средства для контроля состояния программы или определенного условия, играющего важную роль в вашей программе.

УПРАЖНЕНИЯ

5.1. Проверка условий. Напишите последовательность условий. Выведите описание каждой проверки и ваш прогноз относительно ее результата. Код должен выглядеть примерно так:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

- Внимательно просмотрите результаты. Убедитесь в том, что понимаете, почему результат каждой строки равен True или False.
- Создайте как минимум 10 условий. Не менее пяти одних должны давать результат True, а не менее пяти других — результат False.

5.2. Больше проверок условий. Количество условий не ограничивается десятью. Попробуйте написать другие условия и добавить их в файл `conditional_tests.py`. Программа должна выдавать по крайней мере один истинный и один ложный результат для следующих видов проверок:

- проверка равенства и неравенства строк;
- проверки с использованием метода `lower()`;
- числовые проверки равенства и неравенства, условий «больше», «меньше», «больше или равно», «меньше или равно»;
- проверки с помощью ключевых слов `and` и `or`;
- проверка вхождения элемента в список;
- проверка отсутствия элемента в списке.

Использование операторов if

Когда вы поймете, как работают проверки условий, можете переходить к написанию операторов `if`. Существует несколько их разновидностей, и выбор варианта зависит от количества проверяемых условий. Примеры операторов `if` уже встречались выше, когда обсуждались проверки условий, но сейчас мы рассмотрим эту тему более подробно.

Простые операторы if

Простейшая форма оператора `if` состоит из одного условия и одного действия:

```
if проверка_условия:
    действие
```

В первой строке размещается условие, а в блоке с отступом — практически любое действие. Если условие истинно, то Python выполняет код в блоке после оператора `if`, а если ложно, то этот код игнорируется.

Допустим, имеется переменная, представляющая возраст человека. Следующий код проверяет, достаточен ли этот возраст для голосования:

voting.py

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
```

Python проверяет, больше или равно 18 значение переменной `age`. В таком случае выполняется вызов функции `print()` в строке с отступом:

```
You are old enough to vote!
```

Отступы в операторах `if` играют ту же роль, что и в циклах `for`. Если условие истинно, то все строки с отступом после оператора `if` выполняются, а если ложно — весь блок с отступом игнорируется.

Блок оператора `if` может содержать сколько угодно строк. Добавим еще одну строку для вывода дополнительного сообщения в том случае, если возраст достаточен для голосования:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

Условие выполняется, а оба вызова функции `print()` имеют отступ, поэтому выводятся оба сообщения:

```
You are old enough to vote!
Have you registered to vote yet?
```

Если значение `age` меньше 18, то программа ничего не выводит.

Операторы *if-else*

Часто в программе необходимо выполнить разные действия в зависимости от того, истинно условие или ложно. Синтаксис `if-else` делает это возможным. Блок `if-else` в целом похож на оператор `if`, но оператор `else` определяет действие или набор действий, выполняемых при неудачной проверке.

В следующем примере выводится то же сообщение, которое выводилось ранее, если возраст достаточен для голосования, однако на этот раз при любом другом возрасте выдается другое сообщение:

```
age = 17
❶ if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
❷ else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

Если условие ❶ истинно, то выполняется первый блок с вызовами функции `print()`. Если ложно, то выполняется блок `else` ❷. Так как значение `age` на этот раз меньше 18, условие оказывается ложным и выполняется код в блоке `else`:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

Этот код работает, поскольку существуют всего две возможные ситуации: возраст либо достаточен для голосования, либо нет. Структура `if-else` хорошо подходит для тех ситуаций, в которых Python всегда выполняет только одно из двух возможных действий. В подобных простых цепочках `if-else` всегда выполняется одно из двух возможных действий.

Цепочки `if-elif-else`

Нередко в программе требуется проверять несколько возможных ситуаций; для таких ситуаций в Python предусмотрен синтаксис `if-elif-else`. Python выполняет только один блок в цепочке `if-elif-else`. Все условия проверяются по порядку до тех пор, пока одно из них не даст истинного результата. Далее выполняется код, следующий за этим условием, а все остальные проверки Python пропускает.

Во многих реальных ситуациях существует несколько возможных результатов. Представьте парк аттракционов, который взимает разную плату за вход с разных возрастных групп:

- для посетителей младше 4 лет вход бесплатный;
- для посетителей от 4 до 18 лет билет стоит 25 долларов;
- для посетителей от 18 лет и старше билет стоит 40 долларов.

Как использовать оператор `if` для определения платы за вход? Следующий код выясняет, к какой возрастной категории относится посетитель, и выводит сообщение со стоимостью билета:

amusement_park.py

```
age = 12
❶ if age < 4:
    print("Your admission cost is $0.")
❷ elif age < 18:
    print("Your admission cost is $25.")
❸ else:
    print("Your admission cost is $40.")
```

Условие `if` ❶ проверяет, что возраст посетителя меньше 4 лет. Если условие истинно, то программа выводит соответствующее сообщение и Python пропускает остальные проверки. Строка `elif` ❷ в действительности является еще одной проверкой `if`, которая выполняется только в том случае, если предыдущая проверка завершилась неудачей. В этом месте цепочки известно, что возраст посетителя не меньше 4 лет, поскольку первое условие было ложным. Если посетителю меньше 18 лет, то программа выводит соответствующее сообщение и Python пропускает блок `else`. Если ложны оба условия — `if` и `elif`, то Python выполняет код в блоке `else` ❸.

В данном примере условие `if` ❶ дает ложный результат, поэтому его блок не выполняется. Однако условие `elif` оказывается истинным (12 меньше 18), поэтому код будет выполнен. Вывод состоит из одного сообщения с ценой билета:

```
Your admission cost is $25.
```

При любом значении возраста больше 17 первые два условия ложны. В таких ситуациях блок `else` будет выполнен и цена билета составит 40 долларов.

Вместо того чтобы выводить сообщение с ценой билета в блоках `if-elif-else`, лучше использовать другое, более компактное решение: присвоить цену в цепочке `if-elif-else`, а затем добавить один вызов функции `print()` после выполнения цепочки:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
else:
    price = 40

print(f"Your admission cost is ${price}.")
```

Строки с отступами присваивают значение `price` в зависимости от значения `age`, как и в предыдущем примере. После присваивания цены в цепочке `if-elif-else` отдельный вызов функции `print()` без отступа использует это значение для вывода сообщения с ценой билета.

Этот пример выводит тот же результат, что и предыдущий, но цепочка `if-elif-else` имеет более узкую специализацию. Вместо того чтобы определять цену и выводить сообщения, она просто определяет цену билета. Помимо повышения эффективности, этот код имеет еще одно преимущество: его легче изменить. Чтобы изменить текст выходного сообщения, достаточно отредактировать всего один вызов функции `print()` вместо трех разных вызовов.

Серии блоков *elif*

Код может содержать сколько угодно блоков `elif`. Например, если парк аттракционов введет особую скидку для пожилых посетителей, то вы можете добавить в свой код еще одну проверку для определения того, распространяется ли скидка на текущего посетителя. Допустим, посетители в возрасте 65 и выше платят половину обычной цены билета, или 20 долларов:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
else:
    price = 20

print(f"Your admission cost is ${price}.")
```

Большая часть кода осталась неизменной. Второй блок `elif` теперь проверяет, что посетителю меньше 65 лет, прежде чем назначить ему полную цену билета в 40 долларов. Обратите внимание: значение, присвоенное в блоке `else`, должно быть заменено на 20 долларов, поскольку в этот блок попадают только посетители в возрасте 65 лет и старше.

Исключение блока *else*

Python не требует, чтобы цепочка `if-elif` непременно завершалась блоком `else`. В одних случаях этот блок удобен; в других лучше использовать дополнительный оператор `elif` для обработки конкретного условия:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
elif age >= 65:
    price = 20

print(f"Your admission cost is ${price}.")
```

Последний блок `elif` назначает цену 20 долларов, если возраст посетителя — 65 лет и старше; смысл такого кода более понятен, чем у обобщенного блока `else`.

Благодаря такому изменению выполнение каждого блока возможно только при истинности конкретного условия.

Блок `else` «универсален»: он обрабатывает все условия, не подходящие ни под одну конкретную проверку `if` или `elif`, причем в эту категорию иногда могут попасть недействительные или даже вредоносные данные. Если у вас есть завершающее конкретное условие, то лучше используйте завершающий блок `elif` и опустите блок `else`. В этом случае вы можете быть уверены в том, что ваш код будет выполняться только в правильных условиях.

Проверка нескольких условий

Цепочки `if-elif-else` эффективны, но подходят только в том случае, если истинным должно быть лишь одно условие. Как только Python находит выполняющееся условие, все остальные проверки пропускаются. Такое поведение достаточно эффективно, поскольку позволяет проверить одно конкретное условие.

Но иногда бывает важно проверить *все* условия, представляющие интерес. В таких случаях следует применять серии простых операторов `if` без блоков `elif` или `else`. Такое решение уместно, когда истинными могут быть сразу несколько условий и вы хотите отреагировать на все истинные условия.

Вернемся к примеру с пиццей. Если кто-то закажет пиццу с двумя начинками, то программа должна обработать обе:

toppings.py

```
requested_toppings = ['mushrooms', 'extra cheese']
```

```
if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
❶ if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

Обработка начинается со списка, содержащего заказанные начинки. Первый оператор `if` проверяет, хочет ли человек добавить в пиццу грибы. Если да, то выводится сообщение, подтверждающее наличие грибов. Проверка на добавление пепперони ❶ реализована с помощью еще одного простого оператора `if`, а не `elif` или `else`, поэтому данное условие будет проверяться независимо от того, было предыдущее условие истинным или ложным. Последний оператор `if` проверяет, была ли заказана дополнительная порция сыра, независимо от результата первых двух проверок. Эти три независимых условия проверяются при каждом выполнении программы.

В этом коде проверяются все возможные варианты начинок, поэтому в заказ будут добавлены две начинки из трех:

```
Adding mushrooms.  
Adding extra cheese.
```

```
Finished making your pizza!
```

Если бы в программе использовался блок `if-elif-else`, то код функционировал бы неправильно, поскольку прерывал бы работу после обнаружения первого истинного условия. Вот как это выглядело бы:

```
requested_toppings = ['mushrooms', 'extra cheese']  
  
if 'mushrooms' in requested_toppings:  
    print("Adding mushrooms.")  
elif 'pepperoni' in requested_toppings:  
    print("Adding pepperoni.")  
elif 'extra cheese' in requested_toppings:  
    print("Adding extra cheese.")  
  
print("\nFinished making your pizza!")
```

Первое же проверяемое условие (для `'mushrooms'`) оказывается истинным, поэтому в пиццу добавляются грибы. Однако значения `'extra cheese'` и `'pepperoni'` не проверяются никогда, поскольку в цепочках `if-elif-else` после обнаружения первого истинного условия все остальные пропускаются. В результате в пиццу будет добавлена только первая из заказанных начинок:

```
Adding mushrooms.
```

```
Finished making your pizza!
```

Итак, если вы хотите, чтобы в программе выполнялся только один блок кода, — используйте цепочку `if-elif-else`. Выполнить же несколько блоков позволяет серия независимых операторов `if`.

УПРАЖНЕНИЯ

5.3. Цвета пришельцев 1. Представьте, что в вашей компьютерной игре только что был подбит корабль пришельцев. Создайте переменную `alien_color` и присвойте ей значение `'green'`, `'yellow'` или `'red'`.

- Напишите оператор `if` для проверки того, что переменная содержит значение `'green'`. Если условие истинно, то выведите сообщение о том, что игрок только что заработал 5 очков.

- Напишите одну версию программы, в которой условие `if` выполняется, и другую, в которой оно не выполняется. (Во второй версии никакое сообщение выводиться не должно.)

5.4. Цвета пришельцев 2. Выберите цвет, как это было сделано в упражнении 5.3, и напишите цепочку `if-else`.

- Напишите оператор `if` для проверки того, что переменная содержит значение `'green'`. Если условие истинно, то выведите сообщение о том, что игрок только что заработал 5 очков.
- Если переменная содержит любое другое значение, то выведите сообщение о том, что игрок только что заработал 10 очков.
- Напишите одну версию программы, в которой выполняется блок `if`, и другую, в которой выполняется блок `else`.

5.5. Цвета пришельцев 3. Преобразуйте цепочку `if-else` из упражнения 5.4 в цепочку `if-elif-else`.

- Если переменная содержит значение `'green'`, выведите сообщение о том, что игрок только что заработал 5 очков.
- Если переменная содержит значение `'yellow'`, выведите сообщение о том, что игрок только что заработал 10 очков.
- Если переменная содержит значение `'red'`, выведите сообщение о том, что игрок только что заработал 15 очков.
- Напишите три версии программы и проследите за тем, чтобы для каждого цвета пришельца выводилось соответствующее сообщение.

5.6. Периоды жизни. Напишите цепочку `if-elif-else` для определения периода жизни человека. Присвойте значение переменной `age`, а затем выведите сообщение, в котором указывается тот или иной период:

- если значение меньше 2 — «младенец»;
- если значение больше или равно 2, но меньше 4 — «малыш»;
- если значение больше или равно 4, но меньше 13 — «ребенок»;
- если значение больше или равно 13, но меньше 20 — «подросток»;
- если значение больше или равно 20, но меньше 65 — «взрослый»;
- если значение больше или равно 65 — «пожилой человек».

5.7. Любимый фрукт. Составьте список своих любимых фруктов. Напишите серию независимых операторов `if` для проверки того, присутствуют ли некоторые фрукты в списке.

- Создайте список трех своих любимых фруктов и назовите его `favorite_fruits`.
- Напишите пять операторов `if`. Каждый должен проверять, входит ли определенный тип фрукта в список. Если входит, то блок `if` должен выводить сообщение вида «Вы очень любите бананы!»

Использование операторов if со списками

Объединение операторов if со списками открывает ряд интересных возможностей. Например, вы можете отслеживать специальные значения, которые нуждаются в особой обработке по сравнению с другими значениями в списке, или эффективно управлять изменяющимися условиями — например, наличием некоторых блюд в ресторане. Кроме того, объединение операторов if со списками помогает продемонстрировать, что ваш код корректно работает во всех возможных ситуациях.

Проверка специальных значений

Эта глава началась с простого примера, показывающего, как обрабатывать особые значения (такие как 'bmw'), которые должны выводиться в другом формате по сравнению с другими значениями в списке. Теперь, когда вы лучше разбираетесь в проверках условий и операторах if, более подробно рассмотрим процесс поиска и обработки особых значений в списке.

Вернемся к примеру с пиццерией. Программа выводит сообщение каждый раз, когда в пиццу добавляется начинка в процессе приготовления. Код этого действия можно записать чрезвычайно эффективно: создать список начинок, заказанных клиентом, и использовать цикл для перебора всех заказанных по мере их добавления в пиццу:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']
```

```
for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")
```

```
print("\nFinished making your pizza!")
```

Вывод достаточно тривиален, поэтому код сводится к простому циклу for:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.
```

```
Finished making your pizza!
```

А если в пиццерии вдруг кончится зеленый перец? Оператор if в цикле for может правильно обработать эту ситуацию:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']
```

```
for requested_topping in requested_toppings:
    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    else:
        print(f"Adding {requested_topping}.")
```

```
print("\nFinished making your pizza!")
```

На этот раз программа проверяет каждый заказанный элемент перед добавлением его в пиццу. Оператор `if` проверяет, заказал ли клиент зеленый перец, и если заказал — выводит сообщение о том, что этой начинки нет. Блок `else` гарантирует, что все другие начинки будут включены в заказ.

Из выходных данных видно, что все заказанные начинки обрабатываются правильно:

```
Adding mushrooms.  
Sorry, we are out of green peppers right now.  
Adding extra cheese.
```

```
Finished making your pizza!
```

Проверка наличия содержимого в списке

Для всех списков, с которыми мы работали до сих пор, действовало одно простое предположение: мы считали, что в каждом списке есть хотя бы один элемент. Скоро мы предоставим пользователю возможность вводить информацию, хранящуюся в списке, поэтому уже не можем предполагать, что при каждом выполнении цикла `for` в списке есть хотя бы один элемент. В такой ситуации перед выполнением цикла `for` будет полезно проверить содержимое списка.

Выясним, есть ли элементы в списке заказанных начинок, перед изготовлением пиццы. Если список пуст, то программа предлагает пользователю подтвердить, что он хочет обычную пиццу. Если в списке что-то есть, то пицца готовится так же, как в предыдущих примерах:

```
requested_toppings = []  
  
if requested_toppings:  
    for requested_topping in requested_toppings:  
        print(f"Adding {requested_topping}.")  
    print("\nFinished making your pizza!")  
else:  
    print("Are you sure you want a plain pizza?")
```

На этот раз мы начинаем с пустого списка заказанных начинок. Вместо того чтобы сразу переходить к циклу `for`, программа сначала выполняет проверку. Когда имя списка используется в условии `if`, Python возвращает `True`, если список содержит хотя бы один элемент; в случае пустого списка возвращается значение `False`. Если `requested_toppings` проходит проверку условия, то выполняется тот же цикл `for`, который мы использовали в предыдущем примере. Если же условие ложно, то программа выводит сообщение, которое предлагает клиенту подтвердить, действительно ли он хочет получить обычную пиццу без начинок.

В данном примере список пуст, поэтому выводится сообщение:

```
Are you sure you want a plain pizza?
```

Если в списке есть хотя бы один элемент, то в выходные данные добавляется каждая заказанная начинка.

Множественные списки

Посетители могут заказать что угодно, особенно когда речь заходит о начинках пиццы. Что, если клиент захочет добавить в пиццу картофель фри? Списки и операторы `if` позволят вам убедиться в том, что входные данные имеют смысл, прежде чем обрабатывать их.

Проверим наличие нестандартных начинок, прежде чем готовить пиццу. В следующем примере определяются два списка. Первый содержит перечень доступных начинок, а второй — список начинок, заказанных клиентом. На этот раз каждый элемент из `requested_toppings` проверяется по списку доступных начинок перед добавлением в пиццу:

```
available_toppings = ['mushrooms', 'olives', 'green peppers',
                     'pepperoni', 'pineapple', 'extra cheese']

❶ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

for requested_topping in requested_toppings:
❷     if requested_topping in available_toppings:
        print(f"Adding {requested_topping}.")
❸     else:
        print(f"Sorry, we don't have {requested_topping}.")

print("\nFinished making your pizza!")
```

Сначала определяется список доступных начинок. Стоит заметить, что если в пиццерии используется постоянный ассортимент начинок, то список можно реализовать в виде кортежа. Затем создается список начинок, заказанных клиентом. Обратите внимание на необычный заказ `'french fries'` ❶. Далее программа перебирает список заказанных начинок. Внутри цикла она сначала проверяет, что каждая заказанная начинка есть в списке доступных начинок ❷. Если начинка доступна, то добавляется в пиццу. Если заказанная начинка не входит в список, то выполняется блок `else` ❸. Этот блок выводит сообщение о том, что начинка недоступна.

Благодаря этому синтаксису программа выдает четкий, содержательный вывод:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

Всего в нескольких строках кода нам удалось эффективно решить вполне реальную проблему!

УПРАЖНЕНИЯ

5.8. Здравствуйте, админ! Создайте список из пяти и более имен пользователей, содержащий имя 'admin'. Представьте, что пишете код, который выводит приветственное сообщение для каждого пользователя после его входа на сайт. Переберите элементы списка и выведите сообщение для каждого пользователя:

- для пользователя 'admin' выведите особое сообщение: например, «Здравствуйте, admin, хотите просмотреть отчет о состоянии дел?»;
- в остальных случаях выводите универсальное приветствие: например, «Привет, Денис, спасибо, что авторизовался в системе».

5.9. Нет пользователей. Добавьте в программу `hello_admin.py` оператор `if`, который проверит, что список пользователей не пуст.

- Если список пуст, то выведите сообщение «Нам нужно добавить несколько пользователей!»
- Удалите из списка все имена пользователей и убедитесь в том, что программа выводит правильное сообщение.

5.10. Проверка имен пользователей. Выполните следующие действия, чтобы создать программу, имитирующую проверку сайтом уникальности имен пользователей.

- Создайте список `current_users`, содержащий пять и более имен пользователей.
- Создайте список `new_users`, содержащий пять имен пользователей. Убедитесь в том, что одно или два новых имени также присутствуют в списке `current_users`.
- Переберите список `new_users` и проверьте, было ли использовано ранее каждое имя в этом списке. Если да, то выведите сообщение о том, что пользователь должен выбрать новое имя. Если нет, то выведите сообщение о доступности имени.
- Проследите за тем, чтобы сравнение выполнялось без учета регистра символов. Если имя 'John' уже используется, то в регистрации имени 'JOHN' следует отказать. (Для этого необходимо создать копию `current_users`, содержащую версию всех существующих имен пользователей, написанных в нижнем регистре.)

5.11. Порядковые числительные. Порядковые числительные в английском языке заканчиваются суффиксом `th` (кроме `1st`, `2nd` и `3rd`).

- Сохраните числа от 1 до 9 в списке.
- Переберите элементы списка.
- Используйте цепочку `if-elif-else` в цикле для вывода правильного окончания числительного для каждого числа. Программа должна выводить числительные "1st 2nd 3rd 4th 5th 6th 7th 8th 9th", причем каждый результат должен располагаться на отдельной строке.

Оформление операторов if

Во всех примерах этой главы применялись правила форматирования. В PEP 8 приведена только одна рекомендация, касающаяся проверки условий: до и после операторов сравнения (такие как `==`, `>=`, `<=` и т. д.) ставить одиночные пробелы. Например, запись

```
if age < 4:
```

лучше, чем:

```
if age<4:
```

Пробелы не влияют на интерпретацию вашего кода Python; они только упрощают чтение кода для вас и других разработчиков.

УПРАЖНЕНИЯ

5.12. Форматирование операторов if. Проанализируйте программы, написанные в этой главе, и проверьте, правильно ли вы оформляли проверку условий.

5.13. Ваши идеи. К этому моменту вы уже стали более квалифицированным программистом, чем когда начинали читать эту книгу. Теперь вы лучше представляете, как в программах моделируются явления реального мира, и сможете придумать задачи, которые будут решаться в ваших программах. Запишите несколько задач, которые вам хотелось бы решить по мере роста вашего профессионального мастерства. Может быть, это какие-то компьютерные игры, задачи анализа наборов данных или веб-приложения?

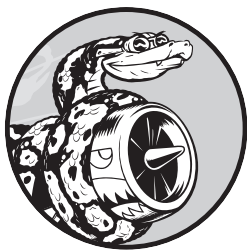
Резюме

В этой главе вы научились писать условия, результатом которых всегда является логическое значение (`True` или `False`). Вы узнали, как писать простые операторы `if`, а также цепочки `if-else` и `if-elif-else`. Вы начали использовать эти структуры для выявления конкретных условий, которые необходимо проверить, и собственно проверки этих условий в ваших программах. Вы научились обрабатывать определенные элементы списка иначе, чем остальные, сохраняя эффективность циклов `for`. Кроме того, вы узнали новые рекомендации по форматированию кода Python, которые упрощают чтение и понимание более сложных программ.

В главе 6 вы познакомитесь со *словарями* Python. Словарь отчасти напоминает список, но позволяет связывать разные виды информации. Вы научитесь создавать словари, перебирать их элементы, использовать их в сочетании со списками и операторами `if`. Словари помогут вам моделировать еще больше реальных ситуаций.

6

Словари



В этой главе речь пойдет о словарях — структурах данных, предназначенных для объединения взаимосвязанной информации. Вы узнаете, как получить доступ к информации, хранящейся в словаре, и как изменить ее. Объем данных в словаре практически безграничен, поэтому мы рассмотрим средства перебора данных в словарях. Кроме того, вы научитесь использовать словари, вложенные в списки и другие словари, а также списки, вложенные в словари.

Операции со словарями позволяют более точно моделировать всевозможные реальные объекты. Вы узнаете, как создать словарь, описывающий человека, и сохранить в нем сколько угодно информации об этом человеке. В словаре могут храниться данные об имени, возрасте, месте жительства, профессии и любых других атрибутах. Вы узнаете, как сохранить любые два вида информации, способные образовать пары: список слов и их значений, список имен людей и их любимых чисел, список гор и их высот и т. д.

Простой словарь

Возьмем игру с пришельцами, которые имеют разные цвета и приносят разное количество очков игроку. В следующем простом словаре хранится информация об одном конкретном пришельце:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
```

```
print(alien_0['color'])
```

```
print(alien_0['points'])
```

В словаре `alien_0` хранятся два атрибута: цвет (`color`) и количество очков (`points`). Последние две строки кода считывают эту информацию из словаря и выводят ее на экран:

```
green
5
```

Работа со словарями, как и большинство других новых концепций, требует определенного опыта. Немного поработав со словарями, вы увидите, насколько они эффективны при моделировании реальных ситуаций.

Работа со словарями

Словарь в языке Python представляет собой совокупность пар «ключ — значение». Каждый ключ связывается с неким значением, и программа может получить значение, связанное с заданным ключом. Значением могут быть число, строка, список и даже другой словарь. Собственно, *любой* объект, создаваемый в программе Python, может стать значением в словаре.

В Python словарь заключается в фигурные скобки {}, в которых приводится последовательность пар «ключ — значение», как в предыдущем примере:

```
alien_0 = {'color': 'green', 'points': 5}
```

Пара «*ключ — значение*» представляет данные, связанные друг с другом. Если вы укажете ключ, то Python вернет связанное с ним значение. Ключ отделяется от значения двоеточием, а пары разделяются запятыми. В словаре может храниться любое количество пар «ключ — значение».

Простейший словарь содержит ровно одну пару «ключ — значение», как в следующей измененной версии словаря `alien_0`:

```
alien_0 = {'color': 'green'}
```

В этом словаре хранится ровно один фрагмент информации о пришельце `alien_0`, а именно его цвет. Строка `'color'` является ключом в словаре; с этим ключом связано значение `'green'`.

Обращение к значениям в словаре

Чтобы получить значение, связанное с ключом, укажите имя словаря, а затем ключ в квадратных скобках:

alien.py

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

Эта конструкция возвращает значение, связанное с ключом `'color'`, из словаря `alien_0`:

```
green
```

Количество пар «ключ — значение» в словаре не ограничено. Например, вот как выглядит исходный словарь `alien_0` с двумя парами «ключ — значение»:

```
alien_0 = {'color': 'green', 'points': 5}
```

Теперь программа может получить значение, связанное с любым из ключей в `alien_0`: `color` или `points`. Если игрок сбивает корабль пришельца, то для получения заработанных очков может использоваться код следующего вида:

```
alien_0 = {'color': 'green', 'points': 5}

new_points = alien_0['points']
print(f"You just earned {new_points} points!")
```

После того как словарь будет определен, интерпретатор извлекает из словаря значение, связанное с ключом `'points'`. Затем оно сохраняется в переменной `new_points`. Последняя строка преобразует целое значение в строку и выводит сообщение с указанием количества заработанных очков:

```
You just earned 5 points!
```

Если этот код будет выполняться каждый раз, когда игрок сбивает очередной корабль пришельца, то программа будет получать правильное количество очков.

Добавление новых пар «ключ — значение»

Словари относятся к динамическим структурам данных: в словарь можно в любой момент добавлять новые пары «ключ — значение». Для этого указывается имя словаря, за которым в квадратных скобках следует новый ключ с новым значением.

Добавим в словарь `alien_0` еще два атрибута: координаты x и y для вывода изображения пришельца в определенной позиции экрана. Допустим, пришелец должен отображаться у левого края экрана, в 25 пикселах от верхнего края. Система экранных координат обычно располагается в левом верхнем углу, поэтому для размещения пришельца у левого края координата x должна быть равна 0, а координата y — 25:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)
```



```
alien_0['x_position'] = 0
alien_0['y_position'] = 25
print(alien_0)
```

Программа начинается с определения того же словаря, с которым мы уже работали ранее. После этого выводится «снимок» текущего состояния словаря. Затем в словарь добавляется новая пара «ключ — значение»: ключ `'x_position'` и значение `0`. То же самое делается для ключа `'y_position'`. При выводе измененного словаря мы видим две дополнительные пары «ключ — значение»:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'x_position': 0, 'y_position': 25}
```

Окончательная версия словаря содержит четыре пары «ключ — значение». Первые две определяют цвет и количество очков, а вторые две — координаты.

Словари сохраняют исходный порядок добавления пар «ключ — значение». Когда вы выводите словарь или перебираете его элементы, то видите элементы в том порядке, в котором они добавлялись в словарь.

Создание пустого словаря

В некоторых ситуациях бывает удобно (или даже необходимо) начать с пустого словаря, а затем добавлять в него новые элементы. Чтобы начать заполнение пустого словаря, определите словарь с пустой парой фигурных скобок, а затем добавляйте новые пары «ключ — значение» (каждая пара в отдельной строке). Например, вот как создается словарь `alien_0`:

```
alien.py
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Программа определяет пустой словарь `alien_0`, после чего добавляет в него значения для цвета и количества очков. В результате создается словарь, который использовался в предыдущих примерах:

```
{'color': 'green', 'points': 5}
```

Обычно пустые словари используются при хранении данных, введенных пользователем, или при написании кода, автоматически генерирующего большое количество пар «ключ — значение».

Изменение значений в словаре

Чтобы изменить значение в словаре, укажите его имя с ключом в квадратных скобках, а затем новое значение, которое должно быть связано с этим ключом. Допустим, в процессе игры цвет пришельца меняется с зеленого на желтый:

alien.py

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")

alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")
```

Сначала определяется словарь `alien_0`, который содержит данные только о цвете пришельца; затем значение, связанное с ключом `'color'`, меняется на `'yellow'`. Из выходных данных видно, что цвет пришельца действительно сменился с зеленого на желтый:

```
The alien is green.
The alien is now yellow.
```

Рассмотрим более интересный пример: отслеживание позиции пришельца, который может двигаться с разными скоростями. Мы сохраним значение, представляющее текущую скорость пришельца, и используем его для определения величины горизонтального смещения:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print(f"Original position: {alien_0['x_position']}")
```

```
# Пришелец перемещается вправо.
# Вычисляем величину смещения на основании текущей скорости.
```

```
❶ if alien_0['speed'] == 'slow':
    x_increment = 1
    elif alien_0['speed'] == 'medium':
        x_increment = 2
    else:
        # Пришелец движется быстро.
        x_increment = 3
```

```
# Новая позиция равна сумме старой позиции и приращения.
```

```
❷ alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']}")
```

Сначала определяется словарь с исходной позицией (координаты x и y) и скоростью `'medium'`. Значения цвета и количества очков для простоты опущены, но с ними этот пример работал бы точно так же. Кроме того, выводится исходное значение `x_position`.

Цепочка `if-elif-else` определяет, на какое расстояние пришелец должен переместиться вправо; полученное значение сохраняется в переменной `x_increment` ❶. Если пришелец движется медленно (`'slow'`), то перемещается на одну единицу вправо; при средней скорости (`'medium'`) перемещается на две единицы вправо; наконец, при высокой скорости (`'fast'`) перемещается на три единицы вправо. Вычисленное смещение прибавляется к значению `x_position` ❷, а результат сохраняется в словаре с ключом `x_position`.

Позиция пришельца со средней скоростью смещается на две единицы:

```
Original x-position: 0
New x-position: 2
```

Получается, что изменение одного значения в словаре изменяет все поведение пришельца. Например, чтобы превратить пришельца, движущегося со средней скоростью, в быстрого, добавьте следующую строку:

```
alien_0['speed'] = fast
```

При следующем выполнении кода блок `if-elif-else` присвоит `x_increment` большее значение.

Удаление пар «ключ — значение»

Когда информация, хранящаяся в словаре, перестает быть ненужной, пару «ключ — значение» можно полностью удалить с помощью оператора `del`. При вызове достаточно передать имя словаря и удаляемый ключ.

Так, в следующем примере из словаря `alien_0` удаляется ключ `'points'` вместе со значением:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)
```

```
❶ del alien_0['points']
print(alien_0)
```

Оператор `del` ❶ сообщает Python о необходимости удалить из словаря `alien_0` ключ `'points'` и связанное с ним значение. Из вывода видно, что ключ `'points'` и его значение 5 исчезли из словаря, но остальные данные остались без изменений:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

ПРИМЕЧАНИЕ

Учтите, что удаление пары «ключ — значение» отменить уже не удастся.

Словарь с однотипными объектами

В предыдущем примере в словаре сохранялась разнообразная информация об одном объекте (пришельце из компьютерной игры). Словарь также может использоваться для хранения одного вида информации о многих объектах. Допустим, вы хотите провести опрос среди коллег и узнать их любимый язык программирования. Результаты простого опроса удобно сохранить в словаре:

favorite_languages.py

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}
```

Пары в данном словаре разбиты по строкам. Ключами являются имена участников опроса, а значениями — выбранные ими языки. Если вы знаете, что определение словаря потребует нескольких строк, то нажмите клавишу **Enter** после ввода открывающей фигурной скобки. Добавьте в следующую строку отступ на один уровень (четыре пробела) и запишите первую пару «ключ — значение», поставив за ней запятую. После этого при нажатии **Enter** ваш редактор кода будет автоматически добавлять во все последующие пары такой же отступ, как у первой.

Завершив определение словаря, добавьте закрывающую фигурную скобку в новой строке после последней пары «ключ — значение» и добавьте в нее отступ на один уровень, чтобы она была выровнена по ключам. За последней парой также рекомендуется поставить запятую, чтобы при необходимости вы смогли легко добавить новую пару «ключ — значение» в следующей строке.

ПРИМЕЧАНИЕ

Во многих редакторах предусмотрены функции, упрощающие форматирование расширенных списков и словарей в описанном стиле. Существуют и другие распространенные способы форматирования длинных словарей — вы можете столкнуться с ними в вашем редакторе или другом источнике.

Этот словарь позволяет легко определить любимый язык конкретного участника опроса:

favorite_languages.py

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}
```

```
❶ language = favorite_languages['sarah'].title()
print(f"Sarah's favorite language is {language}."
```

Чтобы узнать, какой язык выбран пользователем Sarah, мы запрашиваем следующее значение:

```
favorite_languages['sarah']
```

Этот синтаксис используется для получения соответствующего языка программирования из словаря ❶ и присваивания его переменной `language`. Создание новой переменной существенно упрощает вызов функции `print()`. В выходных данных показывается значение, связанное с ключом:

```
Sarah's favorite language is C.
```

Вы можете использовать тот же синтаксис для работы с данными любого участника опроса, содержащимися в словаре.

Обращение к значениям методом `get()`

Использование синтаксиса с ключом в квадратных скобках для получения интересующего вас значения из словаря имеет один потенциальный недостаток: если запрашиваемого ключа не существует, то вы получите сообщение об ошибке.

Посмотрим, что произойдет при запросе количества очков для пришельца, для которого оно не задано:

alien_no_points.py

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

На экране появляется трассировка с сообщением об ошибке `KeyError`:

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
    ~~~~~^~~~~~
KeyError: 'points'
```

Более общие способы обработки подобных ошибок рассматриваются в главе 10. Конкретно для словарей можно воспользоваться методом `get()` и задать значение по умолчанию, которое будет возвращено при отсутствии в словаре требуемого ключа.

В первом аргументе метода `get()` передается ключ. Во втором необязательном аргументе можно передать значение, которое должно возвращаться при отсутствии ключа:

```
alien_0 = {'color': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

Если ключ `'points'` существует в словаре, то вы получите соответствующее значение; если нет — будет получено значение по умолчанию. В данном случае ключа `'points'` не существует, поэтому вместо ошибки выводится понятное сообщение:

```
No point value assigned.
```

Если есть вероятность, что запрашиваемого ключа не существует, то, возможно, стоит использовать метод `get()` вместо синтаксиса с квадратными скобками.

ПРИМЕЧАНИЕ

Если второй аргумент при вызове `get()` опущен, а ключа не существует, то Python вернет специальное значение `None` — признак того, что значения не существует. Это не ошибка, а специальное значение, указывающее на отсутствие значения. Другие применения `None` описаны в главе 8.

УПРАЖНЕНИЯ

6.1. Персона. Используйте словарь для сохранения информации об известном вам человеке. Сохраните данные о его имени и фамилии, возрасте и городе, в котором он живет. Словарь должен содержать ключи с такими именами, как `first_name`, `last_name`, `age` и `city`. Выведите каждый фрагмент информации, хранящийся в словаре.

6.2. Любимые числа. Используйте словарь для хранения любимых чисел людей. Возьмите пять имен и используйте их как ключи словаря. Придумайте любимое число для каждого человека и сохраните его как значение в словаре. Выведите имя каждого человека и его любимое число. Чтобы задача стала более интересной, опросите нескольких друзей и соберите реальные данные для своей программы.

6.3. Глоссарий. Словари Python могут использоваться для моделирования настоящего словаря (чтобы не создавать путаницы, назовем его глоссарием).

- Вспомните пять терминов из области программирования, которые вы узнали в предыдущих главах. Используйте эти слова как ключи глоссария, а их определения — как значения.
- Выведите каждое слово и его определение в отформатированном виде. Например, вы можете вывести слово, затем двоеточие и определение; или же слово на одной строке, а его определение — с отступом на другой. Используйте символ новой строки (`\n`) для вставки пустых строк между парами «слово-определение» в выходных данных.

Перебор словаря

Словарь Python может содержать как несколько, так и миллионы пар «ключ — значение». Поскольку в словаре могут храниться большие объемы данных, Python предоставляет средства для перебора элементов словаря. Информация может храниться в словарях по-разному, поэтому предусмотрены различные способы перебора. Программа может перебрать все пары «ключ — значение» в словаре, только ключи или только значения.

Перебор всех пар «ключ — значение»

Прежде чем говорить о разных способах перебора, рассмотрим новый словарь, предназначенный для хранения информации о пользователе сайта. В следующем словаре хранятся данные об имени пользователя и его фамилии:

user.py

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
}
```

То, что вы уже узнали в этой главе, позволит вам обратиться к любому отдельному атрибуту `user_0`. Но что, если вы хотите просмотреть *все* данные из словаря этого пользователя? Для этого можно воспользоваться перебором в цикле `for`:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
}

for key, value in user_0.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

Чтобы написать цикл `for` для словаря, необходимо создать имена для двух переменных, в которых будут храниться ключ и значение из каждой пары «ключ — значение». Этим двум переменным можно присвоить любые имена — с короткими однобуквенными именами код будет работать точно так же:

```
for k, v in user_0.items()
```

Вторая половина оператора `for` содержит имя словаря, за которым следует вызов метода `items()`, возвращающий список пар «ключ — значение». Цикл `for` сохраняет компоненты пары в двух указанных переменных. В предыдущем примере мы

используем переменные для вывода каждого ключа `key`, за которым следует связанное значение `value`. Элемент `"\n"` в первом вызове функции `print()` гарантирует, что перед каждой парой «ключ — значение» в выводе будет вставлена пустая строка:

```
Key: username
Value: efermi
```

```
Key: first
Value: enrico
```

```
Key: last
Value: fermi
```

Перебор всех пар «ключ — значение» особенно хорошо работает для таких словарей, которые были показаны в примере программы `favorite_languages.py`, приведенном в подразделе «Словарь с однотипными объектами» ранее в данной главе: то есть для словарей, хранящих один вид информации со многими разными ключами. Перебрав словарь `favorite_languages`, вы получите имя каждого человека и его любимый язык программирования. Ключ всегда содержит имя, а значение — язык программирования, поэтому в цикле вместо имен `key` и `value` используются переменные `name` и `language`. Благодаря такому выбору имен читателю кода будет проще следить за тем, что происходит в цикле:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
}

for name, language in favorite_languages.items():
    print(f"{name.title()}'s favorite language is {language.title()}.")
```

Код дает Python указание перебрать все пары «ключ — значение» в словаре. В процессе перебора ключ сохраняется в переменной `name`, а значение — в переменной `language`. Благодаря этим описательным именам намного проще понять, что делает вызов функции `print()`.

Всего в нескольких строках кода выводится вся информация из опроса:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Rust.
Phil's favorite language is Python.
```

Этот способ перебора точно так же работает и в том случае, если в словаре будут храниться результаты опроса тысяч и даже миллионов людей.

Перебор всех ключей в словаре

Метод `keys()` удобен в тех случаях, когда вы не собираетесь работать со всеми значениями в словаре. Переберем словарь `favorite_languages` и выведем имена всех людей, участвовавших в опросе:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
}
for name in favorite_languages.keys():
    print(name.title())
```

Этот цикл `for` дает Python указание извлечь из словаря `favorite_languages` все ключи и последовательно сохранять их в переменной `name`. В выходных данных представлены имена всех людей, участвовавших в опросе:

```
Jen
Sarah
Edward
Phil
```

На самом деле перебор ключей используется по умолчанию при переборе словаря, поэтому данный код будет работать точно так же, как если бы вы написали

```
for name in favorite_languages:
```

ВМЕСТО

```
for name in favorite_languages.keys():
```

Используйте явный вызов метода `keys()`, если считаете, что он упростит чтение вашего кода, или опустите его при желании.

Чтобы обратиться в цикле к значению, связанному с интересующим вас ключом, используйте текущий ключ. В качестве примера выведем для пары друзей сообщение о выбранном ими языке. Мы переберем имена в словаре, как делали ранее, но когда имя совпадает с именем одного из друзей, программа будет выводить специальное сообщение об их любимом языке:

```
favorite_languages = {
    --нпонуск--
}

friends = ['phil', 'sarah']
for name in favorite_languages.keys():
    print(f"Hi {name.title()}")
```

```
❶ if name in friends:
❷     language = favorite_languages[name].title()
    print(f"\t{name.title()}, I see you love {language}!")
```

Сначала формируется список друзей, для которых должно выводиться сообщение. В цикле выводится имя очередного участника опроса, а затем программа проверяет, входит ли текущее имя в список `friends` ❶. Если да, то мы извлекаем любимый язык этого человека из словаря на основе текущего значения `name` в качестве ключа ❷. Затем выводится специальное приветствие с упоминанием выбранного языка.

Выводятся все имена, но для наших друзей выдается специальное сообщение:

```
Hi Jen.
Hi Sarah.
    Sarah, I see you love C!
Hi Edward.
Hi Phil.
    Phil, I see you love Python!
```

Метод `keys()` также может использоваться для проверки того, участвовал ли конкретный человек в опросе:

```
favorite_languages = {
    --пропуск--
}

if 'erin' not in favorite_languages.keys():
    print("Erin, please take our poll!")
```

Метод `keys()` служит не только для перебора: он возвращает список всех ключей, и оператор `if` просто проверяет, есть ли ключ `'erin'` в списке. Его там нет, поэтому программа выводит сообщение:

```
Erin, please take our poll!
```

Перебор ключей словаря в определенном порядке

Перебор содержимого словаря возвращает элементы в том порядке, в котором они вставлялись. Тем не менее иногда требуется перебрать элементы словаря в другом порядке.

Один из способов получения элементов в определенном порядке основан на сортировке ключей, возвращаемых циклом `for`. Для получения упорядоченной копии ключей можно воспользоваться функцией `sorted()`:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
```

```
'phil': 'python',  
}
```

```
for name in sorted(favorite_languages.keys()):  
    print(f"{name.title()}, thank you for taking the poll.")
```

Этот оператор `for` не отличается от других операторов `for`, если не считать того, что метод `dictionary.keys()` заключен в функцию `sorted()`. Благодаря данной конструкции Python получает указание выдать список всех ключей в словаре и отсортировать его до перебора элементов. В выводе перечислены все пользователи, участвовавшие в опросе, а их имена упорядочены по алфавиту:

```
Edward, thank you for taking the poll.  
Jen, thank you for taking the poll.  
Phil, thank you for taking the poll.  
Sarah, thank you for taking the poll.
```

Перебор всех значений в словаре

Если вас прежде всего интересуют значения, содержащиеся в словаре, то используйте метод `values()` для получения списка значений без ключей. Например, вы хотите просто получить список всех языков, выбранных в опросе, и вас не интересуют имена людей, выбравших каждый язык:

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}
```

```
print("The following languages have been mentioned:")  
for language in favorite_languages.values():  
    print(language.title())
```

Оператор `for` считывает каждое значение из словаря и сохраняет его в переменной `language`. При выводе этих значений будет получен список всех выбранных языков:

```
The following languages have been mentioned:  
Python  
C  
Rust  
Python
```

Значения извлекаются из словаря, не подвергаясь проверке на возможные повторения. В случае небольших словарей это может быть приемлемо, но в опросах с большим количеством респондентов список будет содержать слишком много дубликатов. Чтобы получить список выбранных языков без повторений, можно

воспользоваться *множеством* (set). Оно в целом похоже на список, но все его элементы должны быть уникальными:

```
favorite_languages = {  
    --пропуск--  
}  
  
print("The following languages have been mentioned:")  
for language in set(favorite_languages.values()):  
    print(language.title())
```

Когда список, содержащий дубликаты, заключается в функцию `set()`, Python находит уникальные элементы списка и создает множество из этих элементов. В нашем коде данная функция служит для извлечения уникальных языков из `favorite_languages.values()`.

В результате создается не содержащий дубликатов список языков программирования, упомянутых участниками опроса:

```
The following languages have been mentioned:  
Python  
C  
Rust
```

В ходе дальнейшего изучения Python вы часто будете обнаруживать встроенные возможности языка, которые помогают сделать с данными именно то, что вам требуется.

ПРИМЕЧАНИЕ

Множество можно создать прямо в фигурных скобках, разделяя элементы запятыми:

```
>>> languages = {'python', 'rust', 'python', 'c'}  
>>> languages  
{'rust', 'python', 'c'}
```

Словари легко перепутать с множествами, поскольку обе структуры заключаются в фигурные скобки. Когда вы видите фигурные скобки без пар «ключ — значение», скорее всего, перед вами множество. В отличие от списков и словарей, элементы множеств не хранятся в каком-либо определенном порядке.

УПРАЖНЕНИЯ

6.4. Глоссарий 2. Теперь, когда вы знаете, как перебрать элементы словаря, упростите код из упражнения 6.3, заменив серию вызовов функции `print()` циклом, перебирающим ключи и значения словаря. Когда вы будете уверены в том, что цикл работает, добавьте в глоссарий еще пять терминов Python. При повторном запуске программы новые слова и значения должны быть автоматически добавлены в вывод.

6.5. Реки. Создайте словарь с данными о трех больших реках и странах, по которым протекает каждая из этих рек. Одна из возможных пар «ключ — значение» — 'nile': 'egypt'.

- Используйте цикл для вывода сообщения с упоминанием реки и страны — например, «Нил протекает через Египет».
- Используйте цикл для вывода названия каждой реки, добавленной в словарь.
- Используйте цикл для вывода названия каждой страны, добавленной в словарь.

6.6. Опрос. Возьмите за основу код программы `favorite_languages.py` из подраздела «Словарь с однотипными объектами» данной главы.

- Создайте список людей, которые должны участвовать в опросе по поводу любимого языка программирования. Добавьте имена, которые уже присутствуют в списке, и имена, которых в нем еще нет.
- Переберите список людей, которые должны участвовать в опросе. Если они уже прошли его, то выведите сообщение с благодарностью за участие. Если еще не проходили, то выведите сообщение с предложением принять участие.

Вложение данных

Иногда бывает нужно сохранить множество словарей в списке или сохранить список как значение элемента словаря. Создание сложных структур такого рода называется *вложением* (nesting). Вы можете вложить множество словарей в список, список элементов в словарь или даже словарь в другой словарь. Как наглядно показывают следующие примеры, вложение — чрезвычайно мощный механизм.

Список словарей

Словарь `alien_0` содержит разнообразную информацию об одном пришельце, но в нем нет места для хранения данных о втором пришельце, не говоря уже об армаде пришельцев. Как смоделировать флот вторжения? Например, можно создать список, в котором каждый элемент представляет собой словарь с информацией о пришельце. Например, следующий код создает список с данными о трех пришельцах:

aliens.py

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}
```

```
❶ aliens = [alien_0, alien_1, alien_2]
```

```
for alien in aliens:
    print(alien)
```

Сначала создаются три словаря, каждый из которых представляет отдельного пришельца. Каждый словарь заносится в список `aliens` ❶. Наконец, программа перебирает список и выводит данные о каждом пришельце:

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

Конечно, в реалистичном примере будут использоваться данные о более чем трех пришельцах, которые будут генерироваться автоматически. В следующем примере функция `range()` создает флот из 30 пришельцев:

```
# Создание пустого списка для хранения данных о пришельцах.
aliens = []

# Создание 30 зеленых пришельцев.
❶ for alien_number in range(30):
❷     new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
❸     aliens.append(new_alien)

# Вывод данных о первых пяти пришельцах:
❹ for alien in aliens[:5]:
    print(alien)
print("...")

# Вывод количества созданных пришельцев.
print(f"Total number of aliens: {len(aliens)}")
```

В начале примера список для хранения данных обо всех пришельцах, которые будут созданы, пуст. Функция `range()` ❶ возвращает множество чисел, которое просто сообщает Python, сколько раз должен повторяться цикл. При каждом выполнении цикла создается новый пришелец ❷, который затем добавляется в список `aliens` ❸. Срез используется для вывода данных о первых пяти пришельцах ❹, а затем выводится длина списка (для демонстрации того, что программа действительно сгенерировала весь флот из 30 пришельцев):

```
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
...
```

```
Total number of aliens: 30
```

Все пришельцы обладают одинаковыми характеристиками, но Python рассматривает любого пришельца как отдельный объект, что позволяет изменять атрибуты каждого владельца по отдельности.

Как работать с таким множеством? Представьте, что в этой игре некоторые пришельцы меняют цвет и начинают двигаться быстрее. Когда приходит время

смены цветов, мы можем воспользоваться циклом `for` и оператором `if` для изменения цвета. Например, чтобы превратить первых трех пришельцев в желтых, двигающихся со средней скоростью и приносящих игроку по 10 очков, можно действовать так:

```
# Создание пустого списка для хранения данных о пришельцах.
aliens = []

# Создание 30 зеленых пришельцев.
for alien_number in range(30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Вывод данных о первых пяти пришельцах:
for alien in aliens[:5]:
    print(alien)
print("...")
```

Чтобы изменить первых трех пришельцев, мы перебираем элементы среза, содержащего информацию только о них. В данный момент все пришельцы зеленые ('green'), но так будет не всегда, поэтому мы пишем оператор `if`, который гарантирует, что изменяться будут только зеленые пришельцы. Если пришелец зеленый, то его цвет меняется на желтый ('yellow'), скорость на среднюю ('medium'), а награда увеличивается до 10 очков:

```
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'yellow', 'points': 10, 'speed': 'medium'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
{'color': 'green', 'points': 5, 'speed': 'slow'}
...
```

Цикл можно расширить, добавив блок `elif` для превращения желтых пришельцев в красных — быстрых и приносящих игроку по 15 очков. Мы не станем приводить весь код, а цикл выглядит так:

```
for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10
    elif alien['color'] == 'yellow':
        alien['color'] = 'red'
        alien['speed'] = 'fast'
        alien['points'] = 15
```

Решение с хранением словарей в списке встречается достаточно часто, когда каждый словарь содержит разные атрибуты одного объекта. Например, вы можете создать словарь для каждого пользователя сайта, как это было сделано в программе `user.py` в подразделе «Перебор всех пар “ключ — значение”» выше в данной главе, и сохранить отдельные словари в списке `users`. Все словари в списке должны иметь одинаковую структуру, чтобы вы могли перебрать список и выполнить с каждым объектом словаря одни и те же операции.

Список в словаре

Вместо того чтобы помещать словарь в список, иногда бывает удобно совершить обратное действие. Представьте, как бы вы описали в программе заказанную пиццу. Если ограничиться только списком, то сохранить удастся разве что список начинок для пиццы. При использовании словаря список начинок может быть всего лишь одним аспектом описания пиццы.

В следующем примере для каждой пиццы сохраняются два вида информации: тип коржа и список начинок. Последний представляет собой значение, связанное с ключом `'toppings'`. Чтобы использовать элементы в списке, нужно указать имя словаря и ключ `'toppings'`, как и для любого другого значения в словаре. Вместо одного значения будет получен список начинок:

pizza.py

```
# Сохранение информации о заказанной пицце.
```

```
pizza = {  
    'crust': 'thick',  
    'toppings': ['mushrooms', 'extra cheese'],  
}
```

```
# Описание заказа.
```

```
❶ print(f"You ordered a {pizza['crust']}-crust pizza "  
      "with the following toppings:")
```

```
❷ for topping in pizza['toppings']:  
    print(f"\t{topping}")
```

Работа начинается со словаря, в котором хранится информация о заказанной пицце. С ключом в словаре `'crust'` связано строковое значение `'thick'`. С другим ключом `'toppings'` связано значение-список, в котором хранятся данные обо всех заказанных начинках. Затем выводится описание заказа перед изготовлением пиццы ❶. Если вам нужно разбить длинную строку в вызове функции `print()`, то выберите точку для разбиения выводимой строки и закончите ее кавычкой. Добавьте в следующую строку отступ, открывающую кавычку и продолжите строку. Python автоматически объединяет все строки, обнаруженные в круглых скобках. Для вывода начинок пишется цикл `for` ❷. Чтобы вывести список начинок, мы используем ключ `'toppings'`, а Python берет список начинок из словаря.

Следующее сообщение описывает пиццу, которую мы собираемся создать:

```
You ordered a thick-crust pizza with the following toppings:  
    mushrooms  
    extra cheese
```

Вложение списка в словарь может применяться каждый раз, когда с одним ключом словаря должно быть связано несколько значений. Если бы в предыдущем примере с языками программирования ответы сохранялись в списке, то один участник опроса мог бы выбрать сразу несколько любимых языков.

При переборе словаря значение, связанное с каждым человеком, представляло бы собой список языков (вместо одного языка). В цикле `for` словаря создается другой цикл для перебора списка языков, связанных с каждым участником:

favorite_languages.py

```
favorite_languages = {  
    'jen': ['python', 'rust'],  
    'sarah': ['c'],  
    'edward': ['rust', 'go'],  
    'phil': ['python', 'haskell'],  
}  
  
❶ for name, languages in favorite_languages.items():  
    print(f"\n{name.title()}'s favorite languages are:")  
❷    for language in languages:  
        print(f"\t{language.title()}")
```

Значение, связанное с каждым именем, теперь представляет собой список. У одних участников единственный любимый язык программирования, у других таких языков несколько. При переборе словаря ❶ переменная `languages` используется для хранения каждого значения из него, поскольку мы знаем, что оно будет представлять собой список. В основном цикле по элементам словаря другой цикл ❷ перебирает элементы списка любимых языков каждого участника. Теперь каждый участник опроса может указать сколько угодно любимых языков программирования:

```
Jen's favorite languages are:  
    Python  
    Rust
```

```
Sarah's favorite languages are:  
    C
```

```
Edward's favorite languages are:  
    Ruby  
    Go
```

```
Phil's favorite languages are:  
    Python  
    Haskell
```

Чтобы дополнительно усовершенствовать программу, добавьте в начало цикла `for` словаря оператор `if` для проверки того, выбрал ли данный участник несколько языков программирования (проверка основана на значении `len(languages)`). Если у участника только один любимый язык, то текст сообщения изменяется, чтобы в нем использовалось единственное число (например, Sarah's favorite language is C).

ПРИМЕЧАНИЕ

Глубина вложения списков и словарей не должна быть слишком большой. Если вам приходится вкладывать элементы на глубину, существенно бóльшую, чем в предыдущих примерах, или если вы работаете с чужим кодом, в котором глубина вложения довольно велика, то, скорее всего, у задачи существует более простое решение.

Вложение словарей

Словарь можно вложить в другой словарь, но в таких случаях код быстро усложняется. Например, если на сайте есть несколько пользователей с уникальными именами, то вы можете использовать их имена как ключи в словаре. Информация о каждом пользователе при этом хранится в словаре, который применяется как значение, связанное с именем. В следующем примере о каждом пользователе хранятся три вида информации: имя, фамилия и место жительства. Чтобы получить доступ к этим данным, переберите имена пользователей и словарь с информацией, связанной с каждым именем:

many_users.py

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
    },

    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
    },
}
```

- ❶ `for username, user_info in users.items():`
- ❷ `print(f"\nUsername: {username}")`
- ❸ `full_name = f"{user_info['first']} {user_info['last']}"`
 `location = user_info['location']`
- ❹ `print(f"\tFull name: {full_name.title()}")`
 `print(f"\tLocation: {location.title()}")`

В программе определяется словарь `users`, содержащий два ключа: для пользователей `'aeinstein'` и `'mcurie'`. Значение, связанное с каждым ключом, представляет собой словарь с именем, фамилией и местом жительства пользователя. В процессе перебора словаря `users` ❶ Python сохраняет каждый ключ в переменной `username`, а словарь, связанный с каждым именем пользователя, — в переменной `user_info`. Внутри основного цикла в словаре выводится имя пользователя ❷.

Затем начинается работа с внутренним словарем ❸. Переменная `user_info`, содержащая словарь с информацией о пользователе, содержит три ключа: `'first'`, `'last'` и `'location'`. Каждый ключ используется для создания отформатированных данных, содержащих полное имя и место жительства пользователя, а затем для вывода сводки известной информации о пользователе ❹:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton
```

```
Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Обратите внимание на идентичность структур словарей всех пользователей. Хотя Python этого и не требует, наличие единой структуры упрощает работу с вложенными словарями. Если словари разных пользователей будут содержать разные ключи, то код в цикле `for` заметно усложнится.

УПРАЖНЕНИЯ

6.7. Люди. Начните с программы, написанной для упражнения 6.1. Создайте два новых словаря, представляющих разных людей, и сохраните все три словаря в списке `people`. Переберите элементы списка людей. В процессе перебора выведите всю имеющуюся информацию о каждом человеке.

6.8. Домашние животные. Создайте несколько словарей, имена которых представляют клички домашних животных. В каждом словаре сохраните информацию о виде животного и имени владельца. Сохраните словари в списке `pets`. Переберите элементы списка. В процессе перебора выведите всю имеющуюся информацию о каждом животном.

6.9. Любимые места. Создайте словарь `favorite_places`. Придумайте названия трех мест, которые станут ключами словаря, и сохраните для каждого человека из упражнения 6.7 от одного до трех любимых мест. Чтобы задача стала более интересной, опросите нескольких друзей и соберите реальные данные для своей программы. Переберите данные в словаре, выведите имя каждого человека и его любимые места.

6.10. Любимые числа. Измените программу из упражнения 6.2, чтобы для каждого человека можно было хранить несколько любимых чисел. Выведите имя каждого человека в списке и его любимые числа.

6.11. Города. Создайте словарь `cities`. Используйте названия трех городов в качестве ключей словаря. Создайте словарь с информацией о каждом городе; добавьте в него страну, в которой расположен город, примерную численность населения и один примечательный факт, относящийся к этому городу. Ключи словаря каждого города должны называться `country`, `population` и `fact`. Выведите название каждого города и всю сохраненную информацию о нем.

6.12. Расширение. Примеры, с которыми мы работаем, стали достаточно сложными, и в них можно вносить разного рода усовершенствования. Воспользуйтесь одним из примеров этой главы и расширьте его: добавьте новые ключи и значения, измените контекст программы или улучшите форматирование вывода.

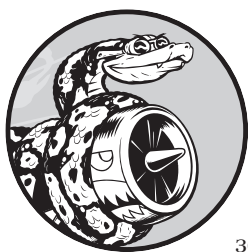
Резюме

В этой главе вы научились определять словари и работать с хранящейся в них информацией. Вы узнали, как обращаться к отдельным элементам словаря и изменять их, как перебрать всю информацию в словаре. Вы изучили способы перебора пар «ключ — значение», ключей и значений словаря. Кроме того, вы узнали о том, как вкладывать словари в список, списки — в словари, а словари — в другие словари.

В следующей главе вы познакомитесь с циклами `while` и получением входных данных от пользователей программ. Эта глава будет особенно интересной, поскольку вы наконец-то сможете сделать свои программы интерактивными: они начнут реагировать на действия пользователя.

7

Ввод данных и циклы `while`



Программы обычно пишутся для решения задач конечного пользователя. Для этого им нужна какая-то информация, которую должен ввести пользователь. Простой пример: допустим, пользователь хочет узнать, достаточен ли его возраст для голосования.

Если вы пишете программу для ответа на этот вопрос, то вам нужно будет узнать возраст пользователя. Программа должна запросить у пользователя значение — его возраст; когда у нее появятся данные, она может сравнить их с возрастом, дающим право на голосование, и сообщить результат.

В этой главе вы узнаете, как получить пользовательский *ввод* (то есть входные данные), чтобы программа могла работать с ним. Если она хочет получить отдельное имя, то запрашивает только его; если ей нужен список имен — выводит соответствующее сообщение. Для получения данных в программах используется функция `input()`.

Вы также научитесь продолжать работу программы, пока пользователь вводит новые данные; после получения всех данных программа переходит к работе с полученной информацией. Цикл `while` в языке Python позволяет выполнять программу, пока некое условие остается истинным.

Научившись работать с пользовательским вводом и управлять продолжительностью выполнения программы, вы сможете создавать полностью интерактивные программы.

Как работает функция `input()`

Функция `input()` приостанавливает выполнение программы и ожидает, пока пользователь введет некий текст. Получив ввод, Python сохраняет его в переменной, чтобы вам было удобнее работать с ним.

Например, следующая программа предлагает пользователю ввести текст, а затем выводит для него сообщение:

parrot.py

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

Функция `input()` получает один аргумент: текст *подсказки* (или инструкции), который выводится на экран, чтобы пользователь понимал, что от него требуется. В данном примере при выполнении первой строки пользователь видит подсказку с предложением ввести любой текст. Программа ожидает, пока пользователь введет ответ, и продолжает работу после того, как он нажмет **Enter**. Ответ сохраняется в переменной `message`, после чего функция `print(message)` дублирует введенные данные:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

ПРИМЕЧАНИЕ

VS Code и многие другие редакторы кода не запускают программы, запрашивающие входные данные у пользователя. Вы можете использовать такие редакторы для создания подобных программ, но запускать их придется из терминального окна. См. раздел «Запуск программ Python из терминала» в главе 1.

Содержательные подсказки

Каждый раз, когда в вашей программе используется функция `input()`, вы должны давать четкую, понятную подсказку, которая точно сообщит пользователю, какую информацию вы хотите получить от него. Подойдет любое предложение, которое объяснит пользователю, что нужно вводить. Пример:

greeter.py

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

Добавьте пробел в конце подсказки (после двоеточия в предыдущем примере), чтобы отделить подсказку от данных, вводимых пользователем, и четко показать, где должен вводиться текст. Пример:

```
Please enter your name: Eric
Hello, Eric!
```

Иногда подсказка занимает несколько строк. Например, вы можете сообщить пользователю, для чего программа запрашивает данные. Текст подсказки можно сохранить в переменной и передать ее функции `input()`: вы создаете длинную

подсказку из нескольких строк, а потом выполняете один компактный оператор `input()`.

greeter.py

```
prompt = "If you share your name, we can personalize the messages you see."
prompt += "\nWhat is your first name? "
```

```
name = input(prompt)
print(f"\nHello, {name}!")
```

В этом примере продемонстрирован один из способов создания длинных строк. Первая часть длинного сообщения сохраняется в переменной `prompt`. Затем оператор `+=` объединяет хранящийся в ней текст с новым фрагментом.

Теперь содержимое `prompt` занимает две строки (вопросительный знак снова отделяется от ввода пробелом для наглядности):

```
If you share your name, we can personalize the messages you see.
What is your first name? Eric
```

```
Hello, Eric!
```

Использование функции `int()` для получения числового ввода

При использовании функции `input()` Python воспринимает все данные, введенные пользователем, как строку. В следующем сеансе интерпретатора программа запрашивает у пользователя возраст:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

Пользователь вводит число 21, но когда мы запрашиваем у Python значение `age`, выводится `'21'` — представление введенного числа в строковом формате. Кавычки, в которые заключены данные, указывают на то, что Python воспринимает ввод как строку. Но попытка использовать данные как число приведет к ошибке:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age >= 18
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
❷ TypeError: '>=' not supported between instances of 'str' and 'int'
```

Когда вы пытаетесь сравнить введенные данные с числом ❶, Python выдает ошибку, поскольку не может сравнить строку с числом: строка `'21'`, хранящаяся в `age`, не сравнивается с числовым значением 18; происходит ошибка ❷.

Проблему можно решить с помощью функции `int()`, которая преобразует входную строку в числовое значение. Это позволяет успешно провести сравнение:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age = int(age)
>>> age >= 18
True
```

В этом примере введенный текст `21` интерпретируется как строка, но затем преобразуется в числовое представление путем вызова функции `int()` ❶. Теперь Python может проверить условие: сравнить переменную `age` (которая теперь содержит числовое значение `21`) с `18`. Условие «значение `age` больше или равно `18`» выполняется, и результат проверки равен `True`.

Как использовать функцию `int()` в реальной программе? Допустим, программа проверяет рост пользователя и определяет, достаточен ли он для катания на аттракционе:

rollercoaster.py

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

Программа может сравнить `height` с `48`, поскольку строка `height = int(height)` преобразует входное значение в число перед проведением сравнения. Если введенное число больше или равно `48`, то программа сообщает пользователю, что он прошел проверку:

```
How tall are you, in inches? 71
```

```
You're tall enough to ride!
```

Если пользователь вводит числовые данные, которые используются в вашей программе для вычислений и сравнений, то обязательно преобразуйте введенное значение в его числовой эквивалент.

Оператор деления по модулю

При работе с числовыми данными может пригодиться *оператор деления по модулю* (modulo operator) (`%`), который делит одно число на другое и возвращает остаток:

```
>>> 4 % 3
1
>>> 5 % 3
2
```



```
>>> 6 % 3
0
>>> 7 % 3
1
```

Оператор `%` не сообщает частное от целочисленного деления; он возвращает только остаток.

Когда одно число нацело делится на другое, остаток равен 0, и оператор `%` возвращает 0. Например, этот факт может использоваться для проверки четности или нечетности числа:

even_or_odd.py

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)
```

```
if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

Четные числа всегда делятся на 2. Следовательно, если остаток от деления на 2 равен 0 (в данном случае `number % 2 == 0`), число четное, а если нет — нечетное.

Enter a number, and I'll tell you if it's even or odd: **42**

The number 42 is even.

УПРАЖНЕНИЯ

7.1. Каршеринг. Напишите программу, которая спрашивает у пользователя, какую машину он хотел бы взять напрокат. Выведите сообщение с введенными данными (например, «Посмотрим, смогу ли я найти вам Subaru»).

7.2. Заказ стола. Напишите программу, которая спрашивает у пользователя, на сколько мест он хочет забронировать стол в ресторане. Если введенное число больше 8, то выведите сообщение о том, что пользователю придется подождать. В противном случае сообщите, что стол готов.

7.3. Числа, кратные 10. Запросите у пользователя число и сообщите, кратно ли оно 10.

Циклы while

Цикл `for` получает коллекцию элементов и выполняет блок кода по одному разу для каждого элемента в коллекции. В отличие от него, цикл `while` продолжает выполняться, пока некое условие остается истинным.

Как работает цикл *while*

Цикл *while* может использоваться для перебора числовой последовательности. Например, следующий цикл считает от 1 до 5:

counting.py

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

В первой строке отсчет начинается с 1, для чего *current_number* присваивается значение 1. Далее запускается цикл *while*, который продолжает работать, пока значение *current_number* остается меньшим или равным 5. Код в цикле выводит значение *current_number* и увеличивает его на 1 с помощью команды *current_number += 1*. (Операция *+=* является сокращенной формой записи для *current_number = current_number + 1*.)

Цикл повторяется, пока условие *current_number <= 5* остается истинным. Так как 1 меньше 5, Python выводит 1, а затем увеличивает значение на 1, отчего *current_number* становится равным 2. Поскольку 2 меньше 5, то Python выводит 2 и снова прибавляет 1 и т. д. Как только значение *current_number* превысит 5, цикл останавливается, а программа завершается:

```
1
2
3
4
5
```

Очень многие повседневные программы содержат циклы *while*. Например, представьте компьютерную игру: цикл *while* выполняется, пока игра продолжается, и завершается, как только игрок захочет остановить игру. Вряд ли кого-нибудь обрадует, если программа завершит работу преждевременно или продолжит работать, когда ей приказали остановиться, так что циклы *while* весьма полезны.

Пользователь решает прервать работу программы

Программа *parrot.py* может выполняться, пока пользователь не захочет остановить ее, — для этого большая часть кода заключается в цикл *while*. В программе определяется *признак завершения* (quit value), и она работает, пока пользователь не введет нужное значение:

parrot.py

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
```

```
while message != 'quit':  
    message = input(prompt)  
    print(message)
```

Сначала определяется подсказка, в которой объясняется, что у пользователя есть два варианта: ввести сообщение или признак завершения (в данном случае это строка 'quit'). Затем переменной `message` присваивается значение, введенное пользователем. В программе данная переменная инициализируется пустой строкой "", чтобы значение проверялось без ошибок при первом выполнении строки `while`. Когда программа запускается впервые и выполнение достигает оператора `while`, значение `message` необходимо сравнить с 'quit', но пользователь еще не вводил никакие данные. Если у Python нет данных для сравнения, то продолжать выполнение невозможно. Чтобы решить эту проблему, необходимо предоставить `message` исходное значение. И хотя это всего лишь пустая строка, для Python такое значение выглядит вполне осмысленно; программа сможет выполнить сравнение, на котором основана работа цикла `while`. Он выполняется, пока значение `message` не равно 'quit'.

При первом выполнении цикла переменная `message` содержит пустую строку, и Python входит в цикл. При выполнении команды `message = input(prompt)` Python отображает подсказку и ожидает введения пользователем данных. Они сохраняются в переменной `message` и выводятся функцией `print()`; после этого Python снова проверяет условие оператора `while`. Пока пользователь не введет слово 'quit', подсказка будет выводиться снова и снова, а Python будет ожидать новых данных. При вводе слова 'quit' Python перестает выполнять цикл `while`, а программа завершается:

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello everyone!  
Hello everyone!
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello again.  
Hello again.
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. quit  
quit
```

Программа работает неплохо, если не считать того, что она выводит слово 'quit', словно оно является обычным сообщением. Простая проверка `if` решает проблему:

```
prompt = "\nTell me something, and I will repeat it back to you:"  
prompt += "\nEnter 'quit' to end the program. "  
  
message = ""  
while message != 'quit':  
    message = input(prompt)  
  
    if message != 'quit':  
        print(message)
```

Теперь программа проводит проверку перед выводом сообщения и выводит сообщение только в том случае, если оно не совпадает с признаком завершения:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!
```

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.
```

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Флаги

В предыдущем примере программа выполняла некие операции, пока заданное условие оставалось истинным. А если вы пишете более сложную программу, выполнение которой может прерываться по нескольким условиям?

Например, компьютерная игра может завершаться по разным причинам: у игрока кончились все «жизни»; прошло отведенное время; все города, которые он должен был защищать, были уничтожены и т. д. Игра должна завершаться при выполнении любого из этих условий. Попытки проверять все возможные условия в одном операторе `while` быстро усложняются и становятся слишком громоздкими.

Если программа должна выполняться только при истинности нескольких условий, то определите одну переменную — *флаг* (flag). Она сообщает, должна ли программа выполняться далее. Программу можно написать так, чтобы она продолжала выполнение, если флаг находится в состоянии `True`, и завершалась, если любое из нескольких событий перевело флаг в состояние `False`. В результате в операторе `while` достаточно проверить всего одно условие: находится ли флаг в состоянии `True`. Все остальные проверки (которые должны определить, произошло ли событие, переводящее флаг в состояние `False`) удобно организуются в остальном коде.

Добавим флаг в программу `parrot.py` из предыдущего раздела. Этот флаг, который мы назовем `active` (хотя переменная может называться как угодно), управляет тем, должно ли продолжаться выполнение программы:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "
```

```
active = True
❶ while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

Переменной `active` присваивается `True`, чтобы программа начинала работу в активном состоянии. Это присваивание упрощает оператор `while`, поскольку в нем самом никакие сравнения не выполняются; вся логика реализуется в других частях программы. Пока переменная `active` остается равной `True`, цикл выполняется ❶.

В операторе `if` внутри цикла `while` значение `message` проверяется после того, как пользователь введет данные. Если он ввел строку `'quit'`, то флаг `active` переходит в состояние `False`, а цикл `while` останавливается. Если пользователь ввел любой текст, кроме `'quit'`, то эти данные выводятся как сообщение.

Результаты работы этой программы ничем не отличаются от результатов в предыдущем примере, в котором условная проверка выполняется прямо в операторе `while`. Но теперь в программе есть флаг, указывающий, находится ли она в активном состоянии, и вы сможете легко добавить новые проверки (в форме оператора `elif`) для событий, с которыми переменная `active` может перейти в состояние `False`. Это может быть удобно в сложных программах — например, в компьютерных играх с многочисленными событиями, каждое из которых может привести к завершению программы. Когда по любому из этих событий флаг `active` переходит в состояние `False`, основной игровой цикл прерывается, выводится сообщение о завершении игры, и у игрока появляется возможность сыграть еще раз.

Оператор `break` и выход из цикла

Чтобы немедленно прервать цикл `while` без выполнения оставшегося в цикле кода независимо от состояния условия, используйте оператор `break`. Он управляет ходом работы программы; она позволит вам управлять тем, какая часть кода выполняется, а какая нет.

Рассмотрим пример — программу, которая спрашивает у пользователя, в каких городах он бывал. Чтобы прервать цикл `while`, программа выполняет оператор `break`, как только пользователь введет значение `'quit'`:

cities.py

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "
```

```
❶ while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")
```

Цикл, который начинается с `while True` ❶, будет выполняться бесконечно — если только в нем не будет выполнен оператор `break`. Цикл в программе продолжает

запрашивать у пользователя названия городов, пока тот не введет строку 'quit'. При ее вводе выполняется оператор `break`, заставляющий Python выйти из цикла:

```
Please enter the name of a city you have visited:  
(Enter 'quit' when you are finished.) New York  
I'd love to go to New York!
```

```
Please enter the name of a city you have visited:  
(Enter 'quit' when you are finished.) San Francisco  
I'd love to go to San Francisco!
```

```
Please enter the name of a city you have visited:  
(Enter 'quit' when you are finished.) quit
```

ПРИМЕЧАНИЕ

Оператор `break` может использоваться в любых циклах Python. Например, его можно добавить в цикл `for` для перебора элементов словаря.

Оператор *continue* и продолжение цикла

Вместо того чтобы полностью выходить из цикла, не выполняя оставшуюся часть кода, вы можете воспользоваться оператором `continue`, который позволяет вернуться к началу цикла и проверить условия. Например, возьмем цикл, который считает от 1 до 10, но выводит только нечетные числа в этом диапазоне:

counting.py

```
current_number = 0  
while current_number < 10:  
    ❶ current_number += 1  
    if current_number % 2 == 0:  
        continue  
  
    print(current_number)
```

Сначала переменной `current_number` присваивается 0. Значение меньше 10, поэтому Python входит в цикл `while`. При этом счетчик увеличивается на 1 ❶, так что `current_number` принимает значение 1. Затем оператор `if` проверяет остаток от деления `current_number` на 2. Если остаток равен 0 (то есть `current_number` делится на 2), то оператор `continue` дает Python указание проигнорировать оставшийся код цикла и вернуться к началу. Если счетчик не делится на 2, то оставшаяся часть цикла выполняется, и Python выводит текущее значение счетчика:

```
1  
3  
5  
7  
9
```

Предотвращение заикливания

У каждого цикла `while` должна быть предусмотрена возможность завершения, чтобы он не выполнялся бесконечно. Например, следующий цикл считает от 1 до 5:

counting.py

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

Но если случайно пропустить строку `x += 1` (см. далее), то цикл будет выполняться бесконечно:

```
# Бесконечный цикл!
x = 1
while x <= 5:
    print(x)
```

Теперь переменной `x` присваивается начальное значение 1, но оно никогда не изменяется в программе. В итоге проверка условия `x <= 5` всегда дает результат `True`, и цикл `while` выводит бесконечную серию единиц:

```
1
1
1
1
--пропуск--
```

Любой программист время от времени пишет бесконечный цикл, особенно если в программе используются неочевидные условия завершения. Если ваша программа заиклилась, то нажмите **Ctrl+C** или просто закройте терминальное окно с выводом программы.

Чтобы избежать заикливания, тщательно проверьте каждый цикл `while` и убедитесь в том, что он прерывается именно тогда, когда это предполагается. Если программа должна завершаться при вводе некоего значения, то запустите программу и введите его. Если программа не завершилась, то проанализируйте обработку значения, которое должно приводить к выходу из цикла. Убедитесь, что хотя бы одна часть программы может сделать условие цикла ложным или привести к выполнению оператора `break`.

ПРИМЕЧАНИЕ

VS Code, как и многие редакторы, отображает выходные данные во встроенном терминальном окне. Чтобы отменить бесконечный цикл, обязательно щелкните в области вывода редактора, прежде чем нажимать **CTRL+C**.

УПРАЖНЕНИЯ

7.4. Начинки для пиццы. Напишите цикл, который предлагает пользователю вводить начинки для пиццы до тех пор, пока не будет введено значение 'quit'. При вводе каждой начинки выведите сообщение о том, что она добавлена в заказ.

7.5. Билеты в кино. Кинотеатр установил несколько вариантов цены на билеты в зависимости от возраста посетителя. Для посетителей младше 3 лет билет бесплатный; если посетителю от 3 до 12 лет, то билет стоит 10 долларов; наконец, если посетитель старше 12 лет, то билет стоит 15 долларов. Напишите цикл, который предлагает пользователю ввести возраст и выводит цену билета.

7.6. Три выхода. Напишите альтернативную версию упражнения 7.4 или упражнения 7.5, в которой каждый пункт следующего списка встречается хотя бы раз:

- завершение цикла при проверке условия в операторе `while`;
- управление продолжительностью выполнения цикла в зависимости от переменной `active`;
- выход из цикла с помощью оператора `break`, если пользователь вводит значение 'quit'.

7.7. Бесконечный цикл. Напишите цикл, который никогда не завершается, и выполните его. (Чтобы выйти из цикла, нажмите `Ctrl+C` или закройте окно с выводом.)

Использование цикла `while` со списками и словарями

До настоящего момента мы работали только с одним фрагментом пользовательской информации. Мы получали ввод, а затем выводили ответ. При следующем проходе цикла `while` программа получала новое входное значение и реагировала на него. Но чтобы работать с несколькими фрагментами информации, необходимо использовать в циклах `while` списки и словари.

Цикл `for` хорошо подходит для перебора списков, но, скорее всего, список не должен изменяться в цикле, поскольку у Python возникнут проблемы с отслеживанием элементов списка. Изменять список в процессе обработки можно с помощью цикла `while`. Использование этих циклов со списками и словарями позволяет собирать, хранить и упорядочивать большие объемы данных в целях их последующего анализа и обработки.

Перемещение элементов между списками

Возьмем список недавно зарегистрированных, но еще не проверенных пользователей сайта. Как переместить их после проверки в отдельный список проверенных пользователей? Одно из возможных решений: используем цикл `while` для извлечения пользователей из списка непроверенных, проверяем их и добавляем в отдельный список проверенных пользователей. Код может выглядеть так:

confirmed_users.py

```
# Начинаем с двух списков: пользователей, которых нужно проверить,  
# и пустого списка для хранения проверенных пользователей.  
❶ unconfirmed_users = ['alice', 'brian', 'candace']  
   confirmed_users = []  
  
# Проверяем каждого пользователя, пока остаются непроверенные пользователи. Каждый  
# пользователь, прошедший проверку, перемещается в список проверенных.  
❷ while unconfirmed_users:  
❸     current_user = unconfirmed_users.pop()  
  
     print(f"Verifying user: {current_user.title()}")  
❹     confirmed_users.append(current_user)  
  
# Вывод всех проверенных пользователей.  
print("\nThe following users have been confirmed:")  
for confirmed_user in confirmed_users:  
    print(confirmed_user.title())
```

Работа программы начинается с двух списков: непроверенных пользователей ❶ (Alice, Brian и Candace) и пустого списка для проверенных пользователей. Цикл `while` выполняется, пока в списке `unconfirmed_users` остаются элементы ❷. Внутри этого списка метод `pop()` извлекает очередного непроверенного пользователя из конца списка `unconfirmed_users` ❸. В данном примере список `unconfirmed_users` завершается пользователем Candace; это имя первым извлекается из списка, сохраняется в `current_user` и добавляется в список `confirmed_users` ❹. Далее следуют пользователи Brian и Alice.

Программа моделирует проверку каждого пользователя с помощью вывода сообщения, после чего переносит пользователя в список проверенных. По мере сокращения списка непроверенных пользователей список проверенных растет. Когда в списке непроверенных пользователей не остается ни одного элемента, цикл останавливается и выводится список проверенных пользователей:

```
Verifying user: Candace  
Verifying user: Brian  
Verifying user: Alice
```

```
The following users have been confirmed:  
Candace  
Brian  
Alice
```

Удаление всех вхождений конкретного значения из списка

В главе 3 функция `remove()` использовалась для удаления конкретного значения из списка. Она работала, поскольку интересное нас значение встречалось в списке только раз. Но что, если вы захотите удалить все вхождения значения из списка?

Допустим, имеется список `pets`, в котором значение `'cat'` встречается многократно. Чтобы удалить все экземпляры этого значения, можно выполнять цикл `while` до тех пор, пока в списке не останется ни одного экземпляра `'cat'`:

pets.py

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)
```

```
while 'cat' in pets:
    pets.remove('cat')
```

```
print(pets)
```

Программа начинает со списка, содержащего множественные экземпляры `'cat'`. После вывода списка Python входит в цикл `while`, поскольку значение `'cat'` есть в списке хотя бы в одном экземпляре. После входа цикл Python удаляет первое вхождение `'cat'`, возвращается к строке `while`, а затем обнаруживает, что экземпляры `'cat'` все еще есть в списке, и проходит цикл заново. Вхождения `'cat'` удаляются до тех пор, пока не окажется, что в списке значений `'cat'` не осталось; в этот момент Python завершает цикл и выводит список заново:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Заполнение словаря данными, введенными пользователем

При каждом проходе цикла `while` ваша программа может запрашивать любое необходимое количество данных. Напишем программу, которая при каждом проходе цикла запрашивает имя участника и его ответ. Собранные данные будут сохраняться в словаре, поскольку каждый ответ должен быть связан с конкретным пользователем:

mountain_poll.py

```
responses = {}
# Установка флага продолжения опроса.
polling_active = True
```

```
while polling_active:
```

```
    # Запрос имени и ответа пользователя.
```

```
    name = input("\nWhat is your name? ")
```

```
    response = input("Which mountain would you like to climb someday? ")
```

❶

```

# Ответ сохраняется в словаре.
❷ responses[name] = response

# Проверка продолжения опроса.
❸ repeat = input("Would you like to let another person respond? (yes/ no) ")
if repeat == 'no':
    polling_active = False

# Опрос завершен, вывести результаты.
print("\n--- Poll Results ---")
❹ for name, response in responses.items():
    print(f"{name} would like to climb {response}.")

```

Сначала программа определяет пустой словарь (`responses`) и устанавливает флаг (`polling_active`), показывающий, что опрос продолжается. Пока `polling_active` содержит `True`, Python будет выполнять код в цикле `while`.

В цикле пользователю предлагается ввести имя и название горы, на которую ему хотелось бы подняться ❶. Эта информация сохраняется в словаре `responses` ❷, после чего программа спрашивает у пользователя, нужно ли продолжать опрос ❸. Если пользователь отвечает положительно, то программа снова входит в цикл `while`. Если же ответ отрицателен, флаг `polling_active` переходит в состояние `False`, цикл `while` перестает выполняться, и завершающий блок кода ❹ выводит результаты опроса.

Если вы запустите эту программу и введете пару ответов, то результат будет выглядеть примерно так:

```

What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
Lynn would like to climb Devil's Thumb.
Eric would like to climb Denali.

```

УПРАЖНЕНИЯ

7.8. Бутерброды. Создайте список `sandwich_orders`, заполните его названиями различных видов бутербродов. Создайте пустой список `finished_sandwiches`. В цикле переберите элементы первого списка и выведите сообщение для каждого элемента (например, «Я приготовил бутерброд с тунцом»). После этого каждый бутерброд из первого списка перемещается в список `finished_sandwiches`. После того как все элементы первого списка будут обработаны, выведите сообщение, в котором перечисляются все изготовленные бутерброды.

7.9. Без пастромы. Используя список `sandwich_orders` из упражнения 7.8, проследите за тем, чтобы значение `'pastrami'` встречалось в списке как минимум три раза. Добавьте в начало программы код для вывода сообщения о том, что пастромы больше нет, и напишите цикл `while`, удаляющий все вхождения `'pastrami'` из `sandwich_orders`. Убедитесь в том, что в `finished_sandwiches` значение `'pastrami'` не встречается ни одного раза.

7.10. Отпуск мечты. Напишите программу, которая опрашивает пользователей, где бы они хотели провести отпуск. Добавьте подсказку вида «Если бы вы могли посетить одно место в мире, то куда бы отправились?» Добавьте блок кода, который выводит результаты опроса.

Резюме

В этой главе вы научились использовать `input()` для того, чтобы пользователи могли вводить собственную информацию в ваших программах. Вы узнали, как работать с числовыми и текстовыми данными, а также научились управлять продолжительностью выполнения своих программ с помощью циклов `while`. Кроме того, вы изучили несколько способов управления циклами `while`: установку флага, операторы `break` и `continue`. Вы узнали, как использовать цикл `while` для перемещения элементов из одного списка в другой и как удалить все вхождения некоего значения из списка. Вдобавок вы изучили возможности применения циклов `while` со словарями.

Глава 8 посвящена *функциям*. Они позволяют разделить программу на меньшие части, каждая из которых решает одну конкретную задачу. Функции можно хранить в отдельных файлах и вызывать их столько раз, сколько потребуется. Благодаря функциям вы сможете писать более эффективный, более простой в отладке и сопровождении код, который к тому же можно повторно использовать в разных программах.

8

Функции



Эта глава посвящена *функциям* — именованным блокам кода, предназначенным для решения одной конкретной задачи. Чтобы выполнить задачу, определенную в виде функции, вы *вызываете* функцию, отвечающую за эту задачу. Если задача должна многократно выполняться в программе, то вам не придется заново вводить весь необходимый код; просто вызовите функцию, предназначенную для решения задачи, и благодаря этому вызову Python получит указание выполнить код, содержащийся внутри функции. Как вы вскоре убедитесь, использование функций упрощает чтение, написание, тестирование кода и исправление ошибок.

В этой главе также рассматриваются возможности передачи информации функциям. Вы узнаете, как писать функции, основной задачей которых является вывод информации, и другие функции, предназначенные для обработки данных и возвращения значения (или набора значений). Наконец, вы научитесь хранить функции в отдельных файлах, называемых *модулями*, в целях упорядочения файлов основной программы.

Определение функции

Вот простая функция `greet_user()`, которая выводит приветствие:

greeter.py

```
def greet_user():
    """Выводит простое приветствие."""
    print("Hello!")

greet_user()
```

В этом примере представлена простейшая структура функции. Первая строка с помощью ключевого слова `def` сообщает Python, что вы определяете функцию.

В *определении функции* (function definition) указываются имя функции и, если необходимо, описание информации, требуемой функции для решения ее задачи. Эта информация заключается в круглые скобки. В данном примере функции присвоено имя `greet_user()`, и она не нуждается в дополнительной информации для решения своей задачи, поэтому круглые скобки пусты. (Но даже в этом случае они обязательны.) Наконец, определение завершается двоеточием.

Все строки с отступами, следующие за `def greet_user():`, образуют *тело* функции. Текст во второй строке представляет собой комментарий — *строку документации* с описанием действий функции (строк может быть несколько). Такие комментарии заключаются в тройные кавычки; Python опознает их по этой последовательности символов во время генерирования документации к функциям в ваших программах.

«Настоящий» код в теле этой функции состоит всего из одной строки `print("Hello!")`. Таким образом, функция `greet_user()` решает всего одну задачу: выполнение функции `print("Hello!")`.

Когда потребуется использовать эту функцию, *вызовите* ее. Так Python получит указание выполнить содержащийся в ней код. Чтобы вызвать функцию, укажите ее имя, за которым следует вся необходимая информация, заключенная в круглые скобки. Никакая дополнительная информация не нужна, поэтому вызов функции эквивалентен простому выполнению функции `greet_user()`. Как и ожидалось, функция выводит сообщение `Hello!`:

```
Hello!
```

Передача данных функции

Если внести небольшие изменения, то с помощью функции `greet_user()` вы сможете поприветствовать пользователя, назвав его по имени. Для этого следует добавить имя `username` в круглых скобках в определении функции `def greet_user()`. Благодаря добавлению `username` функция примет любое значение, которое будет заключено в скобки при вызове. Теперь функция ожидает, что при каждом вызове будет передаваться имя пользователя. При вызове `greet_user()` укажите имя (например, `'jesse'`) в круглых скобках:

```
def greet_user(username):  
    """Выводит простое приветствие."""  
    print(f"Hello, {username.title()}!")  
  
greet_user('jesse')
```

Команда `greet_user('jesse')` вызывает функцию `greet_user()` и передает ей информацию, необходимую для выполнения вызова функции `print()`. Функция получает переданное имя и выводит приветствие для этого имени:

```
Hello, Jesse!
```

Точно так же команда `greet_user('sarah')` вызывает функцию `greet_user()` и передает ей строку `'sarah'`, в результате чего будет выведено сообщение `Hello, Sarah!` Функцию `greet_user()` можно вызвать сколько угодно раз и передать ей любое имя на ваше усмотрение — и вы будете получать ожидаемый результат.

Аргументы и параметры

Функция `greet_user()` определена так, что для работы она должна получить значение переменной `username`. После того как функция будет вызвана и получит необходимую информацию (имя пользователя), она выведет правильное приветствие.

Переменная `username` в определении `greet_user()` — *параметр*, то есть условные данные, необходимые функции для выполнения ее работы. Значение `'jesse'` в `greet_user('jesse')` — *аргумент*, то есть конкретная информация, переданная при вызове функции. Вызывая функцию, вы заключаете значение, с которым функция должна работать, в круглые скобки. В данном случае аргумент `'jesse'` был передан функции `greet_user()`, а его значение было сохранено в переменной `username`.

ПРИМЕЧАНИЕ

Иногда в литературе термины «аргумент» и «параметр» используются как синонимы. Не удивляйтесь, если переменные в определении функции вдруг будут названы аргументами, а значения, переданные при вызове функции, — параметрами.

УПРАЖНЕНИЯ

8.1. Сообщение. Напишите функцию `display_message()` для вывода сообщения по теме, рассматриваемой в этой главе. Вызовите функцию и убедитесь в том, что сообщение выводится правильно.

8.2. Любимая книга. Напишите функцию `favorite_book()`, которая получает один параметр `title`. Функция должна выводить сообщение вида «Одна из моих любимых книг — “Алиса в стране чудес”». Вызовите функцию и убедитесь в том, что название книги правильно передается как аргумент при вызове функции.

Передача аргументов

Определение функции может иметь несколько параметров, поэтому может оказаться, что при вызове функции должны передаваться несколько аргументов. Существует несколько способов передачи аргументов функциям. *Позиционные аргументы* перечисляются в порядке, точно соответствующем порядку записи параметров; *именованные аргументы* состоят из имени переменной и значения; наконец, существуют списки и словари значений. Рассмотрим все эти способы.

Позиционные аргументы

При вызове функции каждый аргумент должен быть сопоставлен с параметром в определении функции. Проще всего сделать это на основании порядка перечисления аргументов. Значения, связываемые с аргументами подобным образом, называются *позиционными аргументами* (positional arguments).

Чтобы понять, как работает эта схема, рассмотрим функцию для вывода информации о домашних животных. Функция сообщает тип животного и его имя:

pets.py

```
❶ def describe_pet(animal_type, pet_name):  
    """Выводит информацию о животном."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}")  
  
❷ describe_pet('hamster', 'harry')
```

Из определения видно, что функции должны передаваться тип животного (`animal_type`) и его имя (`pet_name`) ❶. При вызове `describe_pet()` необходимо передать тип и имя — именно в таком порядке. В этом примере аргумент `'hamster'` сохраняется в параметре `animal_type`, а аргумент `'harry'` — в параметре `pet_name` ❷. В теле функции эти два параметра используются для вывода информации о питомце — хомяке Гарри:

```
I have a hamster.  
My hamster's name is Harry.
```

Многократные вызовы функций

Функция может вызываться в программе столько раз, сколько потребуется. Для вывода информации о другом животном достаточно одного вызова `describe_pet()`:

```
def describe_pet(animal_type, pet_name):  
    """Выводит информацию о животном."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}")  
  
describe_pet('hamster', 'harry')  
describe_pet('dog', 'willie')
```

Во втором вызове функции `describe_pet()` передаются аргументы `'dog'` и `'willie'`. По аналогии с предыдущей парой аргументов Python сопоставляет аргумент `'dog'` с параметром `animal_type`, а аргумент `'willie'` с параметром `pet_name`. Как и в предыдущем случае, функция выполняет свою задачу, однако на этот раз выводятся другие значения. Теперь у нас есть хомяк по имени Гарри и собака по имени Вилли:

```
I have a hamster.  
My hamster's name is Harry.
```

```
I have a dog.  
My dog's name is Willie.
```


Многократный вызов функции — чрезвычайно эффективный механизм. Код вывода информации о домашнем животном пишется один раз в функции. Каждый раз, когда вам понадобится вывести информацию о новом животном, вы вызываете функцию с данными нового животного. Даже если код вывода информации разрастется до 10 строк, вы все равно сможете вывести информацию с помощью всего одной команды — для этого достаточно снова вызвать функцию.

Важность порядка позиционных аргументов

Если нарушить порядок следования аргументов в вызове при использовании позиционных аргументов, возможны неожиданные результаты:

```
def describe_pet(animal_type, pet_name):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}")

describe_pet('harry', 'hamster')
```

В этом вызове функции сначала передается имя, а затем тип животного. Аргумент 'harry' находится в первой позиции, поэтому значение сохраняется в параметре `animal_type`, а аргумент 'hamster' — в параметре `pet_name`. На этот раз вывод получается бессмысленным:

```
I have a harry.
My harry's name is Hamster.
```

Если вы получили подобные странные результаты, то проверьте, что порядок следования аргументов в вызове функции соответствует порядку параметров в ее определении.

Именованные аргументы

Именованный аргумент (keyword argument) представляет собой пару «имя — значение», передаваемую функции. Имя и значение связываются с аргументом напрямую, так что при передаче аргумента путаница с порядком исключается (вы не увидите в выводе «моего гарри зовут Хомяк»). Именованные аргументы избавляют от хлопот с порядком аргументов при вызове функции, а также проясняют роль каждого значения в вызове функции.

Перепишем программу `pets.py` с использованием именованных аргументов при вызове `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}")

describe_pet(animal_type='hamster', pet_name='harry')
```

Функция `describe_pet()` не изменилась. Однако на этот раз при вызове функции мы явно сообщаем Python, с каким параметром должен быть связан каждый аргумент. При обработке вызова функции Python знает, что аргумент `'hamster'` должен быть сохранен в параметре `animal_type`, а аргумент `'harry'` — в параметре `pet_name`.

Порядок следования именованных аргументов в данном случае неважен, поскольку Python знает, где должно храниться каждое значение. Следующие два вызова функции эквивалентны:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

ПРИМЕЧАНИЕ

При использовании именованных аргументов будьте внимательны — имена должны точно совпадать с именами параметров из определения функции.

Значения по умолчанию

Для каждого параметра вашей функции можно определить *значение по умолчанию* (default value). Если при вызове функции передается аргумент, соответствующий данному параметру, то Python использует значение аргумента, а если нет — значение по умолчанию. Таким образом, если для параметра определено значение по умолчанию, то вы можете опустить соответствующий аргумент, который обычно добавляется в вызов функции. Значения по умолчанию упрощают вызовы функций и проясняют типичные способы использования функций.

Например, если вы заметили, что большинство вызовов `describe_pet()` используется для описания собак, то задайте `animal_type` значение по умолчанию `'dog'`. Теперь в любом вызове `describe_pet()` для собаки эту информацию можно опустить:

```
def describe_pet(pet_name, animal_type='dog'):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}")

describe_pet(pet_name='Willie')
```

Мы изменили определение `describe_pet()` и добавили для параметра `animal_type` значение по умолчанию `'dog'`. Если теперь функция будет вызвана без указания `animal_type`, то Python знает, что для этого параметра следует использовать значение `'dog'`:

```
I have a dog.
My dog's name is Willie.
```

Обратите внимание: в определении функции пришлось изменить порядок параметров. Благодаря значению по умолчанию указывать аргумент с типом животного необязательно, поэтому единственным оставшимся аргументом в вызове функции остается имя домашнего животного. Python интерпретирует его как позиционный аргумент, и если функция вызывается только с именем животного, данный аргумент сопоставляется с первым параметром в определении функции. Именно поэтому первым параметром должно быть имя животного.

В простейшем варианте использования этой функции при вызове передается только имя собаки:

```
describe_pet('Willie')
```

Вызов функции выводит тот же результат, что и в предыдущем примере. Единственный переданный аргумент 'Willie' сопоставляется с первым параметром в определении — `pet_name`. Для `animal_type` аргумент не указан, поэтому Python использует значение по умолчанию — 'dog'.

Для вывода информации о любом другом животном, кроме собаки, используется вызов функции следующего вида:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Аргумент для параметра `animal_type` задан явно, поэтому Python игнорирует значение параметра по умолчанию.

ПРИМЕЧАНИЕ

Если вы используете значения по умолчанию, то все параметры со значением по умолчанию должны следовать после параметров, у которых значений по умолчанию нет. Это необходимо для того, чтобы Python правильно интерпретировал позиционные аргументы.

Эквивалентные вызовы функций

Позиционные и именованные аргументы, а также значения по умолчанию могут использоваться одновременно, поэтому часто существует несколько эквивалентных способов вызова функций. Возьмем следующий оператор `describe_pets()` с одним значением по умолчанию:

```
def describe_pet(pet_name, animal_type='dog'):
```

При таком определении аргумент для параметра `pet_name` должен задаваться в любом случае, но это значение может передаваться как в позиционном, так и в именованном формате. Если описываемое животное не является собакой, то аргумент

`animal_type` тоже должен быть добавлен в вызов, и этот аргумент тоже может быть задан как в позиционном, так и в именованном формате.

Все следующие вызовы являются допустимыми для данной функции:

```
# Собака Вилли.
describe_pet('Willie')
describe_pet(pet_name='Willie')

# Хомяк Гарри.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Все вызовы функции выдадут такой же результат, как и в предыдущих примерах.

На самом деле не так важно, какой стиль вызова вы используете. Если ваша функция выдает нужный результат, то выберите тот стиль, который вам кажется более понятным.

Предотвращение ошибок в аргументах

Не удивляйтесь, если на первых порах вашей работы с функциями будут встречаться ошибки несоответствия аргументов. Такие ошибки происходят в том случае, если вы передали меньше или больше аргументов, чем необходимо функции для выполнения ее работы. Например, вот что произойдет при попытке вызвать `describe_pet()` без аргументов:

```
def describe_pet(animal_type, pet_name):
    """Выводит информацию о животном."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}")

describe_pet()
```

Python распознает, что при вызове функции часть информации отсутствует, и мы видим это в данных трассировки:

Traceback (most recent call last):

- ❶ File "pets.py", line 6, in <module>
- ❷ `describe_pet()`
^^^^^^^^^^^^^^^^
- ❸ `TypeError: describe_pet() missing 2 required positional arguments: 'animal_type' and 'pet_name'`

В данных трассировки сначала сообщается местонахождение проблемы ❶, чтобы вы поняли, что с вызовом функции что-то пошло не так. Далее приводится вызов функции, приведший к ошибке ❷. И наконец, Python сообщает, что при вызове

пропущены два аргумента, и указывает их имена ❸. Если бы функция размещалась в отдельном файле, то, вероятно, вы смогли бы исправить вызов и вам не пришлось бы открывать этот файл и читать код функции.

Python помогает еще и тем, что читает код функции и сообщает имена аргументов, которые необходимо передать при вызове. Это еще одна причина присваивать переменным и функциям описательные имена. В этом случае сообщения об ошибках Python принесут больше пользы и вам, и любому другому разработчику, который будет использовать ваш код.

Если при вызове будут переданы лишние аргументы, то вы получите похожую трассировку, которая поможет привести вызов функции в соответствие с ее определением.

УПРАЖНЕНИЯ

8.3. Футболка. Напишите функцию `make_shirt()`, которая получает размер футболки и текст, который должен быть напечатан на ней. Функция должна выводить сообщение с размером и текстом.

Вызовите функцию с помощью позиционных аргументов. Вызовите функцию во второй раз с помощью именованных аргументов.

8.4. Большие футболки. Измените функцию `make_shirt()`, чтобы футболки по умолчанию имели размер L и на них выводился текст «Я люблю Python». Создайте футболку с размером L и текстом по умолчанию, а также футболку любого размера с другим текстом.

8.5. Города. Напишите функцию `describe_city()`, которая получает названия города и страны. Функция должна выводить простое сообщение (например, «Рейкьявик находится в Исландии»). Задайте параметру страны значение по умолчанию. Вызовите свою функцию для трех разных городов, по крайней мере один из которых не находится в стране по умолчанию.

Возвращаемое значение

Функция не обязана выводить результаты своей работы напрямую. Вместо этого она может обработать данные, а затем вернуть значение или набор сообщений. Значение, возвращаемое функцией, называется *возвращаемым значением* (return value). Оператор `return` передает значение из функции в точку программы, в которой эта функция была вызвана. Возвращаемые значения помогают переместить большую часть рутинной работы в вашей программе в функции, чтобы упростить основной код программы.

Возвращение простого значения

Рассмотрим функцию, которая получает имя и фамилию и возвращает отформатированное полное имя:

formatted_name.py

```
def get_formatted_name(first_name, last_name):
    """Возвращает отформатированное полное имя."""
    ❶ full_name = f"{first_name} {last_name}"
    ❷ return full_name.title()

❸ musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

Определение `get_formatted_name()` получает в параметрах имя и фамилию. Функция объединяет эти два имени, добавляет между ними пробел и сохраняет результат в `full_name` ❶. Значение `full_name` преобразуется: начальные буквы переводятся в верхний регистр, — а затем возвращается в строку вызова ❷.

Вызывая функцию, которая возвращает значение, необходимо предоставить переменную, в которой должно храниться это значение. В данном случае оно записывается в переменную `musician` ❸. Результат содержит отформатированное полное имя, созданное из имени и фамилии:

```
Jimi Hendrix
```

Может показаться, что все эти хлопоты излишни — с таким же успехом можно было использовать команду:

```
print("Jimi Hendrix")
```

Но если представить, что вы пишете большую программу, в которой многочисленные имена и фамилии должны храниться по отдельности, то такие функции, как `get_formatted_name()`, становятся чрезвычайно полезными. Вы храните имена отдельно от фамилий, а затем вызываете функцию везде, где потребуется вывести полное имя.

Необязательные аргументы

Иногда бывает удобно сделать аргумент необязательным, чтобы разработчик, использующий функцию, мог передать дополнительную информацию только в том случае, если захочет этого. Чтобы сделать аргумент необязательным, можно воспользоваться значением по умолчанию.

Допустим, вы захотели расширить функцию `get_formatted_name()`, чтобы она работала и со вторыми именами. Первая попытка могла бы выглядеть так:

```
def get_formatted_name(first_name, middle_name, last_name):
    """Возвращает отформатированное полное имя."""
```

```
full_name = f"{first_name} {middle_name} {last_name}"
return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

Функция работает при получении имени, второго имени и фамилии. Она получает все три части имени, а затем создает из них строку. После этого она добавляет пробелы там, где это уместно, и преобразует полное имя — переводит начальные буквы в верхний регистр (выполняет капитализацию):

John Lee Hooker

Однако вторые имена нужны не всегда, а в такой записи функция не будет работать, если при вызове ей передается только имя и фамилия. Чтобы средний аргумент был необязательным, можно присвоить аргументу `middle_name` пустое значение по умолчанию; этот аргумент игнорируется, если пользователь не передал для него значение. Чтобы функция `get_formatted_name()` работала без второго имени, следует назначить для параметра `middle_name` пустую строку значением по умолчанию и переместить его в конец списка параметров:

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Возвращает отформатированное полное имя."""
    ❶ if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    ❷ else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

❸ musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

В этом примере имя создается из трех возможных частей. Поскольку имя и фамилия указываются всегда, эти параметры стоят в начале списка в определении функции. Второе имя необязательно, поэтому находится на последнем месте в определении, а его значением по умолчанию является пустая строка.

В теле функции мы сначала проверяем, было ли задано второе имя. Python интерпретирует непустые строки как истинное значение, и если при вызове задан аргумент второго имени, то `middle_name` дает результат `True` ❶. Если второе имя указано, то из имени, второго имени и фамилии создается полное имя. Затем имя подвергается капитализации символов и возвращается в строку вызова функции, где сохраняется в переменной `musician` и выводится. Если второе имя не указано, то пустая строка не проходит проверку `if` и выполняет блок `else` ❷. В этом случае полное имя создается только из имени и фамилии и отформатированное имя возвращается в строку вызова, где сохраняется в переменной `musician` и выводится.

Вызов этой функции с именем и фамилией достаточно тривиален. Но при использовании второго имени придется проследить за тем, чтобы второе имя было последним из передаваемых аргументов. Это необходимо для правильного связывания позиционных аргументов ❸.

Обновленная версия этой функции подойдет как для людей, у которых задаются только имя и фамилия, так и для людей со вторым именем:

```
Jimi Hendrix  
John Lee Hooker
```

Необязательные значения позволяют функциям работать в максимально широком спектре сценариев использования, не усложняя вызовы.

