
ПРОГРАММИРУЕМ
НА PYTHON
ТИМУР МАШНИН

Тимур Машнин

Программируем на Python

«Автор»
2026

Машнин Т.

Программируем на Python / Т. Машнин — «Автор», 2026

Эта книга обучает основам написания программ на Python с использованием наиболее распространенных структур. Вы освоите базовый синтаксис Python; изучите различные редакторы кода и среды разработки; сможете писать программы на Python; разберетесь, как взаимодействуют строительные блоки программирования; и сможете объединить все эти знания для решения сложных задач программирования. Книга также охватывает автоматическое тестирование и объясняет, как настроить среду разработки на персональном компьютере. Также вы научитесь отслеживать различные версии вашего кода и конфигурационных файлов с помощью популярной системы контроля версий Git. Вы познакомитесь с инструментами для быстрого выявления и решения реальных проблем, с которыми вы можете столкнуться в своей ИТ-деятельности. Вы раскроете потенциал автоматизации управления парком компьютеров, открывая путь к тому, чтобы быть готовым запускать и поддерживать современные облачные приложения и системы.

© Машнин Т., 2026
© Автор, 2026

Содержание

Подготовка к работе 5
Введение в программирование 8
Переменные и типы 23
Условные операторы 33
Циклы 41
Рекурсия 54
Обработка исключений 57
Строки 60
Введение в структуры данных Python 70
Добавление и извлечение данных из структур данных 89
Функции 97
Создание функций 115
Создание классов и методов 118
Модули 131
Управление файлами с помощью Python 141
Регулярные выражения 157
Управление данными и процессами 174
Тестирование 190
Конец ознакомительного фрагмента. 192

Тимур Машнин
Программируем на Python

Подготовка к работе

Python был создан в начале 1990-х годов Гвидо ван Россумом в Stichting Mathematich Centrum в Нидерландах как преемник языка под названием ABC.

В 1995 году Гвидо продолжил свою работу над Python в Корпорации национальных исследовательских инициатив в Рестоне, штат Вирджиния, где он выпустил несколько версий программного обеспечения.

В мае 2000 года Гвидо и основная группа разработчиков Python переехали на BeOpen.com, чтобы сформировать команду BeOpen PythonLabs. В октябре того же года команда PythonLabs переехала в Digital Creations (ныне Zope Corporation). В 2001 году был создан Фонд программного обеспечения Python (PSF), некоммерческая организация, созданная специально для владения интеллектуальной собственностью, связанной с Python. Zope Corporation является членом-спонсором PSF.

Все выпуски Python имеют открытый исходный код. Исторически сложилось так, что большинство (но не все) выпусков Python также были совместимы с GPL (GNU General Public License; лицензия на свободное программное обеспечение, созданная в рамках проекта GNU в 1988 г., по которой автор передаёт программное обеспечение в общественную собственность).

Почему язык называется Python? Когда Гвидо ван Россум начал работать над Python, он также читал опубликованные сценарии из комедийного сериала BBC «Летающий цирк Монти Пайтона» 1970-х годов. Ван Россум подумал, что ему нужно короткое, уникальное и немного загадочное имя, поэтому он решил назвать язык Python.

Python используется для веб-разработки (серверная часть), разработки программного обеспечения, системного администрирования, в математике и науки о данных.

Python можно использовать для создания веб-приложений, для быстрого прототипирования и для разработки программного обеспечения, готового к производству, для создания рабочих процессов. С помощью Python можно подключаться к системам баз данных, читать и изменять файлы.

Python можно использовать для обработки больших данных и выполнения сложных математических вычислений.

Python работает на разных платформах (Windows, Mac, Linux, Raspberry Pi и т. д.) и Python имеет простой синтаксис, похожий на английский язык. Синтаксис Python позволяет разработчикам писать программы с меньшим количеством строк, чем в некоторых других языках программирования.

Python использует новые строки для завершения команды, в отличие от других языков программирования, которые часто используют точки с запятой или круглые скобки.

Также Python использует отступы с использованием пробелов для определения области действия; такие как область действия циклов, функций и классов. Другие языки программирования часто используют для этой цели фигурные скобки.

Python работает в системе интерпретатора, а это означает, что код может быть выполнен сразу после его написания. Это означает, что прототипирование может быть очень быстрым.

И Python можно использовать процедурным, объектно-ориентированным или функциональным способом.

Самая последняя основная версия Python — Python 3, однако Python 2, хотя больше и не обновляется, кроме обновлений безопасности, по-прежнему довольно популярен.

Программы на Python можно писать в текстовом редакторе, в интегрированной среде разработки, такой как Thonny, Pycharm, Netbeans или Eclipse, что особенно полезно при управлении большими коллекциями файлов Python, а также использовать Jupyter Notebook.

Для установки Python перейдите на сайт https://www.python.org/downloads/ и следуйте инструкциям.

Jupyter Notebook – — это серверно-клиентское приложение, которое позволяет редактировать и запускать различного рода документы через веб-браузер. Приложение Jupyter Notebook может быть запущено на локальном рабочем столе, не требующем доступа к Интернету, или может быть установлено на удаленном сервере и доступно через Интернет.

Помимо отображения, редактирования, запуска документов, приложение Jupyter Notebook имеет панель управления (Notebook Dashboard), показывающую локальные файлы и позволяющую открывать документы и запускать их выполнение.

Двадцать лет назад Фернандо Перес был аспирантом, получавшим докторскую степень по физике элементарных частиц в Университете Колорадо в Боулдере. Он искал интерактивный инструмент с открытым исходным кодом для анализа данных своего научного моделирования. Он также только что выучил язык программирования Python и очень хотел применить его в своем рабочем процессе.

Затем однажды днем, Перес разработал свой собственный интерпретатор Python для интерактивных научных исследований и обработки данных. Хотя IPython изначально разрабатывался как командная оболочка для интерактивных вычислений на языке программирования Python, сегодня он работает на нескольких языках программирования, предлагая самоанализ, мультимедийные возможности, синтаксис оболочки, завершение табуляции и историю.

«Причина рождения IPython на самом деле заключалась в том, что я хотел использовать код Python, но хотел использовать его в интерактивном рабочем процессе, где можно выполнять небольшой код, просматривать данные, просматривать файлы, выполнять итерации и изучать данные», — сказал Перес, – «Этот процесс итераций и исследований очень естественен в науке. В исследованиях у нас обычно нет заранее определенного набора требований, когда кто-то говорит нам: вот программное обеспечение, которое мне нужно, идите и создайте мне его, это скорее похоже на то, что у нас есть вопрос, который мы пытаемся понять, и некоторые данные, которые мы хотим получить для анализа.

И поскольку IPython — это приложение с открытым исходным кодом, оно быстро стало популярным и стало использоваться многими учеными.

Многие исследователи не просто использовали IPython; они также вложили свое время и усилия в добавление функций, которые сделали инструмент еще лучше. В 2004 году Перес начал сотрудничать с физиками Брайаном Грейнджером, Мин Рэган-Келли и другими над дальнейшим развитием IPython. Эти совместные усилия дали несколько прототипов Notebooks для IPython, а в 2011 году команда выпустила первую версию продукта, который стал успешным – IPython Notebook на базе браузера. Эта версия заложила основу для сегодняшнего проекта Jupyter и стала началом революции в науке о данных.

Как и другие вычислительные блокноты, IPython Notebook объединяет коды, результаты, графику и текст в одном документе, но, в отличие от некоторых своих предшественников, он имеет открытый исходный код, что привлекает к нему сообщество разработчиков. В 2014 году IPython Notebook превратился в Project Jupyter и в конечном итоге стал поддерживать около 100 языков, позволяя пользователям исследовать данные на удаленных суперкомпьютерах, а также на своих ноутбуках.

В будущем Перес видел возможности для инструментов с открытым исходным кодом, таких как Project Jupyter, в сочетании с облачными вычислениями и суперкомпьютерами, чтобы расширить возможности крупномасштабных сообществ, где собираются и растут рас-

пределенные исследовательские коллективы для решения больших общих проблем, таких как изменение климата или решения глобальных проблем, таких как пандемия COVID-19.

Последняя версия интерактивной веб-среды разработки для блокнотов, кода и данных проекта Jupyter – это инструмент JupyterLab, который вы можете установить с помощью инструмента pip в командной строке.

pip install jupyterlab

Инструмент pip – это установщик пакетов (библиотек) для Python.

Введение в программирование

На базовом уровне компьютерная программа — это набор инструкций, который указывает вашему компьютеру, что делать. Когда вы пишете программу, вы создаёте пошаговый рецепт того, что нужно сделать для выполнения задачи, и когда ваш компьютер выполняет программу, он читает то, что вы написали, и следует вашим инструкциям в точности.

Рецепт написан на языке программирования, который называется языком программирования. Языки программирования на самом деле похожи на разговорные языки людей, поскольку они имеют синтаксис и семантику. Теперь, если прошло много времени с вашего последнего урока грамматики, вот краткое напоминание о синтаксисе и семантике. В человеческом языке синтаксис — это правила построения предложения, а семантика относится к фактическому значению утверждений. В английском языке предложения обычно имеют и подлежащее, то есть человека, место или вещь, и сказуемое, обычно глагол и утверждение, которое объясняет, что делает подлежащее.

Возьмем в качестве примера предложение «Паула любит программировать на Python». В этом предложении Паула — подлежащее, а «любит программировать на Python» — сказуемое. Чтобы составить предложение, которое другие смогут понять, необходимо знать как синтаксис, из которого построено предложение, так и семантику, которая придает ему смысл.

То же самое относится и к языкам программирования. В языке программирования, таком как Python, синтаксис — это правила написания каждой инструкции, а семантика — это эффекты, которые оказывают эти инструкции.

Как и в разговорных языках, существует множество языков программирования на выбор. Каждый имеет свою историю, особенности и приложения, но все они разделяют одни и те же фундаментальные идеи. Поэтому, как только вы поймете основные концепции одного языка программирования, вам будет гораздо проще изучить другой.

Наконец, компьютеры всегда делают именно то, что им говорят. Поэтому, когда вы пишете программу, очень важно четко понимать, что вы хотите, чтобы компьютер сделал. Изучение синтаксиса и семантики выбранного вами языка программирования позволит вам сделать именно это.

Теперь давайте немного поговорим о терминологии. Далее вы часто будете слышать термин «скрипт». Так в чем же разница между скриптом и программой? Граница между ними может быть несколько размытой. И мы будем использовать эти термины взаимозаменяемо.

В целом, скрипты можно рассматривать как программы с коротким циклом разработки, которые можно создать и развернуть быстро. Другими словами, скрипт — это короткая программа, простая и может быть написана очень быстро.

И мы сосредоточимся на конкретном языке скриптов, называемом Python, который мы будем использовать, чтобы изучить основы программирования. Мы узнаем о синтаксисе Python, правилах написания программ на Python и семантике или значении различных составляющих.

Прежде чем мы начнем изучать программирование, и предложим написать первый скрипт на Python, давайте подробнее поговорим о том, что такое автоматизация и почему она полезна.

Хотя мы можем этого и не осознавать, мы постоянно пожинаем плоды автоматизации в нашей повседневной жизни. Вы когда-нибудь оплачивали счета с помощью подписки или пользовались кассой самообслуживания в продуктовом магазине? Я всегда устанавливаю свою кофемашину так, чтобы она начинала варить кофе еще до того, как я встану с постели. Обещание свежего кофе значительно облегчает раннее утро.

Автоматизация — это процесс замены ручного действия автоматическим. Возьмем, к примеру, светофор, который постоянно регулирует поток транспортных средств на перекрестке. Светофор требует вмешательства человека только тогда, когда ему требуется ремонт или техническое обслуживание. Автоматическое регулирование движения означает, что людям не нужно стоять на перекрестке и вручную подавать сигналы, когда автомобили должны остановиться или ехать. Вместо этого люди могут сосредоточиться на более сложных, творческих или трудных задачах, таких как сосредоточение внимания на том, где они едут. Более того, светофоры не устают, не скучают и не показывают случайно зеленый свет, когда он означает красный.

Это подчеркивает еще одно преимущество автоматизации — стабильность. Давайте посмотрим правде в глаза, мы, люди, несовершенны, и иногда мы совершаем ошибки. Человек, выполняющий одну и ту же задачу сотни раз, никогда не будет таким же стабильным, как машина, делающая то же самое.

Но, несмотря на все свои преимущества, автоматизация не является решением для каждой ситуации. Некоторые задачи просто не подходят для автоматизации. Например, они могут требовать определенной степени креативности или гибкости, которые автоматические системы не могут обеспечить. Или для более сложных или реже выполняемых задач создание автоматизации может на самом деле потребовать больше усилий или затрат, чем это того стоит. Подумайте о том, что нужно сделать, чтобы автоматизировать процесс стрижки волос с помощью машины, когда вы стрижетесь? Рост клиента, форма его головы, текущая длина волос и желаемая прическа — все это необходимо учитывать при проектировании автоматической системы. Нам нужно воспроизвести креативность и навыки квалифицированного специалиста, а также провести обширное тестирование, чтобы обеспечить безопасность клиента и качественную стрижку. Если у вас когда-либо был негативный опыт в парикмахерской, вы знаете, что качество может быть субъективным. В этом случае затраты и усилия на автоматизацию просто не стоят тех преимуществ, которые дала бы автоматическая стрижка, поэтому у нас нет роботов-парикмахеров.

