Содержание

От издательства 12

Об авторе 13

О технических рецензентах 14

Благодарности 16

Введение 17

Глава 1. Прототипирование и среды разработки 22

Прототипирование в Python 22
Прототипирование с помощью REPL 23
Прототипирование с помощью Python-скрипта 26
Прототипирование с помощью скриптов и pdb 27
Посмертная отладка 28
Прототипирование с помощью Jupyter 30
Блокноты 31
Прототипирование в этой главе 33
Подготовка окружения 34
Подготовка нового проекта 35
Прототипирование скриптов 37
Установка зависимостей 40
Экспорт в py-файл 42
Построение интерфейса командной строки 44
Модуль sys и переменная argv 45
argparse 47
click 48
Расширение границ возможного 51
Удаленные ядра 51
Разработка кода, который невозможно выполнить локально 54
Окончательный скрипт 58
Резюме 59
Дополнительные ресурсы 59

Глава 2. Тестирование, проверка типов, стандарты кодирования 61

Тестирование 64
Когда писать тесты 66
Создание функций форматирования для повышения тестопригодности 67

pytest 70
Автономное, интеграционное и функциональное тестирование 71
Фикстуры Pytest 74
Покрытие 78
Проверка типов 82
Установка mypy 83
Добавление аннотаций типов 83
Подклассы и наследование 86
Обобщенные типы 88
Отладка и чрезмерное увлечение типизацией 90
Когда прибегать к типизации, а когда избегать ее 92
Хранение аннотаций типов отдельно от кода 93
Стандарты кодирования 95
Установка flake8 и black 96
Исправление существующего кода 96
Автоматический прогон 98
Применение к запросам на включение изменений 99
Резюме 100
Дополнительные ресурсы 102

Глава 3. Скрипты для создания пакетов 103

Терминология 104
Структура каталога 105
Скрипты настройки и метаданные 107
Зависимости 108
Декларативные конфигурации 109
Чего избегать в файле setup.py 110
Условные зависимости 110
Файл Readme в метаданных 112
Номера версий 114
Использование файла setup.cfg 115
Специальные серверы каталогов 117
Настройка pypiserver 118
Устойчивость к сбоям 119
Конфиденциальность 120
Целостность 121
Формат wheel и выполнение кода при установке 122
Создание wheel-файлов по существующим дистрибутивам 123
Установка консольного скрипта с помощью точек входа 125
Файлы README, DEVELOP и CHANGES 126
Формат Markdown 127
Формат reStructured 128
Файл README 130
Файл CHANGES.md и номера версий 131
Семантическое версионирование 131
Календарное версионирование 132
Закрепление версий зависимостей 132

Слабое закрепление ... 133
Строгое закрепление ... 133
Какую схему закрепления использовать ... 134
Загрузка версии на сервер ... 135
Конфигурирование twine ... 136
Резюме ... 137
Дополнительные ресурсы ... 137

Глава 4. От скрипта к каркасу ... 139

Написание плагина датчика ... 140
Разработка плагина ... 141
Добавление нового параметра командной строки ... 144
Подкоманды ... 144
Опции командной строки ... 147
Обработка ошибок ... 148
Делегирование разбора аргументов Click ... 151
Поддержка Click пользовательских типов аргументов ... 152
Встроенные параметры ... 154
Разрешение сторонних плагинов датчиков ... 155
Обнаружение плагинов по фиксированным именам ... 157
Обнаружение плагинов с помощью точек входа ... 158
Конфигурационные файлы ... 161
Переменные окружения ... 164
Сравнение apd.sensors с похожими программами ... 165
Резюме ... 166
Дополнительные ресурсы ... 167

Глава 5. Альтернативные интерфейсы ... 168

Веб-микросервисы ... 168
WSGI ... 169
Проектирование API ... 174
Аутентификация ... 176
Flask ... 176
Декораторы в Python ... 179
Замыкания ... 180
Модификация переменных в родительских областях видимости ... 181
Простые декораторы ... 183
Декораторы с аргументами ... 184
Безопасность на основе декораторов ... 186
Тестирование функции представления ... 190
Развертывание ... 192
Расширение программного обеспечения третьей стороной ... 194
Согласование ситуативной сигнатуры с равноправными пользователями ... 199
Абстрактные базовые классы ... 201
Запасные стратегии ... 204

Паттерн Адаптер 205
Динамическое генерирование класса 206
Другие форматы сериализации 207
Собираем все вместе 209
Исправление ошибки сериализации в нашем коде 211
Наведение порядка 213
Версионирование API 214
Тестопригодность 216
Резюме 217
Дополнительные ресурсы 218

Глава 6. Процесс агрегирования 219

Cookiecutter 219
Создание нового шаблона 220
Создание пакета агрегирования 223
Типы баз данных 224
Наш пример 227
Объектно-реляционные отображения 228
Версионирование базы данных 232
Другие полезные команды alembic 236
Загрузка данных 237
Новые технологии 244
Базы данных 244
Поведение пользовательских атрибутов 244
Генераторы 244
Резюме 245
Дополнительные ресурсы 245

Глава 7. Распараллеливание и асинхронное программирование 246

Неблокирующий ввод-вывод 247
Делаем код неблокирующим 251
Многопоточная и многопроцессная обработка 253
Низкоуровневые потоки 253
Байт-код 257
GIL 258
Блокировки и взаимоблокировки 260
Взаимоблокировки 262
Избегайте глобального состояния 265
Объединение данных 265
Передача данных 266
Другие примитивы синхронизации 269
Реентерабельные блокировки 270
Условия 270
Барьеры 273
Событие 274

Семафор ... 275
Объекты ProcessPoolExecutor ... 276
Делаем нашу программу многопоточной ... 277
Асинхронный ввод-вывод ... 278
async def ... 278
await ... 279
async for ... 281
async with ... 285
Асинхронные примитивы блокировки ... 286
Работа совместно с синхронными библиотеками ... 287
Делаем программу асинхронной ... 289
Сравнение ... 292
Как сделать выбор ... 293
Резюме ... 295
Дополнительные ресурсы ... 295

Глава 8. Дополнительные вопросы асинхронного ввода-вывода ... 296

Тестирование асинхронного кода ... 296
Тестирование нашей программы ... 298
Тестовые серверы и фикстуры pytest с очисткой ... 298
Область видимости фикстур ... 302
Использование подставных объектов для упрощения автономного тестирования ... 305
Подставные объекты с ветвящейся логикой ... 308
Классы данных ... 309
Тестовые методы ... 312
Асинхронная работа с базами данных ... 314
Классический стиль SQLAlchemy ... 315
Неоткомпилированная ... 316
mssql ... 316
mysql ... 316
Postgresql ... 317
sqlite ... 317
Использование метода run_in_executor ... 318
Запрос данных ... 320
Избегайте сложных запросов ... 322
Запросы к представлениям ... 329
Альтернативы ... 332
Глобальные переменные в асинхронном коде ... 333
Резюме ... 335
Дополнительные ресурсы ... 336

Глава 9. Просмотр данных ... 337

Функции запроса ... 337
Фильтрация данных ... 343

Многоуровневые итераторы ... 345
Дополнительные фильтры ... 351
Тестирование функций запроса ... 352
Параметрические тесты ... 354
Отображение нескольких датчиков ... 355
Обработка данных ... 359
Интерактивная работа с виджетами Jupyter ... 363
Глубоко вложенный синхронный и асинхронный коды ... 364
Наведем порядок ... 369
Сохранение оконечных точек ... 370
Нанесение географических данных на карты ... 371
Новые типы графиков ... 373
Поддержка карт в пакете apd.aggregation ... 375
Обратная совместимость в классах данных ... 376
Построение карты с применением новых конфигурационных объектов ... 378
Резюме ... 380
Дополнительные ресурсы ... 381

Глава 10. Повышение быстродействия ... 382

Оптимизация функции ... 382
Профилирование и потоки ... 384
Интерпретация отчета профилировщика ... 387
Другие профилировщики ... 389
timeit ... 389
line_profiler ... 390
yappi ... 390
Tracemalloc ... 393
New Relic ... 394
Оптимизация потока управления ... 395
Сложность ... 395
Визуализация данных профилирования ... 399
Кеширование ... 402
Кешированные свойства ... 409
Резюме ... 411
Дополнительные ресурсы ... 411

Глава 11. Отказоустойчивость ... 413

Обработка ошибок ... 413
Получение элементов из контейнера ... 414
Абстрактные базовые классы ... 414
Типы исключений ... 417
Пользовательские исключения ... 419
Создание новых типов исключений ... 420
Дополнительные метаданные ... 422
Трасса вызовов при наличии нескольких исключений ... 423

Исключение в блоке except или finally ... 424
raise from ... 425
Тестирование обработки исключений ... 427
Новые поведения ... 427
Еще о подставных объектах и unittest.Mock ... 430
Предупреждения ... 432
Фильтры предупреждений ... 435
Протоколирование ... 437
Вложенные регистраторы ... 438
Пользовательские действия ... 439
Дополнительные метаданные ... 440
Конфигурация протоколирования ... 445
Другие обработчики ... 446
Контрольные журналы ... 446
Избегание проблем на этапе проектирования ... 447
Опрос датчиков по расписанию ... 448
API и фильтрация ... 451
Резюме ... 452
Дополнительные ресурсы ... 453

Глава 12. Обратные вызовы и анализ данных ... 454
Поток данных генератора ... 454
Генераторы, потребляющие свой собственный выход ... 456
Улучшенные генераторы ... 459
Использование классов ... 462
Использование улучшенного генератора для обертывания итерируемого объекта ... 463
Рефакторинг функций, возвращающих излишние значения ... 464
Очереди ... 466
Выбор потока управления ... 468
Конструкция для наших действий ... 469
Сопрограммы для анализа ... 470
Подача данных ... 475
Выполнение процесса анализа ... 478
Состояния процесса ... 480
Обратные вызовы ... 483
Расширение состава имеющихся действий ... 485
Резюме ... 488
Дополнительные ресурсы ... 488

Эпилог ... 490

Предметный указатель ... 492

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Мэттью Уилкс – разработчик программного обеспечения из Европы, работает на Python в течение последних 15 лет. Также имеет богатый опыт обучения Python-разработчиков на платных курсах.

Принимает активное участие в проектах с открытым исходным кодом, внес вклад во многие популярные системы. В этом отношении его интересуют прежде всего детали взаимодействия с базами данных и вопросы безопасности в веб-каркасах.

О технических рецензентах

Коэн де Гроот – программист-фрилансер и преподаватель Python. Одержим компьютерами и программированием с конца 1970-х годов, когда собрал свой первый «компьютер».

Едва защитив диплом по информатике в Лейденском университете, Коэн начал работать – в крупной нефтяной компании, в небольших стартапах, в компаниях, разрабатывающих ПО на заказ, и т. д. Написал кучу программ на разных языках. Занимался технической поддержкой, преподавал, возглавлял группы и руководил техническими проектами.

Отдав 20 лет жизни ИТ, Коэн решил попробовать себя на другом поприще, работал бизнес-тренером, основал большое сообщество тренеров и организовал пять конференций. Но вскоре вернулся к разработке сайтов и других сервисов для тренеров и не только.

Последние 10 лет Коэн занимается в основном программированием на Python и попутно на SQL, JavaScript и т. д. По-прежнему получает удовольствие от изучения возможностей Python и от передачи знаний другим – в личном общении, с помощью печатных текстов или видео.

Нейц Зупан стал компьютерным фанатом, едва научившись ходить, свою первую игру написал еще в начальной школе, в средней школе стал победителем национального чемпионата по робототехнике, а будучи студентом колледжа, стал сооснователем сайта niteo.co. Выступал на конференциях на пяти континентах, в основном на темы, связанные с вебом, Python и продуктивностью. Когда не пишет программы, гоняется за большими волнами по всему миру.

Джесси Снайдер начал программировать спустя много лет после того, как забросил изучение музыкальной фольклористики, и был приятно удивлен захватывающими задачами и тем, какое удовольствие доставляло ему проектирование программ. Несколько лет подвизался в некоммерческих технологических организациях на Тихоокенском Северо-Западе, а ныне является независимым консультантом. Если не занят работой и не играет в яванском гамелане, то совершает длинные пробежки по красивым паркам в окрестности своего дома в Сиэттле, штат Вашингтон.

Благодарности

Многие люди так или иначе способствовали появлению этой книги на свет. Прежде всего следует упомянуть тех, кто создавал экосистему Python с открытым исходным кодом, без них просто не о чем было бы писать. Спасибо Джоанне, которая воодушевляла меня, презрев трудности и долгие часы, отданные работе. Спасибо и всей остальной семье за не ослабевающую с годами поддержку.

Что касается конкретно этой книги, то я благодарен Нейцу Зупану (Nejc Zupan), Джесси Снайдеру (Jesse Snyder), Тому Блокли (Tom Blockley), Алану Хоуи (Alan Hoey) и Крису Эвингу (Cris Ewing) – все они поделились ценными замечаниями о плане и результате его осуществления. Также спасибо Марку Уилрайту (Mark Wheelwright) из компании ISO Photography за отличные фотографии меня и всей команды Apress.

Наконец, я хочу поблагодарить всех, чьими усилиями веб остается такой же фантастической и чудесной вещью, какой он был, когда я впервые увлекся интернетом. Томан Хисман-Хаунт (Thomas Heasman-Hunt), Джулия Эванс (Julia Evans), Ян Фигген (Ian Fieggen), Фооне Тьюринг (Foone Turing) и бесчетное множество других – сомневаюсь, что индустрия программного обеспечения заинтересовала бы меня так сильно, не будь таких людей, как вы.

Введение

Python – весьма успешный язык программирования. За тридцать лет своего существования он получил чрезвычайно широкое распространение. Он по умолчанию включен в основные операционные системы, некоторые крупнейшие мировые сайты используют Python на стороне сервера, а ученые применяют Python в повседневной работе для пополнения копилки коллективных знаний. А раз так много людей разрабатывают и используют Python, улучшения идут сплошным потоком. Не у всех Python-разработчиков есть возможность посещать конференции и следить за тем, что происходит в других частях сообщества, поэтому некоторые возможности языка и экосистемы в целом известны не так хорошо, как того заслуживают.

Цель этой книги – исследовать те части языка и инструментария Python, о которых, возможно, не все знают. Если вы – опытный разработчик, то, наверное, многие из них вам знакомы, но еще больше ждут, пока у вас появится время на их изучение. Особенно это верно в том случае, когда вы работаете над сложившимися системами, в которых изменение архитектуры компонента ради того, чтобы воспользоваться новыми возможностями языка, – дело не частое.

Если вы работаете с Python сравнительно недолго, то, вероятно, знакомы с недавними добавлениями в язык, но в меньшей степени с некоторыми библиотеками, входящими в экосистему. Посещение различных мероприятий, в т. ч. конференций по Python, хорошо тем, что дает шанс узнать о небольших, но весьма полезных усовершенствованиях, придуманных коллегами-программистами, и включить их в свой арсенал.