Возвращение словаря

Функция может вернуть любое значение, которое вам потребуется, в том числе и более сложную структуру данных (например, список или словарь). Так, следующая функция получает части имени и возвращает словарь, представляющий человека:

person.py

```
def build_person(first_name, last_name):  
    """Возвращает словарь с информацией о человеке."""  
❶    person = {'first': first_name, 'last': last_name}  
❷    return person  
  
musician = build_person('jimi', 'hendrix')  
❸ print(musician)
```

Функция `build_person()` получает имя и фамилию и сохраняет полученные значения в словаре ❶. Значение `first_name` сохраняется с ключом `'first'`, а значение `last_name` — с ключом `'last'`. Затем весь словарь с описанием человека возвращается ❷. Значение выводится ❸ с двумя исходными фрагментами текстовой информации, теперь хранящимися в словаре:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Функция получает простую текстовую информацию и помещает ее в более удобную структуру данных, которая позволяет работать с информацией (помимо простого вывода). Строки `'jimi'` и `'hendrix'` теперь помечены как имя и фамилия. Функцию можно легко расширить так, чтобы она принимала дополнительные значения — второе имя, возраст, профессию или любую другую информацию о человеке, которую вы хотите сохранить. Например, следующее изменение позволяет сохранить возраст человека:

```
def build_person(first_name, last_name, age=''):  
    """Возвращает словарь с информацией о человеке."""  
    person = {'first': first_name, 'last': last_name}
```



```
if age:
    person['age'] = age
return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

В определение функции добавляется новый необязательный параметр `age`, которому присваивается специальное значение по умолчанию `None` — оно используется для переменных, которым не присвоено никакое значение. Можно рассматривать `None` как значение-заполнитель. При проверке условий `None` интерпретируется как `False`. Если вызов функции содержит значение параметра `age`, то оно сохраняется в словаре. Функция всегда сохраняет имя, но ее можно изменить, чтобы она сохраняла любую необходимую информацию о человеке.

Использование функции в цикле `while`

Функции могут использоваться со всеми структурами Python, уже известными вам. Например, задействуем функцию `get_formatted_name()` в цикле `while`, чтобы поприветствовать пользователей более официально. Первая версия программы, обращающейся к ним по имени и фамилии, может выглядеть так:

greeter.py

```
def get_formatted_name(first_name, last_name):
    """Возвращает отформатированное полное имя."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# Бесконечный цикл!
while True:
    ❶ print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

В этом примере приводится простая версия `get_formatted_name()`, не использующая вторые имена. В цикле `while` имя и фамилия пользователя запрашиваются по отдельности ❶.

Но у этого цикла `while` есть один недостаток: в нем не определено условие завершения. Где следует поместить условие завершения при запросе серии данных? Пользователю нужно предоставить возможность выйти из цикла как можно раньше, так что в приглашении должен содержаться способ завершения. Оператор `break` позволяет немедленно прервать цикл при запросе любого из компонентов:

```
def get_formatted_name(first_name, last_name):
    """Возвращает отформатированное полное имя."""
```

```
full_name = f"{first_name} {last_name}"
return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break

    l_name = input("Last name: ")
    if l_name == 'q':
        break

formatted_name = get_formatted_name(f_name, l_name)
print(f"\nHello, {formatted_name}!")
```

В программу добавляется сообщение, которое объясняет пользователю, как завершить ввод данных, и при вводе признака завершения в любом из приглашений цикл прерывается. Теперь программа будет приветствовать пользователя до тех пор, пока вместо имени или фамилии не будет введен символ 'q':

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
Last name: matthes
```

```
Hello, Eric Matthes!
```

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: q
```

УПРАЖНЕНИЯ

8.6. Названия городов. Напишите функцию `city_country()`, которая получает название города и страну. Функция должна возвращать строку в формате:

```
"Santiago, Chile"
```

Вызовите свою функцию по крайней мере для трех пар «город — страна» и выведите возвращенное значение.

8.7. Альбом. Напишите функцию `make_album()`, которая создает словарь с описанием музыкального альбома. Функция должна получать имя исполнителя и название альбома и возвращать словарь, содержащий эти два вида информации. Используйте функцию для создания трех словарей, представляющих разные альбомы. Выведите все возвращаемые значения, чтобы показать, что информация правильно сохраняется во всех трех словарях.

Добавьте в `make_album()` дополнительный параметр для сохранения количества дорожек в альбоме, имеющий значение по умолчанию `None`. Если в строке вызова есть значение количества дорожек, то добавьте это значение в словарь альбома. Создайте как минимум один новый вызов функции, который передает количество дорожек в альбоме.

8.8. Пользовательские альбомы. Начните с программы из упражнения 8.7. Напишите цикл `while`, в котором пользователь вводит данные об исполнителе и название альбома. Затем в цикле вызывается функция `make_album()` для введенных пользователем данных и выводится созданный словарь. Не забудьте предусмотреть признак завершения в цикле `while`.

Передача списка

Часто при вызове функции удобно передать список — имен, чисел или более сложных объектов (например, словарей). При передаче списка функция получает прямой доступ ко всему его содержимому. Мы воспользуемся функциями для того, чтобы сделать работу со списком более эффективной.

Допустим, вы хотите вывести приветствие для каждого пользователя из списка. В следующем примере список имен передается функции `greet_users()`, которая выводит приветствие для каждого пользователя по отдельности:

greet_users.py

```
def greet_users(names):
    """Выводит простое приветствие для каждого пользователя в списке."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

В соответствии со своим определением функция `greet_users()` рассчитывает получить список имен, который сохраняется в параметре `names`. Функция перебирает полученный список и выводит приветствие для каждого пользователя. Вне функции мы определяем список пользователей `usernames`, который затем передается `greet_users()` в вызове функции:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

Результат выглядит именно так, как ожидалось. Каждый пользователь получает персональное сообщение, и эту функцию можно вызвать для любого нового набора пользователей.

Изменение списка в функции

Если вы передаете список функции, то код функции сможет изменить список. Все изменения, внесенные в список в теле функции, являются постоянными, что позволяет эффективно работать со списком даже при больших объемах данных.

Допустим, компания печатает на 3D-принтере модели, предоставленные пользователем. Проекты хранятся в списке, а после печати перемещаются в отдельный список. В следующем примере приведена реализация, не использующая функции:

printing_models.py

```
# Список моделей, которые необходимо напечатать.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Цикл последовательно печатает каждую модель до конца списка.
# После печати каждая модель перемещается в список completed_models.
while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

# Вывод готовых моделей.
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

В начале программы создается список моделей и пустой список `completed_models`, в который каждая модель перемещается после печати. Пока в `unprinted_designs` остаются модели, цикл `while` имитирует печать каждой модели: текущая модель удаляется с конца списка, сохраняется в `current_design`, а пользователь получает сообщение о том, что она была напечатана. Затем модель перемещается в список напечатанных. После завершения цикла выводится список напечатанных моделей:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case
```

```
The following models have been printed:
dodecahedron
robot pendant
phone case
```

Мы можем изменить структуру кода: для этого следует написать две функции, каждая из которых решает одну конкретную задачу. Большая часть кода останется неизменной; просто программа становится более эффективной. Первая функция занимается печатью, а вторая выводит сводку напечатанных моделей:

```
❶ def print_models(unprinted_designs, completed_models):
    """
    Имитирует печать моделей, пока список не станет пустым.
```

Каждая модель после печати перемещается в `completed_models`.

```
while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)
```

```
❷ def show_completed_models(completed_models):
    """Выводит информацию обо всех напечатанных моделях."""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Сначала определяется функция `print_models()` с двумя параметрами: список моделей для печати и список готовых моделей ❶. С этими двумя списками функция имитирует печать каждой модели, последовательно извлекая модели из первого списка и перемещая их во второй. Затем определяется функция `show_completed_models()` с одним параметром: список напечатанных моделей ❷. Она получает этот список и выводит имена всех напечатанных моделей.

Программа выводит тот же результат, что и версия без функций, но структура кода значительно улучшилась. Код, выполняющий большую часть работы, разнесен по двум разным функциям; это упрощает чтение основной части программы. Теперь любому разработчику будет намного проще просмотреть код программы и понять, что она делает:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Программа создает список моделей для печати и пустой список для готовых моделей. Затем, поскольку обе функции уже определены, остается вызвать их и передать правильные аргументы. Мы вызываем `print_models()` и передаем два необходимых списка; как и ожидалось, эта функция имитирует печать моделей. Затем вызывается функция `show_completed_models()`, и ей передается список готовых моделей, чтобы она могла вывести информацию о напечатанных моделях. Благодаря описательным именам функций другой разработчик сможет прочитать этот код и понять его даже без комментариев.

Вдобавок эта программа создает меньше проблем с расширением и сопровождением, чем версия без функций. Если позднее потребуется напечатать новую

партию моделей, то достаточно снова вызвать `print_models()`. Если окажется, что код печати необходимо модифицировать, то изменения можно внести в одном месте, и они автоматически распространятся на все вызовы функции. Такой подход намного эффективнее независимой правки кода в нескольких местах программы.

В этом примере также демонстрируется принцип, в соответствии с которым каждая функция должна решать одну конкретную задачу. Первая функция печатает каждую модель, а вторая выводит информацию о готовых моделях. Такой подход предпочтительнее решения обеих задач в функции. Если вы пишете функцию и видите, что она решает слишком много разных задач, то попробуйте разделить ее код на две функции. Помните, что функции всегда можно вызывать из других функций. Эта возможность может пригодиться для разбиения сложных задач на серию составляющих.

Запрет изменения списка в функции

Иногда требуется предотвратить изменение списка в функции. Допустим, у вас есть список моделей для печати, и вы пишете функцию для перемещения их в список готовых моделей, как в предыдущем примере. Возможно, даже после печати всех моделей исходный список нужно оставить для отчетности. Но поскольку все имена моделей были перенесены из списка `unprinted_designs`, остался только пустой список; исходная версия списка потеряна. Проблему можно решить, передав функции копию списка вместо оригинала. В этом случае все изменения, которые она вносит в список, будут распространяться только на копию, а оригинал останется неизменным.

Чтобы передать функции копию списка, можно поступить так:

```
имя_функции(имя_списка[:])
```

Синтаксис среза `[:]` создает копию списка для передачи функции. Если удаление элементов из списка `unprinted_designs` в программе `print_models.py` нежелательно, то функцию `print_models()` можно вызвать так:

```
print_models(unprinted_designs[:], completed_models)
```

Функция `print_models()` может выполнить свою работу, поскольку все равно получает имена всех ненапечатанных моделей. Однако на этот раз она использует не сам список `unprinted_designs`, а его копию. Список `completed_models` заполняется именами напечатанных моделей, как и в предыдущем случае, но исходный список функция не изменяет.

Несмотря на то что передача копии позволяет сохранить содержимое списка, обычно функциям следует передавать исходный список (если у вас нет веских причин для передачи копии). Работа с существующим списком более эффективна, поскольку программе не приходится тратить время и память на создание отдельной копии (лишние затраты особенно заметны при работе с большими списками).

УПРАЖНЕНИЯ

8.9. Сообщения. Создайте список с серией коротких сообщений. Передайте список функции `show_messages()`, которая выводит текст каждого сообщения в списке.

8.10. Отправка сообщений. Начните с копии вашей программы из упражнения 8.9. Напишите функцию `send_messages()`, которая выводит каждое сообщение и перемещает его в новый список `sent_messages`. После вызова функции выведите оба списка и убедитесь в том, что перемещение прошло успешно.

8.11. Архивированные сообщения. Начните с программы из упражнения 8.10. Вызовите функцию `send_messages()`, чтобы создать копию списка сообщений. После вызова функции выведите оба списка и убедитесь в том, что в исходном списке остались все сообщения.

Передача произвольного набора аргументов

В некоторых ситуациях вы не знаете заранее, сколько аргументов должно быть передано функции. К счастью, Python позволяет ей получить произвольное количество аргументов из вызывающего оператора.

Для примера рассмотрим функцию для создания пиццы. Она должна получить набор начинок для пиццы, но вы не знаете заранее, сколько начинок закажет клиент. Функция в следующем примере получает один параметр `*toppings`, но он объединяет все аргументы, заданные в вызывающей строке:

pizza.py

```
def make_pizza(*toppings):
    """Выводит список заказанных начинок."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

Благодаря звездочке в имени параметра `*toppings` Python получает указание создать пустой кортеж `toppings` и упаковать в него все полученные значения. Результат вызова `print()` в теле функции показывает, что Python успешно вызывает обе функции: и с одним значением, и с тремя. Разные вызовы обрабатываются похожим образом. Обратите внимание: Python упаковывает аргументы в кортеж даже в том случае, если функция получает всего одно значение:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Теперь вызов функции `print()` можно заменить циклом, который перебирает список начинок и выводит описание заказанной пиццы:

```
def make_pizza(*toppings):
    """Выводит описание пиццы."""
```

```
print("\nMaking a pizza with the following toppings:")
for topping in toppings:
    print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

Функция реагирует соответственно, независимо от того, сколько значений получила — одно или три:

```
Making a pizza with the following toppings:
- pepperoni
```

```
Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Этот синтаксис работает независимо от количества аргументов, переданных функции.

Позиционные аргументы с произвольными наборами аргументов

Если вы хотите, чтобы функция могла вызываться с разным количеством аргументов, то параметр для получения произвольного количества аргументов должен стоять на последнем месте в определении функции. Python сначала подбирает соответствия для позиционных и именованных аргументов, а затем объединяет все остальные аргументы в последнем параметре.

Например, если функция должна получать размер пиццы, то данный параметр должен стоять в списке до параметра `*toppings`:

```
def make_pizza(size, *toppings):
    """Выводит описание пиццы."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

В определении функции Python сохраняет первое полученное значение в параметре `size`. Все остальные значения, следующие за ним, сохраняются в кортеже `toppings`. В вызовах функций на первом месте располагается аргумент для параметра `size`, а за ним следует сколько угодно начинок.

В итоге для каждой пиццы указываются размер и количество начинок, и каждый фрагмент информации выводится в положенном месте: сначала размер, а потом начинки:

```
Making a 16-inch pizza with the following toppings:
- pepperoni
```


Making a 12-inch pizza with the following toppings:

- mushrooms
- green peppers
- extra cheese

ПРИМЕЧАНИЕ

В программах часто используется имя обобщенного параметра `*args`, который служит для хранения произвольного набора позиционных аргументов.

Использование произвольного набора именованных аргументов

Иногда программа должна получать произвольное количество аргументов, но вы не знаете заранее, какая информация будет передаваться функции. В таких случаях можно написать функцию, получающую столько пар «ключ — значение», сколько указано в вызывающем операторе. Один из возможных примеров — создание пользовательских профилей: вы знаете, что получите информацию о пользователе, но заранее неизвестно, какую именно. Функция `build_profile()` в следующем примере всегда получает имя и фамилию, но может получать и произвольное количество именованных аргументов:

user_profile.py

```
def build_profile(first, last, **user_info):
    """Создает словарь, содержащий информацию о пользователе."""
    ❶ user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')

print(user_profile)
```

Определение `build_profile()` ожидает получить имя и фамилию пользователя, а также позволяет передать любое количество пар «имя — значение». Две звездочки перед параметром `**user_info` заставляют Python создать пустой словарь `user_info` и упаковать в него все полученные пары «имя — значение». Внутри функции вы можете обращаться к парам «имя — значение» из `user_info` точно так же, как в любом словаре.

В теле `build_profile()` в словарь `user_info` добавляются имя и фамилия, поскольку эти два значения всегда передаются пользователем ❶ и они еще не были помещены в словарь. Затем словарь `user_info` возвращается в точку вызова функции.

Вызовем функцию `build_profile()` и передадим ей имя 'albert', фамилию 'einstein' и еще две пары «ключ — значение»: `location='princeton'` и `field='physics'`.

Программа сохраняет возвращенный словарь в `user_profile` и выводит его содержимое:

```
{'location': 'princeton', 'field': 'physics',  
'first_name': 'albert', 'last_name': 'einstein'}
```

Возвращаемый словарь содержит имя и фамилию пользователя, а в данном случае еще и местонахождение, и область исследований. Функция будет работать, сколько бы дополнительных пар «ключ — значение» ни было передано при вызове функции.

При написании функций допускаются самые разнообразные комбинации позиционных, именованных и произвольных значений. Полезно знать о существовании всех этих типов аргументов, поскольку они часто будут встречаться вам при чтении чужого кода. Только практикуясь, вы научитесь правильно использовать разные типы аргументов и поймете, когда следует применять каждый тип; а пока просто используйте самый простой способ, который позволит решить задачу. С опытом вы научитесь выбирать наиболее эффективный вариант для каждой конкретной ситуации.

ПРИМЕЧАНИЕ

В программах часто используется имя обобщенного параметра `**kwargs`, который служит для хранения произвольного набора ключевых аргументов.

УПРАЖНЕНИЯ

8.12. Бутерброды. Напишите функцию, которая получает список компонентов бутерброда. Функция должна иметь один параметр для любого количества значений, переданных при вызове функции, и выводить описание заказанного бутерброда. Вызовите функцию три раза с разными количествами аргументов.

8.13. Профиль. Начните с копии программы `user_profile.py`, приведенной в этом подразделе. Создайте собственный профиль с помощью вызова `build_profile()`, укажите имя, фамилию и три другие пары «ключ — значение» для вашего описания.

8.14. Автомобили. Напишите функцию для сохранения данных об автомобиле в словаре. Она всегда должна возвращать информацию о производителе и названии модели, но при этом может получать произвольное количество именованных аргументов. Вызовите функцию с передачей обязательной информации и еще двух пар «имя — значение» (например, цвет и комплектация). Ваша функция должна работать для вызовов следующего вида:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Выведите возвращаемый словарь и убедитесь в том, что вся информация была сохранена правильно.

Хранение функций в модулях

Одно из преимуществ функций заключается в том, что они отделяют блоки кода от основной программы. Если для функций были выбраны описательные имена, то вашу программу будет намного проще читать. Вы можете пойти еще дальше и сохранить функции в отдельном файле, называемом *модулем*, а затем *импортировать* модуль в свою программу. Оператор `import` сообщает Python, что код модуля должен быть доступен в текущем выполняемом программном файле.

Хранение функций в отдельных файлах позволяет скрыть второстепенные детали кода и сосредоточиться на логике более высокого уровня. Кроме того, функции можно использовать во множестве разных программ. Функции, хранящиеся в отдельных файлах, можно передать другим программистам, не распространяя полный код программы. А умение импортировать функции позволит вам использовать библиотеки функций, написанные другими программистами.

Существует несколько способов импортирования модулей; все они кратко рассматриваются ниже.

Импортирование модуля целиком

Чтобы заняться импортированием функций, сначала необходимо создать *модуль*. Это файл с расширением `.py`, содержащий код, который вы хотите импортировать в свою программу. Создадим модуль с функцией `make_pizza()`. Для этого из файла `pizza.py` следует удалить все, кроме функции `make_pizza()`:

pizza.py

```
def make_pizza(size, *toppings):
    """Выводит описание пиццы."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

Теперь создадим отдельный файл `making_pizzas.py` в одном каталоге с `pizza.py`. Файл импортирует только что созданный модуль, а затем дважды вызывает функцию `make_pizza()`:

making_pizzas.py

```
import pizza
```

```
❶ pizza.make_pizza(16, 'pepperoni')
   pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

В процессе обработки этого файла благодаря строке `import pizza` Python получает указание открыть файл `pizza.py` и скопировать все функции из него в программу. Вы не видите, как происходит копирование, поскольку Python копирует код незаметно для пользователя, пока выполняется программа. Вам необходимо знать одно: любая функция, определенная в `pizza.py`, будет доступна в `making_pizzas.py`.

Чтобы вызвать функцию из импортированного модуля, введите имя модуля (`pizza`), точку и имя функции (`make_pizza()`) ❶. Код выдает тот же результат, что и исходная программа, в которой модуль не импортировался:

```
Making a 16-inch pizza with the following toppings:  
- pepperoni
```

```
Making a 12-inch pizza with the following toppings:  
- mushrooms  
- green peppers  
- extra cheese
```

Первый способ импортирования, при котором пишется оператор `import` с именем модуля, открывает доступ программе ко всем функциям из модуля. Если вы используете эту разновидность оператора `import` для импортирования всего модуля `имя_модуля.py`, то каждая функция модуля будет доступна в следующем синтаксисе:

```
имя_модуля.имя_функции()
```

Импортирование конкретных функций из модуля

Общий синтаксис для импортирования конкретной функции из модуля выглядит так:

```
from имя_модуля import имя_функции
```

Вы можете импортировать любое количество функций из модуля, разделив их имена запятыми:

```
from имя_модуля import функция_0, функция_1, функция_2
```

Если ограничиться импортированием лишь той функции, которую вы намереваетесь использовать, то пример `making_pizzas.py` будет выглядеть так:

```
from pizza import make_pizza  
  
make_pizza(16, 'pepperoni')  
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

При таком синтаксисе использовать точечную нотацию (`dot notation`) при вызове функции необязательно. Функция `make_pizza()` явно импортируется в операторе `import`, поэтому при использовании ее можно вызывать прямо по имени.

Назначение псевдонима для функции

Если имя импортируемой функции может конфликтовать с именем существующей или является слишком длинным, то его можно заменить коротким уникальным *псевдонимом* (`alias`) — альтернативным именем для функции. Псевдоним назначается функции при импортировании.

В следующем примере функции `make_pizza()` назначается псевдоним `mp()`, для чего при импортировании используется конструкция `make_pizza as mp`. Ключевое слово `as` переименовывает функцию, используя указанный псевдоним:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Оператор `import` в этом примере назначает функции `make_pizza()` псевдоним `mp()` для этой программы. Каждый раз, когда потребуется вызвать `make_pizza()`, достаточно добавить вызов `mp()` — Python выполнит код `make_pizza()`, избегая конфликтов с другой функцией `make_pizza()`, которую вы могли добавить в этот файл программы.

Общий синтаксис назначения псевдонима выглядит так:

```
from имя_модуля import имя_функции as псевдоним
```

Назначение псевдонима для модуля

Псевдоним можно назначить для всего модуля. Назначение короткого имени для модуля — скажем, `p` для `pizza` — позволит вам быстрее вызывать функции модуля. Вызов `p.make_pizza()` получается более компактным, чем `pizza.make_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Модулю `pizza` в операторе `import` назначается псевдоним `p`, но все функции модуля сохраняют свои исходные имена. Вызов функций в записи `p.make_pizza()` не только компактнее `pizza.make_pizza()`, но и отвлекает внимание от имени модуля и помогает сосредоточиться на описательных именах функций. Благодаря этим именам, четко показывающим, что делает каждая функция, ваш код читать намного удобнее, чем если бы вы использовали полное имя модуля.

Общий синтаксис выглядит так:

```
import имя_модуля as псевдоним
```

Импортирование всех функций модуля

Можно дать Python указание импортировать каждую функцию в модуле; для этого используется оператор `*`:

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Благодаря звездочке в коде оператора `import` Python получает указание скопировать каждую функцию из модуля `pizza` в файл программы. После импортирования всех функций вы сможете вызывать каждую из них по имени, не используя точечную нотацию. Тем не менее лучше не использовать этот способ с большими модулями, написанными другими разработчиками; если модуль содержит функцию, имя которой совпадает с существующим именем из вашего проекта, то возможны неожиданные результаты. Python обнаруживает несколько функций или переменных с одинаковыми именами, и вместо импортирования всех функций по отдельности происходит их замена.

В таких ситуациях лучше всего импортировать только нужную функцию или функции либо импортировать весь модуль и использовать точечную нотацию. При этом создается чистый код, легкочитаемый и понятный. Я добавил этот подраздел только для того, чтобы вы могли распознать подобные операторы `import`, когда встретите их в чужом коде:

```
from имя_модуля import *
```

Форматирование функций

При форматировании функций необходимо учитывать некоторые подробности. Функции должны иметь описательные имена, состоящие из букв нижнего регистра и символов подчеркивания, — они помогают вам и другим разработчикам понять, что делает ваш код. Эти правила следует соблюдать и при создании имен модулей.

Каждая функция должна быть снабжена комментарием, который кратко поясняет, что она делает. Он должен следовать сразу за определением функции и быть оформлен как строки документации. Если функция хорошо документирована, то другие разработчики смогут использовать ее, прочитав только описание. Конечно, для этого они должны доверять тому, что код работает в соответствии с описанием. Но если разработчики знают имя функции, необходимые ей аргументы и какое значение она возвращает, то смогут использовать ее в своих программах.

Если для параметра задается значение по умолчанию, то слева и справа от знака равенства не должно быть пробелов:

```
def имя_функции(параметр_0, параметр_1='значение_по_умолчанию')
```

Те же правила должны применяться для именованных аргументов в вызовах функций:

```
имя_функции(значение_0, параметр_1='значение')
```

Документ PEP 8 (<https://www.python.org/dev/peps/pep-0008/>) рекомендует ограничить длину строк кода 79 символами, чтобы строки были полностью видны в окне редактора нормального размера. Если из-за параметров длина определения функции превышает 79 символов, то нажмите клавишу **Enter** после открывающей круглой скобки в строке определения. В следующей строке дважды нажмите клавишу **Tab**, чтобы отделить список аргументов от тела функции, в котором должен быть только один отступ.

Многие редакторы автоматически выравнивают дополнительные строки параметров по отступам, установленным в первой строке:

```
def имя_функции(
    параметр_0, параметр_1, параметр_2,
    параметр_3, параметр_4, параметр_5):
    тело функции...
```

Если программа или модуль состоит из нескольких функций, то их можно разделить двумя пустыми строками. Так вам будет проще увидеть, где кончается одна функция и начинается другая.

Все операторы **import** следует указывать в начале файла. У этого правила есть только одно исключение: файл может начинаться с комментариев, описывающих программу в целом.

УПРАЖНЕНИЯ

8.15. Печать моделей. Выделите функции примера `print_models.py` в отдельный файл `printing_functions.py`. Укажите оператор **import** в начале файла `print_models.py` и измените файл так, чтобы в нем использовались импортированные функции.

8.16. Импортирование. Возьмите за основу одну из написанных вами программ с одной функцией. Сохраните эту функцию в отдельном файле. Импортируйте ее в файл основной программы и вызовите каждым из следующих способов:

```
import имя_модуля
from имя_модуля import имя_функции
from имя_модуля import имя_функции as псевдоним
import имя_модуля as псевдоним
from имя_модуля import *
```

8.17. Форматирование функций. Выберите любые три программы, написанные для этой главы. Убедитесь в том, что в них соблюдаются рекомендации по форматированию, представленные в этом разделе.

Резюме

В этой главе вы научились писать функции и добавлять аргументы, в которых функциям передается информация, необходимая для их работы. Вы узнали, как использовать позиционные и именованные аргументы и как передать функции произвольное количество аргументов. Вы увидели функции, которые выводят данные, и функции, которые возвращают значения. Вы научились использовать функции со списками, словарями, операторами `if` и циклами `while`. Кроме того, вы узнали, как сохранять функции в отдельных файлах, называемых *модулями*, чтобы код ваших программ стал проще и понятнее. В завершение главы вы изучили рекомендации по форматированию функций, которые помогут вам улучшить структуру ваших программ и упростить их чтение и вами, и другими разработчиками.

Каждый программист должен стремиться к написанию простого кода, который справляется с поставленной задачей, и функции помогают в этом. Вы сможете писать блоки кода и оставлять их на будущее. Если вы знаете, что функция правильно справляется со своей задачей, — считайте, что она работает, и переходите к следующей задаче.

В программе с использованием функций единожды написанный код может заново использоваться сколько угодно раз. Чтобы выполнить код, содержащийся в функции, достаточно написать всего одну строку с вызовом, а функция сделает все остальное. Если же потребуется модифицировать поведение функции, то достаточно внести изменения всего в одном месте; они вступят в силу во всех местах кода, где вызывается эта функция.

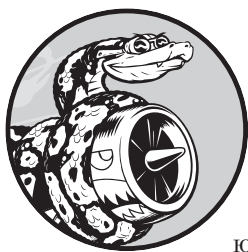
Использование функций упрощает чтение ваших программ, а хорошо выбранные имена функций описывают, что делает та или иная часть программы. Прочитав серию вызовов функций, вы гораздо быстрее поймете, что делает функция, чем если бы читали длинную серию программных блоков.

Помимо вышесказанного, функции упрощают тестирование и отладку кода. Когда основная работа программы выполняется с помощью набора функций, каждая из которых решает одну конкретную задачу, вам будет намного проще тестировать и сопровождать ваш код. Напишите отдельную программу, которая вызывает каждую функцию и проверяет ее работоспособность во всех типичных ситуациях. В этом случае вы можете быть уверены в том, что ваши функции всегда работают правильно.

В главе 9 вы научитесь писать *классы*. Они объединяют функции и данные в один удобный пакет, который можно эффективно использовать для решения разных задач.

9

Классы



Объектно-ориентированное программирование (ООП) по праву считается одной из самых эффективных методологий создания программных продуктов. В ООП вы пишете *классы*, описывающие реально существующие предметы и ситуации, а затем на основе этих описаний создаете *объекты*. При написании класса определяется общее поведение для целой категории объектов.

Когда вы создаете на базе классов конкретные объекты, каждый из них автоматически наделяется общим поведением; после этого вы можете прописать для каждого объекта уникальные особенности на свой вкус. Просто невероятно, насколько хорошо реальные ситуации моделируются в объектно-ориентированном программировании.

Создание объекта на основе класса называется *созданием экземпляра*; таким образом, вы работаете с *экземплярами* класса. В этой главе вы будете писать классы и создавать их экземпляры. Вы будете указывать, какая информация может храниться в экземплярах, и определять действия, которые могут выполняться с экземплярами. Кроме того, вы будете писать новые классы, расширяющие функциональность уже существующих; это позволяет организовать эффективное совместное использование кода похожими классами. Вы будете сохранять классы в модулях и импортировать классы, написанные другими программистами, в ваши программные файлы.

Имея хорошее понимание объектно-ориентированного программирования, вы взглянете на мир с точки зрения программиста. Вам будет проще понять свой код — увидеть не только то, что происходит в каждой его строке, но и более масштабные концепции, на которых он строится. Логика, заложенная в основу классов, научит вас мыслить логически, чтобы ваши программы эффективно решали практически любые задачи, с которыми вы можете столкнуться.

Кроме того, классы упрощают жизнь вам и другим программистам, с которыми вам придется работать совместно над более серьезными проектами. Когда вы и другие программисты пишете код, базирующийся на сходной логике, вам будет проще разобраться в коде, написанном другими людьми. Ваши программы будут понятны коллегам, и в результате все вы сможете добиться больших результатов.

Создание и использование класса

Классы позволяют моделировать практически все, что угодно. Начнем с написания простого класса `Dog`, представляющего собаку — не какую-то конкретную, а собаку вообще. Что мы знаем о собаках? У них есть кличка и возраст, а еще большинство собак умеют садиться и перекатываться по команде. Эти два вида информации (кличка и возраст) и два вида поведения (сидеть и перекатываться) будут добавлены в класс `Dog`, поскольку являются общими для большинства собак. Класс сообщает Python, как создать объект, представляющий собаку. После того как класс будет написан, мы используем его для создания экземпляров, каждый из которых соответствует одной конкретной собаке.

Создание класса `Dog`

В каждом экземпляре, созданном на основе класса `Dog`, будут храниться данные о кличке (`name`) и возрасте (`age`); кроме того, в нем будут присутствовать функции `sit()` и `roll_over()`:

dog.py

```
❶ class Dog:
    """Простая модель собаки."""

❷     def __init__(self, name, age):
        """Инициализирует атрибуты name и age."""

❸         self.name = name
        self.age = age

❹     def sit(self):
        """Имитирует, как собака садится по команде."""
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        """Имитирует, как собака перекатывается по команде."""
        print(f"{self.name} rolled over!")
```

В этом коде есть много мест, заслуживающих вашего внимания, но не беспокойтесь. Эта структура неоднократно встретится вам в данной главе, и вы еще успеете к ней привыкнуть. Сначала мы определили класс `Dog` ❶. Согласно общепринятым правилам, имена, начинающиеся с символа верхнего регистра, в Python обозначают классы. В определении класса нет круглых скобок, поскольку он создается с нуля. Ниже приведена строка документации с кратким описанием класса.

Метод `__init__()`

Функция, являющаяся частью класса, называется *методом*. Все, что вы узнали ранее о функциях, относится и к методам; единственное практическое различие — способ вызова методов. Метод `__init__()` ❷ — специальный метод, который автоматически выполняется при создании каждого нового экземпляра на базе класса `Dog`. Имя метода начинается и заканчивается двумя символами подчеркивания; эта схема предотвращает конфликты имен стандартных методов Python и методов ваших классов. Будьте внимательны: два символа подчеркивания должны стоять *по обе стороны*: `__init__()`. Если вы поставите только один символ подчеркивания с каждой стороны, то метод не будет вызываться автоматически при использовании класса, что может привести к появлению ошибок, которые сложно обнаружить.

Метод `__init__()` определяется с тремя параметрами: `self`, `name` и `age`. Параметр `self` обязателен; он должен предшествовать всем остальным параметрам. Его следует добавить в определение, поскольку при будущем вызове метода `__init__()` (для создания экземпляра `Dog`) Python автоматически передает аргумент `self`. При каждом вызове метода, связанного с классом, автоматически передается `self` — ссылка на экземпляр; она предоставляет конкретному экземпляру доступ к атрибутам и методам класса. Когда мы создаем экземпляр `Dog`, Python вызывает метод `__init__()` из класса `Dog`. Мы передаем `Dog()` кличку и возраст в аргументах; значение `self` передается автоматически — вам не нужно это делать. Каждый раз, когда вы захотите создать экземпляр на основе класса `Dog`, необходимо предоставить значения только двух последних аргументов: `name` и `age`.

Каждая из двух переменных, определяемых в теле метода `__init__()`, снабжена префиксом `self` ❸. Переменная с этим префиксом доступна для любого метода в классе, и вы также сможете обращаться к этим переменным в каждом экземпляре, созданном на основе класса. Конструкция `self.name = name` берет значение, хранящееся в параметре `name`, и сохраняет его в переменной `name`, которая затем связывается с создаваемым экземпляром. Процесс повторяется и с `self.age = age`. Переменные, к которым вы обращаетесь через экземпляры, также называются *атрибутами*.

Кроме того, в классе `Dog` определяются два метода: `sit()` и `roll_over()` ❹. Им не нужна дополнительная информация (кличка или возраст), поэтому они определяются с единственным параметром `self`. Экземпляры, которые будут созданы позднее, смогут вызывать эти методы. Пока данные методы ограничиваются простым выводом сообщения о том, что собака садится или перекачивается. Тем не менее концепцию легко расширить для практического применения: если бы этот класс был частью компьютерной игры, то эти методы вполне могли бы содержать код, позволяющий создать анимацию сажащейся или перекачивающейся собаки. А если бы класс был написан для манипулирования роботом, то методы могли бы управлять механизмами, заставляющими робота-собаку выполнить соответствующую команду.

Создание экземпляра класса

Считайте, что класс — это своего рода инструкция по созданию экземпляров. Соответственно, класс `Dog` — инструкция по созданию экземпляров, представляющих конкретных собак.

Создадим экземпляр для конкретной собаки:

```
class Dog:
    --пропуск--
```

- ❶ `my_dog = Dog('Willie', 6)`
- ❷ `print(f"My dog's name is {my_dog.name}.")`
- ❸ `print(f"My dog is {my_dog.age} years old.")`

Использованный в данном случае класс `Dog` был написан в предыдущем примере. Python получает указание — создать экземпляр собаки с кличкой 'Willie' и возрастом 6 ❶. В процессе обработки этой строки Python вызывает метод `__init__()` класса `Dog` с аргументами 'Willie' и 6. Метод `__init__()` создает экземпляр, представляющий конкретную собаку, и присваивает его атрибутам `name` и `age` переданные значения. Затем Python возвращает экземпляр, представляющий собаку. Он сохраняется в переменной `my_dog`. Здесь нелишне вспомнить правила о записи имен: обычно считается, что имя, начинающееся с символа верхнего регистра (например, `Dog`), обозначает класс, а имя, записанное в нижнем регистре (например, `my_dog`), обозначает отдельный экземпляр, созданный на базе класса.

Обращение к атрибутам

Для обращения к атрибутам экземпляра используется точечная нотация. Мы обращаемся к значению атрибута `name` ❷ экземпляра `my_dog`:

```
my_dog.name
```

Точечная нотация часто используется в Python. Этот синтаксис показывает, как Python ищет значения атрибутов. В данном случае Python обращается к экземпляру `my_dog` и ищет атрибут `name`, связанный с экземпляром `my_dog`. Это тот же атрибут, который обозначался как `self.name` в классе `Dog`. Тот же прием используется для работы с атрибутом `age` ❸.

Пример выводит известную информацию о `my_dog`:

```
My dog's name is Willie.
My dog is 6 years old.
```

Вызов методов

После создания экземпляра на основе класса `Dog` можно применять точечную нотацию для вызова любых методов, определенных в `Dog`:

```
class Dog:
    --пропуск--

my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

Чтобы вызвать метод, укажите экземпляр (в данном случае `my_dog`) и вызываемый метод, разделив их точкой. В ходе обработки `my_dog.sit()` Python ищет метод `sit()` в классе `Dog` и выполняет его код. Строка `my_dog.roll_over()` интерпретируется аналогичным образом.

Теперь экземпляр послушно выполняет полученные команды:

```
Willie is now sitting.
Willie rolled over!
```

Это очень полезный синтаксис. Если атрибутам и методам были присвоены описательные имена (например, `name`, `age`, `sit()` и `roll_over()`), то разработчик сможет легко понять, что делает блок кода, — даже если видит его впервые.

Создание нескольких экземпляров

На основе класса можно создать сколько угодно экземпляров. Создадим второй экземпляр `Dog` с именем `your_dog`:

```
class Dog:
    --пропуск--

my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

В этом примере создаются два экземпляра с именами `Willie` и `Lucy`. Каждый экземпляр обладает своим набором атрибутов и способен выполнять действия из общего набора:

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.
```

```
Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Даже если второй собаке будут назначены те же имя и возраст, Python все равно создаст отдельный экземпляр класса `Dog`. Вы можете создать сколько угодно экземпляров одного класса при условии, что они хранятся в переменных с различными именами или занимают разные позиции в списке или словаре.

УПРАЖНЕНИЯ

9.1. Ресторан. Создайте класс `Restaurant`. Его метод `__init__()` должен содержать два атрибута: `restaurant_name` и `cuisine_type`. Создайте метод `describe_restaurant()`, который выводит два атрибута, и метод `open_restaurant()`, который выводит сообщение о том, что ресторан открыт.

Создайте на основе своего класса экземпляр `restaurant`. Выведите два атрибута по отдельности, затем вызовите оба метода.

9.2. Три ресторана. Начните с класса из упражнения 9.1. Создайте три разных экземпляра и вызовите для каждого экземпляра метод `describe_restaurant()`.

9.3. Пользователи. Создайте класс `User` и два атрибута `first_name` и `last_name`, а затем еще несколько атрибутов, которые обычно хранятся в профиле пользователя. Напишите метод `describe_user()`, который выводит сводку с информацией о пользователе. Создайте еще один метод — `greet_user()`, позволяющий вывести персональное приветствие для пользователя.

Создайте несколько экземпляров, представляющих разных пользователей. Вызовите оба метода для каждого пользователя.

Работа с классами и экземплярами

Классы могут использоваться для моделирования многих реальных ситуаций. После того как класс будет написан, разработчик проводит большую часть времени за работой с экземплярами, созданными на основе этого класса. Одной из первых задач станет изменение атрибутов, связанных с конкретным экземпляром. Атрибуты экземпляра можно изменять напрямую или же написать методы, изменяющие их по особым правилам.

Класс *Car*

Напишем класс, который представляет автомобиль. Он будет содержать информацию о типе машины, а также метод для вывода краткого описания:

car.py

```
class Car:
    """Простая модель автомобиля."""
```

```
❶ def __init__(self, make, model, year):
    """Инициализирует атрибуты описания автомобиля."""
    self.make = make
    self.model = model
    self.year = year

❷ def get_descriptive_name(self):
    """Возвращает отформатированное описание."""
    long_name = f"{self.year} {self.make} {self.model}"
    return long_name.title()

❸ my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
```

В классе `Car` определяется метод `__init__()`; его список параметров начинается с `self` ❶, как и в классе `Dog`. За ним следуют еще три параметра: `make`, `model` и `year`. Метод `__init__()` получает их и сохраняет в атрибутах, которые будут связаны с экземплярами, созданными на основе класса. При создании нового экземпляра `Car` необходимо указать фирму-производителя, модель и год выпуска для данного экземпляра.

Далее мы определяем метод `get_descriptive_name()` ❷, который объединяет данные о годе выпуска, фирме-производителе и модели в одну строку с описанием. Это избавит нас от необходимости выводить значение каждого атрибута по отдельности. Для работы со значениями атрибутов в этом методе используется синтаксис `self.make`, `self.model` и `self.year`. Вне класса мы создаем экземпляр класса `Car`, который сохраняется в переменной `my_new_car` ❸. Затем вызываем метод `get_descriptive_name()`, чтобы увидеть, с какой машиной работает программа:

```
2024 Audi A4
```

Чтобы класс был более интересным, добавим атрибут, изменяющийся со временем, — в нем будут храниться данные о пробеге машины в милях.

Назначение атрибуту значения по умолчанию

При создании экземпляра можно определить атрибуты, не передавая их в качестве параметров. Это можно сделать в методе `__init__()`, где атрибутам присваивается значение по умолчанию.

Добавим атрибут `odometer_reading`, исходное значение которого всегда равно 0. Кроме того, в класс добавим метод `read_odometer()`, позволяющий читать текущие показания одометра:

```
class Car:

    def __init__(self, make, model, year):
        """Инициализирует атрибуты описания автомобиля."""
        self.make = make
```

```

        self.model = model
        self.year = year
        self.odometer_reading = 0

    ❶ def get_descriptive_name(self):
        --пропуск--

    ❷ def read_odometer(self):
        """Выводит данные о пробеге машины в милях."""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Когда Python вызывает метод `__init__()` для создания нового экземпляра, этот метод сохраняет данные о фирме-производителе, модели и годе выпуска в атрибутах, как и в предыдущем случае. Затем Python создает новый атрибут `odometer_reading` и присваивает ему исходное значение 0 ❶. Кроме того, в класс добавляется новый метод `read_odometer()` ❷, который упрощает чтение данных о пробеге машины в милях.

Сразу же после создания машины ее пробег равен 0:

```
2024 Audi A4
This car has 0 miles on it.
```

Впрочем, у продаваемых машин одометр редко показывает ровно 0, поэтому нам понадобится способ изменения значения этого атрибута.

Изменение значений атрибутов

Значение атрибута можно изменить одним из трех способов: изменить его прямо в экземпляре, задать значение с помощью метода или изменить его с приращением (то есть прибавлением определенной величины), используя метод. Рассмотрим все эти способы.

Прямое изменение значения атрибута

Чтобы изменить значение атрибута, проще всего обратиться к нему через экземпляр. В следующем примере на одометре напрямую выставляется значение 23:

```

class Car:
    --пропуск--

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```


Точечная запись используется для обращения к атрибуту `odometer_reading` экземпляра и прямого присваивания его значения. Благодаря этой строке Python получает указание взять экземпляр `my_new_car`, найти связанный с ним атрибут `odometer_reading` и задать его значение равным 23:

```
2024 Audi A4
This car has 23 miles on it.
```

Иногда подобные прямые обращения к атрибутам допустимы, но чаще разработчик пишет вспомогательный метод, который изменяет значение за него.

Изменение значения атрибута с помощью метода

В класс можно добавить методы, которые изменяют определенные атрибуты за вас. Вместо того чтобы задавать атрибут напрямую, вы передаете новое значение методу, который берет обновление атрибута на себя.

В следующем примере в класс добавлен метод `update_odometer()`:

```
class Car:
    --пропуск--

    def update_odometer(self, mileage):
        """Устанавливает заданное значение на одометре."""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
```

```
❶ my_new_car.update_odometer(23)
   my_new_car.read_odometer()
```

Код класса `Car` почти не изменился, в нем только добавился метод `update_odometer()`. Этот метод получает данные о пробеге в милях и сохраняет их в `self.odometer_reading`. Мы вызываем метод `update_odometer()` и передаем ему значение 23 в аргументе ❶. Метод устанавливает на одометре значение 23, а метод `read_odometer()` выводит текущие показания:

```
2024 Audi A4
This car has 23 miles on it.
```

Метод `update_odometer()` можно расширить так, чтобы при каждом изменении показаний одометра выполнялась некая дополнительная работа. Добавим проверку, которая гарантирует, что никто не будет пытаться сбрасывать показания одометра:

```
class Car:
    --пропуск--

    def update_odometer(self, mileage):
```

```

"""
Устанавливает на одометре заданное значение.
При попытке обратной подкрутки изменение отклоняется.
"""
❶ if mileage >= self.odometer_reading:
    self.odometer_reading = mileage
❷ else:
    print("You can't roll back an odometer!")

```

Теперь `update_odometer()` проверяет новое значение перед изменением атрибута. Если новое значение `mileage` больше текущего пробега или равно ему, `self.odometer_reading`, то показания одометра можно обновить с помощью нового значения ❶. Если же новое значение меньше текущего, то вы получите предупреждение о недопустимости обратной подкрутки ❷.

Изменение значения атрибута с приращением

Иногда значение атрибута требуется изменить с заданным приращением (вместо того чтобы присваивать атрибуту произвольное новое значение). Допустим, вы купили поддержанную машину и проехали на ней 100 миль с момента покупки. Следующий метод получает величину приращения и прибавляет ее к текущим показаниям одометра:

```

class Car:
    --пронуск--

    def update_odometer(self, mileage):
        --пронуск--

    def increment_odometer(self, miles):
        """Увеличивает показания одометра с заданным приращением."""
        self.odometer_reading += miles

❶ my_used_car = Car('subaru', 'outback', 2019)
  print(my_used_car.get_descriptive_name())

❷ my_used_car.update_odometer(23_500)
  my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()

```

Новый метод `increment_odometer()` получает расстояние в милях и прибавляет его к `self.odometer_reading`. Сначала создается экземпляр `my_used_car` ❶. Мы инициализируем показания его одометра значением 23 500; для этого вызывается метод `update_odometer()`, которому передается значение 23_500 ❷. Наконец, вызывается метод `increment_odometer()`, которому передается значение 100, чтобы увеличить показания одометра на 100 миль, которые автомобиль проехал с момента покупки:

```
2019 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

При желании можно легко усовершенствовать этот метод, чтобы он отклонял отрицательные приращения; тем самым вы предотвратите обратную подкрутку одометра.

ПРИМЕЧАНИЕ

Подобные методы управляют обновлением внутренних значений экземпляров (таких как показания одометра), однако любой пользователь, имеющий доступ к программному коду, сможет напрямую задать атрибуту любое значение. Создавая эффективную схему безопасности, нужно уделять особое внимание таким подробностям, не ограничиваясь простейшими проверками.

УПРАЖНЕНИЯ

9.4. Посетители. Начните с программы из упражнения 9.1. Добавьте атрибут `number_served` со значением по умолчанию 0; он представляет количество обслуженных посетителей. Создайте экземпляр `restaurant`. Выведите значение `number_served`, потом измените и выведите снова.

Добавьте метод `set_number_served()`, позволяющий задать количество обслуженных посетителей. Вызовите его с новым числом, снова выведите значение.

Добавьте метод `increment_number_served()`, который увеличивает количество обслуженных посетителей на заданную величину. Вызовите его с любым числом, которое могло бы представлять количество обслуженных клиентов — скажем, за один день.

9.5. Попытки входа. Добавьте атрибут `login_attempts` в класс `User` из упражнения 9.3. Напишите метод `increment_login_attempts()`, увеличивающий значение `login_attempts` на 1. Напишите еще один метод, `reset_login_attempts()`, обнуляющий значение `login_attempts`.

Создайте экземпляр класса `User` и вызовите `increment_login_attempts()` несколько раз. Выведите значение `login_attempts`, чтобы убедиться в том, что значение было изменено правильно, а затем вызовите `reset_login_attempts()`. Снова выведите `login_attempts` и убедитесь в том, что значение обнулилось.

Наследование

Работа над новым классом не обязана начинаться с нуля. Если класс, который вы пишете, представляет собой специализированную версию ранее написанного класса, то вы можете воспользоваться *наследованием* (inheritance). Один класс, *наследующий* от другого, автоматически получает все атрибуты и методы первого

класса. Исходный класс называется *родителем* (parent), а новый класс — *потомком* (child class). Класс-потомок наследует атрибуты и методы класса-родителя, но при этом может определять и собственные атрибуты и методы.

Метод `__init__()` класса-потомка

При написании нового класса на базе существующего часто приходится вызывать метод `__init__()` класса-родителя. При этом происходит инициализация любых атрибутов, определенных в данном методе, и они становятся доступными для класса-потомка.

Например, попробуем создать модель электромобиля. Электромобиль представляет собой специализированную разновидность автомобиля, поэтому новый класс `ElectricCar` можно создать на базе класса `Car`, написанного ранее. Тогда нам останется добавить в него код атрибутов и поведения, относящегося только к электромобилям.

Начнем с создания простой версии класса `ElectricCar`, который делает все, что делает класс `Car`:

`electric_car.py`

```
❶ class Car:
    """Простая модель автомобиля."""

    def __init__(self, make, model, year):
        """Инициализирует атрибуты описания автомобиля."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Возвращает отформатированное описание."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Выводит данные о пробеге машины в милях."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """Устанавливает на одометре заданное значение."""
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Увеличивает показания одометра с заданным приращением."""
        self.odometer_reading += miles
```

```
❷ class ElectricCar(Car):  
    """Представляет аспекты машины, специфические для электромобилей."""  
  
❸    def __init__(self, make, model, year):  
        """Инициализирует атрибуты класса-родителя."""  
❹        super().__init__(make, model, year)  
  
❺ my_leaf = ElectricCar('nissan', 'leaf', 2024)  
    print(my_leaf.get_descriptive_name())
```

Сначала создается класс `Car` ❶. При создании класса-потомка класс-родитель должен быть частью текущего файла, а его определение должно предшествовать определению класса-потомка в файле. Затем определяется класс-потомок `ElectricCar` ❷. В определении потомка имя класса-родителя заключается в круглые скобки. Метод `__init__()` получает информацию, необходимую для создания экземпляра `Car` ❺.

Функция `super()` ❹ — специальная; она позволяет вызвать метод родительского класса. Благодаря этой строке Python получает указание вызвать метод `__init__()` класса `Car`, в результате чего экземпляр `ElectricCar` имеет доступ ко всем атрибутам класса-родителя. Имя `super` происходит из общепринятой терминологии: класс-родитель называется *суперклассом*, а класс-потомок — *подклассом*.

Чтобы проверить, правильно ли сработало наследование, попробуем создать электромобиль с такой же информацией, которая передается при создании обычного экземпляра `Car`. Мы создаем экземпляр класса `ElectricCar` и сохраняем его в `my_leaf` ❺. Эта строка вызывает метод `__init__()`, определенный в `ElectricCar`, который, в свою очередь, дает Python указание вызвать метод `__init__()`, определенный в классе-родителе `Car`. При вызове передаются аргументы `'nissan'`, `'leaf'` и `2024`.

Кроме `__init__()`, класс еще не содержит никаких атрибутов или методов, специфических для электромобилей. Пока мы просто убеждаемся в том, что класс электромобиля содержит все поведение, присущее классу `Car`:

```
2024 Nissan Leaf
```

Экземпляр `ElectricCar` работает так же, как экземпляр `Car`. Теперь можно переходить к определению атрибутов и методов, специфических для электромобилей.

Определение атрибутов и методов класса-потомка

После создания класса-потомка, наследующего от класса-родителя, можно переходить к добавлению новых атрибутов и методов, необходимых для того, чтобы потомок отличался от родителя.

Добавим атрибут, специфический для электромобилей (например, мощность аккумулятора), и метод для вывода информации об этом атрибуте:

```
class Car:
    --пропуск--

class ElectricCar(Car):
    """Представляет аспекты машины, специфические для электромобилей."""
    def __init__(self, make, model, year):
        """
        Инициализирует атрибуты класса-родителя.
        Затем инициализирует атрибуты, специфические для электромобиля.
        """
        super().__init__(make, model, year)
    ❶ self.battery_size = 40

    ❷ def describe_battery(self):
        """Выводит информацию о мощности аккумулятора."""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.describe_battery()
```

Сначала добавляется новый атрибут `self.battery_size`, которому присваивается исходное значение — скажем, 40 ❶. Он будет присутствовать во всех экземплярах, созданных на основе класса `ElectricCar` (но не во всяком экземпляре `Car`). Затем добавляется метод `describe_battery()`, который выводит информацию об аккумуляторе ❷. При вызове этого метода выдается описание, которое явно относится только к электромобилям:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
```

Возможности специализации класса `ElectricCar` беспредельны. Вы можете добавить сколько угодно атрибутов и методов, чтобы моделировать электромобиль, задавая любую нужную точность. Атрибуты или методы, которые могут принадлежать любой машине (а не только электромобилю), должны добавляться в класс `Car` вместо `ElectricCar`. Тогда эта информация будет доступна всем пользователям класса `Car`, а класс `ElectricCar` будет содержать только код информации и поведения, специфический для электромобилей.

Переопределение методов класса-родителя

Любой метод родительского класса, который в моделируемой ситуации делает не то, что нужно, можно переопределить. Для этого в классе-потомке определяется метод с тем же именем, что и у метода класса-родителя. Python игнорирует метод родителя и обращает внимание только на метод, определенный в потомке.

Допустим, в классе `Car` имеется метод `fill_gas_tank()`. Для электромобилей заправка бензином бессмысленна, поэтому этот метод логично переопределить. Например, это можно сделать так:

```
class ElectricCar(Car):
    --пропуск--

    def fill_gas_tank(self):
        """У электромобилей нет бензобака."""
        print("This car doesn't have a gas tank!")
```

И если кто-то попытается вызвать метод `fill_gas_tank()` для электромобиля, то Python игнорирует метод `fill_gas_tank()` класса `Car` и выполнит вместо него этот код. Когда вы используете наследование, потомок сохраняет те аспекты родителя, которые вам нужны, и переопределяет все ненужное.

Экземпляры как атрибуты

При моделировании явлений реального мира в программах классы нередко дополняются все большим количеством подробностей. Списки атрибутов и методов растут, и через какое-то время файлы становятся чрезмерно длинными. В такой ситуации часть одного класса нередко можно записать в виде отдельного класса. Большой код разбивается на меньшие классы, которые работают во взаимодействии друг с другом. Такой подход называется *композицией* (composition).

Например, при дальнейшей доработке класса `ElectricCar` может оказаться, что в нем появилось слишком много атрибутов и методов, относящихся к аккумулятору. В таком случае можно остановиться и переместить все эти атрибуты и методы в отдельный класс `Battery`. Затем экземпляр `Battery` становится атрибутом класса `ElectricCar`:

```
class Car:
    --пропуск--

class Battery:
    """Простая модель аккумулятора электромобиля."""

    ❶ def __init__(self, battery_size=40):
        """Инициализирует атрибуты аккумулятора."""
        self.battery_size = battery_size

    ❷ def describe_battery(self):
        """Выводит информацию о мощности аккумулятора."""
        print(f"This car has a {self.battery_size}-kWh battery.")

class ElectricCar(Car):
    """Представляет аспекты машины, специфические для электромобилей."""
```

```
def __init__(self, make, model, year):
    """
    Инициализирует атрибуты класса-родителя.
    Затем инициализирует атрибуты, специфические для электромобиля.
    """
    super().__init__(make, model, year)
    self.battery = Battery()

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
```

Сначала определяется новый класс `Battery`, который не наследует ни от одного из других классов. Метод `__init__()` ❶ получает один параметр `battery_size`, кроме `self`. Если значение не предоставлено, то этот необязательный параметр задает `battery_size` значение 40. Метод `describe_battery()` также перемещен в этот класс ❷.

Затем в класс `ElectricCar` добавляется атрибут `self.battery` ❸. Эта строка дает Python указание создать новый экземпляр `Battery` (со значением `battery_size` по умолчанию, равным 40, поскольку значение не задано) и сохранить его в атрибуте `self.battery`. Это будет происходить при каждом вызове `__init__()`; теперь любой экземпляр `ElectricCar` будет иметь автоматически создаваемый экземпляр `Battery`.

Программа создает экземпляр электромобиля и сохраняет его в переменной `my_leaf`. Когда потребуется вывести описание аккумулятора, необходимо обратиться к атрибуту `battery`:

```
my_leaf.battery.describe_battery()
```

Благодаря этой строке Python получает указание обратиться к экземпляру `my_leaf`, найти его атрибут `battery` и вызвать метод `describe_battery()`, связанный с экземпляром `Battery` из атрибута.

Результат выглядит так же, как и в предыдущей версии:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
```

Казалось бы, новый вариант требует большой дополнительной работы, но теперь аккумулятор можно моделировать, задавая любую степень детализации, при этом не усложняя класс `ElectricCar`. Добавим в `Battery` еще один метод, который выводит данные о запасе хода на основании мощности аккумулятора:

```
class Car:
    --пропуск--

class Battery:
    --пропуск--
```



```
def get_range(self):
    """Выводит данные о приблизительном запасе хода для аккумулятора."""
    if self.battery_size == 40:
        range = 150
    elif self.battery_size == 65:
        range = 225

    print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    --пропуск--

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
❶ my_leaf.battery.get_range()
```

Новый метод `get_range()` проводит простой анализ. Если мощность равна 40 кВт/ч, то `get_range()` устанавливает запас хода 150 миль, а при мощности 65 кВт/ч запас равен 225 милям. Затем программа выводит это значение. Когда вы захотите использовать этот метод, его придется вызывать через атрибут `battery` ❶.

Результат сообщает данные о запасе хода машины в зависимости от мощности аккумулятора:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
This car can go approximately 150 miles on a full charge.
```

Моделирование объектов реального мира

Занявшись моделированием более сложных объектов, таких как электромобили, вы столкнетесь со множеством интересных вопросов. Чьим свойством является запас хода электромобиля: аккумулятора или машины? Если вы описываете только одну машину, то, вероятно, можно связать метод `get_range()` с классом `Battery`. Но если моделируется целая линейка машин от производителя, то, вероятно, метод `get_range()` правильнее переместить в класс `ElectricCar`. Метод `get_range()` по-прежнему будет проверять мощность аккумулятора перед определением запаса хода, но будет сообщать запас хода для той машины, с которой он связан. Кроме того, можно связать метод `get_range()` с аккумулятором, но передавать ему параметр (например, `car_model`). Метод `get_range()` будет определять запас хода на основании мощности аккумулятора и модели автомобиля.

Если вы начнете задумываться над такими вопросами, это будет означать, что вы мыслите на более высоком логическом уровне, не ограничиваясь уровнем синтаксиса. Вы думаете уже не о Python, а о том, как лучше представить реальный мир в своем коде. И достигнув этой точки, вы поймете, что однозначно правильного или

неправильного подхода к моделированию реальных ситуаций часто не существует. Одни методы эффективнее других, но для того, чтобы найти наиболее эффективную реализацию, необходим практический опыт. Если ваш код работает именно так, как вы хотели, — значит, у вас все получается! Не огорчайтесь, если окажется, что вы по несколько раз переписываете свои классы для разных решений. Через этот процесс проходят все программисты, стараясь написать точный, эффективный код.

УПРАЖНЕНИЯ

9.6. Киоск с мороженым. Киоск с мороженым — особая разновидность ресторана. Напишите класс `IceCreamStand`, наследуемый от класса `Restaurant` из упражнения 9.1 или 9.4. Подойдет любая версия класса; просто выберите ту, которая вам больше нравится. Добавьте атрибут `flavors` для хранения списка сортов мороженого. Напишите метод, который выводит этот список. Создайте экземпляр `IceCreamStand` и вызовите данный метод.

9.7. Администратор. Администратор — особая разновидность пользователя. Напишите класс `Admin`, наследуемый от класса `User` из упражнения 9.3 или 9.5. Добавьте атрибут `privileges` для хранения списка строк вида "разрешено добавлять сообщения", "разрешено удалять пользователей", "разрешено банить пользователей" и т. д. Напишите метод `show_privileges()` для вывода набора привилегий администратора. Создайте экземпляр `Admin` и вызовите свой метод.

9.8. Привилегии. Напишите класс `Privileges`. Класс должен содержать всего один атрибут `privileges` со списком строк из упражнения 9.7. Переместите метод `show_privileges()` в этот класс. Создайте экземпляр `Privileges` как атрибут класса `Admin`. Создайте новый экземпляр `Admin` и используйте свой метод для вывода списка привилегий.

9.9. Обновление аккумулятора. Используйте окончательную версию программы `electric_car.py` из этого раздела. Добавьте в класс `Battery` метод `upgrade_battery()`. Он должен проверять размер аккумулятора и устанавливать мощность равной 65, если она имеет другое значение. Создайте экземпляр электромобиля с аккумулятором по умолчанию, вызовите `get_range()`, а затем вызовите его еще раз после `upgrade_battery()`. Убедитесь в том, что запас хода увеличился.

Импортирование классов

По мере добавления новой функциональности в классы файлы могут стать слишком длинными даже при правильном использовании наследования и композиции. В соответствии с общей философией Python файлы не должны содержать лишние подробности. Для этого Python позволяет хранить классы в модулях и импортировать нужные классы в основную программу.

Импортирование одного класса

Начнем с создания модуля, содержащего только класс `Car`. При этом возникает неочевидный конфликт имен: в текущей главе уже был создан файл `car.py`, но этот модуль тоже должен называться `car.py`, поскольку в нем содержится код класса `Car`. Мы решим данную проблему, сохранив класс `Car` в модуле `car.py`, заменяя им файл `car.py`, который использовался ранее. В дальнейшем любой программе, использующей этот модуль, придется присвоить более конкретное имя файла — например, `my_car.py`. Ниже приведен файл `car.py` с кодом класса `Car`:

`car.py`

```
❶ """Класс для представления автомобиля."""

class Car:
    """Простая модель автомобиля."""

    def __init__(self, make, model, year):
        """Инициализирует атрибуты описания автомобиля."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Возвращает отформатированное описание."""
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Выводит данные о пробеге машины в милях."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """
        Устанавливает на одометре заданное значение.
        При попытке обратной подкрутки изменение отклоняется.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Увеличивает показания одометра с заданным приращением."""
        self.odometer_reading += miles
```

Мы добавляем строку документации уровня модуля с кратким описанием содержимого модуля ❶. Пишите строки документации для каждого созданного вами модуля.

Теперь создадим отдельный файл `my_car.py`. Он импортирует класс `Car` и создает экземпляр данного класса:

`my_car.py`

❶ `from car import Car`

```
my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
```

```
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

Оператор `import` ❶ дает Python указание открыть модуль `car` и импортировать класс `Car`. Теперь мы можем использовать этот класс, как если бы он был определен в данном файле. Результат остается тем же, что и в предыдущей версии:

```
2024 Audi A4
This car has 23 miles on it.
```

Импортирование классов повышает эффективность программирования. Представьте, каким длинным получился бы файл этой программы, если бы в него был включен весь класс `Car`. Перемещая класс в модуль и импортируя этот модуль, вы получаете ту же функциональность, но основной файл программы при этом остается чистым и удобочитаемым. Кроме того, большая часть логики может храниться в отдельных файлах; когда ваши классы работают как положено, вы можете забыть об этих файлах и сосредоточиться на высокоуровневой логике основной программы.

Хранение нескольких классов в модуле

В одном модуле можно хранить сколько угодно классов, хотя все они должны быть как-то связаны друг с другом. Оба класса, `Battery` и `ElectricCar`, используются для представления автомобилей, поэтому мы добавим их в модуль `car.py`:

`car.py`

```
"""Классы для представления машин с бензиновым и электродвигателем."""
```

```
class Car:
    --пропуск--
```

```
class Battery:
    """Простая модель аккумулятора электромобиля."""
```

```
    def __init__(self, battery_size=40):
        """Инициализирует атрибуты аккумулятора."""
        self.battery_size = battery_size
```

```
    def describe_battery(self):
        """Выводит информацию о мощности аккумулятора."""
        print(f"This car has a {self.battery_size}-kWh battery.")
```

```

def get_range(self):
    """Выводит данные о приблизительном запасе хода для аккумулятора."""
    if self.battery_size == 40:
        range = 150
    elif self.battery_size == 65:
        range = 225

    print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """Представляет аспекты машины, специфические для электромобилей."""

    def __init__(self, make, model, year):
        """
        Инициализирует атрибуты класса-родителя.
        Затем инициализирует атрибуты, специфические для электромобиля.
        """
        super().__init__(make, model, year)
        self.battery = Battery()

```

Теперь вы можете создать новый файл `my_electric_car.py`, импортировать класс `ElectricCar` и создать новый экземпляр электромобиля:

my_electric_car.py

```

from car import ElectricCar

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
my_leaf.battery.get_range()

```

Программа выводит тот же результат, что и в предыдущем случае, хотя бóльшая часть ее логики скрыта в модуле:

```

2024 Nissan Leaf
This car has a 40-kWh battery.
This car can go approximately 150 miles on a full charge.

```

Импортирование нескольких классов из модуля

В файл программы можно импортировать столько классов, сколько понадобится. Если вы захотите создать обычный автомобиль и электромобиль в одном файле, то потребуется импортировать оба класса, `Car` и `ElectricCar`:

my_cars.py

- ❶ `from car import Car, ElectricCar`
- ❷ `my_mustang = Car('ford', 'mustang', 2024)`
`print(my_mustang.get_descriptive_name())`
- ❸ `my_leaf = ElectricCar('nissan', 'leaf', 2024)`
`print(my_leaf.get_descriptive_name())`

Чтобы импортировать несколько классов из модуля, разделите их имена запятыми ❶. После того как необходимые классы будут импортированы, вы можете создать столько экземпляров каждого класса, сколько потребуется.

В этом примере создается обычный автомобиль Ford Mustang ❷ и электромобиль Nissan Leaf ❸:

```
2024 Ford Mustang
2024 Nissan Leaf
```

Импортирование модуля целиком

Можно импортировать весь модуль, а потом обращаться к нужным классам, используя точечную запись. Этот способ прост, а полученный код легко читается. Каждый вызов, создающий экземпляр класса, содержит имя модуля, поэтому в программе не будет конфликтов с именами, используемыми в текущем файле.

Вот как выглядят импорт всего модуля `car`, а затем создание обычного автомобиля и электромобиля:

my_cars.py

```
❶ import car

❷ my_mustang = car.Car('ford', 'mustang', 2024)
  print(my_mustang.get_descriptive_name())

❸ my_leaf = car.ElectricCar('nissan', 'leaf', 2024)
  print(my_leaf.get_descriptive_name())
```

Сначала импортируется весь модуль `car` ❶, после чего программа обращается к нужным классам, используя синтаксис *имя_модуля.ИмяКласса*. Затем снова создаются экземпляр Ford Mustang ❷ и экземпляр Nissan Leaf ❸.

Импортирование всех классов из модуля

Для импортирования всех классов из модуля используется следующий синтаксис:

```
from имя_модуля import *
```

Применять этот способ не рекомендуется по двум причинам. Прежде всего бывает полезно прочитать операторы `import` в начале файла и получить четкое представление о том, какие классы используются в программе. А этот способ не позволяет понять, какие классы из модуля нужны программе. Кроме того, возможны конфликты с именами в файле. Если вы случайно импортируете класс с именем, уже присутствующим в файле, то в программе могут возникнуть ошибки, которые трудно выявить. Почему я привожу описание этого способа? Хотя использовать его не рекомендуется, скорее всего, вы встретите его в коде других разработчиков.

Итак, если вам нужно импортировать большое количество классов из модуля, то лучше импортировать весь модуль и воспользоваться синтаксисом *имя_модуля.ИмяКласса*. Вы не видите перечень всех используемых классов в начале файла, но по крайней мере понятно, где модуль используется в программе. К тому же предотвращаются потенциальные конфликты имен, которые могут возникнуть при импортировании каждого класса в модуле.

Импортирование модуля в модуль

Иногда классы приходится распределять по нескольким модулям, чтобы избежать чрезмерного разрастания одного файла и хранения несвязанных классов в одном модуле. При хранении классов в нескольких модулях может оказаться, что класс из одного модуля зависит от класса из другого модуля. В таких случаях необходимый класс можно импортировать в первый модуль.

Допустим, класс `Car` хранится в одном модуле, а классы `ElectricCar` и `Battery` — в другом. Мы создадим новый модуль `electric_car.py` (он заменит файл `electric_car.py`, созданный ранее) и скопируем в него только классы `Battery` и `ElectricCar`:

electric_car.py

```
"""Набор классов для представления электромобилей."""
```

```
from car import Car
```

```
class Battery:
    --пропуск--
```

```
class ElectricCar(Car):
    --пропуск--
```

Классу `ElectricCar` необходим доступ к классу-родителю `Car`, поэтому класс `Car` импортируется прямо в модуль. Если вы забудете вставить эту команду, то при попытке создания экземпляра `ElectricCar` произойдет ошибка. Кроме того, необходимо обновить модуль `Car`, чтобы он содержал только класс `Car`:

car.py

```
"""Простая модель автомобиля."""
```

```
class Car:
    --пропуск--
```

Теперь вы можете импортировать классы из каждого модуля по отдельности и создать ту разновидность машины, которая вам нужна:

my_cars.py

```
from car import Car
from electric_car import ElectricCar
```

```
my_mustang = Car('ford', 'mustang', 2024)
print(my_mustang.get_descriptive_name())

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

Сначала класс `Car` импортируется из своего модуля, а класс `ElectricCar` — из своего. После этого создаются экземпляры обоих разновидностей. Вывод показывает, что экземпляры были созданы правильно:

```
2024 Ford Mustang
2024 Nissan Leaf
```

Использование псевдонимов

Как было показано в главе 8, псевдонимы весьма полезны при использовании модулей для организации кода проектов. Кроме того, они позволят импортировать классы.

Для примера возьмем программу, которая должна создать группу экземпляров электрических машин. Многократно вводить (и читать) имя `ElectricCar` будет очень утомительно. Имени `ElectricCar` можно назначить псевдоним в операторе `import`:

```
from electric_car import ElectricCar as EC
```

С этого момента вы сможете использовать этот псевдоним каждый раз, когда вам потребуется создать экземпляр `ElectricCar`:

```
my_leaf = EC('nissan', 'leaf', 2024)
```

Вы также можете присвоить псевдоним модулю. Ниже показано, как импортировать модуль `electric_car` целиком, используя псевдоним:

```
import electric_car as ec
```

Теперь вы можете использовать псевдоним этого модуля с полным именем класса:

```
my_leaf = ec.ElectricCar('nissan', 'leaf', 2024)
```

Выработка рабочего процесса

Как видите, Python предоставляет много возможностей структурирования кода в крупных проектах. Вы должны знать все эти возможности, чтобы найти эффективные способы организации своих проектов, а также лучше понимать код других разработчиков.

На первых порах постарайтесь поддерживать простую структуру своего кода. Попробуйте поместить весь код в один файл, и только когда все заработает, переместите классы в отдельные модули. Если вам нравится схема взаимодействия между

модулями и файлами, то попробуйте сохранить классы в модулях в начале работы над проектом. Найдите подход, при котором у вас получается работоспособный код, и продолжайте работу.

УПРАЖНЕНИЯ

9.10. Импортрование класса Restaurant. Возьмите последнюю версию класса `Restaurant` и сохраните ее в модуле. Создайте отдельный файл, импортирующий этот класс. Создайте экземпляр `Restaurant` и вызовите один из методов `Restaurant`, чтобы показать, что оператор `import` работает правильно.

9.11. Импортрование класса Admin. Начните с версии класса из упражнения 9.8. Сохраните классы `User`, `Privileges` и `Admin` в одном модуле. Создайте отдельный файл, затем экземпляр `Admin` и вызовите метод `show_privileges()`, чтобы показать, что все работает правильно.

9.12. Множественные модули. Сохраните класс `User` в одном модуле, а классы `Privileges` и `Admin` — в другом. В отдельном файле создайте экземпляр `Admin` и вызовите метод `show_privileges()`, чтобы показать, что все работает правильно.

Стандартная библиотека Python

Стандартная библиотека Python (Python standard library) представляет собой набор модулей, добавляемых в каждую установленную копию Python. Сейчас вы уже примерно понимаете, как работают классы, и можете начать использовать модули, написанные другими программистами. Чтобы использовать любую функцию или класс из стандартной библиотеки, достаточно добавить простой оператор `import` в начало файла. Для примера рассмотрим модуль `random`, который может пригодиться для моделирования многих реальных ситуаций.

В частности, модуль `random` содержит интересную функцию `randint()`. Она получает два целочисленных аргумента и возвращает случайно выбранное целое число в диапазоне, определяемом этими двумя числами (включительно).

В следующем примере генерируется случайное число в диапазоне от 1 до 6:

```
>>> from random import randint
>>> randint(1, 6)
3
```

Другая полезная функция `choice()` получает список или кортеж и возвращает случайно выбранный элемент:

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

Модуль `random` не должен использоваться при создании приложений, связанных с безопасностью, но его возможностей достаточно для применения во многих интересных и увлекательных проектах.

ПРИМЕЧАНИЕ

Модули можно скачивать из внешних источников. Соответствующие примеры встретятся вам в части II, в которой мы будем использовать внешние модули для завершения работы над проектами.

УПРАЖНЕНИЯ

9.13. Игра в кости. Создайте класс `Die` с одним атрибутом `sides`, который имеет значение по умолчанию 6. Напишите метод `roll_die()` для вывода случайного числа от 1 до количества граней на кубике. Создайте экземпляр, представляющий шестигранный кубик, и смоделируйте десять бросков.

Создайте экземпляры, представляющие 10- и 20-гранный кубик. Смоделируйте десять бросков каждого кубика.

9.14. Лотерея. Создайте список или кортеж, содержащий серию из десяти чисел и пяти букв. Случайным образом выберите четыре числа или буквы из списка. Выведите сообщение о том, что билет, содержащий эту комбинацию из четырех цифр или букв, является выигрышным.

9.15. Анализ лотереи. Напишите цикл, который проверяет, насколько сложно выиграть в смоделированной вами лотерее. Создайте список или кортеж `my_ticket`. Напишите цикл, который продолжает генерировать комбинации до тех пор, пока не выпадет выигрышная комбинация. Выведите сообщение с информацией о том, сколько выполнений цикла понадобилось для получения выигрышной комбинации.

9.16. Модуль недели. Для знакомства со стандартной библиотекой Python отлично подойдет сайт под названием Python Module of the Week. Откройте сайт <http://pymotw.com/> и просмотрите оглавление. Найдите модуль, который покажется вам интересным; прочитайте его описание или изучите документацию по модулю `random`.

Оформление классов

В форматировании классов есть несколько моментов, о которых стоит упомянуть отдельно, — особенно учитывая тот факт, что ваши программы будут усложняться.

Имена классов должны записываться в *верблюжьем регистре* (CamelCase): первая буква каждого слова записывается в верхнем регистре, слова не разделяются пробелами. Имена экземпляров и модулей записываются в нижнем регистре, слова разделяются символами подчеркивания.

Каждый класс должен иметь строку документации, следующую сразу же за определением класса. В ней вы должны представить краткое описание того, что делает класс, соблюдая те же правила форматирования, которые вы использовали при написании строк документации в функциях. Каждый модуль тоже должен содержать строку документации, в которой описывается возможное применение классов в модуле.

Пустые строки можно использовать для структурирования кода, но злоупотреблять ими не стоит. В классах можно разделять методы одной пустой строкой, а в модулях для разделения классов можно использовать две пустые строки.

Если вам потребуется импортировать модуль из стандартной библиотеки и модуль из библиотеки, написанной вами, то начните с оператора `import` для модуля стандартной библиотеки. Затем добавьте пустую строку и оператор `import` для модуля, написанного вами. В программах с несколькими операторами `import` выполнение этого правила поможет понять, откуда берутся разные модули, использованные в программе.

Резюме

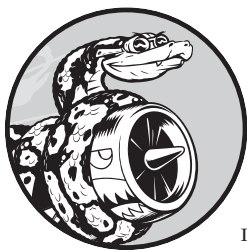
В этой главе вы узнали, как написать собственные классы. Вы научились хранить информацию в классе с помощью атрибутов и наделять свои классы требуемым поведением. Вы узнали, как написать методы `__init__()`, позволяющие создавать экземпляры ваших классов с нужными значениями атрибутов, и как изменять атрибуты экземпляров напрямую и через методы. Кроме того, вы увидели, что наследование может упростить создание логически связанных классов и экземпляры одного класса могут использоваться как атрибуты другого класса в целях упрощения кода классов.

Вы узнали, что хранение классов в модулях и импортирование необходимых классов в файлы, где они будут использоваться, улучшает организацию проектов. Вы познакомились со стандартной библиотекой Python и рассмотрели пример, основанный на модуле `random`. Наконец, вы научились оформлять свои классы в соответствии с общепринятыми правилами Python.

В главе 10 вы научитесь работать с файлами и сохранять результаты работы, выполненной в программе. Вдобавок вы познакомитесь с *исключениями* — экземплярами специального класса Python, предназначенного для передачи информации о возникающих ошибках.

10

Файлы и исключения



Вы уже овладели основными навыками, необходимыми для создания хорошо структурированных и удобных в использовании программ; теперь пора подумать о том, как сделать ваши программы еще более удобными и полезными. В этой главе вы научитесь работать с файлами, чтобы ваши программы могли быстро анализировать большие объемы данных.

Вы научитесь обрабатывать ошибки, чтобы возникновение аномальных ситуаций не приводило к аварийному завершению ваших программ. Мы рассмотрим *исключения* (exceptions) — специальные объекты, которые создаются для управления ошибками, возникающими во время выполнения программ Python. Вдобавок будет описан модуль `json`, позволяющий сохранять пользовательские данные, чтобы они не терялись при завершении работы программы.

Работа с файлами и сохранение данных упрощают использование ваших программ. Пользователь сам выбирает, какие данные и когда нужно вводить. Он может запустить программу, выполнить некую работу, потом закрыть программу и позднее продолжить работу с того момента, на котором он остановился. Умея обрабатывать исключения, вы сможете справиться с такими ситуациями, как отсутствие нужных файлов, а также другими проблемами, приводящими к сбою программ. Обработка исключений повысит устойчивость ваших программ при работе с некорректными данными — как появившимися из-за случайных ошибок, так и злонамеренными попытками взлома ваших программ. Используя материал, представленный в этой главе, вы сделаете ваши программы более практичными, удобными и надежными.

Чтение из файла

Гигантские объемы данных доступны в текстовых файлах. В них могут храниться погодные данные, социально-экономическая информация, литературные произведения и многое другое. Чтение из файла особенно актуально для приложений, предназначенных для анализа данных, но может пригодиться и в любой другой

ситуации, требующей анализа или изменения информации, хранящейся в файле. Например, программа может читать содержимое текстового файла и переписывать его, используя форматирование, рассчитанное на отображение информации в браузере.

Работа с информацией в текстовом файле начинается с чтения данных в память. Вы можете прочитать все содержимое файла или же читать данные по строкам.

Чтение всего файла

Для начала нам понадобится файл с несколькими строками текста. Пусть это будет файл, содержащий число π с точностью до 30 знаков, по 10 знаков на строку:

pi_digits.txt

```
3.1415926535
8979323846
2643383279
```

Чтобы опробовать эти примеры, либо введите данные в редакторе и сохраните файл `pi_digits.txt`, либо скачайте файл из дополнительных материалов на https://ehmatthes.github.io/pcc_3e. Сохраните файл в каталоге, в котором будут располагаться программы этой главы.

Следующая программа открывает данный файл, читает его и выводит содержимое на экран:

file_reader.py

```
from pathlib import Path
```

```
❶ path = Path('pi_digits.txt')
❷ contents = path.read_text()
print(contents)
```

Чтобы получить доступ к содержимому файла, нам нужно указать Python путь к нему. *Путь* (`path`) указывает на точное местоположение файла или папки в системе. В Python доступен модуль `pathlib`, упрощающий работу с файлами и каталогами, независимо от того, в какой операционной системе работаете вы или пользователи вашей программы. Модуль, предоставляющий ту или иную функциональность, часто называют *библиотекой* (`library`), отсюда и название `pathlib`.

Начнем с импорта класса `Path` из `pathlib`. Объект `Path`, указывающий на файл, позволяет вам выполнить многие действия. Например, прежде чем работать с файлом, вы можете проверить, существует ли он, прочитать его содержимое или записать в него новые данные. В нашем случае мы создаем объект `Path`, представляющий файл `pi_digits.txt`, который мы присваиваем переменной `path` ❶. Данный файл сохранен в том же каталоге, что и файл `.py`, который мы пишем, поэтому имя файла — все, что нужно `Path` для доступа к нему.

ПРИМЕЧАНИЕ

Программа VS Code ищет файлы в папке, которая была открыта последней. Если вы пользуетесь этим редактором, то начните с открытия папки, в которой хранятся программы из данной главы. Например, если вы храните файлы программ в папке `chapter_10`, то нажмите сочетание клавиш `Ctrl+O` (`⌘+O` в macOS) и откройте эту папку.

После того как в программе появится объект `Path`, представляющий файл `pi_digits.txt`, используется метод `read_text()`, который читает все содержимое файла ❷ и сохраняет это содержимое в одной длинной строке в переменной `contents`. При выводе значения `contents` на экране появляется все содержимое файла:

```
3.1415926535
8979323846
2643383279
```

Единственное различие между выводом и исходным файлом — лишняя пустая строка в конце вывода. Откуда она взялась? Метод `read_text()` возвращает ее при чтении, если достигнут конец файла. Если вы хотите удалить лишнюю пустую строку, то примените функцию `rstrip()` к строке, хранящейся в переменной `contents`:

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()
contents = contents.rstrip()
print(contents)
```

Напомним (из главы 2), что метод `rstrip()` удаляет все пробельные символы, начиная от правого края строки. Теперь вывод точно соответствует содержимому исходного файла:

```
3.1415926535
8979323846
2643383279
```

Мы можем удалить символ новой строки при чтении содержимого файла, применив метод `rstrip()` сразу после вызова `read_text()`:

```
contents = path.read_text().rstrip()
```

Код дает Python указание применить метод `read_text()` к обрабатываемому файлу. Затем интерпретатор применяет метод `rstrip()` к строке, возвращаемой методом `read_text()`. Итоговая строка присваивается переменной `contents`. Такой подход называется *цепочкой методов* (method chaining) и используется в программировании довольно часто.

Относительные и абсолютные пути к файлам

Если передать `Path` простое имя файла, такое как `pi_digits.txt`, то Python ищет файл в том каталоге, в котором находится файл, выполняемый в настоящий момент (то есть файл программы `.py`).

В некоторых случаях (в зависимости от того, как организованы ваши рабочие файлы) открываемый файл может и не находиться в одном каталоге с файлом программы. Например, файл программы может располагаться в папке `python_work`; в ней создается папка `text_files` для текстовых файлов, с которыми работает программа. И хотя она находится в папке `python_work`, простая передача объекту `Path` имени файла из `text_files` не подойдет, поскольку Python проводит поиск файла в `python_work` и на этом остановится; поиск не будет продолжен во вложенной папке `text_files`. Чтобы открыть файлы из каталога, отличного от того, в котором хранится файл программы, необходимо указать корректный путь — то есть дать Python указание искать файлы в конкретном месте файловой системы.

В программировании применяются два основных способа указания путей. С помощью *относительного пути* (relative file path) вы можете дать Python указание искать файлы в каталоге, который задается *относительно* каталога, содержащего текущий файл программы. Папка `text_files` расположена в папке `python_work`, поэтому для открытия файла из `text_files` нужно создать путь, который начинается с `text_files` и заканчивается именем файла. Вот как создать этот путь:

```
path = Path('text_files/имя_файла.txt')
```

Кроме того, можно точно определить местонахождение файла в вашей системе независимо от того, где хранится выполняемая программа. Такие пути называются *абсолютными* (absolute file path) и используются в том случае, если относительный путь не работает. Например, если папка `text_files` находится не в `python_work`, а в другой папке (скажем, в `other_files`), то передать объекту `Path` путь `'text_files/имя_файла.txt'` не получится, поскольку Python будет искать указанную папку только внутри `python_work`. Чтобы объяснить Python, где следует искать файл, необходимо записать полный путь.

Абсолютные пути обычно длиннее относительных, поскольку начинаются с корневого каталога системы:

```
path = Path('/home/eric/data_files/text_files/имя_файла.txt')
```

Используя абсолютные пути, вы сможете читать файлы из любого каталога вашей системы. Пока будет проще хранить файлы в одном каталоге с файлами программ или в папках, вложенных в каталог с файлами программ (таких как `text_files` из рассмотренного примера).

ПРИМЕЧАНИЕ

В операционной системе Windows для отображения путей к файлам применяется обратный слеш (\) вместо прямого (/), но вы должны использовать прямые слэши в своем коде, даже в Windows. Библиотека `pathlib` автоматически выберет правильное представление пути при работе в вашей системе или системе стороннего пользователя.

Построчное считывание

В процессе чтения файла часто бывает нужно обработать каждую строку. Возможно, вы ищете некую информацию в файле или собираетесь каким-то образом изменить текст. Например, при чтении файла с метеорологическими данными вы обрабатываете каждую строку, у которой в описании погоды встречается слово «солнечно». Или, допустим, в новостях ищете каждую строку с тегом заголовка и заменяете ее специальными элементами форматирования.

Вы можете использовать метод `splitlines()`, чтобы преобразовать длинную строку в группу, а затем добавить цикл `for` для обработки каждой строки по очереди:

file_reader.py

```
from pathlib import Path
```

```
path = Path('pi_digits.txt')
```

```
❶ contents = path.read_text()
```

```
❷ lines = contents.splitlines()
```

```
for line in lines:
```

```
    print(line)
```

Начнем со считывания всего содержимого файла, как мы делали это ранее ❶. Если вы планируете обрабатывать отдельные строки в файле, то вам не нужно удалять пробельные символы при чтении файла. Метод `splitlines()` возвращает список всех строк в файле, и мы присваиваем этот список переменной `lines` ❷. Затем перебираем эти строки и выводим каждую из них:

```
3.1415926535  
8979323846  
2643383279
```

Поскольку мы не изменили ни одной строки, то вывод полностью совпадает с исходным текстовым файлом.

Работа с содержимым файла

После того как файл будет прочитан в память, вы сможете обрабатывать данные так, как посчитаете нужным. Вкратце изучим цифры числа пи. Для начала попробуем создать одну строку со всеми цифрами из файла без промежуточных пробельных символов:

pi_string.py

```
from pathlib import Path
```

```
path = Path('pi_digits.txt')
contents = path.read_text()
```

```
lines = contents.splitlines()
```

```
pi_string = ''
```

```
❶ for line in lines:
    pi_string += line
```

```
print(pi_string)
print(len(pi_string))
```

Сначала интерпретатор считывает содержимое файла и сохраняет каждую строку цифр в списке — точно так же, как это делалось в предыдущем примере. Затем создается переменная `pi_string` для хранения цифр числа пи. Далее следует цикл, который добавляет к `pi_string` каждую серию цифр ❶. Затем программа выводит строку и ее длину:

```
3.1415926535 8979323846 2643383279
36
```

Переменная `pi_string` содержит пробельные символы, которые присутствовали в начале каждой строки цифр. Чтобы удалить их, достаточно использовать функцию `lstrip()` на каждой строке:

```
--пропуск--
for line in lines:
    pi_string += line.strip()

print(pi_string)
print(len(pi_string))
```

В итоге мы получаем строку, содержащую значение пи с точностью до 30 знаков. Длина строки равна 32 символам, поскольку в нее также добавляются начальная цифра 3 и десятичная точка:

```
3.141592653589793238462643383279
32
```

ПРИМЕЧАНИЕ

Считывая содержимое текстового файла, Python интерпретирует весь текст в файле как строку. Если вы считываете из текстового файла число и хотите работать с ним в числовом контексте, то преобразуйте его в целое или вещественное число с помощью функций `int()` или `float()` соответственно.

Большие файлы: миллион цифр

До настоящего момента мы ограничивались анализом текстового файла, который состоял всего из трех строк, но код этих примеров будет работать и с куда бóльшими файлами. Начиная с текстового файла, содержащего значение пи до 1 000 000 знаков (вместо 30), вы сможете создать одну строку, которая содержит все эти цифры. Изменять программу вообще не придется — достаточно передать ей другой файл. Кроме того, мы ограничимся выводом первых 50 цифр, чтобы не пришлось ждать, пока в терминале прокрутится миллион знаков:

pi_string.py

```
from pathlib import Path

path = Path('pi_million_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line.lstrip()

print(f"{pi_string[:52]}...")
print(len(pi_string))
```

Из выходных данных видно, что строка действительно содержит значение числа пи с точностью до 1 000 000 знаков:

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

Python не устанавливает никаких ограничений на длину данных, с которыми вы можете работать. Она ограничивается разве что объемом памяти вашей системы.

ПРИМЕЧАНИЕ

Чтобы запустить эту программу (и многие другие примеры, приведенные ниже), необходимо скачать дополнительные материалы с сайта https://ehmatthes.github.io/pcc_3e.

Проверка даты дня рождения

Меня всегда интересовало, не встречается ли мой день рождения среди цифр числа пи. Воспользуемся только что созданной программой и проверим, есть ли цифры дня рождения пользователя в первом миллионе цифр. Для этого можно записать день рождения в виде строки из цифр и посмотреть, имеется ли эта строка в `pi_string`:

pi_birthday.py

```
--пропуск--
for line in lines:
    pi_string += line.lstrip()
```

```
birthday = input("Enter your birthday, in the form mmddyy: ")
if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of pi.")
```

Сначала программа запрашивает день рождения пользователя, а затем проверяет вхождение этой строки в `pi_string`. Пробуем:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

Оказывается, мой день рождения встречается среди цифр числа пи! После того как данные будут прочитаны из файла, вы сможете делать с ними все, что сочтете нужным.

УПРАЖНЕНИЯ

10.1. Изучение Python. Откройте пустой файл в текстовом редакторе и напишите несколько строк текста о возможностях Python. Каждая строка должна начинаться с фразы «В Python вы можете...» Сохраните файл под именем `learning_python.txt` в каталоге, использованном для примеров этой главы. Напишите программу, которая читает файл и выводит текст два раза: читая весь файл и сохраняя строки в списке, проходя по каждой строке.

10.2. Изучение C. Метод `replace()` может использоваться для замены любого слова в строке другим словом. В следующем примере слово `'dog'` заменяется словом `'cat'`:

```
>>> message = "I really like dogs."
>>> message.replace('dog', 'cat')
'I really like cats.'
```

Прочитайте каждую строку из только что созданного файла `learning_python.txt` и замените слово Python названием другого языка — например, C. Выведите каждую измененную строку на экран.

10.3. Более простой код. В программе `file_reader.py` в этом разделе используется временная переменная, `lines`, демонстрирующая работу метода `splitlines()`. Вы можете опустить временную переменную и выполнить цикл непосредственно над списком, который возвращает метод `splitlines()`, следующим образом:

```
for line in contents.splitlines():
```

Удалите временную переменную из всех программ в этом разделе, чтобы сделать их код более лаконичным.

Запись в файл

Один из простейших способов сохранения данных — запись в файл. Текст, записанный в файл, останется доступным и после того, как терминал с выводом вашей программы будет закрыт. Вы сможете проанализировать результаты после завершения программы или передать свои файлы другим. Вы также сможете написать программы, которые снова читают сохраненный текст в память и работают с ним.

Запись одной строки

Определив путь, вы можете записать текст в файл с помощью метода `write_text()`. Для наглядности напишем простое сообщение и сохраним его в файл вместо вывода на экран:

write_message.py

```
from pathlib import Path

path = Path('programming.txt')
path.write_text("I love programming.")
```

Метод `write_text()` принимает единственный аргумент: строку, которую вы хотите записать в файл. Эта программа не имеет терминального вывода, но если вы откроете файл `programming.txt`, то увидите следующую строку:

programming.txt

```
I love programming.
```

Этот файл ничем не отличается от любого другого текстового файла на вашем компьютере. Его можно открыть, записать в него новый текст, скопировать/вставить текст и т. д.

ПРИМЕЧАНИЕ

Python может записывать в текстовые файлы только строковые данные. Если вы захотите сохранить в текстовом файле числовую информацию, то данные придется предварительно преобразовать в строки с помощью функции `str()`.

Запись нескольких строк

Метод `write_text()` «за кадром» выполняет несколько действий. Если файла, на который указывает путь, не существует, то метод создает его. Кроме того, после записи строки в файл метод проверяет, закрыт ли файл должным образом. Незакрытые файлы могут привести к потере или повреждению данных.

Чтобы записать в файл несколько строк, необходимо создать строку, содержащую все содержимое файла, а затем вызвать функцию `write_text()` и передать ей эту строку. Запишем несколько строк в файл `programming.txt`:

```
from pathlib import Path

contents = "I love programming.\n"
contents += "I love creating new games.\n"
contents += "I also love working with data.\n"

path = Path('programming.txt')
path.write_text(contents)
```

Мы определяем переменную `contents`, в которой будет храниться содержимое файла. В следующей строке используется оператор `+=`, позволяющий добавлять что-либо к этой строке. Вы можете применить его столько раз, сколько нужно, формируя строки любой длины. В данном случае мы добавляем символы новой строки в конце каждой строки, чтобы каждая фраза выводилась на отдельной строке.

Если вы выполните этот код, а затем откроете файл `programming.txt`, то увидите в нем следующие строки:

programming.txt

```
I love programming.
I love creating new games.
I also love working with data.
```

Вы можете форматировать вывод с помощью символов пробелов, табуляции и пустых строк так же, как и в терминале. Длина строк неограниченна, и именно так формируются многие документы, генерируемые компьютером.

ПРИМЕЧАНИЕ

Будьте осторожны при вызове функции `write_text()` на объекте `Path`. Если файл уже существует, то функция `write_text()` перезапишет текущее содержимое файла. Позже в этой главе вы научитесь с помощью модуля `pathlib` проверять, существует ли файл.

УПРАЖНЕНИЯ

10.4. Гость. Напишите программу, которая запрашивает у пользователя его имя. Введенный ответ сохраняется в файле `guest.txt`.

10.5. Гостевая книга. Напишите цикл `while`, который в цикле запрашивает у пользователей имена. При вводе каждого имени выведите на экран приветствие и добавьте строку с сообщением в файл `guest_book.txt`. Проследите за тем, чтобы каждое сообщение размещалось в отдельной строке файла.

Исключения

Для управления ошибками, возникающими в ходе выполнения программы, в Python используются специальные объекты, называемые *исключениями* (exceptions). Если при возникновении ошибки Python не знает, что делать дальше, то создается объект исключения. Если в программу добавлен код обработки исключения, то выполнение программы продолжится, а если нет — программа останавливается и выводит *трассировку* с отчетом об исключении.

Исключения обрабатываются в блоках `try-except`. С помощью такого блока можно дать Python указание выполнить некие действия, но при этом сообщить, что делать при возникновении исключения. Используя блоки `try-except`, ваши программы будут работать даже в том случае, если что-то пошло не так. Вместо невразумительной трассировки выводится понятное сообщение об ошибке, которое вы определяете в программе.

Обработка исключения `ZeroDivisionError`

Рассмотрим простую ошибку, при которой Python инициирует исключение. Конечно, вы знаете, что деление на ноль невозможно, но мы все же прикажем Python выполнить эту операцию:

division_calculator.py

```
print(5/0)
```

Конечно, из этого ничего не выйдет, поэтому на экран выводятся данные трассировки:

```
Traceback (most recent call last):
  File "division.py", line 1, in <module>
    print(5/0)
    ~~~
```

❶ `ZeroDivisionError: division by zero`

Ошибка, упоминаемая в трассировке — `ZeroDivisionError`, — является объектом исключения ❶. Такие объекты создаются в том случае, если Python не может выполнить ваши указания. Обычно в таких случаях Python прерывает выполнение программы и сообщает тип обнаруженного исключения. Эта информация может использоваться в программе; по сути, вы даете Python указание, как следует поступить при возникновении исключения данного типа. В таком случае ваша программа будет подготовлена к его появлению.

Блоки `try-except`

Если вы предполагаете, что в программе может произойти ошибка, то напишите блок `try-except` для обработки возникающего исключения. С помощью такого блока вы даете Python указание выполнить некий код, а также сообщаете, что нужно делать, если при его выполнении произойдет исключение конкретного типа.

Вот как выглядит блок `try-except` для обработки исключений `ZeroDivisionError`:

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Команда `print(5/0)`, порождающая ошибку, находится в блоке `try`. Если код в этом блоке выполнен успешно, то Python пропускает блок `except`. Если код в блоке `try` порождает ошибку, то Python ищет блок `except` с соответствующей ошибкой и выпускает код в нем.

В этом примере код блока `try` порождает ошибку `ZeroDivisionError`, поэтому Python ищет блок `except`, содержащий описание того, как следует действовать в такой ситуации. При выполнении кода этого блока пользователь видит понятное сообщение об ошибке вместо данных трассировки:

```
You can't divide by zero!
```

Если бы за кодом `try-except` следовал другой код, то выполнение программы продолжилось бы, поскольку мы объяснили Python, как обрабатывать эту ошибку. В следующем примере обработка ошибки позволяет программе продолжить выполнение.

Использование исключений для предотвращения аварийного завершения программы

Правильная обработка ошибок особенно важна в том случае, если программа должна продолжить работу после возникновения ошибки. Такая ситуация часто встречается в программах, запрашивающих данные у пользователя. Если программа правильно среагировала на некорректный ввод, то может запросить новые данные после сбоя.

Создадим простой калькулятор, который выполняет только операцию деления:

division_calculator.py

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    ❶ first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    ❷ second_number = input("Second number: ")
    if second_number == 'q':
        break
    ❸ answer = int(first_number) / int(second_number)
    print(answer)
```

Программа запрашивает у пользователя первое число `first_number` ❶, а затем, если он не ввел `q` для завершения работы, запрашивает второе число `second_number` ❷. Далее одно число делится на другое для получения результата `answer` ❸. Программа

никак не пытается обрабатывать ошибки, так что попытка деления на ноль приводит к ее аварийному завершению:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
Traceback (most recent call last):
  File "division_calculator.py", line 11, in <module>
    answer = int(first_number) / int(second_number)
              ~~~~~^~~~~~
ZeroDivisionError: division by zero
```

Конечно, аварийное завершение — это плохо, но еще хуже, что пользователь увидит данные трассировки. Неопытного пользователя они собьют с толку, а при сознательной попытке взлома злоумышленник сможет получить из них куда больше информации, чем вам хотелось бы. Например, он узнает имя файла программы и увидит некорректно работающую часть кода. На основании этой информации опытный хакер иногда может определить, какие атаки следует применять против вашего кода.

Блок `else`

Для повышения устойчивости программы к ошибкам можно поместить строку, выдающую ошибки, в блок `try-except`. Ошибка происходит в строке, выполняющей деление; следовательно, именно эту строку следует поместить в блок `try-except`. Данный пример также содержит блок `else`. Любой код, зависящий от успешного выполнения блока `try`, размещается в блоке `else`:

```
--пропуск--
while True:
    --пропуск--
    if second_number == 'q':
        break
❶  try:
    answer = int(first_number) / int(second_number)
❷  except ZeroDivisionError:
    print("You can't divide by 0!")
❸  else:
    print(answer)
```

Программа пытается выполнить операцию деления в блоке `try` ❶, который содержит только код, способный породить ошибку. Любой код, зависящий от успешного выполнения блока `try`, добавляется в блок `else`. В данном случае если операция деления выполняется успешно, то блок `else` используется для вывода результата ❸.

Блок `except` сообщает Python, как следует поступать при возникновении ошибки `ZeroDivisionError` ❷. Если при выполнении команды из блока `try` происходит ошибка, связанная с делением на ноль, то программа выводит понятное сообщение,

которое объясняет пользователю, как избежать подобных ошибок. Выполнение программы продолжается, и пользователь не увидит трассировку:

```
Give me two numbers, and I'll divide them.  
Enter 'q' to quit.
```

```
First number: 5  
Second number: 0  
You can't divide by 0!
```

```
First number: 5  
Second number: 2  
2.5
```

```
First number: q
```

В блоках `try` следует размещать только тот код, при работе которого может возникнуть исключение. Иногда некий код должен выполняться только в том случае, если выполнение `try` прошло успешно; такой код размещается в блоке `else`. Блок `except` сообщает Python, что делать, если при выполнении кода `try` произошло исключение.

Заранее определяя вероятные источники ошибок, вы повышаете надежность своих программ, которые продолжают работать даже при вводе некорректных данных или при недоступности ресурсов. Ваш код оказывается защищенным от случайных ошибок пользователей и сознательных атак.

Обработка исключения `FileNotFoundError`

Одна из стандартных проблем при работе с файлами — отсутствие необходимых файлов. Тот файл, который вам нужен, может находиться в другом месте, в имени файла может быть допущена ошибка, или файл может вообще не существовать. Все эти ситуации обрабатываются в блоках `try-except`.

Попробуем прочитать данные из несуществующего файла. Следующая программа пытается прочитать содержимое файла с текстом «Алисы в Стране чудес», но я не сохранил файл `alice.txt` в одном каталоге с файлом `alice.py`:

alice.py

```
from pathlib import Path  
  
path = Path('alice.txt')  
contents = path.read_text(encoding='utf-8')
```

Обратите внимание, что здесь мы используем метод `read_text()` несколько иначе, чем в предыдущих случаях. Аргумент `encoding` необходим в тех случаях, когда кодировка вашей системы по умолчанию не совпадает с кодировкой читаемого файла. Обычно так происходит при чтении файла, который был создан не в вашей системе.

Прочитать данные из несуществующего файла нельзя, поэтому Python выдает исключение:

Traceback (most recent call last):

```
❶ File "alice.py", line 4, in <module>
❷   contents = path.read_text(encoding='utf-8')
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "../pathlib.py", line 1056, in read_text
    with self.open(mode='r', encoding=encoding, errors=errors) as f:
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "../pathlib.py", line 1042, in open
    return io.open(self, mode, buffering, encoding, errors, newline)
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
❸ FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

Это более длинная трассировка, чем встречавшиеся вам ранее, поэтому посмотрим, как разобраться в более сложных результатах. Зачастую лучше всего читать трассировки с конца. В последней строке показано, что было выброшено исключение `FileNotFoundError` ❸. Это важно, поскольку так мы можем определить, какой тип исключения использовать в блоке `except`, который мы напишем.

В начале трассировки ❶ мы видим, что ошибка произошла в строке 4 файла `alice.py`. Далее приведена строка кода, вызвавшая ошибку ❷. В остальной части трассировки показан код библиотек, участвовавших в открытии и чтении файлов. Обычно эта часть трассировки при анализе не нужна.

Чтобы решить возникшую проблему, начнем блок `try` с проблемной строки, указанной в трассировке. В нашем примере это строка, содержащая вызов `read_text()`:

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
❶ except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
```

В этом примере код блока `try` выдает исключение `FileNotFoundError`, поэтому Python ищет блок `except` для этой ошибки ❶. Затем выполняется код данного блока, в результате чего вместо трассировки выдается более привычное сообщение об ошибке:

Sorry, the file `alice.txt` does not exist.

Если файл не существует, то программе больше нечего делать, поэтому код обработки ошибок почти ничего в нее не добавляет. Доработаем этот пример и посмотрим, как обработка исключений помогает при работе с несколькими файлами.

Анализ текста

Программа может анализировать текстовые файлы, содержащие целые книги. Многие классические произведения, ставшие общественным достоянием, доступны в виде простых текстовых файлов. Тексты, использованные в этом подразделе,

взяты с сайта проекта «Гутенберг» (<http://gutenberg.org/>). На нем хранится подборка литературных произведений, не защищенных авторским правом; это превосходный ресурс для разработчиков, которые собираются использовать литературные тексты в своих программных проектах.