Автоматизация — мощный инструмент, если использовать его в нужном месте и в нужный момент. Она может сэкономить время, уменьшить количество ошибок, повысить стабильность и обеспечить возможность централизации решений и ошибок, упрощая их исправление.

Работая в IT-сфере, мы часто используем компьютер для выполнения определенных задач. В вашей работе вы можете создавать учетные записи пользователей, настраивать сеть, устанавливать программное обеспечение, создавать резервные копии существующих данных или выполнять целый ряд других задач, связанных с компьютером.

На своей работе в IT, каждый день, приходя на работу, вы можете вводить одни и те же три команды для аутентификации в системах. Эти учетные данные ежедневно могут обновляться по соображениям безопасности. Поэтому вы можете создать скрипт, который автоматически бы запускал эти команды каждое утро, чтобы не приходилось вводить их вручную.

Задачи, выполняемые компьютером и требующие многократного повторения с небольшими изменениями, очень хорошо подходят для автоматизации. Потому что, автоматизируя задачу, вы исключаете возможность человеческих ошибок и сокращаете время ее выполнения.

Представьте себе такую ситуацию. Ваша компания участвовала в недавней конференции и собрала огромный список электронных адресов от людей, заинтересованных в получении дополнительной информации о вашей продукции. Вы хотите отправить этим людям свою ежемесячную рассылку по электронной почте, но некоторые из подписчиков уже подписаны на неё. Как же убедиться, что все получат вашу рассылку, не отправив её случайно одному и тому же человеку дважды? Можно вручную проверять каждый адрес электронной почты по отдельности, чтобы добавлять в список только новые. Звучит скучно и неэффективно, не правда ли? Так и есть, к тому же это более рискованный способ. Вы можете случайно пропустить новые

письма или добавить уже существующие, или это может так наскучить, что вы заснёте за рабочим столом. Даже автоматическая кофемашина вам в этом не поможет. Что же можно сделать вместо этого?

Можно поручить эту работу компьютеру. Можно написать программу, которая проверяет наличие дубликатов, а затем добавляет каждый новый адрес в список. Ваш компьютер будет делать всё в точности так, как ему будет сказано, независимо от количества писем в списке. Таким образом, он не устанет и не допустит ошибок. Более того, написав программу, вы сможете использовать тот же код в будущих ситуациях, экономя ещё больше времени. Круто, правда? И это ещё не всё. Подумайте, как вы будете отправлять эти электронные письма. Если вы будете отправлять их вручную, вам придётся отправлять одно и то же письмо всем. Персонализация писем потребует слишком много ручной работы. Если же вы используете автоматизацию, имя и компания каждого человека будут автоматически добавляться в письмо. Результат? Более эффективные письма без необходимости тратить часы на вставку имён в текст.

Автоматизация задач позволяет сосредоточиться на проектах, которые более эффективно используют ваше время, позволяя компьютерам делать за вас рутинную работу. Изучение программирования — первый шаг к этому. Если вы хотите, чтобы компьютеры делали работу за вас, вы попали по адресу.

Здесь мы будем использовать язык программирования Python, чтобы продемонстрировать основные концепции программирования и как применять их для написания скриптов. Мы упомянули, что существует множество языков программирования, так почему же мы выбрали Python? Мы выбрали Python по нескольким причинам. Во-первых, программирование на Python обычно похоже на использование человеческого языка. Это потому, что Python позволяет легко выражать то, что мы хотим сделать, с помощью синтаксиса, который легко читать и писать.

```
friends = ['Taylor', 'Alex', 'Pat', 'Eli']
for friend in friends:
    print("Hi " + friend)
```

Посмотрите на этот пример. И даже если вы никогда раньше не видели ни одной строки кода, вы, возможно, сможете догадаться, что делает этот код. Он определяет список с именами друзей, а затем создает приветствие для каждого имени в списке.

Лучший способ изучить любой язык программирования — это как можно больше практиковаться самостоятельно. Если у вас установлен Python на компьютере, вы можете запустить интерпретатор, выполнив команду «python3» (или просто «python» в Windows), и закрыть его, набрав «exit()» или нажав Ctrl-D.

Вы также можете практиковаться, используя один из множества онлайн-интерпретаторов Python или программаторов кода, доступных в интернете. Между интерпретатором и программатором кода нет большой разницы. Интерпретатор более интерактивен, чем программатор кода, но оба позволяют выполнять код и видеть результаты.

Ниже вы найдете ссылки на некоторые из самых популярных онлайн-интерпретаторов и программаторов кода. Попробуйте их, чтобы найти свой любимый.

https://www.python.org/shell/

https://www.onlinegdb.com/online_python_interpreter

https://repl.it/languages/python3

https://www.tutorialspoint.com/execute_python3_online.php
https://rextester.com/l/python3_online_compiler
https://trinket.io/python3

Теперь, что же такое интерпретатор Python в программировании? Интерпретатор — это программа, которая читает и выполняет код. Помните, как мы говорили, что компьютерная программа похожа на рецепт с пошаговыми инструкциями? Так вот, если ваш рецепт написан на Python, интерпретатор Python — это программа, которая читает то, что есть в рецепте, и переводит это в инструкции для вашего компьютера.

Python позволяет легко выражать фундаментальные концепты программирования, такие как структуры данных и алгоритмы, с помощью легко читаемого синтаксиса. Это делает Python отличным языком для изучения программирования. Есть и другие причины выбрать Python. Python очень популярен в IT-индустрии, что делает его одним из самых распространенных языков программирования сегодня. Python не новый. Его первая версия была выпущена Гвидо ван Россумом еще в 1991 году. С тех пор сообщество разработчиков выросло, и язык значительно продвинулся. Всякий раз, когда происходят значительные изменения в семантике или синтаксисе языка, выпускается новая основная версия. В 2000 году был выпущен Python 2. В 2008 году мы получили Python 3.

В течение многих лет Python считался языком для начинающих и в основном использовался для обучения концепциям или написания очень небольших простых скриптов. Но в последние годы распространение Python резко возросло. Одна из причин этого — возросшая мощность языка. Также это связано с тем, что в Python доступно больше инструментов для всё более широкого спектра приложений. Вы можете использовать Python для вычисления статистики, управления сайтом электронной коммерции, обработки изображений, взаимодействия с веб-сервисами и выполнения множества других задач. Python идеально подходит для автоматизации. Он позволяет автоматизировать повседневные задачи, создавая простые скрипты, которые легко понять и легко поддерживать. Именно поэтому Python является предпочтительным языком для многих специалистов, работающих в сфере ИТ-поддержки, системного администрирования и веб-разработки. Более того, он используется в быстрорастущих областях ИТ, таких как машинное обучение или анализ данных. И наконец, Python доступен для загрузки на широкий спектр операционных систем, таких как Windows, Linux и Mac OS.

Хотя мы выбрали Python, важно отметить, что это всего лишь один из многих существующих языков программирования. Рассматривайте любой язык программирования как один из множества мощных инструментов в вашем ИТ-инструментарии. Каждый язык имеет свой уникальный набор преимуществ и недостатков. Некоторые работают быстрее, чем другие. Некоторые лучше подходят для корпоративных приложений, другие особенно хороши для обработки чисел. Существуют языки скриптов, специфичные для конкретных платформ, такие как PowerShell, используемый в Windows, и Bash, используемый в Linux. Оба широко используются системными администраторами на этих платформах. Существуют также языки сценариев общего назначения, похожие на Python, такие как Perl или Ruby, которые также широко используются для написания сценариев и автоматизации. JavaScript, который изначально разрабатывался как язык сценариев для клиентской стороны веб-приложений, все чаще используется в качестве серверного языка для более широкого набора задач. И список на этом не заканчивается. Существует огромное количество традиционных языков, которые можно изучить, таких как C, C++, Java или Go. По мере продвижения по карьерной лестнице в этой области, вы, вероятно, столкнетесь с множеством различных языков и научитесь, когда использовать каждый из них. Но давайте не будем забегать вперед. Сначала нам нужно разобраться с Python.

Приятная особенность изучения основ программирования на одном языке заключается в том, что вы, как правило, можете применять те же концепции, которые вы изучаете, к другим языкам. Это означает, что как только вы освоите Python, вам будет легче осваивать новые языки

программирования, поскольку вы будете замечать и понимать сходства и различия между ними. В конце концов, каждый язык должен выполнять некоторые общие действия, такие как создание переменных, управление потоком программы, чтение входных данных и отображение выходных данных. Даже если они выполняют эти задачи, используя разные подходы. Как мы уже отмечали ранее, изучение языка программирования чем-то похоже на изучение иностранного языка. Вам нужно будет понять синтаксис и семантику этого языка. К счастью для нас, как только вы освоите основы программирования, изучение другого языка станет намного проще, чем изучение второго иностранного языка. Между языками программирования гораздо больше сходств, чем различий.

Чтобы изучить некоторые сходства и различия между различными скриптовыми языками, давайте рассмотрим простую программу, которая выводит слова «Hello, World» десять раз на трех разных языках: Python, Bash и PowerShell.



Как видите, каждый язык использует свой подход к выводу «Hello World». Но присмотритесь внимательнее, и вы увидите и сходства. Каждый язык должен каким-то образом вывести текст на экран. Команда для Python — print, для Bash — echo, а для PowerShell — write host. Также обратите внимание, что каждый язык должен каким-то образом считать до 10. В то время как Python делает это, указывая диапазон 10, Bash использует последовательность чисел для счета от 1 до 10. PowerShell имеет самый сложный синтаксис в этом примере, но он также сводится к тому, чтобы начать с 1 и считать до 10.

Таким образом, как мы только что видели, существует очень много языков программирования. Но пусть это вас не пугает. Здесь вам нужно будет сосредоточиться только на изучении Python. Как только вы освоите Python, вы сможете перейти к изучению любого другого языка программирования.

Теперь, давайте рассмотрим один из самых простых примеров и углубимся в то, что происходит. Мы будем использовать интерпретатор Python, чтобы наш компьютер сказал «Привет миру». Когда мы запустим этот код, локально на нашей машине или в веб-интерпретаторе, слова «Hello, World» появятся на экране, как по волшебству.



На самом деле, это не волшебство. Это потому, что print — это функция Python, которая выводит на экран то, что мы ей указываем. И в выражении «Hello, World» функция print является частью базового языка Python. Всякий раз, когда мы используем ключевые слова или функции, которые являются частью языка, мы используем синтаксис языка программирования, чтобы сказать компьютеру, что делать.

Итак, что такое функции и ключевые слова? Функции — это фрагменты кода, которые выполняют единицу работы. Мы еще много поговорим о функциях позже, и вы даже научитесь писать свои собственные. Ключевые слова — это зарезервированные слова, используемые для составления инструкций. Эти слова являются основной частью языка и могут использоваться только определенным образом. Некоторые слова включают if, while и for. Мы объясним все эти слова и многие другие позже.

Как мы уже говорили, ключевые слова и функции, используемые в Python, составляют синтаксис языка. Как только мы поймем, как они работают, мы сможем использовать их для построения более сложных выражений, которые заставят компьютер делать то, что мы от него хотим.

И наконец, обратите внимание, как Hello, World написано между двойными кавычками. Заключение текста в кавычки указывает на то, что текст считается строкой, а это значит, что именно текст будет обрабатываться нашим скриптом. В программировании любой текст, который не находится внутри кавычек, считается частью кода.

А теперь немного любопытного факта: знаете ли вы, почему мы поприветствовали весь мир в нашем примере? Вывод команды "Hello, World" был традиционным способом начать изучение языка программирования ещё с 70-х годов, когда он был использован в качестве первого примера в известной книге по программированию под названием "Язык программирования C".

Хотя изучение написания "Hello, World" не научит вас всему языку, оно даст вам первое представление о том, как используются функции и как выглядит программа, написанная на этом языке.

В целом, чтобы программа была полезной, ей необходимо получить хотя бы некоторую информацию от пользователя. С помощью этих данных программа может выполнять дей-

ствия, которые актуальны для пользователя, а не общие действия, например, выводить "Hello, World". Данные могут передаваться компьютеру множеством различных способов. Например, на веб-сайте вы можете вводить данные, вводя текст в текстовые поля, или щелкая по ссылкам. Если вы используете мобильное приложение, вы можете нажимать на кнопки или выбирать настройки, из выпадающего меню.

В программе командной строки вы можете предоставлять дополнительные данные, передавая строки в качестве параметров программе, или программа может запрашивать данные в интерактивном режиме. Все эти различные платформы, программы и приложения обрабатывают данные по-разному. Некоторые могут принимать содержимое файла в качестве данных для обработки, другие собирают данные из других источников и обрабатывают их в фоновом режиме.