Эта книга – не справочник, в котором каждому языковому средству посвящен отдельный раздел; порядок изложения продиктован тем, как создается реальная программа.

В технической документации имеется тенденция ограничиваться простыми примерами. Простые примеры хороши, когда нужно объяснить, как нечто работает, но если хочется понять, когда это стоит использовать, то они уже не так полезны. На таком фундаменте трудно возвести что-то солидное, потому что архитектуры сложного и простого кода сильно различаются.

Взяв за основу один сквозной пример, мы сможем рассмотреть технологические альтернативы в контексте. Вы узнаете, какие соображения следует иметь в виду при выборе того или иного подхода. Совместно обсуждаются темы, связанные общностью использования, а не схожестью принципов работы.

Об этой книге

При написании этой книги я ставил целью поделиться знаниями из различных частей экосистемы и уроками, усвоенными за 15 лет программирования на Python для добывания средств к существованию. Книга поможет вам по-

высить свою продуктивность при использовании как самого языка, так и дополнительных библиотек. Вы научитесь эффективно использовать языковые средства, которые, строго говоря, необязательны, но полезны программисту, желающему работать продуктивно: асинхронное программирование, создание пакетов, тестирование и т. п.

Однако книга ориентирована на тех, кто хочет писать код, а не стремится познать скрытую за ним магию. Я не стану слишком глубоко вдаваться в вопросы, затрагивающие детали реализации Python. Чтобы получить пользу от этой книги, вам не придется грокать[1] написанные на C расширения Python, метаклассы или алгоритмы.

Содержательные примеры кода пронумерованы, а на сопроводительном сайте книги те же листинги представлены в электронном виде. Иногда результат работы приводится прямо под листингом, а не на нумерованном рисунке.

На сопроводительном сайте вы найдете полный код примера, разбитый по главам, а также вспомогательный код упражнений. В общем, я рекомендую следить за кодом, выгружая части, относящиеся к текущей главе, из Git-репозитория на сайте книги или из дистрибутивного пакета.

Помимо листингов, я привожу распечатки консольных сеансов. Если фрагмент кода содержит строки, начинающиеся знаком >, значит, это сеанс работы в оболочке. Предполагается, что эти команды выполняются в окне терминала операционной системы. Если же строка начинается знаками >>>, то это сеанс в консоли Python, т. е. команды должны вводиться в интерпретаторе Python.

О ПРИМЕРЕ

В качестве примера мы будем рассматривать универсальный агрегатор данных. Если вы занимаетесь DevOps, то, скорее всего, используете такого рода программу, чтобы отслеживать потребление ресурсов серверами. Если же вы веб-разработчик, то, возможно, используете нечто подобное для сбора статистики из разных точек развертывания одной и той же системы. Ученые тоже пользуются похожими методами, например, для сбора данных с датчиков качества воздуха, установленных в городе. Не всякому разработчику приходится создавать такие программы, но постановка задачи знакома многим разработчикам.

Этот пример выбран не потому, что задача типичная, а потому, что позволит изучить многие интересующие нас предметы естественным унифицированным образом. Выполнить код можно на любом современном компьютере с любой современной операционной системой[2], докупать дополнительное

---

[1] Жаргонное словечко, ставшее популярным в 1960-х годах, когда знания о компьютерах были распространены не так широко. Грокать – значит понимать что-то на очень глубоком и интуитивном уровне. Придумано Робертом Хайнлайном в романе «Чужак в чужой стране».

[2] Впрочем, если вы работаете с Windows, то я рекомендую взять что-то типа Windows Subsystem for Linux, потому что большинство дополнительных библиотек пишутся в расчете на Linux или macOS, поэтому лучше работают в среде WSL.

оборудование не придется. Для некоторых примеров стоит использовать дополнительные компьютеры, играющие роль удаленных источников данных.

В примерах будет использоваться одноплатный компьютер Raspberry Pi Zero с доустановленными датчиками. Эту платформу легко купить примерно за 5 долларов и собирать на ней разные интересные данные. Во многих магазинах, торгующих Raspberry Pi, можно приобрести дополнительные датчики для нее.

Хотя я буду рекомендовать вещи, специфичные для Raspberry Pi, чтобы упростить примеры, эта книга не об интернете вещей и не о самой Raspberry Pi. Это просто средство для достижения цели; если хотите, адаптируйте примеры к задачам, которые вас больше интересуют. Для решения любой похожей задачи следует использовать такой же процесс проектирования.

О выборе тем

Темы подобраны так, чтобы пролить свет на разнообразные аспекты программирования на Python. Все они посвящены средствам, которые незаслуженно мало используются или недостаточно хорошо поняты сообществом Python в целом. Ни одна тема не предназначена для включения в курс для начинающих. Это не значит, что материал труден или сложен для понимания (хотя и такое, безусловно, встречается), просто я выбрал средства, с которыми, на мой взгляд, должны быть знакомы все программисты на Python, даже те, кто их не использует.

Глава 1 – введение в разные способы написания очень простых программ на Python, в частности рассматриваются Jupyter-блокноты и основы использования отладчика Python. То и другое – хорошо известные инструменты, но многие поднаторели в использовании только одного из них, но не обоих сразу. Также обсуждаются подходы к написанию интерфейсов командной строки и некоторые сторонние библиотеки, помогающие делать это лаконично.

В главе 2 рассматриваются инструменты, помогающие находить ошибки в коде, в т. ч. средства автоматизированного тестирования и статического анализа. Все они упрощают написание кода, в правильности которого вы можете быть уверены, будь то большая кодовая база, которую редко приходится изменять, или произведения сторонних авторов. Все рассматриваемые инструменты относятся к числу рекомендуемых мной, а упор делается на сравнительный анализ их достоинств и недостатков. Возможно, некоторыми из них вы уже пользовались, не исключено, у вас есть свое мнение об их пригодности. Эта глава поможет вам понять компромиссы и принять обоснованное решение.

В главе 3 рассматриваются пакеты и управление зависимостями в Python. Это очень важно при написании приложений, предназначенных для распространения, и при проектировании надежного механизма развертывания. Мы воспользуемся этими средствами для преобразования автономного скрипта в допускающее установку приложение.

В главе 4 мы познакомимся с архитектурами плагинов. Это очень мощное средство; часто бывает, что изучивший их программист пытается встав-

лять плагины повсюду, поэтому некоторые преподаватели относятся к ним с опаской. Но в нашем примере использование плагинов естественно. Мы также рассмотрим некоторые специальные приемы работы с командными инструментами, упрощающие отладку систем с плагинами.

В главе 5 обсуждаются веб-интерфейсы, а также использование декораторов и замыканий для написания сложных функций. Эти приемы являются идиоматическими в Python, но с трудом выражаются на многих других языках. Рассматривается также вопрос об использовании абстрактных базовых классов (АБК, англ. ABC). Часто можно встретить мнение, что АБК использовать не стоит, поскольку некоторые, узнав про них, суют их куда ни попадя. Но при определенных условиях у АБК имеются несомненные достоинства, особенно если они сочетаются с инструментами, описанными в главе 2.

В главе 6 мы дополним пример еще одним существенным компонентом – сервером агрегирования, который собирает данные. Здесь же демонстрируются некоторые из наиболее полезных сторонних библиотек, применяемых программистами Python, например requests.

Глава 7 посвящена многопоточному и асинхронному программированию на Python. Многопоточность часто оказывается источником тонких ошибок. Асинхронный код можно использовать для решения похожих задач, но многие разработчики пренебрегают этой идиомой, потому что поведение асинхронной программы резко отличается от поведения синхронной. В этой главе предметом нашего внимания станет использование конкурентности в реальной программе, а не демонстрация на простом примере и не объяснение пределов применимости асинхронного программирования. Цель – представить работающий код, который можно вставить в настоящую программу, и детально разобраться во всех компромиссах, а не просто продемонстрировать технологию в отрыве от реальности.

В главе 8 мы продолжим изучение асинхронного программирования и добавим тестирование асинхронного кода, а также различные имеющиеся библиотеки для написания кода, работающего с внешними источниками (например, базами данных) асинхронно. Кроме того, мы кратко рассмотрим продвинутые методы написания хороших API, полезных при асинхронном программировании, в частности контекстные менеджеры и контекстные переменные.

В главе 9 мы вернемся к Jupyter и его средствам визуализации данных и простого взаимодействия с пользователем. Мы также посмотрим, как использовать наши асинхронные функции с виджетами в Jupyter-блокнотах, и поговорим о продвинутом использовании итераторов и способах реализации сложных типов данных.

Глава 10 посвящена ускорению Python-кода с помощью различных типов кеширования и вопросу о том, в каких случаях этим стоит пользоваться. Здесь же мы рассмотрим тестирование производительности отдельных функций приложения и обсудим, как интерпретировать результаты и определять причины медленной работы.

В главе 11 некоторые рассмотренные ранее идеи и методы обсуждаются вновь в контексте более точной обработки ошибок. Мы увидим, как можно модифицировать архитектуру плагинов с целью более органичной обработ-

ки ошибок при полном сохранении обратной совместимости, а также более внимательно приглядимся к процессам проектирования, подразумевающим обработку ошибок в момент возникновения.

В последней главе 12 мы воспользуемся итераторами и сопрограммами, чтобы обогатить разработанные ранее инструментальные панели средствами, которые не ограничиваются ролью пассивных сборщиков данных, а активно исследуют собранные данные, что позволяет строить многошаговые аналитические процессы.

Версия Python

На момент написания книги текущей была версия Python 3.8, поэтому все примеры протестированы в ней и в первых рабочих версиях Python 3.9. Я не рекомендую использовать более старые версии. Некоторые примеры, хотя их очень мало, не работают в версиях Python 3.7 и Python 3.6.

Для проработки примеров вам понадобится программа pip. Если в вашей системе установлен Python, то, наверное, установлена и pip. Но в некоторых операционных системах pip намеренно удаляется из дистрибутива Python, и тогда вам придется установить ее явно, воспользовавшись встроенным в систему диспетчером пакетов. Это типичная ситуация в дистрибутивах на базе Debian, для ее разрешения нужно выполнить команду `sudo apt install python3-pip`. В других операционных системах воспользуйтесь командой `python -m ensurepip --upgrade`, которая заставляет Python найти последнюю версию pip, или изучите инструкции, относящиеся к конкретной системе.

Электронные версии примеров кода и списка опечаток имеются в издательстве и на сайте книги https://advancedpython.dev. Это первое место, куда следует обращаться в случае обнаружения каких-либо проблем при работе с книгой.

Глава 1

Прототипирование и среды разработки

В этой главе мы обсудим различные способы экспериментирования с функциями Python и расскажем, когда какой использовать. Воспользовавшись одним из этих способов, мы напишем несколько простеньких функций для извлечения первых фрагментов данных, которые собираемся агрегировать, и посмотрим, как собрать из них простую командную утилиту.

Прототипирование в Python

В любом проекте на Python, не важно, потрачено на разработку несколько часов или речь идет о системе, работающей годами, приходится прототипировать функции. Быть может, это первое, с чего вы начинаете, а быть может, такая необходимость возникает в середине проекта, но рано или поздно вы будете экспериментировать с кодом в оболочке Python.

Есть два основных подхода к прототипированию: выполнить код целиком и посмотреть на результаты или выполнять предложения по одному и смотреть, что получается. Вообще говоря, выполнение предложений по одному более продуктивно, но иногда проще прогнать сразу целый блок, если вы уверены в его правильности.

Оболочка Python (ее также называют REPL – от **R**ead, **E**val, **P**rint, **L**oop – прочитать, вычислить, напечатать, повторить) – это то, с чего обычно начинают знакомство с Python. Запустить интерпретатор и выполнять команды одну за другой – эффективный способ скорее приступить к кодированию. Так мы можем сразу увидеть результат каждой команды, а затем изменить входные данные, не изменяя значения переменных. Сравните с компилируемым языком, когда приходится компилировать файл, а затем запускать исполняемую программу. Для простых программ, написанных на интерпретируемом языке типа Python, задержка оказывается намного меньше.

Прототипирование с помощью REPL

Сильная сторона цикла REPL в том, что он дает возможность выполнить простой код и получить интуитивное представление о работе функций. В меньшей степени он подходит для случаев, когда в коде много команд управления потоком выполнения, да и ошибок такая методика не прощает. Сделав ошибку при наборе промежуточной строки тела функции, вы должны будете начать с начала, исправить ошибочную строку недостаточно. Модификация переменной с помощью одной строки кода и последующий анализ результата – вот почти оптимальное использование REPL для прототипирования.

Например, я никак не могу запомнить, как работает встроенная функция `filter(...)`. Есть несколько способов освежить память. Первый – посмотреть документацию на сайте Python или в редакторе либо IDE. Альтернатива – включить функцию в программу и проверить, совпадает ли полученный результат с ожидаемым, или воспользоваться оболочкой REPL, чтобы найти ссылку на документацию, либо просто выполнить в ней функцию.

На практике я обычно останавливаюсь на последнем варианте. Ниже показан типичный пример, когда в первой попытке я перепутал порядок аргументов, во второй интерпретатор напомнил мне, что `filter` возвращает специальный объект, а не кортеж и не список, а в третьей я убедился, что `filter` оставляет элементы, удовлетворяющие условию, а не исключает их.

```
>>> filter(range(10), lambda x: x == 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'function' object is not iterable
>>> filter(lambda x: x == 5, range(10))
<filter object at 0x033854F0>
>>> tuple(filter(lambda x: x == 5, range(10)))
(5,)
```

Примечание. Встроенная функция `help(...)` – неоценимый помощник, когда нужно понять, как работает функция. Поскольку `filter` содержит понятную строку документации, было бы проще вызвать `help(filter)` и прочитать, что она напишет. Но если несколько функций сцеплено, особенно при попытке разобраться в существующем коде, возможность интерактивно поэкспериментировать с данными очень полезна.

Если мы попробуем использовать цикл REPL в задаче, где больше команд управления потоком, например в знаменитом примере FizzBuzz, предлагаемом в ходе собеседования (листинг 1.1), то увидим, как именно он не прощает ошибок.

Листинг 1.1 ❖ fizzbuzz.py – типичная реализация

```
for num in range(1, 101):
    val = ''
    if num % 3 == 0:
```

```
        val += 'Fizz'
    if num % 5 == 0:
        val += 'Buzz'
    if not val:
        val = str(num)
    print(val)
```

Если бы мы писали этот код шаг за шагом, то могли бы начать с создания цикла, который просто выводит числа:

```
>>> for num in range(1, 101):
...     print(num)
...
1
.
.
.
98
99
100
```

Теперь мы видим, что числа от 1 до 100 напечатаны подряд, и можем потихоньку добавлять логику:

```
>>> for num in range(1, 101):
...     if num % 3 == 0:
...         print('Fizz')
...     else:
...         print(num)
4
...
1
.
.
.
98
Fizz
100
```

На каждом шаге нам приходится повторно вводить код, который уже был введен прежде, – иногда с мелкими изменениями, а иногда вообще без изменений. Ранее введенные строки редактировать нельзя, поэтому любая опечатка – и цикл нужно будет набирать с самого начала.