Прочитаем текст «Алисы в Стране чудес» и попробуем подсчитать количество слов в тексте. Мы воспользуемся методом `split()`, предназначенным для создания списка слов на основе пробельных символов:

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
else:
    # Подсчет приблизительного количества строк в файле.
    words = contents.split()
    ❶ num_words = len(words)
    ❷ print(f"The file {path} has about {num_words} words.")
```

Я переместил файл `alice.txt` в правильный каталог, чтобы код в блоке `try` был выполнен без ошибок. Программа загружает текст в переменную `contents`, которая теперь содержит весь текст «Алисы в Стране чудес» в виде одной длинной строки, и использует метод `split()` для получения списка всех слов в книге ❶. Запрашивая длину этого списка ❷ с помощью функции `len()`, мы получаем неплохое приближенное значение количества слов в исходной строке. Напоследок выводится сообщение с количеством слов, найденных в файле. Этот код помещен в блок `else`, поскольку он должен выводиться только в случае успешного выполнения блока `try`.

Выходные данные программы сообщают, сколько слов содержит файл `alice.txt`:

```
The file alice.txt has about 29594 words.
```

Количество слов немного завышено, поскольку в нем учитывается дополнительная информация, добавленная в текстовый файл издателем, но в целом оно довольно точно оценивает длину текста «Алисы в Стране чудес».

Работа с несколькими файлами

Добавим еще несколько файлов с книгами для анализа. Но для начала переместим основной код программы в функцию `count_words()`. Это упростит проведение анализа для нескольких книг:

word_count.py

```
from pathlib import Path

def count_words(path):
    ❶ """Подсчитывает приблизительное количество строк в файле."""
    try:
```

```

        contents = path.read_text(encoding='utf-8')
    except FileNotFoundError:
        print(f"Sorry, the file {path} does not exist.")
    else:
        words = contents.split()
        num_words = len(words)
        print(f"The file {path} has about {num_words} words.")

```

```

path = Path('alice.txt')
count_words(path)

```

Большая часть кода не изменилась. Мы просто добавили в код отступ и переместили в тело `count_words()`. При внесении изменений в программу желательно обновлять комментарии, поэтому мы преобразовали комментарий в строку документации и слегка переформулировали его ❶.

Теперь мы можем написать простой цикл для подсчета слов в любом тексте, который нужно проанализировать. Для этого имена анализируемых файлов сохраняются в списке, после чего для каждого файла в списке вызывается функция `count_words()`. Мы попробуем подсчитать слова в «Алисе в Стране чудес», «Сиддхартхе», «Моби Дике» и «Маленьких женщинах» — все эти книги находятся в свободном доступе. Я намеренно не стал копировать файл `siddhartha.txt` в каталог с программой `word_count.py`, чтобы выяснить, насколько хорошо наша программа справляется с отсутствием файла:

```

from pathlib import Path

def count_words(filename):
    --пропуск--

    filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
    for filename in filenames:
        ❶ path = Path(filename)
        count_words(path)

```

Имена файлов хранятся в виде простых строк. Перед вызовом метода `count_words()` каждая строка преобразуется в объект `Path` ❶. Отсутствие файла `siddhartha.txt` не влияет на дальнейшее выполнение программы:

```

The file alice.txt has about 29594 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215864 words.
The file little_women.txt has about 189142 words.

```

Использование блока `try-except` в данном примере предоставляет два важных преимущества: программа ограждает пользователя от получения данных трассировки и продолжает выполнение, анализируя тексты, которые ей удастся найти. Если бы в программе не перехватывалось исключение `FileNotFoundError`, инициированное из-за отсутствия `siddhartha.txt`, то пользователь увидел бы полную трассировку, а работа программы прервалась бы после попытки подсчитать слова в тексте «Сиддхартхи»; до анализа «Моби Дика» или «Маленьких женщин» дело не дошло бы.

Ошибки без уведомления пользователя

В предыдущем примере мы сообщили пользователю о том, что один из файлов оказался недоступен. Тем не менее вы не обязаны сообщать о каждом обнаруженном исключении. Иногда при возникновении исключения программа должна просто проигнорировать сбой и продолжать работу, словно ничего не произошло. Для этого блок `try` пишется так же, как обычно, но в блоке `except` вы даете Python явное указание не предпринимать никаких особых действий в случае ошибки. В языке Python существует оператор `pass`, который сообщает, что в блоке ничего не надо делать:

```
def count_words(path):
    """Подсчитывает приблизительное количество строк в файле."""
    try:
        --пропуск--
    except FileNotFoundError:
        pass
    else:
        --пропуск--
```

Единственное отличие этого листинга от предыдущего — оператор `pass` находится в блоке `except`. Теперь при возникновении ошибки `FileNotFoundError` выполняется код в блоке `except`, но при этом ничего не происходит. Программа не выдает данные трассировки и вообще никакие результаты, указывающие на возникновение ошибки. Пользователи получают данные о количестве слов во всех существующих файлах, однако ничего не знают о том, что какой-то файл не был найден:

```
The file alice.txt has about 29594 words.
The file moby_dick.txt has about 215864 words.
The file little_women.txt has about 189142 words.
```

Оператор `pass` также может служить временным заполнителем. Он напоминает, что в этот конкретный момент выполнения вашей программы вы решили ничего не предпринимать, хотя возможно, что решите сделать что-то позднее. Например, эта программа может записать все имена отсутствующих файлов в файл `missing_files.txt`. Пользователи не увидят его, но создатель программы сможет прочитать данный файл и разобраться с отсутствующими текстами.

О каких ошибках нужно сообщать

Как определить, в каком случае следует сообщить об ошибке пользователю, а когда можно просто проигнорировать ее незаметно для него? Если пользователь знает, с какими текстами должна работать программа, то, вероятно, предпочтет получить сообщение, объясняющее, почему некоторые тексты были пропущены при анализе. Пользователь ожидает увидеть какие-то результаты, но не знает, какие книги должны быть проанализированы. Возможно, ему и не нужно знать о недоступности каких-то файлов. Лишняя информация сделает вашу программу менее удобной для пользователя. Средства обработки ошибок Python позволяют достаточно точно управлять тем, какой объем информации следует предоставить пользователю.

Хорошо написанный, правильно протестированный код редко содержит внутренние ошибки (например, синтаксические или логические). Но в любой ситуации, в которой ваша программа зависит от внешних факторов (пользовательского ввода, существования файла, доступности сетевого подключения), существует риск возникновения исключения. По мере накопления практического опыта вы начнете видеть, в каких местах программы следует разместить блоки обработки исключений и какой объем информации о возникающих ошибках предоставлять пользователям.

УПРАЖНЕНИЯ

10.6. Сложение. При вводе числовых данных часто встречается типичная проблема: пользователь вводит текст вместо чисел. При попытке преобразовать данные в `int` происходит исключение `ValueError`. Напишите программу, которая запрашивает два числа, складывает их и выводит результат. Перехватите исключение `ValueError`, если какое-либо из входных значений не является числом, и выведите удобное сообщение об ошибке. Протестируйте свою программу: сначала введите два числа, а затем текст вместо одного из чисел.

10.7. Калькулятор. Поместите код из упражнения 10.5 в цикл `while`, чтобы пользователь мог продолжать вводить числа, даже если допустил ошибку и ввел текст вместо числа.

10.8. Кошки и собаки. Создайте два файла с именами `cats.txt` и `dogs.txt`. Сохраните по крайней мере три клички кошек в первом файле и три клички собак во втором. Напишите программу, которая пытается прочитать эти файлы и выводит их содержимое на экран. Поместите свой код в блок `try-except` в целях перехвата исключения `FileNotFoundError` и вывода понятного сообщения об отсутствии файла. Переместите один из файлов в другое место файловой системы; убедитесь в том, что код блока `except` выполняется как положено.

10.9. Ошибки без уведомления. Измените блок `except` из упражнения 10.7 так, чтобы при отсутствии файла программа продолжала работу, не уведомляя пользователя о проблеме.

10.10. Распространенные слова. Зайдите на сайт проекта «Гутенберг» (<http://gutenberg.org/>) и найдите несколько книг для анализа. Скачайте текстовые файлы этих произведений или скопируйте текст из браузера в текстовый файл на вашем компьютере.

Чтобы узнать, сколько раз слово или фраза встречается в строке, можно воспользоваться методом `count()`. Например, следующий код подсчитывает количество вхождений `'row'` в строке:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Обратите внимание: преобразование строки в нижний регистр с помощью функции `lower()` позволяет найти все вхождения искомого слова независимо от регистра.

Напишите программу, которая читает файлы из проекта «Гутенберг» и определяет количество вхождений слова 'the' в каждом тексте. Результат будет приближенным, поскольку программа будет учитывать такие слова, как 'then' и 'there'. Попробуйте повторить поиск для строки 'the ' (с пробелом в строке) и посмотрите, насколько уменьшится количество найденных результатов.

Сохранение данных

Многие ваши программы будут запрашивать у пользователя информацию. Например, он может вводить настройки для компьютерной игры или данные для визуального представления. Чем бы ни занималась ваша программа, информация, введенная пользователем, будет сохраняться в структурах данных (таких как списки или словари). Когда пользователь закрывает программу, введенную им информацию почти всегда следует сохранять на будущее. Простейший способ сохранения данных основан на использовании модуля `json`.

Модуль `json` позволяет записывать простые структуры данных Python в строки в формате JSON и загружать данные из файла при следующем запуске программы. Этот модуль также может использоваться для обмена данными между программами Python. Более того, формат данных JSON не привязан к Python, поэтому данные в этом формате можно передавать программам, написанным на многих других языках программирования. Это полезный и универсальный формат, который к тому же легко изучать.

ПРИМЕЧАНИЕ

Формат JSON (JavaScript Object Notation) был изначально разработан для JavaScript. Однако с того времени стал использоваться во многих языках, в том числе Python.

Функции `json.dumps()` и `json.loads()`

Напишем две программы: одну короткую, сохраняющую набор чисел, и вторую, которая будет считывать эти числа обратно в память. Первая программа использует функцию `json.dumps()`, а вторая — функцию `json.loads()`.

Функция `json.dumps()` получает два аргумента: сохраняемые данные и объект файла, используемый для сохранения. В следующем примере `json.dumps()` используется для сохранения списка чисел:

number_writer.py

```
from pathlib import Path
import json
```

```
numbers = [2, 3, 5, 7, 11, 13]
```

```
❶ path = Path('numbers.json')
❷ contents = json.dumps(numbers)
path.write_text(contents)
```

Программа импортирует модуль `json` и создает список чисел для работы. Далее мы указываем имя файла, в котором будет храниться список ❶. Обычно для таких файлов принято задавать расширение `.json`, указывающее, что данные в файле хранятся в формате JSON. Функция `json.dumps()` ❷ используется для генерации строки, содержащей JSON-представление обрабатываемых данных. Получив эту строку, мы записываем ее в файл с помощью уже известного нам метода `write_text()`.

Программа ничего не выводит, но давайте откроем файл `numbers.json` и посмотрим на его содержимое. Данные хранятся в формате, очень похожем на код Python:

```
[2, 3, 5, 7, 11, 13]
```

А теперь напишем следующую программу, которая использует `json.loads()` для чтения списка обратно в память:

number_reader.py

```
from pathlib import Path
import json
```

```
❶ path = Path('numbers.json')
❷ contents = path.read_text()
❸ numbers = json.loads(contents)

print(numbers)
```

Для чтения данных используется тот же файл, в который они были записаны ❶. Поскольку считывается обычный текстовый файл с определенным форматированием, мы можем прочитать его с помощью метода `read_text()` ❷. Затем мы передаем содержимое файла функции `json.loads()` ❸. Она принимает строку в формате JSON и возвращает Python-объект (в данном случае список), который мы присваиваем переменной `numbers`. Наконец, программа выводит прочитанный список. Как видите, это тот же список, который был создан в программе `number_writer.py`:

```
[2, 3, 5, 7, 11, 13]
```

Модуль `json` позволяет организовать простейший обмен данными между программами.

Сохранение и чтение пользовательских данных

Сохранение данных с помощью модуля `json` особенно полезно при работе с данными, сгенерированными пользователем, поскольку при отсутствии сохранения эта информация будет потеряна в случае остановки программы. В следующем

примере программа запрашивает у пользователя имя при первом запуске программы и «вспоминает» его при повторных запусках.

Начнем с сохранения имени пользователя:

remember_me.py

```
from pathlib import Path
import json
```

- ❶ `username = input("What is your name? ")`
- ❷ `path = Path('username.json')`
`contents = json.dumps(username)`
`path.write_text(contents)`
- ❸ `print(f"We'll remember you when you come back, {username}!")`

Программа запрашивает имя пользователя, чтобы сохранить его ❶. Далее собранные данные записываются в файл `username.json` ❷. Затем выводится сообщение о том, что имя пользователя было сохранено ❸:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

А теперь напишем другую программу, которая приветствует пользователя, имя которого уже было сохранено ранее:

greet_user.py

```
from pathlib import Path
import json
```

- ❶ `path = Path('username.json')`
`contents = path.read_text()`
 - ❷ `username = json.loads(contents)`
- ```
print(f>Welcome back, {username}!")
```

Мы считываем содержимое файла ❶, а затем с помощью функции `json.loads()` присваиваем извлеченные данные переменной `username` ❷. После того как данные будут успешно прочитаны, мы можем поприветствовать пользователя по имени:

```
Welcome back, Eric!
```

Теперь эти две программы необходимо объединить в файл. Когда пользователь запускает `remember_me.py`, программа должна извлечь имя пользователя из памяти, если это возможно. В противном случае программа запрашивает имя пользователя и сохраняет его в файле `username.json`, чтобы вывести в следующий раз. Здесь можно использовать конструкцию `try-except`, чтобы соответствующим образом

среагировать, если файла `username.json` не существует, но лучше воспользуемся удобным методом из модуля `pathlib`:

### *remember\_me.py*

```
from pathlib import Path
import json
```

```
path = Path('username.json')
```

```
❶ if path.exists():
 contents = path.read_text()
 username = json.loads(contents)
 print(f"Welcome back, {username}!")
❷ else:
 username = input("What is your name? ")
 contents = json.dumps(username)
 path.write_text(contents)
 print(f"We'll remember you when you come back, {username}!")
```

Для работы с объектами `Path` доступно множество полезных методов. Метод `exists()` возвращает `True`, если файл или папка существует, и `False`, если нет. Здесь мы используем метод `path.exists()`, чтобы узнать, сохранено ли имя пользователя

❶. Если файл `username.json` существует, то мы загружаем имя пользователя и выводим персональное приветствие.

Если файла `username.json` не существует ❷, то мы запрашиваем имя пользователя и сохраняем введенное им значение. Мы также выводим сообщение о том, что вспомним пользователя в следующий раз, когда он вернется.

Какой бы блок ни выполнялся, результатом является имя пользователя и соответствующее сообщение. При первом запуске программы результат выглядит так:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Или же так:

```
Welcome back, Eric!
```

Такой результат вы увидите, если программа запускается не первый раз. В этом подразделе мы передавали программе одну строку, но программа будет работать с любыми данными, которые можно преобразовать в строку в формате JSON.

## Рефакторинг

Часто возникает типичная ситуация: код работает, но вы понимаете, что его структуру можно усовершенствовать, разбив на функции, каждая из которых решает свою конкретную задачу. Этот процесс называется *рефакторингом* (или переработкой). Рефакторинг делает ваш код более чистым, понятным и простым для расширения.

В процессе рефакторинга программы `remember_me.py` мы можем переместить основную часть логики в одну или несколько функций. Главной задачей `remember_me.py`

является вывод приветствия для пользователя, поэтому весь существующий код будет перемещен в функцию `greet_user()`:

### *remember\_me.py*

```
from pathlib import Path
import json
```

```
def greet_user():
```

```
❶ """Приветствует пользователя по имени."""
 path = Path('username.json')
 if path.exists():
 contents = path.read_text()
 username = json.loads(contents)
 print(f"Welcome back, {username}!")
 else:
 username = input("What is your name? ")
 contents = json.dumps(username)
 path.write_text(contents)
 print(f"We'll remember you when you come back, {username}!")
```

```
greet_user()
```

Мы используем функцию, поэтому комментарии заменяются строкой документации, которая описывает работу кода в текущей версии ❶. Код становится немного чище, но функция `greet_user()` не только приветствует пользователя — она загружает хранимое имя пользователя, если оно существует, и запрашивает новое, если имя не было сохранено ранее.

Переработаем функцию `greet_user()`, чтобы она не решала столько разных задач. Начнем с перемещения кода загрузки хранимого имени пользователя в отдельную функцию:

```
from pathlib import Path
import json
```

```
def get_stored_username(path):
```

```
❶ """Получает хранимое имя пользователя, если оно существует."""
 if path.exists():
 contents = path.read_text()
 username = json.loads(contents)
 return username
 else:
❷ return None
```

```
def greet_user():
```

```
 """Приветствует пользователя по имени."""
 path = Path('username.json')
 username = get_stored_username(path)
❸ if username:
 print(f"Welcome back, {username}!")
 else:
 username = input("What is your name? ")
 contents = json.dumps(username)
```

```
path.write_text(contents)
print(f"We'll remember you when you come back, {username}!")
```

```
greet_user()
```

Новая функция `get_stored_username()` ❶ имеет четкое предназначение, изложенное в строке документации. Она считывает и возвращает сохраненное имя пользователя, если его удастся найти. Если пути, переданного функции `get_stored_username()`, не существует, то функция возвращает `None` ❷. И это правильно: функция должна возвращать либо ожидаемое значение, либо `None`. Это позволяет провести простую проверку возвращаемого значения функции. Программа выводит приветствие для пользователя, если попытка получения имени пользователя была успешной ❸; в противном случае программа запрашивает новое имя пользователя.

Из функции `greet_user()` стоит вынести еще один блок кода. Если имени пользователя не существует, то код запроса нового имени должен размещаться в функции, специализирующейся на решении этой задачи:

```
from pathlib import Path
import json

def get_stored_username(path):
 """Получает хранимое имя пользователя, если оно существует."""
 --пропуск--

def get_new_username(path):
 """Запрашивает новое имя пользователя."""
 username = input("What is your name? ")
 contents = json.dumps(username)
 path.write_text(contents)
 return username

def greet_user():
 """Приветствует пользователя по имени."""
 path = Path('username.json')
 username = get_stored_username(path)
 if username:
 print(f>Welcome back, {username}!")
 else:
 ❷ username = get_new_username(path)
 print(f>We'll remember you when you come back, {username}!")

greet_user()
```

Каждая функция в окончательной версии `remember_me.py` имеет конкретное предназначение. Мы вызываем `greet_user()`, и эта функция выводит нужное приветствие: либо для уже знакомого, либо для нового пользователя. Для этого интерпретатор вызывает функцию `get_stored_username()` ❶, которая отвечает только за чтение хранимого имени пользователя (если оно есть). Наконец, функция `greet_user()` при необходимости вызывает функцию `get_new_username()` ❷, которая отвечает

только за получение нового имени пользователя и его сохранение. Такое «разделение обязанностей» является важнейшим аспектом написания чистого кода, простого в сопровождении и расширении.

### УПРАЖНЕНИЯ

**10.11. Любимое число.** Напишите программу, которая запрашивает у пользователя его любимое число. Воспользуйтесь функцией `json.dumps()` для сохранения этого числа в файл. Напишите другую программу, которая читает это значение и выводит сообщение: «Я знаю ваше любимое число! Это \_\_\_\_».

**10.12. Сохраненное любимое число.** Объедините две программы из упражнения 10.11 в файл. Если число уже сохранено, то сообщите его пользователю, а если нет — запросите любимое число пользователя и сохраните в файл. Выполните программу дважды, чтобы убедиться в том, что она работает.

**10.13. Словарь пользователя.** В программе `remember_me.py` хранится только один вид данных — имя пользователя. Дополните этот пример, запросив еще два вида информации о пользователе, а затем сохраните все собранные данные в словарь. Запишите его в файл с помощью функции `json.dumps()` и прочитайте данные с помощью функции `json.loads()`. Выведите сводку, какие именно данные о пользователе сохранила ваша программа.

**10.14. Проверка пользователя.** Последняя версия `remember_me.py` предполагает, что пользователь либо уже ввел свое имя, либо программа выполняется впервые. Ее нужно изменить на тот случай, если текущий пользователь не является тем человеком, который использовал программу последним.

Прежде чем выводить приветствие в `greet_user()`, спросите пользователя, правильно ли определено его имя. Если ответ будет отрицательным, то вызовите `get_new_username()` для получения правильного имени пользователя.

## Резюме

В этой главе вы научились работать с файлами. Вы узнали, как прочитать сразу весь файл и как обрабатывать его построчно. Вы научились записывать в файл столько текста, сколько захотите, а также познакомились с исключениями, возникающими в программе, и средствами их обработки. Кроме того, вы узнали, как использовать структуры данных Python для сохранения введенной информации, чтобы пользователю не приходилось раз вводить данные заново при каждом запуске программы.

В главе 11 вы познакомитесь с эффективными способами тестирования вашего кода. Тестирование поможет убедиться в том, что написанный код работает правильно, а также выявит ошибки, внесенные в процессе расширения уже написанных программ.

# 11

## Тестирование кода



Помимо функций и классов, вы можете написать тесты для своего кода. Тестирование доказывает, что код работает как положено для любых разновидностей входных данных, которые он может получать. Тесты позволят вам быть уверенными в том, что код будет работать правильно и тогда, когда вашими программами начнут пользоваться другие люди. Тестирование при добавлении нового кода гарантирует, что внесенные изменения не повлияют на текущее поведение программы. Все программисты допускают ошибки, поэтому каждый программист должен часто тестировать свой код и выявлять ошибки до того, как с ними столкнутся другие пользователи.

В этой главе вы научитесь тестировать код, используя средства модуля Python `pytest`. Библиотека `pytest` — это набор инструментов, которые помогут вам с легкостью написать первые тесты, а также обслуживать их по мере усложнения ваших проектов. Python не содержит `pytest` по умолчанию, поэтому вы научитесь устанавливать внешние библиотеки. Это поможет вам создавать лаконичный и хорошо структурированный код ваших будущих проектов. Кроме того, внешние библиотеки значительно расширят круг проектов, над которыми вы сможете работать.

Вы научитесь создавать серию тестов и проверять, приводит ли каждый набор входных данных к желаемому результату. Вы увидите, как выглядят пройденный и проваленный тесты, и узнаете, как неудачный тест может помочь вам улучшить ваш код. Вы научитесь тестировать функции и классы и начнете понимать, сколько тестов нужно писать для проекта.

### Установка `pytest` с помощью `pip`

Хотя Python содержит множество функций в стандартной библиотеке, разработчики Python также сильно зависят от пакетов сторонних разработчиков. *Сторонний пакет* (third-party package) — это библиотека, разработанная за пределами ядра

языка Python. Некоторые популярные библиотеки сторонних разработчиков в конечном счете переходят в стандартную библиотеку и с этого момента добавляются в большинство установочных пакетов Python. Чаще всего это происходит с библиотеками, которые вряд ли сильно изменятся после того, как в них будут устранены первые ошибки. Такие библиотеки могут развиваться в том же темпе, что и весь язык.

Тем не менее многие пакеты не входят в стандартную библиотеку, поэтому их развитие происходит в сроки, не зависящие от самого языка. Такие пакеты, как правило, обновляются чаще, чем если бы они были привязаны к графику разработки Python. Это относится к `pytest` и большинству библиотек, которые мы будем использовать во второй половине этой книги. Не стоит слепо доверять каждому стороннему пакету, но вас также не должен отталкивать тот факт, что многие важные функции реализованы с помощью таких пакетов.

## Обновление `pip`

В состав Python входит инструмент `pip`, который используется для установки пакетов сторонних разработчиков. Поскольку `pip` помогает устанавливать пакеты с внешних ресурсов, его часто обновляют с целью решить потенциальные проблемы безопасности. Поэтому и мы начнем с его обновления.

Откройте новое терминальное окно и выполните следующую команду:

```
$ python -m pip install --upgrade pip
```

- ❶ Requirement already satisfied: pip in /.../python3.11/site-packages (22.0.4)  
--пропуск--
- ❷ Successfully installed pip-22.1.2

Первая часть этой команды, `python -m pip`, дает Python указание запустить модуль `pip`. Вторая часть, `install --upgrade`, дает `pip` указание обновить ранее установленный пакет. Последняя часть, `pip`, — это имя стороннего пакета, который должен быть обновлен. Согласно выводу, на моем компьютере текущая версия `pip`, 22.0.4 ❶, была заменена последней версией на момент написания книги, 22.1.2 ❷.

Вы можете использовать эту команду для обновления любых пакетов сторонних разработчиков, установленных в вашей системе:

```
$ python -m pip install --upgrade имя_пакета
```

## ПРИМЕЧАНИЕ

В операционной системе Linux инструмент `pip` может быть не включен в Python. Если при попытке обновить `pip` вы получаете ошибку, то обратитесь к инструкциям, приведенным в приложении А.

## Установка *pytest*

Теперь, обновив версию *pip*, мы можем установить *pytest*:

```
$ python -m pip install --user pytest
Collecting pytest
--пропуск--
Successfully installed attrs-21.4.0 iniconfig-1.1.1 ...pytest-7.x.x
```

Мы по-прежнему используем основную команду, *pip install*, однако на этот раз без флага *--upgrade*. Вместо этого мы используем флаг *--user*, давая Python указание установить этот пакет только для текущего пользователя. Согласно выводу, последняя версия *pytest* успешно установлена, как и ряд других пакетов, необходимых для работы *pytest*.

Вы можете использовать эту команду для установки любых пакетов сторонних разработчиков:

```
$ python -m pip install --user имя_пакета
```

### ПРИМЕЧАНИЕ

Если у вас возникли трудности с выполнением этой команды, то попробуйте выполнить ее без флага *--user*.

## Тестирование функции

Чтобы потренироваться в тестировании, нам понадобится код. Ниже приведена простая функция, которая получает имя и фамилию и возвращает отформатированное полное имя:

### *name\_function.py*

```
def get_formatted_name(first, last):
 """Генерирует отформатированное полное имя."""
 full_name = f"{first} {last}"
 return full_name.title()
```

Функция *get\_formatted\_name()* формирует полное имя из имени и фамилии, разделив их пробелом, преобразует первый символ каждого слова в верхний регистр и возвращает полученный результат. Чтобы убедиться в том, что эта функция работает правильно, мы напишем программу, которая ее использует. Программа *names.py* запрашивает у пользователя имя и фамилию и выдает отформатированное полное имя:

### *names.py*

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
```



```
first = input("\nPlease give me a first name: ")
if first == 'q':
 break
last = input("Please give me a last name: ")
if last == 'q':
 break

formatted_name = get_formatted_name(first, last)
print(f"\tNeatly formatted name: {formatted_name}.")
```

Программа импортирует функцию `get_formatted_name()` из модуля `name_function.py`. Пользователь вводит последовательность имен и фамилий и видит, что программа сгенерировала отформатированные полные имена:

Enter 'q' at any time to quit.

```
Please give me a first name: janis
Please give me a last name: joplin
 Neatly formatted name: Janis Joplin.
```

```
Please give me a first name: bob
Please give me a last name: dylan
 Neatly formatted name: Bob Dylan.
```

```
Please give me a first name: q
```

Как видно из листинга, имена сгенерированы правильно. Но, допустим, вы решили изменить функцию `get_formatted_name()`, чтобы она также работала со вторыми именами. При этом необходимо проследить за тем, чтобы функция не перестала правильно работать для имен, состоящих только из имени и фамилии. Чтобы протестировать код, можно запустить `names.py` и для проверки вводить имя из двух компонентов (скажем, Janis Joplin) при каждом изменении `get_formatted_name()`, но это довольно утомительно. К счастью, Python предоставляет эффективный механизм автоматизации тестирования вывода функций. При автоматизации тестирования `get_formatted_name()` вы будете уверены в том, что функция успешно работает для всех видов имен, для которых написаны тесты.

## Модульные тесты и тестовые сценарии

Существует множество подходов к тестированию программного обеспечения. Одним из самых простых видов тестирования является модульное тестирование. *Модульный тест* (unit test) проверяет правильность работы одного конкретного аспекта поведения функции. *Тестовый сценарий* (test case) представляет собой совокупность модульных тестов, которые совместно доказывают, что функция ведет себя так, как положено, во всем диапазоне ситуаций, которые она должна обрабатывать. Хороший тестовый сценарий учитывает все возможные виды ввода, которые может получать функция, и содержит тесты для представления всех таких ситуаций. Тестовый сценарий *с полным покрытием* (full coverage) содержит обширный спектр модульных тестов, охватывающих все возможные варианты использования

функции. Обеспечение полного покрытия может быть весьма непростой задачей в крупном проекте. Часто бывает достаточно написать модульные тесты для критичных аспектов поведения вашего кода, а затем стремиться к полному покрытию только в том случае, если проект перейдет в фазу масштабного использования.

## Прохождение теста

С помощью `pytest` создать модульный тест достаточно просто. Мы напишем одну тестовую функцию. Она будет вызывать тестируемую функцию, а мы — утверждать возвращаемое значение. Если наше утверждение верно, то тест пройдет; в противном случае — будет провален.

Вот тестовый сценарий, который проверяет, что функция `get_formatted_name()` работает правильно:

### *test\_name\_function.py*

```
from name_function import get_formatted_name
```

```
❶ def test_first_last_name():
 """Поддерживаются ли имена типа 'Janis Joplin'?"""
❷ formatted_name = get_formatted_name('janis', 'joplin')
❸ assert formatted_name == 'Janis Joplin'
```

Прежде чем запустить тест, рассмотрим эту функцию. Имя файла с тестом очень важно; оно должно начинаться со слова `test_`. При запуске написанных нами тестов с помощью `pytest` этот модуль найдет все файлы, имена которых начинаются с `test_`, и запустит все содержащиеся в них тесты.

В файле с тестом мы сначала импортируем функцию, которую хотим протестировать: `get_formatted_name()`. Затем определяем тестовую функцию: в данном случае это `test_first_last_name()` ❶. Это имя функции более длинное, чем использованное ранее, и на то есть веские причины. Так, тестовые функции должны начинаться со слова `test`, за которым следует символ подчеркивания. Все функции, имена которых начинаются с `test_`, будут *определены* модулем `pytest` и запущены в процессе тестирования.

Кроме того, имена тестовых функций должны быть более длинными и описательными, чем имена обычных функций. Вы вряд ли будете вызывать функцию сами; `pytest` сделает это самостоятельно. Имена тестовых функций должны быть достаточно описательными, чтобы по их именам в отчете о тестировании вы могли понять, что именно тестировалось.

Далее мы вызываем тестируемую функцию ❷, в данном случае `get_formatted_name()` с аргументами `'janis'` и `'joplin'`, точно так же, как и при запуске файла `names.py`. Результат выполнения этой функции мы присваиваем переменной `formatted_name`.

Наконец, мы создаем *утверждение* ❸. Так мы утверждаем, что соблюдается то или иное условие. Здесь мы утверждаем, что переменной `formatted_name` должно быть присвоено значение `'Janis Joplin'`.

## Выполнение тестирования

Запустив файл `test_name_function.py` вручную, вы не получите результат, поскольку мы так и не вызвали тестовую функцию. Вместо этого мы попросим `pytest` запустить тестовый файл.

Для этого откройте терминальное окно и перейдите в папку, содержащую файл с тестом. В редакторе VS Code вы можете открыть папку, в которой находится файл с тестом, и использовать терминал, встроенный в окно редактора. В терминальном окне введите команду `pytest`. Вот что вы должны увидеть:

```
$ pytest
===== test session starts =====
❶ platform darwin -- Python 3.x.x, pytest-7.x.x, pluggy-1.x.x
❷ rootdir: /.../python_work/chapter_11
❸ collected 1 item

❹ test_name_function.py . [100%]
===== 1 passed in 0.00s =====
```

Разберем, что мы видим в выводе. Прежде всего здесь отображена информация о системе, в которой выполняется тест ❶. Я тестирую программу в операционной системе macOS, так что в вашем случае вывод может быть несколько иным. Самое главное — указано, какие версии Python, `pytest` и других пакетов используются для выполнения теста.

Далее показан каталог, из которого запускается тест ❷: в моем случае это `python_work/chapter_11`. Далее указано, что `pytest` нашел один файл с тестом для запуска ❸ и имя файла с тестом, который выполняется ❹. Одна точка после имени файла информирует о том, что один тест пройден, а **100%** говорит о том, что все тесты были запущены. В крупных проектах могут быть сотни и даже тысячи тестов, поэтому точки и индикатор завершенности в процентах пригодятся для отслеживания общего хода выполнения тестов.

Последняя строка говорит о том, что один тест пройден и на его выполнение ушло менее 0,01 секунды.

Согласно результатам теста, функция `get_formatted_name()` успешно работает для полных имен, состоящих из имени и фамилии, если только функция не была изменена. В случае внесения изменений в `get_formatted_name()` тест можно запустить снова. И если тестовый сценарий опять пройдет, то мы будем знать, что функция продолжает успешно работать с полными именами типа «Дженис Джоплин».

## ПРИМЕЧАНИЕ

Если вы не знаете, как перейти в нужный каталог в терминале, то см. раздел «Запуск программ Python из терминала» в главе 1. А если выводится сообщение о том, что команда `pytest` не найдена, то вместо команды `pytest` используйте команду `python -m pytest`.

## Сбой теста

Что произойдет при провале теста? Попробуем изменить функцию `get_formatted_name()`, чтобы она работала со вторыми именами, — но сделаем это так, чтобы она перестала работать с полными данными из имени и фамилии типа «Дженис Джоплин».

Новая версия `get_formatted_name()` с дополнительным аргументом второго имени выглядит так:

### *name\_function.py*

```
def get_formatted_name(first, middle, last):
 """Генерирует отформатированное полное имя."""
 full_name = f"{first} {middle} {last}"
 return full_name.title()
```

Эта версия должна работать для полных имен из трех компонентов (со вторым именем), но тестирование показывает, что она перестала работать для полных имен из двух компонентов (имени и фамилии).

На этот раз `pytest` выдает следующий результат:

```
$ pytest
===== test session starts =====
--пропуск--
❶ test_name_function.py F [100%]
❷ ===== FAILURES =====
❸ _____ test_first_last_name _____
 def test_first_last_name():
 """Поддерживаются ли имена типа 'Janis Joplin'?"
❹ > formatted_name = get_formatted_name('janis', 'joplin')
❺ E TypeError: get_formatted_name() missing 1 required positional
 argument: 'last'

test_name_function.py:5: TypeError
===== short test summary info =====
FAILED test_name_function.py::test_first_last_name - TypeError:
get_formatted_name() missing 1 required positional argument: 'last'
===== 1 failed in 0.04s =====
```

На этот раз информации гораздо больше, поскольку при сбое теста разработчик должен знать, почему это произошло. Вывод начинается с одной буквы **F** ❶, которая сообщает, что один модульный тест в тестовом сценарии привел к ошибке. Далее приведен раздел **FAILURES** ❷, так как тесты, завершенные неудачно, обычно наиболее важны и на них следует обратить внимание при тестировании. Затем мы

видим, что ошибка произошла в тесте `test_first_last_name()` ❸. Угловая скобка ❹ указывает на строку кода, которая привела к сбою тестирования. Буква E в следующей строке ❺ отражает фактическую ошибку, которая привела к сбою: ошибку `TypeError` из-за отсутствия необходимого позиционного аргумента `last`. Наиболее важная информация повторяется в краткой выжимке в конце, поскольку при выполнении множества тестов программисту важно быстро понять, какие тесты провалились и почему.

## Реакция на сбойный тест

Что делать в случае провала теста? Если предположить, что проверяются правильные условия, то прохождение тестирования означает, что функция работает правильно, а провал — что в новый код вкралась ошибка. Поэтому не меняйте провальный тест. Если поменяете, то тестирование завершится успешно, а код, вызывающий вашу функцию по аналогии с тестом, перестанет работать. Вместо этого исправьте код, из-за которого тестирование не было завершено успешно. Проанализируйте изменения, внесенные в функцию, и разберитесь, как они привели к нарушению ожидаемого поведения.

В данном случае у функции `get_formatted_name()` было всего два обязательных параметра: имя и фамилия. Теперь она требует три обязательных параметра: имя, второе имя и фамилию. Добавление обязательного параметра для второго имени нарушило ожидаемое поведение `get_formatted_name()`. В таком случае лучше всего сделать параметр второго имени необязательным. После этого тесты для имен с двумя компонентами снова будут завершаться успешно, и программа сможет получать также вторые имена. Изменим функцию `get_formatted_name()`, чтобы параметр второго имени перестал быть обязательным, и снова выполним тестовый сценарий. Если он пройдет, то можно переходить к проверке правильности обработки вторых имен.

Чтобы сделать второе имя необязательным, нужно переместить параметр `middle` в конец списка параметров в определении функции и задать ему пустое значение по умолчанию. Будет добавлена еще и проверка `if`, которая правильно создает полное имя в зависимости от того, передается второе имя или нет:

### *name\_function.py*

```
def get_formatted_name(first, last, middle=''):
 """Создает отформатированное полное имя."""
 if middle:
 full_name = f"{first} {middle} {last}"
 else:
 full_name = f"{first} {last}"
 return full_name.title()
```

В новой версии функции `get_formatted_name()` параметр `middle` необязателен. Если второе имя передается функции, то полное будет содержать имя, второе имя и фамилию. В противном случае полное имя состоит только из имени и фамилии.

Теперь функция должна работать для обеих разновидностей имен. Чтобы узнать, работает ли функция для имен из двух компонентов типа Janis Joplin, снова запустите файл `test_name_function.py`:

```
$ pytest
===== test session starts =====
--пропуск--
test_name_function.py . [100%]
===== 1 passed in 0.00s =====
```

Теперь тестовый сценарий завершается успешно. Такой исход идеален; он означает, что функция снова работает для имен из двух компонентов, и нам не пришлось тестировать ее вручную. Исправить ошибку было несложно, поскольку провальный тест помог выявить новый код, нарушивший существующее поведение.

## Добавление новых тестов

Теперь мы знаем, что `get_formatted_name()` работает для простых имен, и можем написать второй тест для имен из трех компонентов. Для этого в файл `test_name_function.py` добавим еще одну тестовую функцию:

### `test_name_function.py`

```
from name_function import get_formatted_name
```

```
def test_first_last_name():
 --пропуск--
```

```
def test_first_last_middle_name():
```

```
 """Поддерживаются ли такие имена, как 'Wolfgang Amadeus Mozart'?"""
```

```
 formatted_name = get_formatted_name('wolfgang', 'mozart', 'amadeus')
```

```
 ❶ assert formatted_name == 'Wolfgang Amadeus Mozart'
```

Новой функции присваивается имя `test_first_last_middle_name()`. Имя должно начинаться со слова `test_`, чтобы эта функция выполнялась автоматически при запуске `pytest`. В остальном имя выбирается так, чтобы оно четко показывало, какое именно поведение `get_formatted_name()` мы тестируем. В результате при сбое теста вы сразу видите, к каким именам он относится.

Чтобы протестировать функцию, мы вызываем `get_formatted_name()` с тремя компонентами ❶, после чего утверждаем ❷, что возвращенное полное имя совпадает с ожидаемым. При повторном запуске `pytest` оба теста завершаются успешно:

```
$ pytest
===== test session starts =====
--пропуск--
collected 2 items

❶ test_name_function.py .. [100%]
===== 2 passed in 0.01s =====
```

Две точки ❶ означают, что два теста пройдены, что подтверждается в последней строке вывода. Отлично! Теперь мы знаем, что функция по-прежнему работает с именами из двух компонентов, как `Janis Joplin`, но можем быть уверены в том, что она сработает и для имен с тремя компонентами, таких как `Wolfgang Amadeus Mozart`.

## УПРАЖНЕНИЯ

**11.1. Город, страна.** Напишите функцию, которая получает два параметра: название страны и название города. Функция должна возвращать одну строку в формате «Город, Страна» — например, `Santiago, Chile`. Сохраните функцию в модуле `city_functions.py` в новой папке, чтобы `pytest` не выполнял тесты, которые мы уже написали.

Создайте файл `test_cities.py` для тестирования только что написанной функции. Напишите функцию `test_city_country()`, проверяющую, дает ли вызов функции с такими значениями, как `'santiago'` и `'chile'`, правильную строку. Запустите `test_cities.py` и убедитесь в том, что тест `test_city_country()` проходит успешно.

**11.2. Население.** Измените свою функцию так, чтобы у нее был третий обязательный параметр — население. В новой версии функция должна возвращать одну строку вида «Город, Страна — население xxx», например, `Santiago, Chile - population 5000000`. Снова запустите тестирование. Убедитесь в том, что тест `test_city_country()` на этот раз не проходит.

Измените функцию так, чтобы параметр населения стал необязательным. Снова запустите тестирование и убедитесь в том, что тест `test_city_country()` снова проходит успешно.

Напишите второй тест `test_city_country_population()`, который проверяет вызов функции со значениями `'santiago'`, `'chile'` и `'population=5000000'`. Снова запустите тестирование и убедитесь в том, что новый тест завершается успешно.

## Тестирование класса

В первой части этой главы мы писали тесты для отдельной функции. Сейчас мы займемся написанием тестов для класса. Вы будете использовать классы во многих своих программах, поэтому возможность доказать, что ваши классы работают правильно, будет, безусловно, полезной. Если тесты для класса, над которым вы работаете, проходят успешно, то вы можете быть уверены в том, что дальнейшая доработка класса не приведет к случайному нарушению его текущего поведения.

## Разные методы утверждений

До сих пор вы видели только один вид утверждений: что строка имеет определенное значение. При написании теста вы можете сделать любое утверждение, которое может быть выражено в виде условного оператора. Если условие истинно, как и предполагалось, то ваши ожидания относительно поведения части вашей программы подтверждаются; вы можете быть уверены в отсутствии ошибок. Если же условие, которое должно быть истинным, окажется ложным, то тест не будет пройден и вы узнаете, что есть проблема, которую нужно решить. В табл. 11.1 перечислено несколько часто используемых методов `assert`, которые можно добавить в начальные тесты.

**Таблица 11.1.** Распространенные операторы утверждений в тестах

| Утверждение                               | Использование                                    |
|-------------------------------------------|--------------------------------------------------|
| <code>assert a == b</code>                | Проверяет, что два значения равны                |
| <code>assert a != b</code>                | Проверяет, что два значения не равны             |
| <code>assert a</code>                     | Проверяет, что значение оценивается как истинное |
| <code>assert not a</code>                 | Проверяет, что значение оценивается как ложное   |
| <code>assert элемент in список</code>     | Проверяет, что элемент входит в список           |
| <code>assert элемент not in список</code> | Проверяет, что элемент не входит в список        |

Это лишь несколько примеров; все, что может быть выражено как условный оператор, может быть добавлено в тест.

## Класс для тестирования

Тестирование класса имеет много общего с тестированием функции — значительная часть работы направлена на тестирование поведения методов класса. Однако существуют и различия, поэтому мы напишем отдельный класс для тестирования. Возьмем класс для управления проведением анонимных опросов:

### *survey.py*

```
class AnonymousSurvey():
 """Собирает анонимные ответы на опросы."""

 ❶ def __init__(self, question):
 """Сохраняет вопрос и готовится к сохранению ответов."""
 self.question = question
 self.responses = []

 ❷ def show_question(self):
 """Выводит вопрос."""
 print(self.question)
```



```

❸ def store_response(self, new_response):
 """Сохраняет один ответ на опрос."""
 self.responses.append(new_response)

❹ def show_results(self):
 """Выводит все полученные ответы."""
 print("Survey results:")
 for response in self.responses:
 print(f"- {response}")

```

Класс начинается с вопроса, который вы предоставили ❶, и содержит пустой список для хранения ответов. Класс содержит методы для вывода вопроса ❷, добавления нового ответа в список ответов ❸ и вывода всех ответов, хранящихся в списке ❹. Чтобы создать экземпляр на основе этого класса, необходимо предоставить вопрос. После того как будет создан экземпляр, представляющий конкретный опрос, программа выводит вопрос с помощью метода `show_question()`, сохраняет ответ с помощью метода `store_response()` и выводит результаты, используя вызов `show_results()`.

Чтобы продемонстрировать работу класса `AnonymousSurvey`, напомним программу, которая использует его:

### *language\_survey.py*

```

from survey import AnonymousSurvey

Определение вопроса с созданием экземпляра AnonymousSurvey.
question = "What language did you first learn to speak?"
language_survey = AnonymousSurvey(question)

Вывод вопроса и сохранение ответов.
language_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
 response = input("Language: ")
 if response == 'q':
 break
 language_survey.store_response(response)

Вывод результатов опроса.
print("\nThank you to everyone who participated in the survey!")
language_survey.show_results()

```

Программа определяет вопрос и на его основе создает объект `AnonymousSurvey`. Затем она вызывает метод `show_question()`, который позволяет вывести вопрос, после чего переходит к получению ответов. Каждый ответ сохраняется сразу же при получении. Когда ввод ответов был завершен (пользователь ввел `q`), метод `show_results()` выводит результаты опроса:

```

What language did you first learn to speak?
Enter 'q' at any time to quit.

```

```

Language: English
Language: Spanish

```

```
Language: English
Language: Mandarin
Language: q
```

Thank you to everyone who participated in the survey!

Survey results:

- English
- Spanish
- English
- Mandarin

Этот класс работает для простого анонимного опроса. Но допустим, что вы решили усовершенствовать класс `AnonymousSurvey` и модуль `survey`, в котором он находится. Например, каждому пользователю будет разрешено ввести несколько ответов. Или вы напишете метод, который будет выводить только уникальные ответы и сообщать, сколько раз был дан тот или иной ответ. Или напишете другой класс для проведения неанонимных опросов.

Реализация таких изменений грозит повлиять на текущее поведение класса `AnonymousSurvey`. Например, может оказаться, что поддержка ввода нескольких ответов случайно повлияет на процесс обработки одиночных ответов. Чтобы гарантировать, что доработка модуля не нарушит существующего поведения, для класса нужно написать тесты.

## Тестирование класса `AnonymousSurvey`

Напишем тест, проверяющий всего один аспект поведения `AnonymousSurvey`, — что один ответ на опрос сохраняется правильно. После того как метод будет сохранен, метод `assertIn()` проверяет, действительно ли он находится в списке ответов:

### `test_survey.py`

```
from survey import AnonymousSurvey
```

```
❶ def test_store_single_response():
 """Проверяет, что один ответ сохранен правильно."""
 question = "What language did you first learn to speak?"
 ❷ language_survey = AnonymousSurvey(question)
 language_survey.store_response('English')
 ❸ assert 'English' in language_survey.responses
```

Программа начинается с импортирования тестируемого класса `AnonymousSurvey`. Первая тестовая функция проверяет, сохраняется ли ответ на вопрос в список ответов. Этому методу присваивается хорошее описательное имя `test_store_single_response()` ❶. Если тест не проходит, то имя метода в сводке этого теста ясно показывает, что проблема связана с сохранением отдельного ответа на опрос.

Чтобы протестировать поведение класса, необходимо создать экземпляр класса. Мы создаем экземпляр `language_survey` ❷ для вопроса "What language did you first learn to speak?". Один ответ (English) сохраняется с помощью метода `store_response()`. Затем программа убеждается в том, что ответ был сохранен

правильно; для этого она проверяет, что значение `English` присутствует в списке `language_survey.responses` ❸.

По умолчанию команда `pytest` без аргументов запускает все тесты, найденные в текущем каталоге. Чтобы сосредоточиться на тестах в одном файле, передаем имя файла с тестами, который следует выполнить. Здесь мы выполним только один тест, созданный для `AnonymousSurvey`:

```
$ pytest test_survey.py
===== test session starts =====
--snip--
test_survey.py . [100%]
===== 1 passed in 0.01s =====
```

Неплохо, но опрос с одним ответом вряд ли можно назвать полезным. Убедимся в том, что три ответа сохраняются правильно. Для этого в `TestAnonymousSurvey` добавляется еще один метод:

```
from survey import AnonymousSurvey

def test_store_single_response():
 --пропуск--

def test_store_three_responses():
 """Проверяет, что три ответа были сохранены правильно."""
 question = "What language did you first learn to speak?"
 language_survey = AnonymousSurvey(question)
 ❶ responses = ['English', 'Spanish', 'Mandarin']
 for response in responses:
 language_survey.store_response(response)

 ❷ for response in responses:
 assert response in language_survey.responses
```

Новой функции присваивается имя `test_store_three_responses()`. Мы создаем объект опроса по аналогии с тем, как это делалось в `test_store_single_response()`. Затем определяется список, содержащий три разных ответа ❶, и для каждого из этих ответов вызывается метод `store_response()`. После того как ответы будут сохранены, следующий цикл проверяет, что каждый ответ теперь присутствует в `language_survey.responses` ❷.

Если снова запустить `test_survey.py`, то оба теста (для одного ответа и для трех ответов) проходят успешно:

```
$ pytest test_survey.py
===== test session starts =====
--пропуск--
test_survey.py .. [100%]
===== 2 passed in 0.01s =====
```

Все работает прекрасно. Тем не менее тесты выглядят немного однообразно, поэтому мы воспользуемся еще одной возможностью `pytest`, чтобы повысить их эффективность.

## Фикстуры

В программе `test_survey.py` в каждой тестовой функции создавался новый экземпляр `AnonymousSurvey`. В коротком коде, с которым мы работаем, это приемлемо, но в реальном проекте с десятками или сотнями тестов возникает проблема.

Выполнять тестирование помогают *фикстуры* (fixture). Часто это объекты, используемые несколькими тестами. Фикстуры в `pytest` создаются с помощью функций с декоратором `@pytest.fixture`. *Декоратор* (decorator) — это директива, размещаемая непосредственно перед определением функции; Python применяет эту директиву к функции перед ее запуском, чтобы изменить поведение функции. Не волнуйтесь, если не совсем поняли принцип работы; вы можете использовать декораторы из сторонних пакетов, пока не научитесь писать их самостоятельно.

Воспользуемся фикстурой для создания одного экземпляра опроса, который можно использовать в обеих тестовых функциях в файле `test_survey.py`:

```
import pytest
from survey import AnonymousSurvey

❶ @pytest.fixture
❷ def language_survey():
 """Опрос, который будет доступен для всех функций тестирования."""
 question = "What language did you first learn to speak?"
 language_survey = AnonymousSurvey(question)
 return language_survey

❸ def test_store_single_response(language_survey):
 """Проверяет, правильно ли хранится один ответ."""
❹ language_survey.store_response('English')
 assert 'English' in language_survey.responses

❺ def test_store_three_responses(language_survey):
 """Проверяет, правильно ли хранятся три отдельных ответа."""
 responses = ['English', 'Spanish', 'Mandarin']
 for response in responses:
❻ language_survey.store_response(response)

 for response in responses:
 assert response in language_survey.responses
```

Нам нужно импортировать пакет `pytest`, поскольку мы используем определенный в нем декоратор. Мы применяем декоратор `@pytest.fixture` ❶ к созданной функции `language_survey()` ❷. Эта функция создает объект `AnonymousSurvey` и возвращает новый опрос.

Обратите внимание, что определения обеих тестовых функций изменились ❸❹; теперь у каждой тестовой функции есть параметр `language_survey`. Если параметр в тестовой функции совпадает с именем функции, имеющей декоратор `@pytest.fixture`,

то эта фикстура будет запущена автоматически, а возвращаемое значение — передано тестовой функции. В этом примере функция `language_survey()` снабжает `test_store_single_response()` и `test_store_three_responses()` экземпляром `language_survey`.

В тестовых функциях нет нового кода, но обратите внимание, что из них были удалены две строки ❹❺: определяющая вопрос и создающая объект `AnonymousSurvey`.

Когда мы вновь запускаем тестовый файл, оба теста по-прежнему завершаются успешно. Эти тесты будут особенно полезны при попытке расширить `AnonymousSurvey`, чтобы можно было обрабатывать несколько ответов каждого человека. Изменив код таким образом, вы можете запустить эти тесты и убедиться, что не нарушили алгоритм хранения как одного ответа, так и группы отдельных ответов.

Приведенная структура почти наверняка покажется вам сложной; она содержит один из самых абстрактных кодов, которые вы видели до сих пор. Вам не нужно сразу же использовать фикстуры; лучше писать тесты с большим количеством повторяющегося кода, чем вообще не тестировать свои программы. Просто запомните, что, когда вам надоест многократно писать один и тот же код в своих тестах, вы можете воспользоваться прекрасно отработанным способом избавиться от этого нудного действия. Кроме того, в простых примерах типа нашего фикстуры не сокращают и не упрощают код. А вот в проектах со множеством тестов или сложными объектами, использующимися в нескольких тестах, фикстуры могут значительно оптимизировать тестовый код.

Итак, взявшись разрабатывать фикстуру, напишите функцию, которая генерирует ресурс, используемый несколькими тестовыми функциями. Добавьте к новой функции декоратор `@pytest.fixture` и указывайте имя этой функции в качестве параметра всех тестовых функций, использующих этот ресурс. С этого момента ваши тесты станут короче, их будет проще писать и поддерживать.

## УПРАЖНЕНИЯ

**11.3. Штат компании.** Напишите класс `Employee`, представляющий работника. Метод `__init__()` должен получать данные об имени, фамилии и ежегодном окладе; все эти значения должны сохраняться в атрибутах. Напишите метод `give_raise()`, который по умолчанию увеличивает ежегодный оклад на 5000 долларов, но при этом может получать другую сумму прибавки.

Напишите тестовый сценарий для `Employee`. Напишите два тестовых метода: `test_give_default_raise()` и `test_give_custom_raise()`. Используйте метод `setUp()`, чтобы вам не приходилось заново создавать экземпляр `Employee` в каждом тестовом методе. Запустите свой тестовый сценарий и убедитесь в том, что оба теста прошли успешно.

## Резюме

В этой главе вы научились писать тесты для функций и классов с помощью средств модуля `pytest`. Вы узнали, как писать тестовые функции для проверки конкретных аспектов поведения ваших функций и классов. Вы научились использовать фикстуры для эффективного создания ресурсов, которые могут применяться в нескольких тестовых функциях в файлах с тестами.

Тестирование — важная тема, на которую многие новички не обращают внимания. Пока вы делаете свои первые шаги в программировании, писать тесты для простых проектов не нужно. Но как только вы начинаете работать над проектами, требующими значительных затрат ресурсов на разработку, обязательно проводите тестирование критических аспектов поведения ваших функций и классов. Эффективные тесты позволят вам быть уверенными в том, что изменения в проекте не повредят тому, что уже работает, а это даст вам возможность улучшать код. Случайно нарушив существующую функциональность, вы немедленно узнаете об этом, что позволит вам быстро исправить проблему. Отреагировать на сбой теста всегда намного проще, чем на отчет об ошибке, присланный недовольным пользователем.

Если вы добавите предварительные тесты, то другие программисты будут чувствовать себя более комфортно, экспериментируя с вашим кодом, будут более уважительно относиться к вашим проектам и с большей готовностью присоединятся к участию в них. Если вы будете участвовать в проекте, над которым работают другие программисты, то вам придется продемонстрировать, что ваш код проходит существующие тесты; кроме того, от вас будут ждать, что вы напишете тесты для нового поведения, добавленного вами в проект.

Поэкспериментируйте с тестами и освоите процесс тестирования кода. Пишите тесты для критических аспектов поведения ваших функций и классов, но не стремитесь полностью покрывать тестами свои ранние проекты (если только у вас для этого нет особых причин).

# ЧАСТЬ II

## Проекты

Поздравляем! Вы уже знаете о Python достаточно, чтобы взяться за создание интерактивных и реальных проектов. Благодаря созданию собственных проектов вы закрепите новые навыки и упрочите ваше понимание концепций, описанных в части I.

В части II представлены три типа проектов; вы можете взяться за любой из них в том порядке, который вам больше нравится. Ниже приведено краткое описание каждого проекта, чтобы вам было проще решить, с чего начать.

## **Программирование игры на языке Python**

В проекте «Инопланетное вторжение» (главы 12, 13 и 14) мы воспользуемся пакетом Pygame для написания 2D-игры, в которой игрок должен сбивать корабли пришельцев, скорость и сложность падения которых нарастает. К концу этого проекта вы будете знать достаточно для того, чтобы создавать собственные 2D-игры с помощью Pygame.

## **Визуализация данных**

Проект по визуализации данных начинается с главы 15. В нем вы научитесь генерировать данные и создавать практичные, элегантные визуализации этих данных, используя пакеты Matplotlib и Plotly. В главе 16 вы научитесь работать с данными из сетевых источников и передавать их пакету визуализации в целях создания графиков погодных данных и карты глобальной сейсмической активности. Наконец, в главе 17 показано, как написать программу для автоматического скачивания и визуализации данных. Навыки визуализации пригодятся вам для изучения науки о данных — одной из самых востребованных областей программирования на данный момент.

## **Веб-приложения**

В проекте веб-приложения (главы 18, 19 и 20) мы с помощью пакета Django создадим простое веб-приложение для ведения веб-дневника на произвольные темы. Пользователь создает учетную запись с именем и паролем, вводит тему и делает заметки. Кроме того, вы научитесь развертывать свое приложение на удаленном сервере, чтобы доступ к приложению мог получить любой человек в мире.

После завершения проекта вы сможете заняться созданием собственных простых веб-приложений. Кроме того, вы будете готовы к изучению более серьезных ресурсов, посвященных созданию приложений с помощью Django.



# 12

## Атакующий корабль



Давайте создадим собственную игру — «Инопланетное вторжение»! Мы воспользуемся Pygame — подборкой интересных, мощных модулей Python для управления графикой, анимацией и даже звуком, упрощающей создание сложных игр. Pygame берет на себя такие задачи, как отрисовка изображений на экране, что позволяет вам сосредоточиться на высокоуровневой логике игровой динамики.

В этой главе мы настроим Pygame и создадим корабль, который движется влево и вправо и стреляет по приказу пользователя. В следующих двух главах вы создадите флот пришельцев, а затем займетесь внесением усовершенствований — например, ограничением количества попыток и добавлением таблицы рекордов.

Кроме того, в данной главе вы научитесь управлять большими проектами, состоящими из многих файлов. Мы часто будем проводить рефакторинг и изменять структуру содержимого файлов, чтобы проект был четко организован, а код оставался эффективным.

Программирование игр — идеальный способ совместить изучение языка с развлечением. Написание простой игры поможет вам понять, как пишутся профессиональные игры. В процессе работы над этой главой вводите и запускайте код, чтобы понять, как каждый его блок участвует в общем игровом процессе. Экспериментируйте с разными значениями и настройками, чтобы лучше понять, как организовать взаимодействие с пользователем в ваших собственных играх.

### ПРИМЕЧАНИЕ

Игра «Инопланетное вторжение» состоит из множества файлов; создайте в своей системе новую папку `alien_invasion`. Чтобы операторы `import` работали правильно, все файлы проекта должны находиться в этой папке.

Кроме того, если вы уверенно работаете с системами управления версиями — возможно, вам стоит использовать такую систему в этом проекте. Если ранее вы никогда не работали с подобными системами, обратитесь к краткому обзору в приложении Г.

## Планирование проекта

Создание крупного проекта должно начинаться не с написания кода, а с планирования. План поможет вам направить усилия в нужном направлении и повысит вероятность успешного завершения проекта.

Итак, опишем игровой процесс в целом. Это описание не затрагивает все аспекты игры, но дает достаточно четкое представление о том, с чего начинать работу.

Каждый игрок управляет кораблем, который находится в середине нижнего края экрана. Игрок перемещает корабль вправо и влево с помощью клавиш управления курсором; клавиша **Пробел** используется для стрельбы. В начале игры флот пришельцев находится в верхней части экрана и постепенно опускается, при этом смещаясь в сторону. Игрок уничтожает пришельцев, стреляя по ним. Если ему удастся сбить всех пришельцев, то появляется новый флот, который движется быстрее предыдущего. Если пришелец сталкивается с кораблем игрока или доходит до нижнего края экрана, то игрок теряет корабль. Если игрок теряет все три корабля, то игра заканчивается.

На первой фазе разработки мы создадим корабль, который может двигаться вправо и влево. Корабль должен стрелять из пушки, когда игрок нажимает клавишу **Пробел**. Когда это поведение будет реализовано, мы можем заняться пришельцами и доработкой игрового процесса.

## Установка Pygame

Прежде чем браться за программирование, установите пакет Pygame. Он устанавливается так же, как и `pytest` в главе 11: с помощью инструмента `pip`. Если вы не читали главу 11 и не умеете работать с `pip`, то см. раздел «Установка `pytest` с помощью `pip`» данной главы.

Чтобы установить Pygame, введите следующую команду в приглашении терминала:

```
$ python -m pip install --user pygame
```

Если для запуска программ или терминального сеанса вы используете команду, отличающуюся от `python`, например `python3`, то измените приведенную выше команду соответствующим образом.

## Создание проекта игры

Создание игры начнется с создания пустого окна Pygame, в котором позднее будут отображаться игровые элементы — прежде всего корабль и пришельцы. Кроме того, игра должна реагировать на действия пользователя, назначать цвет фона и загружать изображение корабля.

## Создание окна Pygame и обработка ввода

Начнем с создания пустого окна Pygame, для чего будет создан класс, представляющий окно. Создайте в редакторе кода новый файл и сохраните его как `alien_invasion.py`, после чего введите следующий код:

### *alien\_invasion.py*

```
import sys

import pygame

class AlienInvasion:
 """Класс для управления ресурсами и поведением игры."""

 def __init__(self):
 """Инициализирует игру и создает игровые ресурсы."""
 ❶ pygame.init()

 ❷ self.screen = pygame.display.set_mode((1200, 800))
 pygame.display.set_caption("Alien Invasion")

 def run_game(self):
 """Запускает основной цикл игры."""
 ❸ while True:
 # Отслеживание событий клавиатуры и мыши.
 ❹ for event in pygame.event.get():
 ❺ if event.type == pygame.QUIT:
 sys.exit()

 # Отображение последнего прорисованного экрана.
 ❻ pygame.display.flip()

if __name__ == '__main__':
 # Создание экземпляра и запуск игры.
 ai = AlienInvasion()
 ai.run_game()
```

Программа начинается с импортирования модулей `sys` (завершает игру по команде игрока) и `pygame` (содержит функциональность, необходимую для создания игры).

Игра «Инопланетное вторжение» начинается с класса `AlienInvasion`. В методе `__init__()` функция `pygame.init()` инициализирует настройки, необходимые Pygame для нормальной работы ❶. Вызов `pygame.display.set_mode()` создает окно ❷, в котором прорисовываются все графические элементы игры. Аргумент `(1200, 800)` представляет собой кортеж, определяющий размеры игрового окна размером 1200 пикселей в ширину и 800 пикселей в высоту. (Вы можете изменить эти значения в соответствии с размерами своего монитора.) Объект окна присваивается атрибуту `self.screen`, что позволяет работать с ним во всех методах класса.

Объект, присвоенный `self.screen`, называется *поверхностью* (surface). Поверхность в Pygame — это часть экрана, на которой отображается игровой элемент. Каждый элемент в игре (например, пришелец или корабль игрока) представляет собой собственную поверхность. Поверхность, возвращаемая `display.set_mode()`, представляет все игровое окно. При активизации игрового цикла анимации эта поверхность автоматически перерисовывается при каждом проходе цикла, чтобы она обновлялась вследствие всех изменений, обусловленных вводом от пользователя.

Процессом игры управляет метод `run_game()`. В нем находится непрерывно выполняемый цикл `while` ❸, который содержит цикл событий и код, управляющий обновлениями экрана. *Событием* (event) называется действие, выполняемое пользователем во время игры (например, нажатие клавиши или перемещение мыши). Чтобы наша программа реагировала на события, мы напишем *цикл событий* (event loop) для *прослушивания* (listen) событий и выполнения соответствующей операции в зависимости от типа произошедшего события. Этим циклом событий является цикл `for` в строке ❹.

Для получения доступа к событиям, обнаруженным Pygame, используется метод `pygame.event.get()`. Он возвращает список событий, произошедших с момента последнего вызова этой функции. При любом событии клавиатуры или мыши отрабатывается цикл `for`. В этом цикле записывается серия операторов `if` для обнаружения и обработки конкретных событий. Например, когда игрок щелкает на кнопке закрытия игрового окна, программа обнаруживает событие `pygame.QUIT` и вызывает метод `sys.exit()` для выхода из игры ❺.

Вызов `pygame.display.flip()` ❻ дает Pygame указание отобразить последний отрисованный экран. В данном случае при каждом выполнении цикла `while` будет отображаться пустой экран, на котором видно, как стирается старый экран, поэтому будет виден только новый экран. При перемещении игровых элементов вызов `pygame.display.flip()` будет постоянно обновлять экран, отображая игровые элементы в новых позициях и скрывая старые изображения; таким образом создается иллюзия плавного движения.

В последней строке файла создается экземпляр игры, после чего вызывается метод `run_game()`. Вызов `run_game()` заключается в блок `if`, чтобы он выполнялся только при прямом вызове функции. Запустив файл `alien_invasion.py`, вы увидите пустое окно Pygame.

## Управление частотой кадров

В идеале игры должны работать с одинаковой скоростью (частотой кадров), на любых компьютерах. Управление частотой кадров игры, запускаемой в разных системах, — задача, которую обычно сложно решить, но в Pygame доступен относительно простой способ достижения этой цели. Мы создадим объект отслеживания игрового времени, и он будет однократно вычислять время при каждой

итерации основного цикла. Если цикл выполняется быстрее, чем заданная нами скорость, то Pygame добавит необходимую паузу, чтобы игра работала с постоянной скоростью.

Мы определим объект отслеживания игрового времени в методе `__init__()`:

#### *alien\_invasion.py*

```
def __init__(self):
 """Инициализирует игру и создает игровые ресурсы."""
 pygame.init()
 self.clock = pygame.time.Clock()
 --пропуск--
```

После инициализации `pygame` создадим экземпляр класса `Clock` из модуля `pygame.time`. Затем настраиваем скорость игры в конце цикла `while` в функции `run_game()`:

```
def run_game(self):
 """Запускает основной цикл игры."""
 while True:
 --пропуск--
 pygame.display.flip()
 self.clock.tick(60)
```

Метод `tick()` принимает один аргумент: частоту кадров игры. Я указал значение 60, поэтому Pygame сделает все возможное, чтобы цикл повторялся ровно 60 раз в секунду.

### ПРИМЕЧАНИЕ

Объект отслеживания игрового времени Pygame позволяет эффективно управлять скоростью игры в большинстве систем. Если в вашей системе игра работает не так, как требовалось, то попробуйте разные значения частоты кадров. Если вы не можете подобрать подходящую частоту в своей системе, то попробуйте удалить код, управляющий этой частотой, и изменить настройки игры, откорректировав ее работу в вашей системе.

### Задание фонового цвета

Pygame по умолчанию создает черный экран, но это банально — выберем другой цвет фона. Это делается в методе `__init__()`:

#### *alien\_invasion.py*

```
def __init__(self):
 --пропуск--
 pygame.display.set_caption("Alien Invasion")

 # Задание цвета фона.
 self.bg_color = (230, 230, 230)
```

```
def run_game(self):
 --пропуск--
 for event in pygame.event.get():
 if event.type == pygame.QUIT:
 sys.exit()

 # При каждом проходе цикла перерисовывается экран.
 self.screen.fill(self.bg_color)

 # Отображение последнего прорисованного экрана.
 pygame.display.flip()
 self.clock.tick(60)
```

②

Цвета в Pygame задаются по модели RGB: тройками интенсивности красной, зеленой и синей составляющих цвета. Значение каждой составляющей лежит в диапазоне от 0 до 255. Цветовое значение (255, 0, 0) соответствует красному цвету, (0, 255, 0) — зеленому, а (0, 0, 255) — синему. Разные сочетания составляющих RGB позволяют создать до 16 миллионов цветов. В цветовом значении (230, 230, 230) красная, синяя и зеленая составляющие смешиваются в равных долях, давая светло-серый цвет фона. Этот цвет сохраняется в переменной `self.bg_color` ①.

Экран заполняется цветом фона. Для этого вызывается метод `fill()` ②, получающий всего один аргумент: цвет фона.

## Создание класса *Settings*

Каждый раз, когда в нашу игру добавляется новая функциональность, в нее обычно добавляются и новые настройки (параметры конфигурации). Вместо того чтобы задавать настройки в коде, мы напомним модуль `settings`; он содержит класс `Settings`, в котором хранятся все настройки. Такое решение позволит передавать один объект вместо множества отдельных настроек. Кроме того, оно упрощает вызовы функций и изменение внешнего вида игры по мере развития проекта. Чтобы внести изменения в игру, достаточно будет изменить некоторые значения в `settings.py`, а не искать разные настройки в файлах.

Создайте новый файл `settings.py` в папке `alien_invasion` и добавьте этот первоначальный класс `Settings`:

### *settings.py*

```
class Settings:
 """Класс для хранения всех настроек игры "Инопланетное вторжение"."""

 def __init__(self):
 """Инициализирует настройки игры."""
 # Параметры экрана
 self.screen_width = 1200
 self.screen_height = 800
 self.bg_color = (230, 230, 230)
```

Чтобы создать экземпляр класса `Settings` и использовать его для обращения к настройкам, внесите в файл `alien_invasion.py` следующие изменения:

### **`alien_invasion.py`**

```
--пропуск--
import pygame

from settings import Settings

class AlienInvasion:
 """Класс для управления ресурсами и поведением игры."""

 def __init__(self):
 """Инициализирует игру и создает игровые ресурсы."""
 pygame.init()
 self.clock = pygame.time.Clock()
 ❶ self.settings = Settings()

 ❷ self.screen = pygame.display.set_mode(
 (self.settings.screen_width, self.settings.screen_height))
 pygame.display.set_caption("Alien Invasion")

 def run_game(self):
 --пропуск--
 # При каждом проходе цикла перерисовывается экран.
 ❸ self.screen.fill(self.settings.bg_color)

 # Отображение последнего прорисованного экрана.
 pygame.display.flip()
 self.clock.tick(60)
--пропуск--
```

Класс `Settings` импортируется в основной файл программы, после чего она создает экземпляр `Settings` и сохраняет его в `self.settings` ❶ после вызова `pygame.init()`. При создании экрана ❷ используются атрибуты `screen_width` и `screen_height` объекта `self.settings`, после чего объект `self.settings` также используется для получения цвета фона при заполнении экрана ❸.

Запустив файл `alien_invasion.py`, вы не заметите никаких изменений, поскольку в этом разделе мы всего лишь переместили настройки, уже использованные в другом месте. Теперь можно переходить к добавлению новых элементов на экран.

## **Добавление изображения корабля**

А теперь добавим в игру космический корабль, которым управляет игрок. Чтобы вывести его на экран, мы загрузим изображение, после чего воспользуемся методом `Pgame.blit()` для вывода изображения.

Выбирая графику для своих игр, обязательно обращайтесь внимание на условия лицензирования. Самый безопасный и дешевый начальный вариант — использование бесплатной графики с таких сайтов, как <https://opengameart.org/>.

В игре можно задействовать практически любые графические форматы, но проще всего использовать файлы в формате `.bmp`, поскольку этот формат Pygame загружает по умолчанию. И хотя Pygame можно настроить для других типов файлов, некоторые типы зависят от установки на компьютере определенных графических библиотек. (Большинство изображений, которые вы найдете, имеют формат `.jpg`, `.png` или `.gif`, но их можно преобразовать в формат `.bmp` с помощью таких программ, как Photoshop, GIMP или Paint.)

Обратите особое внимание на цвет фона вашего изображения. Попробуйте найти файл с прозрачным фоном, который можно заменить любым цветом фона в графическом редакторе. Чтобы ваша игра хорошо смотрелась, цвет фона изображения должен соответствовать цвету фона игры. Либо же можно подобрать цвет фона игры под цвет фона изображения.

В игре «Инопланетное вторжение» используется файл `ship.bmp` (рис. 12.1), который можно скачать из дополнительных материалов книги на [https://ehmatthes.github.io/pcc\\_3e](https://ehmatthes.github.io/pcc_3e). Цвет фона файла соответствует настройкам, используемым в проекте. Создайте в главной папке проекта (`alien_invasion`) папку `images`. Сохраните файл `ship.bmp` в папке `images`.

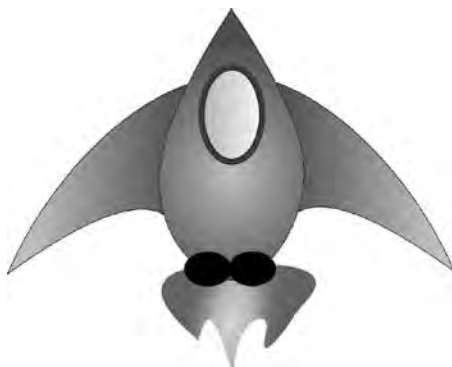


**Рис. 12.1.** Корабль для игры «Инопланетное вторжение»

## Создание класса *Ship*

После того как изображение корабля будет выбрано, его необходимо вывести на экран. Для работы с кораблем мы напишем модуль `ship`, содержащий класс `Ship`. Этот класс реализует большую часть поведения корабля.

**`ship.py`**

```
import pygame
```

```
class Ship:
```

```
 """Класс для управления кораблем."""
```



```
def __init__(self, ai_game):
 """Инициализирует корабль и задает его начальную позицию."""
 ❶ self.screen = ai_game.screen
 ❷ self.screen_rect = ai_game.screen.get_rect()

 # Загружает изображение корабля и получает прямоугольник.
 ❸ self.image = pygame.image.load('images/ship.bmp')
 self.rect = self.image.get_rect()

 # Каждый новый корабль появляется у нижнего края экрана.
 ❹ self.rect.midbottom = self.screen_rect.midbottom

 ❺ def blitme(self):
 """Рисует корабль в текущей позиции."""
 self.screen.blit(self.image, self.rect)
```

Один из факторов эффективности Pygame заключается в том, что программист может выполнять операции с игровыми элементами как с прямоугольниками даже в том случае, если они имеют другую форму. Операции с прямоугольниками эффективны, поскольку прямоугольник — простая геометрическая фигура. Обычно этот подход работает достаточно хорошо, и игроки не замечают, что программа не отслеживает точную геометрическую форму каждого игрового элемента. В этом классе корабль и экран будут рассматриваться как прямоугольные объекты.

Перед определением класса программа импортирует модуль `pygame`. Метод `__init__()` класса `Ship` получает два параметра: ссылку `self` и ссылку на текущий экземпляр класса `AlienInvasion`. Так класс `Ship` получает доступ ко всем игровым ресурсам, определенным в `AlienInvasion`. Экран присваивается атрибуту `Ship` ❶, чтобы к нему можно было легко обращаться во всех модулях класса. Программа обращается к атрибуту `rect` объекта экрана с помощью метода `get_rect()` и присваивает его `self.screen_rect` ❷. Это позволяет поместить корабль в нужной позиции экрана.

Чтобы загрузить изображение, мы вызываем метод `pygame.image.load()` ❸ и передаем ему местоположение изображения корабля. Функция возвращает поверхность, представляющую корабль, которая присваивается `self.image`. Когда изображение будет загружено, программа вызывает `get_rect()` для получения атрибута `rect` поверхности корабля, чтобы позднее использовать ее для позиционирования корабля.

При работе с объектом `rect` вам доступны координаты *x* и *y* верхней, нижней, левой и правой сторон, а также центра. Присваивая любые из этих значений, вы задаете текущую позицию прямоугольника. Местонахождение центра игрового элемента определяется атрибутами `center`, `centerx` или `centery` прямоугольника. Стороны определяются атрибутами `top`, `bottom`, `left` и `right`. Кроме того, есть атрибуты, которые являются комбинацией этих свойств — например, `midbottom`, `midtop`, `midleft` и `midright`. Для изменения горизонтального или вертикального расположения прямоугольника достаточно задать атрибуты *x* и *y*, содержащие координаты левого верхнего угла. Эти атрибуты избавляют вас от вычислений, которые раньше разработчикам игр приходилось выполнять вручную, притом достаточно часто.

### ПРИМЕЧАНИЕ

В Pygame начало координат (0, 0) находится в левом верхнем углу экрана, а оси направлены сверху вниз и слева направо. На экране размером 1200 на 800 начало координат располагается в левом верхнем углу, а правый нижний угол имеет координаты (1200, 800). Они относятся к игровому окну, а не физическому экрану.

Корабль будет расположен в середине нижней стороны экрана. Для этого значение `self.rect.midbottom` выравнивается по атрибуту `midbottom` прямоугольника экрана на ❷. Pygame использует эти атрибуты `rect` для позиционирования изображения, чтобы корабль был выровнен по центру, а его нижний край совпадал с нижним краем экрана.

Наконец, мы определяем метод `blitme()` ❸, который выводит изображение на экран в позиции, заданной `self.rect`.

### Вывод корабля на экран

Изменим программу `alien_invasion.py`, чтобы в ней создавался корабль и вызывался метод `blitme()` класса `Ship`:

#### *alien\_invasion.py*

```
--пропуск--
from settings import Settings
from ship import Ship

class AlienInvasion:
 """Класс для управления ресурсами и поведением игры."""

 def __init__(self):
 --пропуск--
 pygame.display.set_caption("Alien Invasion")

 ❶ self.ship = Ship(screen)

 def run_game(self):
 --пропуск--
 # При каждом проходе цикла перерисовывается экран.
 self.screen.fill(self.settings.bg_color)
 ❷ self.ship.blitme()

 # Отображение последнего прорисованного экрана.
 pygame.display.flip()
 self.clock.tick(60)
--пропуск--
```

После создания экрана программа импортирует класс `Ship` и создает его экземпляр ❶. При вызове `Ship` передается один аргумент — экземпляр `AlienInvasion`. Аргумент `self` относится к текущему экземпляру `AlienInvasion`. Этот параметр предоставляет `Ship` доступ к ресурсам игры — например, к объекту `screen`. Экземпляр `Ship` присваивается `self.ship`.

После заполнения фона корабль рисуется на экране с помощью вызова `ship.blitme()`, так что корабль выводится поверх фона ❷.

Если вы запустите программу `alien_invasion.py` сейчас, то увидите пустой игровой экран, в центре нижней стороны которого находится корабль (рис. 12.2).



**Рис. 12.2.** Корабль в середине нижней стороны экрана

## Рефакторинг: методы `_check_events()` и `_update_screen()`

В больших проектах перед добавлением нового кода часто проводится рефакторинг уже написанного кода. Рефакторинг упрощает структуру существующего кода и дальнейшее развитие проекта. В этом разделе метод `run_game()`, который становится слишком длинным, будет разбит на два вспомогательных метода. *Вспомогательный метод* (helper method) работает во внутренней реализации класса, но не предназначен для использования вне класса. В Python имена вспомогательных методов обозначаются начальным символом подчеркивания (`_`).

## Метод `_check_events()`

Начнем с перемещения кода управления событиями в отдельный метод `_check_events()`. Тем самым вы упростите метод `run_game()` и изолируете цикл управления событиями от остального кода. Изоляция цикла событий позволит организовать управление событиями отдельно от других аспектов игры (например, обновления экрана).

Ниже приведен класс `AlienInvasion` с новым методом `_check_events()`, который используется только в коде метода `run_game()`:

### *alien\_invasion.py*

```
def run_game(self):
 """Запускает основной цикл игры."""
 while True:
 ❶ self._check_events()

 # При каждом проходе цикла перерисовывается экран.
 --пропуск--

 ❷ def _check_events(self):
 """Обработывает нажатия клавиш и события мыши."""
 for event in pygame.event.get():
 if event.type == pygame.QUIT:
 sys.exit()
```

Мы определяем новый метод `_check_events()` ❷ и перемещаем строки, которые проверяют, не закрыл ли игрок окно щелчком кнопки мыши, в этот новый метод.

Для вызова метода внутри класса используется точечная запись с переменной `self` и именем метода ❶. Затем метод вызывается в цикле `while` метода `run_game()`.

## Метод `_update_screen()`

Чтобы еще больше упростить метод `run_game()`, выделим код обновления экрана в отдельный метод `_update_screen()`:

### *alien\_invasion.py*

```
def run_game(self):
 """Запускает основной цикл игры."""
 while True:
 self._check_events()
 self._update_screen()
 self.clock.tick(60)

 def _check_events(self):
 --пропуск--
```

```
def _update_screen(self):
 """Обновляет изображения на экране и отображает новый экран."""
 self.screen.fill(self.settings.bg_color)
 self.ship.blitme()

 pygame.display.flip()
```

Код прорисовки фона и переключения экрана перемещен в метод `_update_screen()`. Тело основного цикла в методе `run_game()` серьезно упростилось. С первого взгляда видно, что программа отслеживает новые события и обновляет экран при каждом проходе цикла.

Если вы уже написали несколько игр, то, вероятно, с самого начала начнете разбивать свой код на такие методы. Но если вы никогда не брались за подобный проект, то, возможно, не знаете, как структурировать данный код. Эта последовательность дает представление о реальном процессе разработки: сначала вы пишете самый простой код, а потом подвергаете его рефакторингу по мере роста сложности проекта.

Теперь, когда мы изменили структуру кода и упростили его расширение, можно переходить к динамическим аспектам игры!

## УПРАЖНЕНИЯ

**12.1. Синее небо.** Создайте окно Pygame с синим фоном.

**12.2. Игровой персонаж.** Найдите изображение игрового персонажа, который вам нравится, в формате `.bmp` (или преобразуйте существующее изображение). Создайте класс, который рисует персонажа в центре экрана, и приведите цвет фона изображения в соответствие с цветом фона экрана (или наоборот).

## Управление кораблем

Реализуем возможность перемещения корабля по горизонтали. Для этого мы напишем код, реагирующий на нажатие клавиш `←` или `→`. Начнем с движения вправо, а затем применим те же принципы к движению влево. Заодно вы научитесь управлять перемещением изображений на экране.

### Обработка нажатия клавиши

Каждый раз, когда пользователь нажимает клавишу, это нажатие регистрируется в Pygame как событие. Каждое событие идентифицируется методом `pygame.event.get()`, поэтому в методе `_check_events()` необходимо указать, какие события должны отслеживаться. Каждое нажатие клавиши регистрируется как событие `KEYDOWN`.

При обнаружении события `KEYDOWN` необходимо проверить, была ли нажата клавиша, инициирующая некое игровое событие. Например, при нажатии клавиши → значение `rect.x` корабля увеличивается для перемещения корабля вправо:

#### *alien\_invasion.py*

```
def _check_events(self):
 """Обрабатывает нажатия клавиш и события мыши."""
 for event in pygame.event.get():
 if event.type == pygame.QUIT:
 sys.exit()
 ❶ elif event.type == pygame.KEYDOWN:
 ❷ if event.key == pygame.K_RIGHT:
 # Переместить корабль вправо.
 ❸ self.ship.rect.x += 1
```

Внутри `_check_events()` в цикл событий добавляется блок `elif` для выполнения кода при обнаружении события `KEYDOWN` ❶. Чтобы проверить, является ли нажатая клавиша клавишей → (`pygame.K_RIGHT`), мы читаем атрибут `event.key` ❷. Если нажата клавиша →, то корабль перемещается вправо, для чего значение `self.ship.rect.x` увеличивается на 1 ❸.

Если вы запустите программу `alien_invasion.py` сейчас, то увидите, что корабль перемещается вправо на 1 пиксел при каждом нажатии клавиши →. Для начала неплохо, но это не лучший способ управления кораблем. Чтобы сделать управление более удобным, нужно реализовать возможность непрерывного перемещения.

### **Непрерывное перемещение**

Если игрок удерживает клавишу →, то корабль должен двигаться вправо до тех пор, пока клавиша не будет отпущена. Чтобы узнать, когда это произойдет, игра отслеживает событие `pygame.KEYUP`; таким образом, реализация непрерывного движения будет основана на отслеживании событий `KEYDOWN` и `KEYUP` в сочетании с флагом `moving_right`.

В неподвижном состоянии корабля флаг `moving_right` равен `False`. При нажатии клавиши → флагу присваивается значение `True`, а когда клавиша будет отпущена, флаг возвращается в состояние `False`.

Класс `Ship` управляет всеми атрибутами корабля, и мы добавим в него атрибут `moving_right` и метод `update()` для проверки состояния флага `moving_right`. Метод `update()` изменяет позицию корабля, если флаг содержит значение `True`. Этот метод будет вызываться каждый раз, когда вы хотите обновить позицию корабля.

Ниже приведены изменения в классе `Ship`:

#### *ship.py*

```
class Ship:
 """Класс для управления кораблем."""
```

```

def __init__(self, ai_game):
 --пропуск--
 # Каждый новый корабль появляется у нижнего края экрана.
 self.rect.midbottom = self.screen_rect.midbottom

 # Флаг перемещения: начинаем с неподвижного корабля.
 self.moving_right = False

def update(self):
 """Обновляет позицию корабля с учетом флага."""
 if self.moving_right:
 self.rect.x += 1

def blitme(self):
 --пропуск--

```

Мы добавляем атрибут `self.moving_right` в метод `__init__()` и инициализируем его значением `False` ❶. Затем вызываем метод `update()`, который перемещает корабль вправо, если флаг равен `True` ❷. Метод `update()` будет вызываться вне класса, поэтому не считается вспомогательным методом.

Теперь внесем изменения в метод `run_game()`, чтобы при нажатии клавиши → флагу `moving_right` присваивалось значение `True`, а при ее отпускании — `False`:

#### *alien\_invasion.py*

```

def check_events(self):
 """Обрабатывает нажатия клавиш и события мыши."""
 for event in pygame.event.get():
 --пропуск--
 elif event.type == pygame.KEYDOWN:
 if event.key == pygame.K_RIGHT:
 self.ship.moving_right = True
 elif event.type == pygame.KEYUP:
 if event.key == pygame.K_RIGHT:
 self.ship.moving_right = False

```

Теперь реакция игры при нажатии клавиши → изменяется; вместо непосредственного изменения позиции корабля программа просто присваивает флагу `moving_right` значение `True` ❶. Затем добавляется новый блок `elif`, реагирующий на события `KEYUP` ❷. Когда игрок отпускает клавишу → (`K_RIGHT`), флагу `moving_right` присваивается значение `False`.

Остается изменить цикл `while` в файле `alien_invasion.py`, чтобы при каждом проходе цикла вызывался метод `update()` корабля:

#### *alien\_invasion.py*

```

def run_game(self):
 # Запуск основного цикла игры.
 while True:
 self._check_events()
 self.ship.update()
 self._update_screen()
 self.clock.tick(60)

```

Позиция корабля будет обновляться после проверки событий клавиатуры, но перед обновлением экрана. Таким образом, позиция корабля обновляется в ответ на действия пользователя и будет использоваться при перерисовке корабля на экране.

Если запустить файл `alien_invasion.py` и удерживать клавишу `→`, то корабль непрерывно движется вправо, пока она не будет отпущена.

## Перемещение влево и вправо

Теперь, когда мы реализовали непрерывное движение вправо, добавить движение влево относительно несложно. Для этого нужно снова изменить класс `Ship` и метод `_check_events()`. Ниже приведены необходимые изменения в методах `__init__()` и `update()` в классе `Ship`:

### *ship.py*

```
def __init__(self, ai_game):
 --пропуск--
 # Флаги перемещения: начинаем с неподвижного корабля
 self.moving_right = False
 self.moving_left = False

def update(self):
 """Обновляет позицию корабля с учетом флагов."""
 if self.moving_right:
 self.rect.x += 1
 if self.moving_left:
 self.rect.x -= 1
```

В методе `__init__()` добавляется флаг `self.moving_left`. В `update()` используются два отдельных блока `if` вместо `elif`, чтобы при нажатии обеих клавиш со стрелками атрибут `rect.x` сначала увеличивался, а потом уменьшался. В результате корабль остается на месте. Если бы для движения влево использовался блок `elif`, то клавиша `→` всегда имела бы приоритет. Два блока `if` повышают точность перемещения при переключении направления, когда игрок может ненадолго удерживать нажатыми обе клавиши.

В метод `_check_events()` необходимо внести два изменения:

### *alien\_invasion.py*

```
def _check_events(self):
 """Обрабатывает нажатия клавиш и события мыши."""
 for event in pygame.event.get():
 --пропуск--
 elif event.type == pygame.KEYDOWN:
 if event.key == pygame.K_RIGHT:
 self.ship.moving_right = True
 elif event.key == pygame.K_LEFT:
 self.ship.moving_left = True
```



```
elif event.type == pygame.KEYUP:
 if event.key == pygame.K_RIGHT:
 self.ship.moving_right = False
 elif event.key == pygame.K_LEFT:
 self.ship.moving_left = False
```

Если событие KEYDOWN происходит для события K\_LEFT, то флагу `moving_left` присваивается `True`. Если событие KEYUP происходит для события K\_LEFT, то `moving_left` присваивается `False`. Здесь возможно использовать блоки `elif`, поскольку каждое событие связано только с одной клавишей. Если же игрок нажимает обе клавиши одновременно, то программа обнаруживает два разных события.

Если вы запустите программу `alien_invasion.py` сейчас, то увидите, что корабль может непрерывно двигаться влево и вправо. Если же нажать обе клавиши, то он останавливается.

Следующий шаг — доработка движения корабля. Внесем изменения в скорость и ограничим величину перемещения, чтобы корабль не выходил за края экрана.

## Управление скоростью корабля

В настоящий момент корабль смещается на один пиксел за каждый проход цикла `while`, но для повышения точности управления скоростью можно добавить в класс `Settings` атрибут `ship_speed`. Он определяет величину смещения корабля при каждом проходе цикла. Новый атрибут в файле `settings.py` выглядит так:

### *settings.py*

```
class Settings:
 """Класс для хранения всех настроек игры "Инопланетное вторжение"."""

 def __init__(self):
 --пропуск--

 # Настройки корабля
 self.ship_speed = 1.5
```

Переменной `ship_speed` присваивается значение 1.5. При перемещении корабля его позиция изменяется на 1,5 пиксела вместо 1.

Вещественные значения скорости позволят лучше управлять скоростью корабля при последующем повышении темпа игры. Однако атрибуты прямоугольников (такие как `x`) принимают только целочисленные значения, поэтому в класс `Ship` необходимо внести ряд изменений:

### *ship.py*

```
class Ship:
 """Класс для управления кораблем."""
```

```

def __init__(self, ai_game):
 """Инициализирует корабль и задает его начальную позицию."""
 self.screen = ai_game.screen
 ❶ self.settings = ai_game.settings
 --пропуск--

 # Каждый новый корабль появляется у нижнего края экрана.
 self.rect.midbottom = self.screen_rect.midbottom

 ❷ # Сохранение вещественной координаты центра корабля.
 self.x = float(self.rect.x)

 # Флаги перемещения: начинаем с неподвижного корабля
 self.moving_right = False
 self.moving_left = False

def update(self):
 """Обновляет позицию корабля с учетом флагов."""
 # Обновляется атрибут x, не rect.
 if self.moving_right:
 ❸ self.x += self.settings.ship_speed
 if self.moving_left:
 self.x -= self.settings.ship_speed

 # Обновление атрибута rect на основании self.x.
 ❹ self.rect.x = self.x

def blitme(self):
 --пропуск--

```

В классе `Ship` создается атрибут `settings`, чтобы он мог использоваться в методе `update()` ❶. Позиция корабля изменяется с нецелым приращением пикселей, поэтому должна храниться в переменной, способной содержать вещественные значения. Формально атрибутам `rect` можно присвоить вещественные значения, но `rect` сохранит только целую часть этого значения. Для точного отслеживания позиции корабля определяется новый атрибут `self.x` ❷. Функция `float()` используется для преобразования значения `self.rect.x` в вещественный формат, затем мы присваиваем это значение переменной `self.x`.

После изменения позиции корабля в `update()` значение `self.x` изменяется на величину, хранящуюся в `settings.ship_speed` ❸. После обновления `self.x` новое значение используется для обновления атрибута `self.rect.x`, управляющего позицией корабля ❹. В `self.rect.x` будет сохранена только целая часть `self.x`, но для отображения корабля этого достаточно.

Теперь можно изменить значение `ship_speed`; при любом значении, превышающем 1, корабль начинает двигаться быстрее. Эта возможность ускорит реакцию корабля на действия игрока, а также позволит нам изменить темп игры с течением времени.

## Ограничение перемещений корабля

Если удерживать какую-нибудь клавишу со стрелкой достаточно долго, то корабль выйдет за край экрана. Сделаем так, чтобы корабль останавливался при достижении края экрана. Задача решается путем изменения метода `update()` в классе `Ship`:

*ship.py*

```
def update(self):
 """Обновляет позицию корабля с учетом флагов."""
 # Обновляется атрибут x объекта ship, не rect.
 ❶ if self.moving_right and self.rect.right < self.screen_rect.right:
 self.x += self.settings.ship_speed
 ❷ if self.moving_left and self.rect.left > 0:
 self.x -= self.settings.ship_speed

 # Обновление атрибута rect на основании self.x.
 self.rect.x = self.x
```

Этот код проверяет позицию корабля перед изменением значения `self.x`. Выражение `self.rect.right` возвращает координату `x` правого края прямоугольника корабля. Если это значение меньше значения, возвращаемого `self.screen_rect.right`, значит, корабль еще не достиг правого края экрана ❶. То же относится и к левому краю: если координата `x` левой стороны прямоугольника больше 0, значит, корабль еще не достиг левого края экрана ❷. Проверка гарантирует, что корабль будет оставаться в пределах экрана перед изменением значения `self.x`.

Если вы запустите программу `alien_invasion.py` сейчас, то движение корабля будет останавливаться у края экрана. Согласитесь, эффектно: мы всего лишь добавили условную проверку в оператор `if`, но все выглядит так, словно у края экрана корабль наталкивается на невидимую стену или силовое поле!

## Рефакторинг метода `_check_events()`

В ходе разработки метод `_check_events()` будет становиться все длиннее, поэтому мы выделим из него еще два отдельных метода для обработки событий `KEYDOWN` и `KEYUP`:

*alien\_invasion.py*

```
def _check_events(self):
 """Реагирует на нажатие клавиш и события мыши."""
 for event in pygame.event.get():
 if event.type == pygame.QUIT:
 sys.exit()
 elif event.type == pygame.KEYDOWN:
 self._check_keydown_events(event)
 elif event.type == pygame.KEYUP:
 self._check_keyup_events(event)
```

```
def _check_keydown_events(self, event):
 """Реагирует на нажатие клавиш."""
 if event.key == pygame.K_RIGHT:
 self.ship.moving_right = True
 elif event.key == pygame.K_LEFT:
 self.ship.moving_left = True

def _check_keyup_events(self, event):
 """Реагирует на отпускание клавиш."""
 if event.key == pygame.K_RIGHT:
 self.ship.moving_right = False
 elif event.key == pygame.K_LEFT:
 self.ship.moving_left = False
```

В программе появились два вспомогательных метода: `_check_keydown_events()` и `_check_keyup_events()`. Каждый метод получает параметры `self` и `event`. Тела двух методов скопированы из метода `_check_events()`, а старый код заменен вызовами новых методов. Новая структура кода упрощает метод `_check_events()` и облегчает последующее программирование реакции на действия игрока.

## Нажатие клавиши Q для завершения

Итак, теперь программа реагирует на нажатия клавиш, и мы можем добавить еще один способ завершения игры. Было бы утомительно щелкать на кнопке X в верхней части игрового окна каждый раз, когда в игру добавляется новая функциональность, поэтому мы добавим специальную клавишу для завершения игры при нажатии клавиши Q:

### *alien\_invasion.py*

```
def _check_keydown_events(self, event):
 --пропуск--
 elif event.key == pygame.K_LEFT:
 self.ship.moving_left = True
 elif event.key == pygame.K_q:
 sys.exit()
```

В код метода `_check_keydown_events()` добавляется новый блок. Теперь в процессе тестирования можно закрыть игру путем нажатия клавиши Q вместо того, чтобы пользоваться кнопкой закрытия окна.

## Запуск игры в полноэкранном режиме

В Pygame поддерживается полноэкранный режим, который, возможно, понравится вам больше запуска в обычном окне. Некоторые игры лучше смотрятся в полноэкранном режиме, а в некоторых системах может улучшиться быстродействие.

Чтобы запустить игру в полноэкранном режиме, внесите в метод `__init__()` следующие изменения:

**alien\_invasion.py**

```
def __init__(self):
 """Инициализирует игру и создает игровые ресурсы."""
 pygame.init()
 self.settings = Settings()

 self.screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
 self.settings.screen_width = self.screen.get_rect().width
 self.settings.screen_height = self.screen.get_rect().height
 pygame.display.set_caption("Alien Invasion")
```

При создании экранной поверхности передается размер (0, 0) и параметр `pygame.FULLSCREEN` ❶. Благодаря этим значениям Pygame получает указание вычислить размер окна, заполняющего весь экран. Так как ширина и высота экрана неизвестны заранее, эти настройки обновляются после создания экрана ❷. Атрибуты `width` и `height` прямоугольника экрана используются для обновления объекта `settings`.

Если вам понравится, как игра выглядит или работает в полноэкранном режиме, то оставьте новые настройки. Если вы предпочитаете, чтобы игра работала в отдельном окне, — вернитесь к исходной реализации, где назначались конкретные размеры экрана.

**ПРИМЕЧАНИЕ**

Прежде чем запускать игру в полноэкранном режиме, убедитесь, что она закрывается при нажатии клавиши Q; в Pygame не существует стандартных средств завершения игры в полноэкранном режиме.

**Обобщим**

В следующем разделе мы реализуем стрельбу, для чего нам потребуется создать новый файл `bullet.py` и внести изменения в некоторые уже имеющиеся файлы. В настоящее время программа состоит из трех файлов с разными классами и методами. Чтобы вы четко представляли себе структуру проекта, кратко проанализируем каждый из этих файлов, прежде чем добавлять новую функциональность.

**Файл alien\_invasion.py**

Главный файл программы `alien_invasion.py` содержит класс `AlienInvasion`, в котором находится ряд важных атрибутов, используемых в процессе игры: настройки хранятся в `settings`, основная поверхность для вывода изображения — в `screen`, а экземпляр `ship` тоже создается в этом файле. Кроме того, в `alien_invasion.py` содержится главный цикл игры — `while` с вызовами методов `_check_events()`, `ship.update()` и `_update_screen()`. Вдобавок при каждой итерации цикла происходит отсчет времени.

Метод `_check_events()` обнаруживает важные события (например, нажатия и отпускания клавиш) и обрабатывает все эти типы событий с помощью методов `_check_keydown_events()` и `_check_keyup_events()`. На данный момент эти методы управляют движением корабля. Класс `AlienInvasion` также содержит метод `_update_screen()`, который перерисовывает экран при каждом проходе основного цикла.

Файл `alien_invasion.py` — единственный файл, который должен запускаться для игры «Инопланетное вторжение». Остальные файлы — `settings.py` и `ship.py` — содержат код, который импортируется в этот файл.

### Файл `settings.py`

Файл `settings.py` содержит класс `Settings`, в котором находится только метод `__init__()`, инициализирующий атрибуты, управляющие внешним видом и скоростью игры.

### Файл `ship.py`

Файл `ship.py` содержит класс `Ship`, в котором определены методы `__init__()`, `update()` для управления позицией корабля и `blitme()` для вывода изображения корабля на экран. Изображение корабля хранится в файле `ship.bmp`, который находится в папке `images`.

## УПРАЖНЕНИЯ

**12.3. Документация Pygame.** Разработка игры зашла уже достаточно далеко, и вам стоит просмотреть документацию Pygame. Главная страница Pygame находится по адресу <https://www.pygame.org/>, а главная страница документации — по адресу <https://www.pygame.org/docs/>. В данный момент вы можете ограничиться простым просмотром документации. Она не понадобится вам для завершения этого проекта, но пригодится, если вы захотите внести изменения в игру или займетесь созданием собственной игры.

**12.4. Ракета.** Создайте игру, у которой в исходном состоянии в центре экрана находится ракета. Игрок может перемещать ракету вверх, вниз, вправо и влево четырьмя клавишами со стрелками. Проследите за тем, чтобы ракета не выходила за края экрана.

**12.5. Клавиши.** Создайте файл Pygame, который создает пустой экран. В цикле событий выводите значение атрибута `event.key` при обнаружении события `pygame.KEYDOWN`. Запустите программу, нажимайте различные клавиши и наблюдайте за реакцией Pygame.

## Стрельба

А теперь добавим в игру функциональность стрельбы. Мы напишем код, благодаря которому при нажатии игроком клавиши **Пробел** выпускается снаряд (маленький прямоугольник). Снаряды летят вертикально вверх, пока не исчезнут у верхнего края экрана.

### Добавление настроек снарядов

Сначала добавим в файл `settings.py` новые настройки для значений, управляющих поведением класса `Bullet`. Эти настройки добавляются в конец метода `__init__()`:

#### `settings.py`

```
def __init__(self):
 --пропуск--
 # Параметры снаряда
 self.bullet_speed = 2.0
 self.bullet_width = 3
 self.bullet_height = 15
 self.bullet_color = (60, 60, 60)
```

Эти настройки создают темно-серые снаряды шириной 3 пиксела и высотой 15 пикселей. Они двигаются немного быстрее, чем корабль.

### Создание класса `Bullet`

Теперь создадим файл `bullet.py` для хранения класса `Bullet`. Первая часть файла выглядит так:

#### `bullet.py`

```
import pygame
from pygame.sprite import Sprite
```

```
class Bullet(Sprite):
```

```
 """Класс для управления снарядами, выпущенными кораблем."""
```

```
 def __init__(self, ai_game):
 """Создает объект снарядов в текущей позиции корабля."""
 super().__init__()
 self.screen = ai_game.screen
 self.settings = ai_game.settings
 self.color = self.settings.bullet_color
```

```
 # Создание снаряда в позиции (0,0) и назначение правильной позиции.
```

```
❶ self.rect = pygame.Rect(0, 0, self.settings.bullet_width,
 self.settings.bullet_height)
```

```
❷ self.rect.midtop = ai_game.ship.rect.midtop
```

```
 # Позиция снаряда хранится в вещественном формате.
```

```
❸ self.y = float(self.rect.y)
```

Класс `Bullet` наследует от класса `Sprite`, импортируемого из модуля `pygame.sprite`. Работая со *спрайтами* (sprite, динамические графические объекты), разработчик группирует связанные элементы в своей игре и выполняет операцию со всеми сгруппированными элементами одновременно. Чтобы создать экземпляр снаряда, методу `__init__()` необходим текущий экземпляр `AlienInvasion`, а вызов `super()` нужен для правильной реализации наследования от `Sprite`. Задаются и атрибуты для объектов экрана и настроек, а также цвета снаряда.

Затем создается атрибут `rect` снаряда ❶. Снаряд не создается на основе готового изображения, поэтому прямоугольник приходится рисовать с нуля с помощью класса `pygame.Rect()`. При создании экземпляра этого класса необходимо задать координаты левого верхнего угла прямоугольника, его ширину и высоту. Прямоугольник инициализируется в строке `(0, 0)`, но в следующих двух строках перемещается в нужное место, так как позиция снаряда зависит от позиции корабля. Ширина и высота снаряда определяются значениями, хранящимися в `self.settings`.

Атрибуту `midtop` снаряда присваивается атрибут `midtop` корабля ❷. Снаряд должен появляться у верхнего края корабля, поэтому верхний край снаряда совмещается с верхним краем прямоугольника корабля для имитации выстрела из корабля. Для координаты `y` снаряда мы используем вещественное значение, позволяющее точно управлять скоростью снаряда ❸.

А вот как выглядит вторая часть файла `bullet.py` — методы `update()` и `draw_bullet()`:

#### *bullet.py*

```
def update(self):
 """Перемещает снаряд вверх по экрану."""
 # Обновление точной позиции снаряда.
 ❶ self.y -= self.settings.bullet_speed
 # Обновление позиции прямоугольника.
 ❷ self.rect.y = self.y

def draw_bullet(self):
 """Выводит снаряд на экран."""
 ❸ pygame.draw.rect(self.screen, self.color, self.rect)
```

Метод `update()` управляет позицией снаряда. Когда происходит выстрел, снаряд двигается вверх по экрану, что соответствует уменьшению координаты `y`; следовательно, для обновления позиции снаряда следует вычесть величину, хранящуюся в `settings.bullet_speed`, из `self.y` ❶. Затем значение `self.y` используется для изменения значения `self.rect.y` ❷.

Атрибут `bullet_speed` позволяет увеличить скорость снарядов по ходу игры или при изменении ее поведения. Координата `x` снаряда после выстрела не изменяется, поэтому снаряд летит вертикально по прямой линии.



Для вывода снаряда на экран вызывается функция `draw_bullet()`. Функция `draw_rect()` заполняет часть экрана, определяемую прямоугольником снаряда, цветом из `self.color` ❸.

## Группировка снарядов

Класс `Bullet` и все необходимые настройки готовы; можно переходить к написанию кода, который будет выпускать снаряд каждый раз, когда игрок нажимает клавишу Пробел. Сначала мы создадим в `AlienInvasion` группу для хранения всех летящих снарядов, чтобы программа могла управлять их полетом. Эта группа будет представлена экземпляром класса `pygame.sprite.Group` — своего рода списком с расширенной функциональностью, которая может быть полезна при создании игр. Мы воспользуемся группой для прорисовки снарядов на экране при каждом проходе основного цикла и обновления текущей позиции каждого снаряда.

Сначала мы импортируем новый класс `Bullet`:

### *alien\_invasion.py*

```
--nponyск--
from ship import Ship
from bullet import Bullet
```

Группа будет создаваться в методе `__init__()`:

```
def __init__(self):
 --nponyск--
 self.ship = Ship(self)
 self.bullets = pygame.sprite.Group()
```

Позиция снаряда будет обновляться при каждом проходе цикла `while`:

```
def run_game(self):
 """Запускает основной цикл игры."""
 while True:
 self._check_events()
 self.ship.update()
 self.bullets.update()
 self._update_screen()
 self.clock.tick(60)
```

Вызов функции `update()` для группы приводит к ее автоматическому вызову для каждого спрайта в группе. Строка `self.bullets.update()` вызывает `bullet.update()` для каждого снаряда, включенного в группу `bullets`.

## Обработка выстрелов

В классе `AlienInvasion` необходимо внести изменения в метод `_check_keydown_events()`, чтобы при нажатии клавиши Пробел происходил выстрел. Изменять метод `_check_keyup_events()` не нужно, поскольку при отпускании клавиши ничего не происходит. Необходимо также изменить `_update_screen()` и вывести каждый снаряд на экран перед вызовом `flip()`.

При обработке выстрела придется выполнить довольно большую работу, для которой мы напишем новый метод `fire_bullet()`:

### *alien\_invasion.py*

```
def _check_keydown_events(self, event):
 --пропуск--
 elif event.key == pygame.K_q:
 sys.exit()
 ❶ elif event.key == pygame.K_SPACE:
 self._fire_bullet()

def _check_keyup_events(self, event):
 --пропуск--

def _fire_bullet(self):
 """Создает новый снаряд и добавляет его в группу bullets."""
 ❷ new_bullet = Bullet(self)
 ❸ self.bullets.add(new_bullet)

def _update_screen(self):
 """Обновляет изображения на экране и отображает новый экран."""
 self.screen.fill(self.settings.bg_color)
 ❹ for bullet in self.bullets.sprites():
 bullet.draw_bullet()
 self.ship.blitme()

 pygame.display.flip()
--пропуск--
```

При нажатии клавиши Пробел вызывается `_fire_bullet()` ❶. В коде `_fire_bullet()` мы создаем экземпляр `Bullet`, которому присваивается имя `new_bullet` ❷. Он добавляется в группу `bullets` путем вызова метода `add()` ❸. Метод `add()` похож на `append()`, но написан специально для групп Pygame.

Метод `bullets.sprites()` возвращает список всех спрайтов в группе `bullets`. Чтобы нарисовать все выпущенные снаряды на экране, программа перебирает спрайты в группе `bullets` и вызывает для каждого `draw_bullet()` ❹. Мы поместили этот цикл перед кодом, рисующим корабль, чтобы снаряды не появлялись поверх корабля.

Если вы запустите программу `alien_invasion.py` сейчас, то сможете двигать корабль влево и вправо и выпускать сколько угодно снарядов. Они перемещаются вверх по экрану и исчезают при достижении верхнего края (рис. 12.3). Размер, цвет и скорость можно изменить с помощью настроек в файле `settings.py`.

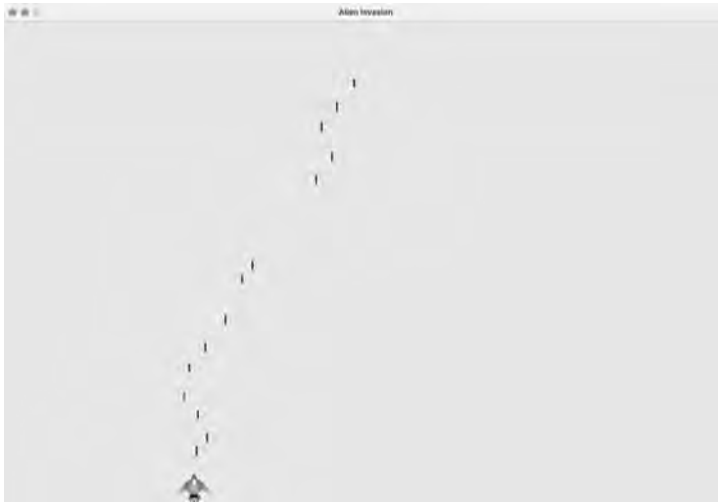


Рис. 12.3. Экран игры после серии выстрелов

### Удаление выпущенных снарядов

На данный момент снаряды исчезают при достижении верхнего края, но только потому, что Pygame не может нарисовать их выше края экрана. На самом деле снаряды по-прежнему существуют; их координата  $y$  продолжает уменьшаться. И это создает проблему, поскольку снаряды продолжают потреблять память и вычислительные мощности.