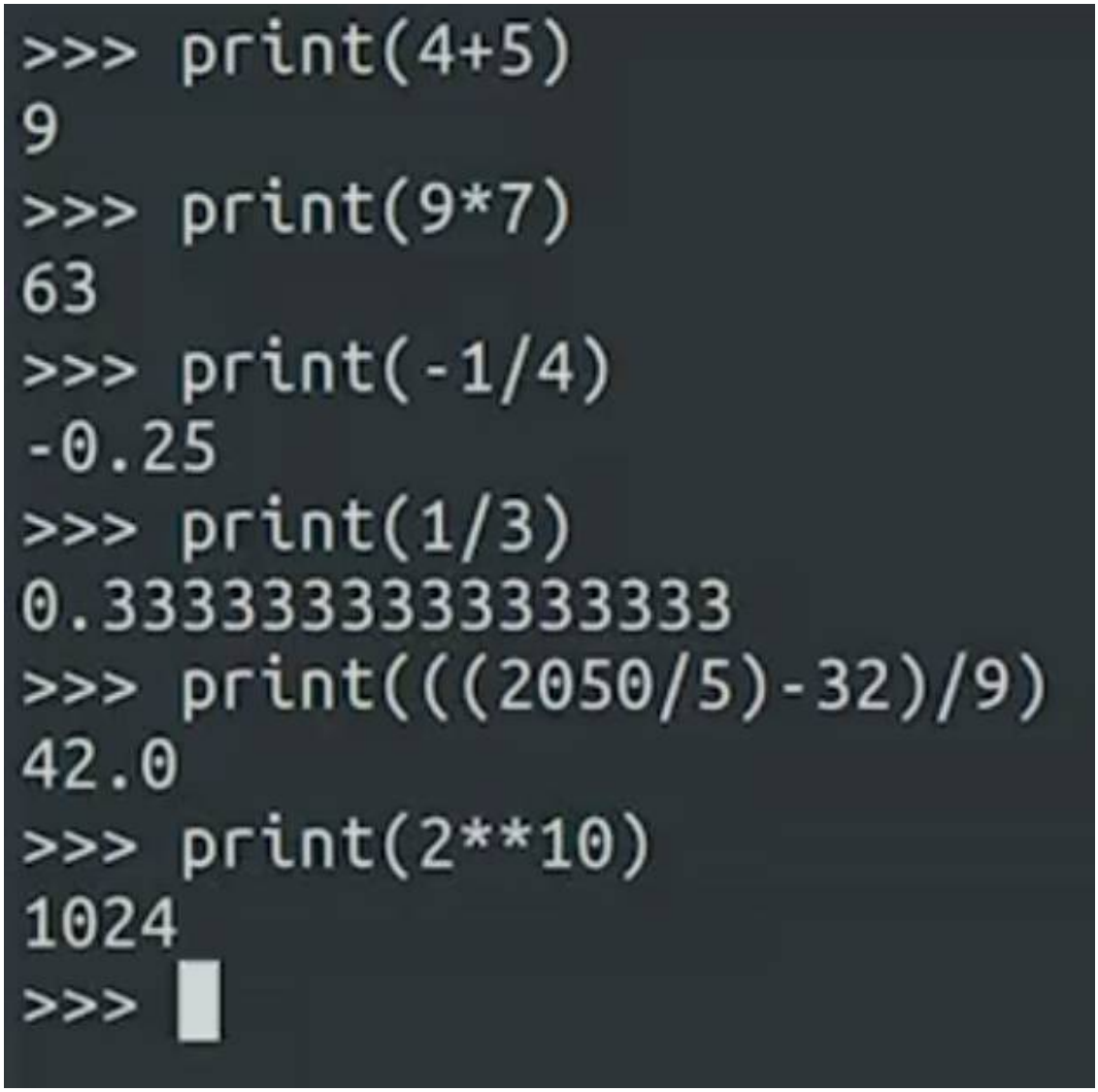
Помните наш предыдущий пример, когда мы автоматизировали процесс идентификации и удаления дубликатов электронных писем? Там данные, предоставляемые программе, представляли собой список электронных писем, который обычно предоставляется в файле, где электронные письма перечислены по одному на строку.

Каким бы способом ваше приложение ни получало данные, они должны откуда-то поступать. В наших первых примерах мы просто будем использовать данные в качестве отдельной строки в нашем блоке кода. Это ограниченный, но простой способ.

```
>>> name = "Brook"
>>> print("Hello " + name)
Hello Brook
>>> name = "Alex"
>>> print("Hello " + name)
Hello Alex
>>>
```

Здесь разделяя имя и вызов функции печати, мы пишем строку кода, которая вызывает функцию печати, делая ее универсальной, но при этом персонализируем приветствие. Если бы мы затем захотели сказать привет другому человеку, нам нужно было бы только изменить имя, но вызов функции печати останется тем же.

Давайте посмотрим другую простую задачу, которую можно выполнить с помощью Python. Мы сделаем Python нашим калькулятором.



Начнем с чего-нибудь простого. 4+5 равно девяти, 9*7 равно 63, -1/4 равно -0,25.

Повторяющиеся или периодические числа выводятся в более длинном формате. Давайте попробуем 1/3. В математической теории, когда единица делится на три, цифра три повторяется бесконечно после десятичной точки. Конечно, сложно отобразить что-то, что повторяется бесконечно. Вместо этого у нас есть представление, показывающее множество десятичных знаков.

Давайте дадим компьютеру что-нибудь посложнее. Допустим, мы хотим разделить 2050 на 5, затем вычесть 32, а затем разделить результат на девять. Для этого нам понадобится использовать скобки, как и в типичных математических задачах.

Вы также можете использовать Python для вычисления квадратов, кубов или любой степени n числа. Например, предположим, мы хотим узнать, чему равно 2^10. Чтобы Python дал нам ответ, мы используем оператор двойной звездочки.

Если вы начинаете беспокоиться, что это превращается в курс алгебры, расслабьтесь. Мы не собираемся делать ничего сложнее, чем то, что мы только что рассмотрели. Но почему бы не использовать Python вместо обычного калькулятора? Экспериментируя таким образом, вы знакомитесь с математическими возможностями языка. В IT-сфере существует множество задач, требующих использования математических вычислений. Вам может понадобиться подсчитать, сколько раз определенное слово появляется в тексте, или вычислить среднее время,

необходимое для выполнения операции, или насколько нужно сжать изображение, чтобы оно соответствовало определенным ограничениям по размеру. Что бы вам ни нужно было вычислить, написание скрипта может помочь вам сделать это быстрее и точнее. Вам нужно знать, какие математические операции доступны вам.

И Python на самом деле обладает гораздо более продвинутыми числовыми возможностями, которые используются для анализа данных, статистики, машинного обучения и других научных приложений.

Теперь, давайте познакомимся с некоторыми инструментами программирования.

Инструменты программирования могут помочь вам разрабатывать, писать, отлаживать и визуализировать ваш код.

Редакторы кода — это инструменты, используемые для упрощения написания кода. С помощью редактора кода вы можете писать, отлаживать и выполнять программы на Python. Редакторы кода предоставляют различные функции, включая подсветку синтаксиса, автоматическое выравнивание, проверку ошибок и автозавершение.

В целом, редакторы кода помогают определять структуру и функции кода. Это позволяет вам писать код более эффективно. Они также упрощают понимание переменных, команд, функций и циклов. VS Code, Jupyter Notebooks и CoLab — это некоторые из редакторов кода.

Интегрированные среды разработки IDE имеют больше функций, чем редакторы кода. Разница между редактором кода и IDE заключается в том, что редактор кода похож на текстовый редактор. А IDE обеспечивают руководство при написании кода и позволяют видеть, как выполняется код. Вы можете представить себе редактор кода как стационарный телефон. Вы можете набирать номера и совершать телефонные звонки с его помощью. IDE больше похожа на смартфон. Вы по-прежнему можете совершать телефонные звонки, но у нее есть дополнительные функции, которые позволяют совершать видеозвонки, отправлять текстовые сообщения, просматривать Интернет и использовать множество других приложений.

IDE — это программный инструмент, который упрощает процесс создания нового программного приложения. IDE — это программное приложение, которое предоставляет комплексные возможности для разработки программного обеспечения. И IDE всегда включают редактор кода. IDE позволяют разрабатывать код более эффективно, объединяя несколько инструментов в одной среде. Вы сможете редактировать, собирать, тестировать и упаковывать все в одном приложении. Они также позволяют программировать приложения быстрее, без необходимости добавлять ручную интеграцию и настройку.

IDLE — это IDE, которая поставляется с Python автоматически.



Это отличная IDE для начинающих, потому что она очень проста в использовании. Возможно, она не подойдёт для больших проектов, но отлично подойдёт для того, чтобы научиться пользоваться IDE.

В Windows откройте меню «Пуск». В поле поиска введите cmd. Щелкните правой кнопкой мыши cmd и выберите «Запуск от имени администратора». Это откроет командную строку.

ОС Windows не поставляется с установленным Python. Посетите официальную страницу загрузки Python для Windows. Выберите установщик Windows (64-разрядная версия) или (32-разрядная версия). После загрузки установщика дважды щелкните файл. Выберите «Установить программу запуска для всех пользователей». Следуйте подсказкам во время установки. Обязательно установите флажок «Добавить путь к python.exe». Это позволит вам запустить Python из командной строки. После завершения установки вы можете проверить наличие Python из командной строки.

Чтобы проверить наличие Python, используйте следующую команду. Появится установленная вами версия Python.

python --version

Вы можете начать писать код Python из терминала. Просто введите python, чтобы использовать интерактивный режим.

Python IDLE включен в установки Python в Windows и MacOS. Python IDLE — это интерактивный интерпретатор или редактор файлов, который позволяет легко писать скрипты и программы на Python. IDLE обеспечивает подсветку синтаксиса, завершение кода и автоматическое выравнивание отступов.

Дважды щелкните значок IDLE, чтобы открыть его на своем компьютере. Откроется пустое окно интерпретатора Python. Вы можете начать писать код прямо сейчас.

```
>>> print("Hello, world!")
Hello, world!
>>>
```

Вы также можете открыть новый файл. Выберите File → Open → New File в строке меню. Здесь вы можете написать файл Python. Завершив запись кода Python в файл, выберите File → Save As. Дайте вашему файлу Python имя. Нажмите Save. Чтобы запустить код Python из сохраненного файла, выберите в меню Run → Run Module.

```
#This program adds two numbers
num1 = 1
num2 = 2

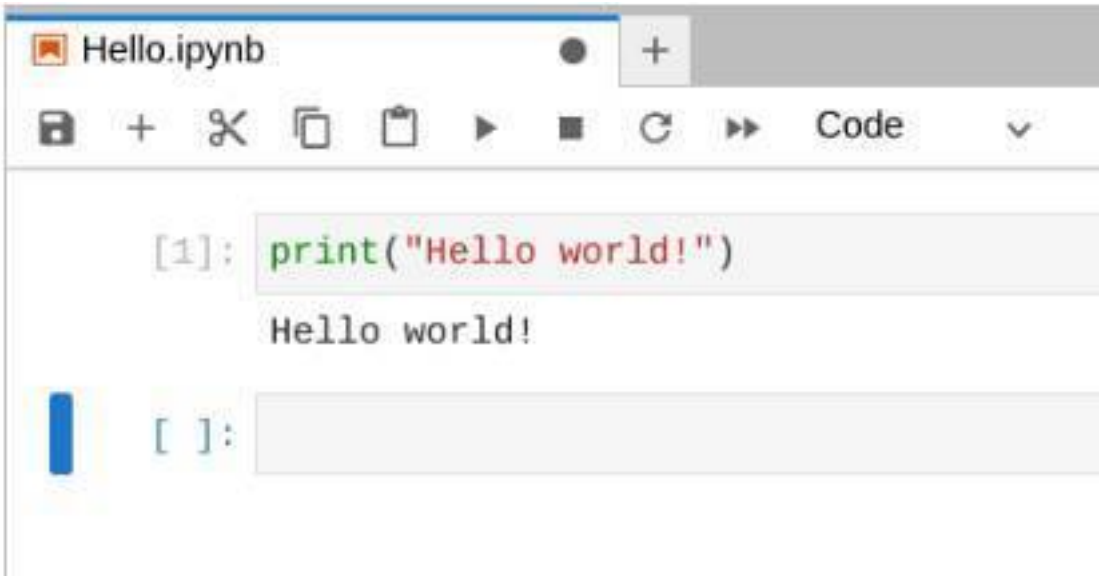
#Add two numbers
sum = num1 + num2

#Display the sum
print("The sum of the numbers is:", sum)
```

Существует множество различных сред, которые можно использовать для написания и запуска кода.

Можно использовать Jupyter Lab и Jupyter Notebooks для написания, запуска и отладки кода Python. Jupyter Lab и Jupyter Notebooks являются частью проекта открытого исходного кода под названием Project Jupyter, который можно использовать бесплатно.

Вы можете узнать больше о проекте Jupyter на jupyter.org. Jupyter Lab — это веб-интерфейс, позволяющий использовать Jupyter Notebooks для написания, запуска и отладки кода Python. Jupyter Lab предоставляет онлайн-среду, позволяющую запускать ваш код в облаке. Блокнот Jupyter Notebooks можно использовать в веб-интерфейсе через Jupyter Lab или на вашей локальной машине. Jupyter Notebook позволяет создать документы, содержащие блоки живого кода. Вы можете писать программы и сценарии Python с помощью Jupyter Notebooks и видеть, как они выполняются, в одном месте. Это отличный инструмент для создания и понимания кода, который вы пишете, потому что вы можете видеть свой ввод и вывод все в одном месте.



Один из самых популярных способов использовать Jupyter Notebooks – в облачной среде. Вы можете использовать эту среду посетив сайт jupyter.org. Как только вы окажетесь на сайте jupyter.org, сначала нажмите Try. Затем выберите Jupyter Lab. Чтобы создать новую записную книжку в лабораторной среде добавьте новую вкладку и выберите Python под блокнотом. Откроется новый блокнот, и вы сможете начать писать и запускать код Python.