Возможно, вы решите прототипировать только тело цикла, а не весь цикл, чтобы было проще следить за тем, какое действие оказывают условия. В данном случае значения `n` от 1 до 14 правильно генерируются предложением `if` с тремя ветвями, а первая ошибка имеет место при `n=15`. Поскольку это происходит в середине тела цикла, трудно понять, как взаимодействуют условия.

Тут мы впервые сталкиваемся с различием между интерпретацией отступов в оболочке REPL и в скрипте. В режиме REPL интерпретатор Python более строго относится к отступам, чем в скрипте, – он *требует*, чтобы вы добавили пустую строку, перед тем как вернуться на уровень отступа 0.

```
>>> num = 15
>>> if num % 3 == 0:
...     print('Fizz')
... if num % 5 == 0:
  File "<stdin>", line 3
    if num % 5 == 0:
     ^
SyntaxError: invalid syntax
```

Кроме того, REPL разрешает пустую строку только при возврате на уровень отступа 0, тогда как в Python-файле она считается неявным продолжением кода на последнем уровне отступа. Программа в листинге 1.2 (который отличается от листинга 1.1 только наличием пустых строк) работает правильно, если вызвать ее командой `python fizzbuzz_blank_lines.py`.

Листинг 1.2 ❖ fizzbuzz_blank_lines.py

```
for num in range(1, 101):
    val = ''
    if num % 3 == 0:
        val += 'Fizz'
    if num % 5 == 0:
        val += 'Buzz'

    if not val:
        val = str(num)

    print(val)
```

Однако при вводе кода из листинга 1.2 в интерпретаторе Python выдаются следующие ошибки из-за различий в правилах разбора отступов:

```
>>> for num in range(1, 101):
...     val = ''
...     if num % 3 == 0:
...         val += 'Fizz'
...     if num % 5 == 0:
...         val += 'Buzz'
...
>>>     if not val:
  File "<stdin>", line 1
  if not val:
  ^
IndentationError: unexpected indent
>>>         val = str(num)
  File "<stdin>", line 1
    val = str(num)
    ^
IndentationError: unexpected indent
>>>
>>>     print(val)
  File "<stdin>", line 1
    print(val)
```

```
    ^
IndentationError: unexpected indent
```

При использовании оболочки REPL для прототипирования цикла или условного предложения легко допустить ошибку, если вы привыкли к файлам с кодом. Раздражение от ошибок и необходимости повторно вводить в код – достаточная причина, чтобы плюнуть на экономию времени и отдать предпочтение простым скриптам. Конечно, клавиши со стрелками позволяют вернуться к ранее введенным строкам, но многострочные конструкции, в частности циклы, не группируются в единое целое, поэтому повторно выполнить тело цикла очень трудно. А из-за приглашений `>>>` и `...` трудно скопировать предыдущие строки через буфер обмена как для повторного выполнения, так и для включения в файл.

Прототипирование с помощью Python-скрипта

Ничто не мешает прототипировать код по-другому: написать простой скрипт на Python и запускать его, пока результат не окажется правильным. В отличие от REPL, при таком подходе легко выполнить программу повторно в случае ошибки, а сам код хранится в файле, а не в буфере прокрутки терминала[1]. К сожалению, это также означает, что с кодом нельзя взаимодействовать в процессе выполнения, что ведет к «отладке с помощью printf», названной так по имени функции печати в языке C.

Как следует из названия, практически единственный способ получить информацию о выполнении скрипта – выводить информацию на консоль с помощью функции `print(...)`. В нашем примере пришлось бы добавить печать в тело цикла, чтобы узнать, что происходит на каждой итерации:

```
for num in range(1,101):
    print(f"n: {num} n%3: {num%3} n%5: {num%5}")
Будет напечатано:
n: 1 n%3: 1 n%5: 1
.
.
.
n: 98 n%3: 2 n%5: 3
n: 99 n%3: 0 n%5: 4
n: 100 n%3: 1 n%5: 0
```

Совет. Для отладочной печати полезны f-строки, поскольку позволяют включать (интерполировать) переменные в строку без дополнительного форматирования.

Из этой распечатки понятно, что делает скрипт, но возникает некоторое дублирование логики. Из-за этого можно пропустить какие-то ошибки, что

[1] Вы возблагодарите судьбу за это в первый раз, когда случайно закроете окно терминала и потеряете весь код, над которым работали.

приведет к потере времени. Тот факт, что код хранится на диске, – основное преимущество по сравнению с REPL, но для программиста так работать менее удобно. Исправление опечаток и простых ошибок раздражает, поскольку приходится переключаться между редактированием файла и запуском его в терминале[1]. Кроме того, для того чтобы сразу увидеть интересующую информацию, необходимо продумывать структуру предложений печати. Несмотря на все недостатки, добавить отладочную печать в существующую систему настолько просто, что этот подход к отладке является одним из самых распространенных, особенно когда требуется понять природу проблемы.

Прототипирование с помощью скриптов и pdb

Встроенный в Python отладчик pdb – один из самых полезных инструментов в арсенале любого Python-разработчика. Это наиболее эффективный способ отладить сложные куски кода и практически единственный способ понять, что скрипт делает внутри многошаговых выражений типа спискового включения[2].

Во многих отношениях прототипирование кода можно считать особой формой отладки. Мы знаем, что написанный код неполон и содержит ошибки, но вместо того чтобы искать единичный дефект, мы пытаемся разобраться со сложностью по частям. Многие средства pdb упрощают эту задачу.

В начале сеанса pdb появляется приглашение (Pdb), позволяющее взаимодействовать с отладчиком. На мой взгляд, самые важные команды – **s**tep, **n**ext, **b**reak, **c**ontinue, **p**retty**p**rint и **d**ebug[3].

Команды step и next выполняют текущее предложение и переходят к следующему. Отличаются они тем, что считать «следующим». Step переходит к следующему предложению, где бы оно ни находилось, так что если текущая строка содержит вызов функции, то следующей строкой будет первая строка этой функции. Next не заходит внутрь функции, т. е. следующим будет следующее предложение в текущей функции. Если вы хотите узнать, что делает функция, зайдите внутрь с помощью step. Если вы уверены, что функция работает правильно, то выполните ее с помощью next, не заходя внутрь, и сразу получите результат. Команды break и continue позволяют выполнять длинные участки кода сразу, а не в пошаговом режиме. В команде break задается номер строки, на которой должен остановиться отладчик, и необязательное условие, которое должно быть выполнено, чтобы остановка произошла, например `break 20 x==1`. Команда continue возвращается в обычный режим выполнения, т. е. приглашение pdb появится, только когда отладчик дойдет до очередной точки прерывания.

---

[1] В некоторые текстовые редакторы терминал интегрирован специально для того, чтобы избежать такой смены контекста.

[2] Pdb позволяет пошагово выполнять итерации спискового включения, как будто это цикл. Это полезно, когда требуется понять, что не так с существующим кодом, но мешает, если списковое включение приходится проходить в процессе отладки, хотя проблема не в нем.

[3] Их можно сокращать до одной или нескольких букв, выделенных полужирным шрифтом, т. е. вместо step писать s, вместо prettyprint – pp и т. д.

Совет. Если вы находите визуальное отображение состояния более естественным, то, возможно, вам будет трудно следить за тем, где сейчас находится отладчик. Я рекомендую установить отладчик pdb++, который выводит текст программы и выделяет в нем текущую строку. Интегрированные среды разработки (IDE), в частности PyCharm, идут дальше и позволяют устанавливать точки прерывания в исполняемой программе, а также управлять пошаговым режимом прямо в окне редактора.

Наконец, команда debug позволяет задать произвольное выражение для выполнения в пошаговом режиме. То есть мы можем вызвать любую функцию с любыми параметрами. Это очень удобно, когда вы уже прошли какую-то точку с помощью команды next или continue и только потом осознали, где ошибка. Команда имеет вид `debug somefunction()` и изменяет приглашение `(Pdb)`, добавляя лишнюю пару скобок – `((Pdb))` – и давая тем самым понять, что вы находитесь во вложенном сеансе pdb[1].

Посмертная отладка

Существует два способа вызвать pdb: явно в коде и непосредственно для проведения так называемой «посмертной отладки». В последнем случае скрипт запускается в pdb, и, если произойдет исключение, pdb получает управление. Для этого скрипт запускается командой `python -m pdb yourscript.py`, а не `python yourscript.py`. Скрипт не начинает работать автоматически, сначала выводится приглашение pdb, чтобы можно было расставить точки прерывания. Чтобы скрипт начал работать, нужно выполнить команду `continue`. Управление возвращается pdb, если встретится точка прерывания или по завершении программы. Если программа завершилась из-за ошибки, то можно будет посмотреть, какие значения имели переменные в момент ошибки.

Вместо этого можно с помощью команд step выполнять предложения программы по одному, но всегда, кроме разве что самых простых скриптов, лучше установить точку прерывания в месте, с которого вы хотите начать отладку, и пошагово выполнять программу, начиная оттуда.

Ниже показано, как запустить программу в листинге 1.1 в pdb и установить условную точку прерывания (вывод сокращен):

```
> python -m pdb fizzbuzz.py
> c:\fizzbuzz_pdb.py(1)<module>()
-> def fizzbuzz(num):
(Pdb) break 2, num==15
Breakpoint 1 at c:\fizzbuzz.py:2
(Pdb) continue
1
.
.
```

[1] Как-то раз я так сильно запутался, ища ошибку, что вынужден был использовать `debug` многократно, пока приглашение pdb не приняло вид ((((((Pdb)))))). Это антипаттерн, потому что очень легко потерять ориентацию в программе. Оказавшись в такой ситуации, попробуйте использовать условные точки прерывания.

```
.
13
14
> c:\fizzbuzz.py(2)fizzbuzz()
-> val = ''
(Pdb) p num
15
```

Этот способ хорошо работает в сочетании с описанным выше запуском скрипта. Он позволяет расставлять точки прерывания на разных этапах выполнения кода и автоматически передает управление pdb в случае возникновения исключения, так что нам не нужно предугадывать тип и место ошибки.

Функция breakpoint

Встроенная функция `breakpoint()`[1] позволяет точно указать, в каком месте программы следует передать управление pdb. При вызове этой функции исполнение немедленно прекращается и выводится приглашение pdb. Все выглядит так, будто в данном месте ранее была установлена точка прерывания. Функцию `breakpoint()` часто используют внутри предложения `if` или в обработчике исключения, чтобы сымитировать условную точку прерывания и посмертную отладку. Конечно, при этом приходится изменять исходный код (поэтому способ не подходит для отладки ошибок, возникающих только в производственном режиме), но зато отпадает необходимость расставлять точки прерывания при каждом запуске программы.

Чтобы отладить скрипт fizzbuzz в месте, где вычисляется значение 15, нужно было бы добавить новое условие `num == 15` и вызов `breakpoint()`, когда оно удовлетворяется (см. листинг 1.3).

Листинг 1.3 ❖ fizzbuzz_with_breakpoint.py

```
for num in range(1, 101):
    val = ''
    if num == 15:
        breakpoint()
    if num % 3 == 0:
        val += 'Fizz'
    if num % 5 == 0:
        val += 'Buzz'
    if not val:
        val = str(num)
    print(val)
```

Чтобы применить этот подход к прототипированию, создайте простой Python-файл, содержащий предложения импорта, которые предположительно могут понадобиться, и тестовые данные. Затем добавьте в конец файла вызов `breakpoint()`. Теперь при выполнении файла вы окажетесь в интерактивной среде, где будут доступны все нужные вам функции и данные.

---

[1] В документации можно встретить рекомендацию включать предложения `import pdb; pdb.set_trace()`. Это устаревший стиль, который все еще широко применяется, но происходит при этом то же самое, только слов больше, а ясности меньше.

Совет. Я всячески рекомендую использовать для отладки сложных многопоточных приложений библиотеку `remote-pdb`. Для этого установите пакет `remote-pdb` и запустите приложение, задав переменную окружения `PYTHONBREAKPOINT=remote_pdb.set_trace python yourscript.py`. При вызове из программы функции `breakpoint()` на консоль будет выведена информация о соединении. Дополнительные сведения см. в документации по `remote-pdb`.

Прототипирование с помощью Jupyter

Jupyter – это комплект инструментов для организации более удобной интерактивной работы на языках, поддерживающих цикл REPL. Он поддерживает различные средства взаимодействия с программой, например отображение виджетов, привязанных к входу или выходу функций, что заметно упрощает работу со сложными функциями для пользователей, не являющихся техническими специалистами. На данном этапе нам полезно то, что Jupyter позволяет разбить код на логические блоки и запускать их независимо, а также сохранять эти блоки, чтобы вернуться к ним позже.

Jupyter написан на Python, но является единым интерфейсом к языкам Julia, Python и R. Он задуман как механизм для организации автономных программ, предлагающих простые пользовательские интерфейсы, например для анализа данных. Многие программисты, пишущие на Python, в особенности научные работники, создают Jupyter-блокноты, а не консольные скрипты. В *этой* главе мы не будем использовать Jupyter подобным образом, а интересует он нас, потому что отлично приспособлен для решения задач прототипирования.

В полном соответствии с поставленной при проектировании целью Jupyter поддерживает также языки Haskell, Lua, Perl, PHP, Rust, Node.js и многие другие. Для всех этих языков есть IDE, оболочка REPL, сайты с документацией и т. д. Одно из главных преимуществ Jupyter с точки зрения прототипирования – возможность разработать технологический процесс, который будет работать с незнакомыми средами и языками. Например, веб-разработчики широкого профиля, занимающиеся программированием на стороне клиента и сервера, часто пишут как на Python, так и на JavaScript. С другой стороны, научным работникам может понадобиться простой доступ к Python и R. Наличие единого интерфейса позволяет сгладить некоторые различия между языками.

Поскольку Jupyter жестко не привязан к Python и располагает встроенной поддержкой для выбора среды исполнения кода, я рекомендую устанавливать его таким образом, чтобы он был доступен из любого места системы. Если обычно вы устанавливаете утилиты Python в виртуальной среде, то все нормально[1]. Я, однако, установил Jupyter в свое пользовательское окружение:

```
> python -m pip install --user jupyter
```

[1] На самом деле многие предпочитают создавать виртуальную среду специально для Jupyter и включать ее в системный список путей, чтобы избежать конфликта версий в своем глобальном пространстве имен.