От старых снарядов необходимо избавиться, иначе игра замедлится из-за большого объема лишней работы. Для этого необходимо определить момент, когда атрибут `bottom` прямоугольника снаряда достигнет 0, — это означает, что снаряд вышел за верхний край экрана:

#### *alien\_invasion.py*

```
def run_game(self):
 # Запуск основного цикла игры.
 while True:
 self._check_events()
 self.ship.update()
 self.bullets.update()

 # Удаление снарядов, вышедших за край экрана.
 for bullet in self.bullets.copy():
 if bullet.rect.bottom <= 0:
 self.bullets.remove(bullet)
 print(len(self.bullets))

 self._update_screen()
 self.clock.tick(60)
```

❶  
❷  
❸  
❹

При использовании цикла `for` со списком (или группой в Pygame) Python ожидает, что длина списка будет оставаться прежней во время выполнения цикла. Таким образом, вы не можете удалять элементы из списка или группы в цикле `for`, поэтому перебирать нужно копию группы. Метод `copy()` используется для создания цикла `for` ❶, в котором можно изменять группу снарядов. Программа проверяет каждый снаряд и определяет, не покинул ли он пределы экрана ❷. Если да, то снаряд удаляется из `bullets` ❸. Затем добавляется вызов функции `print()`, чтобы увидеть, сколько снарядов сейчас существует в игре; по выведенному значению можно убедиться в том, что снаряды действительно удаляются при достижении верхнего края экрана ❹.

Если код работает правильно, то вы можете понаблюдать за выводом на терминале и убедиться в том, что количество снарядов уменьшается до нуля после того, как очередной залп уходит за верхний край экрана. После того как вы запустите игру и убедитесь в том, что снаряды правильно удаляются из группы, удалите вызов функции `print()`. Если команда останется в программе, она существенно замедлит игру, потому что вывод на терминал занимает больше времени, чем отображение графики в игровом окне.

## Ограничение количества снарядов

Многие игры-«стрелялки» ограничивают количество снарядов, одновременно находящихся на экране, чтобы у игроков появился стимул стрелять более метко. То же самое будет сделано и в игре «Инопланетное вторжение».

Сначала сохраним максимально допустимое количество снарядов в файле `settings.py`:

### `settings.py`

```
Параметры снаряда
--пропуск--
self.bullet_color = (60, 60, 60)
self.bullets_allowed = 3
```

В любой момент времени на экране может находиться не более трех снарядов. Эта настройка будет использоваться в классе `AlienInvasion` для проверки количества существующих снарядов перед созданием нового снаряда в методе `_fire_bullet()`:

### `alien_invasion.py`

```
def _fire_bullet(self):
 """Создает новый снаряд и добавляет его в группу bullets."""
 if len(self.bullets) < self.settings.bullets_allowed:
 new_bullet = Bullet(self)
 self.bullets.add(new_bullet)
```

При нажатии клавиши Пробел программа проверяет длину `bullets`. Если значение `len(self.bullets)` меньше трех, то создается новый снаряд. Но если на экране уже находятся три активных снаряда, то при нажатии клавиши Пробел ничего не происходит. Если вы запустите игру сейчас, то сможете выпускать снаряды только группами по три.

### Создание метода `_update_bullets()`

Мы хотим, чтобы класс `AlienInvasion` был как можно более простым, поэтому после написания и проверки кода управления снарядами его можно переместить в отдельный метод. Мы создадим новый метод `_update_bullets()` и добавим его непосредственно перед `_update_screen()`:

#### *alien\_invasion.py*

```
def _update_bullets(self):
 """Обновляет позиции снарядов и уничтожает старые снаряды."""
 # Обновление позиций снарядов.
 self.bullets.update()

 # Удаление снарядов, вышедших за край экрана.
 for bullet in self.bullets.copy():
 if bullet.rect.bottom <= 0:
 self.bullets.remove(bullet)
```

Код `_update_bullets()` вырезается и вставляется из `run_game()`; мы всего лишь немного уточнили комментарии.

Цикл `while` в `run_game()` снова выглядит просто:

#### *alien\_invasion.py*

```
while True:
 self._check_events()
 self.ship.update()
 self._update_bullets()
 self._update_screen()
 self.clock.tick(60)
```

В результате преобразования основной цикл содержит минимум кода, чтобы можно было легко прочитать имена функций и понять, что происходит в игре. Основной цикл проверяет ввод, полученный от игрока, а затем обновляет позицию корабля и всех выпущенных снарядов. Затем обновленные позиции игровых элементов используются для вывода нового экрана и отсчета времени в конце каждой итерации цикла.

Снова запустите программу `alien_invasion.py` и убедитесь в том, что стрельба происходит без ошибок.

## УПРАЖНЕНИЯ

**12.6. Боковая стрельба.** Напишите игру, в которой корабль размещается у левого края экрана, а игрок может перемещать его вверх и вниз. При нажатии клавиши Пробел корабль стреляет и снаряд движется вправо по экрану. Проследите за тем, чтобы снаряды удалялись при выходе за край экрана.

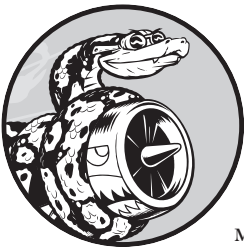
## Резюме

В этой главе вы научились планировать ход игры, а также усвоили базовую структуру игры, написанной с использованием Pygame. Вы узнали, как задать цвет фона и как сохранить настройки в отдельном классе, чтобы они были доступны для всех частей игры. Вы научились выводить изображения на экран и управлять перемещением игровых элементов. Кроме того, вы узнали, как создавать элементы,двигающиеся самостоятельно (например, летящие по экрану снаряды) и как удалять объекты, которые стали лишними. Вдобавок в этой главе вы познакомились с методикой регулярного рефакторинга кода в целях упрощения текущей разработки.

В главе 13 в игру «Инопланетное вторжение» будут добавлены пришельцы. К концу главы игрок сможет сбивать их корабли — конечно, если пришельцы не доберутся до него первыми!

# 13

## Осторожно, пришельцы!



В этой главе в игру «Инопланетное вторжение» будут добавлены пришельцы. Сначала мы добавим одного из них у верхнего края экрана, а потом сгенерируем целый флот. Пришельцы будут перемещаться в сторону и вниз; при этом те, в кого попадают снаряды, будут исчезать с экрана. Наконец, мы ограничим количество кораблей игрока, так что при гибели последнего корабля игра завершится.

В этой главе вы узнаете больше о Pygame и ведении крупного проекта. Вы также научитесь обнаруживать *коллизии* (столкновения) игровых объектов — например, снарядов и пришельцев. Выявление коллизий помогает определять взаимодействие элементов игры: например, ограничить перемещение персонажа областью между стенами лабиринта или организовать передачу мяча между двумя персонажами. Работа будет продолжаться на основе плана, к которому мы будем возвращаться время от времени, чтобы не отклоняться от цели во время написания кода.

Прежде чем браться за новый код для добавления флота пришельцев на экран, рассмотрим проект и обновим план.

### Анализ проекта

Приступая к новой фазе разработки крупного проекта, всегда полезно вернуться к исходному плану и уточнить, чего же вы хотите добиться в том коде, который собираетесь написать. В этой главе мы выполним следующие действия.

- Проанализируем код и определим, нужно ли провести рефакторинг перед реализацией новых возможностей.
- Заполним верхнюю часть экрана таким количеством пришельцев, сколько поместится по горизонтали. Затем будем создавать дополнительные ряды пришельцев, пока не соберем полный флот.

- По величине интервалов вокруг первого пришельца и общим размерам экрана вычислим, сколько пришельцев поместится на экране. Для создания пришельцев, заполняющих верхнюю часть экрана, будет написан цикл.
- Организуем перемещение флота пришельцев в сторону и вниз, пока весь он не будет уничтожен, пришелец не столкнется с кораблем игрока или не достигнет земли. Если весь флот уничтожен, то программа создает новый флот. Если пришелец сталкивается с кораблем или землей, то программа уничтожает корабль и создает новый флот.
- Ограничим количество кораблей, которые могут использоваться игроком, и завершим игру в конце последней попытки.

Этот план будет уточняться по мере реализации новых возможностей, но для начала и этого достаточно.

Кроме того, проводите ревью кода, когда начинаете работу над новой серией возможностей проекта. С каждой новой фазой проект обычно становится более сложным, поэтому лучше всего заняться расчисткой излишне громоздкого или неэффективного кода. Ранее мы уже проводили рефакторинг, так что сейчас особой расчистки не потребуется.

## Создание первого пришельца

Размещение одного пришельца на экране мало чем отличается от размещения корабля. Поведением каждого пришельца будет управлять класс `Alien`, который по своей структуре очень похож на класс `Ship`. Для простоты мы снова воспользуемся готовыми графическими изображениями. Вы можете найти собственное изображение пришельца или использовать изображение на рис. 13.1, доступное в дополнительных материалах к книге, размещенных по адресу [https://ehmatthes.github.io/pcc\\_3e](https://ehmatthes.github.io/pcc_3e). Это изображение имеет серый фон, совпадающий с цветом фона экрана. Не забудьте сохранить выбранный файл в папке `images`.



**Рис. 13.1.** Пришелец, который будет использоваться для создания флота



## Создание класса *Alien*

Теперь можно написать класс `Alien` и сохранить его в файле `alien.py`:

### *alien.py*

```
import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
 """Класс, представляющий одного пришельца."""

 def __init__(self, ai_game):
 """Инициализирует пришельца и задает его начальную позицию."""
 super().__init__()
 self.screen = ai_game.screen

 # Загрузка изображения пришельца и назначение атрибута rect.
 self.image = pygame.image.load('images/alien.bmp')
 self.rect = self.image.get_rect()

 # Каждый новый пришелец появляется в левом верхнем углу экрана.
 self.rect.x = self.rect.width
 self.rect.y = self.rect.height

 # Сохранение точной горизонтальной позиции пришельца.
 self.x = float(self.rect.x)
```

В основном этот класс похож на класс `Ship` (если не считать размещение пришельца). Изначально каждый пришелец размещается в левом верхнем углу экрана, при этом слева от него добавляется интервал, равный ширине пришельца, а над ним — интервал, равный высоте ❶. Нас в первую очередь интересует горизонтальная скорость пришельца, поэтому будем отслеживать точную горизонтальную позицию каждого из них ❷.

Классу `Alien` не нужен метод для вывода на экран; вместо этого мы воспользуемся методом группы `Pygame`, который автоматически рисует все элементы группы на экране.

## Создание экземпляра *Alien*

Начнем с создания экземпляра `Alien`, чтобы первый пришелец появился на экране. Так как эта операция входит в подготовительную часть, код для текущего экземпляра будет добавлен в конец метода `__init__()` в классе `AlienInvasion`. Позднее будет создан целый флот вторжения, что потребует определенной работы, поэтому мы определим новый вспомогательный метод `_create_fleet()`.

Порядок следования методов в классе может быть любым — важно лишь, чтобы в этом порядке существовала некая закономерность. Я размещу `_create_fleet()` непосредственно перед методом `_update_screen()`, но с таким же успехом его можно разместить в любой точке `AlienInvasion`. Начнем с импортирования класса `Alien`.

Обновленные операторы импортирования в файле `alien_invasion.py` выглядят так:

#### ***alien\_invasion.py***

```
--nponyck--
from bullet import Bullet
from alien import Alien
```

А это обновленный метод `__init__()`:

#### ***alien\_invasion.py***

```
def __init__(self):
 --nponyck--
 self.ship = Ship(self)
 self.bullets = pygame.sprite.Group()
 self.aliens = pygame.sprite.Group()

 self._create_fleet()
```

Создадим группу для хранения флота вторжения и вызовем метод `_create_fleet()`, который мы напишем чуть позже.

Новый метод `_create_fleet()` выглядит так:

#### ***alien\_invasion.py***

```
def _create_fleet(self):
 """Создает флот пришельцев."""
 # Создание пришельца.
 alien = Alien(self)
 self.aliens.add(alien)
```

В этом методе создается один экземпляр `Alien`, который затем добавляется в группу для хранения флота. По умолчанию объект размещается в левом верхнем углу экрана — эта позиция прекрасно подходит для первого пришельца.

Чтобы пришелец появился на экране, программа вызывает метод `draw()` группы в `_update_screen()`:

#### ***alien\_invasion.py***

```
def _update_screen(self):
 --nponyck--
 self.ship.blitme()
 self.aliens.draw(self.screen)

 pygame.display.flip()
```

При вызове метода `draw()` для группы Pygame выводит каждый элемент группы в позиции, определяемой его атрибутом `rect`. Метод получает один аргумент: поверхность для вывода элементов группы. На рис. 13.2 изображен первый пришелец.

После того как первый пришелец появится на экране, мы напишем код для вывода всего флота.



**Рис. 13.2.** Появился первый пришелец

## Создание флота

Чтобы нарисовать флот пришельцев, нам нужно решить, как заполнить верхнюю часть экрана пришельцами, не перегружая окно игры. Существует несколько способов решения этой задачи. Мы будем добавлять пришельцев в верхней части экрана, заполняя ее, пока места для нового пришельца не останется. Затем мы повторим этот процесс, заполняя экран по вертикали и добавляя ряды пришельцев.

### Создание ряда пришельцев

Теперь мы готовы сгенерировать полный ряд пришельцев. Чтобы создать полный ряд, мы сначала создадим одного пришельца, чтобы определить его ширину. Мы поместим пришельца в левую часть экрана, а затем будем добавлять его собратьев, пока не закончится пространство:

#### *alien\_invasion.py*

```
def _create_fleet(self):
 """Создает флот пришельцев."""
 # Создание пришельца и вычисление количества пришельцев в ряду.
 # Интервал между соседними пришельцами равен ширине пришельца.
 alien = Alien(self)
 alien_width = alien.rect.width

 ❶ current_x = alien_width
 ❷ while current_x < (self.settings.screen_width - 2 * alien_width):
 ❸ new_alien = Alien(self)
 ❹ new_alien.x = current_x
 new_alien.rect.x = current_x
 self.aliens.add(new_alien)
 ❺ current_x += 2 * alien_width
```

Мы вычисляем ширину пришельца благодаря первому созданному пришельцу и определяем переменную `current_x` ❶. Она хранит позицию по горизонтали следующего пришельца, размещаемого на экране. Изначально мы присваиваем ей значение, равное ширине одного пришельца, чтобы отодвинуть пришельца во флоте от левого края экрана.

Далее запускается цикл `while` ❷, добавляющий пришельцев, пока пространства по горизонтали достаточно. Чтобы определить, хватит ли места для размещения еще одного пришельца, мы сравниваем значение переменной `current_x` с максимальным значением. Попробуем определить этот цикл следующим образом:

```
while current_x < self.settings.screen_width:
```

Рабочий на первый взгляд способ приводит к тому, что последний пришелец в ряду оказывается у крайнего правого края экрана. Поэтому мы добавляем небольшое пространство с правой стороны. Пока у правого края есть свободное пространство, равное ширине как минимум двух пришельцев, цикл повторяется и добавляет еще одного пришельца в ряд.

При каждой итерации, когда горизонтального пространства на экране достаточно для продолжения цикла, мы хотим выполнить две задачи: создать пришельца в корректной позиции и определить горизонтальную позицию следующего пришельца в ряду. Мы создаем пришельца и присваиваем значение переменной `new_alien` ❸. Затем используем точную горизонтальную позицию согласно текущему значению переменной `current_x` ❹. Мы позиционируем прямоугольник пришельца согласно тому же значению по оси *X* и добавляем нового пришельца в группу `self.aliens`.

Наконец, мы инкрементируем значение переменной `current_x` ❺. Мы добавляем значение, равное двукратной ширине пришельца, к горизонтальной позиции, чтобы переместиться за пределы только что добавленного пришельца и допустить интервал между пришельцами. Интерпретатор вновь оценивает условие в начале цикла `while` и определяет, хватит ли места для еще одного пришельца. Когда места не останется, цикл завершится и мы получим полный ряд пришельцев.

Запустив программу «Инопланетное вторжение», вы увидите, что на экране появился первый ряд пришельцев (рис. 13.3).

## ПРИМЕЧАНИЕ

Не всегда очевидно, как именно выстроить цикл, подобный приведенному в этом подразделе. Одна из фишек программирования заключается в том, что первоначальные идеи решения подобной задачи необязательно должны быть правильными. Вполне допустимо написать цикл, размещающий пришельцев слишком далеко друг от друга, а затем корректировать код до тех пор, пока персонажи не разместятся должным образом.

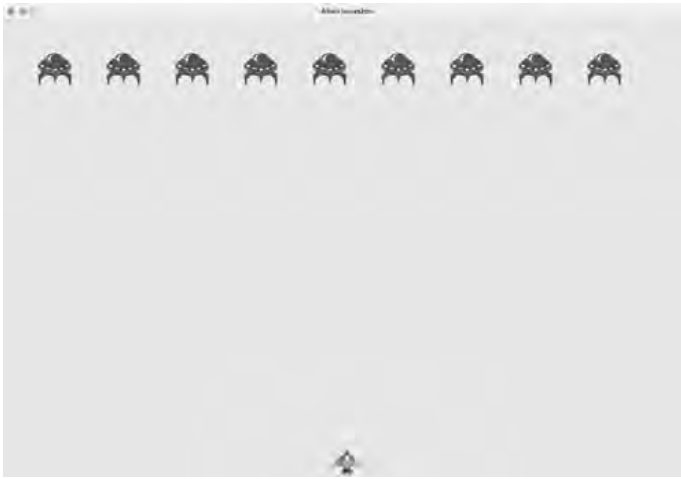


Рис. 13.3. Первый ряд пришельцев

### Рефакторинг функции `_create_fleet()`

Если бы создание флота на этом было завершено, то функцию `_create_fleet()`, пожалуй, можно было бы оставить в таком виде, но нам предстоит еще много работы, поэтому мы немного подчистим код функции. Мы добавим новый вспомогательный метод `_create_alien()` и вызовем его из `_create_fleet()`:

#### *alien\_invasion.py*

```
def _create_fleet(self):
 --пропуск--
 while current_x < (self.settings.screen_width - 2 * alien_width):
 self._create_alien(current_x)
 current_x += 2 * alien_width
```

```
❶ def _create_alien(self, x_position):
 """Создает пришельца и размещает его в ряду."""
 new_alien = Alien(self)
 new_alien.x = x_position
 new_alien.rect.x = x_position
 self.aliens.add(new_alien)
```

Метод `_create_alien()` должен получать еще один параметр, кроме `self`: значение по оси *X*, указывающее, куда должен быть помещен пришелец ❶. В теле `_create_alien()` мы используем тот же код, созданный для `_create_fleet()`, не считая того, что вместо `current_x` применяем параметр `x_position`. Рефакторинг упрощает добавление новых рядов и создание всего флота.

## Добавление рядов

Чтобы завершить создание флота, мы будем добавлять новые ряды, пока не закончится пространство. Мы используем вложенный цикл — поместим текущий цикл в еще один цикл `while`. Внутренний цикл будет размещать пришельцев в ряд по горизонтали согласно их координатам по оси *X*. Внешний — размещать по вертикали согласно их координатам по оси *Y*. Мы перестанем добавлять ряды, добравшись до нижней части экрана и оставив достаточно места для корабля и отстреливания пришельцев.

Ниже показано, как вложить описанные два цикла `while` в `_create_fleet()`:

### *alien\_invasion.py*

```
def _create_fleet(self):
 """Создает флот пришельцев."""
 # Создание пришельца и добавление других, пока остается место.
 # Расстояние между пришельцами составляет одну ширину
 # и одну высоту пришельца.
 alien = Alien(self)
 ❶ alien_width, alien_height = alien.rect.size

 ❷ current_x, current_y = alien_width, alien_height
 ❸ while current_y < (self.settings.screen_height - 3 * alien_height):
 while current_x < (self.settings.screen_width - 2 * alien_width):
 ❹ self._create_alien(current_x, current_y)
 current_x += 2 * alien_width

 ❺ # Конец ряда: сбрасываем значение x и инкрементируем значение y.
 current_x = alien_width
 current_y += 2 * alien_height
```

Нам понадобится значение высоты пришельца, чтобы разместить ряды, поэтому мы извлекаем ширину и высоту пришельца с помощью атрибута `size` объекта `rect` пришельца ❶. Атрибут `size` объекта `rect` — это кортеж, содержащий его ширину и высоту.

Далее мы задаем начальные координаты по осям *X* и *Y* для размещения первого пришельца ❷. Мы размещаем его на одну ширину пришельца правее и на одну высоту пришельца ниже. Затем определяем цикл `while`, управляющий тем, сколько рядов будет размещено на экране ❸. Пока значение по оси *Y* для следующего ряда меньше высоты экрана минус три высоты пришельца, мы будем добавлять ряды. (Если нужное пространство не остается, исправим код позже.)

Мы вызываем функцию `_create_alien()` и передаем ей значения по осям *Y* и *X* ❹. Чуть позже мы изменим ее.

Обратите внимание на отступ в последних двух строках кода ❺. Они расположены внутри внешнего цикла `while` и вне внутреннего цикла `while`. Этот код

запускается после завершения внутреннего цикла, однократно после создания каждого ряда. После добавления каждого ряда мы сбрасываем значение переменной `current_x`, чтобы первый пришелец в следующем ряду был помещен в ту же позицию, что и первый пришелец в предыдущих рядах. Затем добавляем двукратное значение высоты пришельца к текущему значению переменной `current_y`, чтобы следующий ряд располагался ниже на экране. Эти отступы в коде крайне важны; если у вас не получается правильно поместить флот пришельцев при запуске программы `alien_invasion.py`, то проверьте отступы всех строк в этих вложенных циклах.

Нам нужно изменить код функции `_create_alien()`, чтобы настроить положение пришельцев по вертикали:

```
def _create_alien(self, x_position, y_position):
 """Создает пришельца и помещает его во флот."""
 new_alien = Alien(self)
 new_alien.x = x_position
 new_alien.rect.x = x_position
 new_alien.rect.y = y_position
 self.aliens.add(new_alien)
```

Мы изменили определение метода, чтобы использовать значение по оси *Y* нового пришельца, и настраиваем положение прямоугольника по вертикали в теле метода.

Если вы запустите игру сейчас, то увидите целый флот пришельцев (рис. 13.4).

В следующем разделе мы приведем флот в движение.

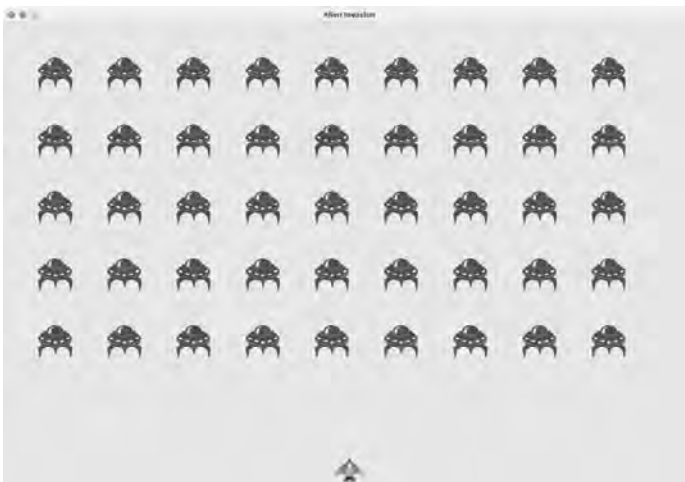


**Рис. 13.4.** На экране появился весь флот пришельцев

## УПРАЖНЕНИЯ

**13.1. Звезды.** Найдите изображение звезды. Создайте на экране сетку из звезд.

**13.2. Улучшенные звезды.** Чтобы звезды выглядели более реалистично, следует внести случайное отклонение при их размещении. Вспомните, что случайные числа генерируются следующим образом:

```
from random import randint
random_number = randint(-10,10)
```

Этот код возвращает случайное целое число в диапазоне от  $-10$  до  $10$ . Используя свой код из упражнения 13.1, измените позицию каждой звезды на случайную величину.

## Перемещение флота

Флот пришельцев должен двигаться вправо по экрану, пока не дойдет до края; тогда флот опускается на заданную величину и начинает двигаться в обратном направлении. Это продолжается до тех пор, пока все пришельцы не будут сбиты, один из них не столкнется с кораблем или не достигнет низа экрана. Начнем с перемещения флота вправо.

### Перемещение пришельцев вправо

Чтобы корабли пришельцев перемещались по экрану, мы воспользуемся методом `update()` из программы `alien.py`, который будет вызываться для каждого пришельца в группе. Сначала добавим настройку для управления скоростью каждого пришельца:

#### *settings.py*

```
def __init__(self):
 --пропуск--
 # Настройки пришельцев
 self.alien_speed = 1.0
```

Настройка используется в реализации метода `update()` в файле `alien.py`:

#### *alien.py*

```
def __init__(self, ai_game):
 """Инициализирует пришельца и задает его начальную позицию."""
 super().__init__()
 self.screen = ai_game.screen
 self.settings = ai_game.settings
 --пропуск--

 def update(self):
 """Перемещает пришельца вправо."""
 1 self.x += self.settings.alien_speed
 2 self.rect.x = self.x
```



Параметр `settings` создается в методе `__init__()`, чтобы к скорости пришельца можно было обратиться в методе `update()`. При каждом обновлении позиции пришельца мы смещаем его вправо на величину, хранящуюся в `alien_speed`. Точная позиция пришельца хранится в атрибуте `self.x`, который может принимать вещественные значения ❶. Затем значение `self.x` используется для обновления позиции прямоугольника пришельца ❷.

В основном цикле `while` уже содержатся вызовы обновления корабля и снарядов. Теперь необходимо обновить позицию каждого пришельца:

#### *alien\_invasion.py*

```
while True:
 self._check_events()
 self.ship.update()
 self._update_bullets()
 self._update.aliens()
 self._update_screen()
 self.clock.tick(60)
```

Сейчас мы напишем код управления флотом, для которого будет создан новый метод `_update.aliens()`. Позиции пришельцев обновляются после обновления снарядов, так как скоро мы будем проверять, попали ли какие-либо снаряды в пришельцев.

Местоположение этого метода в модуле не критично. Но для улучшения структуры кода мы поместим его сразу же после `_update_bullets()` в соответствии с порядком вызова методов в цикле `while`. Первая версия `_update.aliens()` выглядит так:

#### *alien\_invasion.py*

```
def _update.aliens(self):
 """Обновляет позиции всех пришельцев во флоте."""
 self.aliens.update()
```

Мы используем метод `update()` для группы `aliens`, что приводит к автоматическому вызову метода `update()` каждого пришельца. Если вы запустите «Инопланетное вторжение» сейчас, то увидите, как флот двигается вправо и исчезает за краем экрана.

## **Создание настроек для направления флота**

Теперь мы создадим настройки, благодаря которым флот перемещается вниз по экрану, а потом влево при достижении правого края экрана. Вот как реализуется это поведение:

#### *settings.py*

```
Настройки пришельцев
self.alien_speed = 1.0
self.fleet_drop_speed = 10
fleet_direction = 1 обозначает движение вправо; а -1 – влево.
self.fleet_direction = 1
```

Настройка `fleet_drop_speed` управляет величиной снижения флота при достижении им края. Эту скорость полезно отделить от горизонтальной скорости пришельцев, чтобы эти две скорости можно было изменять независимо.

Для настройки `fleet_direction` можно использовать строковое значение (например, `'left'` или `'right'`), но, скорее всего, в итоге придется воспользоваться набором операторов `if-elif` для проверки направления. Сейчас направлений всего два, поэтому мы используем значения 1 и `-1` и будем переключаться между ними при каждом изменении направления флота. (Числа в данном случае особенно удобны, поскольку при движении вправо координата `x` каждого пришельца должна увеличиваться, а при перемещении влево — уменьшаться.)

## Проверка достижения края

Помимо вышеперечисленных, нам понадобится метод, позволяющий проверить, достиг ли пришелец одного из двух краев. Для этого необходимо внести в метод `update()` изменение, позволяющее каждому пришельцу двигаться в соответствующем направлении. Этот код является частью класса `Alien`:

*alien.py*

```
def check_edges(self):
 """Возвращает True, если пришелец находится у края экрана."""
 screen_rect = self.screen.get_rect()
 ❶ return (self.rect.right >= screen_rect.right) or (self.rect.left <= 0)

def update(self):
 """Перемещает пришельца влево или вправо."""
 ❷ self.x += (self.settings.alien_speed * self.settings.fleet_direction)
 self.rect.x = self.x
```

Вызов нового метода `check_edges()` для любого пришельца позволяет проверить, достиг ли он левого или правого края. У пришельца, находящегося у правого края, атрибут `right` его объекта `rect` больше или равен атрибуту `right` объекта `rect` экрана. У пришельца, находящегося у левого края, значение `left` меньше либо равно 0 ❶. Вместо того чтобы размещать это условие в блоке `if`, мы добавили его непосредственно в оператор `return`. В этом случае метод вернет `True`, если пришелец находится у правого или левого края, и `False`, если он не находится ни у одного из краев.

В метод `update()` будут внесены изменения, которые позволяют осуществлять перемещение влево и вправо. Для этого скорость пришельца умножается на значение `fleet_direction` ❷. Если значение `fleet_direction` равно 1, то значение `alien_speed` прибавляется к текущей позиции пришельца и он перемещается вправо; если же значение `fleet_direction` равно `-1`, то значение вычитается из позиции пришельца (который перемещается влево).

## Снижение флота и смена направления

Когда пришелец доходит до края, весь флот должен опуститься вниз и изменить направление движения. Это означает, что в класс `AlienInvasion` придется внести изменения, поскольку именно здесь программа проверяет, достиг ли какой-либо пришелец левого или правого края. Для этого мы напишем функции `_check_fleet_edges()` и `_change_fleet_direction()`, а затем изменим `_update_aliens()`. Новые методы будут располагаться после `_create_alien()`, но я еще раз подчеркну, что конкретное размещение этих методов в классе несущественно.

### *alien\_invasion.py*

```
def _check_fleet_edges(self):
 """Реагирует на достижение пришельцем края экрана."""
❶ for alien in self.aliens.sprites():
 if alien.check_edges():
❷ self.change_fleet_direction()
 break

def _change_fleet_direction(self):
 """Опускает весь флот и меняет его направление."""
 for alien in self.aliens.sprites():
❸ alien.rect.y += self.settings.fleet_drop_speed
 self.settings.fleet_direction *= -1
```

Код `_check_fleet_edges()` перебирает флот и вызывает `check_edges()` для каждого пришельца ❶. Если `check_edges()` возвращает `True`, значит, пришелец находится у края и весь флот должен сменить направление, поэтому вызывается функция `_change_fleet_direction()` и происходит выход из цикла ❷. Данная функция перебирает пришельцев и уменьшает высоту каждого из них с помощью настройки `fleet_drop_speed` ❸; затем направление `fleet_direction` меняется на противоположное, для чего текущее значение умножается на `-1`. Строка, изменяющая направление, не является частью цикла `for`. Вертикальная позиция должна меняться для каждого пришельца, но направление всего флота должно измениться однократно.

Изменения в функции `_update_aliens()` выглядят так:

### *alien\_invasion.py*

```
def _update_aliens(self):
 """
 Проверяет, достиг ли флот края экрана, с последующим обновлением
 позиций всех пришельцев во флоте.
 """
 self._check_fleet_edges()
 self.aliens.update()
```

Перед обновлением позиции каждого пришельца будет вызываться метод `_check_fleet_edges()`.

Если запустить игру сейчас, то флот будет двигаться влево-вправо между краями экрана и опускаться каждый раз, когда доберется до края. Теперь можно переходить к реализации уничтожения пришельцев и отслеживания тех, кто сталкивается с кораблем или достигает нижнего края экрана.

### УПРАЖНЕНИЯ

**13.3. Капли.** Найдите изображение дождевой капли и создайте сетку из капель. Капли должны постепенно опускаться вниз и исчезать у нижнего края экрана.

**13.4. Дождь.** Измените свой код в упражнении 13.3 так, чтобы при исчезновении ряда капель у нижнего края экрана новый ряд появлялся у верхнего края и начинал падение.

## Уничтожение пришельцев

Итак, мы создали корабль и флот пришельцев. Но снаряды, достигая пришельцев, просто проходят насквозь, поскольку программа не проверяет коллизии. В игровом программировании *коллизией* (collision) называется перекрытие игровых элементов. Чтобы снаряды сбивали пришельцев, мы используем функцию `sprite.groupcollide()` для выявления коллизий между элементами двух групп.

### Выявление коллизий

Когда снаряд попадает в пришельца, программа должна немедленно узнать об этом, чтобы сбитый пришелец исчез с экрана. Для этого мы будем проверять коллизии сразу же после обновления позиции снаряда.

Метод `sprite.groupcollide()` сравнивает прямоугольник `rect` каждого элемента с прямоугольником `rect` каждого элемента другой группы. В данном случае он сравнивает прямоугольник каждого снаряда с прямоугольником каждого пришельца и возвращает словарь со снарядами и пришельцами, между которыми обнаружены коллизии. Каждый ключ в словаре представляет снаряд, а связанное с ним значение — пришельца, в которого попал снаряд. (Этот словарь будет использоваться в реализации системы подсчета очков в главе 14.)

Для проверки коллизий добавьте в конец функции `update_bullets()` следующий код:

#### *alien\_invasion.py*

```
def _update_bullets(self):
 """Обновляет позиции снарядов и удаляет старые снаряды."""
 --пропуск--
```

```
Проверка попаданий в пришельцев.
При обнаружении попадания удалить снаряд и пришельца.
collisions = pygame.sprite.groupcollide(
 self.bullets, self.aliens, True, True)
```

Новый код перебирает сначала все снаряды в `self.bullets`, а затем всех пришельцев в `self.aliens`. Каждый раз, когда обнаруживается, что прямоугольник снаряда и пришелец пересекаются, `groupcollide()` добавляет пару «ключ — значение» в возвращаемый словарь. Два аргумента `True` сообщают Pygame, нужно ли удалять столкнувшиеся объекты: снаряд и пришельца. (Чтобы создать сверхмощный снаряд, который будет уничтожать всех пришельцев на своем пути, можно передать в первом аргументе `False`, а во втором `True`. Пришельцы, в которых попадает снаряд, будут исчезать, но все снаряды будут оставаться активными до верхнего края экрана.)

Если вы запустите «Инопланетное вторжение» сейчас, то пришельцы, в которых попадает снаряд, будут исчезать с экрана. На рис. 13.5 изображен частично уничтоженный флот.



**Рис. 13.5.** Снаряды уничтожают пришельцев!

### **Создание увеличенных снарядов для тестирования**

Многие игровые возможности можно протестировать, просто запустив игру, но некоторые аспекты слишком утомительно проверять в обычной версии игры. Например, чтобы проверить, правильно ли обрабатывается уничтожение последнего пришельца, нам пришлось бы несколько раз сбивать всех пришельцев на экране.

Для тестирования конкретных аспектов игры можно изменить настройки так, чтобы упростить конкретную область. Например, можно уменьшить экран, чтобы на нем

было меньше пришельцев, или увеличить скорость снаряда и количество снарядов, одновременно находящихся на экране.

Мое любимое изменение при тестировании игры «Инопланетное вторжение» — использование сверхшироких снарядов, которые остаются активными даже после попадания в пришельца (рис. 13.6). Попробуйте задать настройке `bullet_width` значение 300 (или даже 3000!) и посмотрите, сколько времени вам понадобится для уничтожения флота пришельцев!

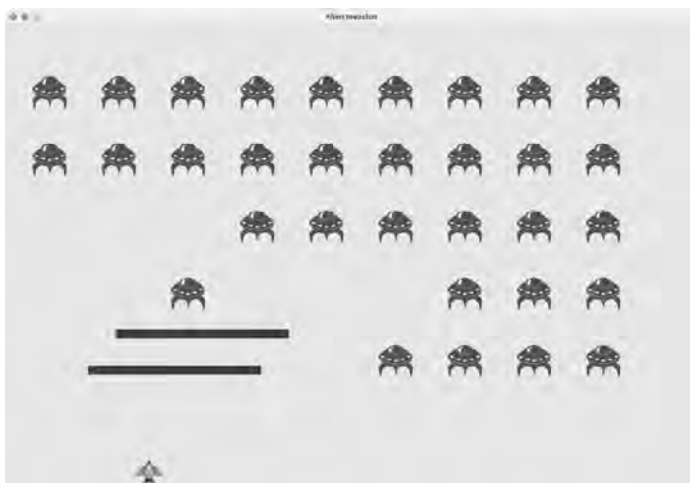


**Рис. 13.6.** Сверхмощные снаряды упрощают тестирование некоторых аспектов игры

Такие изменения повышают эффективность тестирования, а заодно могут подсказывать идеи для всевозможных игровых бонусов. (Только не забудьте восстановить нормальное состояние настроек после завершения тестирования.)

## Восстановление флота

Одна из ключевых особенностей игры «Инопланетное вторжение» — бесконечные орды пришельцев: каждый раз, когда вы уничтожаете один флот, на его месте появляется другой.

Чтобы после уничтожения одного флота появлялся другой, сначала нужно убедиться в том, что группа `aliens` пуста. Если да, то вызывается метод `_create_fleet()`. Проверка будет выполняться в конце метода `_update_bullets()`, поскольку именно здесь уничтожаются отдельные пришельцы:

### *alien\_invasion.py*

```
def _update_bullets(self):
 --пропуск--
```

```
❶ if not self.aliens:
 # Уничтожение существующих снарядов и создание нового флота.
❷ self.bullets.empty()
 self._create_fleet()
```

Программа проверяет, пуста ли группа `aliens` ❶. Пустая группа интерпретируется как `False`; это самый простой способ проверить группу на наличие элементов. Если группа пуста, то все существующие снаряды удаляются методом `empty()`, который убирает все существующие спрайты из группы ❷. Вызов метода `_create_fleet()` снова заполняет экран пришельцами.

Теперь сразу же после уничтожения текущего флота на экране появляется новый.

## Ускорение снарядов

Попытавшись стрелять по пришельцам в текущем состоянии игры, можно заметить, что скорость движения снарядов неоптимальна — в вашей системе она может быть слишком высокой или низкой. На этой стадии можно изменить настройки, чтобы игра была более интересной и приятной. Имейте в виду, что она будет постепенно ускоряться, поэтому не делайте ее слишком быстрой в начале.

Скорость снарядов можно увеличить с помощью настройки `bullet_speed` в файле `settings.py`. Например, если задать в моей системе значение 2.5, то снаряды будут двигаться по экрану немного быстрее:

### *settings.py*

```
Настройки снарядов
self.bullet_speed = 2.5
self.bullet_width = 3
--пропуск--
```

Оптимальное значение этой настройки зависит от производительности вашей системы. Найдите значение, которое лучше подходит для вашей конкретной конфигурации.

## Рефакторинг метода `_update_bullets()`

Переработаем метод `_update_bullets()`, чтобы он не решал такое количество разных задач. Код обработки коллизий будет выделен в отдельный метод:

### *alien\_invasion.py*

```
def _update_bullets(self):
 --пропуск--
 # Уничтожение исчезнувших снарядов.
 for bullet in self.bullets.copy():
 if bullet.rect.bottom <= 0:
 self.bullets.remove(bullet)

 self._check_bullet_alien_collisions()
```

```
def _check_bullet_alien_collisions(self):
 """Обработывает коллизии снарядов с пришельцами."""
 # Удаление снарядов и пришельцев, участвующих в коллизиях.
 collisions = pygame.sprite.groupcollide(
 self.bullets, self.aliens, True, True)
 if not self.aliens:
 # Уничтожение существующих снарядов и создание нового флота.
 self.bullets.empty()
 self._create_fleet()
```

Мы создали новый метод `_check_bullet_alien_collisions()` для выявления коллизий между снарядами и пришельцами и реагирования на уничтожение всего флота. Это сделано для того, чтобы сократить длину метода `_update_bullets()` и упростить дальнейшую разработку.

## УПРАЖНЕНИЯ

**13.5. Боковая стрельба-2.** После упражнения 12.6 «Боковая стрельба» программа была серьезно доработана. В этом упражнении вам предлагается усовершенствовать «Боковую стрельбу» до того же состояния, к которому была приведена игра «Инопланетное вторжение». Добавьте флот пришельцев и заставьте их перемещаться по горизонтали по направлению к кораблю. Другой вариант: напишите код, который размещает пришельцев в случайных позициях у правого края экрана, а затем заставляет их двигаться к кораблю. Кроме того, напишите код, который заставляет пришельцев исчезать при попаданиях.

## Завершение игры

Какое удовольствие от игры, в которой невозможно проиграть? Если игрок не успеет сбить флот достаточно быстро, то пришельцы уничтожат корабль при столкновении. При этом количество кораблей, используемых игроком, ограничено, и корабль уничтожается, когда пришелец достигает нижнего края экрана. Игра завершается в тот момент, когда у игрока кончатся все корабли.

### Обнаружение коллизий между пришельцами и кораблем

Начнем с проверки коллизий между пришельцами и кораблем, чтобы мы могли правильно обработать столкновения. Такие коллизии проверяются немедленно после обновления позиции каждого пришельца в классе `AlienInvasion`:

*alien\_invasion.py*

```
def _update_aliens(self):
 --пропуск--
 self.aliens.update()

 # Проверка коллизий "пришелец — корабль".
 if pygame.sprite.spritecollideany(self.ship, self.aliens):
 print("Ship hit!!!")
```

①  
②



Функция `spritecollideany()` получает два аргумента: спрайт и группу. Функция пытается найти любой элемент группы, вступивший в коллизию со спрайтом, и останавливает цикл по группе сразу же после обнаружения столкнувшихся элементов. В данном случае он перебирает группу `aliens` и возвращает первого пришельца, столкнувшегося с кораблем `ship`.

Если ни одна коллизия не обнаружена, то `spritecollideany()` возвращает `None` и блок `if` не выполняется ❶. Если же будет обнаружен пришелец, столкнувшийся с кораблем, то метод возвращает этого пришельца и выполняется блок `if`: выводится сообщение `Ship hit!!!` ❷. При столкновении пришельца с кораблем необходимо выполнить ряд операций: удалить всех оставшихся пришельцев и снаряды, вернуть корабль в центр и создать новый флот. Прежде чем писать код всех этих операций, необходимо убедиться в том, что решение, позволяющее обнаружить коллизии между пришельцами и кораблем, работает правильно. Вызов функции `print()` всего лишь позволяет легко проверить правильность обнаружения коллизий.

Если вы запустите «Инопланетное вторжение», то при столкновении пришельца с кораблем в терминальном окне появляется сообщение `Ship hit!!!`. В ходе тестирования этого аспекта присвойте `fleet_drop_speed` более высокое значение (например, 50 или 100), чтобы пришельцы быстрее добирались до вашего корабля.

## Обработка столкновений с кораблем

Теперь нужно разобраться, что же происходит при столкновении пришельца с кораблем. Вместо того чтобы уничтожать экземпляр `ship` и создавать новый, мы будем подсчитывать количество уничтоженных кораблей; для этого следует организовать сбор статистики по игре. (Статистика пригодится и для подсчета очков.)

Напишем новый класс `GameStats` для ведения статистики и сохраним его в файле `game_stats.py`:

### `game_stats.py`

```
class GameStats:
 """Отслеживает статистику для игры "Инопланетное вторжение"."""

 def __init__(self, ai_game):
 """Инициализирует статистику."""
 self.settings = ai_game.settings
 self.reset_stats()

 def reset_stats(self):
 """Инициализирует статистику, изменяющуюся в ходе игры."""
 self.ships_left = self.settings.ship_limit
```

На все время работы игры «Инопланетное вторжение» будет создаваться один экземпляр `GameStats`, но часть статистики должна сбрасываться в начале каждой новой игры. Для этого большая часть статистики будет инициализироваться в методе `reset_stats()` вместо `__init__()`. Этот метод будет вызываться из `__init__()`, чтобы статистика правильно инициализировалась при первом создании экземпляра `GameStats` ❶, а метод `reset_stats()` будет вызываться в начале каждой новой игры.

Пока в игре используется всего один вид статистики — значение `ships_left`, изменяющееся в ходе игры. Количество кораблей в начале игры хранится в файле `settings.py` под именем `ship_limit`:

***settings.py***

```
Настройки корабля
self.ship_speed = 1.5
self.ship_limit = 3
```

Кроме того, необходимо внести ряд изменений в файл `alien_invasion.py`, чтобы можно было создать экземпляр класса `GameStats`. Начнем с изменения операторов `import` в начале файла:

***alien\_invasion.py***

```
import sys
from time import sleep

import pygame

from settings import Settings
from game_stats import GameStats
from ship import Ship
--nponyск--
```

Мы импортируем функцию `sleep()` из модуля `time` стандартной библиотеки Python, чтобы игру можно было на короткий момент приостановить в момент столкновения с кораблем. Кроме того, импортируем класс `GameStats`.

Экземпляр `GameStats` создается в методе `__init__()`:

***alien\_invasion.py***

```
def __init__(self):
 --nponyск--
 self.screen = pygame.display.set_mode(
 (self.settings.screen_width, self.settings.screen_height))
 pygame.display.set_caption("Alien Invasion")

 # Создание экземпляра для хранения игровой статистики.
 self.stats = GameStats(self)

 self.ship = Ship(self)
 --nponyск--
```

Экземпляр создается после создания игрового окна, но перед определением других игровых элементов (например, корабля).

Когда пришелец сталкивается с кораблем, программа уменьшает количество оставшихся кораблей на 1, уничтожает всех существующих пришельцев и снаряды, создает новый флот и возвращает корабль в середину экрана. Кроме того, игра ненадолго приостанавливается, чтобы игрок заметил столкновение и перестроился перед появлением нового флота.

Большая часть этого кода будет вынесена в новый метод `_ship_hit()`. Он вызывается из метода `_update aliens()` при столкновении пришельца с кораблем:

#### *alien\_invasion.py*

```
def _ship_hit(self):
 """Обрабатывает столкновение корабля с пришельцем."""
 # Уменьшение ships_left.
 ❶ self.stats.ships_left -= 1

 # Очистка групп aliens и bullets.
 ❷ self.aliens.empty()
 self.bullets.empty()

 # Создание нового флота и размещение корабля в центре.
 ❸ self._create_fleet()
 self.ship.center_ship()

 # Пауза.
 ❹ sleep(0.5)
```

Новый метод `_ship_hit()` управляет реакцией игры на столкновение корабля с пришельцем. Внутри `_ship_hit()` количество оставшихся кораблей уменьшается на 1 ❶, после чего происходит очистка групп `aliens` и `bullets` ❷.

Затем программа создает новый флот и выравнивает корабль по центру нижнего края ❸. (Вскоре мы добавим метод `center_ship()` в класс `Ship`.) Наконец, после внесения изменений во все игровые элементы, но до перерисовки изменений на экране делается короткая пауза, чтобы игрок увидел, что его корабль столкнулся с пришельцем ❹. Вызов функции `sleep()` приостанавливает программу на 0,5 секунды. После завершения паузы управление передается методу `_update_screen()`, который перерисовывает новый флот на экране.

Внутри метода `_update aliens()` вызов функции `print()` заменяется вызовом `_ship_hit()` при столкновении пришельца с кораблем:

#### *alien\_invasion.py*

```
def _update_aliens(self):
 --nponyск--
 if pygame.sprite.spritecollideany(self.ship, self.aliens):
 self._ship_hit()
```

Новый метод `center_ship()` добавляется в конец файла `ship.py`:

#### *ship.py*

```
def center_ship(self):
 """Размещает корабль в центре нижней части экрана."""
 self.rect.midbottom = self.screen_rect.midbottom
 self.x = float(self.rect.x)
```

Выравнивание корабля по центру выполняется так же, как и в методе `__init__()`. После выравнивания сбрасывается атрибут `self.x`, чтобы в программе отслеживалась точная позиция корабля.

### ПРИМЕЧАНИЕ

Обратите внимание: программа никогда не создает несколько кораблей. Один экземпляр ship используется на протяжении всей игры, а при столкновении с пришельцем он просто возвращается к центру экрана. О том, что у игрока не осталось ни одного корабля, программа узнает из атрибута `ships_left`.

Запустите игру, подстрелите нескольких пришельцев, а затем позвольте одному из них столкнуться с кораблем. Происходит небольшая пауза, на экране появляется новый флот вторжения, а корабль возвращается в центр нижней части экрана.

### Достижение нижнего края экрана

Если пришелец добирается до нижнего края экрана, то программа будет реагировать так же, как при столкновении с кораблем. Добавьте для проверки этого условия новый метод в файл `alien_invasion.py`:

#### `alien_invasion.py`

```
def _check.aliens_bottom(self):
 """Проверяет, добрались ли пришельцы до нижнего края экрана."""
 for alien in self.aliens.sprites():
 ❶ if alien.rect.bottom >= self.settings.screen_height:
 # Происходит то же, что при столкновении с кораблем.
 self._ship_hit()
 break
```

Метод `check.aliens_bottom()` проверяет, есть ли хотя бы один пришелец, добравшийся до нижнего края экрана. Условие выполняется, когда атрибут `rect.bottom` пришельца больше атрибута `rect.bottom` экрана или равен ему ❶. Если пришелец добрался до низа, то вызывается функция `_ship_hit()`. Если хотя бы один пришелец столкнулся с нижним краем, то проверять остальных уже не нужно, поэтому после вызова `_ship_hit()` цикл прерывается.

Этот метод вызывается из метода `_update.aliens()`:

#### `alien_invasion.py`

```
def _update.aliens(self):
 --пропуск--
 # Проверка коллизий "пришелец — корабль".
 if pygame.sprite.spritecollideany(self.ship, self.aliens):
 self._ship_hit()

 # Проверить, сталкиваются ли пришельцы с нижним краем экрана.
 self._check.aliens_bottom()
```

Метод `_check.aliens_bottom()` вызывается после обновления позиций всех пришельцев и после проверки коллизий «пришелец — корабль». Теперь новый флот будет появляться как при столкновении корабля с пришельцем, так и в том случае, если кто-то из пришельцев смог добраться до нижнего края экрана.

## Конец игры

Игра «Инопланетное вторжение» кажется завершенной, но длится бесконечно. Значение `ships_left` просто продолжает уходить в отрицательную бесконечность. Добавим новый атрибут — флаг `game_active`, который завершает игру после потери последнего корабля. Этот флаг устанавливается в конце метода `__init__()` в классе `AlienInvasion`:

### *alien\_invasion.py*

```
def __init__(self, ai_game):
 --nponyск--
 # Игра "Инопланетное вторжение" запускается в активном состоянии.
 self.game_active = True
```

Добавим в `ship_hit()` код, который сбрасывает флаг `game_active` в состояние `False` при потере игроком последнего корабля:

### *alien\_invasion.py*

```
def _ship_hit(self):
 """Обрабатывает столкновение корабля с пришельцем."""
 if stats.ships_left > 0:
 # Уменьшение ships_left.
 self.stats.ships_left -= 1
 --nponyск--
 # Пауза.
 sleep(0.5)
 else:
 self.game_active = False
```

Большая часть кода `_ship_hit()` осталась неизменной. Весь существующий код был перемещен в блок `if`, который проверяет, остался ли у игрока хотя бы один корабль. Если корабли не кончились, то программа создает новый флот, делает паузу и продолжает игру. Если же игрок потерял последний корабль, то флаг `game_active` переводится в состояние `False`.

## Определение исполняемых частей игры

В файле `alien_invasion.py` необходимо определить части игры, которые должны выполняться всегда, и те части, которые будут выполняться только при активной игре:

### *alien\_invasion.py*

```
def run_game(self):
 """Запускает основной цикл игры."""
 while True:
 self._check_events()

 if self.game_active:
 self.ship.update()
```

```
self._update_bullets()
self._update.aliens()

self._update_screen()
self.clock.tick(60)
```

В основном цикле всегда должна вызываться функция `_check_events()`, даже если игра находится в неактивном состоянии. Например, программа все равно должна узнать о том, что пользователь нажал клавишу `Q` для завершения игры или щелкнул на кнопке закрытия окна. Кроме того, экран должен обновляться в то время, пока игрок решает, хочет ли он начать новую игру. Остальные вызовы функций должны происходить только при активной игре, поскольку в то время, когда игра неактивна, обновлять позиции игровых элементов не нужно.

В обновленной версии игра должна останавливаться после потери игроком последнего корабля.

## УПРАЖНЕНИЯ

**13.6. Конец игры.** В упражнении «Боковая стрельба» отслеживайте, сколько пришельцев сбил игрок и сколько раз пришельцы столкнулись с кораблем. Определите разумное условие завершения игры и останавливайте ее при возникновении этой ситуации.

## Резюме

В этой главе вы научились добавлять в игру большое количество одинаковых элементов на примере флота пришельцев. Вы узнали, как использовать вложенные циклы для создания сетки с элементами, а также привели игровые элементы в движение, вызывая метод `update()` каждого элемента. Вы научились управлять перемещением объектов на экране и обрабатывать различные события (например, достижение края экрана). Кроме того, вы узнали, как обнаруживать коллизии и реагировать на них (на примере попаданий снарядов в пришельцев и столкновений пришельцев с кораблем). В завершение главы вы изучили, как вести игровую статистику и использовать флаг `game_active` для проверки окончания игры.

В последней главе этого проекта будет добавлена кнопка **Play**, чтобы игрок мог самостоятельно запустить свою первую игру, а также повторить игру после ее завершения. После каждого уничтожения вражеского флота скорость игры будет возрастать, а мы реализуем систему подсчета очков. В результате вы получите полностью рабочую игру!

# 14

## Игровой счет



В этой главе создание игры «Инопланетное вторжение» будет завершено. Мы добавим кнопку **Play** для запуска игры по желанию игрока или перезапуска игры после ее завершения.

Мы также изменим игру, чтобы она ускорялась при переходе игрока на следующий уровень, и реализуем систему подсчета очков.

К концу главы вы будете знать достаточно, чтобы заняться разработкой игр, сложность которых нарастает по ходу игры и в которых реализована система подсчета очков.

### Добавление кнопки **Play**

В этом разделе мы добавим кнопку **Play**, которая отображается перед началом игры и появляется после ее завершения, чтобы игрок мог сыграть снова.

В текущей версии игра начинается сразу же после запуска программы `alien_invasion.py`. После очередных изменений игра будет запускаться в неактивном состоянии и предлагать игроку нажать кнопку **Play** для запуска. Для этого добавьте в метод `__init__()` класса `AlienInvasion` следующий код:

#### *alien\_invasion.py*

```
def __init__(self):
 """Инициализирует игру и создает игровые ресурсы."""
 pygame.init()
 --пропуск--

 # Игра запускается в неактивном состоянии.
 self.game_active = False
```

Итак, программа запускается в неактивном состоянии, а игру можно начать только нажатием кнопки **Play**.

## Создание класса *Button*

Поскольку в Pygame не существует встроенного метода создания кнопок, мы напишем класс `Button` для рисования заполненного прямоугольника с текстовой надписью. Следующий код может использоваться для создания кнопок в любой игре. Ниже приведена первая часть класса `Button`; сохраните ее в файле `button.py`:

### *button.py*

```
import pygame.font
```

```
class Button:
```

```
 """Класс для создания кнопок для игры."""
```

- ```
❶ def __init__(self, ai_game, msg):
    """Инициализирует атрибуты кнопки."""
    self.screen = ai_game.screen
    self.screen_rect = self.screen.get_rect()

    # Назначение размеров и свойств кнопок.
❷ self.width, self.height = 200, 50
    self.button_color = (0, 135, 0)
    self.text_color = (255, 255, 255)
❸ self.font = pygame.font.SysFont(None, 48)

    # Создание объекта rect кнопки и выравнивание по центру экрана.
❹ self.rect = pygame.Rect(0, 0, self.width, self.height)
    self.rect.center = self.screen_rect.center

    # Сообщение кнопки создается только один раз.
❺ self.prep_msg(msg)
```

Сначала программа импортирует модуль `pygame.font`, который позволяет Pygame выводить текст на экран. Метод `__init__()` получает параметры `self`, объект `ai_game` и строку `msg` с текстом кнопки ❶. Затем устанавливаются размеры кнопки ❷, после чего атрибуты `button_color` и `text_color` задаются так, чтобы прямоугольник кнопки был окрашен в ярко-зеленый цвет, а текст выводился белым цветом.

Далее происходит подготовка атрибута `font` для вывода текста ❸. Аргумент `None` сообщает Pygame, что для текста должен использоваться шрифт по умолчанию, а значение `48` определяет кегль. Чтобы выровнять кнопку по центру экрана, мы создаем объект `rect` для кнопки ❹ и задаем его атрибут `center` в соответствии с одноименным атрибутом экрана.

Pygame выводит строку текста в виде графического изображения. Эта задача решается путем вызова метода `_prep_msg()` ❺.

Код `_prep_msg()` выглядит так:

button.py

```
def _prep_msg(self, msg):
    """Преобразует msg в прямоугольник и выравнивает текст по центру."""
```



```
❶ self.msg_image = self.font.render(msg, True, self.text_color,  
    self.button_color)  
❷ self.msg_image_rect = self.msg_image.get_rect()  
  self.msg_image_rect.center = self.rect.center
```

Метод `_prep_msg()` должен получать параметр `self` и текст, который нужно вывести в графическом виде (`msg`). Вызов метода `font.render()` преобразует текст, хранящийся в `msg`, в изображение, которое затем сохраняется в `self.msg_image` ❶. Методу `font.render()` также передается логический признак режима *сглаживания* (antialiasing) текста. В остальных аргументах заданы цвет шрифта и цвет фона. В нашем примере режим сглаживания включен (`True`), а цвет фона совпадает с цветом фона кнопки. (Если цвет фона не указан, то Pygame пытается вывести шрифт с прозрачным фоном.)

Изображение текста выравнивается по центру кнопки, для чего создается объект `rect` изображения, а его атрибут `center` приводится в соответствие с одноименным атрибутом кнопки ❷.

Остается создать метод `draw_button()`, который может вызываться для отображения кнопки на экране:

button.py

```
def draw_button(self):  
    """Отображает пустую кнопку и выводит сообщение."""  
    self.screen.fill(self.button_color, self.rect)  
    self.screen.blit(self.msg_image, self.msg_image_rect)
```

Вызов метода `screen.fill()` рисует прямоугольную часть кнопки. Затем вызов `screen.blit()` выводит изображение текста на экран, передавая изображение и связанный с ним объект `rect`. Класс `Button` готов.

Вывод кнопки на экран

Мы будем использовать класс `Button` для создания кнопки `Play` в классе `AlienInvasion`. Сначала обновим операторы `import`:

alien_invasion.py

```
--пропуск--  
from game_stats import GameStats  
from button import Button
```

Нам нужна только одна кнопка `Play`, поэтому создадим ее в методе `__init__()` класса `AlienInvasion`. Этот код можно разместить в самом конце метода:

alien_invasion.py

```
def __init__(self):  
    --пропуск--  
    self.game_active = False  
  
    # Создание кнопки Play.  
    self.play_button = Button(self, "Play")
```

Программа создает экземпляр `Button` с текстом `Play`, но не выводит кнопку на экран. Чтобы она там появилась, мы вызовем метод `draw_button()` кнопки в `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    --nponyск--
    self.aliens.draw(self.screen)

    # Кнопка Play отображается в том случае, если игра неактивна.
    if not self.game_active:
        self.play_button.draw_button()

    pygame.display.flip()
```

Чтобы кнопка `Play` не закрывалась другими элементами экрана, мы отображаем ее после всех остальных игровых элементов, но перед переключением на новый экран. Код заключается в блок `if`, чтобы кнопка отображалась только в неактивном состоянии игры.

Теперь при запуске «Инопланетное вторжение» в центре экрана отображается кнопка `Play` (рис. 14.1).

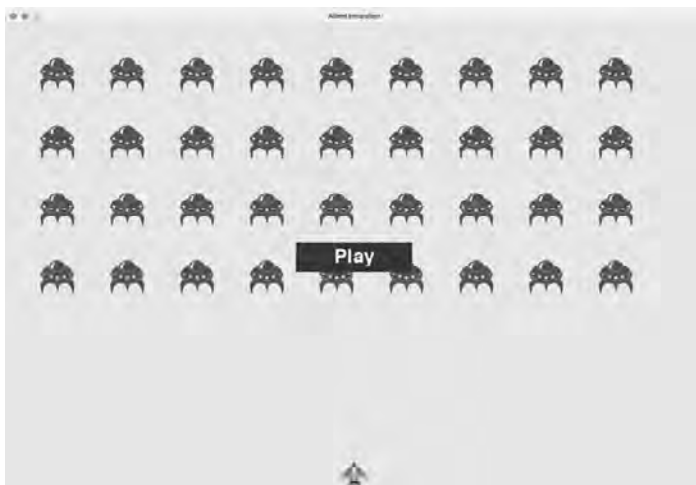


Рис. 14.1. Кнопка `Play` выводится, когда игра неактивна

Запуск игры

Чтобы при нажатии кнопки `Play` запускалась новая игра, добавьте в конец `_check_events()` следующий блок `elif` для отслеживания событий мыши над кнопкой:

alien_invasion.py

```
def _check_events(self):
    """Обрабатывает нажатия клавиш и события мыши."""
    for event in pygame.event.get():
```

```

if event.type == pygame.QUIT:
    --пропуск--
❶ elif event.type == pygame.MOUSEBUTTONDOWN:
❷     mouse_pos = pygame.mouse.get_pos()
❸     self._check_play_button(mouse_pos)

```

Pygame обнаруживает событие `MOUSEBUTTONDOWN`, когда игрок щелкает в любой точке экрана ❶, но мы хотим ограничить игру, чтобы она реагировала только на щелчки на кнопке `Play`. Для этого будет использоваться метод `pygame.mouse.get_pos()`, возвращающий кортеж с координатами x и y точки щелчка ❷. Эти значения передаются новому методу `_check_play_button()` ❸.

Ниже приведен код `_check_play_button()`, который я решил поместить после `_check_events()`:

alien_invasion.py

```

def _check_play_button(self, mouse_pos):
    """Запускает новую игру при нажатии кнопки Play."""
❶ if self.play_button.rect.collidepoint(mouse_pos):
    self.game_active = True

```

Метод `collidepoint()` используется для проверки того, находится ли точка щелчка в пределах области, определяемой прямоугольником кнопки `Play` ❶. Если да, то флаг `game_active` переводится в состояние `True`, и игра начинается!

К этому моменту вы сможете запустить игру и сыграть полноценную партию. После завершения игры значение `game_active` становится равным `False`, а кнопка `Play` снова появится на экране.

Сброс игры

Только что написанный нами код работает при первом нажатии кнопки `Play`, но не работает после завершения первой игры, поскольку условия, приводящие к окончанию игры, еще не были сброшены.

Чтобы игра сбрасывалась при каждом нажатии кнопки `Play`, необходимо сбросить игровую статистику, стереть данные о старых пришельцах и снарядах, создать новый флот и вернуть корабль в центр нижней стороны экрана:

alien_invasion.py

```

def _check_play_button(self, mouse_pos):
    """Запускает новую игру при нажатии кнопки Play."""
    if self.play_button.rect.collidepoint(mouse_pos):
❶         # Сброс игровой статистики.
        self.stats.reset_stats()
        self.game_active = True

        # Очистка групп aliens и bullets.
❷         self.bullets.empty()
        self.aliens.empty()

```

③

```
# Создание нового флота и размещение корабля в центре.
self._create_fleet()
self.ship.center_ship()
```

Игровая статистика сбрасывается ❶, вследствие чего игрок получает три новых корабля. После этого флаг `game_active` переводится в состояние `True` (чтобы игра началась сразу же после выполнения кода функции), группы `aliens` и `bullets` очищаются ❷, создается новый флот, а корабль выравнивается по центру ❸.

После этих изменений игра будет правильно переходить в исходное состояние при каждом нажатии `Play`, и вы сможете сыграть столько раз, сколько вам захочется!

Блокировка кнопки запуска игры

У кнопки `Play` в нашем приложении есть одна проблема: область кнопки на экране продолжает реагировать на щелчки, даже если сама кнопка не отображается. Если случайно щелкнуть на месте кнопки после начала игры, то она перезапустится!

Чтобы исправить этот недостаток, следует запускать игру только в том случае, если флаг `game_active` находится в состоянии `False`:

alien_invasion.py

❶

❷

```
def _check_play_button(self, mouse_pos):
    """Запускает новую игру при нажатии кнопки Play."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.game_active:
        # Сброс игровой статистики.
        self.stats.reset_stats()
        --пропуск--
```

Флаг `button_clicked` содержит значение `True` или `False` ❶, а игра перезапускается только в том случае, если пользователь нажал кнопку `Play` и *при этом* игра неактивна в настоящий момент ❷. Чтобы протестировать это поведение, запустите новую игру и многократно щелкайте в том месте, где должна находиться кнопка `Play`. Если все работает как положено, то нажатия кнопки не должны влиять на ход игры.

Скрытие указателя мыши

Указатель мыши должен быть видимым, чтобы пользователь мог начать игру, но после начала игры он только мешает. Чтобы исправить этот недостаток, мы скроем указатель мыши после того, как игра станет активной. Это можно сделать в блоке `if` в конце `_check_play_button()`:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    """Запускает новую игру при нажатии кнопки Play."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.game_active:
```

```
--пропуск--  
# Указатель мыши скрывается.  
pygame.mouse.set_visible(False)
```

Благодаря вызову `set_visible()` со значением `False` Pygame получает указание скрыть указатель, когда тот находится над окном игры.

После завершения игры указатель должен появляться снова, чтобы игрок мог нажать кнопку `Play` для запуска новой игры. Эту задачу решает следующий код:

alien_invasion.py

```
def _ship_hit(self):  
    """Обрабатывает столкновение корабля с пришельцем."""  
    if self.stats.ships_left > 0:  
        --пропуск--  
    else:  
        self.game_active = False  
        pygame.mouse.set_visible(True)
```

Указатель снова становится видимым сразу же после того, как игра становится неактивной, что происходит в `_ship_hit()`. Внимание к подобным деталям сделает вашу игру более профессиональной, а игрок сможет сосредоточиться на игре вместо того, чтобы разбираться в сложностях пользовательского интерфейса.

УПРАЖНЕНИЯ

14.1. Запуск игры клавишей Р. В «Инопланетном вторжении» игрок управляет кораблем с клавиатуры, поэтому для запуска игры тоже лучше использовать клавиатуру. Добавьте код, который позволит игроку запустить игру путем нажатия клавиши Р. Возможно, часть кода из `_check_play_button()` стоит переместить в функцию `start_game()`, которая будет вызываться из `_check_play_button()` и `_check_keydown_events()`.

14.2. Стрельба по мишени. Создайте у правого края экрана прямоугольник, который двигается вверх и вниз с постоянной скоростью. У левого края располагается корабль, который перемещается вверх и вниз игроком и стреляет по движущейся прямоугольной мишени. Добавьте кнопку `Play` для запуска игры. После трех промахов игра заканчивается, а на экране снова появляется кнопка `Play`. Нажатие этой кнопки перезапускает игру.

Повышение сложности

В текущей версии игры, после того как весь флот пришельцев будет уничтожен, игрок переходит на новый уровень, но сложность игры остается неизменной. Немного оживим игру и повысим ее сложность; для этого скорость игры будет повышаться каждый раз, когда игрок уничтожает весь флот.

Изменение настроек скорости

Начнем с реорганизации класса `Settings` и разделения настроек игры на две категории: постоянные и изменяющиеся. Необходимо также проследить за тем, чтобы настройки, изменяющиеся в ходе игры, сбрасывались в исходное состояние в начале новой игры. Метод `__init__()` из файла `settings.py` выглядит так:

settings.py

```
def __init__(self):
    """Инициализирует статические настройки игры."""
    # Настройки экрана
    self.screen_width = 1200
    self.screen_height = 800
    self.bg_color = (230, 230, 230)

    # Настройки корабля
    self.ship_limit = 3

    # Настройки снарядов
    self.bullet_width = 3
    self.bullet_height = 15
    self.bullet_color = 60, 60, 60
    self.bullets_allowed = 3

    # Настройки пришельцев
    self.fleet_drop_speed = 10

    # Темп ускорения игры
    self.speedup_scale = 1.1

    self.initialize_dynamic_settings()
```

Значения, которые остаются неизменными, по-прежнему инициализируются в методе `__init__()`. Добавляется настройка `speedup_scale` ❶, управляющая быстротой нарастания скорости; значение 2 будет удваивать скорость каждый раз, когда игрок переходит на следующий уровень, а значение 1 сохранит скорость постоянной. С таким значением, как 1.1, скорость будет увеличиваться в достаточной степени, чтобы игра усложнилась, но не стала невозможной. Наконец, вызов `initialize_dynamic_settings()` инициализирует значения атрибутов, которые должны изменяться в ходе игры ❷.

Код `initialize_dynamic_settings()` выглядит так:

settings.py

```
def initialize_dynamic_settings(self):
    """Инициализирует настройки, изменяющиеся в ходе игры."""
    self.ship_speed = 1.5
    self.bullet_speed = 2.5
    self.alien_speed = 1.0

    # fleet_direction = 1 обозначает движение вправо; а -1 — влево.
    self.fleet_direction = 1
```

Метод задает исходные значения скоростей корабля, снарядов и пришельцев. Эти скорости будут увеличиваться по ходу игры и сбрасываться каждый раз, когда игрок запускает новую игру. Мы добавляем в этот метод `fleet_direction`, чтобы пришельцы в начале новой игры всегда двигались вправо. Увеличивать значение `fleet_drop_speed` не нужно: когда пришельцы быстрее двигаются по горизонтали, они будут быстрее перемещаться и по вертикали.

Чтобы скорость корабля, снарядов и пришельцев увеличивалась каждый раз, когда игрок достигает нового уровня, мы напишем новый метод `increase_speed()`:

settings.py

```
def increase_speed(self):
    """Увеличивает настройки скорости."""
    self.ship_speed *= self.speedup_scale
    self.bullet_speed *= self.speedup_scale
    self.alien_speed *= self.speedup_scale
```

Чтобы увеличить скорость этих игровых элементов, мы умножаем каждую настройку скорости на значение `speedup_scale`.

Темп игры повышается путем вызова `increase_speed()` в `check_bullet_alien_collisions()` при уничтожении последнего пришельца во флоте, но перед созданием нового флота:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --пропуск--
    if not self.aliens:
        # Уничтожение снарядов, повышение скорости и создание нового флота.
        self.bullets.empty()
        self._create_fleet()
        self.settings.increase_speed()
```

Изменения значений настроек скорости `ship_speed`, `alien_speed` и `bullet_speed` достаточно для того, чтобы ускорить всю игру!

Сброс скорости

Каждый раз, когда игрок начинает новую игру, все измененные настройки должны вернуться к исходным значениям, иначе каждая новая игра будет начинаться с повышенными настройками скорости предыдущей игры:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    """Запускает новую игру при нажатии кнопки Play."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.game_active:
        # Сброс игровых настроек.
        self.settings.initialize_dynamic_settings()
    --пропуск--
```

Игра «Инопланетное вторжение» стала достаточно сложной и интересной. Каждый раз, когда игрок очищает экран, игра должна слегка ускориться, а ее сложность — слегка возрасти. Если сложность возрастает слишком быстро, то уменьшите значение `settings.speedup_scale`, а если, наоборот, недостаточная — слегка увеличьте это значение. Найдите оптимальное значение, оценивая сложность игры за разумный промежуток времени. Первые несколько появлений флотов должны быть простыми, несколько следующих — сложными, но уничтожимыми, а при последующих попытках сложность должна возрастать до пределов, когда уничтожить флот практически невозможно.

УПРАЖНЕНИЯ

14.3. Учебная стрельба с нарастающей сложностью. Начните с кода упражнения 14.2. Скорость мишени должна увеличиваться по ходу игры, а при нажатии игроком кнопки Play мишень должна возвращаться к исходной скорости.

14.4. Уровни сложности. Создайте в «Инопланетном вторжении» набор кнопок для выбора начальной сложности игры. Каждая кнопка должна присваивать атрибуту `Settings` значения, необходимые для создания различных уровней сложности.

Подсчет очков

Система подсчета очков позволит отслеживать счет игры в реальном времени; кроме того, на экране будут выводиться текущий рекорд, уровень и количество оставшихся кораблей.

Счет игры относится к игровой статистике, поэтому мы добавим атрибут `score` в класс `GameStats`:

game_stats.py

```
class GameStats:
    --ponyuk--
    def reset_stats(self):
        """Инициализирует статистику, изменяющуюся в ходе игры."""
        self.ships_left = self.settings.ship_limit
        self.score = 0
```

Чтобы счет сбрасывался при запуске новой игры, мы инициализируем `score` в `reset_stats()`, а не в методе `__init__()`.

Вывод счета

Чтобы вывести счет на экран, мы сначала создаем новый класс `Scoreboard`. Пока он ограничивается выводом текущего счета, но мы используем его для вывода рекордного счета, уровня и количества оставшихся кораблей. Ниже приведена первая часть класса; сохраните ее под именем `scoreboard.py`:

scoreboard.py

```
import pygame.font
```

```
class Scoreboard:
```

```
    """Класс для вывода игровой информации."""
```

- ```
❶ def __init__(self, ai_game):
 """Инициализирует атрибуты подсчета очков."""
 self.screen = ai_game.screen
 self.screen_rect = self.screen.get_rect()
 self.settings = ai_game.settings
 self.stats = ai_game.stats

 # Настройки шрифта для вывода счета.
❷ self.text_color = (30, 30, 30)
❸ self.font = pygame.font.SysFont(None, 48)

 # Подготовка исходного изображения счета.
❹ self.prep_score()
```

Класс `Scoreboard` выводит текст на экран, поэтому код начинается с импортирования модуля `pygame.font`. Затем методу `__init__()` передается параметр `ai_game` для обращения к объектам `settings`, `screen` и `stats`, чтобы класс мог выводить информацию об отслеживаемых показателях ❶. Далее назначается цвет текста ❷ и создается экземпляр объекта шрифта ❸.

Чтобы преобразовать выводимый текст в изображение, мы вызываем метод `prep_score()` ❹, который определяется следующим образом:

**scoreboard.py**

```
def prep_score(self):
```

```
 """Преобразует текущий счет в графическое изображение."""
```

- ```
❶ score_str = str(self.stats.score)
❷ self.score_image = self.font.render(score_str, True,
    self.text_color, self.settings.bg_color)

    # Вывод счета в правой верхней части экрана.
❸ self.score_rect = self.score_image.get_rect()
❹ self.score_rect.right = self.screen_rect.right - 20
❺ self.score_rect.top = 20
```

В методе `prep_score()` числовое значение `stats.score` преобразуется в строку ❶; эта строка передается методу `render()`, создающему изображение ❷. Чтобы счет был хорошо виден на экране, `render()` передаются цвет фона и цвет текста.

Счет размещается в правой верхней части экрана и расширяется влево по мере увеличения значения и ширины числа. Чтобы счет всегда оставался выровненным по правой стороне, мы создаем прямоугольник `rect` с именем `score_rect` ❸ и смещаем его правую сторону на 20 пикселей от правого края экрана ❹. Затем верхняя сторона прямоугольника смещается на 20 пикселей вниз от верхнего края экрана ❺.

Остается создать метод `show_score()` для вывода созданного графического изображения:

scoreboard.py

```
def show_score(self):
    """Выводит счет на экран."""
    self.screen.blit(self.score_image, self.score_rect)
```

Метод выводит счет на экран в позиции, определяемой `score_rect`.

Создание экземпляра *Scoreboard*

Чтобы вывести счет, мы создадим в классе `AlienInvasion` экземпляра `Scoreboard`. Начнем с изменения операторов `import`:

alien_invasion.py

```
--пропуск--
from game_stats import GameStats
from scoreboard import Scoreboard
--пропуск--
```

Затем создадим экземпляр `Scoreboard` в методе `__init__()`:

alien_invasion.py

```
def __init__(self):
    --пропуск--
    pygame.display.set_caption("Alien Invasion")

    # Создание экземпляров для хранения статистики и панели результатов.
    self.stats = GameStats(self)
    self.sb = Scoreboard(self)
    --пропуск--
```

Затем мы выводим панель результатов на экран с помощью функции `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    --пропуск--
    self.aliens.draw(self.screen)

    # Вывод информации о счете.
    self.sb.show_score()

    # Кнопка Play отображается в том случае, если игра неактивна.
    --пропуск--
```

Метод `show_score()` вызывается непосредственно перед отображением кнопки `Play`.

Если запустить игру сейчас, то в правом верхнем углу экрана отображается счет 0. (Пока мы просто хотим убедиться в том, что счет отображается в нужном месте, прежде чем заниматься дальнейшей доработкой системы подсчета очков.) На рис. 14.2 изображено окно игры перед ее началом.



Рис. 14.2. Счет отображается в правом верхнем углу экрана

А теперь нужно организовать начисление очков за каждого пришельца!

Обновление счета при уничтожении пришельцев

Чтобы на экране выводился актуальный счет, мы будем обновлять значение `stats.score` при каждом попадании в пришельца, а затем вызывать `prep_score()` для обновления изображения счета. Но сначала нужно определить, сколько очков игрок будет получать за каждого пришельца:

settings.py

```
def initialize_dynamic_settings(self):
    --пропуск--

    # Подсчет очков
    self.alien_points = 50
```

Стоимость каждого пришельца в очках будет увеличиваться по ходу игры. Чтобы значение сбрасывалось в начале каждой новой игры, мы задаем значение в `initialize_dynamic_settings()`.

Счет за каждого сбитого пришельца будет обновляться с помощью функции `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    """Обрабатывает столкновение снарядов с пришельцами."""

    # Удаление снарядов и пришельцев, участвующих в коллизиях.
    collisions = pygame.sprite.groupcollide(self.bullets, self.aliens, True, True)
```

```
if collisions:
    self.stats.score += self.settings.alien_points
    self.sb.prep_score()
--пропуск--
```

При попадании снаряда в пришельца Pygame возвращает словарь `collisions`. Программа проверяет, существует ли словарь, и если да — стоимость пришельца добавляется к счету. Затем вызов `prep_score()` создает новое изображение для обновленного счета.

Теперь во время игры вы сможете набирать очки!

Сброс счета

В текущей версии игры счет обновляется только *после* попадания в пришельца; как правило, такой подход работает нормально. Но старый счет выводится и после попадания в первого пришельца в новой игре.

Проблема решается путем инициализации счета при создании новой игры:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    --пропуск--
    if button_clicked and not self.game_active:
        --пропуск--
        # Сброс игровой статистики.
        self.stats.reset_stats()
        self.sb.prep_score()
        --пропуск--
```

Метод `prep_score()` вызывается при сбросе игровой статистики в начале новой игры. Счет, выводимый на экран, обнуляется.

Начисление очков за все попадания

В нынешней версии код при подсчете очков будет пропускать некоторых пришельцев. Например, если два снаряда попадают в пришельцев во время одного прохода цикла или если будет создан широкий снаряд для поражения нескольких пришельцев одновременно, то игрок получит очки только за одного подстреленного пришельца. Чтобы устранить этот недостаток, нужно доработать механизм обнаружения коллизий между снарядами и пришельцами.

В коде `_check_bullet_alien_collisions()` любой снаряд, столкнувшийся с пришельцем, становится ключом словаря `collisions`. С каждым снарядом связывается значение — список пришельцев, участвующих в коллизии. Переберем словарь `collisions` и убедимся в том, что очки начисляются за каждого подбитого пришельца:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --пропуск--
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
    --пропуск--
```

Если словарь `collisions` был определен, то программа перебирает все значения в нем. Вспомните, что каждое значение представляет собой список пришельцев, в которых попал один снаряд. Стоимость каждого пришельца умножается на количество пришельцев в списке, а результат прибавляется к текущему счету. Чтобы протестировать эту систему, увеличьте ширину снаряда до 300 пикселей и убедитесь в том, что игра начисляет очки за каждого пришельца, в которого попал этот большой снаряд; затем верните ширину снаряда к нормальному значению.

Увеличение награды за пришельцев

Так как по мере достижения каждого нового уровня игра усложняется, за пришельцев на этих уровнях следует давать больше очков. Чтобы реализовать эту функциональность, мы добавим код, увеличивающий стоимость пришельцев при возрастании скорости игры:

settings.py

```
class Settings:
    """Класс для хранения всех настроек игры "Инопланетное вторжение"."""

    def __init__(self):
        --пропуск--
        # Темп ускорения игры.
        self.speedup_scale = 1.1
        # Темп роста стоимости пришельцев.
        ❶ self.score_scale = 1.5

        self.initialize_dynamic_settings()

    def initialize_dynamic_settings(self):
        --пропуск--

    def increase_speed(self):
        """Увеличивает настройки скорости и стоимость пришельцев."""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= self.speedup_scale

        ❷ self.alien_points = int(self.alien_points * self.score_scale)
```

В программе определяется коэффициент прироста начисляемых очков; он называется `score_scale` ❶. При небольшом увеличении скорости (1.1) игра быстро

усложняется, но чтобы увидеть заметную разницу в очках, необходимо изменять стоимость пришельцев на бóльшую величину (1.5). После увеличения скорости игры стоимость каждого попадания также увеличивается ❷. Чтобы счет возрастал на целое количество очков, в программе используется функция `int()`.

Чтобы увидеть стоимость каждого пришельца, добавьте в метод `increase_speed()` в классе `Settings` функцию `print()`:

settings.py

```
def increase_speed(self):
    --пропуск--
    self.alien_points = int(self.alien_points * self.score_scale)
    print(self.alien_points)
```

Новое значение должно выводиться в терминальном окне каждый раз, когда игрок переходит на новый уровень.

ПРИМЕЧАНИЕ

Убедившись, что стоимость пришельцев действительно возрастает, не забудьте удалить вызов функции `print()`; в противном случае лишний вывод повлияет на быстроедействие игры и будет отвлекать игрока.

Округление счета

В большинстве аркадных шутеров счет ведется значениями, кратными 10, и мы воспользуемся этой схемой в своей игре. Отформатируем счет так, чтобы в больших числах группы разрядов разделялись запятыми. Изменения вносятся в классе `Scoreboard`:

scoreboard.py

```
def prep_score(self):
    """Преобразует текущий счет в графическое изображение."""
    rounded_score = round(self.stats.score, -1)
    score_str = f"{rounded_score:,"}
    self.score_image = self.font.render(score_str, True,
                                         self.text_color, self.settings.bg_color)
    --пропуск--
```

Функция `round()` обычно округляет дробное число до заданного количества знаков, переданного во втором аргументе. Но если во втором аргументе передается отрицательное число, то `round()` округляет значение до ближайших десятков, сотен, тысяч и т. д. Код дает Python указание округлить значение `stats.score` до десятков и сохранить его в `rounded_score`.

Затем мы используем для счета спецификатор формата в f-строке. *Спецификатор формата* (format specifier) — это специальным образом составленная последовательность символов, определяющая способ представления значения переменной. В данном случае благодаря последовательности `:`, Python получает указание

вставить запятые в соответствующие места в предоставленном числовом значении. Теперь, запуская игру, вы увидите отформатированный, округленный счет, даже если набрали много очков (рис. 14.3).

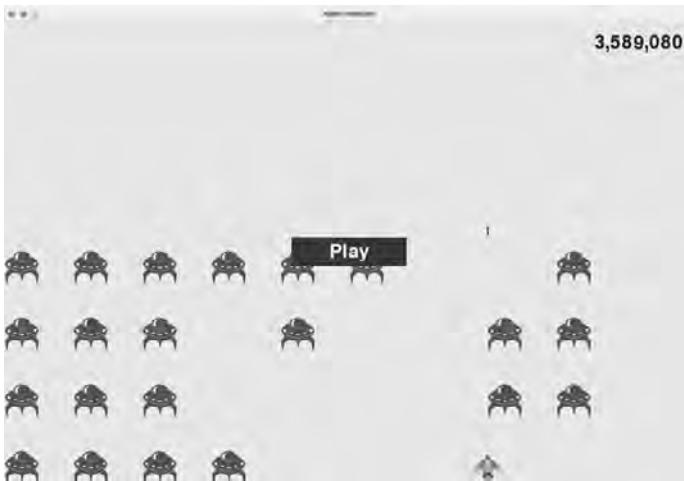


Рис. 14.3. Округленный счет с разделителями групп

Рекорды

Каждый игрок желает превзойти предыдущий рекорд, поэтому мы будем отслеживать и выводить рекорды, чтобы игроку было к чему стремиться. Рекорды будут храниться в классе `GameStats`:

game_stats.py

```
def __init__(self, ai_game):
    --пропуск--
    # Рекорд не должен сбрасываться.
    self.high_score = 0
```

Так как рекорд не должен сбрасываться при повторном запуске, значение `high_score` инициализируется в `__init__()`, а не в `reset_stats()`.

Теперь изменим класс `Scoreboard` для отображения рекорда. Начнем с метода `__init__()`:

scoreboard.py

```
def __init__(self, ai_game):
    --пропуск--
    # Подготовка изображений счетов.
    self.prep_score()
    self.prep_high_score()
```

Рекорд должен отображаться отдельно от текущего счета, поэтому для подготовки его изображения понадобится новый метод `prep_high_score()` ❶:

scoreboard.py

```
def prep_high_score(self):
    """Преобразует рекордный счет в графическое изображение."""
    ❶ high_score = round(self.stats.high_score, -1)
    high_score_str = f"{high_score:}"
    ❷ self.high_score_image = self.font.render(high_score_str, True,
        self.text_color, self.settings.bg_color)

    # Рекорд выравнивается по центру верхней стороны экрана.
    self.high_score_rect = self.high_score_image.get_rect()
    ❸ self.high_score_rect.centerx = self.screen_rect.centerx
    ❹ self.high_score_rect.top = self.score_rect.top
```

Рекорд округляется до десятков и форматируется с помощью запятой ❶. Затем для рекорда создается графическое изображение ❷, выполняется горизонтальное выравнивание прямоугольника по центру экрана ❸, а атрибут `top` прямоугольника приводится в соответствие с верхней стороной изображения счета ❹.

Теперь метод `show_score()` выводит текущий счет в правом верхнем углу, а рекорд — в центре верхней стороны экрана:

scoreboard.py

```
def show_score(self):
    """Выводит счет на экран."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
```

Для обновления рекорда в класс `Scoreboard` добавляется новая функция `check_high_score()`:

scoreboard.py

```
def check_high_score(self):
    """Проверяет, появился ли новый рекорд."""
    if self.stats.score > self.stats.high_score:
        self.stats.high_score = self.stats.score
        self.prep_high_score()
```

Метод `check_high_score()` сравнивает текущий счет с рекордом. Если текущий счет выше, то мы обновляем значение `high_score` и вызываем `prep_high_score()`, чтобы обновить изображение рекорда.

Метод `check_high_score()` должен вызываться при каждом попадании в пришельца после обновления счета в `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def check_bullet_alien_collisions(self):
    --пропуск--
    if collisions:
```



```

for aliens in collisions.values():
    self.stats.score += self.settings.alien_points * len(aliens)
self.sb.prep_score()
self.check_high_score()
--пропуск--

```

Метод `check_high_score()` должен вызываться только в том случае, если словарь `collisions` присутствует, причем вызов выполняется после обновления счета для всех подбитых пришельцев.

Когда вы играете в «Инопланетное вторжение» впервые, текущий счет одновременно является рекордом, поэтому будет отображаться и как текущий счет, и как рекорд. Но в начале второй игры ваш предыдущий рекорд должен отображаться в середине, а текущий счет — справа (рис. 14.4).



Рис. 14.4. Рекордный счет выводится сверху экрана по центру

Отображение уровня

Чтобы в игре выводился текущий уровень, сначала в класс `GameStats` следует добавить атрибут для его представления. Чтобы уровень сбрасывался в начале каждой игры, инициализируйте его в `reset_stats()`:

game_stats.py

```

def reset_stats(self):
    """Инициализирует статистику, изменяющуюся в ходе игры."""
    self.ships_left = self.settings.ship_limit
    self.score = 0
    self.level = 1

```

Чтобы класс `Scoreboard` выводил текущий уровень, мы вызываем новый метод `prep_level()` из метода `__init__()`:

scoreboard.py

```
def __init__(self, ai_game):
    --пропуск--
    self.prep_high_score()
    self.prep_level()
```

Метод `prep_level()` выглядит так:

scoreboard.py

```
def prep_level(self):
    """Преобразует уровень в графическое изображение."""
    level_str = str(self.stats.level)
    ❶ self.level_image = self.font.render(level_str, True,
        self.text_color, self.settings.bg_color)

    # Уровень выводится под текущим счетом.
    self.level_rect = self.level_image.get_rect()
    ❷ self.level_rect.right = self.score_rect.right
    ❸ self.level_rect.top = self.score_rect.bottom + 10
```

Метод `prep_level()` создает изображение на базе значения, хранящегося в `stats.level` ❶, и приводит атрибут `right` изображения в соответствие с атрибутом `right` счета ❷. Затем атрибут `top` сдвигается на 10 пикселей ниже нижнего края изображения текущего счета, чтобы между счетом и уровнем оставался пустой интервал ❸.

В метод `show_score()` также необходимо внести изменения:

scoreboard.py

```
def show_score(self):
    """Выводит текущий счет, рекорд и количество оставшихся кораблей."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
    self.screen.blit(self.level_image, self.level_rect)
```

Добавленная строка выводит на экран изображение, представляющее уровень.

Увеличение `stats.level` и обновление изображения уровня выполняются в `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --пропуск--
    if not self.aliens:
        # Уничтожить существующие снаряды и создать новый флот.
        self.bullets.empty()
        self.create_fleet()
        self.settings.increase_speed()

    # Увеличение уровня.
    self.stats.level += 1
    self.sb.prep_level()
```

Если все пришельцы уничтожены, то программа увеличивает значение `stats.level` и вызывает `prep_level()` для обновления уровня.

Чтобы убедиться в том, что изображения текущего счета и уровня правильно обновляются в начале новой игры, мы вызываем `prep_level()` при нажатии кнопки Play:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    --пропуск--
    if button_clicked and not self.game_active:
        --пропуск--
        self.sb.prep_score()
        self.sb.prep_level()
        --пропуск--
```

Метод `prep_level()` вызывается сразу же после вызова `prep_score()`.

Теперь количество пройденных уровней отображается на экране (рис. 14.5).

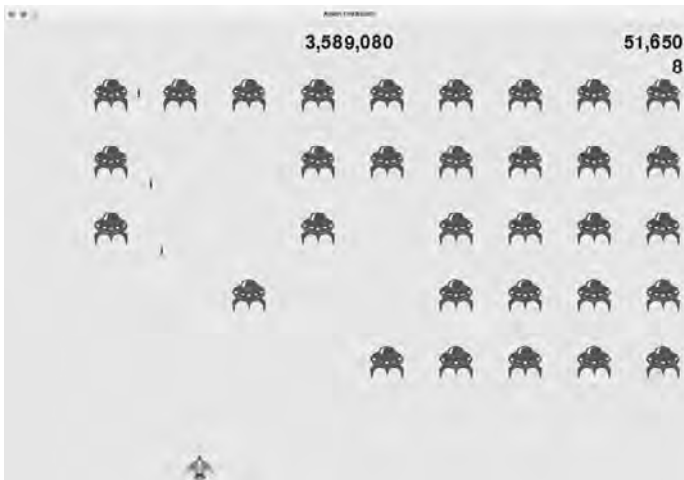


Рис. 14.5. Текущий уровень выводится под текущим счетом

ПРИМЕЧАНИЕ

В некоторых классических играх выводимая информация снабжается текстовыми метками: «Уровень», «Рекорд» и т. д. Мы их опустили, поскольку смысл этих чисел понятен каждому, кто сыграл в «Инопланетное вторжение». Если вы хотите видеть эти метки, то добавьте их в строки непосредственно перед вызовами метода `font.render()` в классе `Scoreboard`.

Отображение количества кораблей

Остается вывести количество кораблей, оставшихся у игрока, однако на этот раз информация будет выводиться в графическом виде. Как во многих классических аркадных играх, в левом верхнем углу экрана программа рисует несколько изображений корабля. Каждый корабль обозначает одну оставшуюся попытку.

Для начала нужно сделать так, чтобы класс `Ship` наследовал от `Sprite`, — это необходимо для создания группы кораблей:

ship.py

```
import pygame
from pygame.sprite import Sprite
```

```
❶ class Ship(Sprite):
    # Класс для управления кораблем.

    def __init__(self, ai_game):
        """Инициализирует корабль и задает его начальную позицию."""
❷    super().__init__()
    --пропуск--
```

Здесь мы импортируем `Sprite`, объявляем о наследовании `Ship` от `Sprite` ❶ и вызываем `super()` в начале метода `__init__()` ❷.

Далее необходимо изменить класс `Scoreboard` и создать группу кораблей для вывода на экран. Операторы `import` выглядят так:

scoreboard.py

```
import pygame.font
from pygame.sprite import Group

from ship import Ship
```

Мы собираемся создать группу кораблей, поэтому программа импортирует классы `Group` и `Ship`.

Метод `__init__()` выглядит так:

scoreboard.py

```
def __init__(self, ai_game):
    """Инициализирует атрибуты подсчета очков."""
    self.ai_game = ai_game
    self.screen = ai_game.screen
    --пропуск--
    self.prep_level()
    self.prep_ships()
```

Экземпляр игры присваивается атрибуту, так как он понадобится нам для создания кораблей. Метод `prep_ships()` будет вызываться после `prep_level()` и выглядит так:

scoreboard.py

```
def prep_ships(self):
    """Сообщает количество оставшихся кораблей."""
    ❶ self.ships = Group()
    ❷ for ship_number in range(self.stats.ships_left):
        ship = Ship(self.ai_game)
        ❸ ship.rect.x = 10 + ship_number * ship.rect.width
        ❹ ship.rect.y = 10
        ❺ self.ships.add(ship)
```

Метод `prep_ships()` создает пустую группу `self.ships` для хранения экземпляров кораблей ❶. В ходе ее заполнения цикл выполняется по одному разу для каждого корабля, оставшегося у игрока ❷. В цикле создается новый корабль, а его координата x задается так, чтобы корабли размещались рядом друг с другом, разделенные интервалами по 10 пикселей ❸. Координата y задается так, чтобы корабли были смещены на 10 пикселей от верхнего края экрана и выровнены по изображению текущего счета ❹. Наконец, каждый корабль добавляется в группу `ships` ❺.

Осталось вывести корабли на экран:

scoreboard.py

```
def show_score(self):
    """Выводит счета, уровень и количество кораблей на экран."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
    self.screen.blit(self.level_image, self.level_rect)
    self.ships.draw(self.screen)
```

При выводе кораблей на экран мы вызываем метод `draw()` для группы, а Pygame рисует каждый отдельный корабль.

Чтобы игрок видел, сколько попыток у него в начале игры, мы вызываем `prep_ships()` при запуске новой игры. Это происходит в функции `_check_play_button()` в классе `AlienInvasion`:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    --пропуск--
    if button_clicked and not self.game_active:
        --пропуск--
        self.sb.prep_level()
        self.sb.prep_ships()
        --пропуск--
```

Метод `prep_ships()` также вызывается при столкновении пришельца с кораблем, чтобы изображение обновлялось при потере корабля:

alien_invasion.py

```
def _ship_hit(self):
    """Обработывает столкновение корабля с пришельцем."""
    if self.stats.ships_left > 0:
        # Уменьшение ships_left и обновление панели счета.
        self.stats.ships_left -= 1
        self.sb.prep_ships()
        --пропуск--
```

Метод `prep_ships()` вызывается после уменьшения значения `ships_left`, так что при каждой потере корабля выводится правильное количество изображений.

На рис. 14.6 показана готовая система подсчета очков, при этом количество оставшихся кораблей отображается в левой верхней части экрана.

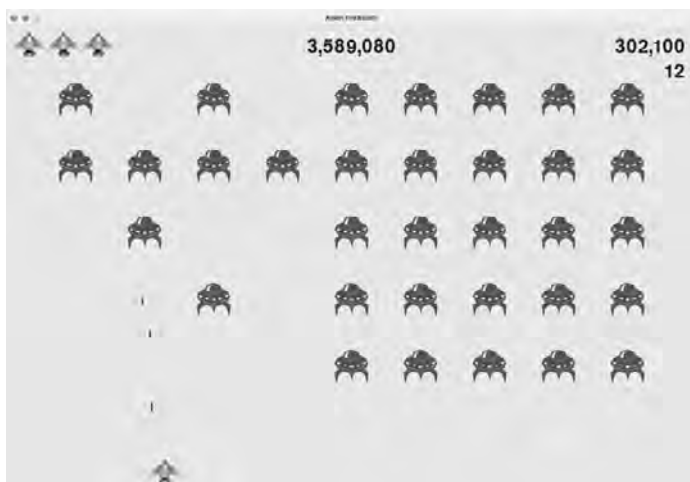


Рис. 14.6. Готовая система подсчета очков в игре «Инопланетное вторжение»

УПРАЖНЕНИЯ

14.5. Исторический рекорд. В текущей версии рекорд сбрасывается каждый раз, когда игрок закрывает игру и перезапускает ее. Чтобы этого не происходило, запишите рекорд в файл перед вызовом `sys.exit()` и загрузите его при инициализации значения в классе `GameStats`.

14.6. Рефакторинг. Найдите функции и методы, которые решают несколько задач, и проведите рефакторинг, улучшающий структуру и эффективность кода. Например,

переместите часть кода функции `_check_bullet_alien_collisions()`, которая запускает новый уровень при уничтожении флота, в функцию `start_new_level()`. Переместите также четыре метода, вызываемых в методе `__init__()` класса `Scoreboard`, в метод `prep_images()` для сокращения длины метода `__init__()`. Метод `prep_images()` также может быть полезным для `_check_play_button()` или `start_game()`, если вы уже провели рефакторинг `_check_play_button()`.

ПРИМЕЧАНИЕ

Прежде чем браться за рефакторинг проекта, обратитесь к приложению Г. В нем рассказано, как восстановить рабочее состояние проекта, если в ходе рефакторинга были допущены ошибки.

14.7. Расширение игры. Подумайте над возможными расширениями. Например, пришельцы тоже могут стрелять по кораблю, или же вы можете добавить укрытия, за которыми может скрываться корабль (укрытия могут разрушаться снарядами, выпускаемыми с обеих сторон). Или добавьте звуковые эффекты (например, взрывы или звуки выстрелов), используя средства модуля `pygame.mixer`.

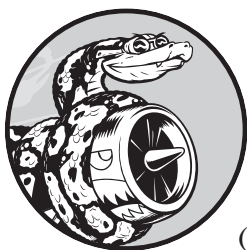
14.8. Боковая стрельба, финальная версия. Продолжайте разрабатывать приложение с боковой стрельбой, используя все, чему научились в этом проекте. Добавьте кнопку `Play`, обеспечьте ускорение игры в нужных местах и разработайте систему начисления очков. Не забывайте проводить рефакторинг в процессе работы и ищите возможности добавить такие настройки игры, которые не были показаны в этой главе.

Резюме

В этой главе вы узнали, как создать кнопку `Play` для запуска новой игры, обнаруживать события мыши и скрывать указатель мыши в активных играх. Полученные знания помогут вам создать другие кнопки в играх — например, кнопку `Help` для вывода инструкций. Кроме того, вы научились изменять скорость по ходу игры, создавать прогрессивную систему подсчета очков и выводить информацию в текстовом и графическом виде.

15

Генерирование данных



Под *визуализацией данных* (data visualization) понимается исследование закономерностей в наборах данных через их визуальные представления. Визуализация тесно связана с *анализом данных* (data analysis), использующим программный код для изучения закономерностей и связей в наборе данных. Набором может быть как маленький список чисел, помещающийся в одной строке кода, так и массив из терабайтов данных, содержащих множество различных видов информации.

Качественное представление данных не сводится к красивой картинке. Если для набора данных подобрано простое, визуально привлекательное представление, то его смысл становится очевидным для зрителя. Люди замечают в наборе данных закономерности, о которых и не подозревали.

К счастью, для визуализации сложных данных не нужен суперкомпьютер. Благодаря эффективности Python вы сможете быстро исследовать наборы данных из миллионов отдельных *элементов данных* (точек данных, data points) на обычном ноутбуке. Элементы данных даже не обязаны быть числовыми. Приемы, о которых вы узнали в первой части книги, позволят вам проанализировать даже нечисловые данные.

Python используется для работы с большими объемами данных в генетике, исследовании климата, политическом и экономическом анализе и множестве других областей. Специалисты по обработке данных создали на Python впечатляющий инструментарий визуализации и анализа, и многие из этих разработок доступны и для вас. Один из самых популярных инструментов такого рода — Matplotlib, математическая библиотека создания диаграмм. В этой главе с ее помощью мы будем добавлять простые диаграммы, графики, диаграммы разброса данных и т. д. А затем перейдем к созданию более интересного набора данных, основанного на концепции случайного блуждания — визуализации, генерируемой на основе серии случайных решений.

Кроме того, в этом проекте для исследования закономерностей различных бросков кубиков будет использоваться пакет Plotly, ориентированный на создание визуализаций, адаптирующихся под различные устройства вывода. Plotly генерирует визуализации, автоматически масштабируемые по размерам экранов различных цифровых устройств. Визуализации также могут содержать различные интерактивные возможности — например, выделение разных аспектов данных набора данных при наведении указателя мыши на те или иные части визуализации. Инструменты Matplotlib и Plotly помогут нам визуализировать наиболее интересные виды данных.

Установка Matplotlib

Чтобы использовать библиотеку Matplotlib для исходных визуализаций, необходимо установить ее с помощью модуля `pip`, как мы делали с модулем `pytest` в главе 11 (см. раздел «Установка `pytest` с помощью `pip`»). Введите следующую команду в приглашении терминала:

```
$ python -m pip install --user matplotlib
```

Если для запуска программ или терминального сеанса вы вместо `python` используете другую команду (например, `python3`), то ваша команда будет выглядеть так:

```
$ python3 -m pip install --user matplotlib
```

Чтобы получить представление о визуализациях, которые можно создать с помощью средств Matplotlib, посетите главную страницу библиотеки по адресу <https://matplotlib.org/> и перейдите в раздел **Plot types** (Типы графиков). Щелкая на визуализации в галерее, вы сможете просмотреть код, использованный для ее создания.

Создание простого графика

Начнем с создания простого линейного графика, а затем настроим его для более содержательной визуализации данных. В качестве данных для графика будет использоваться последовательность квадратов 1, 4, 9, 16 и 25.

Передайте Matplotlib числа так, как показано ниже, а библиотека сделает все остальное:

mpl_squares.py

```
import matplotlib.pyplot as plt
```

```
squares = [1, 4, 9, 16, 25]
```

```
❶ fig, ax = plt.subplots()
```

```
ax.plot(squares)
```

```
plt.show()
```

Сначала импортируйте модуль `pyplot` с псевдонимом `plt`, чтобы вам не приходилось многократно вводить слово `pyplot`. (Это сокращение часто встречается в онлайн-примерах, поэтому мы поступим так же.) Модуль `pyplot` содержит ряд функций для рисования диаграмм и графиков.

Мы создаем список `squares` для хранения данных, которые будем отображать на графике. Затем используем еще одно общепринятое правило Matplotlib — вызов функции `subplots()` ❶. Она позволяет сгенерировать одну или несколько поддиаграмм на одном рисунке. Переменная `fig` обозначает весь *рисунок*, который представляет собой набор генерируемых диаграмм. Переменная `ax` соответствует одной диаграмме на рисунке; это переменная, которую мы будем использовать большую часть времени при определении и настройке отдельного графика.

Затем вызывается функция `plot()`, которая пытается создать осмысленное графическое представление для заданных чисел. Вызов `plt.show()` открывает окно просмотра Matplotlib и выводит график (рис. 15.1). В окне просмотра можно изменять масштаб и перемещаться по созданному графику, а кнопка с дискетой позволяет сохранить любое изображение по вашему выбору.

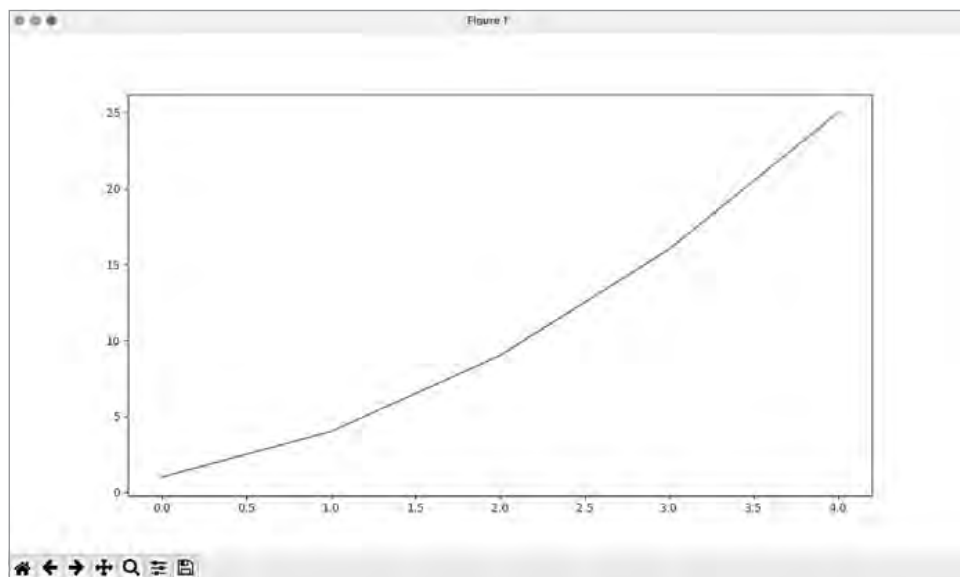


Рис. 15.1. Пример простейшего графика в Matplotlib

Изменение типа надписей и толщины графика

Хотя из графика на рис. 15.1 видно, что числовая последовательность возрастает, текст надписей слишком мелкий, а линия слишком тонкая. К счастью, Matplotlib позволяет настроить практически каждый аспект визуализации.

