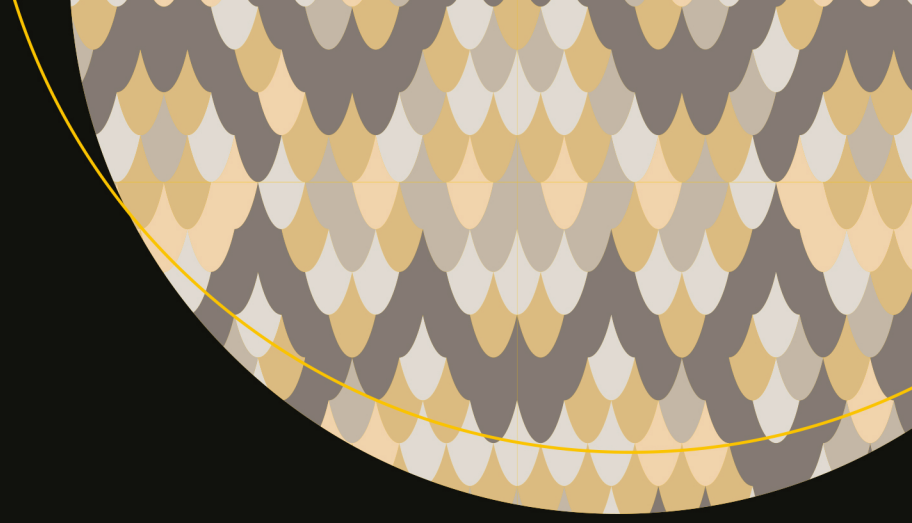


Профессиональная разработка на Python

Использование эффективных средств языка в реальных приложениях

Мэттью Уилкс

ДМК Пресс
издательство

Мэттью Уилкс

Профессиональная разработка на Python

Advanced Python Development

Using Powerful Language Features in Real-World Applications

Matthew Wilkes

Apress®

Профессиональная разработка на Python

Использование эффективных средств языка в реальных приложениях

Мэттью Уилкс

ДМК ПРЕСС
издательство
Москва, 2021

УДК 004.94
ББК 32.972
У36

Уилкс М.
У36 Профессиональная разработка на Python / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2021. – 502 с.: ил.

ISBN 978-5-97060-930-9

В этой книге объясняются языковые средства Python, которые обычно не рассматриваются в пособиях: от повторно используемых консольных скриптов, которые одновременно играют роль микросервисов благодаря точкам входа, до эффективного использования модуля asyncio для объединения данных из различных источников. Попутно рассматривается проверка соблюдения стандартов кодирования с помощью аннотаций типов, тестирование с низкими накладными расходами и другие автоматизированные проверки качества кода, применяемые на практике для организации процесса разработки надежного ПО.

Некоторые мощные возможности Python зачастую иллюстрируются на искусственных примерах, когда то или иное средство описывается в изоляции от всего остального. Здесь же на примере проектирования и создания реального приложения от прототипа до готового продукта читатель видит не только, как работают различные части программы, но и как они интегрируются в процессе разработки более крупной системы. Кроме того, в книге присутствуют интересные отступления и рекомендации по использованию библиотек, взятые из сессий вопросов и ответов на конференциях по Python, а также обсуждение современных передовых практик и методов, позволяющих создавать ясный и удобный для сопровождения код.

Эта книга ориентирована на разработчиков, которые уже умеют писать простые программы на Python и хотят разобраться в том, когда уместно использовать новые прогрессивные средства языка.

УДК 004.94
ББК 32.972

ISBN 978-1-4842-5792-0 (англ.)
ISBN 978-5-97060-930-9 (рус.)

Содержание

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.

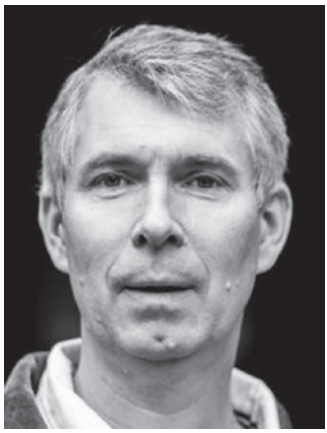
Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

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.