Примечание. Если Jupyter установлен в пользовательском режиме, то необходимо включить каталог, содержащий двоичные файлы, в системный список путей. Допустимые альтернативы – установка в глобальный каталог Python или с помощью диспетчера пакетов; лучше применять единый способ установки инструментов, а не разводить зоопарк.

Если для прототипирования применяется Jupyter, то можно разбить код на логические блоки и запускать их по отдельности или последовательно. Все блоки хранятся на диске и допускают редактирование, как если бы использовался скрипт, но мы можем контролировать, какие блоки работают, и писать новый код, не изменяя содержимое переменных. В этом смысле подход напоминает использование REPL, т. к. мы можем экспериментировать с кодом, не прерываясь на запуск скрипта.

Существует два способа доступа к инструментам Jupyter: через веб с помощью сервера Jupyter-блокнотов или путем замены стандартной оболочки REPL. В обоих случаях в основе лежит идея ячеек, т. е. независимых единиц выполнения, которые можно перезапускать в любое время. И блокнот, и REPL используют один и тот же базовый интерфейс к Python, называемый IPython. В IPython нет проблем с разбором отступов, присущих стандартной оболочке REPL, и он поддерживает простой перезапуск кода, ранее введенного в сеансе.

Блокнот дружелюбнее к пользователю, чем оболочка, но у него есть недостаток: он доступен только в веб-браузере, но не в обычном текстовом редакторе или IDE[1]. Я горячо рекомендую использовать интерфейс блокнота, потому что, когда дело дойдет до перезапуска ячеек и редактирования многострочных ячеек, он заметно повысит вашу продуктивность благодаря интуитивно более понятному интерфейсу.

Блокноты

Чтобы приступить к прототипированию, запустите сервер блокнотов Jupyter и с помощью веб-интерфейса создайте новый блокнот.

```
> jupyter notebook
```

После того как блокнот загружен, введите код в первую ячейку и нажмите кнопку **run**. Поддерживаются многие горячие клавиши, типичные для редакторов кода, а также автоматический отступ в начале нового блока (рис. 1.1).

Pdb работает с Jupyter-блокнотом через веб-интерфейс точно так же, как в командной строке, т. е. прерывает выполнение и отображает приглашение (рис. 1.2). Этот интерфейс поддерживает всю стандартную функциональность pdb, так что все советы, приведенные в разделе о pdb, сохраняют силу и в среде Jupyter.

---

[1] Некоторые редакторы, например профессиональная версия PyCharm IDE и Microsoft VSCode, стали предлагать частичный эквивалент интерфейса блокнота внутри IDE. Функциональность неполная, но на удивление хорошая.

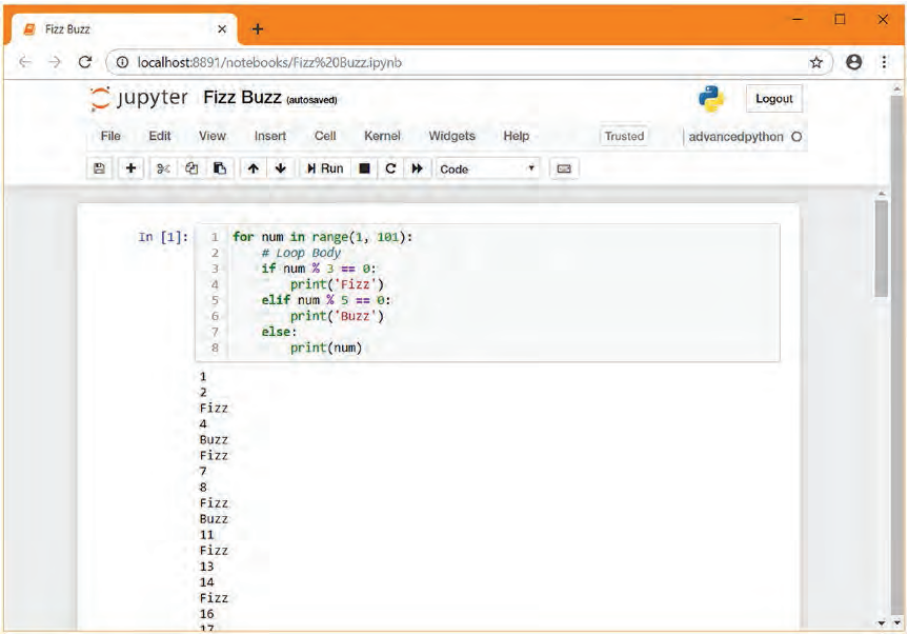

Рис. 1.1 ❖ fizzbuzz в Jupyter-блокноте

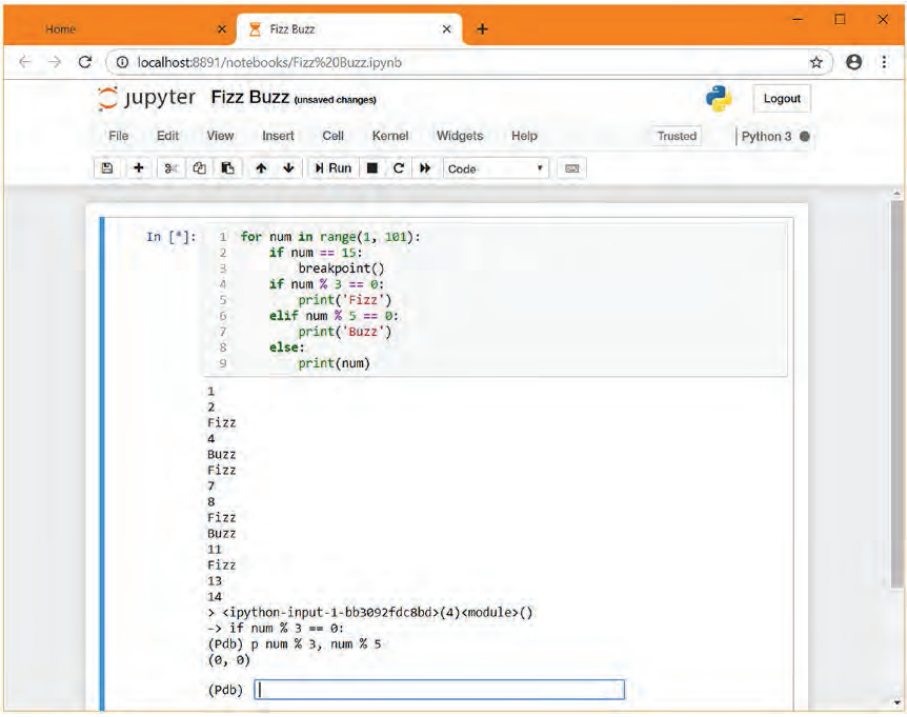

Рис. 1.2 ❖ pdb в Jupyter-блокноте

Прототипирование в этой главе

У всех рассмотренных выше методов имеются плюсы и минусы, но у каждого есть своя ниша. Для очень простых однострочных скриптов, например спискового включения, я часто использую оболочку REPL, поскольку ее быстрее всего запустить, а сложный поток управления, способный затруднить отладку, отсутствует.

Для более сложных задач, например когда используются функции из внешних библиотек для выполнения нескольких вещей, чаще всего полезен более функционально насыщенный подход. Я рекомендую попробовать разные подходы к прототипированию и решить, какой из них для вас удобнее.

Выбирая метод, подходящий в конкретной ситуации, следует учитывать различные факторы. В качестве общего правила я рекомендую брать самый левый столбец в табл. 1.1, удовлетворяющий вашим требованиям. Если выбрать подход, находящийся правее, то пострадает удобство работы, а если левее, то может возникнуть раздражение при попытке выполнить действия, которые проще выполняются в других инструментах.

Таблица 1.1. Сравнение сред для прототипирования

Фактор	REPL	**Скрипт**	**Скрипт + pdb**	**Jupyter**	**Jupyter + pdb**
Отступы в коде	Строгие правила	Обычные правила	Обычные правила	Обычные правила	Обычные правила
Перезапуск предыдущих команд	Одна напечатанная строка	Только скрипт целиком	Скрипт целиком или переход к предыдущей строке	Логические блоки	Логические блоки
Дискретность	Единица выполнения – блок с отступом	Единица выполнения – весь скрипт	Пошаговое выполнение предложений	Единица выполнения – логический блок	Пошаговое выполнение предложений
Интроспекция	Допустима интроспекция между логическими блоками	Отсутствует	Допустима интроспекция между предложениями	Допустима интроспекция между логическими блоками	Допустима интроспекция между предложениями
Сохранение	Ничего не сохраняется	Сохраняются команды	Команды сохраняются, взаимодействия с pdb – нет	Сохраняются команды и вывод	Сохраняются команды и вывод
Редактирование	Команды необходимо вводить заново	Любую команду можно редактировать, но скрипт нужно перезапускать целиком	Любую команду можно редактировать, но скрипт нужно перезапускать целиком	Любую команду можно редактировать, но логический блок нужно перезапускать целиком	Любую команду можно редактировать, но логический блок нужно перезапускать целиком

В этой главе мы будем прототипировать несколько функций, возвращающих данные о системе, в которой они выполняются. Они зависят от внешних библиотек, иногда нам понадобятся простые циклы, но их будет немного.

Поскольку сложные управляющие конструкции нам не встретятся, то отступы в коде не составляют проблемы. Перезапуск ранее введенных команд будет полезен, т. к. мы собираемся работать с несколькими источниками данных. Вполне возможно, что некоторые источники медленные, поэтому

при работе с ними нам бы не хотелось заново выполнять все команды. Это исключает из рассмотрения REPL, а Jupyter кажется более подходящим, чем процедуры на базе скриптов.

Мы хотим просматривать данные из каждого источника, но маловероятно, что понадобится интроспекция внутренних переменных для отдельных источников, поэтому подходы, основанные на pdb, представляются избыточными (а если наша точка зрения изменится, то всегда можно будет добавить вызов `breakpoint()`). Мы хотим сохранять написанный код, но это требование исключает только цикл REPL, а он уже и так исключен. Наконец, мы хотим иметь возможность редактировать код и смотреть, к чему это приводит.

Сравнив эти требования с табл. 1.1, мы приходим к табл. 1.2, из которой видно, что Jupyter отвечает всем пожеланиям, тогда как подход на основе скрипта хорош, но не вполне оптимален с точки зрения возможности перезапуска предыдущих команд.

Поэтому в этой главе мы будем использовать для прототипирования Jupyter-блокнот. Далее мы рассмотрим некоторые преимущества, которые дает Jupyter, а также способы его эффективного использования в процессе разработки на Python. Но мы не будем обсуждать его применение для создания автономных программ, распространяемых в виде блокнотов.

Таблица 1.2. Матрица соответствия между возможностями различных подходов и требованиями[1]

Фактор	REPL	Скрипт	Скрипт + pdb	Jupyter	Jupyter + pdb
Отступы в коде	✓	✓	✓	✓	✓
Перезапуск предыдущих команд	×	⚠	⚠	✓	✓
Дискретность	×	×	⚠	✓	⚠
Интроспекция	✓	✓	✓	✓	✓
Сохранение	×	✓	✓	✓	✓
Редактирование	×	✓	✓	✓	✓

Подготовка окружения

Итак, выбор сделан, и теперь требуется установить библиотеки и управлять зависимостями проекта, а это значит, что нам нужно виртуальное окружение. Мы определим зависимости с помощью программы pipenv, которая одновременно создает изолированное виртуальное окружение и прекрасно справляется с управлением зависимостями.

```
> python -m pip install --user pipenv
```

[1] ✓ означает, что требование удовлетворено, × – что не удовлетворено, а ⚠ – что требование удовлетворено, но работать неудобно.

Почему PIPENV

У систем для создания изолированного окружения для Python долгая история. Скорее всего, раньше вы пользовались системой virtualenv. Возможно, вам также доводилось работать с venv, conda, buildout, virtualenvwrapper или pyenv. Быть может, вы даже создавали среду самостоятельно, манипулируя `sys.path` или создавая `lnk`-файлы во внутренних каталогах Python.

У всех этих методов есть плюсы и минусы (за исключением создания среды вручную – тут я вижу только минусы), но pipenv предоставляет великолепную поддержку для управления прямыми зависимостями, при этом хранит полный список версий зависимостей, которые гарантированно работают правильно, и всегда поддерживает среду в актуальном состоянии. Поэтому она отлично подходит для проектов на современном чистом Python. Если у вас уже есть сложившийся технологический процесс, включающий сборку двоичных файлов или работу с устаревшими версиями пакетов, то, наверное, лучше его придерживаться, чем переходить на pipenv. В частности, если вы пользуетесь Anaconda, потому что занимаетесь научными расчетами, то нет никакой необходимости переключаться на pipenv. При желании можете выполнить команду `pipenv --site-packages`, чтобы pipenv включила пакеты, управляемые conda, в дополнение к своим собственным.

Цикл разработки pipenv довольно долгий по сравнению с другими инструментами Python. Бывает, что проходит несколько месяцев или лет без выпуска новой версии. Вообще говоря, я нахожу pipenv стабильным и надежным продуктом, потому и рекомендую его. Диспетчеры пакетов с более частым графиком выпуска версий ведут себя бесцеремонно, вынуждая пользователя регулярно реагировать на несовместимые изменения.

Чтобы pipenv работала эффективно, требуется, чтобы ответственные за сопровождение нужных вам пакетов правильно объявляли зависимости. В некоторых пакетах это не так, например задается только содержащий зависимость пакет без указания ограничений на версии, даже если такие ограничения существуют. Проблема может возникнуть, например, потому что недавно была выпущена новая основная версия подзависимости. В таких случаях вы можете самостоятельно добавить ограничения на допустимые версии (это называется *закреплением версий*).

Если оказалось, что пакет с требуемым закрепленным номером версии отсутствует, уведомьте об этом ответственных за сопровождение. Эти люди часто сильно заняты и, возможно, еще не заметили проблему – не думайте, что раз они такие опытные, то в вашей помощи не нуждаются. Для большинства Python-пакетов имеются репозитории на GitHub с системой отслеживания ошибок. По ее журналу можно узнать, сообщал ли уже кто-нибудь о данной проблеме, а если нет, то это отличный способ внести свой вклад в разработку пакета, которым вы пользуетесь.

Подготовка нового проекта

Первым делом создайте новый каталог для проекта и перейдите в него. Мы хотим объявить зависимость от пакета `ipykernel`, который содержит код для управления интерфейсом между Python и Jupyter. Поэтому нам нужно, чтобы сам пакет и код его библиотеки были доступны в новом изолированном окружении.

```
> mkdir advancedpython
> cd advancedpython
> pipenv install ipykernel --dev
> pipenv run ipython kernel install --user --name=advancedpython
```

Последняя строка говорит, что копию IPython следует установить в изолированном окружении в качестве ядра, доступного текущему пользователю, под именем `advancedpython`. Это позволит выбирать ядро, не активируя каждый раз данное изолированное окружение вручную. Список установленных ядер можно вывести командой `jupyter kernelspec list`, а для удаления ядра служит команда `jupyter kernelspec remove`.