Мы используем эти возможности настройки для того, чтобы улучшить читабельность графика. Начнем с добавления заголовка и маркировки осей:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
❶ ax.plot(squares, linewidth=3)

# Задание заголовка диаграммы и меток осей.
❷ ax.set_title("Square Numbers", fontsize=24)
❸ ax.set_xlabel("Value", fontsize=14)
  ax.set_ylabel("Square of Value", fontsize=14)

# Задание размера шрифта делений на осях.
❹ ax.tick_params(labelsize=14)

plt.show()
```

Параметр `linewidth` управляет толщиной линии, которая создается вызовом `plot()`

❶. Существует множество способов изменить созданную диаграмму перед представлением. Метод `set_title()` устанавливает заголовок диаграммы ❷. Параметры `fontsize`, неоднократно встречающиеся в коде, управляют размером текста различных элементов диаграммы.

Методы `xlabel()` и `ylabel()` позволяют установить метки (заголовки) каждой из осей ❸, а функция `tick_params()` определяет оформление делений на осях ❹. Аргументы, использованные в данном примере, устанавливают для меток делений размер шрифта 14 (`labelsize=14`).

Как видно из рис. 15.2, график выглядит гораздо лучше. Текст надписей стал крупнее, а линия графика — толще. Часто стоит поэкспериментировать с этими значениями, чтобы получить представление о том, какой вариант оформления будет лучше смотреться на полученной диаграмме.

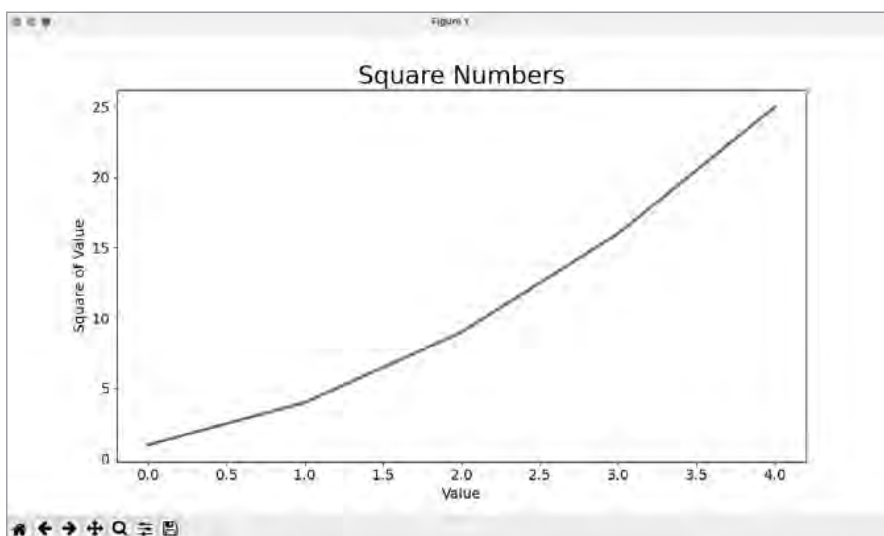


Рис. 15.2. График выглядит гораздо лучше

Корректировка графика

Теперь, когда текст на графике стало легче читать, мы видим, что данные помечены неправильно. Обратите внимание: для точки 4,0 в конце графика указан квадрат 25! Исправим эту проблему.

Если `plot()` передается числовая последовательность, то функция считает, что первый элемент данных соответствует координате x со значением 0, однако в нашем примере первая точка соответствует значению 1. Чтобы переопределить значение по умолчанию, передайте `plot()` как входные значения, так и квадраты:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(input_values, squares, linewidth=3)

# Задание заголовка диаграммы и меток осей.
--пропуск--
```

Теперь функции `plot()` не нужно предполагать, как был сгенерирован выходной набор чисел. На рис. 15.3 изображен правильный график.

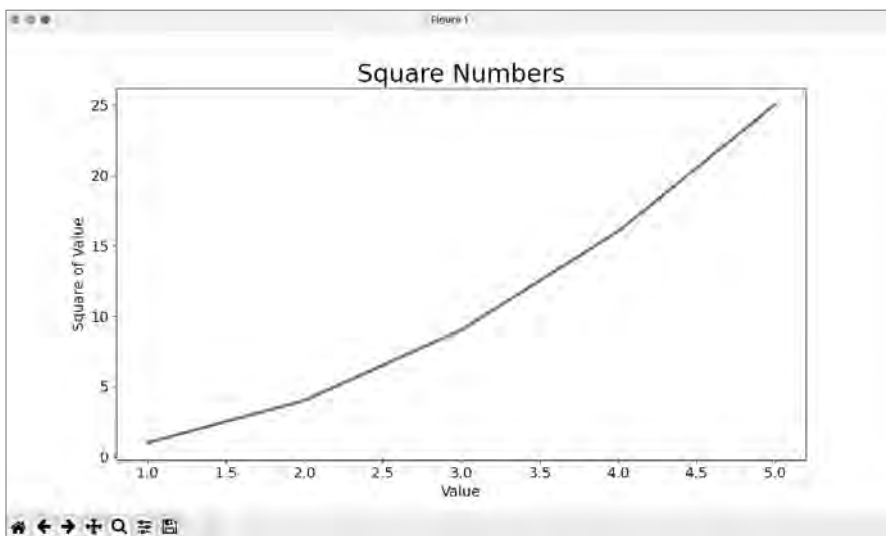


Рис. 15.3. График с правильными данными

При вызове `plot()` можно передавать многочисленные аргументы, а также использовать различные функции для настройки графиков. С этими функциями вы познакомитесь поближе позднее, когда мы начнем работать с более интересными наборами данных в этой главе.

Встроенные стили

В Matplotlib существует целый ряд заранее определенных стилей оформления с хорошей подборкой настроек для цветов фона, линий сетки, толщин линий, шрифтов, размеров шрифтов и т. д.; готовые настройки позволят вам создавать привлекательные диаграммы, не тратя на это много времени. Чтобы узнать, какие стили доступны в вашей системе, выполните следующие команды в терминальном сеансе:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery',
--пропуск--
```

Чтобы использовать эти стили, добавьте одну строку кода перед вызовом функции `subplots()`:

mpl_squares.py

```
import matplotlib.pyplot as plt
```

```
input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]
```

```
plt.style.use('seaborn')
fig, ax = plt.subplots()
--пропуск--
```

Этот код создает график, изображенный на рис. 15.4. Существует множество разнообразных стилей; поэкспериментируйте и найдите те, которые вам больше нравятся.

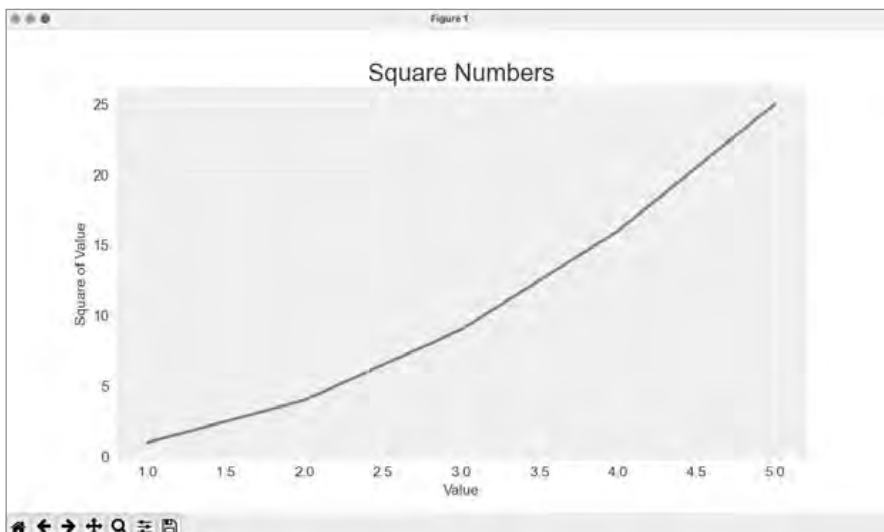


Рис. 15.4. Встроенный стиль seaborn

Нанесение и оформление отдельных точек с помощью функции `scatter()`

Иногда бывает полезно нанести на график отдельные точки, основанные на некоторых характеристиках, и определить их оформление. Например, на графике малые и большие значения могут отображаться разными цветами. Возможны и другие варианты: допустим, сначала нанести множество точек с одним типом оформления, а затем выделить отдельные точки набора, перерисовав их с другим оформлением.

Для нанесения на диаграмму отдельной точки используется функция `scatter()`. Передайте ей координаты (x, y) нужной точки, и функция нанесет эти значения на диаграмму:

`scatter_squares.py`

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
```

```
fig, ax = plt.subplots()
ax.scatter(2, 4)
```

```
plt.show()
```

Применим оформление, чтобы результат выглядел более интересно. Мы добавим название, метки осей, а также увеличим шрифт, чтобы текст читался легче:

```
import matplotlib.pyplot as plt
```

```
plt.style.use('seaborn')
```

```
fig, ax = plt.subplots()
```

```
❶ ax.scatter(2, 4, s=200)
```

```
# Задание заголовка диаграммы и меток осей.
```

```
ax.set_title("Square Numbers", fontsize=24)
```

```
ax.set_xlabel("Value", fontsize=14)
```

```
ax.set_ylabel("Square of Value", fontsize=14)
```

```
# Задание размера шрифта делений на осях.
```

```
ax.tick_params(labelsize=14)
```

```
plt.show()
```

Вызывается функция `scatter()`; аргумент `s` задает размер точек, используемых для рисования диаграммы ❶. Если запустить программу `scatter_squares.py` в текущем состоянии, то вы увидите одну точку в середине диаграммы (рис. 15.5).

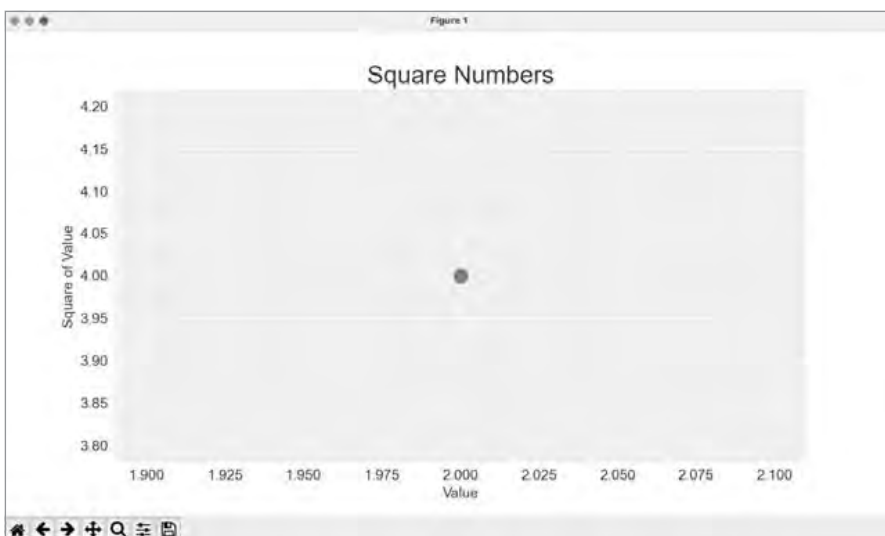


Рис. 15.5. Вывод одной точки

Вывод серии точек с помощью функции `scatter()`

Чтобы вывести на диаграмме серию точек, передайте `scatter()` списки значений координат x и y :

`scatter_squares.py`

```
import matplotlib.pyplot as plt

x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=100)

# Задание заголовка диаграммы и меток осей.
--пропуск--
```

В списке `x_values` содержатся числа, возводимые в квадрат, а в `y_values` — сами квадраты. При передаче этих списков функции `scatter()` библиотека Matplotlib считывает по одному значению из каждого списка и наносит их на диаграмму как точку. Таким образом, на диаграмму будут нанесены точки (1, 1), (2, 4), (3, 9), (4, 16) и (5, 25); результат показан на рис. 15.6.

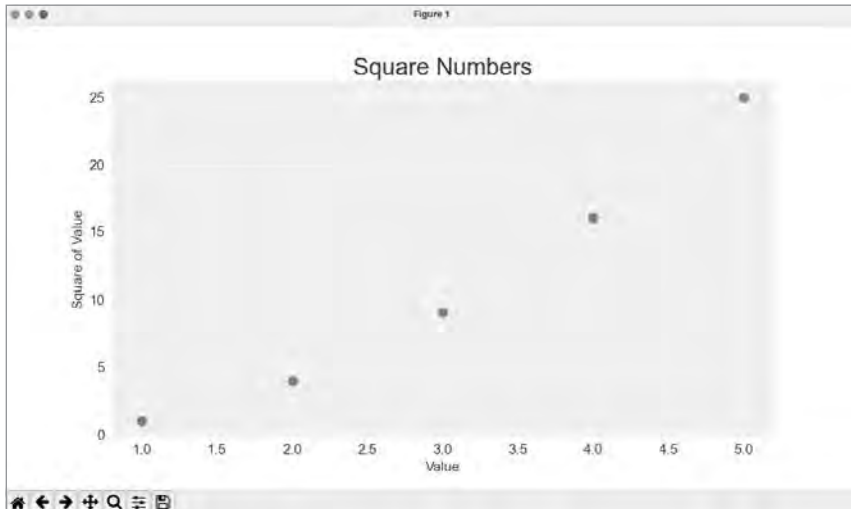


Рис. 15.6. Точечная диаграмма с несколькими точками

Автоматическое вычисление данных

Создавать списки вручную неэффективно, особенно при большом объеме данных. Вместо того чтобы передавать данные в виде списка, мы воспользуемся циклом Python, который выполнит вычисления за нас.

Вот как выглядит такой цикл для тысячи точек:

scatter_squares.py

```
import matplotlib.pyplot as plt

❶ x_values = range(1, 1001)
   y_values = [x**2 for x in x_values]

   plt.style.use('seaborn')
   fig, ax = plt.subplots()
❷ ax.scatter(x_values, y_values, s=10)

   # Задание заголовка диаграммы и меток осей.
   --пропуск--

   # Задание диапазона для каждой оси.
❸ ax.axis([0, 1100, 0, 1_100_000])

plt.show()
```

Все начинается со списка значений координаты x с числами от 1 до 1000 ❶. Затем генератор списка создает значения y , перебирая значения x (`for x in x_values`), возводя каждое число в квадрат ($x**2$) и сохраняя результаты в `y_values`. Затем оба списка (входной и выходной) передаются функции `scatter()` ❷. Набор данных велик, поэтому мы задаем меньший размер точек.

До вывода диаграммы метод `axis()` используется для задания диапазона каждой оси ❸. Метод `axis()` получает четыре значения: минимум и максимум по осям X и Y . В данном случае по оси X откладывается диапазон от 0 до 1100, а по оси Y — диапазон от 0 до 1 100 000. На рис. 15.7 показан результат.

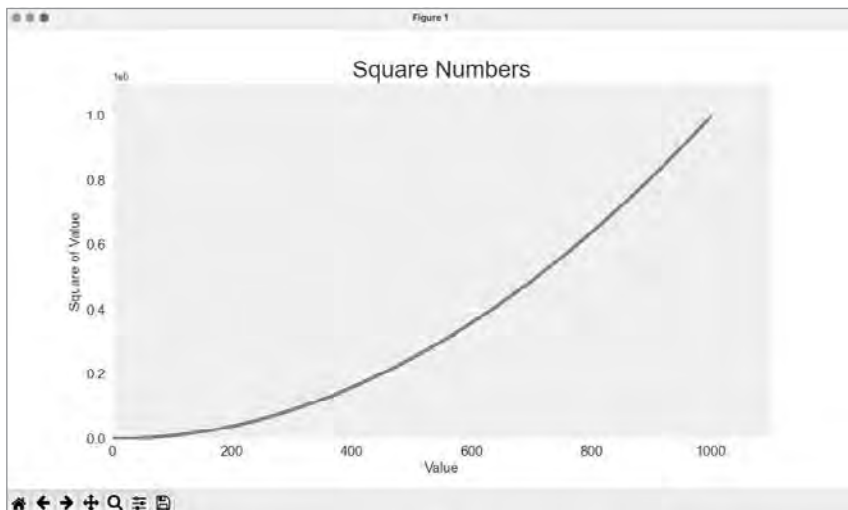


Рис. 15.7. Диаграмма с тысячей точками создается так же легко, как и диаграмма с пятью точками

Настройка меток на осях

Если числа в метках на осях достаточно велики, то Matplotlib по умолчанию оформляет их, используя научную запись. Обычно это приемлемо, поскольку большие числа в обычной записи занимают много места на диаграмме.

Практически все элементы диаграммы можно настроить, поэтому вы можете использовать обычную запись, если она вам больше нравится:

```
--пропуск--
# Назначение диапазона для каждой оси.
ax.axis([0, 1100, 0, 1_100_000])
ax.ticklabel_format(style='plain')

plt.show()
```

Метод `ticklabel_format()` позволяет изменить стиль меток, используемый по умолчанию, для любой диаграммы.

Определение пользовательских цветов

Чтобы изменить цвет точек, передайте `scatter()` аргумент `c` с именем используемого цвета, заключенным в одинарные кавычки:

```
ax.scatter(x_values, y_values, c='red', s=10)
```

Кроме того, можно определять пользовательские цвета в цветовой модели RGB. Чтобы сделать это, передайте аргумент `c` с кортежем из трех дробных значений (для красной, зеленой и синей составляющих) в диапазоне от 0 до 1. Например, следующая строка создает диаграмму со светло-зелеными точками:

```
ax.scatter(x_values, y_values, c=(0, 0.8, 0), s=10)
```

Значения, близкие к 0, дают более темные цвета, а значения, близкие к 1, — более светлые.

Цветовые карты

Цветовая карта (colormap) представляет собой серию цветов градиента, определяющую плавный переход от начального цвета к конечному. Цветовые карты используются в визуализациях для выделения закономерностей в данных. Например, малые значения можно обозначить светлыми цветами, а большие — темными. Использование цветовой карты позволяет гарантировать, что все точки диаграммы меняют цвет плавно и точно по хорошо продуманной схеме.

Модуль `pyplot` содержит набор встроенных цветовых карт. Чтобы воспользоваться одной из них, вы должны указать, как модуль должен присваивать цвет каждой

точке набора данных. В следующем примере цвет каждой точки присваивается на основании значения по оси Y :

scatter_squares.py

```
--пропуск--  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
ax.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues, s=10)
```

```
# Задание заголовка диаграммы и меток осей.
```

```
--пропуск--
```

Аргумент `c` аналогичен аргументу `color`, но используется для связывания последовательности значений с цветовым сопоставлением. Мы передаем в `c` список значений по оси Y , а затем указываем `pyplot`, какая цветовая карта должна использоваться, через аргумент `cmap`. Следующий код окрашивает точки с меньшими значениями y в светло-синий цвет, а точки с большими значениями y — в темно-синий. Полученная диаграмма изображена на рис. 15.8.

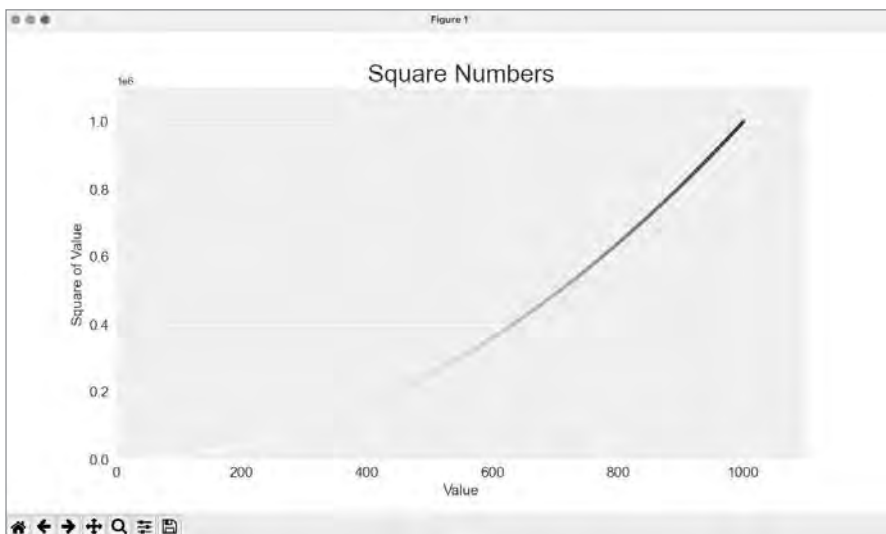


Рис. 15.8. Точечная диаграмма, составленная с помощью цветовой карты Blues

ПРИМЕЧАНИЕ

Все цветовые карты, доступные в `pyplot`, можно просмотреть на сайте <https://matplotlib.org/>; откройте раздел Tutorials (Руководства), прокрутите содержимое до пункта Colors (Цвета) и щелкните на ссылке Choosing Colormaps in Matplotlib (Выбор цветовых карт в Matplotlib).

Автоматическое сохранение диаграмм

Если вы хотите, чтобы программа автоматически сохраняла диаграмму в файле, то замените вызов `plt.show()` вызовом `plt.savefig()`:

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

Первый аргумент содержит имя файла, в котором должна сохраняться диаграмма; файл будет расположен в одном каталоге с `scatter_squares.py`. Второй аргумент удаляет из диаграммы лишние пробельные символы. Если вы хотите оставить эти символы, то данный аргумент можно опустить. Вы также можете вызвать функцию `savefig()`, передав ей объект `Path`, и сохранить выходной файл в любой каталог вашей системы.

УПРАЖНЕНИЯ

15.1. Кубы. Число, возведенное в третью степень, называется *кубом*. Нанесите на диаграмму первые пять кубов, а затем первые 5000 кубов.

15.2. Цветные кубы. Примените цветовую карту к диаграмме с кубами.

Случайное блуждание

В этом разделе мы используем Python для генерирования данных для случайного обхода, а затем с помощью Matplotlib создадим привлекательное представление сгенерированных данных. *Случайным блужданием* (random walk) называется путь, который не имеет четкого направления, но определяется серией полностью случайных решений. Представьте, что муравей делает каждый новый шаг в случайном направлении; его путь напоминает случайное блуждание.

Случайное блуждание находит практическое применение в естественных науках, физике, биологии, химии и экономике. Например, пылинки на поверхности водяной капли движется по поверхности, поскольку ее постоянно подталкивают молекулы воды. Движение молекул в капле воды случайно, поэтому путь пылинки на поверхности представляет собой случайное блуждание. Код, который мы напишем, можно использовать при моделировании многих реальных ситуаций.

Создание класса *RandomWalk*

Чтобы создать путь случайного блуждания, мы напишем класс `RandomWalk`, который принимает случайные решения по выбору направления. Классу нужны три атрибута: переменная для хранения количества точек в пути и два списка для координат *x* и *y* каждой точки.

Класс `RandomWalk` содержит всего два метода: `__init__()` и `fill_walk()` для вычисления точек случайного блуждания. Начнем с метода `__init__()`:

random_walk.py

```
❶ from random import choice

class RandomWalk:
    """Класс для генерирования случайных блужданий."""

    ❷ def __init__(self, num_points=5000):
        """Инициализирует атрибуты блуждания."""
        self.num_points = num_points

        # Все блуждания начинаются с точки (0, 0).
    ❸ self.x_values = [0]
        self.y_values = [0]
```

Чтобы принимать случайные решения, мы сохраним возможные варианты в списке и используем функцию `choice()` из модуля `random` для принятия решения ❶. Затем для списка задаем количество точек по умолчанию равным 5000 — достаточно большим, чтобы генерировать интересные закономерности, но достаточно малым, чтобы блуждания генерировались быстро ❷. Затем в строке ❸ создаем два списка для хранения значений x и y , после чего каждый путь начинается с точки $(0, 0)$.

Выбор направления

Метод `fill_walk()`, как показано ниже, заполняет путь точками и определяет направление каждого шага. Добавьте этот метод в файл `random_walk.py`:

random_walk.py

```
def fill_walk(self):
    """Вычисляет все точки блуждания."""

    # Шаги генерируются, пока не будет достигнута нужная длина.
    ❶ while len(self.x_values) < self.num_points:

        # Определение направления и длины перемещения.
        ❷ x_direction = choice([1, -1])
        x_distance = choice([0, 1, 2, 3, 4])
        ❸ x_step = x_direction * x_distance

        y_direction = choice([1, -1])
        y_distance = choice([0, 1, 2, 3, 4])
        ❹ y_step = y_direction * y_distance

        # Отклонение нулевых перемещений.
        ❺ if x_step == 0 and y_step == 0:
            continue
```

```

6      # Вычисление следующей позиции.
      x = self.x_values[-1] + x_step
      y = self.y_values[-1] + y_step

      self.x_values.append(x)
      self.y_values.append(y)

```

Сначала запускается цикл, который выполняется вплоть до заполнения пути правильным количеством точек ❶. Главная часть метода `fill_walk()` сообщает Python, как следует моделировать четыре случайных решения: двигаться вправо или влево? Как далеко идти в этом направлении? Двигаться ли вверх или вниз? Как далеко идти в этом направлении?

Выражение `choice([1, -1])` выбирает значение `x_direction`; оно возвращает 1 для перемещения вправо или `-1` для движения влево ❷. Затем выражение `choice([0, 1, 2, 3, 4])` определяет дальность перемещения в этом направлении (`x_distance`) случайным выбором целого числа от 0 до 4. (Добавление 0 позволяет выполнять шаги по оси *Y*, а также шаги со смещением по обеим осям.)

В точках ❸ и ❹ определяется длина каждого шага в направлениях *x* и *y*, для чего направление движения умножается на выбранное расстояние. При положительном результате `x_step` смещает вправо, при отрицательном — влево, при нулевом — вертикально. При положительном результате `y_step` смещает вверх, при отрицательном — вниз, при нулевом — горизонтально. Если оба значения `x_step` и `y_step` равны 0, то блуждание останавливается, но цикл продолжается ❺.

Чтобы получить следующее значение *x*, мы прибавляем значение `x_step` к последнему значению, хранящемуся в `x_values` ❻, и делаем то же самое для значений *y*. После того как значения будут получены, они присоединяются к `x_values` и `y_values`.

Вывод случайного блуждания

Ниже приведен код отображения всех точек блуждания:

rw_visual.py

```

import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Создание случайного блуждания.
❶ rw = RandomWalk()
   rw.fill_walk()

# Нанесение точек на диаграмму.
plt.style.use('classic')
fig, ax = plt.subplots()
❷ ax.scatter(rw.x_values, rw.y_values, s=15)
❸ ax.set_aspect('equal')
   plt.show()

```

Сначала программа импортирует модуль `pyplot` и класс `RandomWalk`. Затем создает случайное блуждание и сохраняет его в `rw` ❶, не забывая вызвать `fill_walk()`. Далее программа передает функции `scatter()` координаты x и y блуждания и выбирает подходящий размер точки ❷. По умолчанию Matplotlib масштабирует каждую ось независимо. Но такой подход растягивает большинство блужданий по горизонтали или вертикали. Здесь мы используем метод `set_aspect()`, чтобы обе оси имели равные расстояния между делениями ❸.

На рис. 15.9 показана диаграмма с 5000 точками. (В изображениях этого раздела область просмотра Matplotlib не показана, но вы увидите ее при запуске программы `rw_visual.py`.)

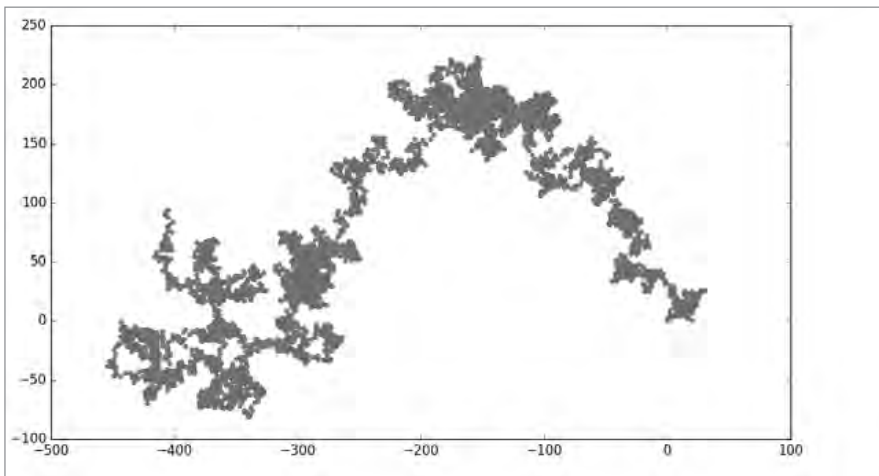


Рис. 15.9. Случайное блуждание с 5000 точками

Генерирование нескольких случайных блужданий

Все случайные блуждания отличаются друг от друга; интересно понаблюдать за тем, какие узоры генерирует программа. Один из способов использования предыдущего кода — создать несколько блужданий, не запуская многократно программу в цикле `while`:

`rw_visual.py`

```
import matplotlib.pyplot as plt
```

```
from random_walk import RandomWalk
```

```
# Новые блуждания создаются до тех пор, пока программа остается активной.
```

```
while True:
```

```
    # Создание случайного блуждания.
```

```
    --пропуск--
```

```
    plt.show()
```

```
keep_running = input("Make another walk? (y/n): ")
if keep_running == 'n':
    break
```

Код генерирует случайное блуждание, отображает его в области просмотра Matplotlib и делает паузу, при этом область просмотра открыта. Когда вы ее закрываете, программа спрашивает, хотите ли вы сгенерировать следующее блуждание. Ответьте *y*, и сможете сгенерировать блуждания, которые начинаются рядом с начальной точкой, а затем отклоняются преимущественно в одном направлении; при этом большие группы будут соединяться тонкими секциями. Чтобы завершить программу, введите *n*.

Форматирование блужданий

В этом подразделе мы настроим диаграмму так, чтобы выделить важные характеристики каждого блуждания и отвести на второй план несущественные элементы. Для этого мы определим характеристики, которые нужно выделить (например, откуда началось блуждание, где оно закончилось и по какому пути следовало). Затем определим характеристики, которым нужно уделять меньше внимания (например, деления шкалы и метки). Результатом должно быть простое визуальное представление, которое четко описывает путь, использованный в каждом случайном блуждании.

Колоризация точек

Мы используем цветовую карту для отображения точек блуждания, а также удаляем черный контур из каждой точки, чтобы цвет точек был лучше виден. Чтобы точки окрашивались в соответствии с их позицией в блуждании, мы передаем в аргументе *s* список с позицией каждой точки. Так как точки выводятся по порядку, список просто содержит числа от 1 до 4999:

rw_visual.py

```
--пропуск--
while True:
    # Создание случайного блуждания
    rw = RandomWalk()
    rw.fill_walk()

    # Нанесение точек на диаграмму.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    ❶ point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
               edgecolors='none', s=15)
    ax.set_aspect('equal')
    plt.show()
--пропуск--
```


Функция `range()` используется для генерирования списка чисел, размер которого равен количеству точек в блуждании ❶. Полученный результат сохраняется в списке `point_numbers`, который используется для назначения цвета каждой точки в блуждании. Мы передаем `point_numbers` в аргументе `c`, используем цветовую карту `Blues` и затем передаем `edgecolors='none'`, чтобы удалить черный контур вокруг каждой точки. В результате создается диаграмма блуждания с градиентным переходом от светло-синего к темно-синему (рис. 15.10).

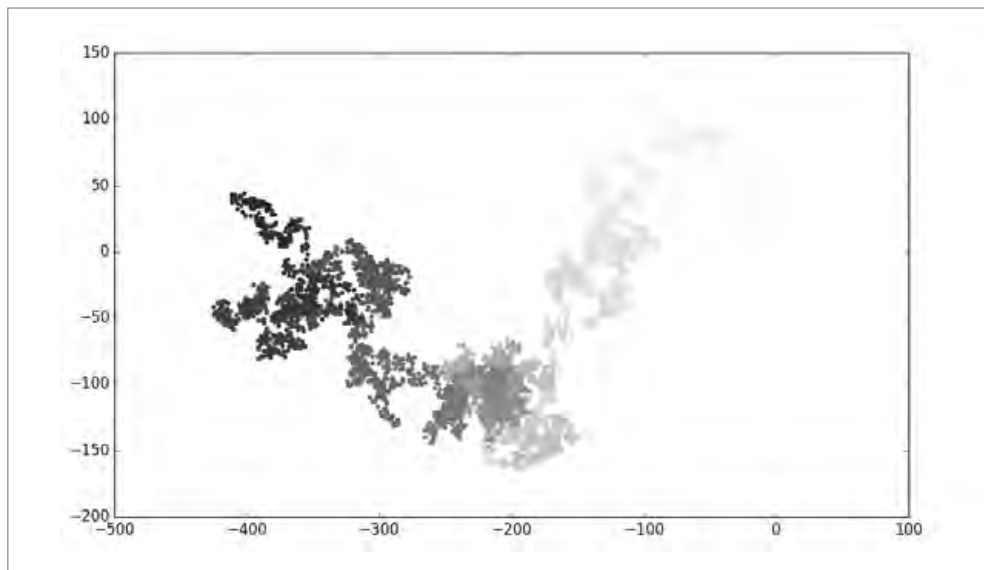


Рис. 15.10. Случайное блуждание, окрашенное с применением цветовой карты `Blues`

Форматирование начальной и конечной точек

Помимо раскраски точек, обозначающей их позицию, было бы неплохо видеть, где начинается и заканчивается каждое блуждание. Для этого можно прорисовать первую и последнюю точки отдельно, после нанесения на диаграмму основной серии. Мы увеличим конечные точки и раскрасим их другим цветом, чтобы они выделялись на общем фоне:

rw_visual.py

```
--nponуск--
while True:
    --nponуск--
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
               edgecolors='none', s=15)
    ax.set_aspect('equal')
```

```
# Выделение первой и последней точек.
ax.scatter(0, 0, c='green', edgecolors='none', s=100)
ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
           s=100)

plt.show()
--пропуск--
```

Чтобы вывести начальную точку, мы рисуем точку (0, 0) зеленым цветом и придаем ей больший размер ($s=100$) по сравнению с остальными точками. Для выделения конечной точки последняя пара координат x и y выводится с размером 100. Обязательно вставьте этот код непосредственно перед вызовом `plt.show()`, чтобы начальная и конечная точки выводились поверх всех остальных.

При выполнении этого кода вы будете точно видеть, где начинается и кончается каждое блуждание. (Если конечные точки не выделяются достаточно четко, то настраивайте их цвет и размер, пока не достигнете желаемого результата.)

Удаление осей

Уберем оси с диаграммы, чтобы они не отвлекали зрителя от общей картины. Для удаления осей используется следующий код:

```
rw_visual.py
--пропуск--
while True:
    --пропуск--
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
               s=100)

    # Удаление осей.
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    plt.show()
    --пропуск--
```

Мы используем методы `ax.get_xaxis()` и `ax.get_yaxis()` для получения доступа к каждой оси, а затем цепочку методов `set_visible()`, чтобы скрыть оси. В процессе работы над визуализацией данных вы часто будете встречать такую совокупность методов для настройки различных аспектов диаграмм.

Запустите программу `rw_visual.py`; теперь выводимые диаграммы не имеют осей.

Добавление точек

Увеличим количество точек, чтобы работать с большим объемом данных. Для этого увеличим значение `num_points` при создании экземпляра `RandomWalk` и отрегулируем размер каждой точки при выводе диаграммы:

rw_visual.py

```
--nponyск--
while True:
    # Создание случайного блуждания.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Вывод точек и отображение диаграммы.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
               edgecolor='none', s=1)
--nponyск--
```

В этом примере создается случайное блуждание из 50 000 точек (что в большей степени соответствует реальным данным), и каждая из них рисуется размером `s=1`. Как видно из рис. 15.11, изображение получается эфемерным и туманным. Простая точечная диаграмма превратилась в произведение искусства!

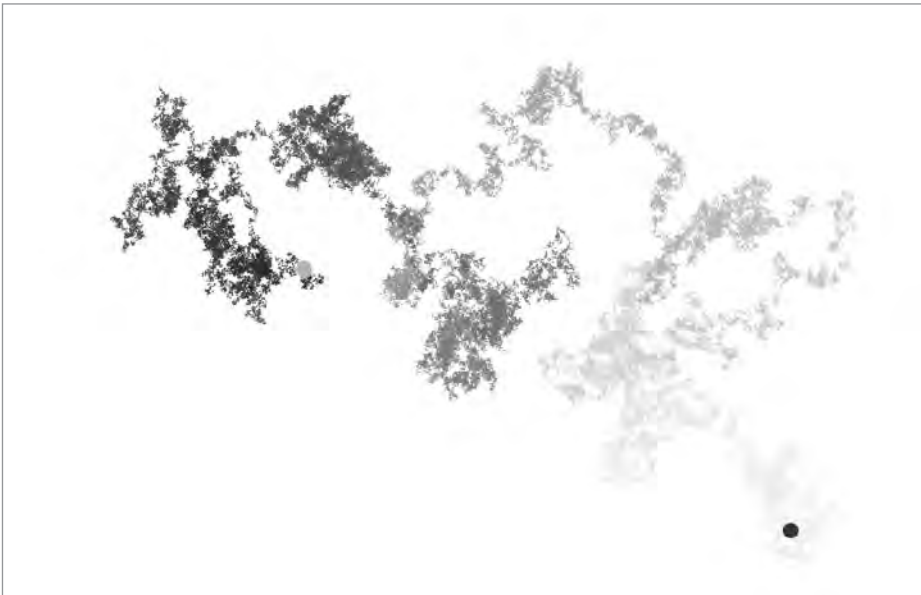


Рис. 15.11. Случайное блуждание с 50 000 точек

Поэкспериментируйте с этим кодом и посмотрите, насколько вам удастся увеличить количество точек в случайном блуждании, прежде чем работа системы начнет сильно замедляться или диаграмма потеряет свою визуальную привлекательность.

Изменение размера диаграммы для заполнения экрана

Визуализация гораздо эффективнее передает закономерности в данных, если адаптирована под размер экрана. Чтобы диаграмма лучше смотрелась на экране, измените размер области просмотра Matplotlib в вызове `subplots()`:

```
fig, ax = plt.subplots(figsize=(15, 9))
```

Функция `figure()` управляет шириной, высотой, разрешением и цветом фона диаграммы. Параметр `figsize` получает кортеж с размерами окна диаграммы в дюймах.

Matplotlib предполагает, что разрешение экрана составляет 100 пикселей на дюйм; если этот код не дает точного размера, то внесите необходимые изменения в числа. Или, если знаете разрешение экрана в вашей системе, передайте его `subplots()` в параметре `dpi` для выбора размера, эффективно использующего доступное пространство:

```
fig, ax = plt.subplots(figsize=(10, 6), dpi=128)
```

Так пространство на экране будет использоваться наиболее эффективно.

УПРАЖНЕНИЯ

15.3. Движение молекул. Измените программу `rw_visual.py` и замените `plt.scatter()` вызовом `plt.plot()`. Чтобы смоделировать путь пылинки на поверхности водной капли, передайте значения `rw.x_values` и `rw.y_values` и добавьте аргумент `linewidth`. Используйте 5000 точек вместо 50 000, чтобы не перегружать диаграмму.

15.4. Измененные случайные блуждания. В классе `RandomWalk` значения `x_step` и `y_step` генерируются по единому набору условий. Направление выбирается случайно из списка `[1, -1]`, а расстояние — из списка `[0, 1, 2, 3, 4]`. Измените значения в этих списках и посмотрите, что произойдет с общей формой диаграммы. Попробуйте применить расширенный список вариантов расстояния (например, от 0 до 8) или удалите `-1` из списка направлений по оси `X` или `Y`.

15.5. Рефакторинг. Метод `fill_walk()` получился слишком длинным. Создайте новый метод `get_step()`, который определяет расстояние и направление для каждого шага, после чего вычисляет этот шаг. В результате метод `fill_walk()` должен содержать два вызова `get_step()`:

```
x_step = self.get_step()
y_step = self.get_step()
```

Рефакторинг сокращает размер `fill_walk()`, а метод становится более простым и понятным.

Моделирование бросков кубиков с помощью Plotly

В этом разделе мы воспользуемся пакетом визуализации Plotly для создания интерактивных визуализаций. Пакет Plotly особенно хорошо подходит для визуализаций, которые будут отображаться в браузере, поскольку изображение автоматически масштабируется по размерам экрана зрителя. Кроме того, Plotly генерирует интерактивные визуализации; когда пользователь наводит указатель мыши на некий элемент, на экране появляется расширенная информация об этом элементе. С помощью Plotly Express мы создадим первичную визуализацию, используя буквально пару строк кода. *Plotly Express* — инструмент семейства Plotly, который ориентирован на создание диаграмм на основе минимального количества кода. Подготовив корректную диаграмму, мы настроим вывод по аналогии с Matplotlib.

В этом проекте мы займемся анализом результатов бросков кубиков. При броске одного шестигранного кубика существует равная вероятность выпадения любого числа от 1 до 6. С другой стороны, при броске двух кубиков одни суммы выпадают с большей вероятностью, чем другие. Чтобы определить, какие числа наиболее вероятны, мы сгенерируем набор данных, представляющих брошенные кубики. Затем на базе данных большого количества бросков будет создана диаграмма, по которой можно определить, какие результаты более вероятны.

Броски кубиков часто используются в математике для пояснения различных типов анализа данных. Кроме того, они находят применение и в реальном мире — например, в карточных играх и многих других ситуациях, где случайность играет важную роль.

Установка Plotly

Установите Plotly с помощью модуля `pip` по аналогии с тем, как вы делали это с Matplotlib:

```
$ python -m pip install --user plotly
$ python -m pip install --user pandas
```

Для эффективной работы с данными Plotly Express требуется библиотека `pandas`, поэтому нам нужно установить и ее. Если при установке Matplotlib вы использовали команду `python3` или другую, то убедитесь в том, что в данном случае используется та же команда.

Примеры визуализаций, которые могут быть созданы с помощью Plotly, представлены в галерее диаграмм: зайдите на сайт <https://plotly.com/python>. Каждый пример сопровождается исходным кодом, так что вы сможете увидеть, как была создана каждая из визуализаций.

Создание класса *Die*

Для моделирования броска одного кубика будет использоваться класс *Die*:

die.py

```
from random import randint
```

```
class Die:
```

```
    """Класс, представляющий один кубик."""
```

```
❶ def __init__(self, num_sides=6):
    """По умолчанию используется шестигранный кубик."""
    self.num_sides = num_sides

    def roll(self):
        """Возвращает случайное число от 1 до количества граней."""
❷ return randint(1, self.num_sides)
```

Метод `__init__()` получает один необязательный аргумент ❶. Если при создании экземпляра кубика аргумент с количеством сторон не передается, то по умолчанию создается шестигранный кубик. Если же аргумент *имеется*, то переданное значение используется для определения количества граней. (Кубики принято обозначать по количеству граней: шестигранный кубик — D6, восьмигранный — D8 и т. д.)

Метод `roll()` использует функцию `randint()` для получения случайного числа в диапазоне от 1 до количества граней ❷. Функция может вернуть начальное значение (1), конечное значение (`num_sides`) или любое целое число в этом диапазоне.

Бросок кубика

Прежде чем создавать визуализацию на основе этого класса, бросим кубик D6, выведем результаты и убедимся в том, что они имеют смысл:

die_visual.py

```
from die import Die
```

```
# Создание кубика D6.
```

```
❶ die = Die()
```

```
# Моделирование серии бросков с сохранением результатов в списке.
```

```
results = []
```

```
❷ for roll_num in range(100):
    result = die.roll()
    results.append(result)
```

```
print(results)
```

Сначала создается экземпляр `Die` с шестью гранями по умолчанию **❶**. Затем моделируются 100 бросков кубика **❷**, а результат каждого броска сохраняется в списке `results`. Выборка выглядит примерно так:

```
[4, 6, 5, 6, 1, 5, 6, 3, 5, 3, 5, 3, 2, 2, 1, 3, 1, 5, 3, 6, 3, 6, 5, 4,
1, 1, 4, 2, 3, 6, 4, 2, 6, 4, 1, 3, 2, 5, 6, 3, 6, 2, 1, 1, 3, 4, 1, 4,
3, 5, 1, 4, 5, 5, 2, 3, 3, 1, 2, 3, 5, 6, 2, 5, 6, 1, 3, 2, 1, 1, 1, 6,
5, 5, 2, 2, 6, 4, 1, 4, 5, 1, 1, 1, 4, 5, 3, 3, 1, 3, 5, 4, 5, 6, 5, 4,
1, 5, 1, 2]
```

Беглое знакомство с результатами показывает, что класс `Die` работает. В результатах встречаются граничные значения 1 и 6, то есть модель возвращает наименьшее и наибольшее возможные значения; значения 0 и 7 не встречаются, а значит, все результаты лежат в диапазоне допустимых значений. Кроме того, в выборке встречаются все числа от 1 до 6, то есть представлены все возможные результаты.

Анализ результатов

Чтобы проанализировать результаты бросков одного кубика D6, мы подсчитаем, сколько раз выпадало каждое число:

die_visual.py

--пропуск--

Моделирование серии бросков с сохранением результатов в списке.

`results = []`

```
❶ for roll_num in range(1000):
    result = die.roll()
    results.append(result)
```

Анализ результатов.

`frequencies = []`

```
❷ poss_results = range(1, die.num_sides+1)
for value in poss_results:
❸     frequency = results.count(value)
❹     frequencies.append(frequency)
```

`print(frequencies)`

Поскольку мы больше не выводим результаты, количество моделируемых бросков можно увеличить до 1000 **❶**. Чтобы проанализировать броски, создадим пустой список `frequencies`, в котором хранится количество выпадений каждого значения. Программа перебирает возможные значения (от 1 до количества сторон кубика) **❷**, подсчитывает количество вхождений каждого числа в результатах **❸**, после чего присоединяет полученное значение к списку `frequencies` **❹**. Содержимое списка выводится перед созданием визуализации:

```
[155, 167, 168, 170, 159, 181]
```

Результаты выглядят разумно: мы видим все шесть частот выпадения, по одной для каждого возможного результата при броске D6, и ни одна из них не выделяется на общем фоне. А теперь займемся наглядным представлением результатов.

Создание гистограммы

Теперь, обладая нужными данными, мы можем сгенерировать визуализацию буквально за пару строк кода с помощью Plotly Express:

die_visual.py

```
import plotly.express as px

from die import Die
--пропуск--

for value in poss_results:
    frequency = results.count(value)
    frequencies.append(frequency)

# Визуализация результатов.
fig = px.bar(x=poss_results, y=frequencies)
fig.show()
```

Сначала мы импортируем модуль `plotly.express` под псевдонимом `px`. Затем с помощью функции `px.bar()` создаем гистограмму. Для создания простейшей гистограммы нам нужно передать этой функции лишь значения координат по осям *X* и *Y*. В данном случае значения *x* — это вероятные результаты броска одного кубика, а значения *y* — частота выпадения каждого возможного значения.

В последней строке вызывается функция `fig.show()`, с помощью которой Plotly визуализирует полученную гистограмму в HTML-файл и открывает его на новой вкладке браузера. Результат показан на рис. 15.12.

Получился очень простой график, и это, разумеется, черновой вариант. Но именно так и следует использовать Plotly Express: вы пишете пару строк кода, изучаете диаграмму и проверяете, что она передает данные корректно. Если вам нравится результат, то вы можете настроить диаграмму, например, изменив ее внешний вид или метки. А если хотите опробовать другие виды диаграмм, то можете заменить функцию `px.bar()` на `px.scatter()` или `px.line()`. Полный список доступных вариантов диаграмм опубликован на сайте <https://plotly.com/python/plotly-express>.

Полученная гистограмма динамична и интерактивна. Если изменить размер окна браузера, гистограмма тоже изменит размер в соответствии с доступным пространством. Наведя указатель мыши на любой столбец гистограммы, вы увидите всплывающее окно с конкретными данными, связанными с ним.

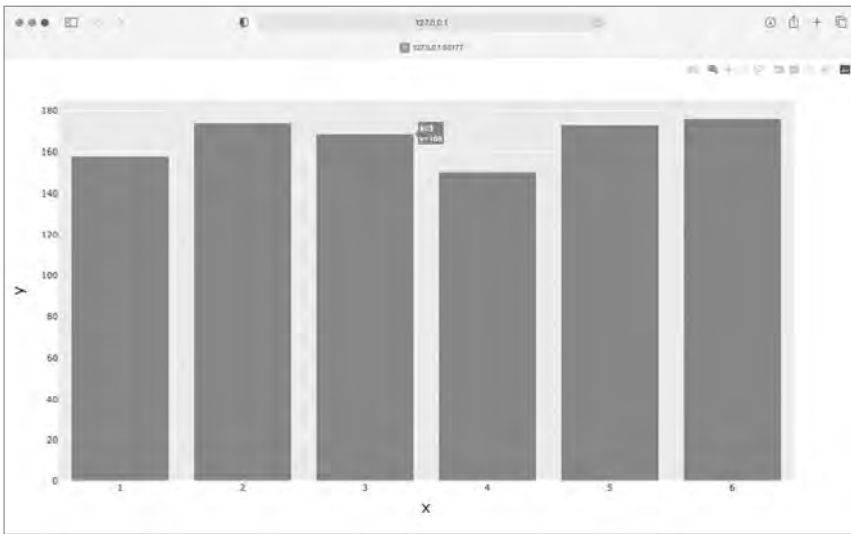


Рис. 15.12. Начальная гистограмма, созданная с помощью Plotly Express

Настройка диаграммы

Создав подходящую диаграмму и убедившись, что она точно отображает наши данные, мы можем добавить подходящие метки и отформатировать диаграмму.

Один из вариантов настройки диаграммы с помощью Plotly заключается в использовании опциональных параметров при вызове функции, генерирующей график, в данном случае `px.bar()`. Ниже показано, как добавить общий заголовок и метку к каждой оси:

die_visual.py

--пропуск--

Визуализация результатов.

- ❶ `title = "Results of Rolling One D6 1,000 Times"`
 - ❷ `labels = {'x': 'Result', 'y': 'Frequency of Result'}`
- ```
fig = px.bar(x=pos_results, y=frequencies, title=title, labels=labels)
fig.show()
```

Сначала мы добавляем заголовок с помощью переменной `title` ❶. Для добавления меток к осям используем словарь ❷. Ключи в нем ссылаются на оси, метки к которым мы добавляем, а значения содержат сам текст меток. В данном примере мы присваиваем оси *X* метку `Result`, а оси *Y* — метку `Frequency of Result`. Вызов функции `px.bar()` теперь содержит необязательные аргументы `title` и `labels`.

Теперь при создании диаграммы добавляются указанные заголовок и метки каждой оси (рис. 15.13).

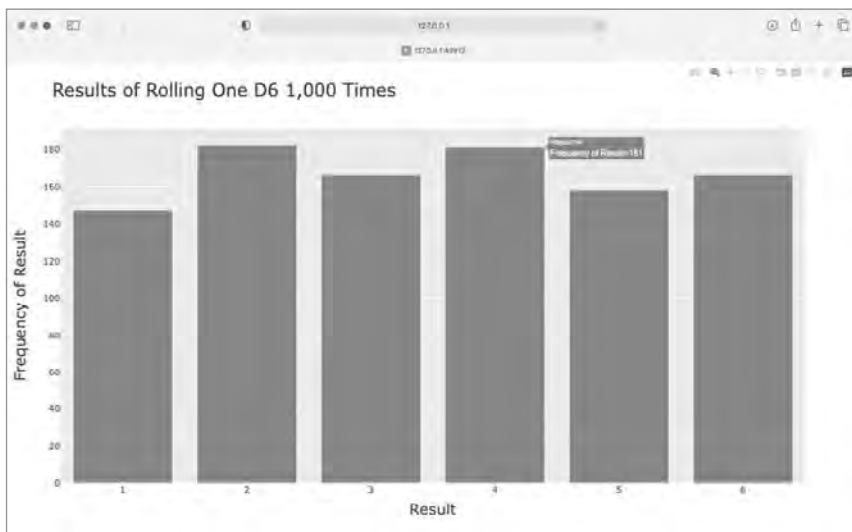


**Рис. 15.13.** Простая диаграмма, созданная с помощью Plotly

## Бросок двух кубиков

При броске двух кубиков вы получаете большие значения с другим распределением результатов. Изменим наш код и создадим два кубика D6, моделирующих бросок пары кубиков. При броске каждой пары программа складывает два числа (по одному с каждого кубика) и сохраняет сумму в `results`. Сохраните копию файла `die_visual.py` под именем `dice_visual.py` и внесите следующие изменения:

### *`dice_visual.py`*

```
import plotly.express as px
```

```
from die import Die
```

```
Создание двух кубиков D6.
```

```
die_1 = Die()
```

```
die_2 = Die()
```

```
Моделирование серии бросков с сохранением результатов в списке.
```

```
results = []
```

```
for roll_num in range(1000):
```

```
❶ result = die_1.roll() + die_2.roll()
 results.append(result)
```

```
Анализ результатов.
frequencies = []
❷ max_result = die_1.num_sides + die_2.num_sides
❸ poss_results = range(2, max_result+1)
for value in poss_results:
 frequency = results.count(value)
 frequencies.append(frequency)

Визуализация результатов.
title = "Results of Rolling Two D6 Dice 1,000 Times"
labels = {'x': 'Result', 'y': 'Frequency of Result'}
fig = px.bar(x=poss_results, y=frequencies, title=title, labels=labels)
fig.show()
```

Создав два экземпляра `Die`, мы бросаем кубики и вычисляем сумму для каждого броска ❶. Наименьший возможный результат (2) равен сумме наименьших результатов на обоих кубиках. Наибольший возможный результат (12) вычисляется путем суммирования наибольших результатов на обоих кубиках; мы сохраняем его в `max_result` ❷. Переменная `max_result` упрощает код для генерации `poss_results` ❸. Кроме того, можно было использовать диапазон `range(2, 13)`, но он работал бы только для двух кубиков D6. При моделировании реальных ситуаций лучше писать код, который легко адаптируется для разных ситуаций. В частности, этот код позволяет смоделировать бросок пары кубиков с любым количеством граней.

После выполнения кода в браузере должна появиться диаграмма, примерный вид которой показан на рис. 15.14.

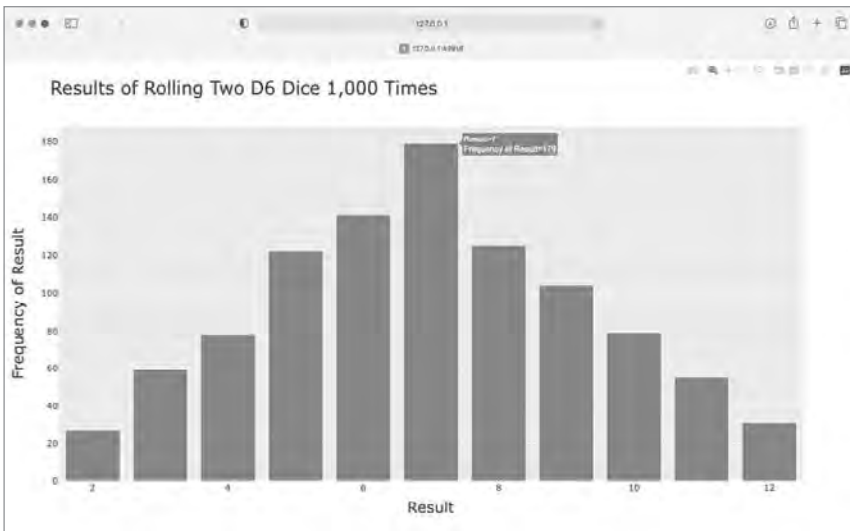


**Рис. 15.14.** Смоделированные результаты 1000 бросков двух шестигранных кубиков

На диаграмме показаны примерные результаты, которые могут быть получены для пары кубиков D6. Как видите, реже всего выпадают результаты 2 и 12, а чаще всего 7, поскольку эта комбинация может быть выброшена шестью способами, а именно:  $1 + 6$ ,  $2 + 5$ ,  $3 + 4$ ,  $4 + 3$ ,  $5 + 2$  и  $6 + 1$ .

## Дальнейшая настройка диаграммы

Созданная диаграмма имеет одну проблему, которую мы должны решить. На ней отображаются 11 столбцов, поэтому стандартные настройки расположения оси X скрывают метки некоторых столбцов. Несмотря на то что стандартные настройки прекрасно подходят для большинства визуализаций, наша диаграмма выглядела бы лучше, если бы все столбцы сопровождалась метками.

В Plotly доступен метод `update_layout()`, используемый для внесения самых разных изменений в диаграмму после ее создания. Рассмотрим, как добавить метки ко всем столбцам:

### *dice\_visual.py*

--пропуск--

```
fig = px.bar(x=pos_results, y=frequencies, title=title, labels=labels)
```

```
Дальнейшая настройка диаграммы.
```

```
fig.update_layout(xaxis_dtick=1)
```

```
fig.show()
```

Метод `update_layout()` влияет на объект `fig`, представляющий собой сам график. В нашем примере мы используем аргумент `xaxis_dtick`, отвечающий за расстояние между метками на оси X. Мы установили его равным 1, чтобы каждый столбец сопровождался меткой. Снова запустив программу `dice_visual.py`, вы увидите метки у всех столбцов.

## Броски кубиков с разным количеством граней

Создадим кубики с шестью и десятью гранями и посмотрим, что произойдет, если бросить их 50 000 раз:

### *dice\_visual\_d6d10.py*

```
import plotly.express as px
```

```
from die import Die
```

```
Создание кубиков D6 и D10.
```

```
die_1 = Die()
```

```
❶ die_2 = Die(10)
```

```
Моделирование серии бросков с сохранением результатов в списке.
```

```
results = []
```

```
for roll_num in range(50_000):
```

```
 result = die_1.roll() + die_2.roll()
```

```
 results.append(result)
```

```
Анализ результатов.
```

```
--пропуск--
```

```
Визуализация результатов.
```

```
❷ title = "Results of Rolling a D6 and a D10 50,000 Times"
```

```
labels = {'x': 'Result', 'y': 'Frequency of Result'}
```

```
--пропуск--
```

Чтобы добавить модель кубика D10, мы передаем аргумент **10** при создании второго экземпляра **Die** ❶ и изменяем первый цикл для моделирования 50 000 бросков вместо 1000. Затем соответственно изменяем заголовок ❷.

На рис. 15.15 показана полученная диаграмма. Вместо одного наиболее вероятного результата их стало пять. Это объясняется тем, что наименьшее ( $1 + 1$ ) и наибольшее ( $6 + 10$ ) значения по-прежнему могут быть получены только одним способом, но кубик D6 ограничивает количество способов генерирования средних чисел: суммы 7, 8, 9, 10 и 11 можно выбросить шестью способами. Следовательно, именно эти результаты являются наиболее частыми, и все эти числа выпадают с равной вероятностью.

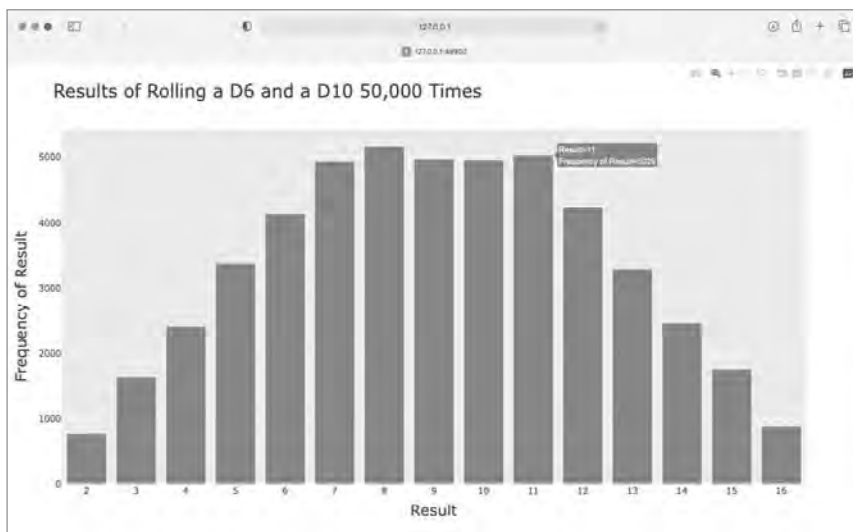


**Рис. 15.15.** Результаты 50 000 бросков шести- и десятигранного кубиков

Возможность применения Plotly для моделирования бросков кубиков дает большую свободу при исследовании этого явления. За считанные минуты вы сможете смоделировать огромное количество бросков с разнообразными кубиками.

## Сохранение диаграммы в файл

Подготовленную диаграмму вы всегда можете сохранить в HTML-файл и просмотреть в браузере. Но можно сделать это и с помощью кода. Чтобы сохранить диаграмму в HTML-файл, замените вызов функции `fig.show()` на вызов `fig.write_html()`:

```
fig.write_html('dice_visual_d6d10.html')
```

Метод `write_html()` принимает лишь один аргумент: имя сохраняемого файла. Если вы вместо полного пути укажете только имя файла, то он будет сохранен в том же каталоге, что и файл `.py`. Вы также можете передать функции `write_html()` объект `Path` и записать выходной файл в любой каталог вашей системы.

## УПРАЖНЕНИЯ

**15.6. Два кубика D8.** Создайте модель, которая показывает, что происходит при 1000-кратном бросании двух восьмигранных кубиков. Попробуйте заранее (перед моделированием) представить, как будет выглядеть визуализация; проверьте правильность своих интуитивных представлений. Постепенно увеличивайте количество бросков, пока не начнете замечать ограничения, связанные с ресурсами вашей системы.

**15.7. Три кубика.** При броске трех кубиков D6 наименьший возможный результат равен 3, а наибольший — 18. Создайте визуализацию, которая показывает, что происходит при броске трех кубиков D6.

**15.8. Умножение.** При броске двух кубиков результат обычно вычисляется путем суммирования двух чисел. Создайте визуализацию, которая показывает, что происходит при умножении этих чисел.

**15.9. Генераторы кубиков.** Для наглядности в списках этого раздела используется длинная форма цикла `for`. Если вы хорошо владеете навыками работы с генераторами списков, то попробуйте написать генератор для одного или обоих циклов в каждой из этих программ.

**15.10. Эксперименты с библиотеками.** Попробуйте использовать `Matplotlib` для создания визуализации бросков кубиков, а `Plotly` — для создания визуализации случайного блуждания. (Для выполнения этого упражнения вам придется обратиться к документациям обеих библиотек.)

## Резюме

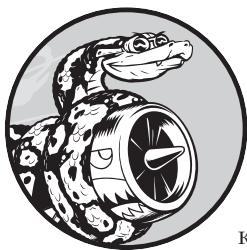
В этой главе вы научились генерировать наборы данных и создавать их визуализации. Вы узнали, как создавать простые диаграммы с помощью Matplotlib и применять точечные диаграммы для анализа случайных блужданий. Вы разобрались с тем, как создать диаграмму с помощью Plotly и исследовать результаты бросков кубиков с разным количеством граней, используя диаграмму.

Генерирование собственных наборов данных в программах — интересный и эффективный способ моделирования и анализа различных реальных ситуаций. В дальнейших проектах визуализации данных обращайтесь особое внимание на ситуации, которые можно смоделировать на программном уровне. Присмотритесь к визуализациям, встречающимся в выпусках новостей, — возможно, они были сгенерированы с помощью методов, сходных с теми, о которых вы узнали в этих проектах.

В главе 16 вы скачаете данные из сетевого источника и продолжите использовать Matplotlib и Plotly для анализа данных.

# 16

## Загрузка данных



В этой главе мы скачаем наборы данных из сетевого источника и создадим их рабочие визуализации. В Интернете можно найти невероятно разнообразную информацию, большая часть которой еще не подвергалась основательному анализу. Умение анализировать данные позволит вам выявить связи и закономерности, не найденные никем другим.

В этой главе рассматривается работа с данными в двух популярных форматах: CSV и JSON. Модуль Python `csv` будет применен для обработки погодных данных в формате CSV (с разделением запятыми) и анализа динамики высоких и низких температур в двух разных местах. Затем библиотека `Matplotlib` будет использована для создания на базе скачанных данных диаграммы колебания температур в двух разных местах: в Ситке (Аляска) и Долине Смерти (Калифорния). Позднее в этой главе модуль `json` будет использован для обращения к данным численности населения, хранимым в формате GeoJSON, а с помощью модуля `Plotly` будет создана карта с данными местоположения и магнитуд недавних землетрясений.

К концу этой главы вы будете готовы к работе с разными типами и форматами наборов данных и начнете лучше понимать принципы создания сложных визуализаций. Возможность загрузки и визуализации сетевых данных разных типов и форматов крайне важна для работы с разнообразными массивами данных в реальном мире.

### Формат CSV

Один из простейших вариантов хранения — запись данных в текстовый файл как серий значений, разделенных запятыми; такой формат хранения получил название *CSV* (от *Comma Separated Values*, то есть «значения, разделенные запятыми»). Например, одна строка погодных данных в формате CSV может выглядеть так:

```
"USW00025333", "SITKA AIRPORT, AK US", "2021-01-01", , "44", "40"
```



Это погодные данные за 1 января 2021 г. в Ситке (Аляска). В данных указаны максимальная и минимальная температуры, а также ряд других показателей за этот день. У человека могут возникнуть проблемы с чтением данных CSV, но такой формат хорошо подходит для программной обработки и извлечения значений, а это ускоряет процесс анализа.

Начнем с небольшого набора погодных данных в формате CSV, записанного в Ситке; файл с данными можно скачать с [https://ehmatthes.github.io/pcc\\_3e](https://ehmatthes.github.io/pcc_3e). Создайте папку `weather_data` в папке, в которой сохраняются программы этой главы. Скопируйте в созданную папку файл `sitka_weather_07-2021_simple.csv`. (После скачивания дополнительных материалов к книге в вашем распоряжении появятся все необходимые файлы для этого проекта.)

### ПРИМЕЧАНИЕ

Погодные данные для этого проекта были скачаны с сайта <https://ncdc.noaa.gov/cdo-web/>.

## Разбор заголовка файлов CSV

Модуль Python `csv` из стандартной библиотеки разбирает строки файла CSV и позволяет быстро извлечь нужные значения. Начнем с первой строки файла, которая содержит серию заголовков данных. Заголовки описывают информацию, хранящуюся в данных:

### *sitka\_highs.py*

```
from pathlib import Path
import csv
```

- ❶ `path = Path('weather_data/sitka_weather_07-2021_simple.csv')`  
`lines = path.read_text().splitlines()`
- ❷ `reader = csv.reader(lines)`
- ❸ `header_row = next(reader)`  
`print(header_row)`

Сначала мы импортируем `Path` и модуль `csv`. Затем создаем объект `Path`, располагаемый в папке `weather_data` и ссылающийся на специальный файл с информацией о погоде ❶. Интерпретатор считывает файл, а затем метод `splitlines()` извлекает из него список всех строк и присваивает переменной `lines`.

Далее создается объект `reader` ❷. Он используется для парсинга всех строк в файле. Чтобы создать объект `reader`, мы вызываем функцию `csv.reader()` и передаем ей список строк из файла CSV.

При передаче объекта `reader` функция `next()` возвращает следующую строку из файла, начиная с начала документа. Функция `next()` вызывается только раз для

получения первой строки файла, содержащей заголовки ❸. Возвращенные данные сохраняются в `header_row`. Как видите, строка включает описательные имена заголовков, которые сообщают, какая информация содержится в каждой строке данных:

```
['STATION', 'NAME', 'DATE', 'TAVG', 'TMAX', 'TMIN']
```

Объект `reader` обрабатывает первую строку значений, разделенных запятыми, и сохраняет все значения из строки в списке. Заголовок `STATION` представляет код метеорологической станции, зарегистрировавшей данные. Позиция заголовка указывает на то, что первым значением в каждой из следующих строк является код метеостанции. Заголовок `NAME` указывает на то, что второе значение в каждой строке — это название метеостанции, регистрирующей погоду. Остальные заголовки сообщают, какая информация хранится в соответствующем поле. В данном примере нас интересуют значения даты (`DATE`), а также высокой и низкой температуры (`TMAX` и `TMIN` соответственно). Мы используем простой набор данных, содержащий информацию только об уровне осадков и температуре. Вы также можете скачать собственный набор погодных данных и добавить в обработку другие показатели: скорость и направление ветра, расширенные данные осадков и т. д.

## Вывод заголовков и их позиций

Чтобы читателю было проще понять структуру данных в файле, выведем каждый заголовок и его позицию в списке:

### *sitka\_highs.py*

```
--пропуск--
reader = csv.reader(lines)
header_row = next(reader)

for index, column_header in enumerate(header_row):
 print(index, column_header)
```

Функция `enumerate()` возвращает индекс каждого элемента и его значение при переборе списка. (Обратите внимание: строка `print(header_row)` удалена ради этой более подробной версии.)

Результат с индексами всех заголовков выглядит так:

```
0 STATION
1 NAME
2 DATE
3 TAVG
4 TMAX
5 TMIN
```

Из этих данных видно, что даты и максимальные температуры за эти дни находятся в столбцах 2 и 4. Чтобы проанализировать температурные данные, мы обработаем каждую запись данных в файле `sitka_weather_07-2021_simple.csv` и извлечем элементы с индексами 2 и 4.

## Извлечение и чтение данных

Итак, нужные столбцы данных известны; попробуем прочитать часть этих данных. Начнем с чтения максимальной температуры за каждый день:

### *sitka\_highs.py*

```
--nponyск--
reader = csv.reader(lines)
header_row = next(reader)

Извлечение максимальных температур.
❶ highs = []
❷ for row in reader:
❸ high = int(row[4])
 highs.append(high)

print(highs)
```

Программа создает пустой список `highs` ❶ и перебирает остальные строки в файле ❷. Объект `reader` продолжает с того места, на котором он остановился в ходе чтения файла CSV, и автоматически возвращает каждую строку после текущей позиции. Заголовок уже прочитан, поэтому цикл продолжается со второй строки, в которой начинаются фактические данные. При каждом проходе цикла значение с индексом 4 (заголовок `TMAX`) присваивается переменной `high` ❸. Функция `int()` преобразует данные, хранящиеся в строковом виде, в числовой формат, чтобы их можно было использовать в дальнейшем. Значение присоединяется к списку `highs`.

В результате будет получен список `highs` со следующим содержимым:

```
[61, 60, 66, 60, 65, 59, 58, 58, 57, 60, 60, 60, 57, 58, 60, 61, 63, 63, 70,
 64, 59, 63, 61, 58, 59, 64, 62, 70, 70, 73, 66]
```

Мы извлекли максимальную температуру для каждого дня и сохранили полученные данные в списке. Следующим шагом станет создание визуализации этих данных.

## Нанесение данных на диаграмму

Для наглядного представления температурных данных мы сначала создадим простую диаграмму дневных максимумов температуры, используя Matplotlib:

### *sitka\_highs.py*

```
from pathlib import Path
import csv

import matplotlib.pyplot as plt

path = Path('weather_data/sitka_weather_07-2021_simple.csv')
lines = path.read_text().splitlines()
--nponyск--
```

```

Нанесение данных на диаграмму.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(highs, color='red')

Форматирование диаграммы.
❷ ax.set_title("Daily High Temperatures, July 2021", fontsize=24)
❸ ax.set_xlabel('', fontsize=16)
ax.set_ylabel("Temperature (F)", fontsize=16)
ax.tick_params(labelsize=16)

plt.show()

```

Мы передаем при вызове `plot()` список `highs` и аргумент `c='red'` для отображения точек красным цветом ❶. (Максимумы будут выводиться красным цветом, а минимумы — синим.) Затем указываем другие аспекты форматирования (например, размер шрифта и метки) ❷, уже знакомые нам по главе 15. Даты еще не добавлены, поэтому метки для оси *X* не задаются, но вызов `ax.set_xlabel()` изменяет размер шрифта, чтобы метки по умолчанию лучше читались ❸. На рис. 16.1 показана полученная диаграмма: это простой график максимальных температур за июль 2021 года в Ситке (штат Аляска).

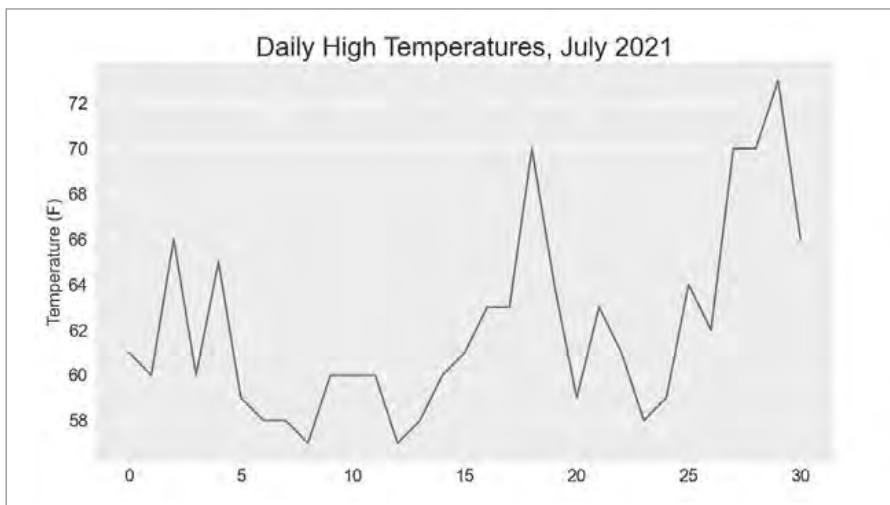


**Рис. 16.1.** График ежедневных максимальных температур в июле 2021 года в Ситке (штат Аляска)

## Модуль *datetime*

Теперь нанесем даты на график, чтобы с ним было удобнее работать. Первая дата из файла погодных данных хранится во второй строке файла:

```
"USW00025333", "SITKA AIRPORT, AK US", "2021-07-01", "61", "53"
```

Данные будут читаться в строковом формате, поэтому нам понадобится способ преобразовать строку '2021-07-1' в объект, представляющий эту дату. Чтобы создать объект, соответствующий 1 июля 2021 года, мы воспользуемся методом `strptime()` из модуля `datetime`. Посмотрим, как работает `strptime()` в терминальном окне:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2021-07-01', '%Y-%m-%d')
>>> print(first_date)
2021-07-01 00:00:00
```

Сначала необходимо импортировать класс `datetime` из модуля `datetime`. Затем вызывается метод `strptime()`, первый аргумент которого содержит строку с датой. Второй аргумент сообщает Python, как отформатирована дата. В данном примере благодаря разным значениям Python получает следующие указания:

- `%Y-` — часть строки, предшествующую первому дефису, интерпретировать как год из четырех цифр;
- `%m-` — часть строки перед вторым дефисом интерпретировать как число из двух цифр, представляющее месяц;
- `%d` — последнюю часть строки интерпретировать как день месяца от 1 до 31.

Метод `strptime()` может получать различные аргументы, которые описывают, как должна интерпретироваться запись даты. В табл. 16.1 перечислены некоторые из таких аргументов.

**Таблица 16.1.** Аргументы форматирования даты и времени из модуля `datetime`

| Аргумент | Описание                                |
|----------|-----------------------------------------|
| %A       | Название дня недели — например, Monday  |
| %B       | Название месяца — например, January     |
| %m       | Порядковый номер месяца (от 01 до 12)   |
| %d       | День месяца (от 01 до 31)               |
| %Y       | Год из четырех цифр (например, 2019)    |
| %y       | Две последние цифры года (например, 19) |
| %H       | Часы в 24-часовом формате (от 00 до 23) |
| %I       | Часы в 12-часовом формате (от 01 до 12) |
| %p       | АМ или РМ                               |
| %M       | Минуты (от 00 до 59)                    |
| %S       | Секунды (от 00 до 61)                   |

## Представление дат на диаграмме

Мы можем улучшить диаграмму температурных данных, извлекая даты ежедневных показаний максимальных температур и накладывая их на ось *x*:

### *sitka\_highs.py*

```
from pathlib import Path
import csv
from datetime import datetime

import matplotlib.pyplot as plt

path = Path('weather_data/sitka_weather_07-2021_simple.csv')
lines = path.read_text().splitlines()

reader = csv.reader(lines)
header_row = next(reader)

Извлечение дат и максимальных температур.
❶ dates, highs = [], []
for row in reader:
 ❷ current_date = datetime.strptime(row[2], '%Y-%m-%d')
 high = int(row[4])
 dates.append(current_date)
 highs.append(high)

Создание диаграммы максимальных температур.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❸ ax.plot(dates, highs, color='red')

Форматирование диаграммы.
ax.set_title("Daily High Temperatures, July 2021", fontsize=24)
ax.set_xlabel('', fontsize=16)
❹ fig.autofmt_xdate()
ax.set_ylabel("Temperature (F)", fontsize=16)
ax.tick_params(labelsize=16)

plt.show()
```

Мы создаем два пустых списка для хранения дат и максимальных температур из файла ❶. Затем программа преобразует данные, содержащие информацию даты (*row[2]*), в объект *datetime* ❷, который присоединяется к *dates*. Значения дат и максимальных температур передаются *plot()* в строке ❸. Вызов *fig.autofmt\_xdate()* ❹ выводит метки дат по диагонали, чтобы они не перекрывались. На рис. 16.2 изображена новая версия графика.

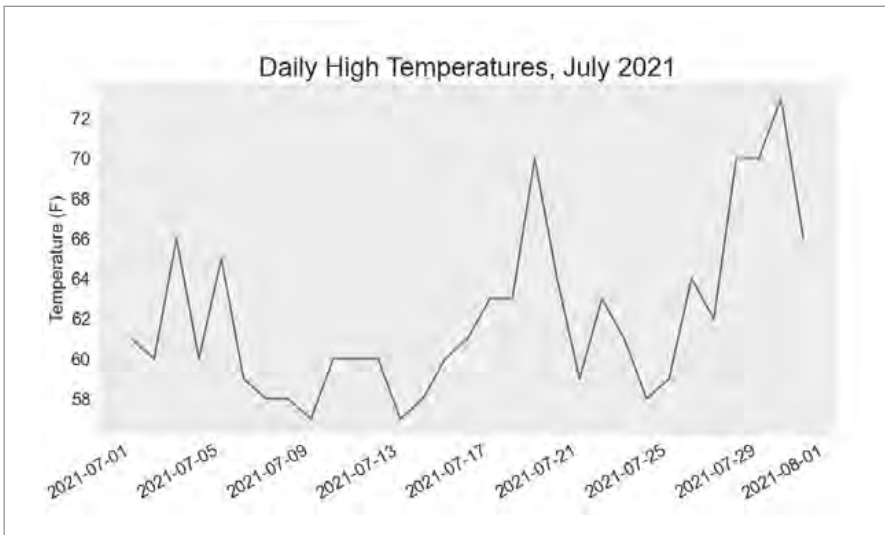


**Рис. 16.2.** График с датами на оси X стал более понятным

### Расширение временного диапазона

Итак, график успешно создан. Добавим на него новые данные для получения более полной картины погоды в Ситке. Скопируйте файл `sitka_weather_2021_simple.csv`, содержащий погодные данные для Ситки за целый год, в папку с программами этой главы.

А теперь мы можем сгенерировать график с погодными данными за год:

#### *sitka\_highs.py*

```
--пропуск--
path = Path('weather_data/sitka_weather_2021_simple.csv')
lines = path.read_text().splitlines()
--пропуск--
Форматирование диаграммы.
ax.set_title("Daily High Temperatures, 2021", fontsize=24)
ax.set_xlabel('', fontsize=16)
--пропуск--
```

Значение `filename` было изменено, чтобы в программе использовался новый файл данных `sitka_weather_2021_simple.csv`, а заголовок диаграммы приведен в соответствие с содержанием. На рис. 16.3 изображена полученная диаграмма.

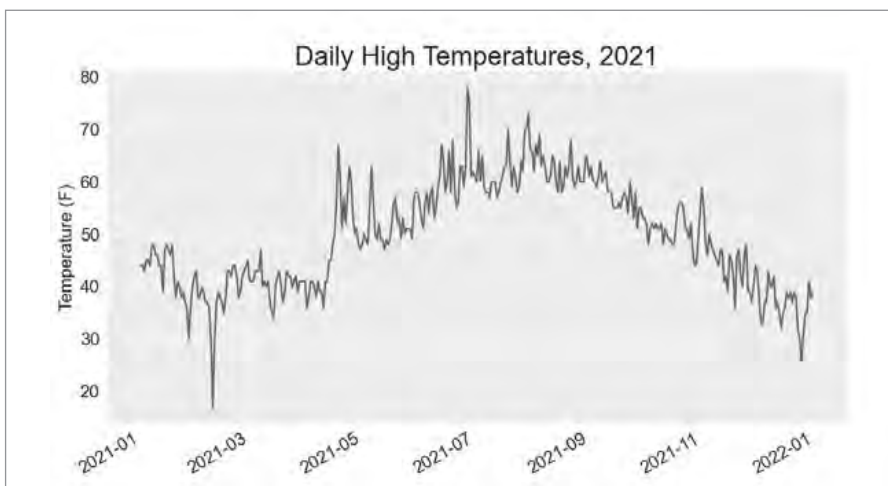


Рис. 16.3. Данные за год

### Добавление в диаграмму второго набора данных

График можно сделать еще полезнее, добавив в него данные минимальных температур. Нам нужно извлечь информацию о низких температурах из файла данных, а затем добавить ее в наш график:

#### *sitka\_highs\_lows.py*

```
--пропуск--
```

```
reader = csv.reader(lines)
```

```
header_row = next(reader)
```

```
Извлечение дат, минимальных и максимальных температур из файла.
```

```
❶ dates, highs, lows = [], [], []
```

```
for row in reader:
```

```
 current_date = datetime.strptime(row[2], '%Y-%m-%d')
```

```
 high = int(row[4])
```

```
❷ low = int(row[5])
```

```
 dates.append(current_date)
```

```
 highs.append(high)
```

```
 lows.append(low)
```

```
Создание диаграммы высоких и низких температур.
```

```
plt.style.use('seaborn')
```

```
fig, ax = plt.subplots()
```

```
ax.plot(dates, highs, color='red')
```

```
❸ ax.plot(dates, lows, color='blue')
```

```
Форматирование диаграммы.
```

```
❹ ax.set_title("Daily High and Low Temperatures, 2021", fontsize=24)
```

```
--пропуск--
```



Сначала создается пустой список `lows` для хранения минимальных температур ❶, после чего программа извлекает и сохраняет температурный минимум для каждой даты из шестой позиции каждой строки данных (`row[5]`) ❷. Далее добавляется вызов `plot()` для минимальных температур, которые окрашиваются в синий цвет ❸. Затем остается лишь обновить заголовок диаграммы ❹. На рис. 16.4 изображена полученная диаграмма.

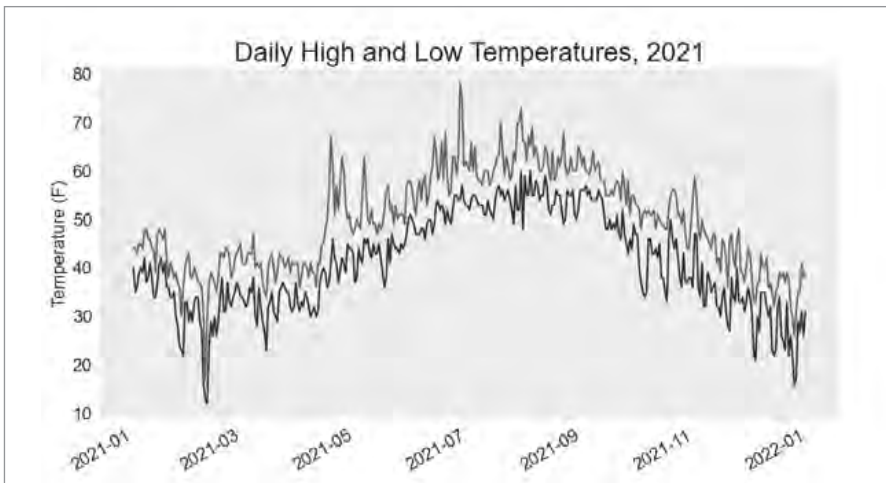


Рис. 16.4. Два набора данных на одной диаграмме

### Цветовое выделение частей диаграммы

После добавления двух серий данных можно переходить к анализу диапазона температур по дням. Пора сделать последний штрих в оформлении диаграммы: затушевать диапазон между минимальной и максимальной дневной температурой. Для этого мы воспользуемся методом `fill_between()`, который получает серию значений `x`, две серии значений `y` и заполняет область между двумя значениями `y`:

#### *sitka\_highs\_lows.py*

--пропуск--

# Создание диаграммы высоких и низких температур.

`plt.style.use('seaborn')`

`fig, ax = plt.subplots()`

❶ `ax.plot(dates, highs, color='red', alpha=0.5)`

`ax.plot(dates, lows, color='blue', alpha=0.5)`

❷ `ax.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)`

--пропуск--

Аргумент `alpha` определяет степень прозрачности вывода ❶. Значение 0 говорит о полной прозрачности, а 1 (по умолчанию) — о полной непрозрачности. Со значением `alpha=0.5` красные и синие линии на графике становятся более светлыми.

Затем `fill_between()` передается список `dates` для значений  $x$  и две серии значений  $y$  `highs` и `lows` ❷. Аргумент `facecolor` определяет цвет закрашиваемой области; мы задаем ему низкое значение `alpha=0.1`, чтобы заполненная область соединяла две серии данных, не отвлекая зрителя от передаваемой информации. На рис. 16.5 показана диаграмма с закрашенной областью между `highs` и `lows`.

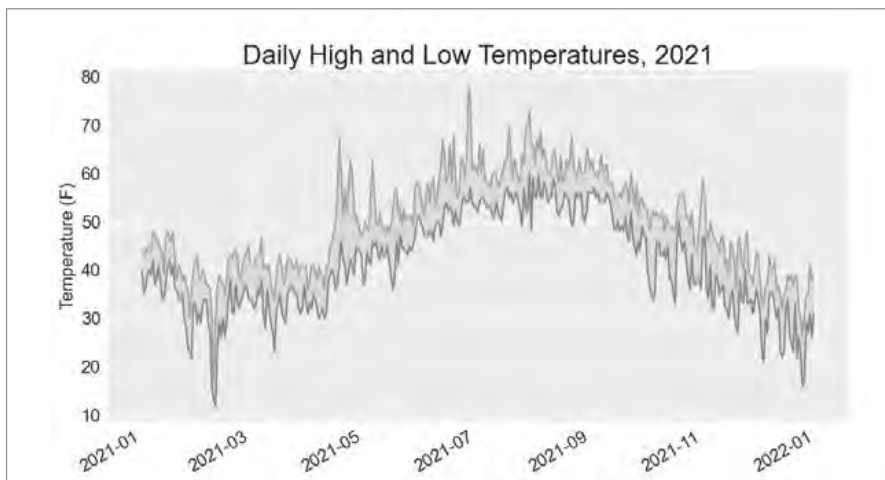


**Рис. 16.5.** Область между двумя наборами данных закрашена

Закрашенная область подчеркивает величину расхождения между двумя наборами данных.

## Проверка ошибок

Программа `sitka_highs_lows.py` должна нормально работать для погодных данных любого места. Однако одни метеорологические станции собирают данные по особым правилам, а другим не удастся собрать данные из-за сбоев (полных или частичных). Отсутствие данных может привести к исключениям; если последние не обработать, то программа завершится со сбоем.

Для примера попробуем создать диаграмму температур для Долины Смерти (штат Калифорния). Скопируйте файл `death_valley_2021_simple.csv` в папку с программами этой главы.

Сначала выполните следующий код, чтобы просмотреть состав заголовков из файла данных:

### *death\_valley\_highs\_lows.py*

```
from pathlib import Path
import csv
```

```
path = Path('weather_data/death_valley_2021_simple.csv')
lines = path.read_text().splitlines()

reader = csv.reader(lines)
header_row = next(reader)

for index, column_header in enumerate(header_row):
 print(index, column_header)
```

Результат выглядит так:

```
0 STATION
1 NAME
2 DATE
3 TMAX
4 TMIN
5 TOBS
```

Дата остается в той же позиции с индексом 2. Но температурные максимумы и минимумы находятся в позициях с индексами 3 и 4, поэтому нам придется изменить индексы в программе в соответствии с новыми позициями. Вместо того чтобы добавлять средние показания температуры за день, эта станция регистрирует **TOBS** — данные за конкретное время наблюдений.

Внесите изменения в `sitka_highs_lows.py`, чтобы создать график температур для Долины Смерти по только что определенным индексам, и проследите за происходящим:

### ***death\_valley\_highs\_lows.py***

```
--пропуск--
path = Path('weather_data/death_valley_2021_simple.csv')
lines = path.read_text().splitlines()
--пропуск--
Извлечение дат, минимальных и максимальных температур из файла.
dates, highs, lows = [], [], []
for row in reader:
 current_date = datetime.strptime(row[2], '%Y-%m-%d')
 high = int(row[3])
 low = int(row[4])
 dates.append(current_date)
--пропуск--
```

Код изменен так, чтобы программа считывала данные из файла с погодой в Долине Смерти; изменены и индексы, чтобы соответствовать позициям **TMAX** и **TMIN** этого файла.

При запуске программы происходит ошибка:

```
Traceback (most recent call last):
 File "death_valley_highs_lows.py", line 17, in <module>
 high = int(row[3])
```

❶ **ValueError: invalid literal for int() with base 10: ''**

В трассировке указано, что Python не сможет обработать максимальную температуру для одной из дат, поскольку не сумеет преобразовать пустую строку ( ' ') в целое число ❶. Вместо того чтобы копаться в данных и искать отсутствующее значение, мы напрямую обрабатываем ситуации с отсутствием данных.

При чтении данных из CSV-файла будет выполняться код проверки ошибок для обработки исключений, которые могут возникнуть при разборе наборов данных. Вот как это делается:

### *death\_valley\_highs\_lows.py*

```
--пропуск--
for row in reader:
 current_date = datetime.strptime(row[2], '%Y-%m-%d')
 ❶ try:
 high = int(row[3])
 low = int(row[4])
 except ValueError:
 ❷ print(f"Missing data for {current_date}")
 ❸ else:
 dates.append(current_date)
 highs.append(high)
 lows.append(low)

Создание диаграммы высоких и низких температур.
--пропуск--

Форматирование диаграммы.
❹ title = "Daily High and Low Temperatures, 2021\nDeath Valley, CA"
ax.set_title(title, fontsize=20)
ax.set_xlabel('', fontsize=16)
--пропуск--
```

При анализе каждой строки данных мы пытаемся извлечь дату, максимальную и минимальную температуру ❶. Если каких-либо данных не хватает, то Python выдает ошибку `ValueError`, а мы обрабатываем ее — выводим сообщение с датой, для которой отсутствуют данные ❷. После вывода ошибки цикл продолжает обработку следующей порции данных. Если все данные, относящиеся к некоторой дате, прочитаны без ошибок, то выполняется блок `else`, а данные присоединяются к соответствующим спискам ❸. На диаграмме отображается информация для нового места, поэтому заголовок изменяется, в него добавляется название места, а для вывода длинного заголовка используется уменьшенный шрифт ❹.

При выполнении `death_valley_highs_lows.py` мы видим, что данные отсутствуют только для одной даты:

```
Missing data for 2021-05-04 00:00:00
```

Ошибка была обработана корректно, поэтому наш код может сгенерировать диаграмму, в которой пропущены отсутствующие данные. Полученная диаграмма изображена на рис. 16.6.

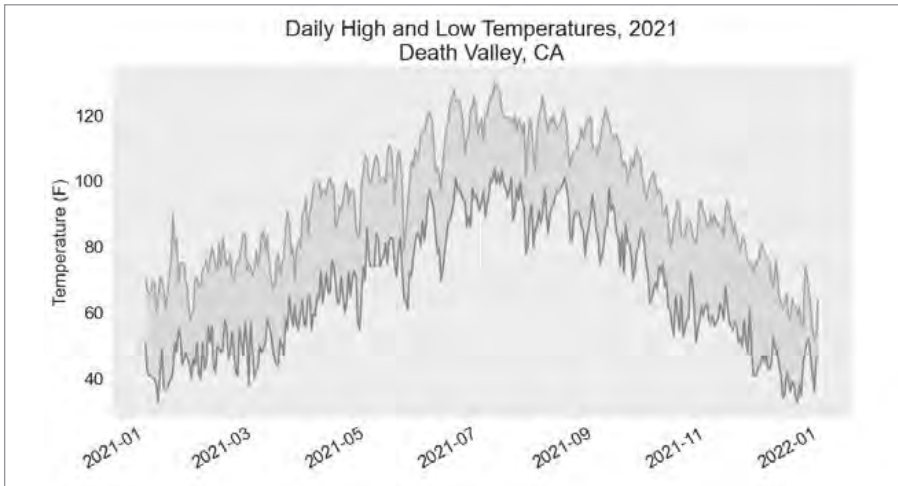


**Рис. 16.6.** Максимальная и минимальная температуры в Долине Смерти

Сравнивая эту диаграмму с диаграммой для Ситки, мы видим, что в Долине Смерти теплее, чем на юго-востоке Аляски (как и следовало ожидать), но при этом температурный диапазон в пустыне более широкий. Высота закрашенной области наглядно демонстрирует этот факт.

Во многих наборах данных в реальной работе будут встречаться отсутствующие, неправильно отформатированные или некорректные данные. В таких ситуациях воспользуйтесь теми инструментами, которые вы освоили, изучая первую половину книги. В данном примере для обработки отсутствующих данных использовался блок `try-except-else`. Иногда оператор `continue` применяется для пропуска части данных, или же данные удаляются после извлечения путем вызова метода `remove()` или оператора `del`. Используйте любое работающее решение — лишь бы в результате у вас получилась имеющая смысл точная визуализация.

### Скачивание собственных данных

Если вы предпочитаете скачать собственные погодные данные, то выполните следующие действия.

1. Посетите сайт NOAA Climate Data Online по адресу <https://www.ncdc.noaa.gov/cdo-web/>. В разделе Discover Data (Поиск данных) нажмите кнопку Search Tool

- (Инструменты поиска). В поле **Select a Dataset** (Выбор набора данных) выберите вариант **Daily Summaries** (Ежедневные сводки).
2. Выберите диапазон дат. В разделе **Search For** (Поиск) выберите вариант **ZIP Codes** (Почтовые индексы). Введите индекс интересующего вас места и нажмите кнопку **Search** (Поиск).
  3. На следующей странице отображаются карта и информация об области, на которой вы желаете сосредоточиться. Под названием места нажмите кнопку **View Full Details** (Просмотреть полную информацию) либо на карте, а затем выберите вариант **Full Details** (Полная информация).
  4. Прокрутите данные и нажмите кнопку **Station List** (Список станций), чтобы просмотреть список метеорологических станций в этой области. Выберите одну из станций и нажмите кнопку **Add to Cart** (Добавить на карту). Данные распространяются бесплатно, несмотря на то что на сайте используется обозначение покупательской корзины. Щелкните на изображении корзины в правом верхнем углу.
  5. Выберите **Output** (Выходные данные), затем **Custom GHCN-Daily CSV** (Пользовательский файл CSV GHCN-Daily). Проверьте на правильность диапазон дат и нажмите кнопку **Continue** (Продолжить).
  6. На следующей странице вы можете выбрать нужные разновидности данных. Например, можно скачать один тип данных, ограничившись температурой воздуха, или же все данные, собираемые станцией. Выберите нужный вариант и нажмите кнопку **Continue** (Продолжить).
  7. На последней странице выводится сводка запроса. Введите свой адрес электронной почты и нажмите кнопку **Submit Order** (Подтвердить запрос). Вы получите подтверждение запроса, а через несколько минут придет другое сообщение электронной почты со ссылкой для скачивания данных.

Скачанные данные будут иметь такую же структуру, как и данные, с которыми вы работали в этом разделе. Их заголовки могут отличаться от представленных в этом разделе, но если вы последуете описанной процедуре, то сможете создать визуализации интересующих вас данных.

## УПРАЖНЕНИЯ

**16.1. Осадки в Ситке.** Ситка находится в зоне умеренных лесов, так что в этой местности выпадает достаточно осадков. В файле данных `sitka_weather_2021_full.csv` есть заголовок `PRCP`, представляющий величину ежедневных осадков. Создайте диаграмму по данным этого столбца. Повторите упражнение для Долины Смерти, если вас интересует, сколько осадков выпадает в пустыне.

**16.2. Сравнение Ситки с Долиной Смерти.** Разные масштабы температур отражают разные диапазоны данных. Чтобы точно сравнить температурный диапазон в Ситке с температурным диапазоном Долины Смерти, необходимо установить одинаковый масштаб по оси  $Y$ . Измените параметры оси  $Y$  для одной или обеих диаграмм на рис. 16.5 и 16.6 и проведите прямое сравнение температурных диапазонов в этих двух местах (или любых других, которые вас интересуют).

**16.3. Сан-Франциско.** К какому месту ближе температура в Сан-Франциско: к Ситке или Долине Смерти? Скачайте данные для этого города, создайте температурную диаграмму для него и сравните.

**16.4. Автоматические индексы.** В этом разделе индексы, соответствующие столбцам `TMIN` и `TMAX`, были жестко зафиксированы в коде. Используйте строку данных заголовка для определения индексов этих значений, чтобы ваша программа работала как для Ситки, так и Долины Смерти. Используйте название станции, чтобы автоматически сгенерировать подходящий заголовок для вашей диаграммы.

**16.5. Исследования.** Создайте еще несколько визуализаций, отражающих любые другие аспекты погоды для интересующих вас мест.

## Создание карт с глобальными наборами данных: формат GeoJSON

В этом разделе вы скачаете данные о землетрясениях, произошедших в мире за последний месяц. Затем создадите карту, на которой будут обозначены места этих землетрясений и уровень силы каждого из них. Данные хранятся в формате GeoJSON, поэтому для работы с ними будет использован модуль `json`. С помощью удобных функций `Plotly scatter_geo()` вы создадите визуализацию, отражающую глобальное распределение землетрясений.

### Скачивание данных о землетрясениях

Создайте папку `eq_data` в папке, в которой хранятся программы для этой главы. Скопируйте файл `q_1_day_m1.geojson` в эту новую папку. Землетрясения классифицируются по магнитуде на основании шкалы Рихтера. Файл содержит данные по всем землетрясениям с магнитудой  $M1$  и выше, произошедшим за последние 24 часа (на момент написания книги). Информация взята из одного из каналов данных Геологического управления США, доступных по адресу <https://earthquake.usgs.gov/earthquakes/feed/>.

## Знакомство с форматом GeoJSON

Открыв файл `eq_1_day_m1.json`, вы увидите, что данные упакованы очень плотно и читать их сложно:

```
{ "type": "FeatureCollection", "metadata": { "generated": 1649052296000, ...
{ "type": "Feature", "properties": { "mag": 1.6, "place": "63 km SE of Ped...
{ "type": "Feature", "properties": { "mag": 2.2, "place": "27 km SSE of Ca...
{ "type": "Feature", "properties": { "mag": 3.7, "place": "102 km SSE of S...
{ "type": "Feature", "properties": { "mag": 2.92000008, "place": "49 km SE...
{ "type": "Feature", "properties": { "mag": 1.4, "place": "44 km NE of Sus...
--пропуск--
```

Формат этого файла больше предназначен для машин, чем для людей. Но мы видим, что файл содержит словари, а также информацию, которая нас интересует, в том числе магнитуды и местоположение землетрясений.

Модуль `json` предоставляет разнообразные инструменты для анализа и обработки данных JSON. Некоторые из этих инструментов позволяют переформатировать файл, чтобы вам было удобнее взаимодействовать с необработанными данными, прежде чем вы начнете работать с ними на программном уровне.

Для начала скачаем данные и выведем их в формате, лучше подходящем для чтения. Файл очень длинный, поэтому вместо того, чтобы выводить его, данные будут записаны в новый файл. После этого вы сможете открыть его и прокрутить к нужной позиции:

### *eq\_explore\_data.py*

```
from pathlib import Path
import json
```

```
Считывает данные в строковом формате и преобразует в объект Python.
path = Path('eq_data/eq_data_1_day_m1.geojson')
contents = path.read_text()
```

```
❶ all_eq_data = json.loads(contents)
```

```
Создает удобную для чтения версию файла данных.
```

```
❷ path = Path('eq_data/readable_eq_data.geojson')
```

```
❸ readable_contents = json.dumps(all_eq_data, indent=4)
path.write_text(readable_contents)
```

Мы считываем данные из файла в строковом формате и вызываем функцию `json.loads()` для преобразования строкового представления файла в объект Python ❶. Такой же подход мы использовали в главе 10. В этом случае весь набор данных преобразуется в один словарь, который мы присваиваем переменной `all_eq_data`. Затем мы определяем новую переменную `path`, в которой можем сохранить полученные данные в формате, более удобном для чтения ❷. Функция `json.dumps()`, которая упоминалась в главе 10, поддерживает необязательный аргумент `indent` ❸, позволяющий указать размер отступа вложенных элементов в структуре данных.



Перейдите в каталог `eq_data` и откройте файл `readable_eq_data.json`. Начальная часть выглядит так:

```
readable_eq_data.json
{
 "type": "FeatureCollection",
 ❶ "metadata": {
 "generated": 1649052296000,
 "url": "https://earthquake.usgs.gov/earthquakes/.../1.0_day.geojson",
 "title": "USGS Magnitude 1.0+ Earthquakes, Past Day",
 "status": 200,
 "api": "1.10.3",
 "count": 160
 },
 ❷ "features": [
 --пропуск--
```

В первую часть файла включена секция с ключом `"metadata"` ❶. По ней можно определить, когда файл был сгенерирован и где можно найти данные в Интернете. Кроме того, в ней содержится понятный заголовок и количество землетрясений, внесенных в файл. За этот 24-часовой период было зарегистрировано **160** землетрясений.

Структура файла GeoJSON хорошо подходит для географических данных. Информация хранится в списке, связанном с ключом `"features"` ❷. В файле содержится информация о землетрясениях, так что эти данные имеют форму списка, в котором каждый элемент соответствует одному землетрясению. На первый взгляд структура кажется запутанной, но она весьма полезна. Например, геолог может сохранить в словаре столько информации о каждом землетрясении, сколько потребуется, а затем объединить все словари в один большой список.

Рассмотрим словарь, представляющий одно землетрясение:

```
readable_eq_data.json
--пропуск--
{
 ❶ "type": "Feature",
 "properties": {
 "mag": 1.6,
 --пропуск--
 ❷ "title": "M 1.6 - 27 km NNW of Susitna, Alaska"
 },
 ❸ "geometry": {
 "type": "Point",
 "coordinates": [
 ❹ -150.7585,
 ❺ 61.7591,
 56.3
]
 },
 "id": "ak0224bju1jx"
 },
 },
```

Ключ "properties" содержит подробную информацию о каждом землетрясении ❶. Нас прежде всего интересует их магнитуда, связанная с ключом "mag". Представляет интерес и заголовок каждого землетрясения, содержащий удобную сводку магнитуды и координат ❷.

Ключ "geometry" помогает определить, где произошло землетрясение ❸. Эта информация потребуется для географической привязки событий. Долгота ❹ и широта ❺ каждого землетрясения содержатся в списке, связанном с ключом "coordinates".

Уровень вложенности в этом коде намного выше, чем мы использовали бы в своем коде, и если он покажется запутанным — не огорчайтесь; Python берет на себя обработку большей части сложности. В любой момент времени мы будем работать с одним или двумя уровнями. Мы начнем с извлечения словаря для каждого землетрясения, зарегистрированного за 24-часовой период.

### ПРИМЕЧАНИЕ

В географических координатах часто сначала указывается широта, а затем долгота. Вероятно, эта система обозначений возникла из-за того, что люди открыли широту задолго до того, как была создана концепция долготы. Тем не менее во многих геопространственных библиотеках сначала указывается долгота, а потом широта, поскольку этот порядок соответствует системе обозначений (x, y), используемой в математических представлениях. Формат GeoJSON использует систему записи (долгота, широта), и если вы будете работать с другой библиотекой, то очень важно заранее узнать, какая система в ней предусмотрена.

## Создание списка всех землетрясений

Начнем с создания списка, содержащего всю информацию обо всех произошедших землетрясениях.

### *eq\_explore\_data.py*

```
from pathlib import Path
import json
```

```
Чтение данных в строковом формате и преобразование в объект Python.
path = Path('eq_data/eq_data_1_day_m1.geojson')
contents = path.read_text()
all_eq_data = json.loads(contents)
```

```
Обработка всех землетрясений в наборе данных.
all_eq_dicts = all_eq_data['features']
print(len(all_eq_dicts))
```

Мы берем данные, связанные с ключом 'features', и сохраняем их в `all_eq_data`. Известно, что файл содержит данные 160 землетрясений, а вывод подтверждает, что были прочитаны данные всех землетрясений:

Обратите внимание на то, каким коротким получился код. Отформатированный файл `readable_eq_data.json` содержит более 6000 строк. Всего в нескольких строках кода мы прочитали все данные и сохранили их в списке Python. Теперь извлечем данные магнитуд по каждому землетрясению.

## Извлечение магнитуд

Имея список, содержащий данные по всем землетрясениям, мы можем перебрать содержимое списка и извлечь всю необходимую информацию. В данном случае это будет магнитуда каждого землетрясения:

### `eq_explore_data.py`

```
--пропуск--
all_eq_dicts = all_eq_data['features']
```

```
❶ mags = []
 for eq_dict in all_eq_dicts:
❷ mag = eq_dict['properties']['mag']
 mags.append(mag)
```

```
print(mags[:10])
```

Создадим пустой список для хранения магнитуд, а затем переберем в цикле словарь `all_eq_dicts` ❶. Внутри цикла каждое землетрясение представляется словарем `eq_dict`. Магнитуда каждого землетрясения хранится в секции `'properties'` словаря с ключом `'mag'` ❷. Каждая магнитуда сохраняется в переменной `mag` и присоединяется к списку `mags`.

Выведем первые десять магнитуд, чтобы убедиться в том, что были получены правильные данные:

```
[1.6, 1.6, 2.2, 3.7, 2.92000008, 1.4, 4.6, 4.5, 1.9, 1.8]
```

Далее мы извлечем данные местоположения (то есть координаты) для каждого землетрясения, а затем создадим карту землетрясений.

## Извлечение данных о местоположении

Данные о местоположении хранятся с ключом `"geometry"`. В словаре `geometry` есть ключ `"coordinates"`, первыми двумя значениями которого являются долгота и широта. Извлечение данных происходит следующим образом:

### `eq_explore_data.py`

```
--пропуск--
all_eq_dicts = all_eq_data['features']
```

```
mags, lons, lats = [], [], []
for eq_dict in all_eq_dicts:
```

```

mag = eq_dict['properties']['mag']
❶ lon = eq_dict['geometry']['coordinates'][0]
lat = eq_dict['geometry']['coordinates'][1]
mags.append(mag)
lons.append(lon)
lats.append(lat)

print(mags[:10])
print(lons[:5])
print(lats[:5])

```

Для долгот и широт создаются пустые списки. Выражение `eq_dict['geometry']` обращается к словарию, представляющему элемент `geometry` данных землетрясения ❶. Второй ключ `'coordinates'` извлекает список значений, связанных с ключом `'coordinates'`. Наконец, индекс 0 запрашивает первое значение в списке координат, соответствующее долготе землетрясения.

При выводе первых пяти долгот и широт становится видно, что данные были извлечены правильно:

```

[1.6, 1.6, 2.2, 3.7, 2.92000008, 1.4, 4.6, 4.5, 1.9, 1.8]
[-150.7585, -153.4716, -148.7531, -159.6267, -155.248336791992]
[61.7591, 59.3152, 63.1633, 54.5612, 18.7551670074463]

```

Имея эти данные, можно переходить к нанесению координат землетрясений на географическую карту.

## Создание карты мира

На основании всей информации, собранной к настоящему моменту, можно создать простую карту мира. И хотя первая версия будет выглядеть довольно простой, нужно убедиться в том, что информация отображается правильно, прежде чем сосредоточиться на стиле и визуальном оформлении. Исходная карта выглядит так:

### *eq\_world\_map.py*

```

from pathlib import Path
import json

import plotly.express as px

--пропуск--
for eq_dict in all_eq_dicts:
 --пропуск--

title = 'Global Earthquakes'
❶ fig = px.scatter_geo(lat=lats, lon=lons, title=title)
fig.show()

```

Мы импортируем модуль `plotly.express` под псевдонимом `px`, как и в главе 15. Функция `scatter_geo()` <sup>❶</sup> позволяет наложить на карту диаграмму разброса географических данных. В простейшем варианте диаграммы вам нужно предоставить только список широт и долгот. Мы передаем список `lats` в качестве аргумента `lat`, а `lons` — в качестве аргумента `lon`.

При выполнении этого файла должна открыться карта, примерный вид которой показан на рис. 16.7. Этот пример еще раз демонстрирует возможности библиотеки Plotly Express: используя всего три строки кода, мы получили глобальную карту активности землетрясений.

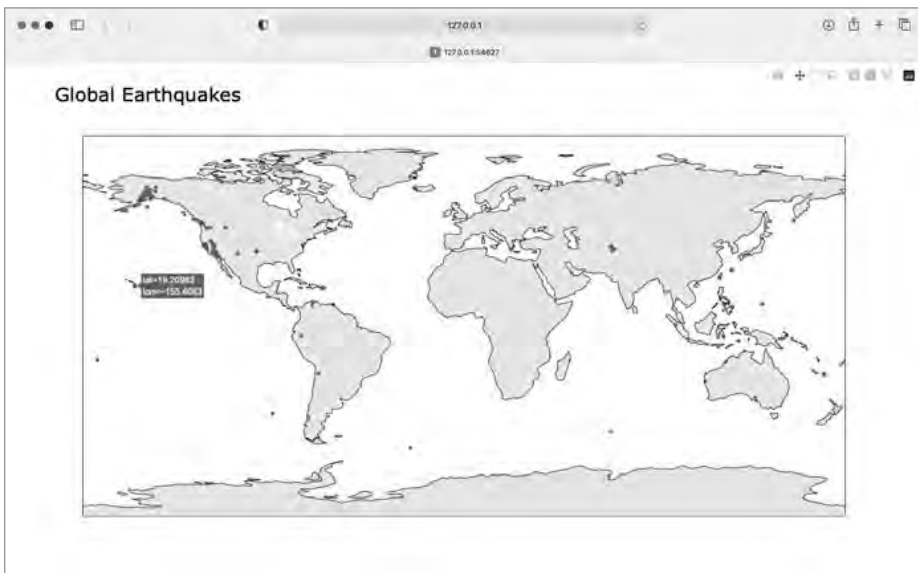


**Рис. 16.7.** Простая карта с информацией о землетрясениях, произошедших за последние 24 часа

Теперь, выяснив, что информация в нашем наборе данных выводится правильно, мы можем внести несколько изменений, чтобы карта стала более информативной и удобочитаемой.

### Представление магнитуд землетрясений

На карте землетрясений должна быть указана магнитуда каждого из них. Мы также можем отобразить больше данных, так как информация наносится правильно.

--пропуск--

# Чтение данных в строковом формате и преобразование в объект Python.