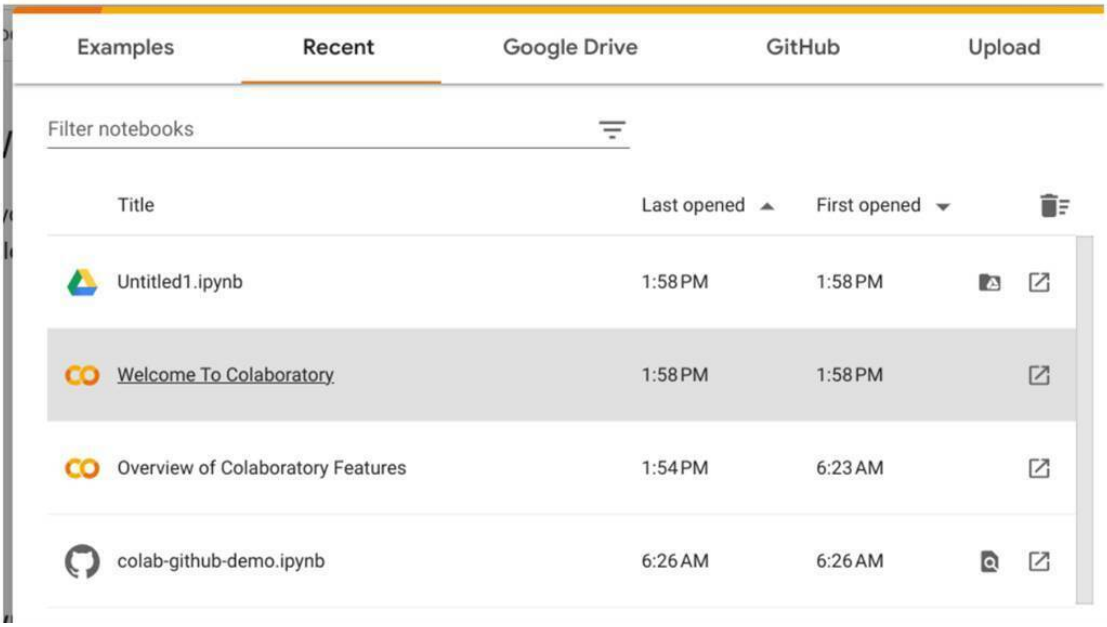
Вы также можете использовать Jupyter Lab и Jupyter Notebooks на вашем локальном компьютере. Вы можете сделать это с помощью команды pip из командной строки терминала Python. Чтобы установить Jupyter Lab, введите команду pip install jupyterlab в командной строке. После установки вы можете запустить его, набрав команду jupyter-lab.

Чтобы установить блокнот Jupyter, введите команду pip install notebook в командной строке. После установки вы можете запустить его с помощью команды jupyter notebook.

Ноутбуки Colab — это Jupyter Notebooks, размещенные в Google Colaboratory. В Colab вы можете написать и запустить код Python.

Colab — веб-платформа, которая позволяет писать и запускать код Python быстро на Google Диске. Это бесплатно и готово к использованию с нулевой настройкой. Вы можете просто посетить colab.sandbox.google.com или выполнить поиск в Google – Colab, чтобы начать. Colab предоставляет всю функциональность, которую предлагает Python. Ячейки внутри Colab могут включать код, текст и изображения. Ячейки кода включают в себя исполняемый код и форматированный текст, что упрощает написание и запуск кода.

Вы можете установить Python пакеты с помощью команды pip в ячейке кода. Блокнотами Colab можно легко поделиться с другими соавторами.



Теперь давайте рассмотрим, как использовать Colab. Сначала откройте Google Colab. Опять же, вы можете сделать это посетив colab.sandbox.google.com. Перейдите к файлу и нажмите New Notebook. Теперь вы можете начать написание кода Python.

В ячейке кода напишем оператор печати, который выводит ваше имя. Чтобы запустить код и увидеть его выполнение, используйте кнопку Run слева от ячейки кода. Вы также можете дать свое имя проекту. Вверху вы можете изменить Untitled.ipynb на что-то вроде Name.ipynb. Это облегчит нахождения своего кода в будущем.

Самое замечательное в Colab то, что он автоматически хранит ваши записные книжки на Google Диске. Чтобы найти свои блокноты, вы можете просто зайти на свой Google Диск, ввести Colab Notebooks в строке поиска и открыть папку под названием Colab Notebooks. Здесь вы увидите все блокноты, которые вы создали.

VS Code — это редактор кода с открытым исходным кодом, который включает в себя инструменты для разработчиков. Он похож на Jupyter Notebooks и Colab, но включает в себя больше функций. VS Code обеспечивает встроенную поддержку автозавершения кода IntelliSense, интерактивный отладчик и другие инструменты сборки и написания скриптов. VS Code имеет простой интерфейс, что облегчает его изучение и использование. Его интуитивно понятные функции делают его отличным выбором для программирования на Python.

Чтобы использовать VS Code для Python, вам потребуется установить Python 3, VS Code и расширение VS Code Python.

Вы можете использовать VS Code в операционных системах Windows, Mac и Linux.

Скачать VS Code можно по адресу codevisualstudio.com/download. Для настройки Python в VS Code нужно выбрать расширения. Чтобы просмотреть расширения в VS Code, нажмите Ctrl+Shift+X. Это покажет список самых популярных расширений VS Code на торговой площадке VS Code. Вы можете найти Python, используя строку поиска, и нажать Install. После завершения установки вам будет предложено начать разработку на Python. Мы сразу же приступим, выбрав Create Python file. Это откроет редактор кода. Давайте протестируем его с помощью простого оператора print("hello world") в новое рабочее пространство. Затем нажмите Run, чтобы запустить код. Вам будет предложено назвать и сохранить файл. Дайте вашему файлу Python имя и нажмите Save. Ваш код запустится, и вы увидите его вывод.

Существует множество редакторов кода и IDE. PyCharm — очень популярная IDE для разработки на Python. Это IDE с открытым исходным кодом, которая предлагает множество

замечательных функций, помогающих писать более качественный код. Это интеллектуальный редактор кода с полезными функциями, включая автозавершение кода, обнаружение ошибок и исправление кода на лету.

PyCharm также позволяет легко перемещаться по коду с помощью сворачивания кода, поиска кода и диаграмм структуры кода. Встроенный отладчик предоставляет точки останова, пошаговое выполнение и проверку переменных, что позволяет очень быстро находить и исправлять ошибки. Вы можете тестировать свой код внутри IDE, что позволяет убедиться в его надежности и отсутствии ошибок. Наконец, PyCharm также предлагает широкий спектр плагинов, созданных сообществом разработчиков, что расширило его функциональность.

В целом, PyCharm — отличный выбор, когда речь идёт об IDE. Он отлично подходит для начинающих, потому что прост в использовании и имеет большое сообщество пользователей, а это значит, что в интернете доступно множество руководств и ресурсов. PyCharm также обладает широкими возможностями настройки.

PyDev — это IDE с открытым исходным кодом для разработки на Python, основанная на Eclipse. PyDev включает в себя интеллектуальный редактор кода, навигацию по коду, отладку, тестирование и рефакторинг. Его механизм рефакторинга может помочь улучшить структуру и читаемость вашего кода, что упрощает сопровождение и отладку. PyDev также полностью интегрирован с Eclipse, что означает, что вы можете использовать все функции Eclipse, такие как обозреватель проектов, система перспектив и встроенный отладчик.

PyDev — отличная IDE для разработчиков на любом этапе. Она бесплатна, совместима со многими версиями Python, а её плагины делают её расширяемой. PyDev обладает большим сообществом пользователей и разработчиков, что означает наличие множества руководств, ресурсов и форумов в сети.

Notepad++ — это текстовый редактор с открытым исходным кодом для среды Microsoft Windows. Он поддерживает множество языков программирования, включая Python. Notepad++ может открывать несколько файлов и вкладок одновременно. Он также подсвечивает синтаксис для известных типов файлов. Подсветка синтаксиса помогает редактировать код, поскольку отображает текст разными цветами и шрифтами, что помогает отслеживать классы, методы, функции и объекты в вашем коде. Он также помогает выявлять ошибки в коде, чтобы вы могли тратить меньше времени на отладку. Notepad++ имеет ряд других расширенных функций, таких как автозавершение и системы управления файлами, которые могут помочь вам писать код более эффективно.

Notepad++ — отличный текстовый редактор для разработчиков, работающих в среде Microsoft Windows. У него активное сообщество, ориентированное на сотрудничество. Пользователи могут обращаться к сообществу за помощью и даже вносить свой вклад в проект.

Последние достижения в области ИИ открыли множество возможностей для улучшения IDE с помощью мощной, контекстно-ориентированной помощи в кодировании. Это инструменты, интегрированные непосредственно в IDE или среду разработки, обеспечивающие более удобный процесс кодирования. Например, Gemini Code Assist — это бесплатный инструмент на основе ИИ, интегрируемый в популярные IDE, такие как Visual Studio Code и JetBrains. Он функционирует как помощник, который может помочь анализировать код и находить ошибки, предлагать изменения, а также взаимодействовать в диалоговом режиме, отвечая на множество других вопросов, как технических, так и более концептуальных.

Эти инструменты быстро становятся незаменимыми для программистов и ИТ-специалистов, поскольку они упрощают и ускоряют рабочие процессы, но если вы решите их использовать, помните, что эта технология все еще развивается. Предложения, код или объяснения не всегда могут быть абсолютно точными, оптимальными или безопасными. Всегда проверяйте и подтверждайте любые результаты, сгенерированные ИИ, прежде чем выполнять запросы или

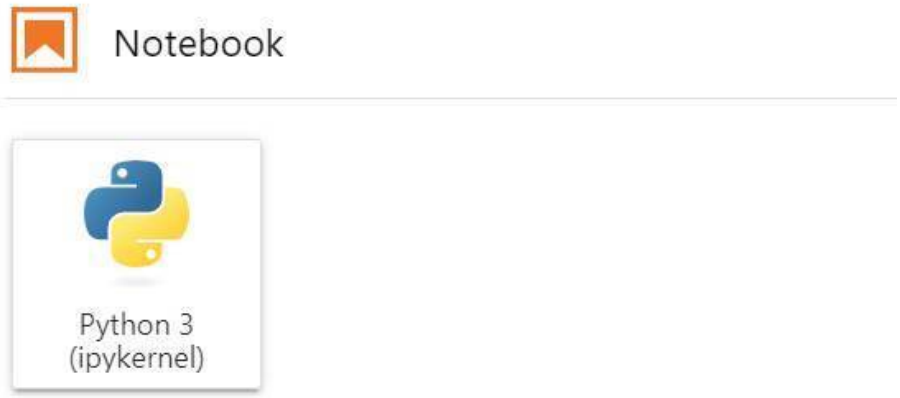
полагаться на эту информацию. Относитесь к помощнику ИИ как к полезному помощнику, но сохраняйте контроль и ответственность за свой конечный код.

Переменные и типы

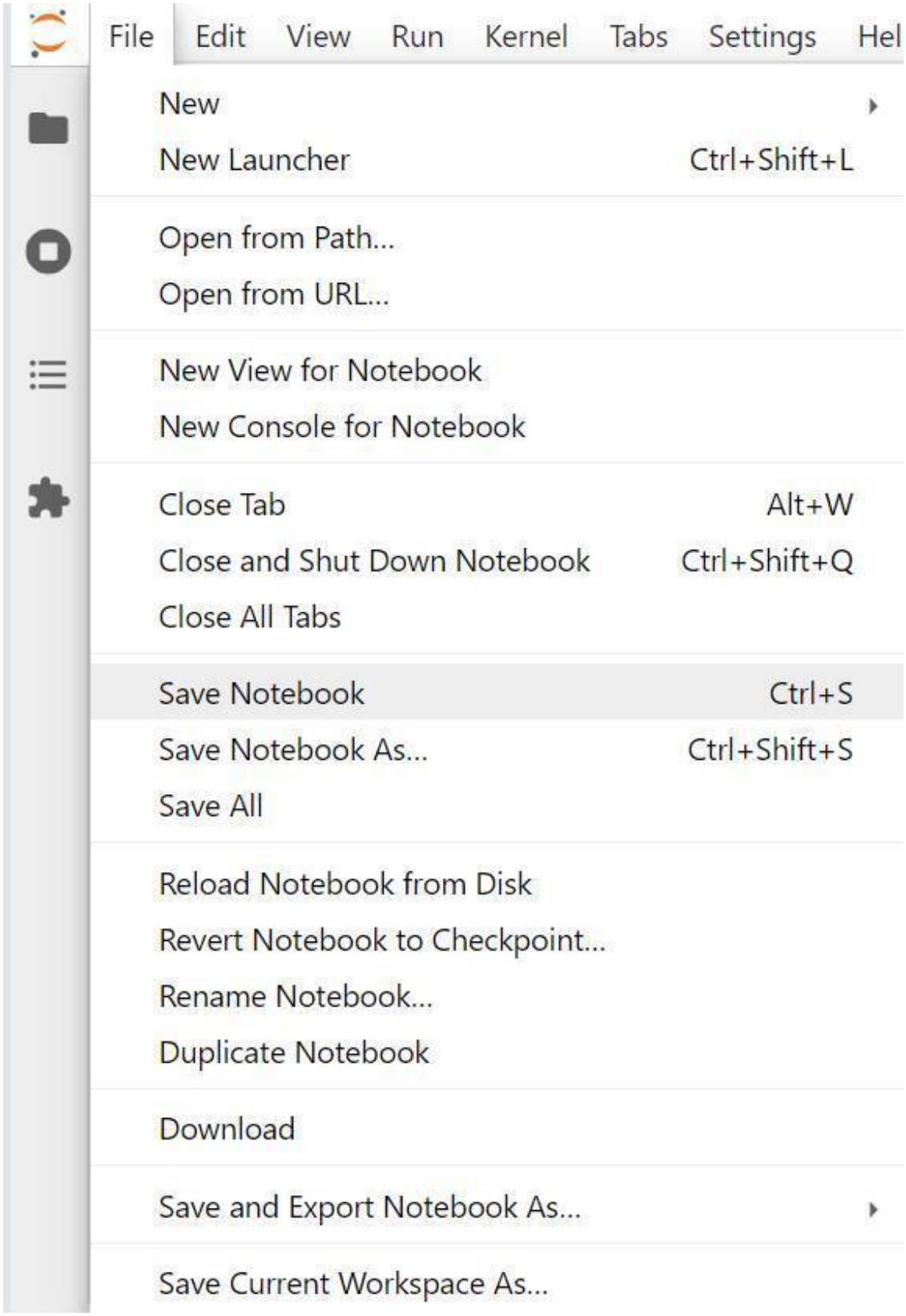
Python не является строго типизированным языком. Это означает, что мы можем назначать переменные и не беспокоиться о их значениях.