Теперь можно запустить Jupyter и решить, хотим ли мы выполнять код с помощью системной версии Python или в своем изолированном окружении. Я рекомендую открывать для этого новое окно команд, потому что Jupyter работает в приоритетном режиме и скоро нам понадобится использовать командную строку. Если ранее при чтении этой главы вы запускали сервер Jupyter, то я рекомендую остановить его, прежде чем запускать новый. Мы хотим использовать созданный ранее рабочий каталог, поэтому перейдите в него, если он не является текущим в новом окне.

```
> cd advancedpython
> jupyter notebook
```

Открывается браузер, в котором отображен интерфейс Jupyter и список созданных нами каталогов, – см. рис. 1.3. Итак, проект подготовлен, и можно

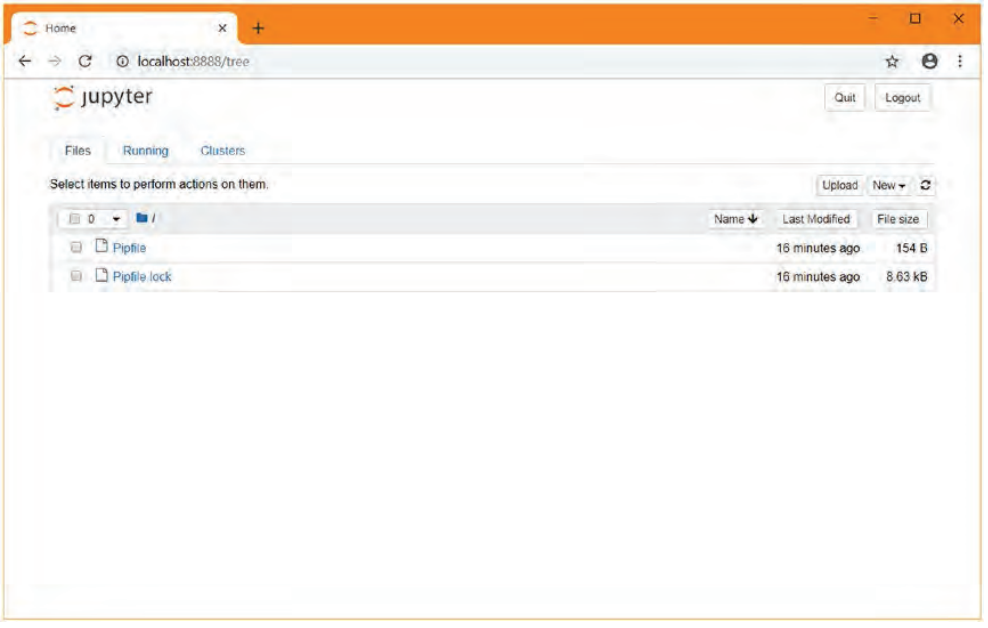

Рис. 1.3 ❖ Начальный экран Jupyter в новом каталоге pipenv

приступать к прототипированию. Нажмите кнопку **New**, а затем выберите advancedpython. Перед нами основной интерфейс редактирования. Мы имеем одну «ячейку», которая ничего не содержит и не исполнялась. Любой введенный в нее код можно выполнить, нажав кнопку **Run** сверху. Jupyter отображает все, что выводит код в ячейке, а также новую пустую ячейку для ввода последующего кода. Можно считать, что ячейка – это приблизительный эквивалент тела функции. Обычно ячейки содержат несколько логически связанных предложений, которые исполняются как единое целое.

Прототипирование скриптов

Первым шагом было бы логично написать Python-программу, которая возвращает различную информацию о системе, в которой выполняется. Впоследствии эти сведения станут частью агрегируемых данных, но пока хватит каких-нибудь простых данных.

Начнем с малого – воспользуемся первой ячейкой, чтобы узнать, с какой версией Python работаем (рис. 1.4). Эта функция находится в стандартной библиотеке Python и работает на всех платформах, а в дальнейшем мы сможем включить в эту ячейку что-нибудь более интересное.

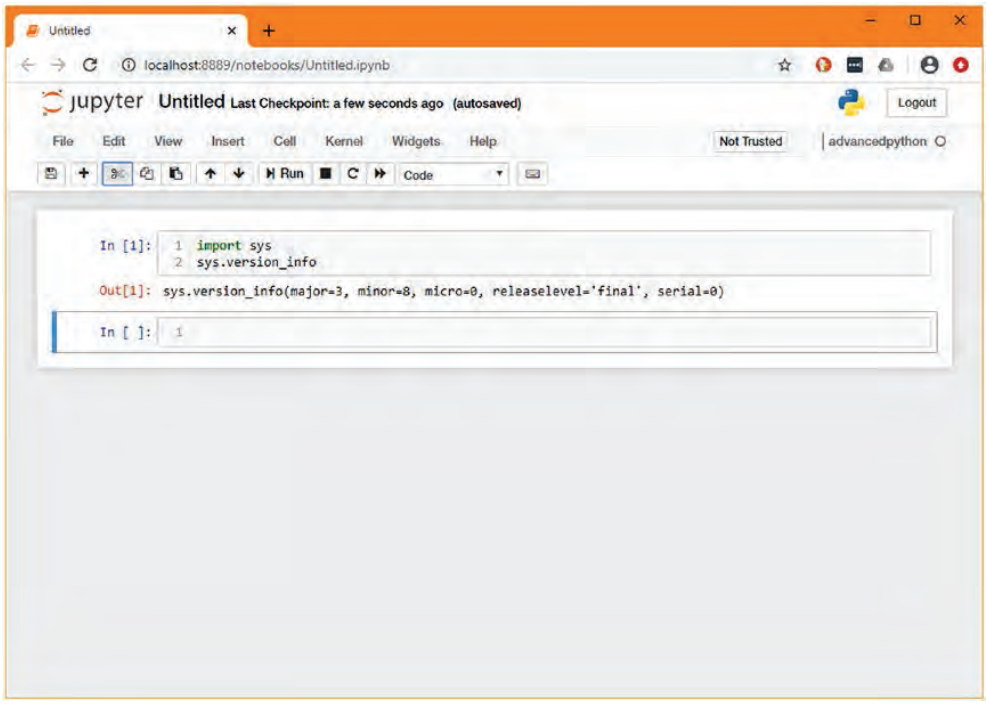

Рис. 1.4 ❖ Простой Jupyter-блокнот,
в котором показано значение переменной sys.version_info

Jupyter показывает значение последней строки в ячейке, а также все, что было напечатано явно. Поскольку последняя строка содержала вызов функции, то показывается возвращенный ей результат[1].

Еще одна полезная для агрегирования информация – IP-адрес машины. Она не возвращается в виде одной переменной, чтобы ее получить, нужно вызвать несколько функций и обработать полученные результаты. Поскольку простого импорта для этого недостаточно, имеет смысл поочередно создавать переменные в нескольких ячейках. Тогда с первого взгляда видно, что вернул предыдущий вызов, и в следующей ячейке все предыдущие переменные доступны. Этот пошаговый процесс позволяет сосредоточиться на новых кусках кода, не обращая внимания на те, что уже отработали.

В конце мы получим что-то похожее на код, изображенный на рис. 1.5, где видны различные IP-адреса данного компьютера. На втором шаге становится понятно, что доступны IPv4- и IPv6-адреса. Из-за этого третий шаг оказывается чуть более сложным, поскольку я решил выделить не только значение адреса, но и его тип. Выполняя эти шаги последовательно, мы можем учитывать информацию, полученную ранее. Возможность повторно выполнить только тело цикла, не изменяя окна, – хороший пример сильных сторон Jupyter в действии.

Сейчас у нас есть три ячейки с кодом для вычисления IP-адресов, т. е. не существует взаимно однозначного соответствия между ячейками и логическими компонентами. Чтобы исправить ситуацию, выберите верхнюю ячейку, а затем выполните команду **Merge Cell Below** (Объединить со следующей ячейкой) из меню **Edit**. Проделайте это два раза, чтобы присоединить обе дополнительные ячейки. Теперь полная реализация хранится как один логический блок (рис. 1.6). И операцию можно выполнить разом, а не прогонять все три ячейки для получения результата. Полезно также привести содержимое ячейки в порядок – нам больше не нужно печатать промежуточные значения, поэтому можно избавиться от повторяющихся адресов.

---

[1] Это означает, что если последней строкой в ячейке было присваивание, то присвоенное значение не показывается. Дело в том, что для предложений присваивания в Python значение не определено. Обычно значения переменных выводят явно, например:

```
version = sys.version_info

version
```

Можно было бы использовать появившийся в Python 3.8 «моржовый» оператор (`version := sys.version_info`), поскольку результатом его вычисления является присвоенное значение, но выглядит это странновато, поэтому я не рекомендую использовать его просто для присваивания. Лучше всего применять его в условиях циклов и в предложениях if, где он выглядит гораздо естественнее, поскольку в таких случаях не нужны скобки.

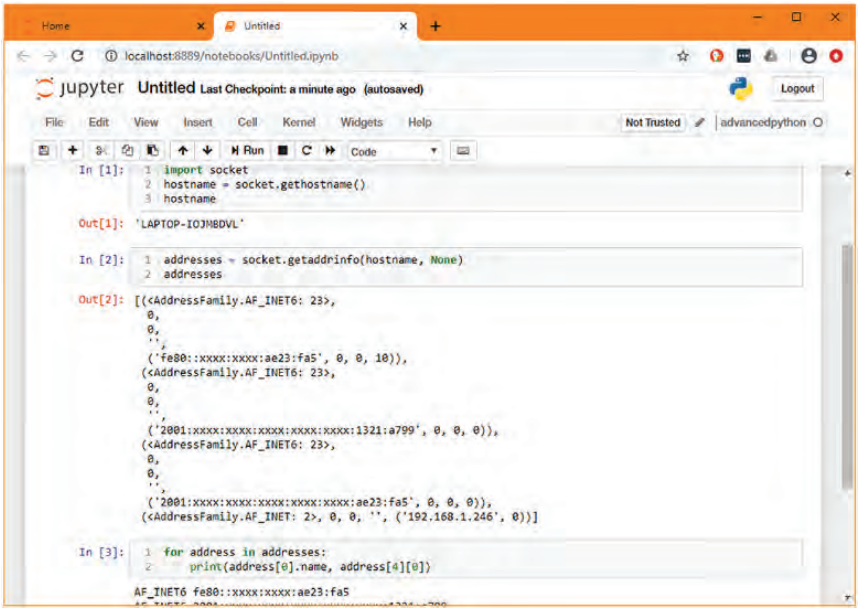

Рис. 1.5 ❖ Прототипирование сложной функции в нескольких ячейках[1]

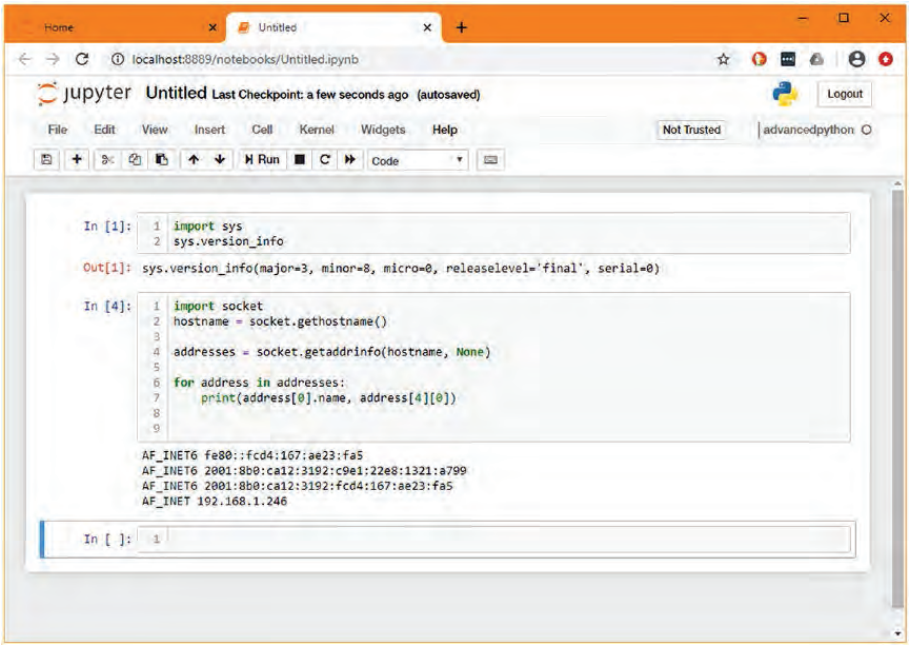

Рис. 1.6 ❖ Результат объединения ячеек, показанных на рис. 1.5

[1] На этих снимках экрана часть открытого маршрутизируемого IPv6-адреса вымарана.

Установка зависимостей

Более полезна информация о текущей загруженности системы. В Linux для ее получения следует прочитать значения из виртуального каталога /proc/loadavg, а в macOS выполнить команду `sysctl -n vm.loadavg`. В обеих системах она является частью вывода других программ, например uptime, но эта задача настолько часто встречается, что наверняка существует библиотека, которая нам поможет. Зачем дополнительная сложность, если ее можно избежать?

Сейчас мы установим первую зависимость `psutil`. Поскольку это библиотека, от которой зависит наш код, а не нужный нам инструмент разработки, следует опустить флаг `--dev`, который мы указывали при установке зависимостей ранее:

```
> pipenv install psutil
```

Примечание. Нам безразлично, какую версию `psutil` использовать, поэтому мы ее и не указываем. Команда `install` добавляет зависимость в файл `Pipfile`, а конкретную выбранную версию – в файл `Pipfile.lock`. Файлы с расширением `.lock` часто добавляются в множество игнорируемых системой управления версий. Но для `Pipfile.lock` следует сделать исключение, поскольку он помогает восстанавливать старые окружения и выполнять повторяемое развертывание.

Вернувшись в блокнот, мы должны перезапустить ядро и убедиться, что новая зависимость стала доступна. Выберите из меню **Kernel** команду **Restart**. Если вы предпочитаете работать с клавиатурой, то нажмите <ESCAPE>, чтобы выйти из режима редактирования (зеленая подсветка текущей ячейки при этом сменится синей) и дважды нажмите клавишу 0 (ноль).

После этого можно приступить к исследованию модуля `psutils`. Во второй ячейке импортируйте `psutil`:

```
import psutil
```

и нажмите **Run** (или клавиши <SHIFT+ENTER>), чтобы выполнить код в ячейке. В новой ячейке наберите `psutil.cpu<TAB>`[1]. Вы увидите список членов `psutil`, которые Jupyter мог бы выбрать в качестве продолжения. В данном случае подойдет `cpu_stats`, на него и нажмите. В этот момент можно нажать <SHIFT+TAB> – будет выведена минимальная документация по `cpu_stats`, из которой ясно, что эта функция не требует никаких аргументов.