```
path = Path('eq_data/eq_data_30_day_m1.geojson')
contents = path.read_text()
--пропуск--

title = 'Global Earthquakes'
fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title)
fig.show()
```

Мы загружаем файл `eq_data_30_day_m1.geojson`, чтобы добавить на карту данные о землетрясениях за полные 30 дней. Мы также используем аргумент `size` в вызове функции `px.scatter_geo()`, чтобы изменять размер точек на карте. С помощью `size` мы передаем список магнитуд `mags`, и теперь землетрясения с большей магнитудой будут отображаться на карте в виде более крупной точки.

Итоговая карта показана на рис. 16.8. Землетрясения обычно происходят вблизи краев тектонических плит, и при анализе землетрясений за более длительный период видно точное расположение этих краев.

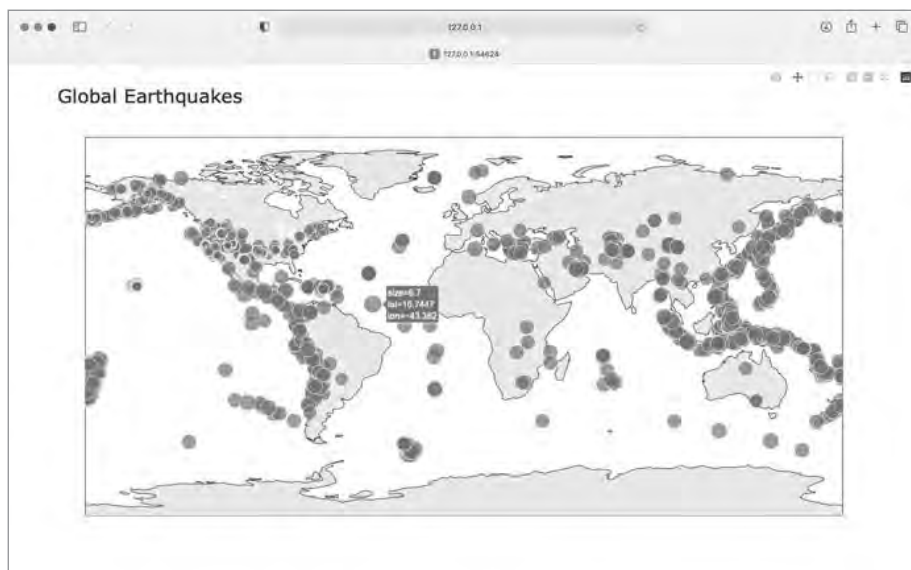


**Рис. 16.8.** Теперь на карте отображаются магнитуды всех землетрясений за последние 30 дней

Карта стала более удобной, но все еще непонятно, какова сила того или иного землетрясения. Мы можем улучшить карту, меняя цвет точек для обозначения магнитуды.

## Настройка цвета точек на карте

С помощью Plotly мы можем настроить цвет каждого маркера в зависимости от магнитуды соответствующего землетрясения. Мы также изменим проекцию для самой карты.

### *eq\_world\_map.py*

--пропуск--

```
fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title,
① color=mags,
② color_continuous_scale='Viridis',
③ labels={'color': 'Magnitude'},
④ projection='natural earth',
)
fig.show()
```

Все существенные изменения вносятся в вызов функции `px.scatter_geo()`. Аргумент `color` определяет, какие значения цветовой шкалы следует использовать для окрашивания каждого маркера ❶. Мы используем список `mags` для определения цвета каждой точки, как и в случае с аргументом `size`.

Аргумент `color_continuous_scale` ссылается на используемую цветовую шкалу ❷. `Viridis` — это цветовая шкала с оттенками от темно-синего до ярко-желтого, которая хорошо подходит для нашего набора данных. По умолчанию цветовая шкала в правой части карты сопровождается меткой `color`; это слово не отражает предназначение цветовой дифференциации. Аргумент `labels`, продемонстрированный впервые в главе 15, принимает словарь в качестве значения ❸. Нам нужна только одна пользовательская метка на нашей карте, чтобы цветовая шкала была обозначена `Magnitude` вместо `color`.

Мы использовали еще один аргумент, меняющий карту землетрясений. Аргумент `projection` принимает одну из распространенных картографических проекций ❹. В нашем примере мы используем проекцию `'natural earth'`, которая округляет края карты. Обратите также внимание на запятую после этого аргумента. Если вызов функции содержит длинный список аргументов, занимающий несколько строк, то обычно в конце ставят запятую, чтобы при необходимости добавить еще один аргумент на следующей строке.

Если запустить программу в этой версии, то карта будет выглядеть намного привлекательнее. На рис. 16.9 цветовая шкала обозначает разрушительность отдельных землетрясений; самые мощные землетрясения выделяются светло-желтыми точками на фоне множества более темных точек. Вы также можете определить, в каких регионах мира землетрясения происходят чаще.

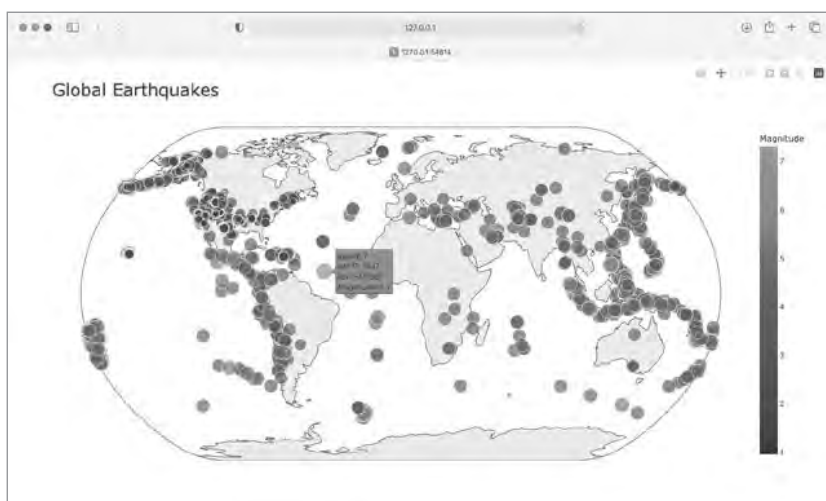


**Рис. 16.9.** Цвета и размеры маркеров представляют магнитуду землетрясений за последние 30 дней

## Другие цветовые шкалы

Для оформления диаграммы можно выбрать другую цветовую шкалу. Чтобы посмотреть доступные варианты цветовых шкал, выполните следующие две строки кода в терминальной сессии Python:

```
>>> import plotly.express as px
>>> px.colors.named_colorscale()
['aggrnyl', 'aggsunset', 'blackbody', ..., 'mygbm']
```

Попробуйте применить эти цветовые шкалы к карте землетрясений или к любому другому набору данных, где оттенки цвета помогут выявить закономерности в данных.

## Добавление подсказки

Чтобы закончить создание карты, мы добавим подсказку, которая будет появляться при наведении указателя мыши на маркер, обозначающий землетрясение. Помимо значений долготы и широты, которые должны выводиться по умолчанию, мы выведем магнитуду и описание приблизительного местоположения.

Для этого нужно извлечь из файла еще немного данных:

**eq\_world\_map.py**

--пропуск--

```
❶ mags, lons, lats, eq_titles = [], [], [], []
 mag = eq_dict['properties']['mag']
```



```

lon = eq_dict['geometry']['coordinates'][0]
lat = eq_dict['geometry']['coordinates'][1]
❷ eq_title = eq_dict['properties']['title']
mags.append(mag)
lons.append(lon)
lats.append(lat)
eq_titles.append(eq_title)

title = 'Global Earthquakes'
fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title,
--пропуск--
projection='natural earth',
❸ hover_name=eq_titles,
)
fig.show()

```

Сначала мы создаем список `eq_titles` для хранения названий всех землетрясений ❶. Раздел `'title'` содержит описательное название магнитуды и местоположения каждого землетрясения, а также его долготу и широту. Мы извлекаем эту информацию и присваиваем переменной `eq_title` ❷, а затем добавляем ее в список `eq_titles`.

В вызове `px.scatter_geo()` мы передаем содержимое переменной `eq_titles` в качестве аргумента `hover_name` ❸. Теперь Plotly добавит информацию о каждом землетрясении в текст подсказки для каждой точки. Запустив готовую программу, вы сможете навести указатель мыши на любой маркер, увидеть описание местоположения землетрясения и узнать его точную магнитуду. Пример показан на рис. 16.10.

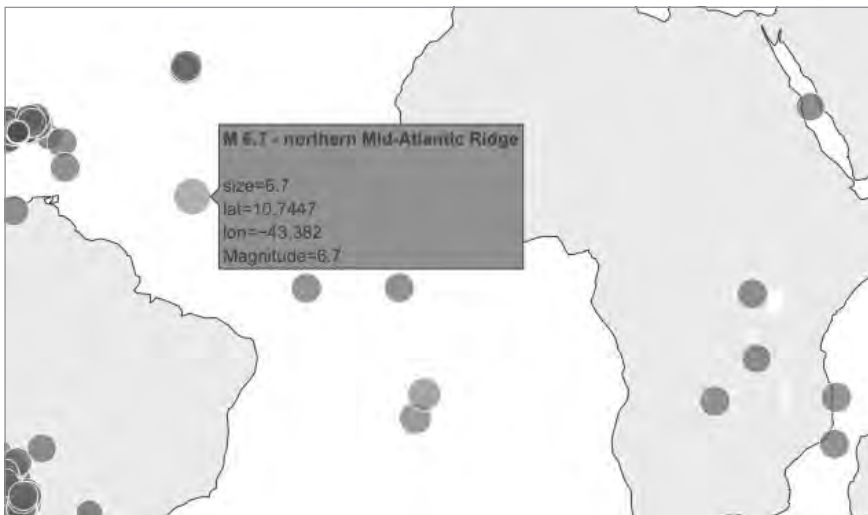


**Рис. 16.10.** Подсказка, появляющаяся при наведении указателя мыши, содержит краткое описание каждого землетрясения

Впечатляет! Используя менее 30 строк кода, мы создали визуально приятную и содержательную карту глобальной сейсмической активности, которая к тому же демонстрирует геологическую структуру планеты. Plotly предоставляет многочисленные средства настройки оформления и поведения ваших визуализаций. С их помощью вы сможете создавать диаграммы и карты, содержащие именно ту информацию, которая вам нужна.

## УПРАЖНЕНИЯ

**16.6. Рефакторинг.** В цикле, извлекающем данные из словаря `all_eq_dicts`, используются переменные для сохранения магнитуды, долготы, широты и заголовка каждого землетрясения перед присоединением этих значений к соответствующим спискам. Такой подход был выбран для того, чтобы процесс извлечения данных из файла GeoJSON был более понятным, но в вашем коде он необязателен. Вместо того чтобы использовать временные переменные, извлеките каждое значение из словаря `eq_dict` и присоедините его к соответствующему списку в одной строке. В результате тело цикла сократится до четырех строк.

**16.7. Автоматизированный заголовок.** В этом разделе мы использовали общий заголовок `Global Earthquakes`. Вместо этого можно воспользоваться заголовком набора данных из метаданных файла GeoJSON. Извлеките это значение и присвойте его переменной `title`.

**16.8. Недавние землетрясения.** В Интернете доступны файлы данных с информацией о последних землетрясениях за 1-часовой, 1-дневный, 7-дневный и 30-дневный период. Откройте страницу <https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php> и найдите список ссылок на наборы данных за разные периоды времени. Скачайте один из этих наборов и создайте визуализацию последней сейсмической активности.

**16.9. Пожары.** В дополнительных материалах к этой главе есть файл `world_fires_1_day.csv`. Он содержит информацию о пожарах по всему миру, в том числе долготу, широту и площадь каждого пожара. Используя процедуру обработки данных из первой части этой главы и картографические средства из этого раздела, создайте карту с информацией о том, в каких частях мира происходят пожары.

Обновленные версии этих данных можно скачать по адресу <https://earthdata.nasa.gov/earth-observation-data/near-real-time/irms/active-fire-data/>. Ссылки на данные в формате CSV находятся в разделах SHP, KML и TXT.

## Резюме

В этой главе вы научились работать с реальными наборами данных. Вы узнали, как обрабатывать файлы CSV и GeoJSON и как извлечь данные, на которых вы хотите сосредоточиться. На примере реальных погодных данных вы освоили новые возможности работы с библиотекой Matplotlib, такие как использование модуля `datetime` и добавление нескольких наборов данных в одну диаграмму. Вы узнали, как нанести данные на карту мира с помощью Plotly и как изменить оформление карт.

По мере накопления опыта взаимодействия с файлами CSV и JSON вы сможете обрабатывать практически любые данные, которые вам будет нужно анализировать. Многие сетевые наборы данных можно скачивать хотя бы в одном из этих форматов. После работы с этими форматами вам будет проще освоить и другие форматы данных.

В следующей главе вы напишете программы для автоматического сбора данных из сетевых источников, а затем создадите визуализации этих данных. Это занятие весьма интересное, если вы рассматриваете программирование как увлечение, и абсолютно необходимое, если программирование — ваша профессия.

# 17

## Работа с API



В этой главе вы научитесь писать специализированные программы для создания визуализаций на основании загруженных ими данных. Ваша программа будет использовать *программный интерфейс* (application programming interface, API) веб-приложения для автоматического запроса конкретной информации с сайта (вместо целых страниц), на базе которой будет создаваться визуализация. Программы, написанные по такой схеме, всегда используют самые свежие данные для создания визуализации, поэтому даже при быстро изменяющихся данных полученная диаграмма будет оставаться актуальной.

### Использование API веб-приложений

API веб-приложения представляет собой часть сайта, предназначенную для взаимодействия с программами, которые используют особым образом созданные URL для запроса информации. Подобные запросы называются *вызовами API* (API call). Запрашиваемые данные возвращаются в удобном формате (например, JSON или CSV). Многие приложения, зависящие от внешних источников данных (такие как приложения, интегрирующиеся с сайтами социальных сетей), используют вызовы API.

### Git и GitHub

Наша визуализация будет создана на базе информации с GitHub (<https://github.com/>) — сайта, организующего совместную работу программистов над проектами. С помощью API GitHub мы запросим информацию о проектах Python, а затем, используя Plotly, сгенерируем интерактивную визуализацию относительной популярности этих проектов.

Имя GitHub происходит от Git — распределенной системы управления версиями, которая позволяет программистам совместно трудиться над проектами.

Пользователи Git управляют своим кодом, который является их вкладом в проект, чтобы изменения, вносимые одним человеком, не конфликтовали с изменениями, вносимыми другими людьми. Когда вы реализуете новую возможность в проекте, Git отслеживает изменения, внесенные в каждый файл. Если новый код работает успешно, то вы *закрепляет*е (commit) внесенные изменения и Git записывает новое состояние проекта. Если же вы допустили ошибку и захотите отменить внесенные изменения, то Git позволяет легко вернуться к любому из предыдущих рабочих состояний. (Дополнительную информацию об управлении версиями с использованием Git см. в приложении Г.) Проекты GitHub хранятся в *репозиториях* (repositories), содержащих все ресурсы, связанные с проектом: код, информацию о других участниках, все проблемы или отчеты об ошибках и т. д.

Если проект нравится пользователям GitHub, то они могут отметить его звездочкой, чтобы продемонстрировать свою поддержку и следить за проектами, которые могут им пригодиться. В этой главе мы напишем программу для автоматического скачивания информации о проектах Python, отмеченных наибольшим количеством звезд на GitHub, а затем создадим информативную визуализацию таких проектов.

## Запрос данных с помощью вызовов API

Программный интерфейс GitHub позволяет запрашивать разнообразную информацию с помощью вызовов API. Чтобы понять, как выглядит такой вызов, введите в адресной строке своего браузера следующий адрес и нажмите клавишу Enter:

**`https://api.github.com/search/repositories?q=language:python+sort:stars`**

Этот вызов возвращает количество проектов Python, размещенных на GitHub в настоящее время, а также информацию о самых популярных репозиториях Python. Рассмотрим вызов подробнее: первая часть <https://api.github.com/> передает запрос части сайта GitHub, отвечающей на вызовы API. Следующая часть, `search/repositories`, дает API указание провести поиск по всем репозиториям в GitHub.

Вопросительный знак после `repositories` означает, что мы собираемся передать аргумент. Символ `q` обозначает *запрос* (query), а знак равенства начинает определение запроса (`q=`). Выражение `language:python` указывает, что запрашивается информация только по репозиториям, для которых основным языком указан Python. Завершающая часть, `+sort:stars`, сортирует проекты по количеству присвоенных им звезд.

В следующем фрагменте приведены несколько начальных строк ответа:

```
{
❶ "total_count": 8961993,
❷ "incomplete_results": true,
❸ "items": [
 {
 "id": 54346799,
 "node_id": "MDEwO1JlCG9zaXRvcnk1NDM0Njc5OQ==",
```

```
"name": "public-apis",
"full_name": "public-apis/public-apis",
--пропуск--
```

Вероятно, по виду ответа вы уже поняли, что этот URL не предназначен для обычных пользователей, поскольку ответ закодирован в формате, рассчитанном на машинную обработку. На момент написания книги на GitHub было найдено чуть менее девяти миллионов проектов Python ❶. Значение "incomplete\_results" равно true, а значит, у GitHub возникли проблемы с полной обработкой запроса ❷. Алгоритмы GitHub ограничивают продолжительность выполнения каждого запроса, чтобы сохранить доступность API для всех пользователей. В данном случае алгоритмы нашли несколько самых популярных репозиторий Python, но не все; сейчас мы это исправим. Возвращаемые данные отображаются в списке "items", содержащем информацию о самых популярных проектах Python на GitHub ❸.

## **Установка пакета requests**

Пакет requests предоставляет удобные средства, позволяющие запрашивать информацию с сайтов из программ Python и анализировать полученные ответы. Для установки requests используется модуль pip:

```
$ python -m pip install --user requests
```

Если для запуска программ или установки пакетов вы используете python3 или другую команду, то проследите за тем, чтобы здесь использовалась та же команда:

```
$ python3 -m pip install --user requests
```

## **Обработка ответа API**

Теперь мы напишем программу для автоматического вызова API и обработки результатов:

### ***python\_repos.py***

```
import requests

Создание вызова API и сохранение ответа.
❶ url = "https://api.github.com/search/repositories"
url += "?q=language:python+sort:stars+stars:>10000"

❷ headers = {"Accept": "application/vnd.github.v3+json"}
❸ r = requests.get(url, headers=headers)
❹ print(f"Status code: {r.status_code}")

Преобразование объекта ответа в словарь.
❺ response_dict = r.json()

Обработка результатов.
print(response_dict.keys())
```

Сначала мы импортируем модуль `requests`. Затем присваиваем URL вызова API переменной `url` ❶. URL длинный, поэтому мы разбиваем его на две строки. Первая строка — это основная часть URL, а вторая — сам запрос. Мы добавили еще одно условие в строку запроса: `stars:>10000`, которое дает GitHub указание искать только репозитории Python, отмеченные более чем 10 000 звезд. Теперь GitHub сможет возвращать полный, последовательный набор результатов.

В настоящее время GitHub использует третью версию API, поэтому для вызова API определяются заголовки, которые явно требуют использовать эту версию API и возвращают результаты в формате JSON ❷. После этого модуль `requests` задействуется для вызова API ❸. Мы вызываем метод `get()` и передаем ему URL и заголовок, а объект ответа сохраняется в переменной `r`.

Объект ответа содержит атрибут `status_code`, в котором хранится признак успешного выполнения запроса. (Код `200` — признак успешного ответа.) Программа выводит значение `status_code`, чтобы вы могли убедиться в том, что вызов был обработан успешно ❹. API должен возвращать информацию в формате JSON, поэтому в программе используется метод `json()` для преобразования информации в словарь Python ❺. Полученный словарь сохраняется в переменной `response_dict`.

Наконец, программа выводит ключи словаря `response_dict`, и мы видим следующее:

```
Status code: 200
dict_keys(['total_count', 'incomplete_results', 'items'])
```

Код статуса `200` означает, что запрос был обработан успешно. Словарь ответа содержит всего три ключа: `'total_count'`, `'incomplete_results'` и `'items'`. Поговорим о том, как взаимодействовать с таким словарем.

## Работа со словарем ответа

Итак, полученная при вызове API информация хранится в словаре, и мы можем заняться работой с данными. Для начала создадим сводку с обобщенными сведениями — это позволит убедиться в том, что вызов вернул ожидаемую информацию, и перейти к анализу интересующих данных:

### *python\_repos.py*

```
import requests
```

```
Создание вызова API и сохранение ответа.
--пропуск--
```

```
Преобразование объекта ответа в словарь.
response_dict = r.json()
```

```
❶ print(f"Total repositories: {response_dict['total_count']}")
print(f"Complete results: {not response_dict['incomplete_results']}")
```

```

Анализ информации о репозиториях.
❷ repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

Анализ первого репозитория.
❸ repo_dict = repo_dicts[0]
❹ pprint(f"\nKeys: {len(repo_dict)}")
❺ for key in sorted(repo_dict.keys()):
 print(key)

```

Рассмотрим словарь ответа, начиная со значения `'total_count'`, которое представляет собой общее количество репозиторий Python, возвращенных этим вызовом API ❶. Мы также используем значение `'incomplete_results'`, позволяющее узнать, смог ли GitHub полностью обработать запрос. Вместо того чтобы выдавать это значение напрямую, мы выводим его противоположность: значение `True` будет означать, что мы получили полный набор результатов.

Значение, связанное с `'items'`, представляет собой список со словарями, каждый из которых содержит данные об одном репозитории Python. Этот список словарей сохраняется в `repo_dicts` ❷. Затем программа выводит длину `repo_dicts`, чтобы пользователь видел, по какому количеству репозиторий имеется информация.

Чтобы дать нам первое представление об информации, возвращенной по каждому репозиторию, программа извлекает первый элемент из `repo_dicts` и сохраняет его в `repo_dict` ❸. Затем программа выдает количество ключей в словаре — это значение определяет объем доступной информации ❹. И наконец, выводятся все ключи словаря; по ним можно понять, какая информация добавлена в ответ ❺.

Из сводки начинает вырисовываться более четкая картина полученных данных:

```

Status code: 200
❶ Total repositories: 248
❷ Complete results: True
Repositories returned: 30

❸ Keys: 78
allow_forking
archive_url
archived
--пропуск--
url
visibility
watchers
watchers_count

```

На момент написания книги было найдено 248 репозиторий Python, отмеченных 10 000 звезд и более ❶. Судя по результату, GitHub смог полностью обработать вызов API ❷. В ответе GitHub вернул информацию о первых 30 репозиториях, которые соответствуют условиям нашего запроса. Если нужно больше репозиторий, то мы можем запросить дополнительные страницы данных.



API GitHub возвращает подробную информацию о каждом репозитории: в `repo_dict` содержится 78 ключей ❸. Просмотр ключей дает представление о том, какие сведения о проекте можно извлечь. (Какую информацию можно получить через API, мы узнаем, либо прочитав документацию, либо проанализировав информацию в коде.)

Прочитаем значения некоторых ключей `repo_dict`:

#### *python\_repos.py*

*--пропуск--*

```
Анализ первого репозитория.
repo_dict = repo_dicts[0]
```

```
print("\nSelected information about first repository:")
```

```
❶ print(f"Name: {repo_dict['name']}")
❷ print(f"Owner: {repo_dict['owner']['login']}")
❸ print(f"Stars: {repo_dict['stargazers_count']}")
print(f"Repository: {repo_dict['html_url']}")
❹ print(f"Created: {repo_dict['created_at']}")
❺ print(f"Updated: {repo_dict['updated_at']}")
print(f"Description: {repo_dict['description']}")
```

В программе выводятся значения, связанные с некоторыми ключами словаря первого репозитория. Сначала выводится имя проекта ❶. Владельца проекта представляет целый словарь, поэтому ключ `owner` используется для обращения к словарю, представляющему владельца, после чего ключ `login` используется для получения регистрационного имени владельца ❷. Далее выводится количество звезд, заработанных проектом ❸, и URL репозитория GitHub проекта. Затем выводится дата создания ❹ и последнего обновления репозитория ❺. В завершение выводится описание репозитория; результат должен выглядеть примерно так:

```
Status code: 200
```

```
Total repositories: 248
```

```
Complete results: True
```

```
Repositories returned: 30
```

```
Selected information about first repository:
```

```
Name: public-apis
```

```
Owner: public-apis
```

```
Stars: 191493
```

```
Repository: https://github.com/public-apis/public-apis
```

```
Created: 2016-03-20T23:49:42Z
```

```
Updated: 2022-05-12T06:37:11Z
```

```
Description: A collective list of free APIs
```

Из вывода видно, что на момент написания книги самым «звездным» проектом Python на GitHub был проект `public-apis`, владельцем которого является одноименная организация, и звезды этот проект получил от почти 200 тыс. пользователей GitHub. Мы видим URL репозитория проекта, дату создания (март 2016 года)

и то, что проект недавно обновлялся. Наконец, из описания следует, что `public-apis` содержит список бесплатных API, которые могут заинтересовать программистов.

## Сводка самых популярных репозиториях

При создании визуализации этих данных в диаграмму необходимо внести несколько репозиториях. Напишем цикл для вывода информации о каждом репозитории, возвращаемом вызовом API, чтобы всех их можно было добавить в визуализацию:

### *python\_repos.py*

```
--пропуск--
Анализ информации о репозиториях.
repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

❶ print("\nSelected information about each repository:")
❷ for repo_dict in repo_dicts:
 print(f"\nName: {repo_dict['name']}")
 print(f"Owner: {repo_dict['owner']['login']}")
 print(f"Stars: {repo_dict['stargazers_count']}")
 print(f"Repository: {repo_dict['html_url']}")
 print(f>Description: {repo_dict['description']}")
```

Сначала выдается приветственное сообщение ❶. Затем перебираются все словари в `repo_dicts` ❷. Внутри цикла выводятся имя каждого проекта, его владелец, количество звезд, URL на GitHub и краткое описание проекта:

```
Status code: 200
Total repositories: 248
Complete results: True
Repositories returned: 30
```

Selected information about each repository:

```
Name: public-apis
Owner: public-apis
Stars: 191494
Repository: https://github.com/public-apis/public-apis
Description: A collective list of free APIs
```

```
Name: system-design-primer
Owner: donnemartin
Stars: 179952
Repository: https://github.com/donnemartin/system-design-primer
Description: Learn how to design large-scale systems. Prep for the system
 design interview.
 Includes Anki flashcards.
--пропуск--
```

Name: PayloadsAllTheThings  
Owner: swisskyrepo  
Stars: 37227  
Repository: <https://github.com/swisskyrepo/PayloadsAllTheThings>  
Description: A list of useful payloads and bypass for Web Application Security and Pentest/CTF

В этих результатах встречаются интересные проекты; возможно, вам стоит присмотреться к некоторым из них... Но не увлекайтесь, поскольку мы собираемся создать визуализацию, которая существенно упростит чтение результатов.

## Проверка ограничений частоты обращений API

Многие API ограничивают *частоту обращений* (rate limits); иначе говоря, существует предел для количества запросов в определенный промежуток времени. Чтобы узнать, не приближаетесь ли вы к ограничениям GitHub, введите в браузере адрес [https://api.github.com/rate\\_limit](https://api.github.com/rate_limit). Вы получите ответ, который выглядит примерно так:

```
{
 "resources": {
 --пропуск--
 "search": {
 ❶ "limit": 10,
 ❷ "remaining": 9,
 ❸ "reset": 1652338832,
 ❹ "used": 1,
 "resource": "search"
 }
 },
 --пропуск--
 }
```

В этих данных нас интересует частота обращений для поискового API ❶. Видно, что предельная частота составляет 10 запросов в минуту ❷ и что на текущую минуту осталось еще 9 запросов ❸. Значение "reset" представляет *Unix-время*, или *эпохальное время* (epoch time) (количество секунд, прошедших с полуночи 1 января 1970 года), когда произойдет сброс квоты ❹. При достижении предельного количества обращений вы получите короткий ответ, уведомляющий об этом. Тогда просто подождите, пока квота не будет сброшена.

### ПРИМЕЧАНИЕ

Многие API требуют регистрации и получения API-ключа (токена доступа) для совершения API-вызовов. На момент написания книги для GitHub такого требования не было, но если вы получите токен доступа, то предельная частота обращений для ваших программ значительно увеличится.

## Визуализация репозитория с помощью Plotly

Теперь, имея данные о проектах Python, мы создадим визуализацию, демонстрирующую их относительную популярность в GitHub, — интерактивную столбцовую диаграмму: высота каждого столбца будет представлять количество звезд, полученных проектом. Щелчок на столбце будет открывать главную страницу проекта на GitHub.

Сохраните копию программы, над которой вы работаете, под именем `python_repos_visual.py`, а затем приведите ее к следующему виду:

### *python\_repos\_visual.py*

```
import requests
import plotly.express as px

Создание вызова API и сохранение ответа.
url = "https://api.github.com/search/repositories"
url += "?q=language:python+sort:stars+stars:>10000"

headers = {"Accept": "application/vnd.github.v3+json"}
r = requests.get(url, headers=headers)
❶ print(f"Status code: {r.status_code}")

Обработка результатов.
response_dict = r.json()
❷ print(f"Complete results: {not response_dict['incomplete_results']}")

Обработка информации о репозитории.
repo_dicts = response_dict['items']
❸ repo_names, stars = [], []
for repo_dict in repo_dicts:
 repo_names.append(repo_dict['name'])
 stars.append(repo_dict['stargazers_count'])

Создание визуализации.
❹ fig = px.bar(x=repo_names, y=stars)
fig.show()
```

Мы импортируем функционал Plotly Express, а затем выполняем вызов API, как и ранее. Затем выводится статус ответа на вызов API, чтобы мы сразу узнали о возможной проблеме с вызовом API ❶. Обработывая общие результаты, интерпретатор выдает сообщение, подтверждающее, что мы получили полный набор результатов ❷. Остальные вызовы функции `print()` удалены, поскольку фаза исследования данных осталась позади; мы знаем, что получены именно те данные, которые нам нужны.

Затем создаются два пустых списка ❸ для хранения данных, добавляемых в диаграмму. Нам понадобятся имя каждого проекта для пометки столбцов (`repo_names`) и количество звезд, определяющее их высоту (`stars`). В цикле имя каждого проекта и количество звезд присоединяются к соответствующему списку.

Мы создаем первичную визуализацию, используя всего пару строк кода ❹ в соответствии с философией Plotly Express, согласно которой вы должны иметь возможность быстро увидеть диаграмму, прежде чем дорабатывать ее внешний вид. Здесь мы используем функцию `px.bar()` для создания диаграммы. В качестве аргумента `x` передается список `repo_names`, а в качестве аргумента `y` — список `stars`.

Полученная диаграмма изображена на рис. 17.1. Мы видим, что несколько первых проектов существенно популярнее остальных, но все они занимают важное место в экосистеме Python.

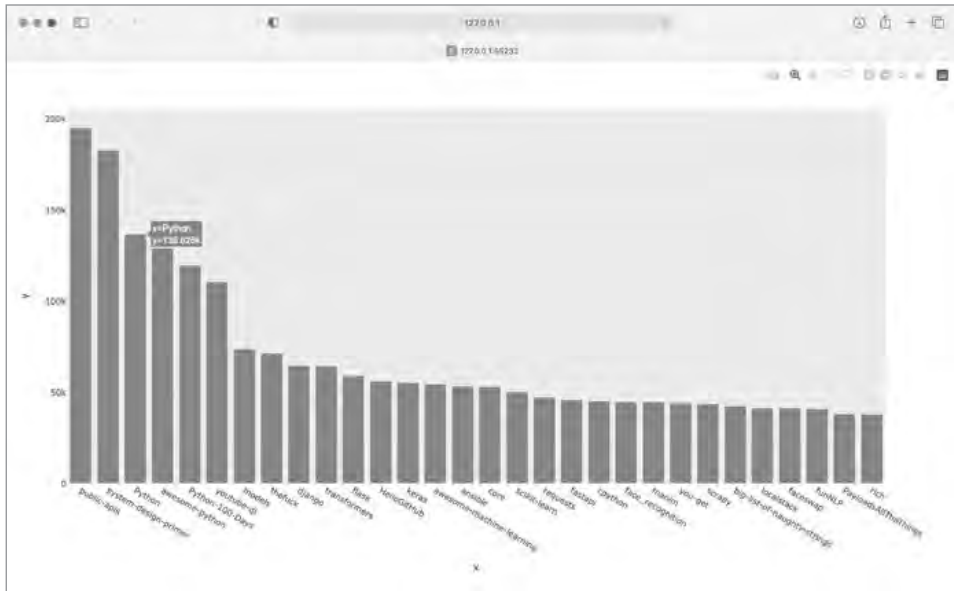


**Рис. 17.1.** Проекты Python на GitHub, имеющие наибольшее количество звезд

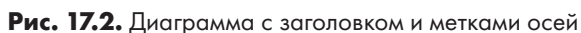
## Форматирование диаграммы

С помощью Plotly доступно множество способов форматирования и настройки диаграмм после того, как все данные отображены на них корректно. Мы внесем некоторые изменения в первоначальный вызов функции `px.bar()`, а затем слегка изменим объект `fig` после его создания.

*python\_repos\_visual.py*

```
1 fig.update_layout(title_font_size=28, xaxis_title_font_size=20,
 yaxis title font size=20)
```

Сначала мы добавляем заголовок диаграммы и метки к каждой оси, как делали это в главах 15 и 16. Затем используем метод `fig.update_layout()` для изменения некоторых элементов диаграммы ❶. Согласно рекомендациям Plotly, слова в именах параметров диаграммы соединяются с помощью подчеркивания. По мере знакомства с документацией Plotly вы начнете видеть закономерности в том, как называются и настраиваются различные элементы диаграммы. В нашем примере мы установили размер шрифта заголовка равным 28, а размер шрифта меток осей — равным 20. Результат показан на рис. 17.2.



## Добавление подсказок

В Plotly при наведении указателя мыши на отдельный столбец выводится информация, которую он представляет. В текущей версии экранная *подсказка* (tooltip) отображает количество звезд проекта. Создадим нестандартную подсказку, которая будет выводить описание каждого проекта, а также его владельца.

Чтобы сгенерировать подсказки, необходимо извлечь дополнительные данные:

### *python\_repos\_visual.py*

```
--пропуск--
```

```
Обработка результатов.
```

```
repo_dicts = response_dict['items']
```

```
❶ repo_names, stars, hover_texts = [], [], []
```

```
for repo_dict in repo_dicts:
```

```
 repo_names.append(repo_dict['name'])
```

```
 stars.append(repo_dict['stargazers_count'])
```

```
 # Создание всплывающих текстов.
```

```
❷ owner = repo_dict['owner']['login']
```

```
 description = repo_dict['description']
```

```
❸ hover_text = f"{owner}
{description}"
```

```
 hover_texts.append(hover_text)
```

```
Создание визуализации.
```

```
title = "Most-Starred Python Projects on GitHub"
```

```
labels = {'x': 'Repository', 'y': 'Stars'}
```

```
❹ fig = px.bar(x=repo_names, y=stars, title=title, labels=labels,
 hover_name=hover_texts)
```

```
fig.update_layout(title_font_size=28, xaxis_title_font_size=20,
 yaxis_title_font_size=20)
```

```
fig.show()
```

Сначала определяется новый пустой список `hover_texts` для хранения текста, который должен выводиться для каждого проекта ❶. В цикле, где происходит обработка данных, мы извлекаем данные о владельце и описание для каждого проекта ❷. Plotly позволяет использовать разметку HTML в текстовых элементах, поэтому мы сгенерируем для метки текст с разрывом строки (`<br />`) между именем владельца проекта и описанием ❸. Затем метка сохраняется в списке `hover_texts`.

В вызов функции `px.bar()` мы добавляем аргумент `hover_name` и передаем ему значение переменной `hover_texts` ❹. С помощью такого же подхода мы настраивали метки для точек на карте активности землетрясений. При создании каждого столбца Plotly извлекает метки из списка и выводит их только в тот момент, когда пользователь наводит указатель мыши на столбец. На рис. 17.3 показана одна из таких пользовательских всплывающих подсказок.

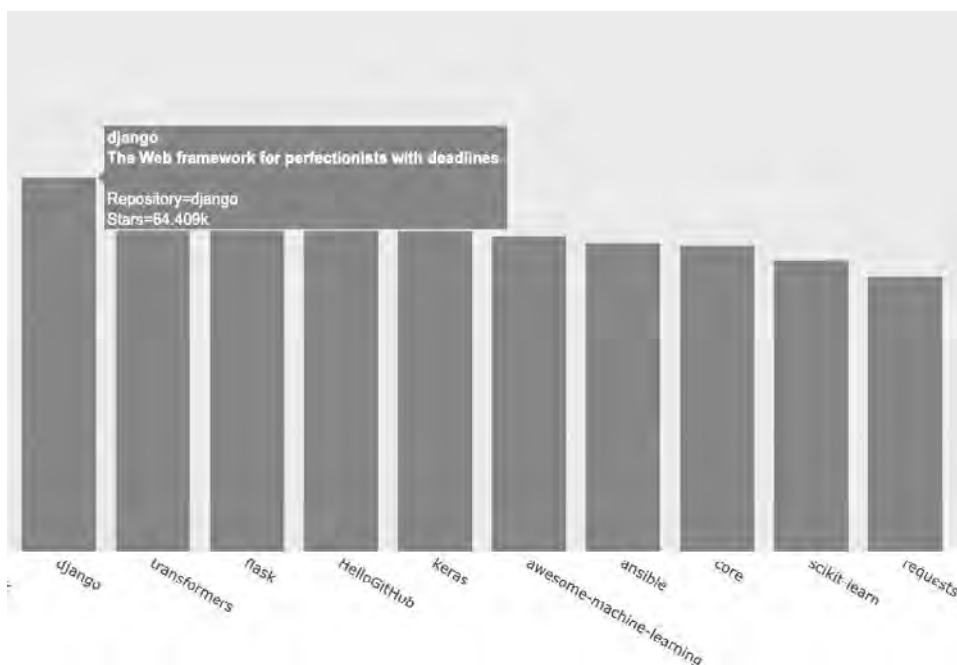


**Рис. 17.3.** При наведении указателя мыши на столбец выводятся информация о владельце и описание проекта

## Добавление ссылок в диаграмму

Plotly позволяет использовать HTML-элементы в текстовых элементах, поэтому диаграмму можно легко дополнить ссылками. Используем метки оси *X* для того, чтобы пользователь мог открыть главную страницу проекта на GitHub. Необходимо извлечь URL из данных и использовать их при генерировании меток для оси *X*.

### *python\_repos\_visual.py*

```
--nпропуск--
Обработка результатов.
repo_dicts = response_dict['items']
❶ repo_links, stars, hover_texts = [], [], []
for repo_dict in repo_dicts:
 # Создание ссылок из названий репозитория.
 repo_name = repo_dict['name']
 ❷ repo_url = repo_dict['html_url']
 repo_link = f"{repo_name}"
 repo_links.append(repo_link)

 stars.append(repo_dict['stargazers_count'])
--nпропуск--
```



```
Создание визуализации.
title = "Most-Starred Python Projects on GitHub"
labels = {'x': 'Repository', 'y': 'Stars'}
fig = px.bar(x=repo_links, y=stars, title=title, labels=labels,
 hover_name=hover_texts)

fig.update_layout(title_font_size=28, xaxis_title_font_size=20,
 yaxis_title_font_size=20)

fig.show()
```

Список, создаваемый на базе `repo_names`, переименован в `repo_links`, чтобы он более точно отражал свойства информации, объединяемой для диаграммы ❶. Затем мы извлекаем URL проекта из `repo_dict` и присваиваем его временной переменной `repo_url` ❷. Далее генерируется ссылка на проект ❸. Для этого используется HTML-тег привязки вида `<a href='URL'>текст ссылки</a>`. Затем ссылка присоединяется к списку `repo_links`.

Вызывая функцию `px.bar()`, мы используем список `repo_links` для получения значений оси *X* диаграммы. Результат выглядит так же, как прежде, но теперь пользователь может щелкнуть на любом из имен проектов в нижней части диаграммы, чтобы посетить главную страницу этого проекта на GitHub. Нам удалось создать интерактивную, информативную визуализацию данных, полученных через API!

## Настройка цвета маркеров

Когда диаграмма создана, практически любой ее аспект можно настроить с помощью метода `update`. Ранее мы использовали метод `update_layout()`. Другой метод, `update_traces()`, применяется для настройки данных, представленных на диаграмме.

Изменим цвет столбцов на темно-синий, чуть прозрачный:

```
--пропуск--
fig.update_layout(title_font_size=28, xaxis_title_font_size=20,
 yaxis_title_font_size=20)

fig.update_traces(marker_color='SteelBlue', marker_opacity=0.6)

fig.show()
```

В контексте Plotly под *трассировкой* (trace) понимается набор данных на графике. Метод `update_traces()` принимает несколько аргументов; те из них, которые начинаются со слова `marker_`, форматируют маркеры на диаграмме. В примере мы задаем для всех маркеров цвет `'SteelBlue'` (можно использовать любые имена из спецификации CSS). Мы также настроили непрозрачность каждого маркера и сделали ее равной `0.6`. При непрозрачности `1.0` маркер будет полностью непрозрачным, а при непрозрачности `0` — полностью невидимым.

## Дополнительная информация о Plotly и GitHub API

Документация Plotly объемна и хорошо структурирована, однако бывает трудно понять, с чего начать чтение. Прочтите статью *Plotly Express in Python* на сайте <https://plotly.com/python/plotly-express>. В ней приведен обзор всех диаграмм, которые вы можете создать с помощью Plotly Express, а также ссылки на более подробные публикации о каждом виде диаграммы.

Если вы хотите основательно разобраться в настройках диаграмм Plotly, то благодаря статье *Styling Plotly Express Figures in Python* расширите ваши познания, полученные в главах 15–17. Вы найдете ее по адресу <https://plotly.com/python/styling-plotly-express>.

За дополнительной информацией о GitHub API обращайтесь к документации, размещенной на <https://docs.github.com/en/rest>. Из нее вы узнаете, как извлекать разнообразную информацию с сайта GitHub. Чтобы расширить знания, полученные в этом проекте, воспользуйтесь системой поиска на боковой панели сайта с документацией. Если у вас уже есть учетная запись GitHub, то вы можете работать как со своими данными, так и с общедоступными из репозиториях других пользователей.

## API Hacker News

Поговорим о том, как использовать вызовы API для других сайтов, и для этого обратимся к сайту Hacker News (<http://news.ycombinator.com/>). На нем пользователи делятся друг с другом статьями, посвященными программированию и технологиям, а также активно обсуждают материалы этих статей. API сайта Hacker News предоставляет доступ ко всем статьям и комментариям на сайте, а его использование не требует регистрации с получением ключа.

Следующий вызов возвращает информацию о текущей самой популярной статье (на момент написания книги):

```
https://hacker-news.firebaseio.com/v0/item/31353677.json
```

Если ввести этот URL в браузере, то вы увидите, что текст на странице заключен в фигурные скобки; это означает, что перед вами словарь. Но в ответе трудно разобраться без дополнительного форматирования. Обработаем URL методом `json.dumps()`, как мы делали в проекте с землетрясениями из главы 16, чтобы было удобнее изучать возвращенную информацию:

### *hn\_article.py*

```
import requests
import json
```

```
Вызов API и сохранение ответа.
```

```
url = "https://hacker-news.firebaseio.com/v0/item/31353677.json"
```

```
r = requests.get(url)
print(f"Status code: {r.status_code}")

Анализ структуры данных.
response_dict = r.json()
response_string = json.dumps(response_dict, indent=4)
❶ print(response_string)
```

В этой программе все должно быть вам знакомо, поскольку все эти элементы использовались в двух предшествующих главах. Основное отличие заключается в том, что отформатированную строку ответа мы можем вывести **❶**, а не записывать в файл, так как вывод не очень длинный.

Ответ представляет собой словарь с информацией о статье с идентификатором 31353677:

```
{
 "by": "sohkamyung",
 ❶ "descendants": 302,
 "id": 31353677,
 ❷ "kids": [
 31354987,
 31354235,
 --nponyск--
],
 "score": 785,
 "time": 1652361401,
 ❸ "title": "Astronomers reveal first image of the black hole
 at the heart of our galaxy",
 "type": "story",
 ❹ "url": "https://public.nrao.edu/news/.../"
}
```

В словаре хранится ряд ключей, которые могут нам пригодиться. Ключ 'descendants' содержит количество комментариев, полученных статьей **❶**. Ключ 'kids' предоставляет идентификаторы всех комментариев, сделанных непосредственно в ответ на эту статью **❷**. У каждого из этих комментариев могут быть дочерние комментарии, так что количество потомков у статьи может превышать количество дочерних комментариев. Кроме того, в данных видны заголовок обсуждаемой статьи **❸** и ее URL **❹**.

Следующий URL возвращает простой список всех идентификаторов текущих популярных статей на сайте Hacker News:

```
https://hacker-news.firebaseio.com/v0/topstories.json
```

С помощью этого вызова можно узнать, какие статьи находятся на главной странице, а затем сгенерировать серию вызовов API, аналогичных приведенному выше.

Это позволит нам вывести сводку всех статей, находящихся на главной странице Hacker News в данный момент:

### *hn\_submissions.py*

```
from operator import itemgetter

import requests

Создание вызова API и сохранение ответа.
❶ url = "https://hacker-news.firebaseio.com/v0/topstories.json"
r = requests.get(url)
print(f"Status code: {r.status_code}")

Обработка информации о каждой статье.
❷ submission_ids = r.json()
❸ submission_dicts = []
for submission_id in submission_ids[:30]:
 # Создание отдельного вызова API для каждой статьи.
 ❹ url = f"https://hacker-news.firebaseio.com/v0/item/{submission_id}.json"
 r = requests.get(url)
 print(f"id: {submission_id}\tstatus: {r.status_code}")
 response_dict = r.json()

 # Создание словаря для каждой статьи.
 ❺ submission_dict = {
 'title': response_dict['title'],
 'hn_link': f"http://news.ycombinator.com/item?id={submission_id}",
 'comments': response_dict['descendants'],
 }
 ❻ submission_dicts.append(submission_dict)
❽ submission_dicts = sorted(submission_dicts, key=itemgetter('comments'),
 reverse=True)

❿ for submission_dict in submission_dicts:
 print(f"\nTitle: {submission_dict['title']}")
 print(f"Discussion link: {submission_dict['hn_link']}")
 print(f"Comments: {submission_dict['comments']}")
```

Сначала программа создает вызов API и выводит статус ответа ❶. Этот вызов возвращает список идентификаторов 500 самых популярных статей на Hacker News на момент выдачи вызова. Текст ответа преобразуется в список Python ❷, который сохраняется в переменной `submission_ids`. Идентификаторы будут использованы для создания набора словарей, каждый из которых содержит информацию об одной из текущих статей.

Далее создается пустой список `submission_dicts` для хранения словарей ❸. Затем программа перебирает идентификаторы 30 самых популярных статей и выдает новый вызов API для каждой статьи, генерируя URL с текущим значением `submission_id` ❹. Выводится и статус каждого запроса, чтобы мы могли проверить, успешно ли он был обработан.

Далее создается словарь для текущей статьи ❺, в котором сохраняются заголовок статьи, ссылка на страницу с ее обсуждением и количество комментариев

к статье. Затем каждый элемент `submission_dict` присоединяется к списку `submission_dicts` ⑥.

Статьи Hacker News ранжируются по общей системе, основанной на нескольких факторах: сколько раз за статью голосовали, сколько комментариев она получила и давно ли была опубликована. Требуется отсортировать список словарей по количеству комментариев. Для этого мы используем функцию `itemgetter()` ⑦ из модуля `operator`. Мы передаем ей ключ `'comments'`, а она извлекает связанное с ним значение из каждого словаря в списке. Затем функция `sorted()` использует это значение для сортировки списка. Мы сортируем список в обратном порядке, чтобы публикации с наибольшим количеством комментариев оказались на первом месте.

После того как список будет отсортирован, мы перебираем элементы ⑧ и выводим для каждой из самых популярных статей три атрибута: заголовок, ссылку на страницу обсуждения и текущее количество комментариев:

```
Status code: 200
id: 31390506 status: 200
id: 31389893 status: 200
id: 31390742 status: 200
--nponyск--

Title: Fly.io: The reclaimer of Heroku's magic
Discussion link: https://news.ycombinator.com/item?id=31390506
Comments: 134

Title: The weird Hewlett Packard FreeDOS option
Discussion link: https://news.ycombinator.com/item?id=31389893
Comments: 64

Title: Modern JavaScript Tutorial
Discussion link: https://news.ycombinator.com/item?id=31390742
Comments: 20
--nponyск--
```

Аналогичный процесс применяется для обращения и анализа информации из любого API. С такими данными вы сможете создать визуализацию, показывающую, какие публикации вызывали наиболее активные обсуждения за последнее время. Этот метод также лежит в основе приложений, предоставляющих специализированные средства чтения таких сайтов, как Hacker News. Чтобы узнать больше об информации, доступной через Hacker News API, посетите страницу документации по адресу <https://github.com/HackerNews/API/>.

## ПРИМЕЧАНИЕ

Сотрудники компании Hacker News иногда позволяют определенным компаниям публиковать специальные посты о найме, комментарии к которым отключены. Если вы запустите программу `hn_submissions.py` в момент, когда один из таких постов опубликован, то увидите ошибку `KeyError`. При возникновении подобной проблемы вы можете поместить код `submission_dict` в блок `try`-`except` и пропустить эти посты.

## УПРАЖНЕНИЯ

**17.1. Другие языки.** Измените вызов API в программе `python_repos.py` так, чтобы на диаграмме отображались самые популярные проекты на других языках. Попробуйте такие языки, как JavaScript, Ruby, C, Java, Perl, Haskell и Go.

**17.2. Активные обсуждения.** На основании данных из файла `hn_submissions.py` создайте столбцовую диаграмму самых активных обсуждений, проходящих на Hacker News. Высота каждого столбца должна соответствовать количеству комментариев к каждой статье. Метка столбца должна содержать заголовок статьи и служить ссылкой на страницу обсуждения этой публикации. Если вы получаете ошибку `KeyError` при создании диаграммы, то используйте блок `try-except`, чтобы пропустить рекламные сообщения.

**17.3. Тестирование программы `python_repos.py`.** В `python_repos.py` для проверки успешности вызова API выводится значение `status_code`. Напишите программу `test_python_repos.py`, которая использует модуль `pytest` для проверки того, что значение `status_code` равно 200. Придумайте другие условия, которые могут проверяться при тестировании, — например, что количество возвращаемых элементов совпадает с ожидаемым, а общее количество репозитория превышает некий порог.

**17.4. Дальнейшие исследования.** Ознакомьтесь с документацией Plotly и GitHub API (или Hacker News API). Используйте полученную информацию для настройки форматирования уже созданных диаграмм или загрузите другие данные и создайте собственные визуализации. Если вам интересно изучить другие API, упомянутые в репозитории GitHub, то посетите страницу <https://github.com/public-apis>.

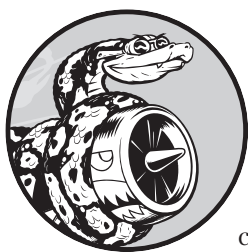
## Резюме

В этой главе вы узнали, как задействовать API для написания программ, которые автоматически собирают необходимые данные и используют их для создания визуализации. Вы использовали GitHub API для получения информации о самых популярных проектах Python на GitHub, а также в общих чертах рассмотрели API Hacker News. Вы узнали, как использовать пакет `requests` для автоматического вызова API и как обработать его результаты. Кроме того, в главе были описаны некоторые средства конфигурации Plotly, позволяющие выполнить дополнительную настройку внешнего вида создаваемых диаграмм.

В следующей главе мы рассмотрим работу с Django для создания веб-приложения, которое станет последним проектом в этой книге.

# 18

## Знакомство с Django



Развитие Интернета привело к тому, что грань между сайтами и мобильными приложениями размылась. И те и другие позволяют пользователям различными способами взаимодействовать с данными. К счастью, вы можете использовать Django для создания единого проекта, помогающего обслуживать как динамический сайт, так и группу мобильных приложений. Django представляет собой популярный *веб-фреймворк* Python — набор средств, упрощающих создание интерактивных веб-приложений. В этой главе вы с помощью Django создадите проект «Журнал обучения» — сетевую журнальную систему для отслеживания информации, полученной по определенной теме.

Мы напомним спецификацию для этого проекта, а затем определим модели для данных, с которыми будет работать приложение. Мы воспользуемся административной системой Django для ввода некоторых начальных данных, а затем научимся писать представления и шаблоны, на базе которых Django будет создавать страницы нашего сайта.

Django может реагировать на запросы страниц, упрощает чтение и запись данных в базы, управление действиями пользователей и многие другие операции. В главах 19 и 20 мы доработаем «Журнал обучения», а затем развернем его на рабочем сервере, чтобы вы (и любой другой пользователь) могли пользоваться этим журналом.

### Подготовка к созданию проекта

Приступая к работе над таким значимым проектом, как веб-приложение, необходимо сначала описать цели проекта в *спецификации*. Четко определившись с целями, можно начинать работу над управляемыми задачами, чтобы достичь этих целей.

В этом разделе мы напомним спецификацию для проекта «Журнал обучения» и начнем работать над первой фазой проекта. Она будет состоять из настройки виртуальной среды и формирования начальных аспектов проекта Django.

## Написание спецификации

В полной спецификации описываются цели проекта, его функциональность, а также внешний вид и интерфейс пользователя. Как и любой хороший проект или бизнес-план, спецификация должна сосредоточиться на самых важных аспектах и обеспечивать планомерную разработку проекта. Здесь мы не будем писать полную спецификацию, а сформулируем несколько четких целей, которые будут задавать направление процесса разработки. Вот как выглядит спецификация.

Мы напомним веб-приложение «Журнал обучения», с помощью которого пользователь сможет вести журнал интересных его тем и создавать в нем записи во время изучения каждой темы. Главная страница «Журнала обучения» содержит описание сайта и приглашает пользователя зарегистрироваться либо ввести свои учетные данные. После успешного входа пользователь получает возможность создавать новые темы, добавлять новые записи, читать и редактировать существующие записи.

Во время изучения нового материала бывает полезно вести журнал того, что вы узнали, — записи пригодятся для контроля и возвращения к необходимой информации. Это особенно актуально при изучении технических тем. Хорошее приложение повышает эффективность этого процесса.

## Создание виртуальной среды

Для работы с Django необходимо сначала создать виртуальную среду для работы. *Виртуальная среда* (virtual environment) представляет собой подраздел системы, в котором вы можете устанавливать пакеты изолированно от всех остальных пакетов Python. Отделение библиотек одного проекта от других проектов принесет пользу при развертывании приложения «Журнал обучения» на сервере в главе 20.

Создайте для проекта новый каталог `learning_log`, перейдите в этот каталог в терминальном режиме и создайте виртуальную среду, используя следующие команды:

```
learning_log$ python -m venv ll_env
learning_log$
```

Команда запускает модуль виртуальной среды `venv` и использует его для создания виртуальной среды `ll_env` (обратите внимание: в имени `ll_env` две буквы `l`, а не одна). Если для запуска программ или установки пакетов вы используете другую команду (например, `python3`), то подставьте ее на место `python`.



## Активация виртуальной среды

После того как виртуальная среда будет создана, ее необходимо активировать с помощью этой команды:

```
learning_log$ source ll_env/bin/activate
(ll_env)learning_log$
```

Команда запускает сценарий `activate` из каталога `ll_env/bin`. Когда среда активируется, ее имя выводится в круглых скобках; теперь вы можете устанавливать пакеты в среде и использовать те пакеты, которые были загружены ранее. Пакеты, установленные в среде `ll_env`, будут доступны только в то время, пока она остается активной.

### ПРИМЕЧАНИЕ

Если вы работаете в системе Windows, то используйте команду `ll_env\Scripts\activate` (без слова `source`) для активизации виртуальной среды. Если используете PowerShell, то слово `Activate` должно начинаться с прописной буквы.

Чтобы завершить использование виртуальной среды, введите команду `deactivate`:

```
(ll_env)learning_log$ deactivate
learning_log$
```

Среда деактивируется и при закрытии терминального окна, в котором она работает.

## Установка Django

После того как вы создали свою виртуальную среду и активизировали ее, установите Django с помощью инструмента `pip`:

```
(ll_env)learning_log$ pip install --upgrade pip
(ll_env)learning_log$ pip install django
Collecting django
--пропуск--
Installing collected packages: sqlparse, asgiref, django
Successfully installed asgiref-3.5.2 django-4.1 sqlparse-0.4.2
(ll_env)learning_log$
```

Поскольку `pip` скачивает ресурсы из различных источников, он обновляется довольно часто. Рекомендуется выполнять обновление `pip` каждый раз, когда вы создаете новую виртуальную среду.

Вы работаете в виртуальной среде, поэтому команда для установки Django выглядит одинаково во всех системах. Использовать флаг `--user` не нужно, как и более

длинные команды вида `python -m pip install имя_пакета`. Помните, что с Django можно работать только в то время, пока среда (в нашем случае `ll_env`) остается активной.

### ПРИМЕЧАНИЕ

Новая версия Django выходит приблизительно раз в восемь месяцев; возможно, при установке Django будет выведен новый номер версии. Скорее всего, проект будет работать в том виде, в котором он приведен здесь, даже в новых версиях Django. Если вы хотите использовать ту же версию Django, которая используется здесь, то введите команду `pip install django==4.1.*`. Будет установлен последний выпуск Django 4.1. Если у вас возникнут проблемы, связанные с версией, то обращайтесь к онлайн-ресурсам книги по адресу [https://ehmatthes.github.io/pcc\\_3e](https://ehmatthes.github.io/pcc_3e).

## Создание проекта в Django

Не выходя из активной виртуальной среды (пока `ll_env` выводится в круглых скобках), введите следующие команды для создания нового проекта:

- ❶ `(ll_env)learning_log$ django-admin startproject ll_project .`
- ❷ `(ll_env)learning_log$ ls`  
`ll_env ll_project manage.py`
- ❸ `(ll_env)learning_log$ ls ll_project`  
`__init__.py asgi.py settings.py urls.py wsgi.py`

Команда `startproject` ❶ дает Django указание создать новый проект `ll_project`. Благодаря точке в конце команды создается новый проект со структурой каталогов, которая упрощает развертывание приложения на сервере после завершения разработки.

### ПРИМЕЧАНИЕ

Не забывайте о точке, иначе у вас могут возникнуть проблемы с конфигурацией при развертывании приложения. А если все же забыли, то удалите созданные файлы и папки (кроме `ll_env`) и снова выполните команду.

Команда `ls` (`dir` в Windows) ❷ показывает, что Django создает новый каталог `ll_project`. Вдобавок создается файл `manage.py` — короткая программа, которая получает команды и передает их соответствующей части Django для выполнения. Мы используем эти команды для управления такими задачами, как работа с базами данных и запуск серверов.

В каталоге `ll_project` находятся четыре файла ❸, важнейшими из которых являются `settings.py`, `urls.py` и `wsgi.py`. Файл `settings.py` определяет, как Django взаимодействует с вашей системой и управляет вашим проектом. Мы изменим некоторые из существующих настроек и добавим несколько новых в ходе разработки проекта. Файл `urls.py` сообщает Django, какие страницы следует создавать

в ответ на запросы браузера. Файл `wsgi.py` помогает Django предоставлять созданные файлы (имя файла является сокращением от `web server gateway interface` (интерфейс шлюза веб-сервера)).

## Создание базы данных

Django хранит большую часть информации, относящейся к проекту, в базе данных, поэтому на следующем этапе необходимо создать базу данных, с которой Django сможет работать. Введите следующую команду (все еще не покидая активную среду):

```
(ll_env)learning_log$ python manage.py migrate
❶ Operations to perform:
 Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
 Applying contenttypes.0001_initial... OK
 Applying auth.0001_initial... OK
 --пропуск--
 Applying sessions.0001_initial... OK
❷ (ll_env)learning_log$ ls
db.sqlite3 ll_env ll_project manage.py
```

Каждое изменение базы данных называется *миграцией* (migrating). Благодаря первому выполнению команды `migrate` Django получает указание проверить, что база данных соответствует текущему состоянию проекта. Когда мы впервые выполняем эту команду в новом проекте с помощью SQLite (подробнее о SQLite поговорим позже), Django создает новую базу данных за нас. Django сообщает о создании и подготовке базы к хранению информации, необходимой для выполнения административных операций и аутентификации ❶.

Выполнение команды `ls` показывает, что Django создает другой файл `db.sqlite3` ❷. SQLite — база данных, работающая с одним файлом; она идеально подходит для написания простых приложений, поскольку вам не нужно пристально следить за управлением базой данных.

### ПРИМЕЧАНИЕ

В активной виртуальной среде для выполнения команд `manage.py` предназначена команда `python`, даже если для запуска других программ вы используете другую команду (например, `python3`). В виртуальной среде команда `python` относится к версии Python, создавшей виртуальную среду.

## Просмотр проекта

Убедимся в том, что проект был создан правильно. Введите команду `runserver` для просмотра текущего состояния проекта:

```
(ll_env)learning_log$ python manage.py runserver
Watching for file changes with StatReloader
```

Performing system checks...

- ❶ System check identified no issues (0 silenced).  
May 19, 2022 - 21:52:35
- ❷ Django version 4.1, using settings 'll\_project.settings'
- ❸ Starting development server at <http://127.0.0.1:8000/>  
Quit the server with CONTROL-C.

Django запускает сервер, называемый *сервером разработки* (development server), чтобы вы могли просмотреть проект в своей системе и проверить, как он работает. Когда вы запрашиваете страницу, вводя URL в браузере, сервер Django отвечает на запрос; для этого он создает соответствующую страницу и отправляет страницу браузеру.

Сначала Django проверяет правильность созданного проекта ❶: выводятся версия Django и имя используемого файла настроек ❷, а затем возвращается URL, по которому доступен проект ❸. URL <http://127.0.0.1:8000/> означает, что проект прослушивает запросы на порте 8000 локального хоста, то есть вашего компьютера. Термином «*локальный хост*» (localhost) обозначается сервер, который обрабатывает только запросы вашей системы; он не позволяет никому другому просмотреть разрабатываемые страницы.

Теперь откройте браузер и введите URL <http://localhost:8000/> или <http://127.0.0.1:8000/>, если первый адрес не работает. Вы увидите нечто похожее на рис. 18.1 — страницу, которую создает Django, чтобы сообщить вам, что все пока работает правильно. Пока не завершайте работу сервера (но когда захотите прервать ее, это можно сделать, нажав сочетание клавиш Ctrl+C в терминале, в котором была введена команда `runserver`).

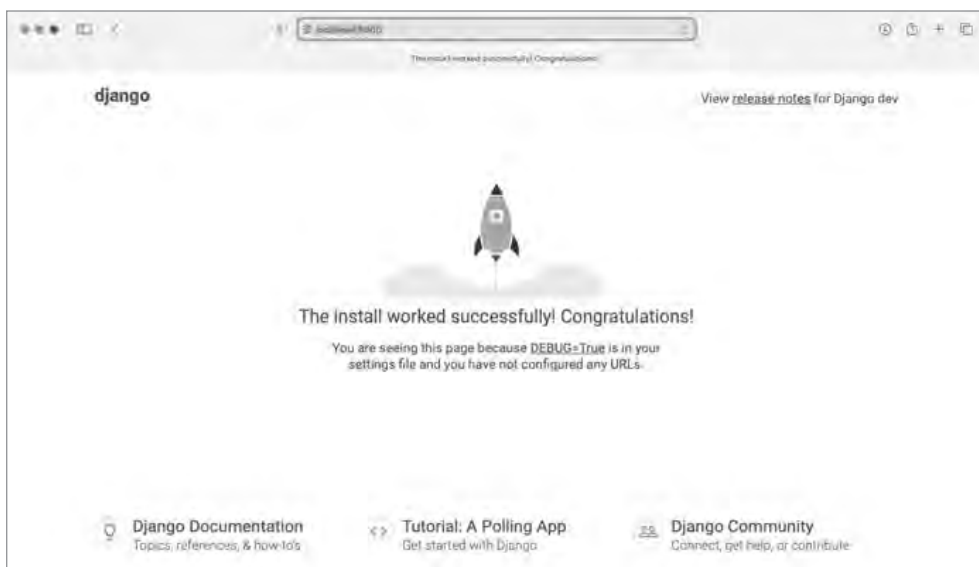


**Рис. 18.1.** Пока все работает правильно

### ПРИМЕЧАНИЕ

Если вы получаете сообщение об ошибке `That port is already in use` («Порт уже используется»), то настройте в Django другой порт; для этого введите команду `python manage.py runserver 8001` и продолжайте перебирать номера портов по возрастанию, пока не найдете открытый.

### УПРАЖНЕНИЯ

**18.1. Новые проекты.** Чтобы лучше понять, что делает Django, создайте пару пустых проектов и посмотрите, что получится. Создайте новую папку с простым именем типа `tik_gram` или `insta_tok` (за пределами каталога `learning_log`), перейдите в нее в терминальном окне и создайте виртуальную среду. Установите Django и выполните команду `django-admin.py startproject tg_project`. (убедитесь, что поставили точку в конце команды).

Просмотрите файлы и каталоги, созданные командой, и сравните их с файлами и каталогами «Журнала обучения». Прodelайте это несколько раз, пока не начнете хорошо понимать, что именно создает Django при создании нового проекта, а затем по желанию удалите каталоги проектов.

## Начало работы над приложением

Проект Django представляет собой группу отдельных *приложений*, совместная работа которых обеспечивает работу проекта в целом. Пока мы создадим одно приложение, которое будет выполнять большую часть работы в нашем проекте. Другое приложение для управления учетными записями пользователей будет добавлено в главе 19.

Оставьте сервер разработки выполняться в терминальном окне, открытом ранее. Откройте новое терминальное окно (или вкладку) и перейдите в каталог, содержащий файл `manage.py`. Активируйте виртуальную среду и выполните команду `startapp`:

```
learning_log$ source ll_env/bin/activate
(ll_env)learning_log$ python manage.py startapp learning_logs
❶ (ll_env)learning_log$ ls
db.sqlite3 learning_logs ll_env ll_project manage.py
❷ (ll_env)learning_log$ ls learning_logs/
__init__.py admin.py apps.py migrations models.py tests.py views.py
```

Команда `startapp имя_приложения` дает Django указание создать инфраструктуру, необходимую для создания приложения. Заглянув сейчас в каталог проекта, вы найдете в нем новую папку `learning_logs` ❶. Воспользуйтесь командой `ls`, чтобы увидеть, какие файлы были созданы Django ❷. Самые важные файлы в этой папке — `models.py`, `admin.py` и `views.py`. Файл `models.py` будет использоваться для определения данных, которыми нужно управлять в нашем приложении. К файлам `admin.py` и `views.py` мы вернемся позднее.

## Определение моделей

Давайте подумаем, какие данные нам понадобятся. Каждый пользователь создает в своем журнале набор тем. Любая запись, которую он сделает, будет привязана к определенной теме, а записи будут выводиться в текстовом виде. Кроме того, необходимо хранить временную метку каждой записи, чтобы пользователь знал, когда была создана эта запись.

Откройте файл `models.py` и просмотрите его текущее содержимое:

### *models.py*

```
from django.db import models
```

```
Создайте здесь свои модели.
```

Модуль `models` импортируется автоматически, и нам предлагается создать свои модели. Модель сообщает Django, как работать с данными, которые будут храниться в приложении. С точки зрения кода модель представляет собой обычный класс; она содержит атрибуты и методы, как и все остальные классы, с которыми вы познакомились ранее. Вот как выглядит модель тем обсуждения, которые будут сохраняться пользователями:

```
from django.db import models
```

```
class Topic(models.Model):
```

```
 """Тема, которую изучает пользователь."""
```

```
❶ text = models.CharField(max_length=200)
```

```
❷ date_added = models.DateTimeField(auto_now_add=True)
```

```
❸ def __str__(self):
```

```
 """Возвращает строковое представление модели."""
```

```
 return self.text
```

Мы создали класс `Topic`, наследуемый от `Model` — родительского класса, включенного в Django и определяющего базовую функциональность модели. В класс `Topic` добавляются два атрибута: `text` и `date_added`.

Атрибут `text` содержит данные `CharField` — блок данных, состоящий из символов, то есть текст ❶. Атрибуты `CharField` могут использоваться для хранения небольших объемов текста: имен, заголовков, названий городов и т. д. При определении атрибута `CharField` необходимо сообщить Django, сколько места нужно зарезервировать для него в базе данных. В данном случае задается максимальная длина `max_length`, равная 200 символам; этого должно быть достаточно для хранения большинства имен тем.

Атрибут `date_added` содержит данные `DateTimeField` — блок данных для хранения даты и времени ❷. Благодаря аргументу `auto_add_now=True` Django получает указание автоматически присвоить этому атрибуту текущую дату и время каждый раз, когда пользователь создает новую тему.

Рекомендуется сообщить Django, как представлять экземпляр модели. Django вызывает метод `__str__()` для вывода простого представления экземпляра модели. Мы написали реализацию `__str__()`, которая возвращает строку, хранящуюся в атрибуте `text` ❸.

Полный список всех полей, которые могут использоваться в модели, приведен в документе Django Model Field Reference на сайте <https://docs.djangoproject.com/en/4.1/ref/models/fields>. Возможно, вся эта информация вам сейчас не понадобится, но она будет в высшей степени полезной, когда вы начнете разрабатывать собственные приложения.

## Активизация моделей

Чтобы использовать модели, необходимо дать Django указание добавить приложение в общий проект. Откройте файл `settings.py` (из каталога `ll_project`) и найдите в нем раздел, который сообщает Django, какие приложения установлены в проекте.

### `settings.py`

```
--пропуск--
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
]
--пропуск--
```

Добавьте наше приложение в этот список, изменив содержимое `INSTALLED_APPS`, чтобы оно выглядело так:

```
--пропуск--
INSTALLED_APPS = [
 # Мои приложения
 'learning_logs',

 # Приложения Django по умолчанию.
 'django.contrib.admin',
 --пропуск--
]
--пропуск--
```

Группирование приложений в проекте упрощает управление ими по мере того, как проект развивается, а количество приложений увеличивается. Здесь мы создаем раздел `My apps`, который пока содержит только приложение `'learning_logs'`. Очень важно разместить свои приложения перед приложениями по умолчанию на случай, если вам понадобится переопределить поведение последних.

Затем необходимо дать Django указание изменить базу данных для хранения информации, относящейся к модели `Topic`. В терминальном окне введите следующую команду:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
 learning_logs/migrations/0001_initial.py
 - Create model Topic
(ll_env)learning_log$
```

По команде `makemigrations` Django определяет, как изменить базу данных для хранения информации, связанной с новыми моделями. Из результатов видно, что Django создает файл миграции `0001_initial.py`. Эта миграция создает в базе данных таблицу для модели `Topic`.

Теперь применим миграцию для автоматического изменения базы данных:

```
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
 Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
 Applying learning_logs.0001_initial... OK
```

Большая часть вывода этой команды совпадает с выводом, полученным при первом выполнении команды `migrate`. Нам нужно проверить последнюю строку в этом выводе: здесь Django подтверждает, что применение миграции для `learning_logs` прошло успешно.

Каждый раз, когда вы захотите изменить данные, которыми управляет приложение «Журнал обучения», выполните эти три действия: внесите изменения в `models.py`, вызовите `makemigrations` для `learning_logs` и дайте Django указание выполнить миграцию проекта (`migrate`).

## Административный сайт Django

Django позволяет легко работать с моделями, определенными для приложения, через *административный сайт* (admin site). Он используется администраторами сайта, а не рядовыми пользователями. В этом разделе мы создадим такой сайт и используем его для добавления некоторых тем через модель `Topic`.

### Создание суперпользователя

Django позволяет создать пользователя, обладающего полным набором привилегий на сайте; такой пользователь называется *суперпользователем* (superuser). *Привилегии* (privileges) управляют действиями, которые разрешено выполнять пользователю. При самом жестком уровне привилегий пользователь может только читать общедоступную информацию на сайте. Зарегистрированным пользователям обычно предоставляется привилегия чтения своих приватных данных, а также избранной



информации, доступной только для участников сообщества. Эффективное администрирование веб-приложения обычно требует, чтобы владельцу сайта была доступна вся хранящаяся на нем информация. Хороший администратор внимательно относится к конфиденциальной информации пользователя, поскольку пользователи доверяют приложениям, с которыми работают.

Чтобы создать суперпользователя в Django, введите следующую команду и ответьте на запросы:

```
(ll_env)learning_log$ python manage.py createsuperuser
❶ Username (leave blank to use 'eric'): ll_admin
❷ Email address:
❸ Password:
Password (again):
Superuser created successfully.
(ll_env)learning_log$
```

При получении команды `createsuperuser` Django предлагает указать имя пользователя, который будет суперпользователем ❶. Здесь я ввел имя `ll_admin`, но вы можете добавить любое имя на свое усмотрение. Кроме того, можно ввести адрес электронной почты или оставить это поле пустым ❷. После этого следует дважды ввести пароль ❸.

### ПРИМЕЧАНИЕ

Часть конфиденциальной информации может быть скрыта от администраторов сайта. Например, Django на самом деле не сохраняет введенный пароль; вместо этого сохраняется хеш — специальная строка, созданная на основе пароля. И когда в будущем вы вводите пароль, Django снова хеширует введенные данные и сравнивает результат с хранимым хешем. Если два хеша совпадают, то проверка пройдена. Если же хакер в результате атаки получит доступ к базе данных сайта, то сможет прочесть только хранящийся в базе хеш, но не пароли. При правильной настройке сайта восстановить исходные пароли из хешей почти невозможно.

### Регистрация модели на административном сайте

Django добавляет некоторые модели (например, `User` и `Group`) на административный сайт автоматически, но модели, которые мы создали, придется регистрировать вручную.

При запуске приложения `learning_logs` Django создает файл `admin.py` в одном каталоге с `models.py`. Откройте файл `admin.py`:

#### *admin.py*

```
from django.contrib import admin
```

```
Зарегистрируйте здесь ваши модели.
```

Чтобы зарегистрировать `Topic` на административном сайте, введите следующую команду:

```
from django.contrib import admin

from .models import Topic

admin.site.register(Topic)
```

Этот код импортирует регистрируемую модель `Topic`. Точка перед `models` сообщает Django, что файл `models.py` следует искать в одном каталоге с `admin.py`. Вызов `admin.site.register()` сообщает Django, что управление моделью должно осуществляться через административный сайт.

Теперь используйте учетную запись суперпользователя для входа на административный сайт. Введите адрес <http://localhost:8000/admin/>, введите имя пользователя и пароль для только что созданного суперпользователя — и увидите экран наподобие изображенного на рис. 18.2. На этой странице можно добавлять новых пользователей и группы, а также вносить изменения в уже существующие настройки. Кроме того, здесь можно работать с данными, связанными с только что определенной моделью `Topic`.

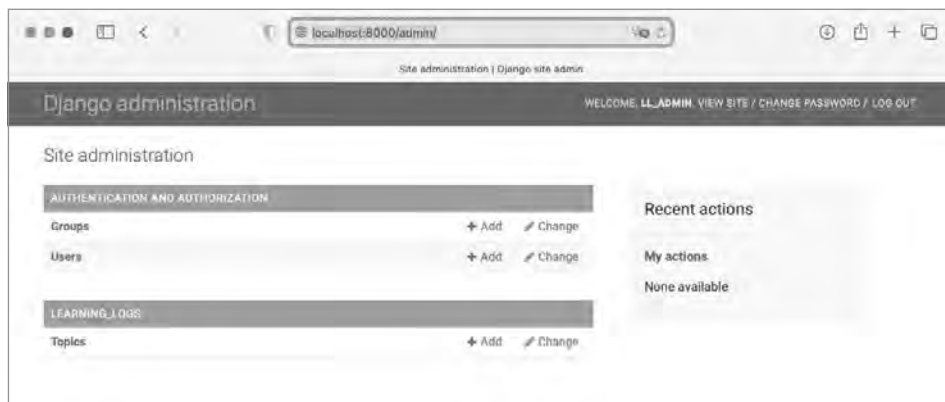


**Рис. 18.2.** Административный сайт с добавленной моделью `Topic`

### ПРИМЕЧАНИЕ

Если в браузере появляется сообщение о недоступности веб-страницы, то убедитесь в том, что сервер Django работает в терминальном окне. Если нет, то активизируйте виртуальную среду и снова введите команду `python manage.py runserver`. Если у вас возникнут проблемы с просмотром проекта в любой момент в процессе разработки, то закройте все открытые терминалы и снова введите команду `runserver`; это станет хорошим первым шагом в процессе диагностики.

## Добавление тем

Когда модель `Topic` была зарегистрирована на административном сайте, добавим первую тему. Щелкните на ссылке **Topics** (Темы), чтобы перейти к странице **Topics** (Темы); она практически пуста, поскольку еще нет ни одной темы, с которой можно выполнять операции. Щелкните на ссылке **Add Topic** (Добавить тему); открывается форма для добавления новой темы. Введите в первом поле текст **Chess** (Шахматы) и щелкните на ссылке **Save** (Сохранить). Вы возвращаетесь к административной странице **Topics** (Темы), на которой появляется только что созданная тема.

Создадим вторую тему, чтобы у вас было больше данных для работы. Снова щелкните на ссылке **Add Topic** (Добавить тему) и создайте вторую тему **Rock Climbing** (Скалолазание). Ссылка **Save** (Сохранить) снова возвращает вас к основной странице **Topics** (Темы), на которой отображаются обе темы: **Chess** (Шахматы) и **Rock Climbing** (Скалолазание).

## Определение модели *Entry*

Чтобы сохранить информацию о том, что вы узнали по этим двум темам, необходимо определить модель для записей, которые пользователь делает в своих журналах. Каждая запись должна быть связана с конкретной темой. Такое отношение называется *отношением «многие-к-одному»* (many-to-one relationship), поскольку многие записи могут быть связаны с одной темой.

Код модели `Entry` выглядит так (поместите его в файл `models.py`):

### *models.py*

```
from django.db import models
```

```
class Topic(models.Model):
 --пропуск--
```

```
❶ class Entry(models.Model):
 """Информация, изученная пользователем по теме."""
❷ topic = models.ForeignKey(Topic, on_delete=models.CASCADE)
❸ text = models.TextField()
 date_added = models.DateTimeField(auto_now_add=True)

❹ class Meta:
 verbose_name_plural = 'entries'

 def __str__(self):
 """Возвращает строковое представление модели."""
❺ return f"{self.text[:50]}..."
```

Класс `Entry` наследует от базового класса `Model`, как и рассмотренный ранее класс `Topic` ❶. Первый атрибут, `topic`, является экземпляром `ForeignKey` ❷. Термин «внешний ключ» (foreign key) происходит из теории баз данных; внешний ключ

содержит ссылку на другую запись в базе данных. Таким образом каждая запись связывается с конкретной темой. Каждой теме при создании присваивается *ключ*, или идентификатор. Если потребуется установить связь между двумя записями данных, то Django использует ключ, связанный с каждым блоком информации. Вскоре мы применим такие связи для получения всех записей, имеющих отношение к заданной теме. Аргумент `on_delete=models.CASCADE` сообщает Django, что при удалении темы все записи, связанные с этой темой, должны быть удалены (это называется *каскадным удалением* (cascading delete)).

Затем идет атрибут `text`, который является экземпляром `TextField` ❸. Полю такого типа ограничение размера не требуется, поскольку размер отдельных записей не ограничивается. Атрибут `date_added` позволяет отображать записи в порядке их создания и добавлять в каждую запись временную метку.

Класс `Meta` вкладывается в класс `Entry` ❹. Класс `Meta` хранит дополнительную информацию по управлению моделью; в данном случае он позволяет задать специальный атрибут, благодаря которому Django получает указание использовать форму множественного числа `Entries` при обращении к нескольким записям. (Без этого Django будет использовать неправильную форму `Entrys`.)

Метод `__str__()` сообщает Django, какая информация должна отображаться при обращении к отдельным записям. Запись может быть достаточно длинным блоком текста, поэтому `__str__()` выводит только первые 50 символов ❺. Кроме того, добавляется многоточие — признак вывода неполного текста.

## Миграция модели `Entry`

Мы добавили новую модель, поэтому миграцию базы данных необходимо провести снова. Вскоре вы привыкнете к этому процессу: изменяете `models.py`, выполняете команду `python manage.py makemigrations имя_приложения`, а затем команду `python manage.py migrate`.

Проведите миграцию базы данных и проверьте вывод:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
❶ learning_logs/migrations/0002_entry.py
 - Create model Entry
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
 --nponуск--
❷ Applying learning_logs.0002_entry... OK
```

Команда генерирует новую миграцию `0002_entry.py`, которая сообщает Django, как изменить базу данных, чтобы можно было хранить информацию, связанную с моделью `Entry` ❶. При выдаче команды `migrate` Django подтверждает, что применение миграции прошло успешно ❷.

## Регистрация *Entry* на административном сайте

Модель *Entry* тоже необходимо зарегистрировать. Файл `admin.py` должен выглядеть так:

### *admin.py*

```
from django.contrib import admin

from .models import Topic, Entry

admin.site.register(Topic)
admin.site.register(Entry)
```

Вернитесь на страницу <http://localhost/admin/> — и увидите раздел *Entries* (Записи) в категории *learning\_logs*. Щелкните на ссылке *Add* (Добавить) для *Entries* (Записи) или щелкните на *Entries* (Записи) и выберите вариант *Add* (Добавить). На экране должны появиться раскрывающийся список, позволяющий выбрать тему, для которой создается запись, и текстовое поле для ввода записи. Выберите в раскрывающемся списке вариант *Chess* (Шахматы) и добавьте запись. Вот первая запись, которую я сделал:

The opening is the first part of the game, roughly the first ten moves or so. In the opening, it's a good idea to do three things — bring out your bishops and knights, try to control the center of the board, and castle your king.

Of course, these are just guidelines. It will be important to learn when to follow these guidelines and when to disregard these suggestions.

Если вы щелкнете на ссылке *Save* (Сохранить), то вернетесь к основной административной странице. Здесь проявляются преимущества использования формата `text[:50]` в качестве строкового представления каждой записи; работать с несколькими записями в административном интерфейсе намного удобнее, если вы видите только часть записи вместо ее полного текста.

Создайте вторую запись для темы *Chess* (Шахматы) и одну запись для темы *Rock Climbing* (Скалолазание), чтобы у нас были исходные данные для дальнейшей разработки «Журнала обучения». Вот вторая запись для темы *Chess* (Шахматы):

In the opening phase of the game, it's important to bring out your bishops and knights. These pieces are powerful and maneuverable enough to play a significant role in the beginning moves of a game.

А вот первая запись для темы *Rock Climbing* (Скалолазание):

One of the most important concepts in climbing is to keep your weight on your feet as much as possible. There's a myth that climbers can hang all day on their arms. In reality, good climbers have practiced specific ways of keeping their weight over their feet whenever possible.

Эти три записи нам пригодятся, когда мы продолжим разработку приложения «Журнал обучения».

## Интерактивная оболочка Django

Введенные данные можно проанализировать на программном уровне в интерактивном терминальном сеансе. Эта интерактивная среда, называемая *оболочкой* (shell) Django, прекрасно подходит для тестирования и диагностики проекта. Вот пример сеанса, проходящего в интерактивной оболочке:

```
(ll_env)learning_log$ python manage.py shell
❶ >>> from learning_logs.models import Topic
>>> Topic.objects.all()
<QuerySet [<Topic: Chess>, <Topic: Rock Climbing>]>
```

Команда `python manage.py shell` (выполняемая в активной виртуальной среде) запускает интерпретатор Python, который может использоваться для работы с информацией в базе данных проекта. В данном случае мы импортируем модель `Topic` из модуля `learning_logs.models` ❶. Затем метод `Topic.objects.all()` используется для получения всех экземпляров модели `Topic`; возвращаемый список называется *итоговым набором* (queryset).

Содержимое итогового набора перебирается точно так же, как и содержимое списка. Например, просмотр идентификаторов, назначенных каждому объекту темы, выполняется так:

```
>>> topics = Topic.objects.all()
>>> for topic in topics:
... print(topic.id, topic)
...
1 Chess
2 Rock Climbing
```

Итоговый набор сохраняется в `topics`, после чего выводятся атрибут `id` каждого объекта `topic` и его строковое представление. Мы видим, что теме `Chess` присвоен идентификатор 1, а `Rock Climbing` — идентификатор 2.

Зная идентификатор конкретного объекта, можно получить его с помощью метода `Topic.objects.get()` и проанализировать содержащиеся в нем атрибуты. Просмотрим значения `text` и `date_added` для темы `Chess`:

```
>>> t = Topic.objects.get(id=1)
>>> t.text
'Chess'
>>> t.date_added
datetime.datetime(2022, 5, 20, 3, 33, 36, 928759,
tzinfo=datetime.timezone.utc)
```

Кроме того, можно просмотреть записи, относящиеся к конкретной теме. Ранее мы определили атрибут `topic` для модели `Entry`. Он был экземпляром `ForeignKey`, представляющим связь между записью и темой. Django может использовать эту связь для получения всех записей, относящихся к некой теме:

```
❶ >>> t.entry_set.all()
<QuerySet [<Entry: The opening is the first part of the game, roughly...>,
<Entry:
In the opening phase of the game, it's important t...>]>
```

Чтобы получить данные через отношение по внешнему ключу, используйте имя связанной модели, записанное в нижнем регистре, за которым следует символ подчеркивания и слово `set` ❶. Допустим, у вас есть модели `Pizza` и `Topping`, и вторая связана с первой через внешний ключ. Если ваш объект называется `my_pizza`, то для получения всех связанных с ним экземпляров `Topping` используется выражение `my_pizza.topping_set.all()`.

Мы будем задействовать такой синтаксис при переходе к программированию страниц, которые могут запрашивать пользователи. Оболочка очень удобна, когда вы хотите проверить, получает ли ваш код нужные данные. Если в оболочке код работает как задумано, то можно ожидать, что он будет правильно работать и в файлах, которые вы создаете в своем проекте. Если код выдает ошибки или не загружает данные, которые должен загружать, то вам будет намного проще отладить его в простой оболочке, чем при работе с файлами, генерирующими веб-страницы. В книге мы не будем часто возвращаться к оболочке, но вам не стоит забывать о ней — это полезный инструмент, который поможет вам освоить синтаксис Django, позволяющий работать с данными проекта.

При каждом изменении модели необходимо перезапустить оболочку, чтобы увидеть результаты этих изменений. Чтобы завершить сеанс работы с оболочкой, нажмите сочетание клавиш `Ctrl+D`; в Windows нажмите сочетание клавиш `Ctrl+Z`, а затем клавишу `Enter`.

## УПРАЖНЕНИЯ

**18.2. Короткие записи.** Метод `__str__()` в модели `Entry` в настоящее время присоединяет многоточие к каждому экземпляру `Entry`, отображаемому Django на административном сайте или в оболочке. Добавьте в метод `__str__()` оператор `if`, добавляющий многоточие только для записей, длина которых превышает 50 символов. Воспользуйтесь административным сайтом, чтобы ввести запись длиной менее 50 символов, и убедитесь, что при ее просмотре многоточие не отображается.

**18.3. Django API.** При написании кода для работы с данными проекта вы создаете запрос. Просмотрите документацию по созданию запросов к данным, доступную

по адресу <https://docs.djangoproject.com/en/4.1/topics/db/queries>. Многое из того, что вы увидите, покажется новым, но эта информация пригодится, когда вы начнете работать над собственными проектами.

**18.4. Пиццерия.** Создайте новый проект `pizzeria_project`, содержащий приложение `pizzas`. Определите модель `Pizza` с полем `name`, в котором хранятся названия видов пиццы (например, «Гавайская» (`Hawaiian`) или «Любители мяса» (`Meat Lovers`)). Определите модель `Topping` с полями `pizza` и `name`. Поле `pizza` должно содержать внешний ключ к модели `Pizza`, а поле `name` должно позволять хранить такие значения, как `pineapple`, `Canadian bacon` и `sausage`.

Зарегистрируйте обе модели на административном сайте. Используйте сайт для ввода названий пиццы и начинок. Изучите введенные данные в интерактивной оболочке.

## Создание страниц: главная страница «Журнала обучения»

Обычно процесс создания веб-страниц в Django состоит из трех стадий: определения URL, написания представлений и шаблонов. Сначала следует определить *схему* (pattern) URL. Она описывает структуру адреса и сообщает Django, на какие компоненты следует обращать внимание при сопоставлении запроса браузера с URL на сайте, чтобы выбрать возвращаемую страницу.

Затем каждый URL связывается с конкретным *представлением* (view) — функция представления читает и обрабатывает данные, необходимые странице. Функция представления часто вызывает *шаблон* (template), который создает страницу, подходящую для передачи браузеру. Чтобы вы лучше поняли, как работает этот механизм, создадим главную страницу приложения «Журнал обучения». Мы определим URL главной страницы, напомним для него функцию представления и создадим простой шаблон.

Сейчас мы всего лишь убеждаемся в том, что «Журнал обучения» работает как положено, поэтому страница пока останется простой. Когда приложение будет завершено, вы можете заниматься его оформлением сколько захотите; приложение, которое хорошо выглядит, но не работает, бессмысленно. Пока что на главной странице будут отображаться только заголовок и краткое описание.

### Связывание URL

Пользователь запрашивает страницы, вводя URL в браузере и щелкая на ссылках, поэтому мы должны решить, какие URL понадобятся в нашем проекте. Начнем с URL главной страницы: это базовый адрес, используемый для обращения к проекту. На данный момент базовый URL <http://localhost:8000/> возвращает сайт,



сгенерированный Django по умолчанию; он сообщает о том, что проект был создан успешно. Мы изменим главную страницу, связав базовый URL с главной страницей «Журнала обучения».

В главной папке проекта `ll_project` откройте файл `urls.py`. Вы увидите в нем следующий код:

#### *ll\_project/urls.py*

```
❶ from django.contrib import admin
 from django.urls import path

❷ urlpatterns = [
❸ path('admin/', admin.site.urls),
]
```

Первые две строки импортируют модуль `admin` и функцию для создания путей URL ❶. В теле файла определяется переменная `urlpatterns` ❷. В файле `urls.py`, представляющем проект в целом, переменная `urlpatterns` добавляет наборы URL из приложений в проект. Список содержит модуль `admin.site.urls`, определяющий все URL, которые могут запрашиваться с административного сайта ❸.

Добавим в этот файл URL для `learning_logs`:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
 path('admin/', admin.site.urls),
 path('', include('learning_logs.urls')),
]
```

Мы импортировали функцию `include()`, а также добавили строку кода для добавления модуля `learning_logs.urls`.

Файл `urls.py` по умолчанию находится в папке `ll_project`; теперь нужно создать второй файл `urls.py` в папке `learning_logs`. Создайте новый файл Python, сохраните его под именем `urls.py` в `learning_logs`, и добавьте в него следующий код:

#### *learning\_logs/urls.py*

```
❶ """Определяет схемы URL для learning_logs."""

❷ from django.urls import path

❸ from . import views

❹ app_name = 'learning_logs'
❺ urlpatterns = [
 # Главная страница
❻ path('', views.index, name='index'),
]
```

Чтобы было понятно, с какой версией `urls.py` мы работаем, в начало файла добавляется строка документации ❶. Затем импортируется функция `path`, она необходима для связывания URL с представлениями ❷. Кроме того, импортируется модуль `views` ❸; благодаря точке Python получает указание импортировать модуль `views.py` из каталога, в котором находится текущий модуль `urls.py`. Переменная `app_name` помогает Django отличить этот файл `urls.py` от одноименных файлов в других приложениях проекта ❹. Переменная `urlpatterns` в этом модуле представляет собой список страниц, которые могут запрашиваться из приложения `learning_logs` ❺.

Схема URL представляет собой вызов функции `path()` с тремя аргументами ❻. Первый содержит строку, которая помогает Django правильно маршрутизировать текущий запрос. Django получает запрашиваемый URL и пытается отобразить его на представление. Для этого он ищет среди всех определенных схем URL ту, которая соответствует текущему запросу. Базовый URL проекта (<http://localhost:8000/>) игнорируется, так что пустая строка совпадает с базовым URL. Любой другой URL не будет соответствовать этому выражению, и Django вернет страницу с ошибкой, если запрашиваемый URL не соответствует ни одной из существующих схем.

Второй аргумент в `path()` ❼ определяет вызываемую функцию из `views.py`. Когда запрашиваемый URL соответствует регулярному выражению, Django вызывает `index()` из `views.py` (мы напишем эту функцию представления в следующем подразделе). Третий аргумент определяет имя `index` для этой схемы URL, чтобы на нее можно было сослаться в других частях кода. Каждый раз, когда потребуется предоставить ссылку на главную страницу, мы будем использовать это имя вместо URL.

## Написание представления

Функция представления получает информацию из запроса, подготавливает данные, необходимые для создания страницы, и возвращает данные браузеру — часто с помощью шаблона, который определяет внешний вид страницы.

Файл `views.py` в папке `learning_logs` был сгенерирован автоматически при выполнении команды `python manage.py startapp`. На данный момент его содержимое выглядит так:

### **`views.py`**

```
from django.shortcuts import render
```

```
Создайте здесь свои представления.
```

Сейчас файл только импортирует функцию `render()`, которая генерирует ответ на основании данных, полученных от представлений. Откройте файл представления и добавьте следующий код главной страницы:

```
from django.shortcuts import render

def index(request):
 """Главная страница приложения "Журнал обучения"."""
 return render(request, 'learning_logs/index.html')
```

Если URL запроса совпадает с только что определенной схемой, Django ищет в файле `views.py` функцию представления `index()`, после чего передает ей объект `request`. В нашем случае никакая обработка данных для страницы не нужна, поэтому код функции сводится к вызову функции `render()`. Она использует два аргумента: исходный объект запроса и шаблон, применяемый для создания страницы. Напишем этот шаблон.

## Написание шаблона

Шаблон определяет общий внешний вид страницы, а Django заполняет его соответствующими данными при каждом запросе страницы. Шаблон может обращаться к любым данным, полученным от представления. Наше представление главной страницы никаких данных не предоставляет, поэтому шаблон получается относительно простым.

В папке `learning_logs` создайте новую папку `templates`, а в ней — папку `learning_logs`. На первый взгляд такая структура кажется избыточной (папка `learning_logs` в папке `templates` в папке `learning_logs`), но созданную таким образом структуру Django будет интерпретировать однозначно даже в контексте большого проекта, состоящего из множества отдельных приложений. Во внутренней папке `learning_logs` создайте новый файл `index.html` (таким образом, полное имя файла имеет вид `11_project/learning_logs/templates/learning_logs/index.html`). Добавьте в него следующий текст:

### *index.html*

```
<p>Learning Log</p>
```

```
<p>Learning Log helps you keep track of your learning, for any topic you're
interested in.</p>
```

Это очень простой файл. Если вы не знакомы с синтаксисом HTML, то теги `<p></p>` обозначают абзацы: `<p>` открывает абзац, а `</p>` закрывает его. Наша страница содержит два абзаца: в первом находится заголовок, а во втором — описание, что пользователь может сделать с помощью приложения «Журнал обучения».

Теперь при запросе базового URL проекта <http://localhost:8000/> вы увидите только что созданную страницу вместо страницы по умолчанию. Django берет запрошенный URL и видит, что он совпадает со схемой `' '`. В этом случае Django вызывает функцию `views.index()`, что приводит к созданию страницы с помощью шаблона, содержащегося в `index.html`. Полученная страница показана на рис. 18.3.

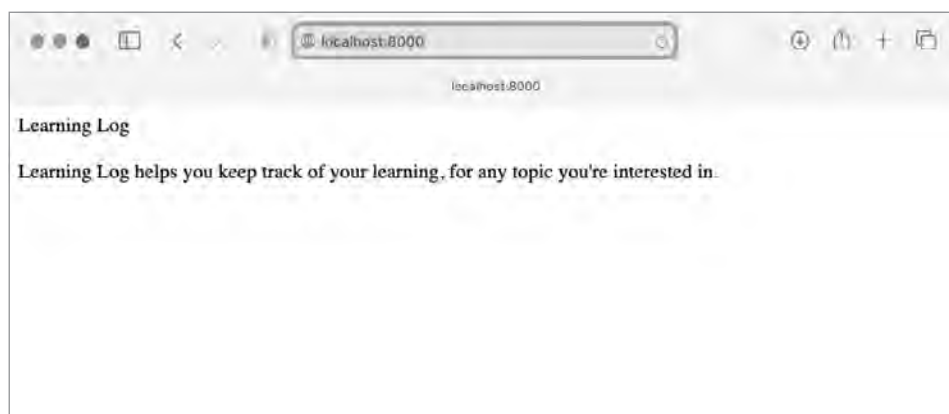


**Рис. 18.3.** Главная страница «Журнала обучения»

И хотя может показаться, что для одной страницы этот процесс слишком сложен, такое разделение URL, представлений и шаблонов очень удобно. Оно позволяет сосредоточиться на отдельных аспектах проекта, а в более крупных проектах некоторые участники могут сосредоточиться на областях, в которых они наиболее сильны. Например, специалист по базам данных может заняться моделями, программист — кодом представления, а веб-дизайнер — шаблонами.

### ПРИМЕЧАНИЕ

Вы можете получить следующее сообщение об ошибке:

```
ModuleNotFoundError: No module named 'learning_logs.urls'
```

В таком случае остановите сервер разработки, нажав сочетание клавиш `Ctrl+C` в терминальном окне, в котором была введена команда `gunserver`. Затем снова введите команду `python manage.py gunserver`. Каждый раз, когда вы сталкиваетесь с подобными ошибками, попробуйте остановить и перезапустить сервер.

### УПРАЖНЕНИЯ

**18.5. План питания.** Представьте приложение для составления плана питания на неделю. Создайте новую папку `meal_planner`, затем создайте в ней новый проект Django. Создайте новое приложение `meal_plans`. Сформируйте простую главную страницу для этого проекта.

**18.6. Главная страница пиццерии.** Добавьте главную страницу в проект «Пиццерия», который вы начали создавать в упражнении 18.4.

## Создание других страниц

Теперь, когда вы начали представлять процесс создания страниц, можно переходить к проекту «Журнал обучения». Мы создадим две страницы для вывода данных: на одной будет отображаться список всех тем, а на другой — все записи по конкретной теме. Для каждой страницы мы добавим схему URL, напомним функцию представления и создадим шаблон. Но прежде чем переходить к работе, нужно создать базовый шаблон, от которого смогут наследовать все шаблоны этого проекта.

### Наследование шаблонов

При создании сайта некоторые элементы почти всегда повторяются на каждой странице. Вместо того чтобы встраивать эти элементы непосредственно в страницы, вы можете написать базовый шаблон с повторяющимися элементами; все страницы будут наследовать от него. Такое решение позволит сосредоточиться на разработке уникальных аспектов каждой страницы и существенно упростит изменение общего оформления проекта в целом.

### Родительский шаблон

Начнем с создания шаблона `base.html` в одном каталоге с файлом `index.html`. Этот файл будет содержать элементы, общие для всех страниц; все остальные шаблоны наследуют от `base.html`. Пока единственным элементом, который должен повторяться на каждой странице, остается заголовок в верхней части страницы. Шаблон будет добавляться в каждую страницу, поэтому преобразуем заголовок в ссылку на главную страницу:

#### *base.html*

```
<p>
```

```
❶ Learning Log
```

```
</p>
```

```
❷ {% block content %}{% endblock content %}
```

Первая часть файла создает абзац с именем проекта, который также работает как ссылка на главную страницу. Для создания ссылки использовался *шаблонный тег* (template tag), обозначенный фигурными скобками и знаками процента `{% %}`. Шаблонный тег представляет собой блок кода, который генерирует информацию для вывода на странице. В данном примере шаблонный тег `{% url 'learning_logs:index' %}` генерирует URL, соответствующий схеме URL с именем `'index'`, определенной в файле `learning_logs/urls.py` ❶. В данном примере `learning_logs` — пространство имен, а `index` — схема URL, имеющая уникальное имя в этом пространстве. Данное пространство определяется значением, присвоенным переменной `app_name` в файле `learning_logs/urls.py`.

В этой простой странице HTML ссылка заключается в *якорный тег* (anchor tag) `<a>`:

```
текст ссылки
```

Генерирование URL с помощью шаблонного тега сильно упрощает актуализацию ссылок. Чтобы изменить URL в проекте, достаточно изменить схему URL в `urls.py`, а Django автоматически вставит обновленный URL при следующем запросе страницы. Каждая страница в проекте будет наследовать от `base.html`, так что в дальнейшем на каждой странице будет содержаться ссылка на главную страницу.

В последней строке вставляется пара тегов **block** ❷. Блок `content` резервирует место; попадающая в него информация будет определяться дочерним шаблоном.

Дочерний шаблон не обязан определять каждый блок в своем родителе, так что в родительских шаблонах можно зарезервировать место для любого количества блоков, а дочерний шаблон будет использовать столько из них, сколько потребуется.

### ПРИМЕЧАНИЕ

В коде Python почти всегда используются отступы в четыре пробела. Файлы шаблонов обычно имеют больший уровень вложенности, чем файлы Python, поэтому каждый уровень отступа обычно обозначается двумя пробелами. Будьте внимательны и действуйте последовательно.

### Дочерний шаблон

Теперь нужно переписать файл `index.html` так, чтобы он наследовал от `base.html`. Обновленный файл `index.html` выглядит так:

*index.html*

- ```
❶ {% extends "learning_logs/base.html" %}

❷ {% block content %}
    <p>Learning Log helps you keep track of your learning, for any topic
    you're interested in.</p>
❸ {% endblock content %}
```

Сравнивая этот файл с исходной версией `index.html`, мы видим, что заголовок `Learning Log` заменен кодом наследования от родительского шаблона ❶. В первой строке дочернего шаблона должен находиться тег `{% extends %}`, который сообщает Django, от какого родительского шаблона он наследует. Файл `base.html` является частью `learning_logs`, поэтому имя папки `learning_logs` добавляется в путь к родительскому шаблону. Эта строка извлекает все содержимое из шаблона `base.html` и позволяет `index.html` определить, что должно попасть в пространство, зарезервированное блоком `content`.

Блок `content` определяется вставкой тега `{% block %}` с именем `content` ❷. Все, что не наследуется от родительского шаблона, попадает в блок `content`. В данном случае это абзац с описанием проекта «Журнал обучения». О том, что определение `content`

завершено, мы сообщаем с помощью тега `{% endblock content %}` ❸. Наличие имени у тега `{% endblock %}` не обязательно, но если шаблон увеличится и будет содержать несколько блоков, то будет полезно сразу видеть, какой именно блок завершается.