Здесь мы будем работать в Jupyter Notebooks. Для запуска приложения наберите в командной строке

jupyter lab

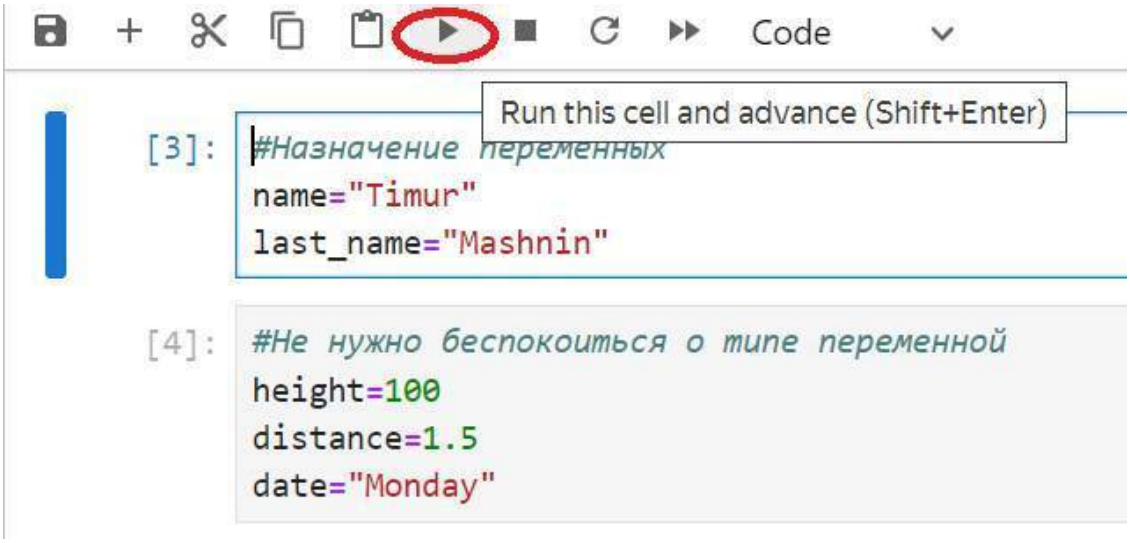


И нажмите значок создания записной книжки.



В меню File вы можете сохранить и переименовать созданную записную книжку.

В ячейках записной книжки вы можете набирать и выполнять код Python с помощью меню Run или соответствующей кнопки.



Здесь мы назначили переменные, указав имя переменной и ее значение. Ниже в следующей ячейке мы назначили переменные с разными типами значений, не указывая тип значения при присваивании его переменной.

Здесь важно отметить, что вы можете присваивать какие угодно значения переменным. Вот почему Python относят к языку программирования, который не является строго типизированным. Но здесь вы должны быть осторожны, потому что отсутствие проверки типов может привести к проблемам.

```
height="100"
width=30
length= height + width
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 3
      1 height="100"
      2 width=30
----> 3 length= height + width

TypeError: can only concatenate str (not "int") to str
```

Здесь я назначил высоту 100. Но здесь 100 — это строка. Поэтому, если вы захотите выполнить числовую операцию со строкой, у вас могут возникнуть проблемы.

```
name="Timur"
name="NoTimur"
print(name, last_name)
```

```
NoTimur Mashnin
```

Вы также можете переназначать переменные. Здесь у нас есть имя, которое вы также можете свободно переназначить, никаких проблем не произойдет. Далее вы можете использовать печать для переменных с помощью оператора print. Это хороший способ проверить выходные данные.

В данном случае я использую запятую, поэтому здесь появляется имя и фамилия. Вы можете использовать запятую или не использовать запятую. Если использовать значения, разделенные запятыми, вы получите интервал между ними при печати. Если вы не используете запятые, вы не получите такой интервал.

```
full_name=f"{name} {last_name}"
print(full_name)
```

```
NoTimur Mashnin
```

Далее вы можете создавать новые переменные из существующих переменных. Здесь у нас есть имя и фамилия, но мы можем их переназначить, сконструировать что-то совершенно другое и присвоить это новой переменной.

В данном случае мы хотим создать переменную полное имя, и для этого мы используем здесь F-строку — вариант форматирования строк в Python (https://docs.python.org/3/tutorial/inputoutput.html). F-строки являются строковыми литералами с «f» в начале выражения и фигурными скобками, содержащими выражения, которые будут заменены своими значениями.

```
name="NoName"
new_name=name
print(new_name)
name="Timur"
print(name)
```

```
NoName
Timur
```

Таким образом, мы можем создавать новые переменные из существующих переменных. В данном случае это новое имя, и мы его напечатаем. Затем мы присваиваем существующей переменной новое значение. И мы напечатаем это значение.

Давайте подробнее рассмотрим некоторые из наиболее распространенных типов значений, с которыми вам придется работать.

Итак, начнем со строк. Строки — это очень мощный инструмент, но иногда они могут сбивать с толку в Python.

```
full_name='Timur Mashnin'
print(full_name)
full_name="Timur Mashnin"
print(full_name)
full_name="""Timur Mashnin"""
print(full_name)
```

```
Timur Mashnin
Timur Mashnin
Timur Mashnin
```

```
full_name="'Timur Mashnin'"
print(full_name)
summary="""My name is 'Timur' """
print(summary)
```

```
'Timur Mashnin'
My name is 'Timur'
```

Здесь у нас есть переменная, в которой используются одинарные кавычки, двойные кавычки и тройные кавычки. Так какая разница, почему?

Иногда вам могут понадобиться двойные или даже тройные кавычки, чтобы избежать экранирования специальных символов, то есть комбинаций символов, которые обозначают служебные или непечатаемые символы, которые нельзя вставить обычным способом.

В противном случае у вас возникнут проблемы и, вероятно, вы получите синтаксическую ошибку от Python.

Точно так же вы можете использовать одинарные кавычки, когда внутри строки есть двойные кавычка, или наоборот.

```
full_name="Timur Mashnin"
summary="My name is " +  full_name
print(summary)
```

```
My name is Timur Mashnin
```

Также, строка поддерживает добавление к другим строкам для составления текста. Вы можете составлять новые строки из существующих с помощью оператора «+».

```
type(15)
```

```
int
```

```
14/7
```

```
2.0
```

```
7/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[29], line 1
----> 1 7/0

ZeroDivisionError: division by zero
```

```
7+"14"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[30], line 1
----> 1 7+"14"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Теперь, способ проверки различных типов, — это использование функции type. Итак, вы вводите type, а затем открываете круглые скобки, а затем значение или переменную. И в этом случае вы увидите представление типа.

15 это целое число. А в Python это называется int.

Также вы можете выполнять различные математические операции с числами.

Здесь целое число 14 вы делите на целое число 7. Но обратите внимание, что результат это уже не целое число 2.

Вы также можете выполнить недопустимые математические операции, если не будете осторожны. Здесь мы получили ошибку деления на ноль.

И у вас также могут возникнуть проблемы, когда вы пытаетесь добавить к одним типам другие типы, которые в данном случае несовместимы. Здесь мы пытаемся добавить целое число к строке. В результате мы получим ошибку «неподдерживаемый тип операнда».

```
type(14/7)
```

```
float
```

```
type(True)
```

```
bool
```

```
print(bool(1))
type(bool(1))
```

```
True
bool
```

```
print(bool(0))
type(bool(0))
```

```
False
bool
```

```
type(None)
```

```
NoneType
```

Теперь, типы могут меняться в зависимости от операций, и здесь, я делю целое число 14 на целое число 7 и получаю число 2.0 с плавающей точкой. Так что это уже не целое число.

Некоторые типы данных представлены как логические значения true или false. С заглавной буквы T и заглавной буквы F для истинного и ложного значения соответственно.

Вы также можете вызвать функцию bool с 1 и 0 в качестве аргумента, и Python будет интерпретировать эти значения как истинные или ложные.

И, наконец, у нас есть значение None, которое очень распространено в Python. Некоторые функции, некоторый код могут возвращать пустое значение None. Вы можете назначить значение None переменным, вы можете использовать его для возврата значения или сигнала о том, что ничего не происходит.

Ранее мы упомянули, что текст, написанный в кавычках в Python, называется строкой. В терминологии программирования строка известна как тип данных. Большинству программ необходимо манипулировать какими-либо данными, и эти данные могут иметь множество различных форм, или, как мы их называем, типов данных. Строка — это лишь один из видов типов данных, встречающихся в Python.

Существует множество других, например, целое число, которое представляет целые числа без дробной части, например, 1, и число с плавающей запятой, которое представляет действительные числа. Или, другими словами, число с дробной частью, например, 2,5. Как правило, ваш компьютер не знает, как смешивать разные типы данных. Например, сложение двух целых чисел имеет определенный смысл для компьютеров.

Сложение двух строк также имеет смысл. В итоге мы просто получаем более длинные строки. Но ваш компьютер не знает, как сложить целое число и строку. Если вы укажете компьютеру смешивать эти два разных типа данных, он не будет знать, что делать, и выдаст ошибку. Но не паникуйте. Ошибки — обычное явление в программировании, и вам, вероятно, придётся часто с ними сталкиваться.

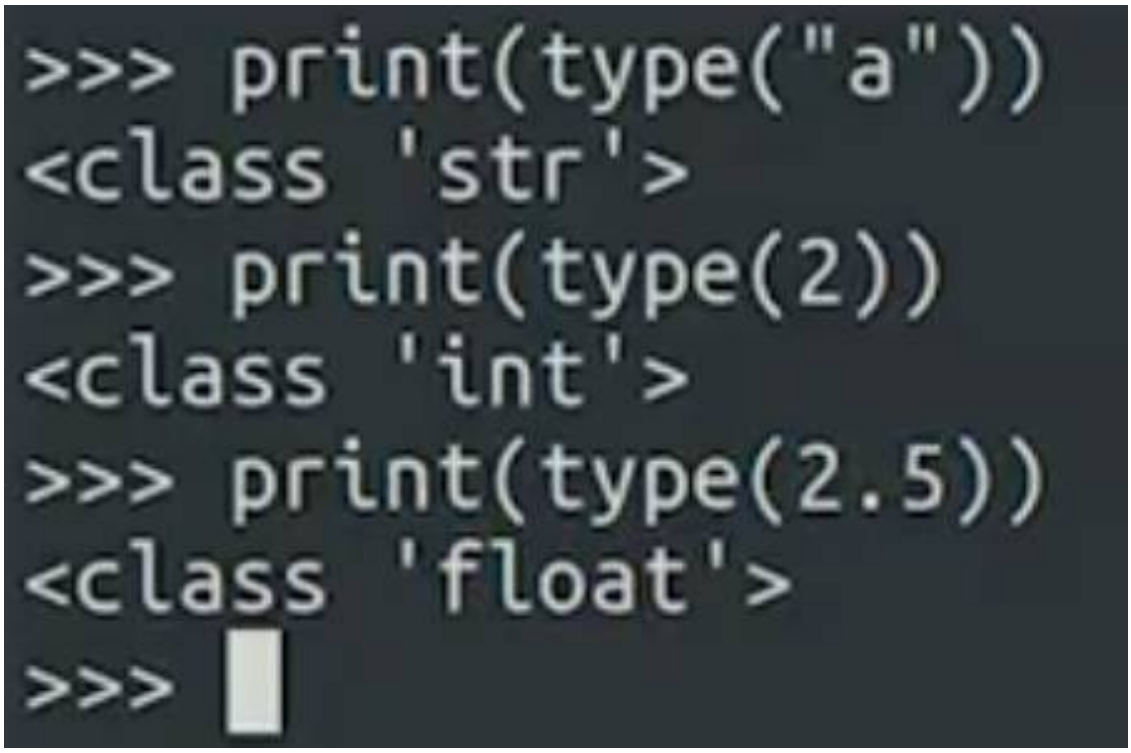
Хитрость заключается в том, чтобы воспринимать ошибки как небольшие подсказки от вашего компьютера, которые помогут вам улучшить ваши навыки программирования. Внимательно читайте сообщения об ошибках, понимайте, что они вам говорят, а затем используйте эти новые знания, чтобы исправить ошибку.