Закончите ввод строки, так чтобы ячейки содержали такой текст:

```
import psutil

psutil.cpu_stats()
```

---

[1] Эта горячая клавиша работает, только если переменная известна ядру, поэтому прежде чем воспользоваться автозавершением, вам, возможно, придется выполнить ячейку, в которой переменная определена. Если вы присвоите переменной с тем же именем другое значение, то можете увидеть неправильную информацию, но во избежание путаницы я предостерегаю против такой практики.

Выполнив вторую ячейку, мы увидим, что `cpu_stats` дает маловразумительную информацию о том, как операционная система использует процессор. Попробуем вместо этого функцию `cpu_percent`. Нажав на ней <SHIFT+TAB>, мы увидим, что она принимает два необязательных параметра. Параметр `interval` определяет, сколько времени должна работать функция, прежде чем вернет управление; лучше, если он будет отличен от нуля. Поэтому изменим код, как показано ниже, и в ответ получим число с плавающей точкой от 0 до 100:

```
import psutil
psutil.cpu_percent(interval=0.1)
```

Упражнение 1.1: исследование библиотеки

В библиотеке `psutil` есть много других функций, которые могут служить хорошим источником информации, поэтому создадим по ячейке для каждой потенциально интересной функции. Состав множества функций зависит от операционной системы, поэтому имейте в виду, что если вы прорабатываете эту главу в Windows, то выбор функций будет довольно ограниченным.

Попробуйте предлагаемые Jupyter средства автозавершения и подсказки, чтобы оценить, какая информация может быть вам полезна, и создайте по крайней мере одну ячейку, возвращающую данные.

Включать импорт `psutil` в каждую ячейку избыточно, и в Python-файле такая практика порицается, но мы хотим, чтобы любую функцию можно было выполнить независимо от других. Для решения проблемы перенесем все предложения `import` в новую верхнюю ячейку, это эквивалентно области видимости модуля в обычном Python-файле.

После того как вы создадите дополнительные ячейки, играющие роль источников данных, блокнот будет выглядеть, как показано на рис. 1.7.

Заметьте, что числа в квадратных скобках слева от ячейки увеличиваются. Это число равно порядковому номеру операции. Число слева от первой ячейки остается постоянным, потому что эта ячейка не выполнялась, пока мы экспериментировали со следующими за ней.

В меню **Cell** (Ячейка) имеется команда **Run All** (Выполнить все), которая выполняет все ячейки по порядку, как если бы они находились в обычном Python-файле. Конечно, возможность выполнить все ячейки разом и тем самым протестировать блокнот целиком полезна, но, имея возможность выполнять ячейки по одной, мы можем разбить на части сложную и медленную программу, не выполняя ее каждый раз с самого начала.

Чтобы продемонстрировать, когда это может быть полезно, модифицируем пример с использованием функции `cpu_percent`. Мы выбрали интервал 0.1, потому что такого времени достаточно для получения точных данных. Задание большего интервала на практике не столь осмысленно, но поможет нам понять, как Jupyter позволяет написать дорогостоящий код инициализации, но при этом выполнять более быстрые части, не дожидаясь завершения работы медленных.

```
import psutil
psutil.cpu_percent(interval=5)
```

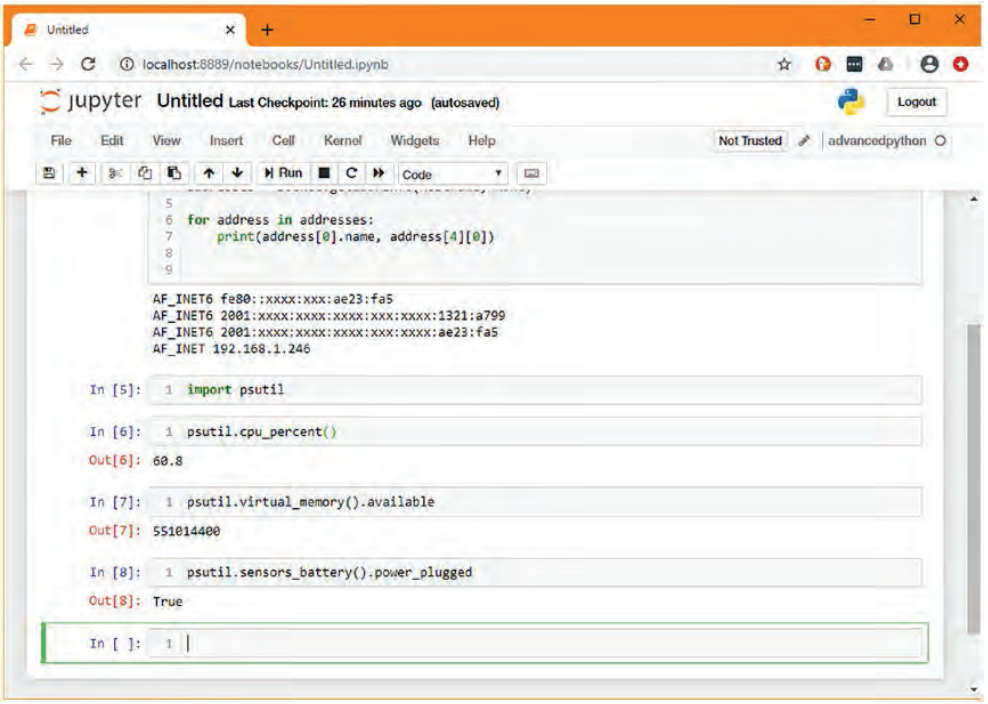

Рис. 1.7 ❖ Пример заполненного блокнота после выполнения упражнения

Экспорт в py-файл

Jupyter неплохо послужил нам в качестве средства прототипирования, но на роль основы проекта он не годится. Нам нужно традиционное Python-приложение, а замечательные презентационные возможности Jupyter пока ни к чему. В Jupyter встроена поддержка экспорта блокнотов в различных форматах, от слайд-шоу до HTML, но нам интересны скрипты на Python.

Преобразование в формат скрипта реализовано подкомандой `nbconvert` (notebook convert) команды Jupyter[1].

```
> jupyter nbconvert --to script Untitled.ipynb
```

Созданный нами безымянный блокнот остается неизменным, и создается новый файл `Untitled.py` (листинг 1.4). Если вы переименовывали блокнот, то имя файла будет соответствовать имени блокнота. Если нет, но хотите

[1] В IDE и редакторах, совместимых с блокнотами, аналогичная функциональность обычно доступна также прямо из окна редактора.

переименовать сейчас, поскольку не обратили внимания, что он назывался `Untitled.ipynb`, то щелкните по слову «Untitled» в начале блокнота и введите новое название.

Листинг 1.4 ❖ Файл Untitled.py, сгенерированный на основе созданного блокнота

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:

import sys
sys.version_info

# In[4]:

import socket
hostname = socket.gethostname()

addresses = socket.getaddrinfo(hostname, None)

for address in addresses:
    print(address[0].name, address[4][0])

# In[5]:

import psutil

# In[6]:

psutil.cpu_percent()

# In[7]:

psutil.virtual_memory().available

# In[8]:

psutil.sensors_battery().power_plugged

# In[ ]:
```

Как видите, ячейки разделены комментариями, а в начале файла присутствуют стандартные строки: путь к интерпретатору `#!...` и указание кодировки. То, что мы начали прототипирование в Jupyter, а не сразу в Python-скрипте или в оболочке REPL, никак не сказалось ни на гибкости, ни на потраченном времени, но зато позволило лучше контролировать выполнение отдельных блоков в процессе исследования.

Теперь можно убрать лишнее и превратить набор разрозненных предложений в служебный скрипт, для этого переместим предложения импорта в начало файла, а каждую ячейку превратим в именованную функцию. Комментарии `# In` отмечают начала ячеек и служат полезным напоминанием о том, где должна начинаться функция. Кроме того, мы должны преобразовать код, так чтобы каждая функция возвращала значение, а не просто вычисляла его (или печатала – в случае IP-адресов). Результат показан в листинге 1.5.

Листинг 1.5 ❖ serverstatus.py

```
# coding: utf-8
import sys
import socket

import psutil

def python_version():
    return sys.version_info

def ip_addresses():
    hostname = socket.gethostname()

    addresses = socket.getaddrinfo(hostname, None)
    address_info = []
    for address in addresses:
        address_info.append(address[0].name, address[4][0])
    return address_info

def cpu_load():
    return psutil.cpu_percent()

def ram_available():
    return psutil.virtual_memory().available
def ac_connected():
    return psutil.sensors_battery().power_plugged
```

Построение интерфейса командной строки

Эти функции сами по себе не особенно полезны, по большей части они просто обертывают уже существующие функции. Нам, в общем-то, нужно только напечатать возвращаемые ими данные, поэтому возникает законный вопрос: к чему было создавать однострочные обертки? Это станет понятным, когда мы создадим более сложные источники данных и различные способы их потребления, поскольку не придется заводить особые случаи для простейших источников. А пока, чтобы найти им применение, предложим пользователям простое приложение с командной строкой, которое будет отображать эти данные.

Раз мы работаем с автономным Python-скриптом, а не с чем-то, требующим установки, можно использовать идиому ifmain. Она встроена во многие текстовые редакторы для программистов и IDE в виде готового фрагмента, поскольку запомнить ее трудно и интуитивно она далеко не очевидна. Выглядит это так:

```
def do_something():
    print("Сделать что-то")

if __name__ == '__main__':
    do_something()
```

Код действительно страховидный. Переменная `__name__`[1] содержит ссылку на полное имя модуля. Если модуль импортирован, то атрибут `__name__` будет содержать путь к каталогу, в котором он находится.

```
>>> from json import encoder
>>> type(encoder)
<class 'module'>
>>> encoder.__name__
'json.encoder'
```

Но если код загружен в интерактивном сеансе или с указанием пути к исполняемому скрипту, то он никак не может быть импортирован. Поэтому такие модули получают специальное имя `__main__`. Идиома ifmain используется, чтобы распознать этот случай. В результате если модуль был указан в командной строке в качестве имени файла, то код внутри этого блока будет выполнен. А если модуль импортирован, то этот код *не* будет выполнен, потому что переменная `__name__` будет содержать путь к модулю. Не будь этого условия, обработчик командной строки исполнялся бы при каждом импорте модуля вместо программы, которая использует содержащиеся в модуле функции.

Предостережение. Поскольку код в ifmain-блоке исполняется, только если модуль является точкой входа в приложение, делайте его как можно короче. В идеале лучше бы оставить в нем всего одно предложение, вызывающее какую-то служебную функцию. Тогда эта функция допускает тестирование, это необходимо в некоторых приемах, которые мы будем рассматривать в следующей главе.

Модуль sys и переменная argv

В большинстве языков программирования имеется переменная `argv`, которая дает доступ к имени программы и аргументам, переданным ей при вызове. В Python это список строк, в котором первый элемент – имя скрипта (а не местонахождение интерпретатора Python), а следующие – аргументы программы.

Без анализа переменной `argv` можно писать только самые простые скрипты. Пользователь ожидает, что в командной строке, как минимум, будет флаг, позволяющий получить справку о программе. Кроме того, все программы, кроме совсем уж тривиальных, позволяют задать в командной строке конфигурационные параметры.

Реализовать эти требования проще всего, проверив, какие значения присутствуют в `sys.argv`, и обработав их в условных предложениях. Реализация флага справки может выглядеть, как показано в листинге 1.6.

---

[1] Обычно произносится «dunder main», поскольку «underscore» (подчерк), произнесенное четыре раза, содержит 12 согласных и звучит тяжеловато.

Листинг 1.6 ❖ sensors_argv.py – реализация интерфейса командной строки с ручной проверкой argv

```
#!/usr/bin/env python
# coding: utf-8

import socket
import sys

import psutil

HELP_TEXT = """порядок вызова: python {program_name:s}

Отображает значения датчиков

Флаги и аргументы:

--help: вывести это сообщение"""

def python_version():
    return sys.version_info

def ip_addresses():
    hostname = socket.gethostname()
    addresses = socket.getaddrinfo(socket.gethostname(), None)

    address_info = []

    for address in addresses:
        address_info.append((address[0].name, address[4][0]))
    return address_info

def cpu_load():
    return psutil.cpu_percent(interval=0.1)

def ram_available():
    return psutil.virtual_memory().available

def ac_connected():
    return psutil.sensors_battery().power_plugged

def show_sensors():
    print("Версия Python: {0.major}.{0.minor}".format(python_version()))
    for address in ip_addresses():
        print("IP-адреса: {0[1]} ({0[0]})".format(address))
    print("Загрузка ЦП: {:.1f}".format(cpu_load()))
    print("Доступная память: {} MiB".format(ram_available() / 1024**2))
    print("Кабель AC подключен: {}".format(ac_connected()))

def command_line(argv):
    program_name, *arguments = argv
    if not arguments:
        show_sensors()
    elif arguments and arguments[0] == '--help':
        print(HELP_TEXT.format(program_name=program_name))
        return
```

```
    else:
        raise ValueError("Неизвестные аргументы {}".format(arguments))

if __name__ == '__main__':
    command_line(sys.argv)
```

Функция `command_line(...)` не особенно сложна, но и программа очень простая. Легко представить себе ситуацию, когда имеется несколько флагов, которые могут быть заданы в любом порядке, а сами конфигурационные параметры имеют гораздо более сложную структуру. На практике только так и может быть, потому что никакого упорядочения или способа разбора значений не предполагается. В стандартной библиотеке имеется кое-какая вспомогательная функциональность, позволяющая создавать более сложные командные утилиты.

argparse

Модуль `argparse` – стандартный способ разбора аргументов командной строки, не зависящий от внешних библиотек. Он заметно упрощает обработку упомянутых выше сложных ситуаций, но, как и во многих других библиотеках, из которых может выбирать разработчик, интерфейс довольно трудно запомнить. Если вы не пишете командные утилиты регулярно, то, скорее всего, придется каждый раз заглядывать в документацию.

Модель, которой следует `argparse`, предполагает, что программист явно конструирует анализатор, создавая объект класса `argparse.ArgumentParser`, которому передается базовая информация о программе, а затем вызывает методы этого объекта для добавления возможных параметров. Эти методы определяют имя аргумента, текст справки, значение по умолчанию, а также способ его обработки анализатором. Одни аргументы являются простыми флагами, например `--dry-run`, другие аддитивны, например `-v`, `-vv`, `-vvv`, а третьи принимают явное значение, например `--config config.ini`.

В этой программе мы еще не используем параметры, поэтому пропустим их добавление и поручим анализатору разобрать аргументы из `sys.argv`. Результатом вызова функции будет информация, заданная пользователем. На этом этапе производится также простая обработка аргумента `--help` – вывод автоматически сгенерированной справки на основе добавленных аргументов.

После использования `argparse` наша программа принимает вид, показанный в листинге 1.7.

Листинг 1.7 ❖ sensors_argparse.py – интерфейс командной строки с использованием стандартного библиотечного модуля argparse