Вероятно, вы уже начинаете понимать преимущества наследования шаблонов: в дочерний шаблон достаточно добавить информацию, уникальную для этой страницы. Такой подход упрощает не только каждый шаблон, но и сам процесс изменения сайта. Чтобы изменить элемент, общий для многих страниц, достаточно поправить элемент в родительском шаблоне. Внесенные изменения будут автоматически перенесены на каждую страницу, наследующую от этого шаблона. В проекте из десятков и сотен страниц такая структура позволяет значительно упростить и ускорить доработку сайта.

В больших проектах часто создаются один родительский шаблон `base.html` для всего сайта и родительские шаблоны для каждого его крупного раздела. Все шаблоны разделов наследуют от `base.html`, и каждая страница сайта наследует от шаблона раздела. При такой структуре вы сможете легко изменять оформление и поведение самого сайта, любого его раздела или отдельной страницы. Данная конфигурация позволяет сильно повысить эффективность работы и стимулирует разработчика к дальнейшему совершенствованию своего проекта.

Страница со списком тем

Разобравшись с тем, как эффективно организовать создание страниц, мы можем сосредоточиться на следующих двух страницах: списке всех тем и списке записей по одной теме. На странице тем выводится перечень всех тем, созданных пользователями, и это первая страница, на которой нам придется работать с данными.

Схема URL для тем

Сначала нужно определить URL для страницы тем. Обычно в таких случаях выбирается простой фрагмент URL, который отражает суть информации, представленной на странице. Мы возьмем слово `topics`, так что для получения страницы будет использоваться URL <http://localhost:8000/topics/>. А вот какие изменения следует внести в файл `learning_logs/urls.py`:

`learning_logs/urls.py`

```
"""Определяет схемы URL для learning_logs."""
--пропуск--
urlpatterns = [
    # Главная страница.
    path('', views.index, name='index'),
    # Страница со списком всех тем.
    ❶ path('topics/', views.topics, name='topics'),
]
```

Мы просто добавили `topics/` в аргумент регулярного выражения, используемый с URL главной страницы ❶. Когда Django проверяет запрашиваемый URL, эта

схема совпадет с любым URL, который состоит из базового URL и слова `topics`. Косую черту в конце можно либо добавить, либо нет, но после слова `topics` ничего быть не должно, иначе схема не совпадет. Любой запрос с URL, соответствующим этой схеме, будет передан функции `topics()` в файле `views.py`.

Представление тем

Функция `topics()` должна получать данные из базы данных и отправлять их шаблону. Обновленная версия файла `views.py` выглядит так:

views.py

```
from django.shortcuts import render
```

```
❶ from .models import Topic
```

```
def index(request):  
    --пропуск--
```

```
❷ def topics(request):  
    """Выводит список тем."""
```

```
❸     topics = Topic.objects.order_by('date_added')
```

```
❹     context = {'topics': topics}
```

```
❺     return render(request, 'learning_logs/topics.html', context)
```

Сначала импортируется модель, связанная с нужными данными ❶. Функции `topics()` необходим один параметр: объект `request`, полученный Django от сервера ❷. База данных получает запрос на получение объектов `Topic`, отсортированных по атрибуту `date_added` ❸. Полученный итоговый набор сохраняется в `topics`.

Затем определяется контекст, который будет передаваться шаблону ❹. *Контекст* (`context`) представляет собой словарь, в котором ключами являются имена, используемые в шаблоне для обращения к данным, а значениями — данные, которые должны передаваться шаблону. В данном случае существует всего одна пара «ключ — значение», которая содержит набор тем, отображаемых на странице. При создании страницы, использующей данные, функции `render()` передается словарь `context`, а также объект `request` и путь к шаблону ❺.

Шаблон тем

Шаблон страницы со списком тем получает словарь `context`, чтобы использовать данные, предоставленные функцией `topics()`. Создайте файл `topics.html` в одном каталоге с `index.html`. Вывод списка тем в шаблоне осуществляется следующим образом:

topics.html

```
{% extends "learning_logs/base.html" %}
```

```
{% block content %}
```

```
<p>Topics</p>
```



```

❶ <ul>
❷     {% for topic in topics %}
❸         <li>{{ topic }}</li>
❹     {% empty %}
❺         <li>No topics have been added yet.</li>
❻     {% endfor %}
❻ </ul>

{% endblock content %}

```

Сначала тег `{% extends %}` объявляет о наследовании от `base.html`, как и в случае с шаблоном `index`, после чего открывается блок `content`. Тело страницы содержит маркированный (bulleted) список введенных тем. В стандартном языке HTML маркированный список называется *неупорядоченным* (unordered) и обозначается тегами `<ul></ul>`. Список тем начинается с открывающегося тега `<ul>` ❶.

Далее шаблонный тег, эквивалентный циклу `for`, применяется для перебора списка `topics` из словаря `context` ❷. Код, используемый в шаблоне, отличается от Python несколькими важными моментами. В Python для обозначения строк, входящих в тело цикла, используются отступы. В шаблоне каждый цикл `for` должен иметь явный тег `{% endfor %}`, обозначающий конец цикла. Таким образом, в шаблонах часто встречаются циклы следующего вида:

```

{% for элемент in список %}
    действия для каждого элемента
{% endfor %}

```

В цикле каждая тема должна быть преобразована в элемент маркированного списка. Чтобы вывести значение переменной в шаблоне, заключите ее имя в двойные фигурные скобки. Они на странице не появятся и всего лишь сообщают Django об использовании шаблонной переменной. Код `{{ topic }}` ❸ будет заменен значением `topic` при каждом проходе цикла. Тег HTML `<li></li>` обозначает *элемент списка* (list item). Все, что находится между тегами, в паре тегов `<ul></ul>`, будет отображаться как элемент маркированного списка.

Шаблонный тег `{% empty %}` ❹ сообщает Django, что делать при отсутствии элементов в списке. В нашем примере выводится сообщение о том, что темы еще не созданы. Последние две строки завершают цикл `for` ❺ и маркированный список ❻.

Теперь необходимо изменить базовый шаблон и вставить ссылку на страницу с темами. Добавьте следующий код в `base.html`:

base.html

```

<p>
❶ <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
❷ <a href="{% url 'learning_logs:topics' %}">Topics</a>
</p>

{% block content %}{% endblock content %}

```

Сначала добавляется ссылка на главную страницу ❶, а затем дефис, после которого вставляется ссылка на страницу тем, которая также представлена шаблонным тегом `{% url %}` ❷. Эта строка дает Django указание сгенерировать ссылку, соответствующую схеме URL с именем 'topics' в файле `learning_logs/urls.py`.

Обновив главную страницу в браузере, вы увидите ссылку **Topics** (Темы). Щелчок на ней открывает страницу, похожую на рис. 18.4.



Рис. 18.4. Страница со списком тем

Страницы отдельных тем

Далее создадим страницу для вывода информации по одной теме, с названием темы и всеми записями по этой теме. Мы снова определим новую схему URL, напомним представление и создадим шаблон. Кроме того, на странице со списком тем каждый элемент маркированного списка будет преобразован в ссылку на соответствующую страницу отдельной темы.

Схема URL для отдельных тем

Схема URL для страницы отдельной темы немного отличается от других схем URL, которые встречались нам ранее, поскольку в ней используется атрибут `id` темы для обозначения запрашиваемой темы. Например, если пользователь хочет просмотреть страницу с подробной информацией по теме **Chess** (`id=1`), эта страница будет иметь URL `http://localhost:8000/topics/1/`. Вот как выглядит схема для этого URL из файла `learning_logs/urls.py`:

learning_logs/urls.py

```
--пропуск--
urlpatterns = [
--пропуск--
```

```
# Страница с подробной информацией по отдельной теме.  
path('topics/<int:topic_id>/', views.topic, name='topic'),  
]
```

Рассмотрим строку `'topics/<int:topic_id>/'` в этой схеме URL. Первая часть строки сообщает Django, что искать следует URL, у которых за базовым адресом идет слово `topics`. Вторая часть строки, `/<int:topic_id>/`, описывает целое число, помещенное между двумя косыми чертами; оно сохраняется в аргументе `topic_id`.

Когда Django находит URL, соответствующий этой схеме, вызывается функция представления `topic()`, в аргументе которой передается значение, хранящееся в `topic_id`. Значение `topic_id` используется для получения нужной темы внутри функции.

Представление отдельной темы

Функция `topic()` должна получить тему и все связанные с ней записи из базы данных, подобно тому как мы делали это ранее в оболочке Django:

views.py

--пропуск--

```
❶ def topic(request, topic_id):  
    """Выводит одну тему и все ее записи."""  
❷    topic = Topic.objects.get(id=topic_id)  
❸    entries = topic.entry_set.order_by('-date_added')  
❹    context = {'topic': topic, 'entries': entries}  
❺    return render(request, 'learning_logs/topic.html', context)
```

Это первая функция представления, которой требуется параметр, отличающийся от объекта `request`. Функция получает значение, совпавшее с выражением `/<int:topic_id>/`, и сохраняет его в `topic_id` ❶. Затем функция `get()` используется для получения темы, по аналогии с тем, как мы это делали в оболочке Django ❷. Далее загружаются записи, связанные с данной темой, и они упорядочиваются по значению `date_added` ❸: благодаря знаку «минус» перед `date_added` результаты сортируются в обратном порядке, то есть самые последние записи будут находиться на первых местах. Тема и записи сохраняются в словаре `context` ❹, после чего вызывается функция `render()` с объектом запроса, шаблоном `topic.html` и словарем `context` ❺.

ПРИМЕЧАНИЕ

Выражения в строках ❷ и ❸, которые обращаются к базе данных за конкретной информацией, называются запросами. Когда вы пишете подобные запросы для своих проектов, сначала опробуйте их в оболочке Django. Вы сможете проверить результат намного быстрее, чем если напишете представление и шаблон, а затем проверите результаты в браузере.

Шаблон отдельной темы

В шаблоне должны отображаться название темы и текст записей. Кроме того, необходимо отправить пользователю сообщение, если по теме еще не было сделано ни одной записи:

topic.html

```
{% extends 'learning_logs/base.html' %}

{% block content %}

❶ <p>Topic: {{ topic }}</p>

    <p>Entries:</p>
❷ <ul>
❸ {% for entry in entries %}
    <li>
❹     <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
❺     <p>{{ entry.text|linebreaks }}</p>
    </li>
❻ {% empty %}
    <li>
        There are no entries for this topic yet.
    </li>
    {% endfor %}
</ul>

{% endblock content %}
```

Шаблон расширяет `base.html`, как и для всех страниц проекта. Затем выводится атрибут `text` темы, которая была запрошена ❶. Переменная `topic` доступна, поскольку добавлена в словарь `context`. Затем создается маркированный список со всеми записями по теме ❷; перебор записей осуществляется так же, как это делалось ранее для тем ❸.

С каждым элементом списка связываются два значения: временная метка и полный текст каждой записи. Для временной метки ❹ выводится значение атрибута `date_added`. В шаблонах Django вертикальная черта (`|`) представляет *фильтр* — функцию, изменяющую значение шаблонной переменной. Фильтр `date:'M d, Y H:i'` выводит временные метки в формате *January 1, 2022 23:00*. Следующая строка выводит полное значение атрибута `text` текущей записи. Фильтр `linebreaks` ❺ следит за тем, чтобы длинный текст содержал разрывы строк в формате, поддерживаемом браузером (вместо блока непрерывного текста). Шаблонный тег `{% empty %}` ❻ используется для вывода сообщения об отсутствии записей.

Ссылки на странице

Прежде чем просматривать страницу отдельной темы в браузере, необходимо изменить шаблон списка так, чтобы каждая тема вела на соответствующую страницу. Внесите в файл `topics.html` следующие изменения:

topics.html

```
--пропуск--
{% for topic in topics %}
    <li>
        <a href="{% url 'learning_logs:topic' topic.id %}">
            {{ topic }}</a>
        </li>
    {% empty %}
--пропуск--
```

Шаблонный тег URL используется для генерирования ссылки на основании схемы URL с именем 'topic' из `learning_logs`. Этой схеме необходим аргумент `topic_id`, поэтому в шаблонный тег URL добавляется атрибут `topic.id`. Теперь каждая тема в списке представляет собой ссылку на страницу темы — например, <http://localhost:8000/topics/1/>.

Если теперь обновить страницу тем и щелкнуть на теме, то открывается страница, изображенная на рис. 18.5.

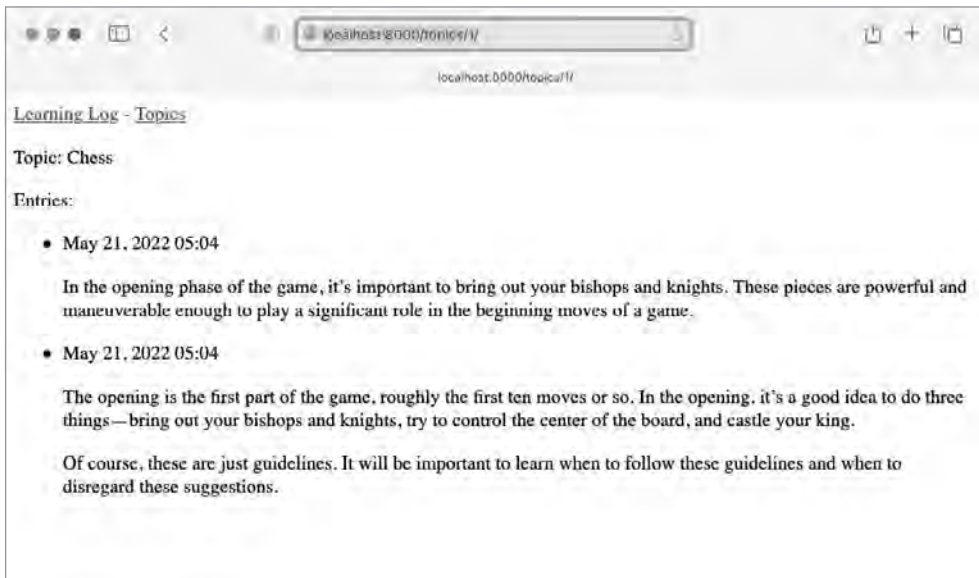


Рис. 18.5. Страница со списком всех записей по отдельной теме

ПРИМЕЧАНИЕ

Между `topic.id` и `topic_id` существует неочевидное, но важное различие. Выражение `topic.id` проверяет тему и получает значение соответствующего идентификатора. Переменная `topic_id` содержит ссылку на этот идентификатор в коде. Если вы столкнетесь с ошибками при работе с идентификаторами, то убедитесь, что эти выражения используются правильно.

УПРАЖНЕНИЯ

18.7. Документация шаблонов. Просмотрите документацию по шаблонам Django, доступную по адресу <https://docs.djangoproject.com/en/2.2/ref/templates/>. Используйте ее в работе над собственными проектами.

18.8. Страницы проекта «Пиццерия». Добавьте страницу в проект Pizzeria из упражнения 18.6, на которой показываются названия видов пиццы. Свяжите каждое название со страницей, на которой выводится список начинок к этой пицце. Обязательно примените наследование шаблонов, чтобы повысить эффективность создания страниц.

Резюме

В этой главе вы начали осваивать создание веб-приложений с помощью инфраструктуры Django. Вы написали короткую спецификацию проекта, установили Django в виртуальной среде, узнали, как настроить проект, и проверили правильность настройки. Вы узнали, как создать приложение и определить модели для представления данных в вашем приложении. Кроме того, вы познакомились с базами данных и узнали, как Django упрощает миграцию баз данных после внесения изменений в модель. Вы научились создавать суперпользователей для административного сайта, а также использовали такой сайт для ввода исходных данных.

Помимо этого, в данной главе была представлена оболочка Django, позволяющая работать с данными проекта в терминальном сеансе. Вы научились определять URL, создавать функции представления и писать шаблоны для создания страниц сайта. Наконец, вы применили механизм наследования шаблонов, который упрощает структуру отдельных шаблонов и облегчает изменение сайта по мере развития проекта.

В главе 19 вы создадите интуитивно понятные, удобные страницы, на которых пользователи смогут добавлять новые темы и записи, а также редактировать существующие записи без участия административного сайта. Кроме того, вы добавите систему регистрации пользователей, чтобы любой из них мог создать учетную запись и вести свой журнал. Собственно, в этом и заключается сущность веб-приложения — создание функциональности, которую может применять любое количество пользователей.

19

Учетные записи пользователей



Что является самым главным для веб-приложения? Возможность для любого пользователя, живущего в любой стране мира, создать учетную запись в вашем приложении и начать работать с ним. В этой главе мы создадим формы, в которых пользователи смогут вводить свои темы и записи, а также редактировать существующие данные. Кроме того, вы узнаете, как Django защищает приложения от распространенных атак на страницы с формами, чтобы вам не приходилось тратить много времени на продумывание средств защиты вашего приложения.

Затем будет реализована система проверки пользователей. Мы создадим страницу регистрации, на которой пользователи смогут создавать учетные записи, и ограничим доступ к некоторым страницам для анонимных пользователей. Затем некоторые функции представления будут изменены так, чтобы пользователь мог видеть только собственные данные. Вы узнаете, как обеспечить безопасность и конфиденциальность данных пользователей.

Введение данных пользователями

Прежде чем создавать систему аутентификации пользователей, позволяющую заводить учетные записи, сначала добавим несколько страниц, на которых пользователи смогут вводить собственные данные. У них появится возможность создавать новые темы, добавлять новые записи и редактировать сделанные ранее.

В настоящее время данные может вводить только суперпользователь на административном сайте. Однако разрешать пользователям работать на нем явно нежелательно, поэтому мы воспользуемся средствами создания форм Django для создания страниц, на которых пользователи смогут вводить данные.

Добавление новых тем

Начнем с возможности создания новых тем. Страницы на базе форм добавляются практически так же, как и те страницы, которые мы уже создали ранее: вы определяете URL, пишете функцию представления и создаете шаблон. Принципиальное отличие — добавление нового модуля `forms.py`, содержащего функциональность форм.

Объект `ModelForm`

Любая страница, на которой пользователь может вводить и отправлять информацию, является *формой*, даже если на первый взгляд она на форму не похожа. Когда пользователь вводит информацию, необходимо *проверить* (validate), что он ввел корректные данные, а не вредоносный код (например, нарушающий работу сервера). Затем проверенная информация обрабатывается и сохраняется в нужном месте базы данных. Django автоматизирует большую часть этой работы.

Простейший способ создания форм в Django основан на использовании класса `ModelForm`, который автоматически создает форму на основании моделей, определенных в главе 18. Ваша первая форма будет создана в файле `forms.py`, который должен находиться в одном каталоге с `models.py`:

`forms.py`

```
from django import forms

from .models import Topic
```

```
❶ class TopicForm(forms.ModelForm):
    class Meta:
❷         model = Topic
❸         fields = ['text']
❹         labels = {'text': ''}
```

Сначала импортируется модуль `forms` и модель, с которой мы будем работать: `Topic`. В строке ❶ определяется класс `TopicForm`, наследующий от `forms.ModelForm`.

Простейшая версия `ModelForm` состоит из вложенного класса `Meta`, который сообщает Django, на какой модели должна базироваться форма и какие поля должны на ней находиться. Форма создается на базе модели `Topic` ❷, а на ней размещается только поле `text` ❸. Благодаря пустой строке в словаре `labels` Django получает указание не генерировать подпись для текстового поля ❹.

URL для страницы `new_topic`

URL новой страницы должен быть простым и содержательным, поэтому после того, как пользователь выбрал команду создания новой темы, его направляют по адресу

http://localhost:8000/new_topic/. Ниже приведена схема URL для страницы `new_topic`, которая добавляется в файл `learning_logs/urls.py`:

learning_logs/urls.py

```
--nponyск--
urlpatterns = [
    --nponyск--
    # Страница для добавления новой темы.
    path('new_topic/', views.new_topic, name='new_topic'),
]
```

Эта схема URL будет отправлять запросы функции представления `new_topic()`, которую мы сейчас напишем.

Функция представления `new_topic()`

Функция `new_topic()` должна обрабатывать две разные ситуации: исходные запросы страницы `new_topic` (в этом случае должна отображаться пустая форма) и обработка данных, отправленных в форме. Затем функция должна перенаправить пользователя обратно на страницу `topics`:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm

--nponyск--
def new_topic(request):
    """Добавляет новую тему."""
    ❶ if request.method != 'POST':
        # Данные не отправлялись; создается пустая форма.
        ❷ form = TopicForm()
    else:
        # Отправлены данные POST; обработать данные.
        ❸ form = TopicForm(data=request.POST)
        ❹ if form.is_valid():
            ❺ form.save()
            ❻ return redirect('learning_logs:topics')

    # Вывести пустую или недействительную форму.
    ❼ context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
```

Мы импортируем класс `HttpResponseRedirect`, который будет использоваться для перенаправления пользователя к странице `topics` после отправки введенной темы. Функция `reverse()` определяет URL по заданной схеме URL (то есть Django сгенерирует URL при запросе страницы). Вдобавок импортируется только что написанная форма `TopicForm`.

Запросы *GET* и *POST*

При создании веб-приложений применяются два основных типа запросов: *GET* и *POST*. Запросы *GET* используются для страниц, которые только читают данные с сервера, а запросы *POST* обычно используются в тех случаях, когда пользователь должен отправить информацию в форме. Для обработки всех наших форм будет использоваться метод *POST* (существуют и другие разновидности запросов, но в нашем проекте они не используются).

Функция `new_topic()` получает в параметре объект запроса. Когда пользователь впервые запрашивает эту страницу, его браузер отправляет запрос *GET*. Когда пользователь уже заполнил и отправил форму, его браузер отправляет запрос *POST*. В зависимости от типа запроса мы определяем, запросил пользователь пустую форму (запрос *GET*) или предлагает обработать заполненную (запрос *POST*).

Метод запроса — *GET* или *POST* — проверяется условием `if` в строке ❶. Если метод отличается от *POST*, то, вероятно, используется запрос *GET*, поэтому необходимо вернуть пустую форму (даже если это запрос другого типа, это все равно безопасно). Мы создаем экземпляр `TopicForm` ❷, сохраняем его в переменной `form` и отправляем форму шаблону в словаре `context` ❸. При создании `TopicForm` аргументы не передавались, поэтому Django создает пустую форму, которая заполняется пользователем.

Если используется метод запроса *POST*, то выполняется блок `else`, который обрабатывает данные, отправленные в форме. Мы создаем экземпляр `TopicForm` ❹ и передаем ему данные, введенные пользователем, хранящиеся в `request.POST`. Возвращаемый объект `form` содержит информацию, отправленную пользователем.

Отправленную информацию нельзя сохранять в базе данных до тех пор, пока она не будет проверена ❺. Функция `is_valid()` проверяет, что все обязательные поля были заполнены (все поля формы по умолчанию являются обязательными), а введенные данные соответствуют типам полей — например, что длина текста меньше 200 символов, как было указано в файле `models.py` в главе 18. Автоматическая проверка избавляет нас от большого объема работы. Если все данные действительны, то можно вызвать метод `save()` ❻, который записывает данные из формы в базу данных.

После того как данные будут сохранены, страницу можно покинуть. Функция `redirect()` принимает имя представления и перенаправляет пользователя на связанную с ним страницу. Мы используем вызов `redirect()` для перенаправления браузера на страницу `topics` ❼, на которой пользователь увидит только что введенную им тему в общем списке тем.

Переменная `context` определяется в конце функции представления ❽, а страница создается на базе шаблона `new_topic.html`, который будет создан на следующем шаге. Код размещается за пределами любых блоков `if`; он выполняется при создании пустой формы, а также при определении того, что отправленная форма была недействительной. Недействительная форма содержит стандартные сообщения об ошибках, чтобы помочь пользователю передать действительные данные.

Шаблон `new_topic`

Теперь создадим новый шаблон `new_topic.html` для отображения только что созданной формы:

`new_topic.html`

```
{% extends "learning_logs/base.html" %}

{% block content %}
    <p>Add a new topic:</p>

    ❶ <form action="{% url 'learning_logs:new_topic' %}" method='post'>
      ❷   {% csrf_token %}
      ❸   {{ form.as_p }}
      ❹   <button name="submit">add topic</button>
    </form>

{% endblock content %}
```

Этот шаблон расширяет `base.html`, поэтому имеет такую же базовую структуру, как и остальные страницы «Журнала обучения». Сначала определяется форма HTML ❶ с помощью тегов `<form></form>`. Аргумент `action` сообщает серверу, куда передавать данные, отправленные формой; в данном случае они возвращаются функции представления `new_topic()`. Аргумент `method` дает браузеру указание отправить данные в запросе типа POST.

Django использует шаблонный тег `{% csrf_token %}` ❷ для предотвращения попыток получения несанкционированного доступа к серверу (атаки такого рода называются *межсайтовой подделкой запросов* (cross-site request forgery)). Далее отображается форма; это наглядный пример того, насколько легко в Django выполняются такие стандартные операции, как отображение формы. Чтобы автоматически создать все необходимые для этой операции поля, достаточно добавить шаблонную переменную `{{ form.as_p }}` ❸. Модификатор `as_p` дает Django указание отобразить все элементы формы в формате абзацев (`<div></div>`) — это простой способ аккуратного отображения формы.

Django не создает кнопку отправки данных для форм, поэтому мы определяем ее, прежде чем закрыть форму ❹.

Создание ссылки на страницу `new_topic`

Далее ссылка на страницу `new_topic` создается на странице `topics`:

`topics.html`

```
{% extends "learning_logs/base.html" %}

{% block content %}

    <p>Topics</p>
```

```
<ul>
  --пропуск--
</ul>
```

```
<a href="{% url 'learning_logs:new_topic' %}">Add a new topic:</a>
```

```
{% endblock content %}
```

Поместите ссылку после списка существующих тем. Полученная форма изображена на рис. 19.1. Воспользуйтесь ею и добавьте несколько своих тем.



Рис. 19.1. Страница для добавления новой темы

Добавление новых записей

Теперь, когда пользователь может добавлять новые темы, он захочет добавлять и новые записи. Мы снова определим URL, напомним новую функцию, шаблон и создадим ссылку на страницу. Но сначала нужно добавить в файл `forms.py` еще один класс.

Класс *EntryForm*

Мы должны создать форму, связанную с моделью `Entry`, но более специализированную по сравнению с `TopicForm`:

forms.py

```
from django import forms
```

```
from .models import Topic, Entry
```

```
class TopicForm(forms.ModelForm):
  --пропуск--
```

```
class EntryForm(forms.ModelForm):
    class Meta:
        model = Entry
        fields = ['text']
        labels = {'text': ''}
        widgets = {'text': forms.Textarea(attrs={'cols': 80})}
```

Сначала в команду `import` к `Topic` добавляется `Entry`. Новый класс `EntryForm` наследует от `forms.ModelForm` и содержит вложенный класс `Meta` с указанием модели, на которой он базируется, и поле, добавляемое в форму. Полю `'text'` снова назначается пустая надпись ❶.

Для класса `EntryForm` мы добавляем атрибут `widgets` ❷. *Виджет* (widget) представляет собой элемент формы HTML: однострочное или многострочное текстовое поле, раскрывающийся список и т. д. Добавляя атрибут `widgets`, вы можете переопределить виджеты, выбранные Django по умолчанию. Давая Django указание использовать элемент `forms.Textarea`, мы настраиваем виджет ввода для поля `'text'`, чтобы ширина текстовой области составляла 80 столбцов вместо значения по умолчанию 40. У пользователя будет достаточно места для создания содержательных записей.

URL для `new_entry`

Необходимо добавить аргумент `topic_id` в URL для создания новой записи, поскольку запись должна быть связана с конкретной темой. Вот как выглядит URL, который мы добавляем в файл `learning_logs/urls.py`:

`learning_logs/urls.py`

```
--пропуск--
urlpatterns = [
    --пропуск--
    # Страница для добавления новой записи.
    path('new_entry/<int:topic_id>', views.new_entry, name='new_entry'),
]
```

Эта схема URL соответствует любому URL в форме http://localhost:8000/new_entry/id/, где `id` — число, равное идентификатору темы. Код `<int:topic_id>` захватывает числовое значение и сохраняет его в переменной `topic_id`. При запросе URL, соответствующего этой схеме, Django передает запрос и идентификатор темы функции представления `new_entry()`.

Функция представления `new_entry()`

Данная функция очень похожа на функцию добавления новой темы. Добавьте в файл `views.py` следующий код:

`views.py`

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm, EntryForm
```

```
--пропуск--
def new_entry(request, topic_id):
    """Добавляет новую запись по конкретной теме."""
    ❶ topic = Topic.objects.get(id=topic_id)

    ❷ if request.method != 'POST':
        # Данные не отправлялись; создается пустая форма.
    ❸ form = EntryForm()
    else:
        # Отправлены данные POST; обработать данные.
    ❹ form = EntryForm(data=request.POST)
        if form.is_valid():
    ❺ new_entry = form.save(commit=False)
    ❻ new_entry.topic = topic
            new_entry.save()
    ❼ return redirect('learning_logs:topic', topic_id=topic_id)

    # Вывести пустую или недействительную форму.
    context = {'topic': topic, 'form': form}
    return render(request, 'learning_logs/new_entry.html', context)
```

Мы обновляем команду `import` и добавляем в нее только что созданный класс `EntryForm`. Определение `new_entry()` содержит параметр `topic_id`, позволяющий сохранять полученное из URL значение. Идентификатор темы понадобится для отображения страницы и обработки данных формы, поэтому мы используем `topic_id` для получения правильного объекта темы ❶.

Далее проверяется метод запроса: POST или GET ❷. Блок `if` выполняется для запроса GET, и мы создаем пустой экземпляр `EntryForm` ❸.

Для метода запроса POST мы обрабатываем данные, создавая экземпляр `EntryForm`, заполненный данными POST из объекта `request` ❹. Затем проверяется, корректны ли данные формы. Если да, то необходимо задать атрибут `topic` объекта записи перед сохранением его в базе данных. При вызове `save()` мы добавляем аргумент `commit=False` ❺ для того, чтобы дать Django указание создать новый объект записи и сохранить его в `new_entry`, не включая пока в базу данных. Мы присваиваем атрибуту `topic` объекта `new_entry` тему, прочитанную из базы данных в начале функции ❻, после чего вызываем `save()` без аргументов. В результате запись сохраняется в базе данных с правильной связанной темой.

Вызов `redirect()` в строке ❼ получает два аргумента: имя представления, которому передается управление, и аргумент для функции представления. В данном случае происходит перенаправление функции `topic()`, которой должен передаваться аргумент `topic_id`. Вызов перенаправляет пользователя на страницу темы, для которой была создана запись, и пользователь видит новую запись в соответствующем списке.

В конце функции создается словарь `context`, а страница создается на базе шаблона `new_entry.html`. Этот код выполняется для пустой или отправленной формы, которая была определена как недействительная.

Шаблон `new_entry`

Как видно из следующего кода, шаблон `new_entry` похож на шаблон `new_topic`:

`new_entry.html`

```
{% extends "learning_logs/base.html" %}

{% block content %}

❶ <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

    <p>Add a new entry:</p>
❷ <form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'>
    {% csrf_token %}
    {{ form.as_p }}
    <button name='submit'>add entry</button>
</form>

{% endblock content %}
```

В начале страницы выводится тема ❶, чтобы пользователь мог видеть, в какую тему добавляется новая запись. Вдобавок тема служит ссылкой для возврата к основной странице этой темы.

Аргумент `action` формы содержит значение `topic_id` из URL, чтобы функция представления могла связать новую запись с правильной темой ❷. В остальном этот шаблон почти не отличается от `new_topic.html`.

Создание ссылки на страницу `new_entry`

Затем необходимо создать ссылку на страницу `new_entry` на каждой странице темы:

`topic.html`

```
{% extends "learning_logs/base.html" %}

{% block content %}

    <p>Topic: {{ topic }}</p>

    <p>Entries:</p>
    <p>
        <a href="{% url 'learning_logs:new_entry' topic.id %}">add new entry</a>
    </p>

    <ul>
        --пропуск--
    </ul>

{% endblock content %}
```

Ссылка добавляется перед выводом записей, поскольку добавление новой записи является самым частым действием на этой странице. На рис. 19.2 изображена страница `new_entry`. Теперь пользователь может добавить сколько угодно новых тем и записей по каждой из них. Попробуйте страницу `new_entry`, добавив несколько записей для каждой из созданных вами тем.



Рис. 19.2. Страница `new_entry`

Редактирование записей

А теперь мы создадим страницу, на которой пользователи смогут редактировать ранее добавленные записи.

URL для `edit_entry`

В URL страницы должен передаваться идентификатор редактируемой записи. Для этого в файл `learning_logs/urls.py` вносятся следующие изменения:

`urls.py`

```
--пропуск--
urlpatterns = [
    --пропуск--
    # Страница для редактирования записи.
    path('edit_entry/<int:entry_id>', views.edit_entry, name='edit_entry'),
]
```

Эта схема URL соответствует URL типа http://localhost:8000/edit_entry/id/. В примере значение `id` присваивается параметру `entry_id`. Django отправляет запросы, соответствующие этому формату, функции представления `edit_entry()`.

Функция представления `edit_entry()`

Когда страница `edit_entry` получает запрос GET, `edit_entry()` возвращает форму для редактирования записи. При получении запроса POST с отредактированной записью страница сохраняет измененный текст в базе данных:

`views.py`

```
from django.shortcuts import render, redirect

from .models import Topic, Entry
from .forms import TopicForm, EntryForm
--nponyск--

def edit_entry(request, entry_id):
    """Редактирует существующую запись."""
    ❶ entry = Entry.objects.get(id=entry_id)
    topic = entry.topic

    if request.method != 'POST':
        # Исходный запрос; форма заполняется данными текущей записи.
        ❷ form = EntryForm(instance=entry)
    else:
        # Отправка данных POST; обработать данные.
        ❸ form = EntryForm(instance=entry, data=request.POST)
        if form.is_valid():
            ❹ form.save()
            ❺ return redirect('learning_logs:topic', topic_id=topic.id)

    context = {'entry': entry, 'topic': topic, 'form': form}
    return render(request, 'learning_logs/edit_entry.html', context)
```

Сначала необходимо импортировать модель `Entry`. Мы получаем объект записи, который пользователь хочет изменить ❶, и связанную с ней тему. В блоке `if`, который выполняется для запроса GET, создается экземпляр `EntryForm` с аргументом `instance=entry` ❷. Благодаря этому аргументу Django получает указание создать форму, заранее заполненную информацией из существующего объекта записи. Пользователь видит имеющиеся данные и может отредактировать их.

При обработке запроса POST передаются аргументы `instance=entry` и `data=request.POST` ❸. Они сообщают Django, что нужно создать экземпляр формы на основании информации существующего объекта записи, обновленный данными из `request.POST`. Затем проверяется, корректны ли данные формы. Если да, то следует вызов `save()` без аргументов ❹. Далее происходит перенаправление на страницу темы ❺, и пользователь видит обновленную версию отредактированной им записи.

Если отображается исходная форма для редактирования записи или отправленная форма недействительна, то создается словарь `context`, а страница создается на базе шаблона `edit_entry.html`.

Шаблон `edit_entry`

Шаблон `edit_entry.html` очень похож на `new_entry.html`:

`edit_entry.html`

```
{% extends "learning_logs/base.html" %}

{% block content %}

    <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

    <p>Edit entry:</p>

    ❶ <form action="{% url 'learning_logs:edit_entry' entry.id %}" method='post'>
        {% csrf_token %}
        {{ form.as_div }}
    ❷ <button name="submit">save changes</button>
    </form>

{% endblock content %}
```

Аргумент `action` отправляет форму функции `edit_entry()` для обработки ❶. Идентификатор записи добавляется как аргумент в тег `{% url %}`, чтобы функция представления могла изменить правильный объект записи. Кнопка отправки данных создается с текстом, который напоминает пользователю, что он сохраняет изменения, а не создает новую запись ❷.

Создание ссылки на страницу `edit_entry`

Теперь необходимо добавить ссылку на страницу `edit_entry` в каждую тему на странице со списком тем:

`topic.html`

```
--пропуск--
    {% for entry in entries %}
    <li>
        <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
        <p>{{ entry.text|linebreaks }}</p>
        <p>
            <a href="{% url 'learning_logs:edit_entry' entry.id %}">Edit entry</a>
        </p>
    </li>
--пропуск--
```

После даты и текста каждой записи добавляется ссылка редактирования. Мы используем шаблонный тег `{% url %}` для определения схемы URL из именованной схемы `edit_entry` и идентификатора текущей записи в цикле (`entry.id`). Текст ссылки "edit entry" выводится после каждой записи на странице. На рис. 19.3 показано, как выглядит страница со списком тем с этими ссылками.

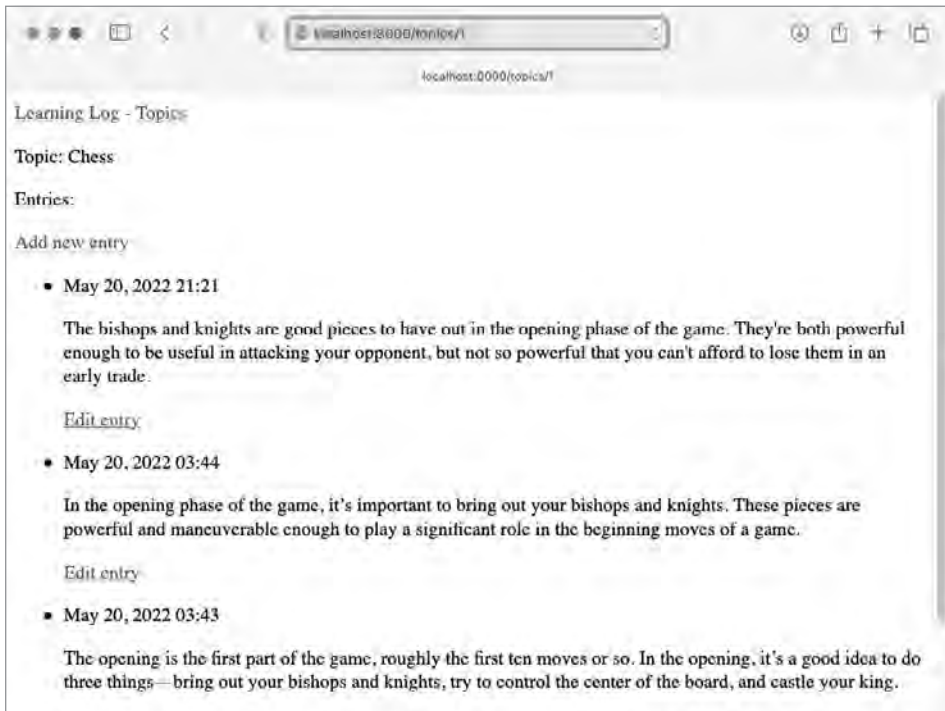


Рис. 19.3. Каждая запись снабжается ссылкой, позволяющей редактировать эту запись

Приложение «Журнал обучения» уже сейчас содержит большую часть необходимой функциональности. Пользователи могут добавлять темы и записи, а также читать любые записи по своему усмотрению. В следующем разделе мы реализуем систему регистрации пользователей, чтобы любой желающий мог создать свою учетную запись в «Журнале обучения» и добавить собственный набор тем и записей.

УПРАЖНЕНИЯ

19.1. Блог. Создайте новый проект Django blog. Создайте приложение blogs с двумя моделями: одна представляет весь блог, а вторая — отдельную публикацию в блоге. Создайте для каждой модели соответствующий набор полей. Создайте суперпользователя для проекта и с помощью административного сайта напишите пару коротких сообщений. Создайте главную страницу, на которой все сообщения будут выводиться в хронологическом порядке.

Создайте формы для создания блога и новых сообщений, а также для редактирования существующих сообщений. Заполните формы и убедитесь, что они работают.

Создание учетных записей пользователей

В этом разделе мы создадим систему регистрации и авторизации пользователей, чтобы люди могли создать учетную запись, начать и завершать сеанс работы с приложением. Для всей функциональности, относящейся к работе с пользователями, будет создано отдельное приложение. Мы также слегка изменим модель `Topic`, чтобы каждая тема была связана с конкретным пользователем.

Приложение *accounts*

Начнем с создания нового приложения `accounts` с помощью команды `startapp`:

```
(ll_env)learning_log$ python manage.py startapp accounts
(ll_env)learning_log$ ls
❶ accounts db.sqlite3 learning_logs ll_env ll_project manage.py
(ll_env)learning_log$ ls accounts
❷ __init__.py admin.py apps.py migrations models.py tests.py views.py
```

Система аутентификации по умолчанию создается вокруг концепции учетных записей пользователей, поэтому имя `accounts` упрощает интеграцию с системой по умолчанию. Команда `startapp`, показанная выше, создает каталог `accounts` ❶ со структурой, идентичной приложению `learning_logs` ❷.

Добавление учетных записей в *settings.py*

Новое приложение необходимо добавить в `INSTALLED_APPS` файла `settings.py`:

```
settings.py
--пропуск--
INSTALLED_APPS = [
    # Мои приложения
    'learning_logs',
    'accounts',

    # Приложения Django по умолчанию.
    --пропуск--
]
--пропуск--
```

Django добавляет приложение `accounts` в общий проект.

Добавление URL из *accounts*

Затем необходимо изменить корневой файл `urls.py`, чтобы он содержал URL, написанные для приложения `accounts`:

```
ll_project/urls.py
from django.contrib import admin
from django.urls import path, include
```

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('accounts/', include('accounts.urls')),  
    path('', include('learning_logs.urls')),  
]
```

Вставим строку для добавления файла `urls.py` из `accounts`. Эта строка будет соответствовать любому URL, начинающемуся со слова `accounts`, — например, <http://localhost:8000/accounts/login/>.

Страница входа

Начнем с реализации страницы входа. Мы воспользуемся стандартным представлением `login`, которое предоставляет Django, так что шаблон URL выглядит немного иначе. Создайте новый файл `urls.py` в каталоге `ll_project/accounts/` и добавьте в него следующий код:

accounts/urls.py

```
"""Определяет схемы URL для пользователей."""  
  
from django.urls import path, include  
  
app_name = 'accounts'  
urlpatterns = [  
    # Добавить URL авторизации по умолчанию.  
    path('', include('django.contrib.auth.urls')),  
]
```

Сначала импортируется функция `path`, а затем функция `include`, позволяющая добавить аутентификационные URL по умолчанию, определенные Django. Эти URL по умолчанию содержат именованные схемы, такие как `'login'` и `'logout'`. Переменной `app_name` присваивается значение `'accounts'`, чтобы инфраструктура Django могла отличить эти URL от URL, принадлежащих другим приложениям. Даже URL по умолчанию, предоставляемые Django, при добавлении в файл `urls.py` приложения `accounts` будут доступны через пространство имен `accounts`.

Схема страницы входа соответствует URL <http://localhost:8000/accounts/login/>. Когда Django читает этот URL, слово `accounts` указывает, что следует обратиться к `accounts/urls.py`, а `login` — что запросы должны отправляться представлению `login` по умолчанию.

Шаблон *login*

Когда пользователь запрашивает страницу входа, Django использует свое представление `login` по умолчанию, но мы все равно должны предоставить шаблон для этой страницы. Аутентификационные представления по умолчанию ищут шаблоны

в каталоге `registration`, поэтому вы должны создать его. В каталоге `ll_project/accounts/` создайте каталог `templates`, а внутри него — еще один каталог `registration`. Вот как выглядит шаблон `login.html`, который должен находиться в `ll_project/accounts/templates/registration/`:

login.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

❶  {% if form.errors %}
    <p>Your username and password didn't match. Please try again.</p>
    {% endif %}

❷  <form action="{% url 'accounts:login' %}" method='post'>
    {% csrf_token %}
❸    {{ form.as_p }}

❹    <button name="submit">log in</button>
    </form>

{% endblock content %}
```

Шаблон расширяет `base.html`, чтобы страница входа по оформлению и поведению была похожа на другие страницы сайта. Обратите внимание: шаблон в одном приложении может расширять шаблон из другого приложения.

Если у формы установлен атрибут `errors`, то выводится сообщение об ошибке ❶. В нем говорится, что комбинация имени пользователя и пароля не соответствует информации, хранящейся в базе данных.

Мы хотим, чтобы представление обработало форму, поэтому аргументу `action` присваивается URL страницы входа ❷. Представление отправляет объект `form` шаблону, мы должны вывести форму ❸ и добавить кнопку отправки данных ❹.

Настройка LOGIN_REDIRECT_URL

После успешной авторизации пользователя в системе Django необходимо перенаправить его на нужную страницу. Мы можем настроить это поведение в файле настроек.

Добавьте в конец файла `settings.py` следующий код:

settings.py

```
--пропуск--
# Мои настройки.
LOGIN_REDIRECT_URL = 'learning_logs:index'
```

Поскольку в файле `settings.py` указаны все настройки по умолчанию, полезно выделить раздел, в котором мы будем добавлять новые. Первой мы добавим настройку `LOGIN_REDIRECT_URL`, которая сообщает Django, по какому URL перенаправлять пользователя после успешной авторизации.

Создание ссылки на страницу входа

Добавим ссылку на страницу входа в `base.html`, чтобы она имелась на каждой странице. Ссылка не должна отображаться, если пользователь уже прошел процедуру входа, поэтому мы вкладываем ее в тег `{% if %}`:

base.html

```
<p>
  <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
  <a href="{% url 'learning_logs:topics' %}">Topics</a> -
❶  {% if user.is_authenticated %}
❷  Hello, {{ user.username }}.
    {% else %}
❸  <a href="{% url 'accounts:login' %}">Log in</a>
    {% endif %}
</p>

{% block content %}{% endblock content %}
```

В системе аутентификации Django в каждом шаблоне доступна переменная `user`, которая всегда содержит атрибут `is_authenticated`: он равен `True`, если пользователь прошел проверку, и `False` в противном случае. Это позволяет вам выводить разные сообщения для проверенных и непроверенных пользователей.

В данном случае мы выводим приветствие для пользователей, выполнивших вход ❶. Для проверенных пользователей устанавливается дополнительный атрибут `username`, с помощью которого можно настроить персональное приветствие и напомнить пользователю о том, что вход был выполнен ❷. Затем выводится ссылка на страницу входа для пользователей, которые еще не прошли проверку ❸.

Использование страницы входа

Учетная запись пользователя уже создана; попробуем ввести данные и посмотрим, работает ли страница. Откройте страницу <http://localhost:8000/admin/>. Если вы все еще работаете с правами администратора, то найдите ссылку `logout` в заголовке и щелкните на ней.

После выхода перейдите по адресу <http://localhost:8000/accounts/login/>. На экране должна появиться страница входа, похожая на рис. 19.4. Введите имя пользователя и пароль, заданные ранее, и вы снова должны оказаться на странице со списком. В заголовке страницы должно выводиться сообщение с указанием имени пользователя.



Рис. 19.4. Страница входа

Выход из учетной записи

Теперь необходимо предоставить пользователям возможность выйти из приложения. Запросы на выход из системы должны отправляться в виде POST-запросов, поэтому мы добавим небольшую форму выхода из системы в файл `base.html`. При щелчке на этой ссылке открывается страница, подтверждающая, что выход был выполнен успешно.

Добавление ссылки для выхода

Форму для выхода из системы мы добавим в файл `base.html`, чтобы она была доступна на каждой странице, и добавим в другой блок `if`, чтобы ее было видно только пользователям, уже выполнившим вход:

`base.html`

```
--пропуск--
{% block content %}{% endblock content %}

{% if user.is_authenticated %}
❶ <hr />
❷ <form action="{% url 'accounts:logout' %}" method='post'>
    {% csrf_token %}
    <button name='submit'>Log out</button>
</form>
{% endif %}
```

По умолчанию для выхода из системы используется схема URL `'accounts/logout/'`. Однако запрос должен быть отправлен по протоколу POST; в противном случае злоумышленники могут легко перехватить запрос на выход. Чтобы в запросах на выход из системы использовать протокол POST, мы определим простую форму.

Мы поместим форму в нижнюю часть страницы, под элементом горизонтальной прямой линии (`<hr />`) ❶. Это позволяет расположить кнопку выхода из системы гармонично под любым другим содержимым на странице. Сама форма имеет URL выхода из системы в аргументе `action` и значение `'post'` в качестве метода запроса ❷. Все формы в Django должны содержать код `{% csrf_token %}`, даже такая простая форма, как наша. Она пуста, за исключением кнопки отправки запроса.

Настройка `LOGOUT_REDIRECT_URL`

Когда пользователь нажимает кнопку выхода из системы, Django нужно знать, куда его отправить. Мы управляем этим поведением в файле `settings.py`:

`settings.py`

```
--пропуск--
# Мои настройки.
LOGIN_REDIRECT_URL = 'learning_logs:index'
LOGOUT_REDIRECT_URL = 'learning_logs:index'
```

Благодаря настройке `LOGOUT_REDIRECT_URL`, показанной выше, Django получает указание перенаправлять вышедших из системы пользователей обратно на главную страницу. Это простой способ подтвердить, что пользователь вышел из системы, поскольку после выхода он больше не должен видеть свое имя.

Страница регистрации

Теперь мы создадим страницу для регистрации новых пользователей. Для этой цели используем класс `UserCreationForm`, но напишем собственную функцию представления и шаблон.

URL страницы регистрации

Следующий код предоставляет шаблон URL для страницы регистрации, также размещенный в файле `accounts/urls.py`:

`accounts/urls.py`

```
"""Определяет схемы URL для пользователей."""

from django.urls import path, include

from . import views

app_name = 'accounts'
urlpatterns = [
    # Включить URL авторизации по умолчанию.
    path('', include('django.contrib.auth.urls')),
    # Страница регистрации.
    path('register/', views.register, name='register'),
]
```

Мы импортируем модуль `views` из `accounts`; он необходим, поскольку мы пишем собственное представление для страницы регрессии. Шаблон соответствует URL <http://localhost:8000/accounts/register/> и отправляет запросы функции `register()`, которую мы сейчас напомним.

Функция представления `register()`

Данная функция должна вывести пустую форму регистрации при первом запросе страницы регистрации, а затем обрабатывает заполненную форму при отправке данных. Если регистрация прошла успешно, то функция должна выполнить вход для нового пользователя. Добавьте в файл `accounts/views.py` следующий код:

accounts/views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm

def register(request):
    """Регистрирует нового пользователя."""
    if request.method != 'POST':
        # Вывод пустой формы регистрации.
        ❶ form = UserCreationForm()
    else:
        # Обработка заполненной формы.
        ❷ form = UserCreationForm(data=request.POST)

        ❸ if form.is_valid():
            ❹ new_user = form.save()
            # Выполнение входа и перенаправление на главную страницу.
            ❺ login(request, new_user)
            ❻ return redirect('learning_logs:index')

    # Вывод пустой или недействительной формы.
    context = {'form': form}
    return render(request, 'accounts/register.html', context)
```

Сначала импортируются функции `render()` и `redirect()`. Затем мы импортируем функцию `login()` для выполнения входа пользователя, если регистрационная информация верна. Вдобавок импортируется класс `UserCreationForm` по умолчанию. В функции `register()` мы проверяем, отвечает ли функция на запрос POST. Если нет, то создается экземпляр `UserCreationForm`, не содержащий исходных данных ❶.

В случае ответа на запрос POST создается экземпляр `UserCreationForm`, основанный на отправленных данных ❷. Мы проверяем, что они верны ❸; в данном случае — что имя пользователя содержит правильные символы, пароли совпадают, а пользователь не пытается вставить вредоносные конструкции в отправленные данные.

Если отправленные данные верны, то мы вызываем метод `save()` формы для сохранения имени пользователя и хеша пароля в базе данных ❹. Метод возвращает только что созданный объект пользователя, который сохраняется в `new_user`. После того как информация пользователя будет сохранена, мы выполняем вход; этот процесс состоит из двух шагов: сначала вызывается функция `login()` с объектами `request` и `new_user` ❺, которая создает действительный сеанс для нового пользователя. Наконец, пользователь перенаправляется на главную страницу ❻, где персонализированное приветствие в заголовке сообщает о том, что регистрация прошла успешно.

В конце функции создается страница, которая будет либо пустой формой, либо отправленной формой, содержащей недействительные данные.

Шаблон регистрации

Шаблон страницы регистрации похож на шаблон страницы входа. Проследите за тем, чтобы он был сохранен в одном каталоге с `login.html`:

register.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

    <form action="{% url 'accounts:register' %}" method='post'>
        {% csrf_token %}
        {{ form.as_div }}

        <button name="submit">register</button>
    </form>

{% endblock content %}
```

Шаблон должен выглядеть так же, как и другие шаблоны с формами, которые мы создали. Мы снова используем метод `as_p`, чтобы инфраструктура Django могла правильно отобразить все поля формы, в том числе все сообщения об ошибках, если форма была заполнена неправильно.

Создание ссылки на страницу регистрации

Следующий шаг — добавление кода, выводящего ссылку на страницу регистрации для любого пользователя, который еще не выполнил вход:

base.html

```
--пропуск--
{% if user.is_authenticated %}
    Hello, {{ user.username }}.
{% else %}
    <a href="{% url 'accounts:register' %}">Register</a> -
```

```
<a href="{% url 'accounts:login' %}">Log in</a>
{% endif %}
--пропуск--
```

Теперь пользователи, выполнившие вход, получают персональное приветствие и ссылку для выхода. Другие пользователи видят ссылку на страницу регистрации и ссылку для входа. Проверьте страницу регистрации, создав несколько учетных записей с разными именами пользователей.

В следующем разделе доступ к некоторым страницам будет ограничен, чтобы страницы были доступны только для зарегистрированных пользователей. Необходимо позаботиться и о том, чтобы каждая тема принадлежала конкретному пользователю.

ПРИМЕЧАНИЕ

Такая система регистрации позволяет любому пользователю создать сколько угодно учетных записей в «Журнале обучения». Однако некоторые системы требуют, чтобы пользователь подтвердил свою заявку, отправляя сообщение электронной почты, на которое он должен ответить. При таком подходе в системе будет создано меньше спам-мерских учетных записей, чем в простейшей системе из нашего примера. Но пока вы только учитесь создавать приложения, вполне нормально тренироваться на упрощенной системе регистрации вроде используемой нами.

УПРАЖНЕНИЯ

19.2. Учетные записи в блоге. Добавьте систему аутентификации и регистрации в проект Blog, работа над которым началась в упражнении 19.1. Убедитесь, что пользователь, выполнивший вход, видит свое имя где-то на экране, а незарегистрированные пользователи — ссылку на страницу регистрации.

Предоставление пользователям доступа к своим данным

Пользователь должен иметь возможность вводить личные данные. Мы создадим систему, которая будет определять, какому пользователю принадлежат те или иные данные, и ограничивать доступ к страницам, чтобы пользователь мог работать только со своими данными.

В этом разделе мы изменим модель Topic, чтобы каждая тема принадлежала конкретному пользователю. При этом автоматически решается проблема с записями, поскольку каждая запись принадлежит конкретной теме. Начнем с ограничения доступа к страницам.

Ограничение доступа с помощью `@login_required`

Django позволяет легко ограничить доступ к определенным страницам тем пользователям, которые выполнили вход. Для этого используется декоратор `@login_required`. Декоратор (decorator) представляет собой директиву, которая размещена непосредственно перед определением функции, применяется к ней перед ее выполнением и влияет на поведение кода. Рассмотрим пример.

Ограничение доступа к страницам тем

Каждая тема будет принадлежать пользователю, поэтому только зарегистрированные пользователи смогут запрашивать страницы тем. Добавьте в `learning_logs/views.py` следующий код:

`learning_logs/views.py`

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required

from .models import Topic, Entry
--пропуск--

@login_required
def topics(request):
    """Выводит все темы."""
    --пропуск--
```

Сначала импортируется функция `login_required()`. Мы применяем `login_required()` как декоратор для функции представления `topics()`; для этого перед именем `login_required()` ставится знак `@`. Он сообщает Python, что этот код должен выполняться перед кодом `topics()`.

Код `login_required()` проверяет, залогинился ли пользователь, и Django выполняет код `topics()` только при выполнении этого условия. В ином случае пользователь перенаправляется на страницу входа.

Чтобы перенаправление работало, необходимо внести изменения в файл `settings.py` и сообщить Django, где искать страницу входа. Добавьте в самый конец `settings.py` следующий фрагмент:

`settings.py`

```
--пропуск--
# Мои настройки.
LOGIN_REDIRECT_URL = 'learning_logs:index'
LOGOUT_REDIRECT_URL = 'learning_logs:index'
LOGIN_URL = 'accounts:login'
```

Когда пользователь, не прошедший проверку, запрашивает страницу, защищенную декоратором `@login_required`, Django отправляет пользователя на URL, определяемый `LOGIN_URL` в `settings.py`.

Чтобы протестировать эту возможность, завершите сеанс в любой из своих учетных записей и вернитесь на главную страницу. Щелкните на ссылке **Topics** (Темы), которая должна направить вас на страницу входа. Выполните вход с любой из своих учетных записей и на главной странице снова щелкните на ссылке **Topics** (Темы). На этот раз вы получите доступ к странице со списком тем.

Ограничение доступа к «Журналу обучения»

Django упрощает ограничение доступа к страницам, но вы должны решить, какие страницы следует защищать. Лучше сначала подумать, к каким страницам можно разрешить полный доступ, а затем ограничить его для всех остальных страниц. Снять излишние ограничения несложно, причем это куда менее рискованно, чем оставлять действительно важные страницы в открытом доступе.

В приложении «Журнал обучения» мы оставим неограниченный доступ к главной странице, странице регистрации и выхода. Доступ ко всем остальным страницам будет ограничен.

Вот как выглядит файл `learning_logs/views.py` с декораторами `@login_required`, примененными к каждому представлению, кроме `index()`:

learning_logs/views.py

```
--python--
@login_required
def topics(request):
    --python--

@login_required
def topic(request, topic_id):
    --python--

@login_required
def new_topic(request):
    --python--

@login_required
def new_entry(request, topic_id):
    --python--

@login_required
def edit_entry(request, entry_id):
    --python--
```

Попробуйте обратиться к любой из этих страниц, не выполняя входа: вы будете перенаправлены обратно на страницу входа. Кроме того, вы не сможете щелкать на ссылках на такие страницы, как `new_topic`. Но если ввести URL http://localhost:8000/new_topic/, то вы будете перенаправлены на страницу входа. Ограничьте доступ ко всем URL, связанным с личными данными пользователей.

Связывание данных с конкретными пользователями

Теперь данные, отправленные пользователем, необходимо связать с тем пользователем, который их отправил. Связь достаточно установить только с данными, находящимися на высшем уровне иерархии, а низкоуровневые данные последуют за ними автоматически. Например, в приложении «Журнал обучения» на высшем уровне находятся темы, а каждая запись связывается с некой темой. Если каждая тема принадлежит конкретному пользователю, то мы сможем отследить владельца каждой записи в базе данных.

Изменим модель `Topic` и добавим для пользователя отношение по внешнему ключу. После этого необходимо провести миграцию базы данных. Наконец, необходимо изменить некоторые представления, чтобы в них отображались только данные, связанные с текущим пользователем.

Изменение модели `Topic`

В файле `models.py` изменяются всего две строки:

`models.py`

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """Тема, которую изучает пользователь."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        """Возвращает строковое представление модели."""
        return self.text

class Entry(models.Model):
    --пропуск--
```

Сначала модель `User` импортируется из `django.contrib.auth`. Затем в `Topic` добавляется поле `owner`, используемое в отношении по внешнему ключу для модели `User`. Если пользователь удаляется, то все связанные с ним темы также будут удалены.

Идентификация существующих пользователей

При проведении миграции Django изменяет базу данных, чтобы в ней хранилась связь между каждой темой и пользователем. Для выполнения миграции Django необходимо знать, с каким пользователем должна быть связана каждая существующая тема. Проще всего связать все имеющиеся темы с одним пользователем, например суперпользователем. Но для этого сначала необходимо узнать его идентификатор.

Просмотрим идентификаторы всех пользователей, созданных до настоящего момента. Запустите сеанс оболочки Django и введите следующие команды:

```
(ll_env)learning_log$ python manage.py shell
❶ >>> from django.contrib.auth.models import User
❷ >>> User.objects.all()
<QuerySet [
```

В сеанс оболочки ❶ импортируется модель `User`. После этого просматриваются все пользователи, созданные до настоящего момента ❷. В выходных данных перечислены три пользователя: `ll_admin`, `eric` и `willie`.

Далее перебирается список пользователей, и для каждого выводятся его имя и идентификатор ❸. Когда Django спросит, с каким пользователем связать существующие темы, мы используем один из этих идентификаторов.

Миграция базы данных

Зная значение идентификатора, можно провести миграцию базы данных. Когда вы это делаете, Python предлагает временно связать модель `Topic` с конкретным владельцем или добавить в файл `models.py` значение по умолчанию, которое сообщи́т, как следует поступить. Выберите вариант 1:

```
❶ (ll_env)learning_log$ python manage.py makemigrations learning_logs
❷ It is impossible to add a non-nullable field 'owner' to topic without
specifying a default. This is because...
❸ Please select a fix:
  1) Provide a one-off default now (will be set on all existing rows with
    a null value for this column)
  2) Quit and manually define a default value in models.py.
```



```

❷ Select an option: 1
❸ Please enter the default value now, as valid Python
  The datetime and django.utils.timezone modules are available...
  Type 'exit' to exit this prompt
❹ >>> 1
Migrations for 'learning_logs':
  learning_logs/migrations/0003_topic_owner.py
- Add field owner to topic
(11_env)learning_log$

```

Сначала выдается команда `makemigrations` ❶. В ее выходных данных Django сообщает, что мы пытаемся добавить обязательное поле (значения которого отличаются от `null`) в существующую модель (`topic`) без указания значения по умолчанию ❷. Django предоставляет два варианта: мы можем либо указать значение по умолчанию прямо сейчас, либо завершить выполнение программы и добавить значение по умолчанию в `models.py` ❸. Мы выбираем первый вариант ❹. Тогда Django запрашивает значение по умолчанию ❺.

Чтобы связать все существующие темы с исходным административным пользователем `ll_admin`, я ввел идентификатор пользователя 1 ❻. Вы можете использовать идентификатор любого из созданных пользователей; он не обязан быть суперпользователем. Django проводит миграцию базы данных, используя это значение, и создает файл миграции `0003_topic_owner.py`, добавляющий поле `owner` в модель `Topic`.

Теперь можно провести миграцию. Введите следующую команду в активной виртуальной среде:

```

(11_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
❶ Applying learning_logs.0003_topic_owner... OK
(11_env)learning_log$

```

Django применяет новую миграцию с результатом `OK` ❶.

Чтобы убедиться в том, что миграция сработала ожидаемым образом, можно воспользоваться интерактивной оболочкой:

```

>>> from learning_logs.models import Topic
>>> for topic in Topic.objects.all():
...     print(topic, topic.owner)
...
Chess ll_admin
Rock Climbing ll_admin
>>>

```

После импортирования `Topic` из `learning_logs.models` мы перебираем все существующие темы, выводим каждую из них и имя пользователя, которому она принадлежит. Как видите, сейчас каждая тема принадлежит пользователю `ll_admin`. (Если при выполнении кода произойдет ошибка, то попробуйте выйти из оболочки и запустить ее заново.)

ПРИМЕЧАНИЕ

Вместо выполнения миграции можно просто сбросить содержимое базы данных, но это приведет к потере всех существующих данных. Полезно научиться выполнять миграцию базы данных, не нарушая целостности данных пользователей. Если вы хотите начать с новой базы данных, то используйте команду `python manage.py flush` для повторного создания структуры базы данных. Вам придется создать нового суперпользователя, а все данные будут потеряны.

Ограничение доступа к темам

В настоящее время пользователь, выполнивший вход, будет видеть все темы независимо от того, под какой учетной записью он вошел. Сейчас мы изменим приложение, чтобы каждый пользователь видел только принадлежащие ему темы.

Внесите в функцию `topics()` в файле `views.py` следующее изменение:

learning_logs/views.py

```
--пропуск--
@login_required
def topics(request):
    """Выводит список тем."""
    topics = Topic.objects.filter(owner=request.user).order_by('date_added')
    context = {'topics': topics}
    return render(request, 'learning_logs/topics.html', context)
--пропуск--
```

Если пользователь выполнил вход, в объекте запроса устанавливается атрибут `request.user` с информацией о пользователе. Благодаря фрагменту кода `Topic.objects.filter(owner=request.user)` Django получает указание извлечь из базы данных только те объекты `Topic`, у которых атрибут `owner` соответствует текущему пользователю. Способ отображения остается прежним, поэтому изменять шаблон для страницы тем вообще не нужно.

Чтобы увидеть, как работает этот способ, выполните вход в качестве пользователя, с которым связаны все существующие темы, и перейдите к странице со списком тем. На ней должны отображаться все темы. Теперь завершите сеанс и войдите снова через другую учетную запись. На этот раз вы увидите сообщение `No topics have been added yet` (Темы пока не добавлены).

Защита тем пользователя

Никаких реальных ограничений на доступ к страницам еще не существует, поэтому любой зарегистрированный пользователь может опробовать разные URL (например, <http://localhost:8000/topics/1/>) и просмотреть страницы тем, которые ему удастся подобрать.

Попробуйте сделать это. После входа через учетную запись суперпользователя скопируйте URL или запишите идентификатор в URL темы, после чего завершите сеанс и войдите снова от имени другого пользователя. Введите URL этой темы. Вам удастся прочитать все записи, хотя сейчас вы вошли под именем другого пользователя.

Чтобы решить эту проблему, мы будем выполнять проверку перед получением запрошенных данных в функции представления `topic()`:

learning_logs/views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
❶ from django.http import Http404

--пропуск--
@login_required
def topic(request, topic_id):
    """Выводит одну тему и все ее записи."""
    topic = Topic.objects.get(id=topic_id)
    # Проверка того, что тема принадлежит текущему пользователю.
    ❷ if topic.owner != request.user:
        raise Http404

    entries = topic.entry_set.order_by('-date_added')
    context = {'topic': topic, 'entries': entries}
    return render(request, 'learning_logs/topic.html', context)
--пропуск--
```

Код 404 — стандартное сообщение об ошибке, которое возвращается в тех случаях, когда запрошенный ресурс не существует на сервере. В данном случае мы импортируем исключение `Http404` ❶, которое будет выдаваться программой при запросе пользователем темы, которую ему видеть не положено. Получив запрос темы, прежде чем отображать страницу, мы убеждаемся в том, что пользователь этой темы является текущим пользователем приложения. Если тема не принадлежит ему, то выдается исключение `Http404` ❷, а Django возвращает страницу с ошибкой 404.

Пока при попытке просмотреть записи другого пользователя вы получите от Django сообщение `Page Not Found` (Страница не найдена). В главе 20 мы настроим проект так, чтобы пользователи видели полноценную страницу ошибки вместо страницы отладки.

Защита страницы `edit_entry`

Страницы `edit_entry` используют URL в форме http://localhost:8000/edit_entry/entry_id/, где `entry_id` — число. Защитим эту страницу, чтобы никто не мог подобрать URL для получения доступа к чужим записям:

learning_logs/views.py

```
--пропуск--
@login_required
def edit_entry(request, entry_id):
    """Редактирует существующую запись."""
    entry = Entry.objects.get(id=entry_id)
    topic = entry.topic
    if topic.owner != request.user:
        raise Http404

    if request.method != 'POST':
        --пропуск--
```

Программа читает запись и связанную с ней тему. Затем мы проверяем, совпадает ли владелец темы с текущим пользователем; при несовпадении выдается исключение `Http404`.

Связывание новых тем с текущим пользователем

В настоящее время страница добавления новых тем несовершенна, поскольку не связывает новые темы с конкретным пользователем. При попытке добавить новую тему выдается сообщение об ошибке `IntegrityError` с уточнением `NOT NULL constraint failed: learning_logs_topic.owner_id` (Ограничение `NOT NULL` не выполнено: `Learning_logs_topic.owner_id`). Django сообщает, что при создании новой темы обязательно должно быть задано значение поля `owner`.

Проблема легко решается, поскольку мы можем получить доступ к информации текущего пользователя через объект `request`. Добавьте следующий код, связывающий новую тему с текущим пользователем:

learning_logs/views.py

```
--пропуск--
@login_required
def new_topic(request):
    --пропуск--
    else:
        # Отправлены данные POST; обработать данные.
        form = TopicForm(data=request.POST)
        if form.is_valid():
            ❶ new_topic = form.save(commit=False)
            ❷ new_topic.owner = request.user
            ❸ new_topic.save()
            return redirect('learning_logs:topics')
```

```
# Вывод пустой или недействительной формы.  
context = {'form': form}  
return render(request, 'learning_logs/new_topic.html', context)  
--pronyск--
```

При первом вызове `form.save()` передается аргумент `commit=False`, поскольку новая тема должна быть изменена перед сохранением в базе данных ❶. Атрибуту `owner` новой темы присваивается текущий пользователь ❷. Наконец, мы вызываем `save()` для только что определенного экземпляра темы ❸. Теперь она содержит все обязательные данные, и ее сохранение пройдет успешно.

Вы сможете добавить сколько угодно новых тем для любого количества разных пользователей. Каждому из них будут доступны только его собственные данные, какие бы операции он ни пытался выполнять: просмотр данных, ввод новых или изменение существующих данных.

УПРАЖНЕНИЯ

19.3. Рефакторинг. В файле `views.py` есть два места, в которых программа проверяет, что пользователь, связанный с темой, является текущим пользователем, вошедшим в систему. Поместите код этой проверки в функцию `check_topic_owner()` и вызовите ее при необходимости.

19.4. Защита страницы `new_entry`. Пользователь может попытаться добавить новую запись в журнал другого пользователя, вводя URL с идентификатором темы, принадлежащей другому пользователю. Чтобы предотвратить подобные атаки, перед сохранением новой записи проверьте, что текущий пользователь является владельцем темы, к которой относится запись.

19.5. Защищенный блог. В проекте `Vlog` примите меры к тому, чтобы каждое сообщение было связано с конкретным пользователем. Убедитесь в том, что чтение всех сообщений доступно всем пользователям, но только зарегистрированные пользователи могут создавать новые сообщения и редактировать существующие. В представлении, в котором пользователи редактируют сообщения, перед обработкой формы убедитесь в том, что редактируемое сообщение принадлежит именно этому пользователю.

Резюме

В этой главе вы научились использовать формы для создания новых тем и записей, а также редактирования существующих данных. Далее вы перешли к реализации системы учетных записей. Вы предоставили существующим пользователям возможность начинать и завершать сеанс работы с приложением, а также научились использовать класс `Django UserCreationForm` для создания новых учетных записей.

После создания простой системы аутентификации и регистрации пользователей вы ограничили доступ пользователей к некоторым страницам; для этого использовался декоратор `@login_required`. Затем связали данные с конкретными пользователями с помощью отношения по внешнему ключу. Вы также узнали, как выполнить миграцию базы данных, когда миграция требует ввести данные по умолчанию.

В последней части главы вы узнали, как ограничить состав данных, просматриваемых пользователем, с помощью функций представления. Для чтения соответствующих данных использовался метод `filter()`, а владелец запрашиваемых данных сравнивался с текущим пользователем.

Не всегда бывает сразу понятно, какие данные должны быть доступны всем пользователям, а какие данные следует защищать, но этот навык приходит с практикой. Решения, принятые нами в этой главе для защиты данных пользователей, наглядно показывают, почему при создании проекта желательно работать в команде: если кто-то просматривает код вашего проекта, это повышает вероятность выявления плохо защищенных областей.

К настоящему моменту вы создали полностью работоспособный проект, работающий на локальной машине. В последней главе вы усовершенствуете внешний вид приложения «Журнал обучения», чтобы оно выглядело более привлекательно. Кроме того, развернете проект на сервере, чтобы любой пользователь с доступом к Интернету мог зарегистрироваться и создать учетную запись.

20

Оформление и развертывание приложения



Приложение «Журнал обучения» уже полностью функционально, но не оформлено стилистически и работает только на локальной машине. В этой главе мы подберем для проекта простое, но профессиональное оформление, а затем развернем его на сервере, чтобы любой желающий мог создать учетную запись.

Для форматирования будет использоваться библиотека Bootstrap — набор инструментов для оформления веб-приложений, с которыми они будут выглядеть профессионально на любых современных устройствах, от большого монитора с плоским экраном до смартфона. Для этого мы применим приложение `django-bootstrap5`, а вы заодно потренируетесь в использовании приложений, созданных другими разработчиками Django.

Для развертывания «Журнала обучения» будет использоваться `Platform.sh` — сайт, позволяющий загрузить ваш проект на один из его серверов, чтобы сделать его доступным для любого пользователя с подключением к Сети. Кроме того, мы начнем пользоваться системой управления версиями Git для отслеживания изменений в проекте.

Завершив работу с «Журналом обучения», вы будете уметь разрабатывать простые веб-приложения, придавать им качественный внешний вид и развертывать их на работающих серверах. Вдобавок по мере накопления опыта вы научитесь пользоваться обучающими ресурсами с материалами более высокого уровня.

Оформление «Журнала обучения»

До сих пор мы намеренно игнорировали оформление приложения, чтобы сосредоточиться на его функциональности. И это вполне разумный подход к разработке, поскольку приложение приносит пользу, только если работает. Конечно, когда

приложение начинает работать, оформление выходит на первый план, чтобы пользователи захотели работать с ним.

В этом разделе мы установим приложение `django-bootstrap5` и добавим его в проект. Затем используем его для настройки оформления отдельных страниц проекта, чтобы все они имели единый внешний вид и стиль.

Приложение `django-bootstrap5`

Для интеграции Bootstrap в наш проект будет использоваться приложение `django-bootstrap5`. Оно скачивает необходимые файлы Bootstrap, размещает их в правильных каталогах проекта и предоставляет доступ к стилевым директивам в шаблонах проекта.

Чтобы установить `django-bootstrap5`, введите в активной виртуальной среде следующую команду:

```
(ll_env)learning_log$ pip install django-bootstrap5
--пропуск--
Successfully installed beautifulsoup4-4.11.1 django-bootstrap5-21.3
soupsieve-2.3.2.post1
```

Затем необходимо ввести следующий код для добавления `django-bootstrap5` в список `INSTALLED_APPS` в файле `settings.py`:

`settings.py`

```
--пропуск--
INSTALLED_APPS = (
    # Мои приложения.
    'learning_logs',
    'accounts',

    # Сторонние приложения.
    'django_bootstrap5',

    # Приложения Django по умолчанию.
    'django.contrib.admin',
--пропуск--
```

Создайте новую секцию для приложений, созданных другими разработчиками, и добавьте в нее запись `'django_bootstrap5'`. Проследите за тем, чтобы секция располагалась после секции `# Мои приложения`, но перед секцией, содержащей приложения Django по умолчанию.

Использование Bootstrap для оформления «Журнала обучения»

По сути, Bootstrap представляет собой большой набор инструментов форматирования. Кроме того, библиотека содержит ряд шаблонов, с помощью которых можно сформировать общий стиль проекта. Пользоваться этими шаблонами намного

проще, чем использовать отдельные инструменты оформления. Чтобы просмотреть шаблоны, предоставляемые Bootstrap, перейдите по ссылке <http://getbootstrap.com> и найдите раздел **Examples** (Примеры). Мы воспользуемся шаблоном **Navbar static**, который предоставляет простую панель навигации и контейнер для содержимого страницы.

На рис. 20.1 показано, как будет выглядеть главная страница после применения шаблона Bootstrap к `base.html` и незначительного изменения `index.html`.

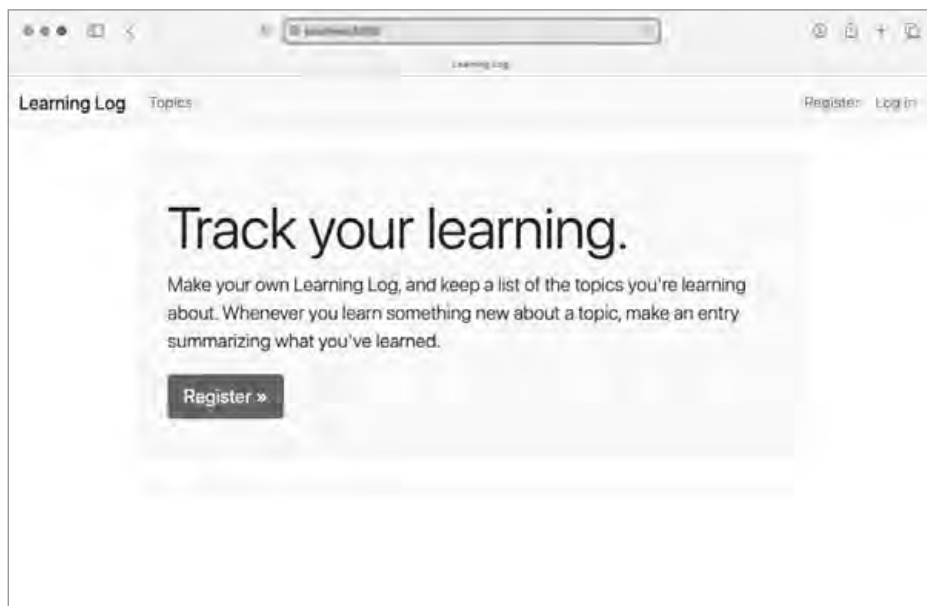


Рис. 20.1. Вид главной страницы «Журнала обучения» после использования Bootstrap

Изменение файла `base.html`

Шаблон `base.html` необходимо изменить так, чтобы в нем был задействован шаблон Bootstrap. Новая версия `base.html` будет представлена в несколько этапов. Код в этом файле объемный; возможно, вы захотите скопировать его с сайта https://ehmatthes.github.io/pcc_3e. Если вы скопируете файл, то вам все равно следует прочитать данный подраздел, чтобы понять, какие изменения были внесены.

Определение заголовков HTML

Первое изменение в `base.html` таково: заголовки HTML определяются в файле, чтобы при открытии страницы «Журнала обучения» в строке заголовка браузера выводилось имя сайта. Кроме того, будут добавлены некоторые требования для

использования Bootstrap в шаблонах. Удалите все содержимое `base.html` и замените его следующим кодом:

base.html

```
❶ <!doctype html>
❷ <html lang="en">
❸ <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
❹ <title>Learning Log</title>

❺ {% load django_bootstrap5 %}
    {% bootstrap_css %}
    {% bootstrap_javascript %}

</head>
```

Сначала файл объявляется как документ HTML ❶, написанный на английском языке ❷. Файл HTML состоит из двух основных частей: *заголовка* (head) и *тела* (body). Заголовок файла HTML начинается с открывающего тега `<head>` ❸ и не содержит контента: он всего лишь передает браузеру информацию, необходимую для правильного отображения страницы. Далее добавляется элемент `<title>` для страницы; его содержимое будет выводиться в строке заголовка браузера при каждом открытии «Журнала обучения» ❹.

Перед закрытием раздела `head` мы загружаем коллекцию шаблонных тегов, доступных в `django-bootstrap5` ❺. Так, шаблонный тег `{% bootstrap_css %}` загружает все CSS-файлы, необходимые для реализации стилей Bootstrap. Следующий тег активизирует все интерактивное поведение, которое может использоваться на странице, — например, раздвижные панели навигации. Закрывающий тег `</head>` указывается в последней строке.

Все настройки стилей Bootstrap теперь доступны в любом шаблоне, который наследуется от `base.html`. Если вы хотите использовать пользовательские теги шаблонов `django-bootstrap5`, то каждый шаблон должен содержать тег `{% load django_bootstrap5 %}`.

Определение панели навигации

Код, определяющий панель в верхней части страницы, получается довольно длинным, поскольку должен хорошо работать как на узких экранах смартфонов, так и на широких экранах мониторов настольных компьютеров. Мы рассмотрим код панели навигации по частям.

Первая часть панели выглядит так:

base.html

```
--пропуск--
</head>
<body>
```

```

❶ <nav class="navbar navbar-expand-md navbar-light bg-light mb-4 border">
  <div class="container-fluid">
❷    <a class="navbar-brand" href="{% url 'learning_logs:index' %}">
      Learning Log</a>
❸
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
      data-bs-target="#navbarCollapse" aria-controls="navbarCollapse"
      aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
❹
    <div class="collapse navbar-collapse" id="navbarCollapse">
❺      <ul class="navbar-nav me-auto mb-2 mb-md-0">
❻        <li class="nav-item">
❼          <a class="nav-link" href="{% url 'learning_logs:topics' %}">
              Topics</a></li>
          </ul> <!-- Конечный ссылок в левой части панели навигации -->
        </div> <!-- Закрывает сворачивающиеся части панели навигации -->

      </div> <!-- Закрывает контейнер панели навигации -->
    </nav> <!-- Конец панели навигации -->

❽ {% block content %}{% endblock content %}

</body>
</html>

```

Первый элемент — открывающий тег `<body>`. Тело файла HTML содержит контент, который будет виден пользователям на странице. Далее расположен тег `<nav>`, который обозначает раздел навигационных ссылок в верхней части страницы ❶. Весь контент внутри этого элемента оформляется по правилам Bootstrap, устанавливаемым селекторами `navbar`, `navbar-expand-md` и другими, которые вы видите в этой части кода. *Селектор* (selector) определяет, к каким элементам страницы должно применяться правило стилей. Селекторы `navbar-light` и `bg-light` оформляют панель навигации, используя тему со светлым фоном. Сокращение `mb` в `mb-4` происходит от `margin-bottom`, то есть «нижнее поле»; этот селектор гарантирует, что между панелью и остальным контентом страницы остается свободное место. Селектор `border` создает тонкую рамку вокруг светлого фона, чтобы немного отделить его от остального содержимого страницы.

Тег `<div>` в следующей строке открывает изменяемый по размеру контейнер, который будет содержать общую панель навигации. *Div* — сокращение от слова *division* (деление); вы создаете веб-страницу, деля ее на разделы и определяя правила стиля и поведения, которые применяются к этим разделам. Все правила стиля и поведения, заданные в открывающем теге `<div>`, влияют на все содержимое до соответствующего закрывающего тега, `</div>`.

Далее задается название проекта, `Learning Log`, которое будет отображаться в качестве первого элемента на панели навигации ❷ и служить ссылкой на главную страницу; она будет выводиться на каждой странице проекта, как это было

в неотформатированной версии проекта, созданной в предыдущих двух главах. Селектор `navbar-brand` меняет вид этой ссылки так, чтобы она выделялась на фоне остальных; это оформление становится одной из составляющих фирменной символики сайта.

Шаблон Bootstrap определяет кнопку, которая будет видна, если ширины окна браузера не хватает для горизонтального отображения всей панели навигации ❸. Когда пользователь нажимает кнопку, навигационные элементы выводятся в раскрывающемся списке. Атрибут `collapse` сворачивает панель при уменьшении размеров окна браузера и отображении сайта на мобильных устройствах с малыми экранами.

Далее открывается новая секция панели навигации (`<div>`) ❹. Это начало той части панели, которая будет сворачиваться на узких экранах и окнах.

В Bootstrap элементы навигации определяются как элементы неупорядоченного списка ❺, при этом правила стилей делают его совсем непохожим на список. Все ссылки и пункты панели добавляются как элементы неупорядоченного списка ❻. В нашем примере единственный элемент в списке — ссылка на страницу тем ❼. Обратите внимание на закрывающий тег `</li>` в конце ссылки; каждому открывающему тегу необходим соответствующий закрывающий.

Остальные строки, показанные выше, содержат соответствующие закрывающие теги. В HTML комментарий пишется следующим образом:

```
<!-- Это HTML-комментарий. -->
```

Закрывающие теги обычно не имеют комментариев, но если вы еще не набрались опыта работы с HTML, то рекомендуется сопровождать комментариями некоторые закрывающие теги. Всего один пропущенный или лишний тег может нарушить верстку всей страницы. Мы добавили блок `content` ❽, а также закрывающие теги `</body>` и `</html>`.

Мы еще не закончили с панелью навигации, но у нас уже получился полноценный HTML-документ. Если в данный момент сеанс `runserver` активен, то остановите его и перезапустите сервер. Перейдите на главную страницу проекта, чтобы посмотреть панель навигации с некоторыми элементами, показанными на рис. 20.1 (см. выше). Далее добавим остальные элементы на панель.

Добавление ссылок для управления учетными записями пользователей

Нам все еще нужно добавить ссылки, связанные с учетными записями пользователей. Давайте начнем с добавления всех таких ссылок, кроме формы выхода из системы.

Внесите в файл `base.html` следующие изменения:

base.html

```
--пропуск--
</ul> <!-- Конец ссылок в левой части панели навигации -->

<!-- Ссылки, связанные с аккаунтом -->
❶ <ul class="navbar-nav ms-auto mb-2 mb-md-0">
❷     {% if user.is_authenticated %}
        <li class="nav-item">
❸             <span class="navbar-text me-2">Hello, {{ user.username }}.
                </span></li>
❹     {% else %}
        <li class="nav-item">
            <a class="nav-link" href="{% url 'accounts:register' %}">
                Register</a></li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'accounts:login' %}">
                    Log in</a></li>
        {% endif %}

</ul> <!-- Конец ссылок, связанных с аккаунтом -->

</div> <!-- Закрывает сворачивающиеся части панели навигации -->
--пропуск--
```

Новый набор ссылок начинается с помощью еще одного открывающего тега `<ul>` ❶. На странице можно создать сколько угодно групп ссылок. Имя селектора `ml-auto` означает `margin-left-automatic`, то есть «автоматическое левое поле»; этот селектор анализирует другие элементы на панели навигации и определяет величину левого поля, которое сдвигает эту группу ссылок к правому краю экрана.

Блок `if` уже использовался ранее для вывода сообщений для пользователей в зависимости от того, выполнили они вход или нет ❷. На этот раз блок стал немного длиннее, поскольку некоторые правила стилей находятся внутри условных тегов. Приветствие для авторизованных пользователей расположено в элементе `<span>` ❸. Элемент `span` оформляет фрагменты текста или элементы страницы, которые являются частью более длинной строки. Если элементы `div` создают собственный раздел страницы, то элементы `span` следуют непрерывно внутри большего раздела. На первый взгляд такая структура кажется запутанной, поскольку многие страницы содержат элементы `div` с большой вложенностью. Здесь элемент `span` используется для оформления текста на панели навигации — например, имени пользователя, выполнившего вход.

В блоке `else`, который запускается для неавторизованных пользователей, мы добавили ссылки для регистрации новой учетной записи и входа в систему ❹. Они должны выглядеть так же, как и ссылка на страницу тем.

Если вы захотите добавить на панель навигации дополнительные ссылки, то добавьте еще один элемент `<li>` в одну из групп `<ul>`, которые мы определили, используя правила стилей, подобные приведенным выше.

Теперь добавим на панель навигации форму выхода из системы.

Добавление формы выхода из системы на панель навигации

Изначально мы добавили форму выхода из системы в нижнюю часть файла `base.html`. Теперь поместим ее в более подходящее место, на панель навигации:

`base.html`

```
--пропуск--
</ul> <!-- Конец ссылок, связанных с аккаунтом -->

{% if user.is_authenticated %}
  <form action="{% url 'accounts:logout' %}" method='post'>
    {% csrf_token %}
    <button name='submit' class='btn btn-outline-secondary btn-sm'>
      Log out</button>
    </form>
  {% endif %}

</div> <!-- Закрывает сворачивающиеся части панели навигации -->
--пропуск--
```

Форма выхода из системы должна располагаться после набора ссылок для управления учетной записью, но внутри сворачивающейся части панели навигации. Единственное изменение в форме — добавление нескольких классов стилей Bootstrap в элемент `<button>`, которые применяют правила стилей Bootstrap к кнопке выхода из системы ❶.

После перезагрузки главной страницы вы сможете входить и выходить из системы, используя одну из своих учетных записей.

Работа с файлом `base.html` еще не закончена. Необходимо определить два блока, которые могут использоваться отдельными страницами для размещения относящегося к ним контента.

Определение основного раздела страницы

Оставшаяся часть `base.html` содержит основной контент страницы:

`base.html`

```
--пропуск--
</nav> <!-- Конец панели навигации -->

❶ <main class="container">
❷   <div class="pb-2 mb-2 border-bottom">
```

```
        {% block page_header %}{% endblock page_header %}
    </div>
    3 </div>
      {% block content %}{% endblock content %}
    </div>
  </main>

</body>
</html>
```

Сначала открывается тег `<main>` ❸. Элемент `main` используется для основного контента в теле страницы. В данном случае назначается селектор `container`, что является простым способом группировки элементов на странице. В этот контейнер будут добавлены два элемента `div`.

Первый элемент `div` содержит блок `page_header` ❷. Мы будем использовать этот блок для отображения заголовка большинства страниц. Чтобы секция выделялась на фоне других частей страницы, мы разместим отступы под заголовком. *Отступ* (`padding`) представляет собой пространство между контентом элемента и его границей. Селектор `pb-2` — директива Bootstrap, создающая отступы умеренной величины в нижней части оформляемого элемента. *Поле* (`margin`) называется пространство между границей элемента и другими элементами страницы. Селектор `mb-2` создает небольшое поле в нижней части этого `div`.

Нам нужна граница в нижней части этого блока, поэтому мы используем селектор `border-bottom`, который создает тонкую границу в нижней части блока `page_header`.

Далее определяется еще один элемент `div`, содержащий блок `content` ❸. К этому блоку не применяется никакой конкретный стиль, поэтому контент любой страницы можно оформить так, как вы считаете нужным для этой страницы. Файл `base.html` завершается закрывающими тегами для элементов `main`, `body` и `html`.

Загрузив главную страницу «Журнала обучения» в браузере, вы увидите профессиональную панель навигации, изображенную на рис. 20.1 (см. выше). Попробуйте изменить размеры окна, заметно уменьшив его ширину; панель навигации должна превратиться в кнопку. Щелкните на кнопке, и все ссылки появятся в раскрывающемся списке.

Оформление главной страницы с помощью табло

Изменим главную страницу с помощью нового блока `header` и другого элемента Bootstrap, так называемого *джамботрона* (`jumbotron`) — большого блока, который выделяется на общем фоне страницы. Обычно этот элемент используется на главных страницах для размещения краткого описания проекта и призыва к действию, в котором зрителя приглашают принять участие.

Обновленный файл `index.html` выглядит так:

index.html

```
{% extends "learning_logs/base.html" %}

❶ {% block page_header %}
❷   <div class="p-3 mb-4 bg-light border rounded-3">
     <div class="container-fluid py-4">
❸       <h1 class="display-3">Track your learning.</h1>

❹       <p class="lead">Make your own Learning Log, and keep a list of the
        topics you're learning about. Whenever you learn something new
        about a topic, make an entry summarizing what you've learned.</p>

❺       <a class="btn btn-primary btn-lg mt-1"
           href="{% url 'accounts:register' %}">Register &raquo;</a>
     </div>
  </div>
{% endblock page_header %}
```

Сначала мы сообщаем Django о том, что следует определение содержимого блока `_header` страницы ❶. Элемент `jumbotron` представляет собой пару элементов `div`, к которому применяется набор стилевых директив ❷. Внешний `div` настраивает свойства `padding` и `margin`, применяет светлый цвет фона и закругленные углы. Внутренний контейнер `div` изменяется в зависимости от размера окна и также настраивает отступы. Селектор `py-4` применяет отступ сверху и снизу к элементу `div`. Вы можете изменять значения этих свойств и смотреть, как меняется главная страница.

Внутри элемента `jumbotron` содержатся три элемента. Первый — короткое сообщение `Track your learning` (Отслеживайте свое обучение), которое дает новым посетителям представление о том, что делает приложение «Журнал обучения» ❸. Элемент `h1` — это заголовок первого уровня, а благодаря селектору `display-3` заголовок становится более тонким и высоким. Далее добавляется более длинное сообщение с дополнительной информацией о том, что пользователь может сделать со своим дневником ❹. Это сообщение оформлено как абзац `lead`, который должен выделяться на фоне обычных абзацев текста.

Вместо простой текстовой ссылки мы создаем кнопку, которая предлагает пользователю зарегистрировать свою учетную запись в «Журнале обучения» ❺. Это та же ссылка, что и в заголовке, но кнопка выделяется на фоне страницы и показывает пользователю, что необходимо сделать для того, чтобы приступить к работе над проектом. В результате применения селекторов получается крупная кнопка, обеспечивающая желаемое действие. Код `»` является *сущностью HTML* (HTML entity), представляющей две правые угловые скобки (`>>`). Наконец, закрывающие теги `div` завершают блок `page_header`. Поскольку в этом файле всего два элемента

`div`, то нет необходимости обозначать метками закрывающие теги `div` в листинге. Контент на эту страницу добавляться не будет, поэтому определять на ней блок `content` не нужно.

Теперь страница выглядит так, как показано на рис. 20.1 (см. выше). Она смотрится намного лучше по сравнению с проектом, в котором оформление не применялось.

Оформление страницы входа

Мы усовершенствовали внешний вид страницы входа, но формы входа изменения пока не коснулись. Приведем внешний вид формы в соответствие с остальными элементами страницы, внося в файл `login.html` следующие изменения:

login.html

```
{% extends "learning_logs/base.html" %}
❶ {% load django_bootstrap5 %}

❷ {% block page_header %}
    <h2>Log in to your account.</h2>
{% endblock page_header %}

{% block content %}

    <form action="{% url 'accounts:login' %}" method='post'>
        {% csrf_token %}
        ❸ {% bootstrap_form form %}
        ❹ {% bootstrap_button button_type="submit" content="Log in" %}
    </form>

{% endblock content %}
```

Сначала в шаблон загружаются шаблонные теги `bootstrap5` ❶. Затем определяется блок `page_header`, который описывает, для чего нужна страница ❷. Обратите внимание: блок `{% if form.errors %}` удален из шаблона; `django-bootstrap5` управляет ошибками формы автоматически.

Для отображения формы используется шаблонный тег `{% bootstrap_form %}` ❸; он заменяет тег `{{ form.as_div }}`, который мы использовали в главе 19, и вставляет правила в стиле Bootstrap в отдельные элементы формы по мере ее отображения. Чтобы создать кнопку отправки, мы используем тег `{% bootstrap_button %}` с соответствующими аргументами и присваиваем ей метку `Log in` ❹.

На рис. 20.2 показана форма входа так, как она выглядит сейчас. Страница стала гораздо более чистой, ее оформление последовательно, а предназначение предельно ясно. Попробуйте выполнить вход, указав неверное имя пользователя или пароль; вы увидите, что даже сообщения об ошибках имеют единый стиль оформления и хорошо интегрируются с сайтом в целом.

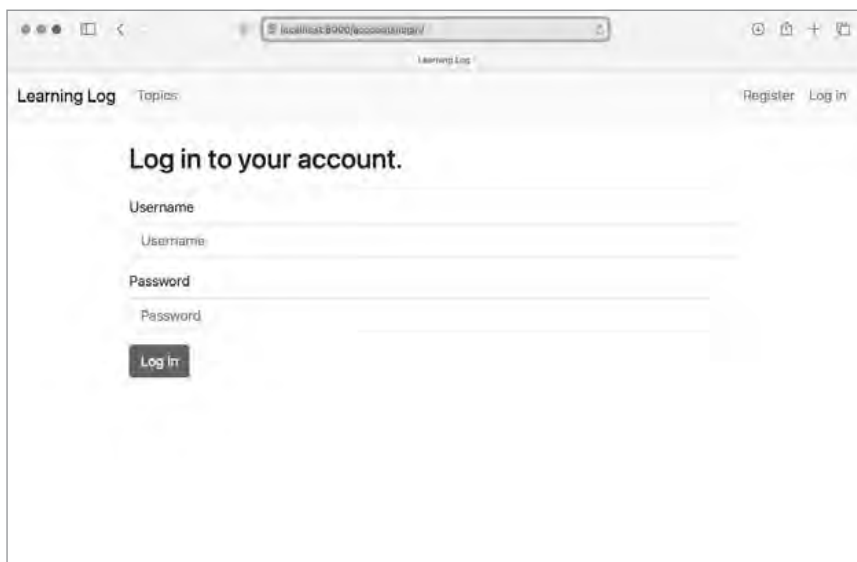


Рис. 20.2. Страница входа, оформленная с использованием Bootstrap

Оформление страницы со списком тем

А теперь позаботимся о том, чтобы страницы для просмотра информации также были оформлены в том же стиле. Начнем со страницы, на которой расположен список тем:

topics.html

```
{% extends 'learning_logs/base.html' %}
```

```
{% block page_header %}
```

```
① <h1>Topics</h1>
```

```
{% endblock page_header %}
```

```
{% block content %}
```

```
② <ul class="list-group border-bottom pb-2 mb-4">
```

```
    {% for topic in topics %}
```

```
③ <li class="list-group-item border-0">
```

```
    <a href="{% url 'learning_logs:topic' topic.id %}">
```

```
        {{ topic.text }}</a>
```

```
    </li>
```

```
    {% empty %}
```

```
④ <li class="list-group-item border-0">No topics have been added yet.</li>
```

```
    {% endfor %}
```

```
</ul>
```

```
<a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a>
```

```
{% endblock content %}
```

Тег `{% load bootstrap5 %}` не нужен, поскольку в этом файле не используются никакие шаблонные теги `bootstrap5`. Заголовок `Topics` перемещается в блок `page_header`, и вместо простого абзачного тега ему назначается оформление заголовка `<h1>` ❶.

Основное содержимое этой страницы — список тем, поэтому мы используем компонент *группы списков* (`list group`) Bootstrap для визуализации страницы. Он применяет набор простых правил стилей к списку целиком и к каждому его элементу. Открывая тег `<ul>`, мы сначала добавляем класс `list-group`, чтобы применить к списку правила стилей по умолчанию ❷. Мы дополнительно форматируем список, добавляя границу внизу списка, небольшой отступ под списком (`pb-2`) и поле под нижней границей (`mb-4`).

К каждому элементу списка добавляется класс `list-group-item`, и мы меняем стили по умолчанию, удаляя границы вокруг отдельных элементов ❸. Сообщение, которое отображается, когда список пуст, требует тех же классов ❹.

Теперь страница тем отформатирована так же, как и главная.

Оформление записей на странице темы

На странице темы мы используем панели Bootstrap, чтобы выделить каждую запись. *Панель* (`card`) — это набор вложенных контейнеров `div` с гибкими, предопределенными стилями, которые идеально подходят для отображения записей темы:

topic.html

```
{% extends 'learning_logs/base.html' %}
```

```
❶ {% block page_header %}
    <h1>{{ topic.text }}</h1>
{% endblock page_header %}

{% block content %}
    <p>
        <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
    </p>

    {% for entry in entries %}
❷     <div class="card mb-3">
❸         <!-- Заголовок панели с временной меткой и ссылкой для редактирования -->
❹         <h4 class="card-header">
            {{ entry.date_added|date:'M d, Y H:i' }}
❺         <small><a href="{% url 'learning_logs:edit_entry' entry.id %}">
                edit entry</a></small>
        </h4>
        <!-- Тело панели с текстом ввода -->
❻         <div class="card-body">{{ entry.text|linebreaks }}</div>
    </div>
    {% empty %}
❽     <p>There are no entries for this topic yet.</p>
    {% endfor %}

{% endblock content %}
```

Сначала тема размещается в блоке `page_header` ❶. Затем удаляется структура неупорядоченного списка, использовавшаяся ранее в этом шаблоне. Вместо того чтобы превращать каждую запись в элемент списка, мы создаем элемент с селектором `card` ❷. Он имеет два вложенных элемента: первый предназначен для хранения временной метки и ссылки для редактирования, а второй — для хранения тела записи. Селектор `card` учитывает большинство стилей, необходимых для форматирования этого контейнера `div`; мы настраиваем панель, добавляя небольшое поле к нижней части каждой панели (`mb-3`).

Первый элемент в панели представляет собой заголовок — элемент `<h4>` с селектором `card-header` ❸. Заголовок содержит дату создания записи и ссылку для ее редактирования. Ссылка `edit_entry` заключается в тег `<small>`, чтобы она была чуть меньше временной метки ❹. Второй элемент представляет собой `div` с селектором `card-body` ❺, который размещает текст записи в простом поле на панели. Обратите внимание: код Django для добавления информации на страницу остался прежним; изменились только элементы, влияющие на внешний вид страницы. Поскольку у нас больше нет неупорядоченного списка, мы заменили теги элементов списка вокруг сообщения с пустым списком простыми тегами абзаца ❻.

На рис. 20.3 изображена страница темы с новым оформлением. Функциональность приложения «Журнал обучения» не изменилась, но приложение выглядит более профессионально и привлекательно для пользователя.

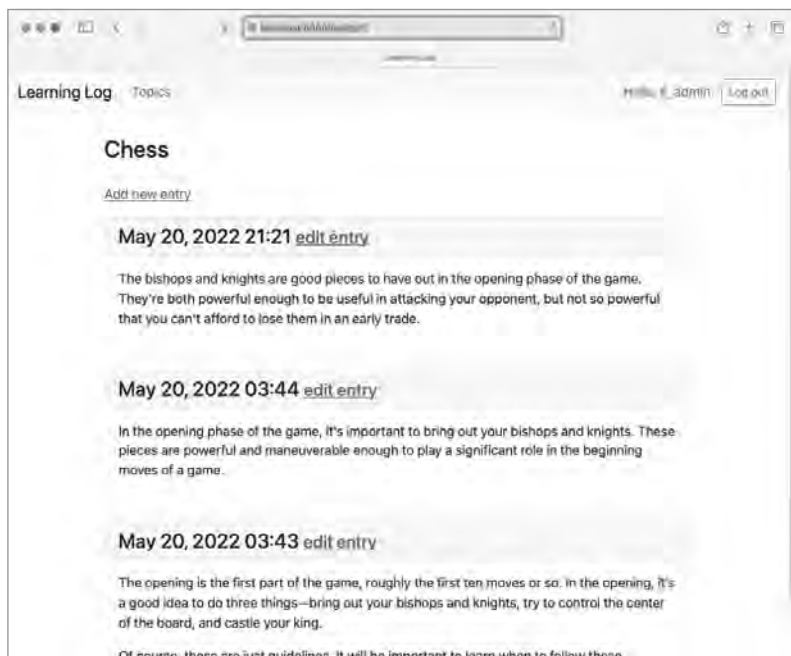


Рис. 20.3. Страница темы, оформленная с помощью стилей Bootstrap

Если вы хотите использовать другой шаблон Bootstrap, то действуйте в той же последовательности, которая уже описывалась в этой главе. Скопируйте шаблон в `base.html` и измените элементы, содержащие контент, чтобы шаблон отображал информацию вашего проекта. Затем воспользуйтесь средствами индивидуального форматирования Bootstrap для оформления содержимого каждой страницы.

ПРИМЕЧАНИЕ

Инструмент Bootstrap имеет отличную документацию. Посетите страницу <https://getbootstrap.com> и выберите раздел Docs (Документы), чтобы узнать больше о возможностях Bootstrap.

УПРАЖНЕНИЯ

20.1. Другие формы. Мы применили стили Bootstrap к странице `login`. Внесите аналогичные изменения в другие страницы на базе форм: `new_topic`, `new_entry`, `edit_entry` и `register`.

20.2. Форматирование проекта Blog. Используйте Bootstrap для форматирования проекта Blog из главы 19.

Развертывание «Журнала обучения»

После того как проекту был придан профессиональный вид, мы развернем его на реальном сервере, чтобы любой пользователь с подключением к Интернету мог работать с приложением. Мы воспользуемся Platform.sh — веб-платформой, позволяющей управлять развертыванием веб-приложений.

Создание учетной записи Platform.sh

Чтобы создать учетную запись, откройте сайт <https://platform.sh> и нажмите кнопку Free Trial (Бесплатная пробная версия). Учетные записи создаются бесплатно, и Platform.sh предоставляет бесплатный уровень, который на момент написания книги не требует оплаты. Пробный период позволяет развернуть приложение с минимальными затратами ресурсов, давая возможность протестировать проект в реальном развертывании, прежде чем переходить на платный план хостинга.

ПРИМЕЧАНИЕ

На бесплатном уровне Platform.sh существуют свои ограничения, поскольку хостинг-провайдеры борются со спамом и злоупотреблением ресурсами. Текущие лимиты бесплатной пробной версии можно посмотреть на сайте <https://platform.sh/free-trial>.

Установка инструментария *Platform.sh CLI*

Чтобы развернуть проект на серверах Platform.sh и управлять им, вам понадобятся инструменты *интерфейса командной строки* (Command Line Interface, CLI). Чтобы установить новейшую версию CLI, откройте сайт <https://docs.platform.sh/development/cli.html> и следуйте инструкциям для своей операционной системы.

В большинстве систем вы можете установить CLI, выполнив следующую команду в терминале:

```
$ curl -fsS https://platform.sh/cli/installer | php
```

После выполнения этой команды вам нужно перезагрузить терминал, прежде чем вы сможете использовать CLI.

ПРИМЕЧАНИЕ

Эта команда, вероятно, не будет работать в стандартном терминале Windows. Вы можете использовать инструментарий Windows Subsystem for Linux (WSL) или терминал Git Bash. Если вам нужно установить PHP, то можете использовать программу установки XAMPP с сайта <https://arachefriends.org>. Если у вас возникли трудности с установкой Platform.sh CLI, то обратитесь к более подробным инструкциям по установке, изложенным в приложении Д.

Установка пакета *platformshconfig*

Вам также нужно установить дополнительный пакет, *platformshconfig*. Он помогает определить, запущен ли проект в вашей локальной системе или на сервере Platform.sh. В активной виртуальной среде выполните следующую команду:

```
(ll_env)learning_log$ pip install platformshconfig
```

Мы будем использовать этот пакет для изменения настроек проекта, когда он будет запущен на реальном сервере.

Создание файла *requirements.txt*

Удаленному серверу необходимо знать, от каких пакетов зависит наш проект, поэтому мы воспользуемся модулем *pip* для создания файла со списком. Оставаясь в активной виртуальной среде, введите следующую команду:

```
(ll_env)learning_log$ pip freeze > requirements.txt
```

Команда *freeze* дает модулю *pip* указание записать имена всех пакетов, в настоящее время установленных в системе, в файл *requirements.txt*. Откройте файл *requirements.txt* и просмотрите пакеты и номера версий, установленных в вашей системе:

requirements.txt

```
asgiref==3.5.2
beautifulsoup4==4.11.1
Django==4.1
django-bootstrap5==21.3
platformshconfig==2.4.0
soupsieve==2.3.2.post1
sqlparse==0.4.2
```

Приложение «Журнал обучения» уже зависит от семи разных пакетов с конкретными номерами версий, поэтому для его правильной работы требуется конкретная конфигурация среды на удаленном сервере. (Мы установили три из этих пакетов вручную, а четыре были загружены автоматически как зависимости этих пакетов.)

При развертывании «Журнал обучения» Platform.sh устанавливает все пакеты, перечисленные в `requirements.txt`, и создает среду с теми же пакетами, которые мы используем локально. Поэтому разработчик может быть уверен в том, что развернутый проект будет работать точно так же, как в его локальной системе. Вы поймете, насколько это полезно, когда начнете создавать и вести в своей системе несколько разных проектов.

ПРИМЕЧАНИЕ

Если пакет добавлен в список в вашей системе, но номер версии отличается от показанного, то используйте версию, которая есть в вашей системе.

Дополнительные требования к развертыванию

Для рабочего сервера требуются два дополнительных пакета. Они используются для обслуживания проекта в рабочей среде, где многие пользователи могут выполнять запросы одновременно.

В каталоге с файлом `requirements.txt` создайте новый файл `requirements_remote.txt`. Добавьте в него следующие два пакета:

requirements_remote.txt

```
# Требования для рабочего проекта.
gunicorn
psycogp2
```

Пакет `gunicorn` реагирует на запросы по мере их поступления на удаленный сервер; он играет роль рабочего сервера, который мы использовали локально. Пакет `psycogp2` необходим для управления со стороны Django базой данных Postgres, которую использует Platform.sh. *Postgres* — это система баз данных с открытым исходным кодом, которая очень хорошо подходит для рабочих приложений.

Добавление файлов конфигурации

Каждый хостинг-провайдер требует определенную конфигурацию для корректной работы проекта на его серверах. В этом подразделе мы добавим три файла конфигурации:

- `.platform.app.yaml` — основной конфигурационный файл для проекта. Информировывает Platform.sh, какой проект мы пытаемся развернуть и какие ресурсы ему требуются, а также содержит команды для сборки проекта на сервере;
- `.platform/routes.yaml` — определяет маршруты к нашему проекту. Когда Platform.sh получает запрос, именно эта конфигурация помогает направить запрос в наш конкретный проект;
- `.platform/services.yaml` — определяет все дополнительные сервисы, необходимые нашему проекту.

Все эти файлы имеют формат YAML (YAML Ain't Markup Language (YAML не является языком разметки)). *YAML* — это язык, предназначенный для написания конфигурационных файлов; он спроектирован так, чтобы его могли легко читать как люди, так и машины. Вы можете написать или изменить файл YAML вручную, а компьютер сможет его прочесть и однозначно интерпретировать.

Файлы YAML отлично подходят для настройки процесса развертывания, поскольку позволяют управлять этим процессом.

Просмотр скрытых файлов

Большинство операционных систем скрывают файлы и папки, начинающиеся с точки, например `.platform`. Когда вы открываете менеджер файлов, то по умолчанию не видите такие файлы и папки. Но как программисту вам необходимо их видеть. Просмотреть скрытые файлы и папки в разных операционных системах можно, выполнив следующие действия.

- В операционной системе Windows запустите **Проводник**, а затем откройте любую папку, например **Рабочий стол**. Перейдите на вкладку **View** (Вид) и убедитесь, что установлены флажки **File name extensions** (Расширения имен файлов) и **Hidden items** (Скрытые элементы).
- В macOS вы можете нажать сочетание клавиш **⌘+Shift+.** (точка) в любом окне программы Finder.
- В системах Linux, таких как Ubuntu, можно нажать **Ctrl+H** в любом файловом браузере. Чтобы сделать эту настройку постоянной, откройте менеджер файлов, например Nautilus, и перейдите на вкладку с настройками (обозначена тремя линиями). Установите флажок **Show Hidden Files** (Показывать скрытые файлы).

Файл конфигурации `.platform.app.yaml`

Первый файл конфигурации — самый длинный, поскольку управляет всем процессом развертывания. Ниже он представлен частями; вы можете либо ввести его код вручную в текстовом редакторе, либо загрузить копию с сайта по адресу https://ehmatthes.github.io/pcc_3e.

Вот первая часть файла `.platform.app.yaml`, который должен быть сохранен в том же каталоге, что и файл `manage.py`:

`.platform.app.yaml`

```
❶ name: "ll_project"
   type: "python:3.10"

❷ relationships:
    database: "db:postgresql"

# Конфигурация приложения при его публикации в Интернете.
❸ web:
    upstream:
        socket_family: unix
    commands:
❹     start: "gunicorn -w 4 -b unix:$SOCKET ll_project.wsgi:application"
❺ locations:
    "/":
        passthru: true
    "/static":
        root: "static"
        expires: 1h
        allow: true

# Размер постоянного диска приложения (в мегабайтах).
❻ disk: 512
```

Сохраняя этот файл, убедитесь, что в начале его имени указана точка. Если вы опустите точку, то Platform.sh не обнаружит файл и ваш проект не будет развернут.

Сейчас вам не нужно понимать все содержимое файла `.platform.app.yaml`; я обозначу лишь наиболее важные части конфигурации. Файл начинается с указания имени проекта, `'ll_project'`, соответствующего имени, которое мы использовали при создании проекта ❶. Нужно обозначить и версию Python, которую мы используем (на момент написания книги — 3.10). Список поддерживаемых версий опубликован на сайте <https://docs.platform.sh/languages/python.html>.

Далее следует раздел `relationships`, в котором определяются сервисы, необходимые для работы проекта ❷. Здесь указана единственная связь — с базой данных Postgres. Ниже расположен раздел `web` ❸. Раздел `commands:start` сообщает Platform.sh, какой процесс следует использовать для обслуживания входящих запросов. Мы указали,

что запросы будет обрабатывать инструмент `gunicorn` ④. Эта команда заменяет команду `python manage.py runserver`, которую мы использовали локально.

Раздел `locations` сообщает Platform.sh, куда отправлять входящие запросы ⑤. Большинство запросов должно передаваться через `gunicorn`; файлы `urls.py` будут указывать `gunicorn`, как именно обрабатывать эти запросы. Запросы на статические файлы будут обрабатываться отдельно и обновляться раз в час. Последняя строка показывает, что мы запрашиваем 512 Мбайт дискового пространства на одном из серверов Platform.sh ⑥.

Вторая часть файла `.platform.app.yaml` выглядит следующим образом:

```
--пропуск--  
disk: 512
```

```
# Установка локального монтирования для чтения/записи журналов.
```

① `mounts:`

```
  "logs":  
    source: local  
    source_path: logs
```

```
# Хуки, выполняемые в различные моменты жизненного цикла приложения.
```

② `hooks:`

```
  build: |  
    ③ pip install --upgrade pip  
      pip install -r requirements.txt  
      pip install -r requirements_remote.txt
```

```
    mkdir logs  
    ④ python manage.py collectstatic  
    rm -rf logs
```

⑤ `deploy: |`
 `python manage.py migrate`

В разделе `mounts` ① определяются каталоги, в которых будут производиться чтение и запись данных во время работы проекта. В этом разделе задается каталог `logs/` для развернутого проекта.

В разделе `hooks` ② определяются действия, которые выполняются в различные моменты процесса развертывания. В разделе `build` мы указываем команды установки всех пакетов, необходимых для работы проекта в реальной среде ③. Мы также приводим команду `collectstatic` ④, которая собирает все статические файлы, необходимые для проекта, в одной локации, чтобы их можно было эффективно обслуживать.

Наконец, в разделе `deploy` ⑤ мы указываем, что миграции должны выполняться каждый раз при развертывании проекта. В простом проекте это не будет иметь никакого эффекта, если в нем не было изменений.

Два других конфигурационных файла намного короче; рассмотрим их далее.

Файл конфигурации *routes.yaml*

Этот файл управляет маршрутами. *Маршрут* (route) — это путь, по которому проходит запрос при его обработке сервером. Получая запрос, Platform.sh должен знать, куда его отправить.

Создайте папку `.platform` в том же каталоге, где расположен файл `manage.py`. Убедитесь, что добавили точку в начало имени. В новой папке создайте файл `routes.yaml` и введите в него следующий код:

.platform/routes.yaml

Каждый маршрут описывает, как входящий URL будет обработан Platform.sh.