```
>>> print(7+8)
15
>>> print("hello " + "world")
hello world
>>> print(7 + "8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

В этом примере последняя строка сообщения об ошибке показывает нам, что мы столкнулись с чем-то, называемым TypeError. И мы получаем пояснительный текст, который говорит нам, что знак плюса нельзя использовать между типом int и типом str, которые являются сокращенными названиями для целого числа и строки. Сообщение "неподдерживаемый тип операнда" говорит нам о том, что мы не можем сложить целое число 7 в строке "8", потому что это разные типы данных.

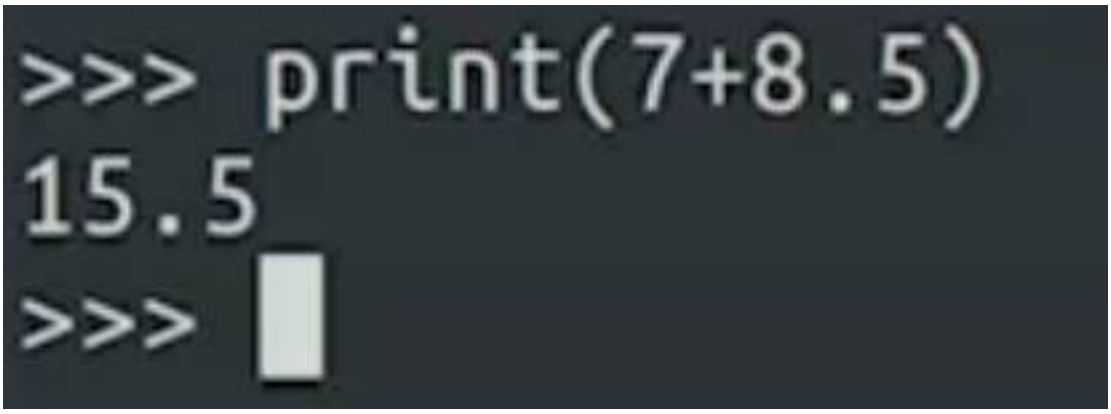
Для компьютера сложение 7 плюс 8 так же странно, как сложение 7 плюс а для нас, и 7 плюс а не имеет никакого смысла. Возможно, будет полезно подумать о типах данных с точки

зрения информации, которую они могут представлять. Например, имя файла будет представлено как строковый тип данных, а размер этого файла может быть целочисленным типом данных. Если вы когда-либо не будете на 100% уверены в том, какой тип данных имеет определенное значение, Python предоставляет вам удобный способ это выяснить.



Вы можете использовать функцию `type`, чтобы компьютер сообщил вам тип. Это может пригодиться при работе с кодом, написанным кем-то другим, если вы не уверены, какой тип данных используется.

Таким образом, мы видели, что нельзя использовать оператор сложения между целым числом и строкой, потому что это разные типы данных. Но что произойдет, если мы попытаемся работать с целым числом и числом с плавающей запятой?



Python без проблем выполняет эту операцию, но в чем дело? Разве целое число и число с плавающей запятой — это не два разных типа данных? Конечно, это так, но здесь за кулисами компьютер автоматически преобразует наше целое число 7 в число с плавающей запятой 7. Это

позволяет Python затем сложить значения, чтобы получить результат, который также является числом с плавающей запятой. Мы называем этот процесс неявным преобразованием.

Интерпретатор автоматически преобразует один тип данных в другой. Мы уже говорили об этом раньше, но стоит еще раз подчеркнуть, что операции Python не ограничиваются только числами. Вы также можете использовать оператор сложения для сложения строк.

```
>>> print("a"+"b"+"c")
abc
>>> print("This "+ "is "+ "pretty " + "neat!")
This is pretty neat!
>>>
```

Это позволяет создавать предложения из отдельных слов. Только не забудьте добавить пробелы к каждому слову, иначе компьютер соберет их все вместе.

А что, если вы действительно хотите объединить строку и число? Возможно ли это? Конечно, возможно, но только с явным преобразованием. В Python для преобразования между двумя типами данных мы вызываем функцию с именем типа, в который мы преобразуем.

```
>>> base = 6
>>> height = 3
>>> area = (base*height)/2
>>> print("The area of the triangle is: " + str(area))
The area of the triangle is: 9.0
>>>
```

Давайте посмотрим, как это работает. В этом скрипте мы сначала вычисляем площадь треугольника, а при выводе результата добавляем ее к строке. Для этого нам нужно вызвать функцию str, чтобы преобразовать число в строку. Здесь наше число было преобразовано в строку и выведено вместе с сообщением.

Условные операторы

Мы уже рассмотрели арифметические выражения, таких как сложение, вычитание и деление. Но Python также может сравнивать значения. Это позволяет нам проверять, меньше ли что-то, равно ли или больше чего-то другого. Это позволяет нам брать результаты наших выражений и использовать их для принятия решений.

```
>>> print(10>1)
True
>>> print("cat" == "dog")
False
>>> print(1 != 2)
True
>>>
```

Посмотрите на эти три примера. В первом примере 10 больше единицы, поэтому в результате выводится значение True. True — это значение, принадлежащее типу данных, называемому логическим значением (Boolean). Логические значения имеют одно из двух возможных состояний: True или False. Каждый раз, когда вы сравниваете что-либо в Python, результатом является логическое значение.

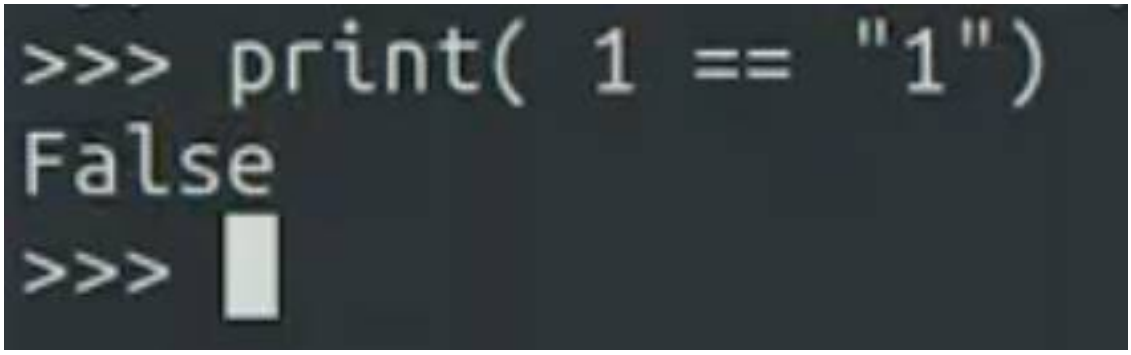
Во втором примере мы видим оператор равенства, который образуется путем объединения двух знаков равенства. Мы используем этот оператор для проверки того, равны ли два значения друг другу. В этом примере строка cat не равна строке dog, поэтому выводимое логическое значение равно False.

В третьем примере мы выполняем противоположное сравнение. Сочетая восклицательный знак и знак равенства, мы используем оператор отрицания. В этой строке кода оператор проверяет, что один не равен двум.

```
>>> print(1 < "1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
>>>
```

Ранее мы упомянули, что оператор сложения не работает между целыми числами и строками. Как вы думаете, что произойдет, если мы попытаемся сравнить целое число и строку?

Давайте выясним это, проверив, меньше ли число один, чем строка 1. Мы получаем ошибку типа. Это происходит потому, что Python не знает, как проверить, меньше ли число, чем строка.



А что насчет оператора равенства? В данном случае интерпретатор без проблем сообщает нам, что целое число 1 и строка 1 — это не одно и то же. Для компьютера очевидно, что это совершенно разные сущности.

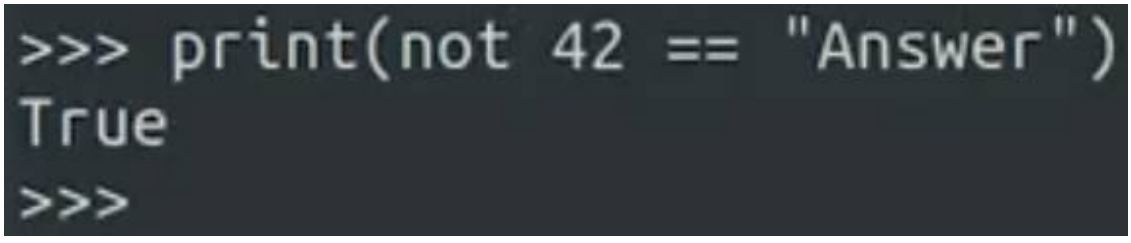
Помимо операторов сравнения и равенства, в Python также есть набор логических операторов. Эти операторы позволяют связывать несколько операторов вместе и выполнять более сложные сравнения. В Python логическими операторами являются слова «и», «или» и «не».

```
>>> print("Yellow" > "Cyan" and "Brown" > "Magenta")
```

Чтобы оператор «и» был истинным, оба выражения должны быть истинными одновременно. Здесь мы сравниваем строки, а операторы «больше» и «меньше» относятся к алфавитному порядку. Жёлтый идёт после голубого, а коричневый — не после пурпурного. Это означает, что первое утверждение истинно, но второе не истинно, что делает результат всего выражения ложным.

```
>>> print("Yellow" > "Cyan" and "Brown" > "Magenta")
False
>>> print(25 > 50 or 1 != 2)
True
>>>
```

Если мы используем оператор «или», вместо этого выражение будет истинным, если хотя бы одно из выражений истинно, и ложным только тогда, когда оба выражения ложны.



Наконец, оператор «не» инвертирует значение выражения, которое находится перед ним. Если выражение истинно, оно становится ложным. Если оно ложно, оно становится истинным.

Логические операторы важны, потому что они помогают писать сложные выражения.

Способность программы изменять последовательность выполнения называется ветвлением, и это ключевой компонент, делающий скрипты полезными. Вы, вероятно, часто используете идею ветвления в своей повседневной жизни. Например, если до полудня, вы можете поприветствовать кого-то, сказав «доброе утро» вместо «добрый день», или «добрый вечер». Если на улице дождь, вы можете выбрать вариант взять зонтик. Если холодно, вы, вероятно, наденете куртку.

В своих скриптах вы также можете дать указание компьютеру принимать решения, на основе входных данных.

Первое, что вам нужно знать, это ключевое слово if.

```
# all conditionals evaluate the result of the condition (expression)
condition = True
if condition:
    print("The condition was met")
```

✓ 0.2s

```
The condition was met
```

```
# if the condition changes to false, then it get skipped
condition = False
if condition:
    print("Was the condition met?")
```

✓ 0.2s

if — ключевое слово в Python, которое позволяет нам установить определенное условие, оценить это условие и выполнить некоторые действия в зависимости от результата.

Здесь, в первой ячейке, вы можете видеть, что условие равно True. И если условие выполнено, то будет напечатано: «Условие выполнено».

И если какое бы то ни было условие оценивается как истинное, то следующая строка, следующий блок кода будет выполнен – это основа обработки логики в Python. Условие должно иметь значение True, чтобы выполнилась следующая строка кода.

Что произойдет, если условие будет ложно? Ничего не произойдет, потому что Python увидит значение False, как определено здесь переменной, и поэтому оператор печати никогда не выполнится.

```
# some data structures are truthy: false when empty, but true when they contain items
groceries = []
if groceries:
    print("we have some groceries!")

invites = ["John", "George"]
if invites:
    print("we have some invites!")
```

✓ 0.2s

```
we have some invites!
```

Нужно отметить, что некоторые структуры данных, когда они пустые, они оцениваются как False, и оцениваются как True, когда содержат определенные элементы.

Здесь вы можете видеть, что переменная groceries пустая, поэтому в коде не будет выполнен оператор печати.

В другом блоке кода переменная invites инициализирована списком, поэтому оператор печати будет выполнен.

И это в значительной степени верно для каждой структуры данных в Python, которая является встроенной в Python структурой данных, и когда она инициализирована, она будет вычислена как истина.