```
#!/usr/bin/env python
# coding: utf-8

import argparse
import socket
import sys
```

```
import psutil

def python_version():
    return sys.version_info

def ip_addresses():
    hostname = socket.gethostname()
    addresses = socket.getaddrinfo(socket.gethostname(), None)

    address_info = []
    for address in addresses:
        address_info.append((address[0].name, address[4][0]))
    return address_info

def cpu_load():
    return psutil.cpu_percent(interval=0.1)

def ram_available():
    return psutil.virtual_memory().available

def ac_connected():
    return psutil.sensors_battery().power_plugged

def show_sensors():
    print("Версия Python: {0.major}.{0.minor}".format(python_version()))
    for address in ip_addresses():
        print("IP-адреса: {0[1]} ({0[0]})".format(address))
    print("Загрузка ЦП: {:.1f}".format(cpu_load()))
    print("Доступная память: {} MiB".format(ram_available() / 1024**2))
    print("Кабель AC подключен: {}".format(ac_connected()))

def command_line(argv):
    parser = argparse.ArgumentParser(
        description='Отображает значения датчиков',
        add_help=True,
    )
    arguments = parser.parse_args()
    show_sensors()

if __name__ == '__main__':
    command_line(sys.argv)
```

click

Click – это дополнительный модуль, который упрощает создание интерфейсов командной строки в предположении, что интерфейс в общих чертах напоминает стандартный, ожидаемый пользователями. Применение этого модуля делает построение командного интерфейса более естественным, что поощряет создание интуитивно понятных интерфейсов.

Если `argparse` требует от программиста задавать допустимые параметры при конструировании анализатора, то `click` использует декораторы методов, чтобы понять, какие параметры допустимы. Этот подход чуть менее гибкий,

но охватывает 80 % типичных сценариев. При написании командного интерфейса мы обычно хотим походить на другие программы, а не удивлять конечного пользователя.

`click` – не часть стандартной библиотеки, поэтому его необходимо установить в наше окружение. Как и `psutil`, `click` является зависимостью программы, а не средством разработки, поэтому устанавливается следующим образом:

```
> pipenv install click
```

Поскольку в нашем примере команда не имеет параметров, при использовании `click` нужно добавить в код всего две строчки: импорт и декоратор `@click.command(...)`. Все вызовы `print(...)` заменяются на `click.echo(...)`, хотя это необязательно. Результат показан в листинге 1.8. `click.echo` – вспомогательная функция, которая ведет себя как `print`, но при этом обрабатывает несоответствие кодировок символов и автоматически убирает или сохраняет маркеры цвета и форматирования в зависимости от возможностей терминала, на котором выполнена программа, и с учетом перенаправления вывода по конвейеру.

Листинг 1.8 ❖ sensors_click.py – интерфейс командной строки с использованием сторонней библиотеки click

```
#!/usr/bin/env python
# coding: utf-8
import socket
import sys

import click
import psutil

def python_version():
    return sys.version_info

def ip_addresses():
    hostname = socket.gethostname()
    addresses = socket.getaddrinfo(socket.gethostname(), None)

    address_info = []
    for address in addresses:
        address_info.append((address[0].name, address[4][0]))
    return address_info

def cpu_load():
    return psutil.cpu_percent(interval=0.1)

def ram_available():
    return psutil.virtual_memory().available

def ac_connected():
    return psutil.sensors_battery().power_plugged

@click.command(help="Отображает значения датчиков")
```

```
def show_sensors():
    click.echo("Версия Python: {0.major}.{0.minor}".format(python_version()))
    for address in ip_addresses():
        click.echo("IP-адреса: {0[1]} ({0[0]})".format(address))
    click.echo("Загрузка ЦП: {:.1f}".format(cpu_load()))
    click.echo("Доступная память: {} MiB".format(ram_available() / 1024**2))
    click.echo("Кабель AC подключен: {}".format(ac_connected()))

if __name__ == '__main__':
    show_sensors()
```

В библиотеке есть еще много функций, упрощающих создание более сложных интерфейсов и исправляющих поведение программы на нестандартных терминалах в системе конечного пользователя. Например, если мы захотим печатать полужирным шрифтом заголовки в команде `show_sensors`, то можем воспользоваться командой `secho(...)`, которая форматирует вывод. Версия со стилями показана в листинге 1.9.

Листинг 1.9 ❖ Фрагмент файла sensors_click_bold.py

```
@click.command(help="Displays the values of the sensors")
def show_sensors():
    click.secho("Версия Python: ", bold=True, nl=False)
    click.echo("{0.major}.{0.minor}".format(python_version()))
    for address in ip_addresses():
        click.secho("IP-адреса: ", bold=True, nl=False)
        click.echo("{0[1]} ({0[0]})".format(address))
    click.secho("Загрузка ЦП: ", bold=True, nl=False)
    click.echo("{:.1f}".format(cpu_load()))
    click.secho("Доступная память: ", bold=True, nl=False)
    click.echo("{} MiB".format(ram_available() / 1024**2))
    click.secho("Кабель AC подключен: ", bold=True, nl=False)
    click.echo("{}".format(ac_connected()))
```

Функция `secho(...)` выводит на экран отформатированную информацию. Аргумент `nl=` говорит, нужно ли печатать символ новой строки. Без `click` сделать то же самое можно было бы следующим образом:

```
BOLD = '\033[1m'
END = '\033[0m'
def show_sensors():
    print(BOLD + "Версия Python:" + END + " ({0.major}.{0.minor})".
    format(python_version()))
    for address in ip_addresses():
        print(BOLD + "IP-адреса: " + END + "{0[1]} ({0[0]})".
        format(address))
    print(BOLD + "Загрузка ЦП:" + END + " {:.1f}".format(cpu_load()))
    print(BOLD + "Доступная память:" + END + "{} MiB".format(ram_available() / 1024**2))
    print(BOLD + "Кабель AC подключен:" + END + " {}".format(ac_connected()))
```

Кроме того, `click` прозрачно поддерживает автозавершение на терминале и много других полезных функций. Мы еще вернемся к этому модулю позже, когда расширим командный интерфейс.

Расширение границ возможного

Мы рассмотрели использование Jupyter и IPython для прототипирования, но иногда возникает необходимость выполнить код прототипа на конкретном компьютере, а не на том, где мы обычно занимаемся разработкой. Например, потому что к этому компьютеру подключено нужное нам периферийное устройство или установлена некоторая программа.

Это может вызвать дискомфорт: иногда редактировать и запускать код на удаленной машине просто неудобно, а иногда очень трудно, особенно если операционные системы различаются.

В предыдущих примерах весь код запускался локально. Но мы планируем выполнять окончательный код на Raspberry Pi, потому что именно к нему будут подключены специальные датчики. Это встраиваемая система, поэтому имеются существенные аппаратные отличия – с точки зрения как производительности, так и состава периферии.

Удаленные ядра

Для тестирования этого кода необходимо запустить окружение Jupyter на Raspberry Pi, подключиться к нему по HTTP или по SSH и вручную взаимодействовать с интерпретатором Python. Это неудобно, поскольку на Raspberry Pi требуется открыть порты, к которым может привязаться Jupyter, и вручную синхронизировать содержание блокнотов на локальной и удаленной машинах с помощью какой-нибудь программы типа `scp`. На практике эта проблема еще серьезнее. Трудно представить себе, что кто-то откроет порт на сервере, чтобы мы могли подключиться к Jupyter и тестировать код анализа журналов.

Вместо этого можно использовать инфраструктуру сменных ядер в Jupyter и IPython, чтобы подключить локально работающий Jupyter-блокнот к одному из многих удаленных компьютеров. Это позволяет прозрачно тестировать один и тот же код на нескольких машинах при минимуме ручной работы.

При отображении списка потенциальных целей выполнения Jupyter включает в него известные *спецификации ядра*. Если выбрана спецификация ядра, то создается *экземпляр* этого ядра, который связывается с блокнотом. Можно подключиться к удаленной машине и вручную запустить ядро, к которому подключится локальный экземпляр Jupyter. Однако такой способ работы редко бывает эффективным. Когда в начале этой главы мы выполнили команду `pipenv run ipython kernel install`, мы создали новую спецификацию ядра для текущего окружения и установили ее в список известных спецификаций.

Чтобы добавить спецификации ядер для удаленных машин, мы можем воспользоваться утилитой `remote_ikernel`. Ее нужно установить туда же, где находится Jupyter, потому что это ассистент Jupyter, а не специальное средство разработки в данном окружении.

```
> pip install --user remote_ikernel
```

Затем нужно подготовить окружение и вспомогательную программу ядра на удаленной машине. Подключимся к Raspberry Pi (или другой машине, на

которую мы хотим отправить код) и создадим pipenv на этом компьютере, как делали раньше:

```
rpi> python -m pip install --user pipenv
rpi> mkdir development-testing
rpi> cd development-testing
rpi> pipenv install ipykernel
```

Совет. На некоторых низкопроизводительных машинах, в частности на Raspberry Pi, установка ipython_kernel может продолжаться мучительно долго. В таком случае можно рассмотреть возможность установки версии ipython_kernel, предлагаемой диспетчером пакетов. ipython_kernel действительно требует много дополнительных библиотек, из-за чего установка на слабый компьютер работает долго. В таком случае подготовить окружение можно следующим образом:

```
rpi> sudo apt install python3-ipykernel
rpi> pipenv --three --site-packages
```

С другой стороны, если вы работаете с Raspberry Pi, то на сайте https://www.piwheels.org существует репозиторий уже откомпилированных пакетов (в формате wheel), который можно подключить, добавив еще один источник в файл Pipfile:

```
[[source]]
url = "https://www.piwheels.org/simple"
name = "piwheels"
verify_ssl = true
```

После этого пакет ipython_kernel устанавливается, как обычно, командой pipenv install. Если вы используете Raspberry Pi под управлением Raspbian, то всегда следует добавлять piwheels в Pipfile, поскольку Raspbian уже глобально сконфигурирована для работы с PiWheels. Если не добавить этот источник в Pipfile, то установка пакетов может завершаться аварийно.

Итак, мы установили ядро IPython на машину Raspberry Pi, но еще нужно установить его и на нашу локальную машину. Для начала установим ядро, указывающее на созданное нами окружение pipenv. После этого на Raspberry Pi будет доступно два ядра: одно для системной установки Python и другое, называемое ядром разработки и тестирования, для нашего окружения. После установки ядра мы можем просмотреть конфигурационный файл спецификации:

```
rpi> pipenv run ipython kernel install --user --name=development-testing
Installed kernelspec development-testing in /home/pi/.local/share/jupyter/
kernels/development-testing
> cat /home/pi/.local/share/jupyter/kernels/development-testing/kernel.json
 {
 "argv": [
  "/home/pi/.local/share/virtualenvs/development-testing-nbi70cWI/bin/python",
  "-m",
  "ipykernel_launcher",
  "-f",
```

```
  "{connection_file}"
 ],
 "display_name": "development-testing",
 "language": "python"
}
```

Отсюда видно, как Jupyter стал бы выполнять ядро, если бы оно было установлено на этом компьютере. Мы можем воспользоваться информацией из этой спецификации для создания новой спецификации remote_ikernel на нашей машине разработки, которая будет указывать на то же окружение, что ядро разработки и тестирования на Raspberry Pi.

В приведенной выше спецификации ядра указано, как ядро запускается на Raspberry Pi. Мы можем проверить это, выполнив команду по SSH-соединению с Raspberry Pi, например заменив `-f {connection_file}` на `--help`, чтобы вывести текст справки.

```
rpi> /home/pi/.local/share/virtualenvs/development-testing-nbi70cWI/bin/
python -m ipykernel -help
```

Теперь можно вернуться на машину разработки и создать спецификацию удаленного ядра:

```
> remote_ikernel manage --add --kernel_cmd="/home/pi/.local/share/
virtualenvs/development-testing-nbi70cWI/bin/python
-m ipykernel_launcher -f {connection_file}"
--name="development-testing" --interface=ssh --host=pi@raspberrypi
--workdir="/home/pi/developmenttesting" --language=python
```

Выглядит пугающе, все-таки целых пять строк текста. Но эту команду можно разделить на части:

- параметр `--kernel_cmd` – содержание раздела `argv` из файла спецификации ядра. Его значение – строка, в которой части разделены пробелами и отсутствуют внутренние кавычки. Это команда, запускающая само ядро;
- параметр `--name` эквивалентен параметру `display_name` из исходной спецификации ядра. Именно эту строку показывает Jupyter вместе с информацией SSH, когда вы выбираете ядро. Это имя не обязано совпадать с именем удаленного ядра, из которого оно скопировано, приводится только для справки;
- параметры `--interface` и `--host` определяют, как подключаться к удаленной машине. Необходимо сделать так, чтобы к машине был возможен доступ по SSH без ввода пароля[1], чтобы Jupyter мог подключиться автоматически;
- параметр `--workdir` – рабочий каталог, который окружение будет использовать по умолчанию. Рекомендую указывать каталог, содержащий удаленный файл Pipfile;

---

[1] Используйте `ssh-copy-id user@host`, чтобы настроить это автоматически, а не редактируйте вручную файл `authorized_hosts`.

- параметр `--language` – язык, указанный в исходной спецификации ядра, он позволяет различать языки программирования.

Совет. Если возникают трудности с подключением к удаленному ядру, попробуйте открыть оболочку, запустив Jupyter из командной строки. Часто при этом выдаются полезные сообщения об ошибках. Найдите имя ядра в списке, который выводит `jupyter kernelspec list`, и укажите его в команде:

```
> jupyter kernelspec list
Available kernels:
advancedpython
C:\Users\micro\AppData\Roaming\jupyter\kernels\advancedpython
rik_ssh_pi_raspberrypi_developmenttesting
C:\Users\micro\AppData\Roaming\jupyter\kernels\
rik_ssh_pi_raspberrypi_developmenttesting
> jupyter console --kernel= rik_ssh_pi_raspberrypi_developmenttesting
In [1]:
```

Если теперь повторно войти в окружение Jupyter, то мы увидим новое ядро, которое соответствует добавленной информации о подключении. Мы можем выбрать это ядро и выполнять команды, требующие такого окружения[1], а ядро Jupyter позаботится о подключении к Raspberry Pi и активации окружения в каталоге `~/development-testing`.

Разработка кода, который невозможно выполнить локально

В системе Raspberry Pi имеются полезные датчики, которые дают интересующие нас данные. В других случаях это может быть информация, собираемая специальными командными утилитами, анализ базы данных или выполнение вызовов локальных API.

В этой книге нас не интересует, как извлечь максимум пользы из Raspberry Pi, поэтому детали ее работы мы опустим. Скажем лишь, что существует много документации и людей, готовых объяснить, как с помощью Python делать потрясающие вещи. Мы же воспользуемся библиотекой, которая содержит функцию, возвращающую данные о температуре и относительной влажности, получаемую с подключенного к плате датчика. Как и во многих других случаях, измерение производится довольно медленно (около секунды), и для решения задачи нужно определенное окружение (наличие внешнего датчика). В этом смысле можно провести аналогию с мониторингом активных процессов на веб-сервере путем подключения к их портам управления.

---

[1] Если вы предпочитаете консольное окружение веб-окружению Jupyter-блокнота, то можете просмотреть список доступных ядер с помощью команды `jupyter kernelspec list` и открыть оболочку IPython, подключенную к выбранному ядру, командой `jupyter console --kernel kernelname`.

Для начала добавим в наше окружение библиотеку Adafruit DHT[1]. В данный момент у нас уже есть копии файла Pipfile на Raspberry Pi и на локальной машине. Удаленная копия содержит только зависимость от `ipykernel`, которая уже имеется в локальной копии, поэтому можно, ничего не опасаясь, заменить удаленный файл созданным локально. Поскольку мы знаем, что библиотека DHT полезна лишь для работы с Raspberry Pi, можно ввести ограничение: устанавливать ее только на машинах под управлением Linux с процессорами ARM, для чего применяется синтаксис условной зависимости[2]:

```
> pipenv install "Adafruit-CircuitPython-DHT ; 'arm' in platform_machine"
```

В результате в файлы `Pipfile` и `Pipfile.lock` будет добавлена эта зависимость. Мы хотим воспользоваться ей на удаленной машине, поэтому должны скопировать на нее файлы и установить их с помощью Pipenv. Вообще-то эту команду можно запускать в обоих окружениях, но по неосторожности могут вкрасться ошибки. Программа Pipenv предполагает, что для разработки и развертывания используется одна и та же версия Python, следуя заложенной в ней идее избегать проблем на этапе развертывания. Поэтому, если вы планируете развертывать систему с определенной версией Python, то должны использовать ее и для локальной разработки.

Но если вы не хотите устанавливать необычные версии Python в свое локальное окружение или ведете разработку для разных машин, то эту проверку можно подавить. Для этого удалите строчку `python_version` в конце файла Pipfile. Это позволит развертывать в окружении любую версию Python. Однако вы должны понимать, какие версии поддерживаете, и соответственно организовать тестирование.

Скопируйте файлы Pipfile и Pipfile.lock на удаленную машину с помощью `scp` (или любого другого инструмента по своему выбору) и выполните на удаленной машине команду `pipenv install` с флагом `--deploy`. Этот флаг означает, что pipenv может продолжать работу, только если версии совпадают, что очень полезно для переноса заведомо работоспособного окружения с одной машины на другую.

```
rpi> cd /home/pi/development-testing
rpi> pipenv install --deploy
```

Однако имейте в виду, что если вы создавали Pipfile в другой операционной системе или на машине с иной архитектурой процессора (например, файлы создавались на стандартном ноутбуке, а устанавливаются на Raspberry Pi), то может случиться, что закрепленные пакеты непригодны для развертывания на другой машине. В таком случае можно изменить закрепление

---

[1] Это часть великолепной экосистемы CircuitPython от компании Adafruit. В различных проектах на сайте https://learn.adafruit.com/dht рассказано гораздо больше об этих датчиках и способах их использования.

[2] Этот синтаксис определен в документе PEP508 по адресу www.python.org/dev/peps/pep-0508/. Там имеется таблица допустимых фильтров, которая в будущем может быть расширена.

зависимостей без перехода на новые версии, выполнив команду `pipenv lock --keep-outdated`.

Теперь в удаленном окружении есть заданные нами зависимости. Если вы изменяли закрепление, то скопируйте и сохраните измененный lock-файл, чтобы в будущем можно было развернуть его повторно, не перегенерируя. В этот момент вы можете подключиться к удаленному серверу с помощью клиента Jupyter и приступить к прототипированию. Нас интересует добавление датчика влажности, поэтому воспользуемся только что установленной библиотекой и посмотрим, как получить значение относительной влажности.

На компьютере Raspberry Pi, на который я скопировал эти файлы, датчик DHT22 подключен к контакту D4, как показано на рис. 1.8. Этот датчик можно приобрести у производителя Raspberry Pi или у какого-нибудь поставщика электронного оборудования. Если его нет под рукой, то попробуйте какую-нибудь другую команду, которая доказывает, что программа работает на Pi, например `platform.uname()`.

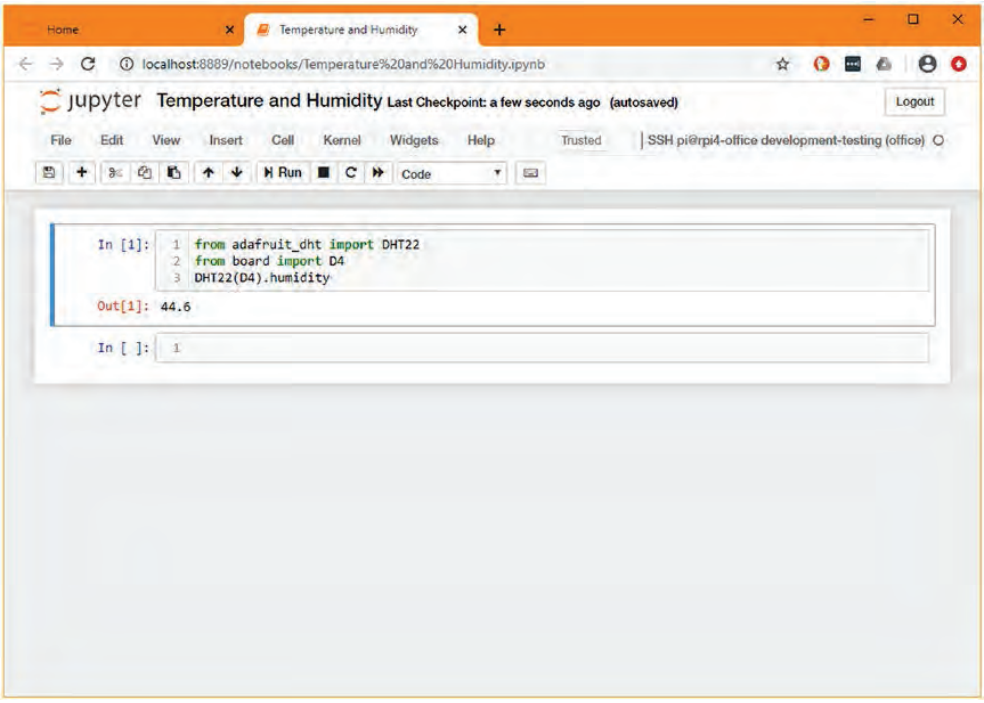

Рис. 1.8 ❖ Jupyter подключен к удаленному компьютеру Raspberry Pi

Этот блокнот хранится на вашей локальной машине разработки, а не на удаленном сервере. Его можно преобразовать в Python-скрипт командой `nbconvert` так же, как мы делали раньше. Но прежде изменим ядро, восстановив локальный экземпляр, чтобы проверить, что код правильно работает

в нем. Наша цель – написать код, работающий в обоих окружениях: он должен возвращать либо влажность, либо некое фиктивное значение.

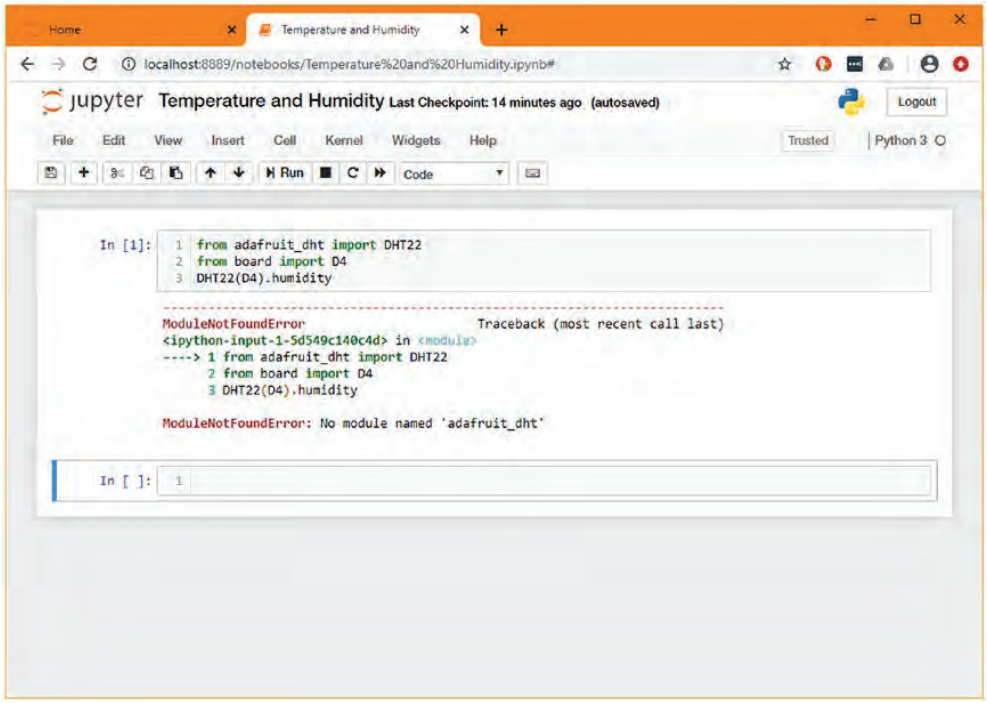

Рис. 1.9 ❖ Тот же код на локальной машине

На рис. 1.9 видно, что этот код работает не во всех окружениях. Мы очень хотели бы, чтобы по крайней мере часть кода могла работать локально, поэтому подправим его с учетом ограничений на других платформах. После преобразования в более общую функцию код выглядит так:

```
def get_relative_humidity():
    try:
        # Подключиться к датчику DHT22 на контакте GPIO 4
        from adafruit_dht import DHT22
        from board import D4
    except (ImportError, NotImplementedError):
        # Если библиотеки DHT нет, возбуждается исключение ImportError.
        # Запуск на неизвестной платформе приводит к исключению NotImplementedError
        # при попытке обратиться к контакту
        return None
    return DHT22(D4).humidity
```

Теперь функцию можно вызывать на любой машине, но если отсутствует датчик, подключенный к контакту `D4`, то она вернет `None`.

Окончательный скрипт

В листинге 1.10 приведен окончательный скрипт. Нам еще предстоит устранить некоторые трудности, если мы хотим сделать эту библиотеку полезной, и прежде всего избавиться от форматирования значений в функции `show_sensors`. Нам не нужно, чтобы источники осуществляли форматирование, поскольку мы хотим предоставлять исходные данные другим интерфейсам. Этой задачей мы займемся в следующей главе.

Листинг 1.10 ❖ Окончательная версия скрипта в этой главе

```
#!/usr/bin/env python
# coding: utf-8
import socket
import sys

import click
import psutil

def python_version():
    return sys.version_info

def ip_addresses():
    hostname = socket.gethostname()
    addresses = socket.getaddrinfo(socket.gethostname(), None)
    address_info = []
    for address in addresses:
        address_info.append((address[0].name, address[4][0]))
    return address_info

def cpu_load():
    return psutil.cpu_percent(interval=0.1) / 100.0

def ram_available():
    return psutil.virtual_memory().available

def ac_connected():
    return psutil.sensors_battery().power_plugged

def get_relative_humidity():
    try:
        # Подключиться к датчику DHT22 на контакте GPIO 4
        from adafruit_dht import DHT22
        from board import D4
    except (ImportError, NotImplementedError):
        # Если библиотеки DHT нет, возбуждается исключение ImportError.
        # Запуск на неизвестной платформе приводит к исключению NotImplementedError
        # при попытке обратиться к контакту
        return None
    return DHT22(D4).humidity

@click.command(help="Отображает значения датчиков")
```

```
def show_sensors():
    click.echo("Версия Python: {0.major}.{0.minor}".format(python_version()))
    for address in ip_addresses():
        click.echo("IP-адреса: {0[1]} ({0[0]})".format(address))
    click.echo("Загрузка ЦА: {:.1%}".format(cpu_load()))
    click.echo("Доступная память: {:.0f} MiB".format(ram_available() / 1024**2))
    click.echo("Кабель АС подключен: {!r}".format(ac_connected()))
    click.echo("Влажность: {!r}".format(get_relative_humidity()))

if __name__ == '__main__':
    show_sensors()
```

Резюме

На этом мы завершаем главу о прототипировании. В следующих главах мы воспользуемся разработанными здесь функциями извлечения данных для создания библиотек и инструментов в соответствии с передовой практикой программирования на Python. Мы прошли путь от экспериментов с библиотекой до создания работоспособного скрипта, представляющего реальную ценность. Далее мы изменим его так, чтобы он лучше отвечал конечной цели – агрегированию распределенных данных.

Полученные знания могут пригодиться на многих этапах жизненного цикла разработки программного обеспечения, но важно не терять гибкость, тупо следуя единственному процессу. Хотя эти методы эффективны, иногда открытие оболочки REPL, использование pdb или даже простые вызовы `print(...)` могут оказаться проще, чем настройка удаленного ядра. Невозможно выбрать лучший путь решения проблемы, не зная об альтернативах.

Подведем итоги.

1. Jupyter – отличный инструмент для изучения библиотек и создания начального прототипа программы.
2. Для Python существуют специальные отладчики, которые легко встраиваются в технологический процесс с помощью функции `breakpoint()` и переменных окружения.
3. Pipenv помогает определить требования к версиям и вовремя загружать актуальные версии при минимальном объеме спецификаций. Эта программа обеспечивает воспроизводимые сборки.
4. Библиотека `click` позволяет легко создавать на Python интерфейсы командной строки в идиоматичном стиле.
5. Система ядер в Jupyter обеспечивает органичную интеграцию в единый процесс разработки на различных языках программирования, а также локальное и удаленное выполнение.

Дополнительные ресурсы

Рассказывая в этой главе о различных инструментах, мы лишь пробежались по верхам, решая свои непосредственные задачи.

- Документация по pipenv по адресу https://pipenv.pypa.io/en/latest/ содержит массу полезных советов о том, как настроить pipenv под себя, особенно в части настройки виртуального окружения и интеграции в существующие процессы. Если вы раньше не сталкивались с pipenv, но много работали с виртуальными средами, то там же найдете хорошую документацию по переходу на эту систему.
- Если вас интересует прототипирование на других языках программирования в Jupyter, то рекомендую ознакомиться с документацией по Jupyter по адресу https://jupyter.readthedocs.io/en/latest/, особенно с разделом, посвященным ядрам.
- О Raspberry Pi и совместимых датчиках рекомендую почитать документацию в проекте CircuitPython по адресу https://learn.adafruit.com/circuitpython-onraspberrypi-linux.