```
"https://{default}/*":
  type: upstream
  upstream: "ll_project:http"

"https://www.{default}/*":
  type: redirect
  to: "https://{default}/*"
```

Этот файл гарантирует, что такие запросы, как https://project_url.com и www.project_url.com, будут направляться в одно и то же место.

Файл конфигурации *services.yaml*

В этом конфигурационном файле перечисляются сервисы, которые необходимы проекту для работы. Сохраните этот файл в каталоге `.platform/` вместе с файлом `routes.yaml`:

.platform/routes.yaml

Каждый из перечисленных сервисов будет развернут в собственном контейнере
проекта Platform.sh.

```
db:
  type: postgresql:12
  disk: 1024
```

Этот файл содержит лишь один сервис — базу данных Postgres.

Изменение файла *settings.py* для *Platform.sh*

В конец файла `settings.py` необходимо добавить раздел для определения настроек, предназначенных конкретно для среды Platform.sh:

settings.py

--пропуск--

Настройки Platform.sh.

❶ from platformshconfig import Config

```
config = Config()
❷ if config.is_valid_platform():
❸     ALLOWED_HOSTS.append('.platformsh.site')

❹     if config.appDir:
        STATIC_ROOT = Path(config.appDir) / 'static'
❺     if config.projectEntropy:
        SECRET_KEY = config.projectEntropy

if not config.in_build():
❻     db_settings = config.credentials('database')
        DATABASES = {
            'default': {
                'ENGINE': 'django.db.backends.postgresql',
                'NAME': db_settings['path'],
                'USER': db_settings['username'],
                'PASSWORD': db_settings['password'],
                'HOST': db_settings['host'],
                'PORT': db_settings['port'],
            },
        }
```

Обычно мы помещаем операторы импорта в начало модуля, но в данном случае полезно хранить все настройки удаленного доступа в одном разделе. В этом примере мы импортируем `Config` из `platformshconfig` ❶, который помогает определить настройки на удаленном сервере. Мы изменяем настройки только в том случае, если метод `config.is_valid_platform()` возвращает `True` ❷, указывая, что настройки будут использоваться на сервере `Platform.sh`.

Мы изменяем значение `ALLOWED_HOSTS`, чтобы допустить обслуживание проекта хостами, заканчивающимися на `.platformsh.site` ❸. Все проекты, развернутые на бесплатном уровне, будут обслуживаться на этом хосте. Если настройки загружаются в каталог развернутого приложения ❹, то мы прописываем переменную `STATIC_ROOT`, чтобы статические файлы обслуживались корректно. Мы также настраиваем на удаленном сервере более безопасный ключ `SECRET_KEY` ❺.

Наконец, мы настраиваем рабочую базу данных ❻. Она используется только в том случае, если процесс сборки завершен и проект обслуживается. Весь показанный код необходим для успешного взаимодействия Django с сервером Postgres, настроенным `Platform.sh` для проекта.

Использование *Git* для управления файлами проекта

Как упоминалось в главе 17, `Git` — система управления версиями, которая позволяет создать «мгновенный снимок» состояния кода проекта при реализации каждой новой функции. Это дает возможность легко вернуться к последнему работоспособному состоянию проекта при возникновении каких-либо проблем (например,

если в ходе работы над новой функцией была случайно внесена ошибка). Каждый снимок называется *фиксацией* или *коммитом* (commit).

Если в вашем проекте используется Git, это означает, что вы можете дорабатывать проект, не беспокоясь, что он выйдет из строя. Развертывая проект на сервере, вы должны убедиться в том, что ваша версия проекта работоспособна. Подробная информация о Git и управлении версиями приведена в приложении Г.

Установка Git

Вполне возможно, пакет Git уже установлен в вашей системе. Чтобы проверить это, откройте новое терминальное окно и введите команду `git --version`:

```
(ll_env)learning_log$ git --version
git version 2.30.1 (Apple Git-130)
```

Если вы увидели сообщение о том, что Git не установлен, то обратитесь к инструкциям по установке Git, изложенным в приложении Г.

Настройка Git

Git следит за тем, кто внес изменения в проект, даже в том случае, если над проектом работает только один человек. Для этого Git необходимо знать имя пользователя и адрес электронной почты. Имя следует указывать обязательно, но ничего не мешает использовать вымышленный адрес электронной почты для учебных проектов:

```
(ll_env)learning_log$ git config --global user.name "eric"
(ll_env)learning_log$ git config --global user.email "eric@example.com"
```

Если пропустить этот шаг, то Git запросит у вас эту информацию при первой фиксации.

Игнорирование файлов

Нам не нужно, чтобы система Git отслеживала все файлы в проекте, поэтому мы настроим ее на пропуск некоторых файлов. Создайте файл `.gitignore` в папке с файлом `manage.py`. Обратите внимание: имя файла начинается с точки, а файл не имеет расширения. Содержимое файла `.gitignore` выглядит так:

```
.gitignore
ll_env/
__pycache__/
*.sqlite3
```

Мы даем Git указание игнорировать весь каталог `ll_env`, поскольку можем автоматически воссоздать его в любой момент. Кроме того, в системе управления версиями не отслеживается каталог `__pycache__` с файлами `.pyc`, которые создаются автоматически, когда Django выполняет файлы `.py`. Мы не отслеживаем изменения в локальной базе данных, поскольку так поступать вообще нежелательно: если на сервере будет использоваться SQLite, то вы можете случайно переписать «живую» базу данных локальной тестовой базой данных при отправке проекта на сервер. Благодаря звездочке в выражении `*.sqlite3` Git получает указание игнорировать любые файлы с расширением `.sqlite3`.

ПРИМЕЧАНИЕ

Если вы используете macOS, то добавьте `.DS_Store` в файл `.gitignore`. В нем хранится информация о настройках папок в macOS, и он не имеет никакого отношения к этому проекту.

Фиксация состояния проекта

Чтобы инициализировать репозиторий Git для «Журнала обучения», добавьте все необходимые файлы в репозиторий и зафиксируйте исходное состояние проекта. Вот как это делается:

- ❶ `(ll_env)learning_log$ git init`
Initialized empty Git repository in /Users/eric/.../learning_log/.git/
- ❷ `(ll_env)learning_log$ git add .`
- ❸ `(ll_env)learning_log$ git commit -am "Ready for deployment to Platform.sh."`
[main (root-commit) c7ffaad] Ready for deployment to Platform.sh.
42 files changed, 879 insertions(+)
create mode 100644 .gitignore
create mode 100644 .platform.app.yaml
--пропуск--
create mode 100644 requirements_remote.txt
- ❹ `(ll_env)learning_log$ git status`
On branch main
nothing to commit, working tree clean
`(ll_env)learning_log$`

Сначала вводится команда `git init`, которая инициализирует пустой репозиторий в каталоге, содержащем «Журнал обучения» ❶. Затем команда `git add .` добавляет все файлы (кроме игнорируемых) в репозиторий ❷. (Не забудьте точку.) Далее вводится команда `git commit -am "сообщение"`: благодаря флагу `-a` Git получает указание добавить все измененные файлы в фиксированное состояние, а флаг `-m` дает Git указание сохранить сообщение в журнале ❸.

Вывод команды `git status` ❹ сообщает, что текущей является главная ветвь, а рабочий каталог пуст. Этот статус должен выводиться каждый раз, когда вы отправляете свой проект на Platform.sh.

Создание проекта на Platform.sh

На данный момент «Журнал обучения» по-прежнему работает в нашей локальной системе, а также настроен для корректной работы на удаленном сервере. Мы воспользуемся инструментом Platform.sh CLI, чтобы создать новый проект на сервере, а затем загрузить наш проект на удаленный сервер.

В терминале перейдите в каталог `learning_log/` и выполните следующую команду:

```
(ll_env)learning_log$ platform login
Opened URL: http://127.0.0.1:5000
Please use the browser to log in.
--пропуск--
```

- ❶ Do you want to create an SSH configuration file automatically? [Y/n] Y

Эта команда откроет вкладку браузера, на которой вы сможете войти в систему. Войдя, вы можете закрыть вкладку браузера и вернуться в терминал. Если будет предложено создать файл конфигурации SSH ❶, то введите Y, чтобы подключиться к удаленному серверу позже.

Теперь создадим проект. Выходных данных будет много, поэтому рассмотрим процесс по частям. Начните с команды `create`:

```
(ll_env)learning_log$ platform create
* Project title (--title)
Default: Untitled Project
```

- ❶ > ll_project

```
* Region (--region)
The region where the project will be hosted
--пропуск--
[us-3.platform.sh] Moses Lake, United States (AZURE) [514 gC02eq/kWh]
```

- ❷ > us-3.platform.sh

```
* Plan (--plan)
Default: development
Enter a number to choose:
[0] development
--пропуск--
```

- ❸ > 0

```
* Environments (--environments)
The number of environments
Default: 3
```

- ❹ > 3

```
* Storage (--storage)
The amount of storage per environment, in GiB
Default: 5
```

- ❺ > 5

Первое приглашение запрашивает имя для проекта ❶, и мы указываем имя `ll_project`. Далее появляется запрос, в каком регионе мы хотели бы разместить сервер ❷. Выберите ближайший к вам сервер; для меня это `us-3.platform.sh`. Для ответа на остальные запросы можно принять значения по умолчанию: сервер на самом низком плане разработки ❸, три окружения для проекта ❹ и 5 Гбайт памяти под весь проект ❺.

Осталось ответить еще на три запроса:

```
Default branch (--default-branch)
The default Git branch name for the project (the production environment)
Default: main
```

❶ `> main`

```
Git repository detected: /Users/eric/.../learning_log
```

❷ Set the new project `ll_project` as the remote for this repository? [Y/n] Y

```
The estimated monthly cost of this project is: $10 USD
```

❸ Are you sure you want to continue? [Y/n] Y

The Platform.sh Bot is activating your project



The project is now ready!

Репозиторий Git может иметь несколько ветвей; Platform.sh спрашивает, должна ли по умолчанию использоваться ветвь `main` ❶. Затем появляется запрос, хотим ли мы подключить репозиторий локального проекта к удаленному репозиторию ❷. Наконец, нам сообщают, что этот проект будет стоить около 10 долларов в месяц, если мы будем поддерживать его после окончания бесплатного пробного периода ❸. Если вы еще не вводили реквизиты банковской карты, то вам не стоит беспокоиться об оплате. Platform.sh просто заморозит ваш проект, если вы превысите лимиты бесплатной пробной версии.

Загрузка в Platform.sh

Наконец-то все готово для отправки проекта на удаленный сервер. В активном терминальном сеансе введите следующие команды:

```
(ll_env)learning_log$ platform push
```

❶ Are you sure you want to push to the main (production) branch? [Y/n] Y

```
--пропуск--
```

```
The authenticity of host 'git.us-3.platform.sh (...)' can't be established.
RSA key fingerprint is SHA256:Tvn...7PM
```



```
② Are you sure you want to continue connecting (yes/no/[fingerprint])? Y
Pushing HEAD to the existing environment main
--пропуск--
To git.us-3.platform.sh:3pp3mqcexhlvy.git
* [new branch]      HEAD -> main
```

В процессе выполнения команды `platform push` появится запрос на подтверждение, хотите ли вы загрузить проект ❶. Вдобавок может появиться сообщение о проверке подлинности Platform.sh, если вы впервые подключаетесь к сайту ❷. Введите Y для каждого из этих запросов и увидите, как прокручивается объемный вывод. Поначалу эти результаты, вероятно, покажутся вам непонятными, но если что-то пойдет не так, то они пригодятся при устранении неполадок. Внимательно изучив вывод, вы увидите, как Platform.sh устанавливает необходимые пакеты, собирает статические файлы, выполняет миграции и настраивает URL для проекта.

ПРИМЕЧАНИЕ

Может произойти ошибка из-за простой оплошности, например опечатки в одном из конфигурационных файлов. В этом случае исправьте ошибку в текстовом редакторе, сохраните файл и повторно выполните команду `git commit`. После этого вы можете снова запустить команду `platform push`.

Просмотр проекта в реальном времени

После завершения загрузки вы можете открыть проект:

```
(ll_env)learning_log$ platform url
Enter a number to open a URL
[0] https://main-bvxea6i-wmye2fx7wwqgu.us-3.platformsh.site/
--пропуск--
> 0
```

Команда `platform url` перечисляет URL, связанные с развернутым проектом; вам будет предложено выбрать несколько URL, которые подходят для вашего проекта. Выберите один из них, и ваш проект откроется в новой вкладке браузера! Он будет выглядеть точно так же, как и локальная версия, но вы можете поделиться этим URL с любым пользователем в мире, и он сможет получить доступ к вашему проекту и использовать его.

Доработка развернутого приложения

В этом подразделе мы доработаем развернутое приложение и создадим суперпользователя (так же как делали в локальной версии). Заодно повысим уровень защиты проекта, переведя настройку отладочного режима `DEBUG` в состояние `False`, чтобы пользователи не получали в сообщениях об ошибке дополнительную информацию, которая может использоваться для проведения атак на сервер.

Создание суперпользователя в Platform.sh

База данных для рабочего проекта создана, но совершенно пуста. Все пользователи, которых мы создали ранее, существуют только в локальной версии проекта.

Чтобы создать суперпользователя в рабочей версии проекта, мы запустим SSH-сеанс (secure socket shell), в котором сможем выполнять команды управления на удаленном сервере:

```
(ll_env)learning_log$ platform environment:ssh
```

```
|_ _ \ |_ _ | |_ / |_ _ - - - _ |_ _ | |_
| |_ / / _ | |_ / _ \ ' ' \ _ < ' \
| | | _ \ , \ \ | | \ \ / | | | | ( ) \ | | |
```

Welcome to Platform.sh.

- ❶ web@ll_project.0:~\$ ls
accounts learning_logs ll_project logs manage.py requirements.txt
requirements_remote.txt static
- ❷ web@ll_project.0:~\$ python manage.py createsuperuser
- ❸ Username (leave blank to use 'web'): ll_admin_live
Email address:
Password:
Password (again):
Superuser created successfully.
- ❹ web@ll_project.0:~\$ exit
logout
Connection to ssh.us-3.platform.sh closed.
- ❺ (ll_env)learning_log\$

При первом запуске команды `platform environment:ssh` вы можете получить сообщение о проверке подлинности этого узла. Если увидите такое сообщение, то введите Y, и откроется сеанс удаленного терминала.

После выполнения команды `ssh` ваш терминал будет работать так же, как и на удаленном сервере. Обратите внимание, что подсказка изменилась и указывает на сеанс связи с проектом `ll_project` ❶. Если вы выполните команду `ls`, то увидите файлы, загруженные на сервер Platform.sh.

Выполните команду `createsuperuser`, которую мы использовали в главе 18 ❷. На этот раз я ввел имя администратора, `ll_admin_live`, которое отличается от использовавшегося локально ❸. Завершив работу в удаленной терминальной сессии, выполните команду `exit` ❹. В подсказке будет указано, что вы снова работаете в локальной системе ❺.

Теперь вы можете добавить каталог `/admin/` в конец URL рабочего приложения и перейти на страницу администратора. Если другие пользователи уже

используют ваш проект, то имейте в виду, что у вас будет доступ ко всем их данным! Отнеситесь к этому ответственно, и пользователи будут продолжать доверять вам свои данные.

ПРИМЕЧАНИЕ

Пользователи Windows могут использовать те же команды, которые показаны здесь (например, `ls` вместо `dir`), поскольку вы работаете с терминалом Linux через удаленное соединение.

Безопасность проекта

В том, как сейчас развернут наш проект, есть одна вопиющая уязвимость: настройка `DEBUG = True` в файле `settings.py`, которая выдает отладочные сообщения при возникновении ошибок. Страницы ошибок Django предоставляют важную отладочную информацию при разработке проекта; однако они дают слишком много информации злоумышленникам, если такие страницы будут доступны на рабочем сервере.

Чтобы понять, насколько это опасно, перейдите на главную страницу вашего развернутого проекта. Войдите в учетную запись пользователя и добавьте путь `/topics/999/` в конец URL главной страницы. Если вы еще не успели создать тысячи тем, то по этому адресу увидите страницу с сообщением `DoesNotExist at /topics/999/`. Прокрутив страницу вниз, вы увидите большой объем информации о проекте и сервере. Вряд ли вы хотите, чтобы ваши пользователи видели ее, и уж точно против того, чтобы эта информация была доступна потенциальным злоумышленникам.

Мы можем предотвратить показ этой информации на реальном сайте, установив настройку `DEBUG = False` в части файла `settings.py`, относящейся только к развернутой версии проекта. Таким образом, вы продолжите видеть отладочную информацию локально, где она полезна, однако на развернутом сайте она отображаться не будет.

Откройте файл `settings.py` в вашем редакторе кода и добавьте следующую строку кода в часть, определяющую настройки для Platform.sh:

settings.py

```
--nponyск--
if config.is_valid_platform():
    ALLOWED_HOSTS.append('.platformsh.site')
    DEBUG = False
--nponyск--
```

Вот и все, что нужно сделать в рамках настройки конфигурации для развернутой версии проекта. Когда нужно настроить рабочую версию проекта, мы просто изменяем соответствующую часть конфигурации, созданную ранее.

Фиксация и отправка изменений

Теперь изменения, внесенные в `settings.py`, необходимо зафиксировать, а затем отправить их на Platform.sh. Следующий терминальный сеанс показывает, как это делается:

- ❶

```
(ll_env)learning_log$ git commit -am "Set DEBUG False on live site."
[main d2ad0f7] Set DEBUG False on live site.
1 file changed, 1 insertion(+)
```
- ❷

```
(ll_env)learning_log$ git status
On branch main
nothing to commit, working tree clean
(ll_env)learning_log$
```

Мы вводим команду `git commit` с коротким, но содержательным сообщением ❶. Напомню, что флаг `-am` обеспечивает фиксацию всех изменившихся файлов и регистрацию сообщения в журнале. Git видит, что изменился один файл, и фиксирует изменение в репозитории.

Из результата выполнения команды `git status` понятно, что мы работаем с главной ветвью репозитория, а новые изменения для фиксации отсутствуют ❷. Очень важно проверять информацию о статусе перед отправкой на Platform.sh. Если вы не видите сообщения, значит, некоторые изменения не были зафиксированы и они не будут отправлены на сервер. Попробуйте снова ввести команду `commit`, а если не уверены в том, как решить проблему, — прочитайте приложение Г, чтобы лучше понять, как работать с Git.

Теперь отправим обновленный репозиторий на Platform.sh:

```
(ll_env)learning_log$ platform push
Are you sure you want to push to the main (production) branch? [Y/n] Y
Pushing HEAD to the existing environment main
--пропуск--
To git.us-3.platform.sh:wmye2fx7wwqgu.git
fce0206..d2ad0f7 HEAD -> main
(ll_env)learning_log$
```

Platform.sh видит, что репозиторий обновился, и заново создает проект, чтобы все изменения были учтены. Он не перестраивает базу данных, поэтому мы не потеряли никакие данные.

Чтобы убедиться в том, что эти изменения вступили в силу, снова откройте URL `/topics/999/`. Вы должны увидеть лишь сообщение **Server Error (500)**, без какой-либо конфиденциальной информации о проекте.

Создание специализированных страниц ошибок

В главе 19 мы настроили приложение «Журнал обучения» так, чтобы при запросе темы или записи, которая не принадлежит пользователю, он получал ошибку 404. Вероятно, вы также сталкивались с примерами ошибок 500 (внутренние ошибки).

Ошибка 404 обычно означает, что код Django правильный, но запрашиваемый объект не существует; ошибка 500 обычно означает, что в написанном вами коде есть ошибка (например, ошибка в функции из файла `views.py`). В настоящее время Django возвращает одну обобщенную страницу ошибки в обеих ситуациях, но мы можем написать собственные шаблоны страниц ошибок 404 и 500, которые соответствуют общему оформлению «Журнала обучения». Эти шаблоны должны находиться в корневом каталоге шаблонов.

Создание пользовательских шаблонов

В папке `learning_log` добавьте новую папку `templates`. Затем создайте новый файл `404.html`; полное имя файла имеет вид `learning_log/templates/404.html`. Код файла выглядит так:

404.html

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
    <h2>The item you requested is not available. (404)</h2>
{% endblock page_header %}
```

Этот простой шаблон предоставляет ту же информацию, что и обобщенная страница ошибки 404, но его оформление соответствует остальным страницам сайта.

Создайте другой файл `500.html`, используя данный код:

500.html

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
    <h2>There has been an internal error. (500)</h2>
{% endblock page_header %}
```

Новые файлы потребуют внести небольшие изменения в файл `settings.py`.

settings.py

```
--пропуск--
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],
        'APP_DIRS': True,
        --пропуск--
    },
]
--пропуск--
```

Благодаря этому изменению Django получает указание искать в корневом каталоге `template` шаблоны страниц ошибок и любые другие шаблоны, не связанные с конкретным приложением.

Отправка изменений на Platform.sh

Теперь необходимо зафиксировать изменения в шаблоне и отправить их на Platform.sh:

```
❶ (ll_env)learning_log$ git add .
❷ (ll_env)learning_log$ git commit -am "Added custom 404 and 500 error pages."
 3 files changed, 11 insertions(+), 1 deletion(-)
 create mode 100644 templates/404.html
 create mode 100644 templates/500.html
❸ (ll_env)learning_log$ platform push
--пропуск--
remote: Verifying deploy--пропуск--. done.
To git.us-3.platform.sh:wmye2fx7wwqgu.git
 d2ad0f7..9f042ef HEAD -> main
(ll_env)learning_log$
```

Сначала выдается команда `git add .` ❶, поскольку в проекте были созданы новые файлы, и теперь нужно дать Git указание начать отслеживание этих файлов. Затем мы фиксируем изменения ❷ и отправляем обновленный проект на Platform.sh ❸.

Теперь страницы ошибок оформлены так же, как остальные страницы сайта, а приложение при возникновении ошибок выглядит более профессионально.

Текущая разработка

Возможно, вы захотите продолжить разработку «Журнала обучения» после исходной отправки данных на сервер или создать и развернуть собственные проекты. Процесс обновления проектов определен достаточно четко.

Сначала все необходимые изменения вносятся в локальный проект. Если они приводят к появлению новых файлов, то добавьте эти файлы в репозиторий Git с помощью команды `git add .` (не забудьте поставить точку в конце команды). Эта команда необходима для любого изменения, требующего миграции базы данных, поскольку для каждой миграции генерируется новый файл.

Затем закрепите изменения в репозитории с помощью команды `git commit -am "сообщение"`. Отправьте изменения на Platform.sh, используя команду `platform push`. После этого посетите свой проект и убедитесь в том, что предполагаемые изменения вступили в силу.

В данном процессе легко допустить ошибку, поэтому не удивляйтесь, если что-то пойдет не так. Если код не работает, то проанализируйте сделанное и попробуйте найти ошибку. Если это не удастся или вы не можете понять, как отменить ошибку, то обращайтесь к рекомендациям, изложенным в приложении В. Не стесняйтесь обращаться за помощью: все программисты учились создавать проекты и задавали те же вопросы, которые возникнут и у вас, так что вы наверняка найдете

кого-нибудь, кто согласится помочь. Решение всех возникающих проблем будет помогать развивать ваши навыки до того момента, когда вы начнете создавать содержательные надежные проекты и отвечать на вопросы других людей.

Удаление проекта с Platform.sh

Очень полезно многократно отработать процесс развертывания на одном проекте или серии малых проектов, чтобы получить представление о развертывании. Однако вы должны знать, как удалить проект после развертывания. Кроме того, Platform.sh может ограничивать количество бесплатно развернутых проектов, и загромождать учетную запись учебными проектами нежелательно.

Вы можете удалить проект с помощью CLI следующим образом:

```
(ll_env)learning_log$ platform project:delete
```

Появится запрос с подтверждением, хотите ли вы действительно совершить это деструктивное действие. Ответьте утвердительно, и ваш проект будет удален.

Команда `platform create` также передает локальному Git-репозиторию ссылку на удаленный репозиторий на сервере Platform.sh. Вы можете удалить эту ссылку с помощью командной строки:

```
(ll_env)learning_log$ git remote  
platform  
(ll_env)learning_log$ git remote remove platform
```

Команда `git remote` выводит список имен всех удаленных URL, связанных с текущим репозиторием. Команда `git remote remove имя_удаленного_репозитория` удаляет такие URL из локального репозитория.

Вы также можете удалить ресурсы проекта, войдя на сайт Platform.sh и перейдя на панель управления по адресу <https://console.platform.sh>. На этой странице перечислены все ваши активные проекты. Нажмите кнопку в виде трех точек в строке проекта и выберите пункт **Edit Plan** (Редактировать план). Откройте страницу с тарифными планами для проекта. Нажмите кнопку **Delete Project** (Удалить проект) в нижней части страницы, после чего откроется страница подтверждения, на которой вы сможете подтвердить удаление. Даже если вы удалили свой проект с помощью CLI, нелишним будет ознакомиться с панелью управления хостинг-провайдера, на сервере которого вы разместили свой проект.

ПРИМЕЧАНИЕ

При удалении проекта на Platform.sh с локальной версией проекта ничего не происходит. Если никто не использовал ваш развернутый проект и вы просто отработываете процесс развертывания, то ничто не мешает вам удалить проект с Platform.sh и развернуть его заново. Только имейте в виду, что если проект при этом перестал работать, то вы, возможно, столкнулись с ограничениями бесплатного тарифного плана хостинг-провайдера.

УПРАЖНЕНИЯ

20.3. Блог в Интернете. Разверните проект `Blog`, над которым вы работали ранее, на сервере `Platform.sh`. Проследите за тем, чтобы переменная `DEBUG` имела значение `False`: это помешает пользователям видеть полную страницу ошибки Django при возникновении каких-либо проблем.

20.4. Расширенное приложение «Журнал обучения». Добавьте простую функцию в «Журнал обучения» (например, вывод расширенной информации о проекте на главной странице) и отправьте изменение в развернутую копию. Затем попробуйте внести более сложное изменение — например, дайте пользователю возможность сделать тему общедоступной. Для этого в модель `Topic` добавляется атрибут `public` (по умолчанию он должен быть равен `False`), а на страницу `new_topic` — элемент формы, позволяющий превратить личную тему в общедоступную. После этого проведите миграцию проекта и переработайте файл `views.py`, чтобы любая общедоступная тема была видимой и для пользователей, не прошедших аутентификацию.

Резюме

В этой главе вы узнали, как придать вашему проекту простой, но профессиональный внешний вид с помощью библиотеки `Bootstrap` и приложения `django-bootstrap5`. В случае применения `Bootstrap` выбранные вами стили будут работать одинаково практически на всех устройствах, используемых для работы с вашим проектом.

Вы узнали о шаблонах `Bootstrap` и использовали шаблон *`Navbar static`* для создания простого оформления «Журнала обучения». Вы научились использовать элемент `jumbotron` для визуального выделения сообщений главной страницы и узнали, как оформлять все страницы на сайте в едином стиле.

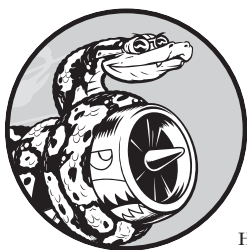
В последней части проекта вы узнали, как развернуть его на удаленном сервере, чтобы с ним мог работать любой желающий. Вы создали учетную запись `Platform.sh` и установили инструменты, упрощающие процесс развертывания. Вы использовали `Git` для фиксации рабочего проекта в репозитории и отправили репозиторий на удаленный сервер `Platform.sh`. Наконец, вы узнали, как защитить приложение, включив режим `DEBUG = False` на работающем сервере. Вы также создали собственные страницы ошибок, поэтому неизбежные возникающие ошибки будут выглядеть хорошо управляемыми.

Итак, работа над «Журналом обучения» закончена, и вы можете перейти к созданию собственных проектов. Начните с простых приложений и убедитесь в том, что приложение заработало, прежде чем повысить уровень сложности. Пусть ваше обучение будет интересным! Удачи!

Приложения

A

Установка Python и диагностика проблем



Python существует в нескольких версиях, с разными вариантами конфигурации в каждой операционной системе. Это приложение пригодится вам в том случае, если описание из главы 1 не сработало или вы захотите установить другую версию Python вместо той, которая поставлялась с вашей системой.

Python в Windows

Инструкции из главы 1 описывают установку Python с помощью официальной программы установки, скачиваемой по адресу <https://python.org/>. Если вам не удастся запустить Python после использования программы установки, то прочитайте инструкции по диагностике, представленные в этом разделе.

Выполнение команды `py` вместо `python`

Если вы установите Python из официального дистрибутива, а затем выполните команду `python` в терминале, то увидите в нем приглашение Python (`>>>`).

Если операционная система Windows не распознает команду `python`, то либо откроет магазин Microsoft Store, посчитав, что Python не установлен, либо выведет сообщение типа `Python was not found` (Python не был найден). Если открылся магазин Microsoft Store, то закройте его; рекомендуется использовать официальный установщик Python с сайта <https://python.org>, а не предлагаемый корпорацией Microsoft.

Самое простое решение, не требующее внесения изменений в систему, — попробовать выполнить команду `py`. Это утилита Windows, которая ищет последнюю версию Python, установленную в системе, и запускает найденный интерпретатор.

Если эта команда работает и вы можете ее использовать, то пишите команду `ru` вместо `python` или `python3` в примерах из этой книги.

Переустановка Python

Чаще всего `python` не запускается из-за того, что люди забывают установить флажок **Add Python to PATH** (Добавить Python в PATH) при установке программы; такую ошибку легко допустить. Переменная `PATH` — это системный параметр, который указывает операционной системе, где искать часто используемые программы. В данном случае Windows не знает, как найти интерпретатор Python.

Самое простое решение в этой ситуации — запустить программу установки еще раз. Если на сайте <https://python.org> доступна более новая версия инсталлятора, то скачайте ее и запустите, не забыв установить флажок **Add Python to PATH** (Добавить Python в PATH).

Если у вас уже скачана последняя версия программы установки, то запустите ее снова и выберите пункт **Modify** (Изменить). Вы увидите список дополнительных параметров; в этом окне оставьте их настроенными по умолчанию. Затем нажмите кнопку **Next** (Далее) и установите флажок **Add Python to Environment Variables** (Добавить Python в переменные среды). Наконец, нажмите кнопку **Install** (Установить). Программа установки распознает, что Python уже установлен, и добавит расположение интерпретатора Python в переменную `PATH`. Убедитесь, что закрыли все открытые окна терминала, поскольку они все еще будут обращаться к предыдущему значению переменной `PATH`. Откройте новое окно терминала и вновь выполните команду `python`; вы должны увидеть приглашение Python (`>>>`).

Python в macOS

В инструкциях из главы 1 описывается официальная программа установки Python, доступная по адресу <https://python.org/>. Она безупречно работает уже много лет, но есть несколько моментов, которые следует учитывать. Этот раздел поможет вам, если возникают затруднения.

Непреднамеренная установка версии Apple

Если вы запустили команду `python3`, а Python еще не установлен в операционной системе, то, скорее всего, увидите сообщение о необходимости установки *консольных инструментов разработчика* (command line developer tools). В этом случае рекомендуется закрыть всплывающее окно с этим сообщением, скачать программу установки Python с сайта <https://python.org> и запустить ее.

Если в этот момент вы согласитесь с предложением установки инструментов разработчика, то вместе с ними система macOS установит версию Python,

распространяемую компанией Apple. Единственная проблема заключается в том, что такая версия Python обычно старше официальной. Тем не менее вы все равно можете скачать официальный установщик с сайта <https://python.org> и запустить его, и тогда команда `python3` будет ссылаться на новую версию. Не волнуйтесь о том, что у вас установлены инструменты разработчика; они содержат несколько полезных дополнений, например систему управления версиями Git, о которой говорится в приложении Г.

Python 2 в старых версиях macOS

В старых версиях операционной системы macOS, до версии Monterey (macOS 12), по умолчанию установлена устаревшая версия Python 2. В таких системах команда `python` ссылается на устаревший системный интерпретатор. Если вы используете версию macOS с предустановленным интерпретатором Python 2, то всегда выполняйте команду `python3`, чтобы использовать актуальную версию Python 3, установленную вами вручную.

Python в системе Linux

Интерпретатор Python предустанавливается по умолчанию практически во всех операционных системах Linux. Но если по умолчанию в вашей системе используется версия до 3.9, то вам нужно установить обновление. Вместе с новой версией вы получите доступ к актуальным возможностям, например улучшенной системе оповещения об ошибках Python. Приведенные ниже инструкции применимы в большинстве систем на базе apt.

Использование стандартной установки Python

Если вы хотите использовать версию `python3`, то убедитесь, что у вас установлены эти три дополнительных пакета:

```
$ sudo apt install python3-dev python3-pip python3-venv
```

Они содержат инструменты для разработчиков и компоненты, позволяющие устанавливать сторонние пакеты, например описанные в части II.

Установка последней версии Python

Мы воспользуемся пакетом `deadsnakes`, который позволяет легко установить несколько версий Python. Выполните следующие команды:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt update
$ sudo apt install python3.11
```

Эти команды установят версию Python 3.11 в вашей операционной системе.

Введите следующую команду, чтобы запустить сеанс терминала с Python 3.11:

```
$ python3.11
>>>
```

Во всех примерах в этой книге, где указана команда `python`, используйте вместо нее `python3.11`. Кроме того, эту команду следует вводить, чтобы запускать программы из терминала.

Вам нужно установить дополнительные два пакета, чтобы использовать возможности вашего интерпретатора Python по максимуму:

```
$ sudo apt install python3.11-dev python3.11-venv
```

Эти пакеты содержат модули, которые понадобятся вам при установке и запуске пакетов сторонних разработчиков, например используемых в проектах из второй части книги.

ПРИМЕЧАНИЕ

Пакет `deadsnakes` активно поддерживается на длительной основе. По мере выпуска новых версий Python вы можете выполнять схожие команды, заменив команду `python3.11` на используемую версию.

Проверка установленной версии Python

Если у вас возникли проблемы с запуском Python или установкой дополнительных пакетов, то рекомендуется начать с уточнения, какую версию Python вы используете. Возможно, у вас установлено несколько версий Python, и вы не знаете, какая из них задействуется в данный момент.

Выполните в терминале следующую команду:

```
$ python --version
Python 3.11.0
```

В выводе будет указано, на какую версию в данный момент ссылается команда `python`. Сокращенная версия команды `python -V` выдаст тот же результат.

Ключевые слова и встроенные функции Python

Python содержит целый набор ключевых слов и встроенных функций. Учитывайте их, выбирая имена переменных. В качестве имен нельзя использовать ключевые слова Python, а также имена встроенных функций, поскольку это приведет к конфликту функций.

В этом разделе перечислены ключевые слова Python и имена встроенных функций, которых вам следует избегать при именовании собственных переменных.

Ключевые слова Python

Каждое ключевое слово из следующего списка имеет собственное предназначение в программах Python. При попытке использовать эти слова в качестве имен переменных произойдет ошибка.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Встроенные функции Python

Если вы используете любую из следующих встроенных функций в качестве имени переменной, это приведет не к ошибке, а изменению поведения этой функции:

abs()	complex()	hash()	min()	slice()
aiter()	delattr()	help()	next()	sorted()
all()	dict()	hex()	object()	staticmethod()
any()	dir()	id()	oct()	str()
anext()	divmod()	input()	open()	sum()
ascii()	enumerate()	int()	ord()	super()
bin()	eval()	isinstance()	pow()	tuple()
bool()	exec()	issubclass()	print()	type()
breakpoint()	filter()	iter()	property()	vars()
bytearray()	float()	len()	range()	zip()
bytes()	format()	list()	repr()	__import__()
callable()	frozenset()	locals()	reversed()	
chr()	getattr()	map()	round()	
classmethod()	globals()	max()	set()	
compile()	hasattr()	memoryview()	setattr()	

Б

Редакторы кода и IDE



Программисты проводят много времени за написанием, чтением и редактированием кода в редакторе кода или *интегрированной среде разработки* (Integrated Development Environment, IDE).

Очень важно, чтобы эта работа выполнялась по возможности результативно. Эффективный редактор должен реализовывать простые задачи: например, выделять структуру кода, чтобы вы могли обнаружить типичные ошибки во время работы, — но не до такой степени, чтобы это отвлекало вас от процесса. Кроме того, редактор должен поддерживать автоматическую расстановку отступов, маркеры для обозначения длины строки и комбинации клавиш для часто выполняемых операций.

IDE представляет собой редактор кода, дополненный встроенными инструментами — например, интерактивными отладчиками и интроспекцией. IDE анализирует ваш код в процессе ввода и пытается использовать информацию о создаваемом проекте. Например, когда вы начинаете вводить имя функции, IDE может вывести список всех получаемых ею аргументов. Такое поведение может быть очень полезно, когда среда работает идеально, а вы хорошо понимаете все происходящее. Но оно же может сбить с толку новичка и усложнить исправление ошибок, когда вы не понимаете, почему ваш код не работает в IDE.

В наши дни границы между редакторами кода и IDE размыты. В большинстве популярных редакторов есть ряд функций, которые раньше имели место только в IDE. Аналогично большинство IDE можно настроить на работу в упрощенном удобном режиме, позволяющем использовать расширенные функции по мере необходимости.

Если у вас уже есть редактор кода или IDE и в нем настроена поддержка последней версии Python, установленной в вашей системе, то я советую вам пользоваться именно им. Изучение различных редакторов может быть очень интересным занятием, но усложняет темп изучения нового языка программирования.

Если у вас еще не установлен редактор кода или IDE, то я рекомендую использовать программу VS Code, и вот почему.

- Распространяется бесплатно в соответствии с лицензией с открытым исходным кодом.
- Поддерживается всеми основными операционными системами.
- Удобна для начинающих, но при этом достаточно мощная, поэтому многие профессиональные программисты используют ее в качестве основного редактора.
- Автоматически определяет версии установленных интерпретаторов Python и обычно не требует специальной настройки для запуска разрабатываемых программ.
- Имеет терминал, в котором вывод отображается рядом с исходным кодом.
- Поддерживает расширение Python, повышающее эффективность редактора в части написания и сопровождения кода на Python.
- Обладает гибкими настройками, поэтому вы можете настроить редактор в соответствии со своим стилем работы с кодом.

В этом приложении вы узнаете, как настроить VS Code под свои потребности. Вы также узнаете о некоторых сочетаниях клавиш, повышающих эффективность работы. Большая скорость набора кода в программировании не так важна, как думают многие, но знать функционал редактора и уметь эффективно его использовать крайне полезно.

При всем этом программа VS Code подойдет не всем. Если она по каким-то причинам не работает в вашей системе или затрудняет работу, то воспользуйтесь другими редакторами, которые могут подойти вам больше. В этом приложении приведено краткое описание некоторых распространенных редакторов кода и IDE, которые следует рассмотреть.

Эффективная работа в программе VS Code

В главе 1 вы установили программу VS Code и добавили расширение Python. В этом разделе вы узнаете о некоторых дополнительных настройках, а также о сочетаниях клавиш для эффективной работы с кодом.

Настройка программы VS Code

Существует несколько способов изменить стандартные настройки программы VS Code. Некоторые изменения можно выполнить через интерфейс, а другие требуют внесения изменений в конфигурационные файлы. Одни изменения влияют на все проекты VS Code, а другие — только на файлы в папке, содержащей файл конфигурации.

Например, если файл конфигурации находится в папке `python_work`, то внесенные в него изменения будут влиять только на файлы в этой папке (и вложенных в нее). Это удобно, так как вы можете настраивать отдельно конкретный проект, не затрагивая глобальные настройки.

Использование табуляций и пробелов

Чередование символов табуляции и пробелов в коде может создать трудно диагностируемые проблемы в программах. При работе в файле `.py` с установленным расширением Python программа VS Code автоматически настраивается на вставку четырех пробелов при нажатии клавиши **Tab**. Если вы работаете только со своим кодом и у вас установлено расширение Python, то, скорее всего, у вас никогда не возникнет проблем с табуляцией и пробелами.

Однако, возможно, ваша программа VS Code настроена неправильно. Кроме того, вы можете открыть файл стороннего разработчика, содержащий только табуляцию или смесь табуляции и пробелов. Если у вас возникли подозрения на этот счет, то взгляните на строку состояния в нижней части окна программы VS Code и щелкните кнопкой мыши на пункте **Spaces** (Пробелы) или **Tab Size** (Размер табуляции). Вы увидите раскрывающийся список, в котором можно переключиться между использованием табуляции и пробелов. Вы также можете изменить уровень отступа по умолчанию и преобразовать все отступы в файле в табуляцию или же пробелы.

Если вы просматриваете код и не уверены в том, из каких символов состоит отступ — из табуляции или пробелов, — то выделите несколько строк кода. Так невидимые пробельные символы станут видимыми. Пробелы будут представлены точками, а табуляция — стрелками.

ПРИМЕЧАНИЕ

В программировании пробелы предпочтительнее табуляции, поскольку пробелы могут быть корректно интерпретированы всеми программами, в которых открываются файлы с кодом. Ширина символа табуляции может по-разному интерпретироваться разными инструментами, что приводит к ошибкам, которые крайне сложно диагностировать.

Изменение цвета интерфейса

По умолчанию в программе VS Code используется темная тема интерфейса. Если вы хотите изменить ее, то откройте меню **File** (Файл) (**Code** (Код) в операционной системе macOS), затем выберите пункт **Preferences** (Настройки) и перейдите в раздел **Color Theme** (Цветовая тема). В раскрывающемся списке выберите подходящую тему интерфейса.

Выбор индикатора длины строки

В большинстве редакторов можно задать визуальный признак (обычно вертикальную линию), обозначающий рекомендуемую длину строки. В сообществе Python, согласно общепринятым правилам, строки могут иметь длину 79 символов и менее.

Чтобы задать эту настройку, откройте меню **File (Файл) (Code (Код))** в операционной системе macOS), затем выберите пункт **Preferences (Настройки)** и в появившемся диалоговом окне введите слово **rulers**. Вы увидите раздел настроек **Editor: Rulers (Редактор: Rulers)**. Щелкните на ссылке **Edit in settings.json (Редактировать в settings.json)**. В открывшемся файле добавьте в параметр **editor.rulers** следующее значение:

settings.json

```
"editor.rulers": [  
    80,  
]
```

Так вы добавите в окно редактирования программы VS Code вертикальную линию в позиции 80-го символа. Вы можете добавить несколько вертикальных линий; например, если нужна дополнительная линия в позиции 120-го символа, то присвойте параметру значение **[80, 120]**. Если вы не видите вертикальные линии, то убедитесь, что сохранили файл настроек. Кроме того, в некоторых системах может потребоваться перезапуск программы VS Code, чтобы изменения вступили в силу.

Сокращение вывода

По умолчанию VS Code отображает вывод ваших программ в окне встроенного терминала. По умолчанию он хранит предыдущие команды, использовавшиеся для запуска файла. В большинстве случаев это идеальный вариант, но для новичков в программировании на Python вывод может показаться излишним.

Чтобы сократить вывод, закройте все вкладки, открытые в VS Code, а затем закройте саму программу. Снова запустите ее и откройте папку, в которой расположены файлы Python, над которыми вы работаете. К примеру, это может быть папка **python_work**, в которой сохранен файл **hello_world.py**.

Нажмите кнопку **Run/Debug (Запуск/Отладка)** (похожа на треугольник с маленьким жучком), а затем выберите пункт **Create a launch.json File (Создать файл launch.json)**. Выберите варианты Python в появившихся запросах. В открывшийся файл **launch.json** внесите следующие изменения:

launch.json

```
{  
    --пропуск--  
    "configurations": [  

```

```
{
    --пропуск--
    "console": "internalConsole",
    "justMyCode": true
}
]
```

Мы изменили настройки терминала с `integratedTerminal` на `internalConsole`. Сохранив файл настроек, откройте файл `.py`, например, `hello_world.py`, и запустите его, нажав `Ctrl+F5`. На панели вывода программы VS Code перейдите на вкладку `Debug Console` (Консоль отладки), если она еще не открыта. Вы увидите только вывод вашей программы, который будет обновляться при каждом запуске программы.

ПРИМЕЧАНИЕ

Консоль отладки доступна только для чтения. В ней нельзя взаимодействовать с файлами, содержащими функцию `input()`, с которой мы познакомились в главе 7. Когда вам понадобится работать с такими программами, вы можете либо откатить настройки консоли до стандартного значения `integratedTerminal`, либо запускать эти программы в отдельном окне терминала, как описано в разделе «Запуск программ Python из терминала» в главе 1.

Изучение дополнительных настроек

Вы можете настроить программу VS Code различными способами, чтобы работать более эффективно. Чтобы изучить доступные настройки, откройте меню `File` (Файл) (`Code` (Код) в операционной системе `macOS`), а затем выберите пункт `Preferences` (Настройки). Вы увидите список `Commonly Used` (Часто используемые); щелкните кнопкой мыши на любом из подзаголовков, чтобы увидеть распространенные способы изменения настроек программы VS Code. Не пожалейте времени и посмотрите, есть ли среди них те, которые помогут вам работать в редакторе эффективнее, но не заикливайтесь на его настройке и не откладывайте изучение Python на потом!

Горячие клавиши в VS Code

Все редакторы кода и IDE подразумевают эффективные способы выполнения рутинных задач, которые приходится часто решать при написании и сопровождении кода. Например, вы можете легко сделать отступ для одной строки кода или нескольких, быстро переместить блок кода вверх или вниз в файле и т. п.

Существует громадное количество сочетаний клавиш, поэтому будет неразумно давать здесь их полное описание. В этом подразделе я расскажу лишь о тех из них, которые гарантированно пригодятся вам при разработке программ на Python. Если же в итоге вы будете использовать не VS Code, а другой редактор, то обязательно изучите сочетания клавиш для решения тех же задач в выбранной вами программе.

Создание и удаление отступов кода

Чтобы сделать отступ для нескольких строк кода, выделите их и нажмите сочетание клавиш **Ctrl+]** или **⌘+]** в macOS. Чтобы удалить отступ, выделите строки кода и нажмите **Ctrl+[** или **⌘+[** в macOS.

Комментирование кода

Чтобы временно отключить блок кода, вы можете выделить его и закомментировать; тогда интерпретатор Python будет его игнорировать. Выделите фрагмент кода, который хотите проигнорировать, и нажмите **Ctrl+/** или **⌘+/** в macOS. Выбранные строки кода будут закомментированы с помощью символов решетки (**#**), имеющих такой же уровень отступа, как и строка кода; таким образом указывается, что это не обычные комментарии. Чтобы раскомментировать блок кода, выделите его и вновь нажмите то же сочетание клавиш.

Перемещение кода вверх/вниз

По мере усложнения программ вам может потребоваться переместить блок кода в файле вверх или вниз. Для перемещения вверх выделите нужный код и нажмите сочетание клавиш **Alt+↑** или **Option+↑** в macOS. Та же комбинация клавиш со стрелкой **↓** переместит код в файле вниз.

Если вы перемещаете вверх или вниз одну строку, то можете щелкнуть кнопкой мыши в любой позиции этой строки; вам не нужно выделять всю строку, чтобы переместить ее.

Скрытие обозревателя файлов

Встроенный обозреватель файлов в программе VS Code очень удобен. Однако во время написания кода он может отвлекать и занимать много пространства на небольшом экране. Сочетание клавиш **Ctrl+B** или **⌘+B** в macOS отображает/скрывает панель обозревателя файлов.

Дополнительные сочетания клавиш

Эффективная работа в программе редактирования кода требует не только практики, но и внимания. Обучаясь работать с кодом, обращайте внимание на действия, которые приходится выполнять постоянно. Любое действие, которое вы выполняете в редакторе, скорее всего, обозначается сочетанием клавиш. Если открыть меню с соответствующей командой редактирования, рядом с ней может быть указано сочетание клавиш. Если вы часто переключаетесь между клавиатурой и мышью, то найдите сочетания клавиш, которые позволят вам реже обращаться к мышью.

Вы можете просмотреть все сочетания клавиш, используемые в программе VS Code, открыв меню **File (Файл) (Code (Код))** в операционной системе macOS, выбрав пункт

Preferences (Настройки), а затем перейдя в раздел **Keyboard Shortcuts** (Сочетания клавиш). Вы можете воспользоваться поиском, чтобы найти конкретное сочетание клавиш, или прокрутить список, чтобы найти сочетания, которые помогут вам работать более эффективно.

Помните, что лучше сосредоточиться на коде, над которым вы работаете, и сократить время неэффективного использования инструментов.

Другие редакторы кода и IDE

Многие разработчики пользуются другими редакторами кода; наверняка вы не раз услышите о них. Как правило, такие редакторы настраиваются по тем же принципам, что и VS Code. Ниже приведена небольшая подборка редакторов кода, с которыми вы можете столкнуться.

IDLE

IDLE — редактор кода, добавленный в дистрибутив Python. В работе он чуть менее интуитивен, чем другие аналогичные программы, но часто упоминается в других учебниках, предназначенных для начинающих, поэтому вам стоит познакомиться с ним.

Geany

Geany — простой редактор кода, который выводит результаты в терминальном окне, что поможет вам освоить работу в терминале. Интерфейс этого редактора очень простой, но сам он достаточно мощный, поэтому многие опытные программисты продолжают пользоваться им.

Если вы считаете, что программа VS Code излишне сложна для вас, то попробуйте использовать *Geany*.

Sublime Text

Sublime Text — еще один редактор с простым интерфейсом, который рекомендуется использовать, если VS Code кажется вам перегруженным. Интерфейс *Sublime Text* очень понятный и известен хорошей работой даже с очень большими файлами. Это редактор, который не будет вас отвлекать и позволит сосредоточиться на коде, который вы пишете.

Существует неограниченная по времени использования пробная версия *Sublime Text*, но редактор не является бесплатным и не распространяется как открытый исходный код. Если эта программа придется вам по душе, то вы можете приобрести лицензию на ее использование. Заплатить нужно будет один раз; лицензия пожизненная, а не в виде подписки.

Emacs и Vim

Многие опытные программисты отдают предпочтение *Emacs* или *Vim*. Эти два популярных редактора спроектированы так, чтобы пользователю не приходилось отрывать руки от клавиатуры. Это означает, что опытный пользователь может читать, писать и редактировать код весьма эффективно. С другой стороны, для освоения этих редакторов придется основательно потрудиться. *Vim* входит в поставку большинства систем Linux и macOS, причем как *Emacs*, так и *Vim* могут выполняться полностью в режиме терминала. Поэтому они часто используются для написания кода на серверах через удаленный терминальный сеанс.

Программисты часто советуют хотя бы опробовать эти редакторы, но многие профессионалы забывают, как много всего нового узнает новичок. Знать о существовании этих редакторов полезно, но отложите знакомство с ними до того момента, когда начнете уверенно писать программы и работать с ними в более простых редакторах, которые позволяют сосредоточиться на изучении программирования, а не на работе с редактором.

PyCharm

Среда *PyCharm* популярна среди программистов Python, поскольку была создана специально для работы с этим языком. Полная версия требует платной подписки, но существует и бесплатная версия *PyCharm Community Edition*, которую многие разработчики считают полезной.

Касательно работы в программе *PyCharm* имейте в виду, что она по умолчанию создает изолированную среду для каждого проекта. Обычно ничего плохого в этом нет, но в некоторых случаях поведение может быть неожиданным, особенно если вы не осведомлены о нем.

Jupyter Notebook

Jupyter Notebook отличается от традиционных редакторов кода или IDE тем, что это веб-приложение, собирающее проекты из блоков. Они могут быть как программными, так и текстовыми. В текстовых может содержаться разметка Markdown, что позволяет добавлять в них простое форматирование.

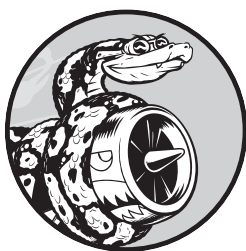
Документы *Jupyter Notebook* были разработаны для применения Python в научных приложениях, но с того времени развивались и нашли применение в широком спектре ситуаций. Вместо того чтобы добавлять обычные комментарии в файл `.py`, вы вставляете между частями кода понятный текст с несложным форматированием: заголовками, маркированными списками и гиперссылками. Каждый блок кода может выполняться независимо от других, что позволяет тестировать небольшие части программы; также можно выполнить все блоки одновременно. Каждый блок имеет собственную область вывода, причем эти области можно включать и отключать по мере надобности.

Документы Jupyter Notebook могут создавать некоторую путаницу из-за взаимодействия между ячейками. Функция, определенная в одной ячейке, становится доступной для других ячеек. В большинстве случаев это удобно, но возможны и недоразумения — например, в больших документах, или если вы недостаточно хорошо понимаете, как работает среда Notebook.

Каждый разработчик, занимающийся научной работой или задачами обработки данных на Python, почти неизбежно столкнется с Jupyter Notebook в какой-то момент своей деятельности.

В

Помощь и поддержка



Во время изучения программирования каждый из нас в какой-то момент оказывается в тупике. Один из важнейших навыков, которые должен освоить каждый программист, — умение быстро найти из него выход. В этом приложении описаны способы решения проблем, которые помогут вам выйти из сложной ситуации.

Первые шаги

Если у вас возникли трудности, то прежде всего оцените ситуацию. Прежде чем обращаться за помощью, убедитесь в том, что можете четко ответить на следующие три вопроса:

- Что вы пытаетесь сделать?
- Что вы делали до настоящего момента?
- Какие результаты вы получили?

Ваши ответы должны быть максимально конкретными. Например, в первом вопросе развернутое утверждение «Я пытаюсь установить последнюю версию Python на свой новый ноутбук с Windows» достаточно подробно, чтобы другие пользователи сообщества Python могли вам помочь. Формулировки типа «Я пытаюсь установить Python» не содержат столько информации, чтобы кто-то мог предложить вам помощь.

Ответ на второй вопрос должен быть достаточно развернутым, чтобы вам не предлагали делать то, что уже было сделано: описание «Я открыл страницу <https://python.org/downloads/> и выбрал кнопку Download (Скачать) для моей системы. Затем

я запустил программу установки» более полезно, чем «Я зашел на сайт Python и что-то скачал».

Что касается последнего вопроса, то при поиске в Интернете или обращении за помощью желательно знать точные сообщения об ошибках.

Иногда в процессе поиска ответов на эти три вопроса вы сами понимаете, где была допущена оплошность, и выходите из тупика самостоятельно. У программистов даже имеется специальный термин для таких ситуаций: *«отладка методом утенка»* (rubber duck debugging). Если вы четко объясните свою ситуацию резиновому утенку (или любому другому неодушевленному объекту) и зададите конкретный вопрос, то часто сможете ответить на него. Некоторые организации даже приобретают резиновую уточку, чтобы побудить своих программистов к использованию этого метода.

Попробуйте заново

Просто вернитесь к началу и попробуйте еще раз; часто этого оказывается достаточно для решения многих проблем. Допустим, вы пытаетесь написать цикл `for` на основе примера из книги. Возможно, вы пропустили что-то совсем простое — скажем, двоеточие в конце строки. Повторное выполнение всех действий поможет избежать той же ошибки.

Сделайте перерыв

Если вы уже долго пытаетесь решить какую-то проблему, то перерыв — едва ли не лучшее, что можно сделать. Когда мы трудимся над одной задачей в течение долгого времени, наш мозг начинает концентрироваться на единственном решении. Мы забываем о сделанных предположениях, а перерыв помогает взглянуть на проблему под новым углом. Перерыв даже не обязан быть долгим, просто нужно заняться чем-то, что выведет вас из текущего мысленного настроения. Если вы давно сидите на одном месте, то переключитесь на какую-нибудь физическую нагрузку: пройдите или выйдите на улицу; может, выпейте стакан воды или съешьте что-нибудь легкое и здоровое.

Если вы начинаете отчаиваться, то попробуйте отложить работу на следующий день. Хороший сон почти всегда упрощает задачу.

Обратитесь к дополнительным материалам этой книги

В список онлайн-ресурсов этой книги (https://ehmatthes.github.io/pcc_3e) добавлен ряд полезных разделов, посвященных настройке системы и обзорам каждой главы. Просмотрите эти материалы — возможно, вы найдете в них то, что вам поможет.

Поиск в Интернете

Вполне вероятно, что кто-то уже столкнулся с такой же проблемой и написал о ней в Интернете. Хорошие навыки поиска и конкретные запросы помогут вам найти информацию для решения ваших проблем. Например, если у вас возникли трудности с установкой Python в новой версии Windows, то поиск по условию *установка python windows* может привести вас к ответу.

Поиск по точным сообщениям об ошибках тоже может оказаться исключительно полезным. Допустим, при попытке запустить Python в терминальном сеансе в новой операционной системе Windows произошла следующая ошибка:

```
> python hello_world.py
Python was not found; run without arguments to install from the Microsoft Store...
```

Вероятно, поиск по полному тексту сообщения принесет полезную информацию.

В результатах поиска, связанного с программированием, особенно часто встречаются некоторые сайты. Я опишу часть из них, чтобы вы знали, чего от них можно ждать.

Stack Overflow

Stack Overflow (<http://stackoverflow.com/>) — один из самых популярных сайтов с вопросами и ответами для программистов, который часто встречается на первой странице результатов поиска, связанного с Python. Пользователи публикуют вопросы по возникшим проблемам, а другие участники пытаются дать полезные ответы. Пользователи могут голосовать за ответы, которые, по их мнению, были наиболее полезны, так что лучшими ответами обычно оказываются первые из найденных.

На сайте Stack Overflow можно найти ответы на многие распространенные вопросы по языку Python, поскольку со временем они были хорошо проработаны. Пользователи также публикуют обновления, так что ответы остаются относительно актуальными. На момент написания книги на сайте Stack Overflow были опубликованы ответы более чем на 2 миллиона вопросов, связанных с Python.

Прежде чем публиковать свой вопрос на сайте Stack Overflow, учитывайте следующее требование. Вопрос должен содержать кратчайший пример проблемного кода. Если вы опубликуете листинг из не более чем 5–20 строк кода, выдающих ошибку, и ответите на вопросы, упомянутые в начале этого приложения, то вам наверняка помогут. Если вы поделитесь ссылкой на свой проект с огромным количеством файлов, то люди вряд ли помогут. На странице <https://stackoverflow.com/help/how-to-ask> опубликовано отличное руководство по составлению грамотного вопроса. Советы, приведенные в этом руководстве, применимы для получения помощи в любом сообществе программистов.

Официальная документация Python

Официальная документация Python (<http://docs.python.org/>) уже не столь бесспорно полезна для новичков, поскольку написана для документирования языка, а не для разъяснений. Примеры в официальной документации должны работать, но, возможно, что-то в них останется для вас непонятным. Тем не менее это полезный ресурс, к которому стоит обращаться при поиске, а по мере углубления вашего понимания Python он будет приносить еще больше пользы.

Официальная документация библиотек

Если вы используете конкретную библиотеку (например, Pygame, Matplotlib, Django и т. д.), то в поиске часто будут встречаться ссылки на официальную документацию этого проекта — например, документация <http://docs.djangoproject.com/> чрезвычайно полезна. Если вы собираетесь работать с любыми из этих библиотек, то вам стоит ознакомиться с их официальной документацией.

Форум r/learnpython

Форум Reddit состоит из ряда *подфорумов* (subreddits). Участники подфорума r/learnpython (<http://reddit.com/r/learnpython/>) достаточно активны и настроены благожелательно. Здесь вы сможете прочитать вопросы других участников и опубликовать собственные.

Сообщения в блогах

Многие программисты ведут блоги и пишут об аспектах языка, с которыми работают. Прежде чем брать на вооружение любой совет, просмотрите несколько первых комментариев к сообщению в блоге. Если комментариев нет, то к сообщению следует относиться скептически. Вполне возможно, что никто другой не смог убедиться в полезности этого совета.

Discord

Discord — еще одна чат-среда с сообществом Python, в которой можно попросить о помощи и читать обсуждения, связанные с Python.

Чтобы поближе познакомиться с Discord, откройте страницу <https://pythondiscord.com/> и щелкните на ссылке Discord. Если у вас уже есть учетная запись Discord, то вы можете выполнить вход, используя ее данные. Если же записи еще нет, то введите имя пользователя и следуйте инструкциям для завершения процесса регистрации.

Если вы впервые посещаете Python-площадку Discord, то для полноценного участия необходимо принять правила сообщества. Когда это будет сделано, вы сможете присоединиться к любому из интересующих вас каналов. Если вам потребуется помощь, то отправьте свой вопрос на одном из каналов Python Help.

Slack

Slack — еще одна среда для онлайн-общения. Каналы Slack часто используются для внутренней коммуникации в компаниях, но существует и много публичных групп, к которым вы можете присоединиться. Если вы захотите просмотреть Slack-группы Python, то начните с <https://pyslackers.com/>. Чтобы получить приглашение, щелкните на ссылке **Slack** в верхней части страницы и введите свой адрес электронной почты.

Оказавшись в рабочем пространстве Python Developers, вы увидите список каналов. Щелкните на ссылке **Channels** (Каналы) и выберите тему, которая вас интересует. Возможно, вам стоит начать с каналов **#help** и **#django**.



Управление версиями с помощью Git



Программы управления версиями позволяют создать снимок состояния программы (snapshot), находящейся в рабочем состоянии. При внесении изменений в проект (например, при реализации новой возможности) можно вернуться к предыдущему работоспособному состоянию, если в новом состоянии проект работает не так, как ожидалось.

Благодаря программам управления версиями программист может работать над усовершенствованием проекта и совершать ошибки, не опасаясь нарушить его работоспособность. Это особенно важно в больших проектах, но может быть полезно и в маленьких — даже в случае программ, содержащихся в одном файле.

В этом приложении вы узнаете, как установить Git и использовать эту систему для управления версиями программ, над которыми вы работаете. В настоящее время Git — самая популярная программа управления версиями. Многие расширенные возможности Git предназначены для групп, совместно работающих над большими проектами, но базовые функции хорошо подходят и для разработчиков-одиночек. Управление версиями в Git основано на отслеживании изменений, вносимых в каждый файл проекта; если вы совершите ошибку, то сможете просто вернуться к ранее сохраненному состоянию.

Установка Git

Git работает во всех операционных системах, но способы установки в конкретных системах разнятся. Ниже приведены инструкции для всех операционных систем.

Программа Git добавлена в некоторые системы по умолчанию и часто входит в состав других пакетов, которые вы, возможно, уже установили. Прежде чем

устанавливать по умолчанию Git, проверьте, есть ли она в вашей системе. Откройте окно терминала и выполните команду `git --version`. Если в выводе будет указан номер версии, значит, Git уже установлена. Если увидите сообщение с предложением установить или обновить Git, то следуйте инструкциям на экране.

Если на экране нет инструкций и вы пользуетесь операционной системой Windows или macOS, то можете скачать программу установки с сайта <https://git-scm.com>. Если вы пользователь apt-версии операционной системы Linux, то можете установить Git с помощью команды `sudo apt install git`.

Настройка Git

Git отслеживает пользователей, вносящих изменения в проект, — даже если над проектом работает всего один человек. Для этого Git необходимо знать ваше имя пользователя и пароль. Имя пользователя следует указать обязательно, но ничто не мешает вам ввести фиктивный адрес электронной почты:

```
$ git config --global user.name "имя_пользователя"
$ git config --global user.email "имя_пользователя@пример.com"
```

Если вы забудете это сделать, то Git запросит информацию при первой фиксации.

Кроме того, рекомендуется настроить имя по умолчанию для главной ветви каждого проекта. Подходящее имя для этой ветви — `main`:

```
$ git config --global init.defaultBranch main
```

Согласно этой конфигурации, каждый новый проект, для управления версиями которого используется Git, будет начинаться с единственной ветви фиксаций, названной `main`.

Создание проекта

Для начала создадим проект для работы. Создайте в системе папку `git_practice`. В эту папку поместите файл с простой программой Python:

```
hello_git.py
print("Hello Git world!")
```

Эта программа поможет вам изучить базовую функциональность Git.

Игнорирование файлов

Файлы с расширением `.рус` автоматически создаются на основе файлов `.py`, и отслеживать их в Git не нужно. Эти файлы хранятся в каталоге `__pycache__`. Чтобы Git игнорировал его, создайте специальный файл `.gitignore` (с точкой в начале имени и без расширения) и вставьте в него следующую строку:

```
.gitignore
__pycache__/
```

В результате Git будет игнорировать все файлы, находящиеся в каталоге `__pycache__`. Файл `.gitignore` избавляет проект от излишнего «балласта» и упрощает работу с ним.

Возможно, для открытия файла `.gitignore` вам придется изменить настройки своего файлового менеджера, чтобы в нем отображались скрытые файлы (имена которых начинаются с точки). В Проводнике установите флажок **Hidden Items** (Скрытые элементы) в меню **View** (Вид). В macOS нажмите сочетание клавиш **⌘+Shift+.** (точка). В операционной системе Linux найдите настройку **Show Hidden Files** (Показать скрытые файлы).

ПРИМЕЧАНИЕ

Если вы используете macOS, то добавьте в `.gitignore` еще одну строку: имя `.DS_Store`. Это скрытые файлы, содержащие информацию о каждом каталоге в macOS, и они будут загромождать ваш проект, если вы не добавите их в `.gitignore`.

Инициализация репозитория

Теперь, когда у вас есть каталог с файлом Python и файлом `.gitignore`, можно переходить к инициализации репозитория Git. Откройте терминал, перейдите в каталог `git_practice` и выполните следующую команду:

```
git_practice$ git init
Initialized empty Git repository in git_practice/.git/
git_practice$
```

Из выходных данных видно, что Git инициализирует пустой репозиторий в каталоге `git_practice`. *Репозиторий* (repository) — это набор файлов программы, который активно отслеживается системой Git. Все файлы, используемые Git для управления репозиторием, хранятся в скрытом каталоге `.git/`, с которым вам вообще не придется работать. Просто не удаляйте этот каталог, иначе вся история проекта будет потеряна.

Проверка статуса

Прежде чем делать что-либо, проверьте статус проекта:

```
git_practice$ git status
❶ On branch main
No commits yet

❷ Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
.gitignore
hello_git.py
```

- ❸ nothing added to commit but untracked files present (use "git add" to track)
git_practice\$

В Git *ветвью* (branch) называется версия проекта, над которым вы работаете; из вывода видно, что текущей является ветвь **main** ❶. Каждый раз, когда вы проверяете статус своего проекта, программа должна сообщать имя текущей ветви. После этого мы видим, что система готова к начальной фиксации. *Фиксацией* (commit) называется снимок состояния проекта на определенный момент времени.

Git сообщает, что в проекте имеются неотслеживаемые файлы ❷, поскольку мы еще не сообщили системе, какие файлы должны отслеживаться. Затем мы узнаем, что в текущую фиксацию еще ничего не добавлено, но существуют неотслеживаемые файлы, которые следует добавить в репозиторий ❸.

Добавление файлов в репозиторий

Добавим в репозиторий два файла и снова проверим статус:

- ```
❶ git_practice$ git add .
❷ git_practice$ git status
On branch main
No commits yet

Changes to be committed:
 (use "git rm --cached <file>..." to unstage)
❸ new file: .gitignore
 new file: hello_git.py

git_practice$
```

Команда `git add .` добавляет в репозиторий все файлы проекта, которые еще не отслеживаются ❶, если они не внесены в список файла `.gitignore`. Фиксация при этом не выполняется; команда просто сообщает Git, что на эти файлы нужно обратить внимание. При проверке статуса проекта мы видим, что Git находит изменения, которые необходимо зафиксировать ❷. Метка `new file` означает, что эти файлы были только что добавлены в репозиторий ❸.

## Фиксация изменений

Выполним первую фиксацию:

- ```
❶ git_practice$ git commit -m "Started project."
❷ [main (root-commit) cea13dd] Started project.
❸ 2 files changed, 5 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 hello_git.py
```



```
④ git_practice$ git status
On branch main
nothing to commit, working tree clean
git_practice$
```

Команда `git commit -m "сообщение"` ❶ создает снимок состояния проекта. Благодаря флагу `-m` Git получает указание сохранить следующее сообщение ("Started project.") в журнале проекта. Из вывода следует, что текущей является ветвь `main` ❷, а в проекте изменились два файла ❸.

Проверка статуса показывает, что текущей является ветвь `main`, а рабочий каталог чист ❹. Это сообщение должно выводиться каждый раз, когда вы фиксируете рабочее состояние своего проекта. Если вы получите другое сообщение, то внимательно прочитайте его; скорее всего, вы забыли добавить файл после фиксации.

Просмотр журнала

Git ведет журнал всех фиксаций, выполненных в проекте. Проверим содержимое журнала:

```
git_practice$ git log
commit ceal3ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main)
Author: eric <eric@example.com>
Date:   Mon Jun 6 19:37:26 2022 -0800
```

```
    Started project.
git_practice$
```

Каждый раз, когда вы фиксируете изменения, Git генерирует уникальный идентификатор из 40 символов. Сохраняется информация о том, кто выполнил изменение, когда, а также передаваемое сообщение. Не вся эта информация вам понадобится, поэтому Git предоставляет возможность вывести упрощенный вариант записи журнала:

```
git_practice$ git log --pretty=oneline
ceal3ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main) Started project.
git_practice$
```

Флаг `--pretty=oneline` выводит два важнейших атрибута: идентификатор фиксации и сохраненное для нее сообщение.

Последующая фиксация

Чтобы увидеть все возможности системы управления версиями, следует внести в проект изменение и зафиксировать его. Добавим еще одну строку в файл `hello_git.py`:

```
hello_git.py
print("Hello Git world!")
print("Hello everyone.")
```

Проверяя статус проекта, мы видим, что Git заметила изменение в файле :

```
git_practice$ git status
❶ On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

❷ modified:   hello_git.py

❸ no changes added to commit (use "git add" and/or "git commit -a")
git_practice$
```

В результатах указываются текущая ветвь ❶, имя измененного файла ❷, а также то, что изменения не были зафиксированы ❸. Закрепим изменения и снова проверим статус:

```
❶ git_practice$ git commit -am "Extended greeting."
[main 945fa13] Extended greeting.
  1 file changed, 1 insertion(+), 1 deletion(-)

❷ git_practice$ git status
On branch main
nothing to commit, working tree clean

❸ git_practice$ git log --pretty=oneline
945fa13af128a266d0114eebb7a3276f7d58ecd2 (HEAD -> main) Extended greeting.
cea13ddc51b885d05a410201a54faf20e0d2e246 Started project.
git_practice$
```

Команда `git commit` с флагом `-am` выполняет новую фиксацию ❶. Благодаря флагу `-a` Git получает указание добавить все измененные файлы в репозитории в текущую фиксацию. (Если вы создали новые файлы между фиксациями, то просто снова выполните команду `git add .`, чтобы добавить новые файлы в репозиторий.) Флаг `-m` дает Git указание сохранить сообщение в журнале.

При проверке статуса проекта рабочий каталог снова оказывается чистым ❷. Наконец, в журнале хранится информация о двух фиксациях ❸.

Откат изменений

А теперь посмотрим, как отменить изменение и вернуться к предыдущему рабочему состоянию. Сначала добавьте в файл `hello_git.py` новую строку:

```
hello_git.py
print("Hello Git world!")
print("Hello everyone.")

print("Oh no, I broke the project!")
```

Сохраните и запустите этот файл.

Из информации статуса видно, что Git видит изменения:

```
git_practice$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
```

```
❶    modified:   hello_git.py
```

```
no changes added to commit (use "git add" and/or "git commit -a")
git_practice$
```

Git определяет, что файл `hello_git.py` был изменен ❶, и при желании мы можем зафиксировать изменения. Но вместо этого мы хотим вернуться к последней фиксации, в которой проект был заведомо работоспособным. С `hello_git.py` ничего делать не нужно — ни удалять строку, ни использовать функцию отмены в текстовом редакторе. Вместо этого в терминальном сеансе следующие команды:

```
git_practice$ git restore .
git_practice$ git status
On branch main
nothing to commit, working tree clean
git_practice$
```

Команда `git restore имя_файла` позволяет откатить все изменения с момента последней фиксации в определенном файле. Команда `git restore .` откатывает все изменения, сделанные во всех файлах с момента последней фиксации; это действие восстанавливает проект до состояния последней фиксации.

Вернувшись в редактор кода, вы увидите, что файл `hello_git.py` возвратился к следующему состоянию:

```
print("Hello Git world!")
print("Hello everyone.")
```

В таком простом проекте возврат к предыдущему состоянию может показаться тривиальным, но если бы мы работали над большим проектом с десятками измененных файлов, то все файлы, измененные с момента последней фиксации, вернулись бы к предыдущему состоянию. Эта возможность невероятно полезна: вы можете вносить любые изменения в ходе реализации новой функции, а если они не работают, то вы просто отменяете их без вреда для проекта. Не нужно запоминать эти изменения и отменять их вручную; Git делает все это за вас.

ПРИМЕЧАНИЕ

Возможно, вам придется обновить файл в редакторе, чтобы увидеть предыдущую версию.

Извлечение предыдущих фиксаций

Вы можете вернуться к любой фиксации в журнале с помощью команды `checkout`, указав первые шесть символов идентификатора ссылки. После проверки и просмотра предыдущей фиксации вы можете вернуться к последней фиксации или отказаться от внесенных изменений и продолжить разработку с предыдущей фиксации:

```
git_practice$ git log --pretty=oneline
945fa13af128a266d0114eebb7a3276f7d58ecd2 (HEAD -> main) Extended greeting.
cea13ddc51b885d05a410201a54faf20e0d2e246 Started project.
git_practice$ git checkout cea13d
Note: switching to 'cea13d'.
```

- ❶ You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-c` with the `switch` command. Example:

```
git switch -c <new-branch-name>
```

- ❷ Or undo this operation with:

```
git switch -
```

Turn off this advice by setting config variable `advice.detachedHead` to `false`

```
HEAD is now at cea13d Started project.
git_practice$
```

При извлечении предыдущей фиксации вы покидаете ветвь `main` и входите в состояние, которое Git называет *отсоединенным состоянием HEAD* (detached HEAD) ❶. HEAD — текущее состояние проекта; *отсоединенным* оно называется потому, что мы покинули именованную ветвь (`main` в данном случае).

Чтобы вернуться к ветви `main`, следуйте предложению ❷, которое позволяет отменить предыдущую операцию:

```
git_practice$ git switch -
Previous HEAD position was cea13d Started project.
Switched to branch 'main'
git_practice$
```

Вы снова возвращаетесь к ветви `main`. Если вы не собираетесь использовать расширенные возможности Git, то лучше не вносить изменения в проект при извлечении старой фиксации. Но если над проектом больше никто не работает и вы хотите откатить все более поздние фиксации и вернуться к предыдущему состоянию,

то можете вернуть проект к предыдущей фиксации. Работая в ветви `main`, введите следующую команду:

```
❶ git_practice$ git status
On branch main
nothing to commit, working directory clean
❷ git_practice$ git log --pretty=oneline
945fa13af128a266d0114eebb7a3276f7d58ecd2 (HEAD -> main) Extended greeting.
cea13ddc51b885d05a410201a54faf20e0d2e246 Started project.
❸ git_practice$ git reset --hard cea13d
HEAD is now at cea13dd Started project.
❹ git_practice$ git status
On branch main
nothing to commit, working directory clean
❺ git_practice$ git log --pretty=oneline
cea13ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main) Started project.
git_practice$
```

Сначала мы проверяем статус и убеждаемся в том, что текущей является ветвь `main` ❶. В журнале есть обе фиксации ❷. Затем выдается команда `git reset --hard` с первыми шестью символами идентификатора той фиксации, к которой нужно вернуться ❸. Далее повторная проверка статуса показывает, что мы находимся в главной ветви, а закреплять нечего ❹. Повторное обращение к журналу показывает, что проект находится в том состоянии, к которому мы решили вернуться ❺.

Удаление репозитория

Иногда история репозитория повреждается, и вы не знаете, как ее восстановить. Если это произойдет, то для начала используйте методы, описанные в приложении В. Если исправить ошибку не удалось, а вы занимаетесь проектом в одиночку, то продолжайте работать с файлами, но сотрите историю проекта, удалив каталог `.git`. Это не повлияет на текущее состояние файлов, но приведет к удалению всех фиксаций, так что вы потеряете возможность извлекать другие состояния проекта.

Для этого либо откройте программу для работы с файлами и удалите репозиторий `.git`, либо сделайте это в командной строке. После этого необходимо создать новый репозиторий, чтобы снова отслеживать изменения в проекте. Вот как выглядит весь процесс в терминальном сеансе:

```
❶ git_practice$ git status
On branch main
nothing to commit, working directory clean
❷ git_practice$ rm -rf .git/
❸ git_practice$ git status
fatal: Not a git repository (or any of the parent directories): .git
❹ git_practice$ git init
Initialized empty Git repository in git_practice/.git/
❺ git_practice$ git status
```

```
On branch main
No commits yet
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    hello_git.py
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

```
❹ git_practice$ git add .
git_practice$ git commit -m "Starting over."
[main (root-commit) 14ed9db] Starting over.
 2 files changed, 5 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 hello_git.py
❺ git_practice$ git status
On branch main
nothing to commit, working tree clean
git_practice$
```

Сначала мы проверяем статус и видим, что рабочий каталог чист ❶. Затем команда `rm -rf .git` (`rmdir /s .git` в Windows) удаляет каталог `.git` ❷. При проверке статуса после удаления каталога `.git` выдается сообщение об отсутствии репозитория Git ❸. Вся информация, используемая Git для отслеживания репозитория, хранится в каталоге `.git`, поэтому его удаление приводит к уничтожению всего репозитория.

Мы можем использовать команду `git init` для создания нового репозитория ❹. Проверка статуса показывает, что все вернулось к исходному состоянию, ожидая первой фиксации ❺. Мы добавляем файлы и выполняем первую фиксацию ❻. Теперь проверка статуса показывает, что проект находится в новой ветви `main`, а закреплять пока нечего ❼.

Работа с системой управления версиями потребует некоторых усилий, но стоит немного освоиться — и вам уже не захочется работать без нее.

Д

Устранение неполадок при развертывании



Развертывание приложения доставляет огромное удовольствие, когда выполняется успешно, особенно в первый раз. Но в процессе может возникнуть множество проблем, и, к сожалению, некоторые из них бывает сложно выявить и устранить. В этом приложении вы узнаете о современных подходах к развертыванию и эффективных способах устранения неполадок в процессе развертывания, когда что-то идет не так, как хочется.

Если дополнительной информации в этом приложении недостаточно для решения проблем, возникающих в процессе развертывания, то обратитесь к ресурсам, указанным на сайте https://ehmatthes.github.io/pcc_3e; их содержимое почти наверняка поможет вам успешно выполнить развертывание.

Основы развертывания приложений

Когда вы пытаетесь устранить неполадки в конкретной ситуации развертывания, полезно иметь четкое представление о том, как устроено типичное развертывание. Под *развертыванием* (deployment) понимается процесс копирования проекта, работающего в вашей локальной системе, на удаленный сервер таким образом, чтобы проект мог обрабатывать запросы любого пользователя в Интернете. Удаленная среда отличается от локальной системы несколькими важными моментами: скорее всего, операционная система (ОС) будет не такой, как у вас, и, по всей видимости, это один из множества виртуальных серверов, развернутых на одном аппаратном сервере.

Когда вы разворачиваете (*копируете*, push) проект на удаленный сервер, необходимо выполнить следующие действия.

- Создать виртуальный сервер на компьютере в центре обработки данных.
- Установить соединение между локальной системой и удаленным сервером.

- Скопировать код проекта на удаленный сервер.
- Определить зависимости проекта и установить их на удаленном сервере.
- Настроить базу данных (БД) и выполнить все необходимые миграции.
- Скопировать статические файлы (CSS, JavaScript и мультимедийные) в место, где их можно будет эффективно обслуживать.
- Запустить сервер для обработки входящих запросов.
- Маршрутизировать входящие запросы в проект, как только он будет готов к их обработке.

Если учесть все, что касается развертывания, то неудивительно, что процесс часто оказывается неудачным. К счастью, поняв его суть, вы с большой долей вероятности сможете определить, что именно пошло не так. И тогда, возможно, сумеете найти решение, которое сделает следующую попытку развертывания успешной.

Вы можете разрабатывать проекты локально в одной ОС, а публиковать на сервере под управлением другой. Важно знать, в какой системе вы разворачиваете проект, поскольку это поможет в устранении неполадок. На момент написания этой книги типичный удаленный сервер на Platform.sh работал под управлением операционной системы Debian Linux; большинство удаленных серверов работают на базе Linux.

Устранение распространенных неполадок

Некоторые шаги по устранению неполадок характерны для той или иной ОС, и мы разберем их чуть позже. Сначала рассмотрим универсальные приемы, которые следует попробовать применить при устранении неполадок в развертывании.

Лучший источник информации о проблеме — вывод, полученный во время попытки развертывания. Эти данные могут напугать неопытных пользователей; они могут выглядеть крайне сложными технически, и их обычно очень много. Не спешите впадать в уныние; вам не нужно понимать все, что написано в выходных данных. При просмотре журнала вы преследуете две цели: определить успешные и неуспешные шаги развертывания. Если справитесь с этой задачей, то сможете понять, что нужно изменить в вашем проекте или процессе развертывания, чтобы следующая попытка увенчалась успехом.

Следуйте инструкциям на экране

Иногда платформа, на которой вы разворачиваете проект, выдает сообщение с четким предложением, как решить проблему. Например, показанное ниже сообщение вы увидите, если создадите проект Platform.sh до инициализации репозитория Git, а затем попытаетесь развернуть проект:

```
$ platform push
```

- ❶ Enter a number to choose a project:
[0] ll_project (votohz445ljyg)
> 0

- ❷ [RootNotFoundException]
Project root not found. This can only be run from inside a project directory.
- ❸ To set the project for this Git repository, run:
platform project:set-remote [id]

Мы пытаемся развернуть проект, но локальный проект еще не связан с удаленным. Поэтому в CLI Platform.sh появляется запрос, какой удаленный проект мы хотим развернуть ❶. Мы указываем 0, чтобы выбрать единственный проект из списка. Но возникает ошибка `RootNotFoundException` ❷. Так происходит потому, что Platform.sh при проверке локального проекта ищет каталог `.git`, чтобы связать локальный проект с удаленным. В данном случае, поскольку при создании удаленного проекта каталога `.git` не существует, соединение так и не было установлено. CLI предлагает с помощью команды `project:set-remote` указать удаленный проект, который должен быть связан с локальным ❸.

Попробуем воспользоваться этим предложением:

```
$ platform project:set-remote votohz4451jyg
Setting the remote project for this repository to: ll_project (votohz4451jyg)
```

```
The remote project for this repository is
now set to: ll_project (votohz4451jyg)
```

В предыдущем выводе CLI был указан идентификатор удаленного проекта, `votohz4451jyg`. Поэтому мы дополняем предложенную команду, указав этот идентификатор, и CLI устанавливает соединение между локальным и удаленным проектами.

Теперь снова попробуем развернуть проект:

```
$ platform push
Are you sure you want to push to the main (production) branch? [Y/n] y
Pushing HEAD to the existing environment main
--nponyск--
```

Развертывание прошло успешно; следование инструкциям на экране помогло.

С осторожностью выполняйте команды, предназначение которых понимаете не до конца. Однако если у вас есть веские причины считать, что команда неdestructивна, и если вы доверяете источнику рекомендаций, то, думаю, разумно воспользоваться предложениями, отображающимися на экране.

ПРИМЕЧАНИЕ

Не забывайте, что существуют злоумышленники, способные посоветовать вам выполнить команды, которые выведут вашу систему из строя или подвергнут ее удаленной эксплуатации. Следование рекомендациям инструмента, предоставленного доверенной компанией или организацией, — это одно, а выполнение рекомендаций случайных людей в Интернете — другое. В любом случае, когда имеете дело с удаленными подключениями, действуйте с осторожностью.

Читайте журнал событий

Как уже говорилось, системный вывод, который отображается при выполнении таких команд, как `platform push`, может быть как информативным, так и поначалу совершенно непонятным. Ознакомьтесь с показанным ниже фрагментом журнала, созданным в одном из случаев выполнения команды `platform push`, и попробуйте выявить проблему:

```
--пропуск--
Collecting soupsieve==2.3.2.post1
  Using cached soupsieve-2.3.2.post1-py3-none-any.whl (37 kB)
Collecting sqlparse==0.4.2
  Using cached sqlparse-0.4.2-py3-none-any.whl (42 kB)
Installing collected packages: platformshconfig, sqlparse,...
Successfully installed Django-4.1 asgiref-3.5.2 beautifulsoup4-4.11.1...
W: ERROR: Could not find a version that satisfies the requirement gunicornn
W: ERROR: No matching distribution found for gunicornn

130 static files copied to '/app/static'.

Executing pre-flight checks...
--пропуск--
```

Если попытка развертывания не удалась, то рекомендуется просмотреть системный журнал и найти в нем нечто похожее на предупреждения или ошибки. Предупреждения — довольно распространенное явление; часто это сообщения о предстоящих изменениях в зависимостях проекта, позволяющие разработчикам решить проблемы до того, как они приведут к реальным сбоям.

Успешный процесс развертывания может сопровождаться предупреждениями, но не ошибками! В приведенном выше случае `Platform.sh` не смог соблюсти требование `gunicornn`. Это опечатка, закрывшаяся в файл `requirements_remote.txt`, которая должна была интерпретироваться как `gunicorn` (с одной буквой `r`). Не всегда легко обнаружить источник проблемы в журнале, особенно если проблема вызывает каскад ошибок и предупреждений. Как и при чтении трассировки в локальной системе, стоит внимательно изучить первые несколько ошибок в списке, а также последние. Большинство ошибок между ними — это конфликты внутренних пакетов из-за нештатных ситуаций и передача сообщений об ошибке другим внутренним пакетам. Фактическая ошибка, которую мы можем исправить, обычно значится одной из первых или последних в списке.

Иногда вы сможете обнаружить ошибку, а иногда не будете иметь ни малейшего представления, что означает вывод. Попробовать разобраться в нем, конечно, стоит, и анализ журнала с последующей успешной диагностикой проблемы приносит огромное удовлетворение. Проводя больше времени за просмотром журнала, вы сможете лучше выявлять наиболее значимую для вас информацию.

Устранение неполадок, специфичных для ОС

Вы можете разрабатывать приложения в любой операционной системе, которая вам по душе, и разворачивать на серверах любых хостинг-провайдеров. Инструменты для разворачивания проектов развиты настолько, что позволяют корректировать ваши проекты для успешной работы в любой удаленной системе. Однако могут возникнуть некоторые проблемы, связанные с той или иной ОС.

В процессе развертывания на платформе Platform.sh одним из наиболее вероятных источников проблем служит установка CLI. Взгляните на следующую команду:

```
$ curl -fsS https://platform.sh/cli/installer | php
```

Команда начинается с имени `curl`, инструмента, позволяющего запрашивать в терминале удаленные ресурсы, доступ к которым осуществляется с помощью URL. В данном случае этот инструмент используется для скачивания программы установки CLI с сервера Platform.sh. Сочетание символов `-fsS` в команде представляет собой набор флагов, которые изменяют работу инструмента `curl`. Флаг `f` дает `curl` указание подавлять большинство сообщений об ошибках, чтобы программа установки CLI могла обрабатывать их самостоятельно, а не сообщать вам обо всех. Флаг `s` определяет параметр тихого запуска в `curl` и позволяет программе установки CLI решать, какую информацию выводить в терминале. Флаг `S` дает `curl` указание выводить сообщение об ошибке, если команда в целом не была выполнена. Благодаря флагу `| php` в конце команды ваша система получает указание запустить скачанный файл установщика с помощью интерпретатора PHP, поскольку инструмент CLI Platform.sh написан на PHP.

Таким образом, для установки CLI Platform.sh в вашей системе необходимы инструменты `curl` и PHP. Чтобы использовать CLI, вам также понадобятся Git и терминал, в котором можно выполнять команды *Bash*. Это язык, доступный в большинстве серверных сред. В большинстве современных систем достаточно пространства для установки нескольких подобных инструментов.

Материал следующих подразделов поможет вам разобраться с требованиями для вашей ОС. Если у вас еще не установлен Git, то ознакомьтесь с инструкциями по работе с ним, изложенными в приложении Г, а затем перейдите к подразделу в этом приложении, соответствующему вашей операционной системе.

ПРИМЕЧАНИЕ

Отличный ресурс для изучения терминальных команд, подобных показанной выше, расположен на <https://explainshell.com>. Укажите изучаемую команду — и сайт отобразит документацию по всем ее компонентам. Попробуйте найти информацию по команде, используемой для установки CLI Platform.sh.

Развертывание из Windows

В последние годы операционная система Windows вновь обрела популярность среди программистов. В нее интегрировано множество различных компонентов для других операционных систем, благодаря чему пользователи получили ряд возможностей для локальной разработки и взаимодействия с удаленными системами.

Одна из наиболее серьезных трудностей при развертывании из Windows заключается в том, что базовая операционная система Windows не похожа на систему Linux, которая используется на удаленном сервере. В базовой системе Windows другой набор инструментов и языков, поэтому для развертывания проектов из нее нужно выбрать способ интеграции Linux-инструментов в локальную среду.

Подсистема Windows для Linux

Один из популярных подходов — использование *подсистемы Windows для Linux* (Windows Subsystem for Linux, WSL), среды, которая позволяет запускать Linux непосредственно в Windows. Если у вас настроена подсистема WSL, то использовать CLI Platform.sh в Windows будет так же просто, как и в Linux. CLI не сможет определить, что запущен в Windows; он обнаружит только среду Linux, в которой вы его используете.

Настройка WSL состоит из двух этапов: сначала вы устанавливаете подсистему WSL, а затем выбираете дистрибутив Linux для развертывания в среде WSL. Тема настройки среды WSL выходит за рамки этой книги; если она вам интересна, то обратитесь к документации по адресу <https://learn.microsoft.com/ru-ru/windows/wsl/about>. После установки WSL вы можете следовать инструкциям для операционной системы Linux, изложенным в этом приложении, чтобы продолжить развертывание.

Git Bash

Другой подход к формированию локального окружения, из которого можно развернуть систему, касается *Git Bash*, терминальной среды, совместимой с Bash, но работающей в Windows. Git Bash устанавливается вместе с Git, с помощью дистрибутива, доступного на сайте <https://git-scm.com>. Этот подход вполне рабочий, но не так удобен, как в случае с WSL. Для выполнения некоторых шагов вам придется использовать терминал Windows, а для других — терминал Git Bash.

Сначала вам нужно установить PHP. Вы можете сделать это с помощью *ХАМРР*, пакета, который, помимо PHP, содержит несколько других инструментов для разработчиков. Чтобы скачать ХАМРР для Windows, перейдите по адресу <https://apachefriends.org>. Запустите программу установки и, если увидите предупреждение об ограничениях контроля учетных записей пользователей (УАК), нажмите кнопку ОК. Примите все рекомендации мастера установки по умолчанию и установите программу.

После завершения установки необходимо добавить PHP в системное окружение (Path); так операционная система Windows будет знать, где искать PHP при его запуске. В строке поиска меню **Start** (Пуск) введите `path` (в русской версии — *переменн. — Примеч. пер.*) и выберите пункт **Edit the System Environment Variables** (Изменение системных переменных среды); в появившемся диалоговом окне нажмите кнопку **Environment Variables** (Переменные среды). Выделите переменную `Path`; нажмите кнопку **Edit** (Изменить). В появившемся диалоговом окне нажмите кнопку **New** (Создать), чтобы добавить новый путь в список. Если вы использовали путь по умолчанию в процессе установки программы XAMPP, то добавьте строку `C:\xampp\php` в появившемся поле, затем нажмите кнопку **OK**. Теперь закройте все системные диалоговые окна, которые были открыты.

Выполнив эти требования, вы можете установить CLI Platform.sh. Для этого вам понадобится терминал Windows с правами администратора; введите команду `cmd` в строке поиска меню **Start** (Пуск) и, щелкнув на пункте **Command Prompt** (Командная строка) правой кнопкой мыши, выберите пункт **Run as administrator** (Запуск от имени администратора) в контекстном меню. В открывшемся терминале введите следующую команду:

```
> curl -fsS https://platform.sh/cli/installer | php
```

Так вы сможете установить CLI Platform.sh, как было описано ранее.

Итак, вы можете работать в Git Bash. Чтобы открыть его терминал, в меню **Start** (Пуск) найдите приложение Git Bash и щелкните на его ярлыке кнопкой мыши; у вас должно открыться окно терминала. В этом терминале вы можете использовать команды, традиционные как для Linux (например, `ls`), так и для Windows (к примеру, `dir`). Чтобы убедиться в успешности установки, выполните команду `platform list`. Вы должны увидеть список всех команд CLI Platform.sh. С этого момента все операции по развертыванию выполняйте с помощью CLI Platform.sh в окне терминала Git Bash.

Развертывание из macOS

Операционная система macOS отличается от Linux, но обе системы были разработаны на схожих алгоритмах Unix. Благодаря этому многие команды и рабочие процессы, которые вы используете в macOS, будут работать и в удаленной серверной среде. Возможно, вам потребуется установить дополнительные компоненты для разработчиков, чтобы все необходимые инструменты были доступны в вашей локальной среде macOS. Если в процессе работы вам будет предложено установить *инструменты разработчика командной строки* (command line developer tools), то нажмите кнопку **Install** (Установить), чтобы подтвердить установку.

Наиболее распространенная проблема при установке CLI Platform.sh заключается в отсутствии в системе PHP. Если вы видите сообщение о том, что команда `php` не найдена, то вам нужно установить PHP. Один из самых простых способов сделать

это — использовать менеджер пакетов *Homebrew*, который упрощает установку множества пакетов, требующихся программистам. Если у вас еще не установлена программа Homebrew, то посетите сайт <https://brew.sh> и следуйте инструкциям по ее установке.

После установки Homebrew используйте следующую команду для установки PHP:

```
$ brew install php
```

Через некоторое время, когда установка завершится, вы сможете успешно установить CLI Platform.sh.

Развертывание из Linux

Поскольку большинство серверных сред функционируют под управлением операционной системы Linux, у вас не должно возникнуть особых проблем с установкой и использованием Platform.sh CLI. В процессе установки CLI в свежеразвернутой системе типа Ubuntu вы увидите уведомления о том, какие именно пакеты вам нужны:

```
$ curl -fsS https://platform.sh/cli/installer | php
Command 'curl' not found, but can be installed with:
sudo apt install curl
Command 'php' not found, but can be installed with:
sudo apt install php-cli
```

В вашем случае будет больше информации о других пакетах, которые нужно установить, а также данные о версии. Показанная ниже команда установит `curl` и PHP:

```
$ sudo apt install curl php-cli
```

После этого команда установки CLI Platform.sh должна завершиться успешно. Поскольку ваша локальная среда очень похожа на большинство Linux-сред хостинг-провайдеров, многие моменты, касающиеся работы в терминале, будут идентичны и в удаленной среде.

Другие подходы к развертыванию

Если платформа Platform.sh по каким-то причинам вам не подходит или вы хотите попробовать другой подход, то существует множество хостинговых компаний, из которых можно выбрать наиболее подходящую. Одни хостинг-платформы функционируют идентично процессу, описанному в главе 20, а другие используют совершенно иной подход, описанный в начале этого приложения.

- Система Platform.sh позволяет выполнять шаги в браузере вместо CLI. Если вам больше нравится графический интерфейс, чем терминальный, то вы можете выбрать этот подход.

- Некоторые хостинг-провайдеры предлагают взаимодействие как через CLI, так и через браузер. В ряде случаев в браузере запускается специальный терминал, так что вам не нужно ничего устанавливать на свой компьютер.
- Некоторые провайдеры позволяют размещать проекты на сайтах типа GitHub, а затем подключать репозитории GitHub к хостингу. После этого провайдер разворачивает ваш код из GitHub, а не требует от вас выгрузить код из локальной системы на сервер. На сайте Platform.sh тоже поддерживается такая схема работы.
- Некоторые провайдеры предлагают целый набор услуг, из которых можно сконструировать инфраструктуру, подходящую именно для вашего проекта. В подобных случаях, как правило, от пользователя требуются глубокое понимание процесса развертывания и знание необходимых инструментов для удаленного обслуживания проекта. К таким провайдерам относятся Amazon Web Services (AWS) и Microsoft Azure. Отслеживать расходы на таких платформах гораздо сложнее, поскольку каждая услуга оплачивается отдельно.
- Многие пользователи разворачивают свои проекты на виртуальных частных серверах (virtual private server, VPS). При таком подходе вы арендуете виртуальный сервер, который действует как удаленный компьютер, авторизуетесь на нем, устанавливаете ПО, необходимое для работы вашего проекта, разворачиваете код, устанавливаете требуемые соединения и разрешаете серверу начать принимать запросы.

Регулярно появляются новые хостинговые платформы и способы взаимодействия с ними; выберите наиболее подходящие и потратьте время на изучение соответствующего процесса развертывания проектов. Длительное время поддерживайте свой проект, чтобы разобраться, какой аспект, опирающийся на подход вашего провайдера, эффективен, а какой — нет. Нет идеальных хостинг-провайдеров; вам придется постоянно оценивать, подходит ли выбранный провайдер под ваш проект.

Напоследок я хотел бы предостеречь вас от необдуманного выбора платформы хостинга и поиска некоего универсального подхода к развертыванию. Другие пользователи могут с энтузиазмом склонять вас к использованию сложных подходов к развертыванию и сервисов, которые призваны сделать ваш проект высоконадежным и способным обслуживать миллионы пользователей одновременно. Многие программисты тратят много времени, денег и сил на выработку сложной стратегии развертывания, а потом обнаруживают, что их проектом почти никто не пользуется. Большинство проектов Django можно разместить на небольшом хостинге и настроить на обслуживание тысяч запросов в минуту. Если трафик вашего проекта укладывается в этот уровень, то потратьте время на конфигурацию развертывания для работы на минимальной платформе, прежде чем вкладывать деньги в инфраструктуру, предназначенную для крупнейших сайтов в мире.

Порой развертывание — невероятно сложная задача, но констатация факта, что ваш проект работает хорошо, вдохновляет. Наслаждайтесь решением этой сложной задачи и обращайтесь за помощью по мере необходимости.

Эрик Мэтиз

**Изучаем Python: программирование игр,
визуализация данных, веб-приложения**

3-е издание, дополненное и переработанное

Перевели с английского Е. Матвеев, С. Черников

Руководитель дивизиона
Ведущий редактор
Литературный редактор
Художественный редактор
Корректоры
Верстка

*Ю. Сергиенко
Н. Гринчик
Н. Хлебина
В. Мостипан
О. Андриевич, Т. Никифорова
Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 08.10.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 45,150. Тираж 3000. Заказ 0000.