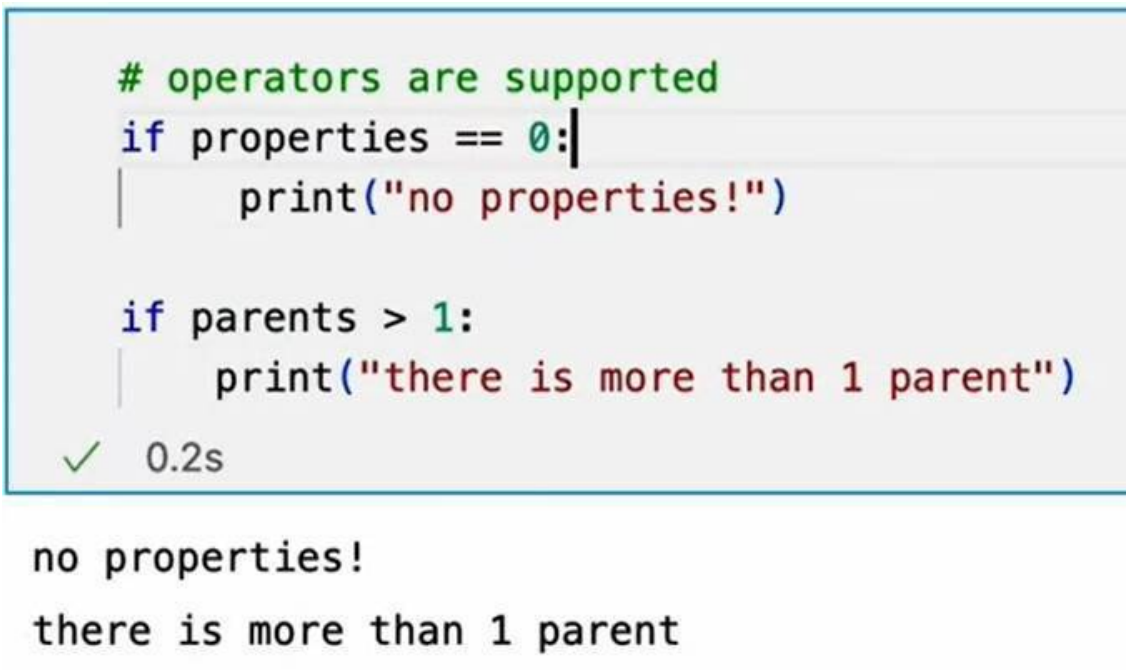
```
# other types like integers (0, and any positive integer) are truthy
properties = 0
if properties:
    print("We have properties!")

parents = 2
if parents:
    print("we have parents!")
```

✓ 0.2s

```
we have parents!
```

Теперь, другие типы данных, такие как целые числа, в данном случае ноль, вычисляются как False, а любое положительное целое число вычисляется как True.



Также поддерживаются различные операторы сравнения, которые в результате вычисления дают значение True или False для выражения if.

Оператор if уже является довольно полезной конструкцией, но мы можем расширить его, чтобы сделать его еще более мощным.

```
def hint_username(username):
    if len(username) < 3:
        print("Invalid username. Must be at least 3 characters long")
    else:
        print("Valid username")
```

Здесь мы добавили оператор else, и теперь программа может двигаться

в одном из двух направлений, в зависимости от длины имени пользователя. Если оно недостаточно длинное, мы получаем сообщение, указывающее на то, что имя пользователя недействительно. Но если программа проверяет, что имя пользователя достаточно длинное, она выведет сообщение, указывающее на то, что оно действительно.

Обратите внимание на то, как написан оператор else. Здесь используется ключевое слово else, за которым следует двоеточие, указывающее на начало блока else. Еще раз, тело блока смещено вправо. Как мы уже отмечали ранее, эти блоки могут содержать несколько строк и делать больше, чем просто выводить сообщения. Они могут выполнять вычисления, изменять значения, возвращать значения и многое другое. И помните, что вы можете использовать сколько угодно пробелов для отступа, но отступ всегда необходим, и количество пробелов всегда должно быть одинаковым.

```
if properties:
    print("We have properties")
else:
    print("We don't have any properties")
```

✓ 0.2s

```
We don't have any properties
```

Условие else в сочетании с if дает нам вилку алгоритма – если соблюдены определенные условия, мы можем сделать это, в противном случае мы можем сделать что-то еще.

В этом примере, если условие if получает значение False, здесь в игру вступит блок else.

```
if properties:
    print("We have properties")
# evaluate if parents is valid
elif parents:
    print("We don't have any properties, but we have parents")
```

✓ 0.1s

```
We don't have any properties, but we have parents
```

Другое дело — использование условия elif (иначе если).

Условия elif полезно, когда вы хотите оценивать как True или False еще и else. Это похоже на else, но с дополнительным преимуществом, заключающимся в том, что вы можете оценить другое условие.

```
name = None

if not name:
    print("Didn't get a name!")
```

✓ 0.2s

```
Didn't get a name!
```

У вас также могут быть ситуации, когда требуется применение отрицательного условия. Здесь переменной name присвоено значение None. И вы можете использовать ключевое слово not. По сути, это меняет оценку условия на противоположное значение.

Таким образом, мы можем сказать что-то вроде: «Если нет имени, тогда печатаем».

```
# same is possible with elif conditions
name = "Alfredo"
last_name = None

if not name:
    print("No name!")
elif not last_name:
    print("No last name either!")
```

✓ 0.2s

```
No last name either!
```

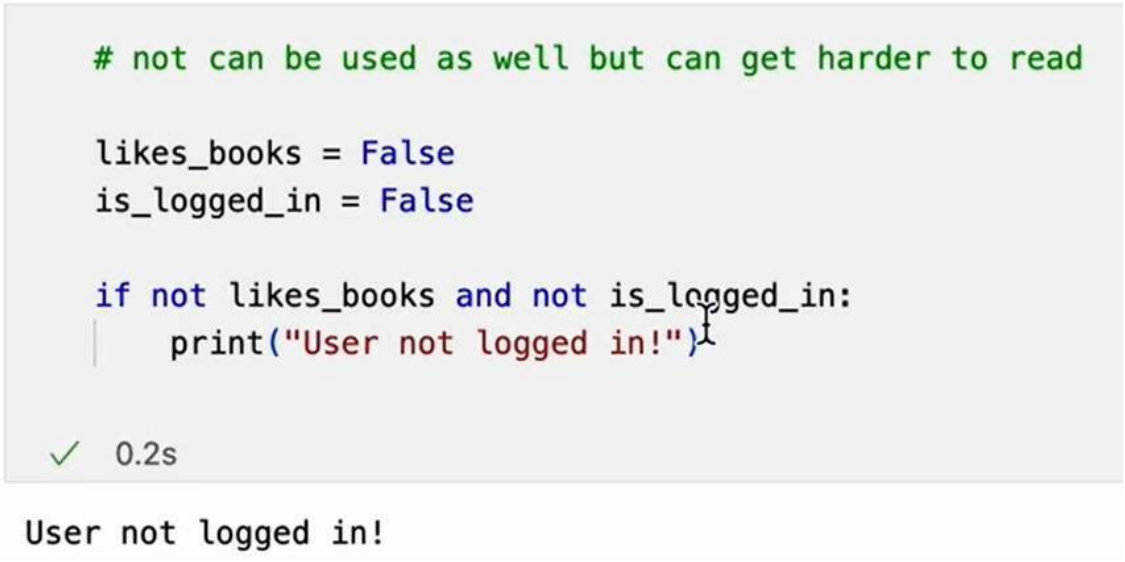
И то же самое возможно и с условиями elif. Здесь мы проверяем если нет имени, а затем проверяем если нет фамилии.

```
has_kids = True
married = True

if has_kids and married:
    print("This person is married and has kids")
```

✓ 0.1s

```
This person is married and has kids
```



Теперь будьте осторожны с составными условиями, потому что их может быть трудно прочитать.

Здесь в первой ячейки оба условия True, поэтому выполняется оператор печати. То же самое можно сделать и с not. Здесь во второй ячейке оба условия меняются с False на True и также выполняется оператор печати.

Циклы

Как мы уже говорили, компьютеры отлично умеют повторять одну и ту же задачу снова и снова. Им никогда не надоедает, и они никогда не ошибаются. Вы можете попросить компьютер выполнить одно и то же вычисление 1000 раз, и первый результат будет таким же точным, как и последний, чего мы, люди, не можем сделать наверняка. Вы когда-нибудь пытались сделать что-то 1000 раз подряд? Этого было бы достаточно, чтобы свести вас с ума.

Способность точно выполнять повторяющиеся задачи и никогда не уставать — вот почему компьютеры так хороши для автоматизации. Автоматизированная задача может быть любой, например, копирование файлов на множество компьютеров в сети, отправка персонализированных электронных писем списку пользователей или проверка того, что процесс все еще запущен. Неважно, насколько сложна задача, ваш компьютер выполнит ее столько раз, сколько вы ему скажете, что оставит вам время для более интересных вещей, таких как планирование будущих потребностей в оборудовании или управление развертыванием программного обеспечения.

Здесь мы рассмотрим три метода автоматизации повторяющихся задач: циклы while, циклы for и рекурсия. Каждый из этих методов используется для того, чтобы указать компьютеру повторить задачу, но каждый из них использует немного различный подход.

Для начала поговорим о циклах while. Циклы while дают указание компьютеру непрерывно выполнять код в зависимости от значения условия. Это работает аналогично оператору ветвления if. Разница в том, что тело блока может выполняться несколько раз, а не только один раз.

```
x =   0
while x < 5:
    print("Not there yet, x=" + str(x))
    x = x + 1
print("x=" + str(x))
```

Посмотрите на эту программу. Можете ли вы угадать, что она делает? Давайте разберем ее построчно.

В первой строке мы присваиваем переменной x значение ноль. Мы называем это действие инициализацией, чтобы задать начальное значение переменной. В следующей строке мы запускаем цикл while. Мы устанавливаем условие для этого цикла: x должно быть меньше пяти. Сейчас мы знаем, что x равно 0, поскольку мы только что инициализировали его, поэтому это условие в данный момент истинно.

В следующих двух строках у нас есть блок, смещенный вправо, — это тело цикла while. В теле цикла две строки. В первой строке мы выводим сообщение, за которым следует текущее значение x. Во второй строке мы увеличиваем значение x. Мы делаем это, добавляя единицу к его текущему значению и присваивая его обратно x. После первого выполнения тела цикла x будет равен 1 вместо 0. Поскольку это цикл, компьютер не просто продолжает выполнение

со следующей строки скрипта. Вместо этого он возвращается к началу цикла, чтобы повторно оценить условие для цикла while.

Поскольку единица здесь все еще меньше пяти, он снова выполняет тело цикла. Затем он выводит сообщение и снова увеличивает x на единицу. Теперь x равен 2. Компьютер будет продолжать делать это до тех пор, пока условие перестанет быть истинным.

В этом примере условие будет ложным, когда x больше не будет меньше 5. Как только условие станет ложным, цикл завершится, и будет выполнена следующая строка. Наконец, последняя строка нашего кода выводит последнее значение x.

Это был простой пример того, как работает цикл while. Как мы уже говорили, мы изучаем строительные блоки программирования. Зная эти строительные блоки, вы можете комбинировать их для создания более сложных выражений. Для IT-специалиста циклы while могут быть очень полезны. Вы можете использовать их, чтобы продолжать запрашивать имя пользователя, если предоставленное недействительно, или, возможно, пытаться выполнить операцию, пока она не будет успешной. Умение создавать такие выражения может помочь вам заставить компьютер делать очень многое, используя лишь небольшое количество кода.

Давайте теперь посмотрим пример цикла while внутри функции.

```
def attempts(n):
    x = 1
    while x <= n:
        print("Attempt " + str(x))
        x += 1
    print("Done")

attempts(5)
```

Сможете ли вы понять, что делает эта функция? В этом примере мы начинаем с инициализации переменной с именем x. В данном случае мы инициализируем ее значением 1. Затем мы входим в цикл while, который проверяет, меньше ли значение переменной x параметра n, полученного функцией. Если это сравнение истинно, то выполняется код внутри блока while.

Допустим, мы передаем значение 5 в качестве параметра этой функции. При первом проходе цикла x всегда равно 1, поэтому сравнение «на 1 меньше или равно 5» будет истинным. Затем мы входим в тело цикла. В теле мы сначала выводим сообщение, указывающее на номер текущей попытки. Затем мы увеличиваем значение x на 1, чтобы увеличить число. Здесь мы используем немного другое выражение, чем раньше: x + = 1 — это сокращенная версия выражения x = x + 1. Вы можете использовать любое из этих выражений, поскольку они оба означают одно и то же.

Процесс продолжается до тех пор, пока результат сравнения перестанет быть истинным, что происходит, когда x больше n. В нашем текущем примере это произойдет, когда значение x равно 6.

```
username = get_username()
while not valid_username(username):
    print("Invalid username")
    username = get_username()
```

В предыдущих примерах мы использовали простые условия: число меньше или меньше или равно другому числу. Это распространенные условия, но они отнюдь не единственные условия, которые могут быть в цикле while.

Например, часто вызывают отдельную функцию, которая оценивает условие. В этом случае за функциями скрывается много кода, который делает то, чего мы не видим. Здесь используется функция «getusername», которая запрашивает у пользователя имя пользователя, и функция «valid username», которая проверяет это имя пользователя. И всё это происходит всего лишь в нескольких символах.

В данном случае тело цикла while будет выполняться до тех пор, пока пользователь не введёт действительное имя пользователя.

Важно помнить, что условие, используемое циклом while, должно оцениваться как истинное или ложное. Неважно, делается ли это с помощью операторов сравнения или вызова дополнительных функций. Условия, используемые в циклах while, также могут стать более сложными, если мы используем логические операторы and, or и not. Это позволяет комбинировать значения нескольких выражений, чтобы получить желаемый результат.

Как мы уже отмечали ранее, написание циклов позволяет нам заставлять компьютер выполнять за нас повторяющуюся работу. Поскольку одним из главных преимуществ написания скриптов в IT является экономия времени за счет автоматизации повторяющихся задач, циклы очень полезны. Поэтому давайте убедимся, что вы избегаете некоторых наиболее распространенных ошибок, которые люди допускают при написании циклов.

Одна из самых распространенных ошибок — это забывать инициализировать переменные правильным значением. Все мы совершали эту ошибку, начиная программировать. Когда мы забываем инициализировать переменную, могут произойти две разные вещи.

```
>>> while my_variable < 10:
...     print("Hello")
...     my_variable +=1
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'my_variable' is not defined
>>>
```

Первый возможный результат, и его проще всего обнаружить, — это ошибка Python, сообщающая нам, что мы используем неопределенную переменную. Как и в случае с другими ошибками, с которыми мы сталкивались, мы можем посмотреть на последнюю строку, чтобы понять, что происходит. Этот тип ошибки — ошибка имени, и сообщение, которое появляется после нее, говорит о том, что мы используем неопределенную переменную. Исправить это довольно просто. Нам просто нужно инициализировать переменную перед её использованием.

```
>>> x = 1
>>> sum = 0
>>> while x < 10:
...     sum += x
...     x += 1
...
>>> product = 1
>>> while x < 10:
...     product = product * x
...     x += 1
...
>>>
```

Теперь есть вторая проблема, с которой мы можем столкнуться, если забудем инициализировать переменную правильным значением. Мы могли уже использовать эту переменную в нашей программе. В этом случае, если мы повторно используем переменную, не установив правильное значение с самого начала, она всё ещё будет иметь предыдущее значение. Это может привести к довольно неожиданному поведению.

Посмотрите на этот скрипт. Можете ли вы найти проблему? В первом блоке мы правильно инициализируем x значением 1 и sum значением 0, а затем итерируем до тех пор, пока x не станет равным 10, суммируя все промежуточные значения. Таким образом, к концу этого блока sum равно результату сложения всех чисел от 1 до 10, а x равно 10.

Во второй части кода первоначальное намерение состояло в том, чтобы получить произведение всех чисел от 1 до 10. Но, если вы присмотритесь, вы увидите, что мы инициализируем product, но забываем инициализировать x. Итак, x по-прежнему равно 10. Это означает, что, когда проверяется условие цикла while, x уже равно 10 в начале итерации. Условие цикла while ложно еще до начала цикла, и тело цикла никогда не выполняется.

В этом случае обнаружить проблему может быть сложнее, потому что Python не выдает ошибку. Если у вас есть цикл, который вышел из-под контроля и ведет себя не так, как ожидалось, полезно проверить, правильно ли инициализированы все переменные. В этом примере нам нужно установить x обратно в 1 перед началом второго цикла.

Итак, подведем итог: всякий раз, когда вы пишете цикл, проверяйте, что вы инициализируете все переменные, которые хотите использовать, прежде чем использовать их.

Возможно, вы уже запомнили, что циклы while используют условие, чтобы проверить, когда следует выйти. Тело цикла while должно убедиться, что проверяемое условие изменится. Если оно не изменится, цикл может никогда не завершиться, и мы получим так называемый бесконечный цикл — цикл, который продолжает выполняться и никогда не останавливается.

```
while x % 2 == 0:
    x = x / 2
```

Посмотрите этот пример. В нем используется оператор по модулю. Этот цикл завершится для положительных и отрицательных значений x. Но что произойдет, если x будет равно нулю? Остаток от деления 0/2 равен нулю, поэтому условие будет истинным. Результат деления нуля на два также будет равен нулю, поэтому значение x не изменится, этот цикл будет продолжаться бесконечно, и мы получим бесконечный цикл.

Чтобы этого избежать, нам нужно подумать о том, что должно произойти, чтобы цикл был успешным.

```
if x != 0:
    while x % 2 == 0:
        x = x / 2
```

В этом примере мы бы сказали, что x должно быть отлично от нуля. Мы могли бы вложить этот цикл while в оператор if. При таком подходе цикл while выполняется только тогда, когда x не равно нулю.

```
while x != 0 and x % 2 == 0:
    x = x / 2
```

В качестве альтернативы мы могли бы добавить условие непосредственно в цикл, используя логический оператор, как в этом примере. Это гарантирует, что мы входим в тело цикла только для значений x, которые одновременно отличаются от нуля и являются четными.

Помните, что при написании циклов полезно уделить немного времени рассмотрению различных значений, которые может принимать переменная. Это поможет вам убедиться, что ваш цикл не зависнет. Если вы видите, что ваша программа работает бесконечно, не завершаясь, еще раз взгляните на ваши циклы, чтобы убедиться, что в коде нет бесконечного цикла.

Хотя нужно остерегаться бесконечных циклов, они не всегда плохи. Иногда вам действительно нужно, чтобы ваша программа выполнялась непрерывно, пока не будет выполнено какое-либо внешнее условие. Если вы использовали утилиту ping в Linux или macOS, или ping-t в Windows, вы видели бесконечный цикл в действии. Этот инструмент будет продолжать отправлять пакеты и выводить результаты в терминал, пока вы не отправите ему сигнал прерывания, обычно нажав Ctrl+C.

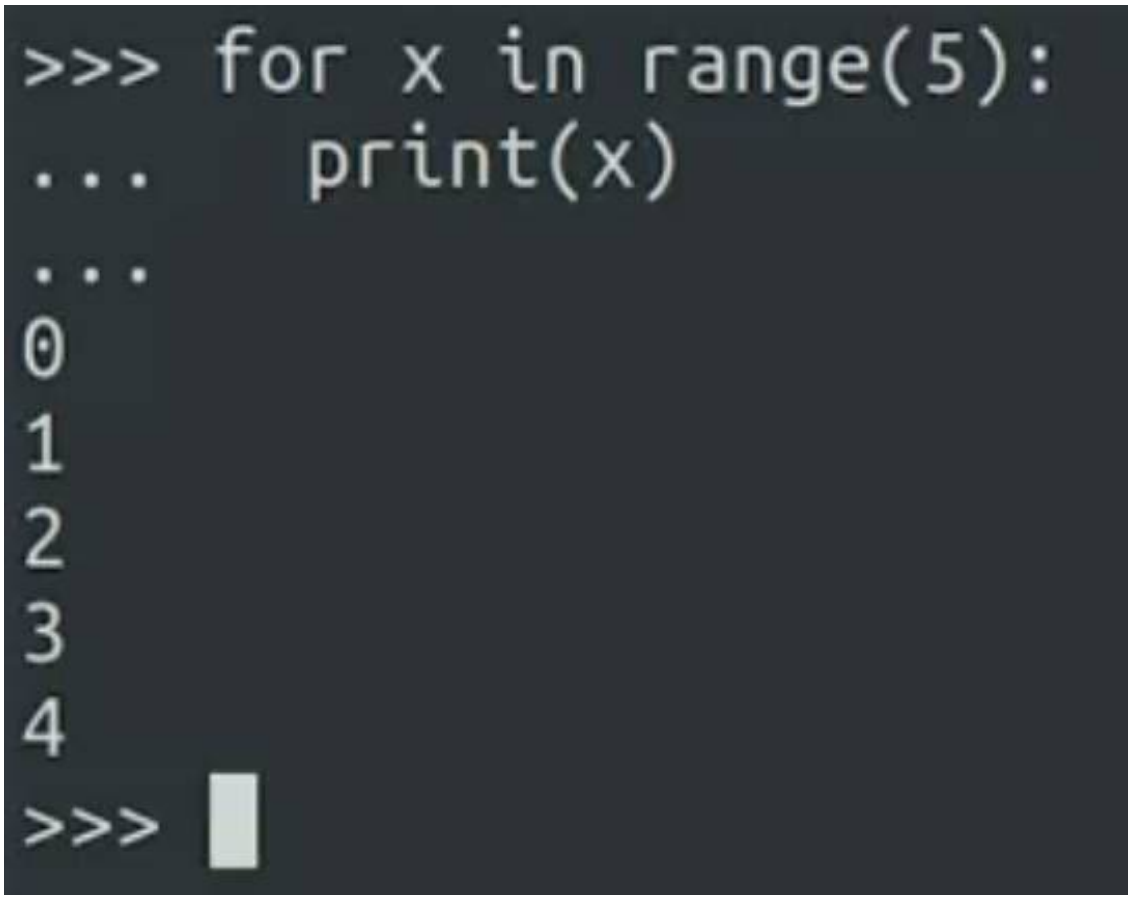
Если бы вы посмотрели исходный код этой программы, вы бы увидели, что она использует бесконечный цикл для этого с блоком кода, содержащим инструкции для бесконечной отправки пакетов. Важно отметить, что всегда должна быть возможность разорвать цикл, отправив определенный сигнал. В примере с ping, этим сигналом является нажатие пользователем Ctrl+C. В других случаях это может быть нажатие пользователем кнопки в графическом приложении, или отправка другой программой определенного сигнала, или даже достижение временного лимита.

```
while True:
    do_something_cool()
    if user_requested_to_stop():
        break
```

В вашем коде может быть бесконечный цикл, который выглядит примерно так. В Python мы используем ключевое слово brake, чтобы сигнализировать о том, что текущий цикл должен прекратить выполнение. Мы можем использовать его не только для остановки бесконечных циклов, но и для преждевременной остановки цикла, если код уже достиг необходимого результата.

Как избежать наиболее распространенных ошибок при написании циклов while? Во-первых, не забудьте инициализировать ваши переменные, а во-вторых, убедитесь, что ваши циклы не будут работать бесконечно.

Далее познакомимся с циклом for. Цикл for перебирает последовательность значений.



Очень простой пример цикла for — это перебор последовательности чисел. Обратите внимание, как эта структура похожа на структуры, которые мы уже видели. Первая строка указывает на ключевое слово. В данном случае это for, и она заканчивается двоеточием. Тело цикла имеет отступ справа, как мы видели в цикле while и блоке if.

В данном случае отличие заключается в наличии ключевого слова «in». Кроме того, между ключевыми словами «for» и «in» находится имя переменной. Эта переменная будет принимать каждое значение из последовательности, по которой проходит цикл. В этом примере цикл будет перебирать последовательность чисел, сгенерированных с помощью функции «range». Есть две важные вещи, которые я хочу отметить об этой функции «range».

Во-первых, в Python и многих других языках программирования диапазон чисел по умолчанию начинается со значения ноль. Во-вторых, список сгенерированных чисел будет на единицу меньше заданного значения. В этом примере «x» будет принимать значения 0, 1, 2, 3 и 4.

```
friends = ['Taylor', 'Alex', 'Pat', 'Eli']
for friend in friends:
    print("Hi " + friend)
```

Итерация по числам очень похожа на примеры с циклом while. Вы можете задаться вопросом, зачем нужны два цикла, которые выглядят так, будто делают одно и то же. Дело в том, что сила цикла for заключается в том, что мы можем использовать его для итерации по последовательности значений любого типа, а не только по диапазону чисел. В этом примере мы видим цикл for, который перебирает список строк. Здесь мы перебираем список строк, и для каждой строки в списке мы выводим приветствие.

Последовательность, по которой перебирает цикл for, может содержать элементы любого типа, а не только строки. Например, мы могли бы перебрать список чисел, чтобы вычислить общую сумму и среднее значение.

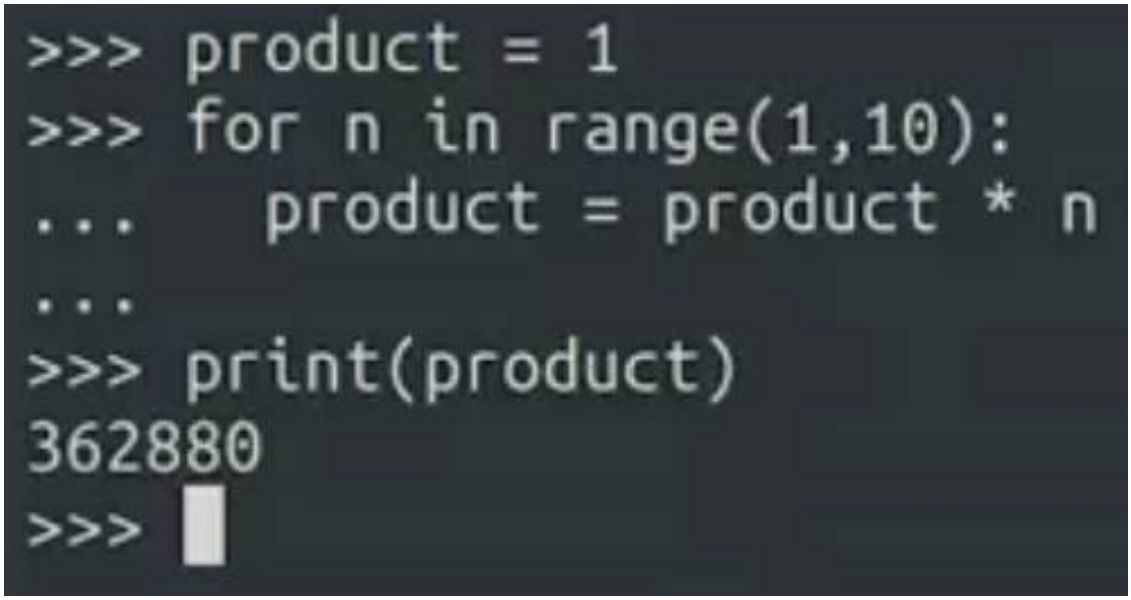
```
values = [ 23, 52, 59, 37, 48 ]
sum = 0
length = 0
for value in values:
    sum += value
    length += 1

print("Total sum: " + str(sum) + " - Average: " + str(sum/length))
```

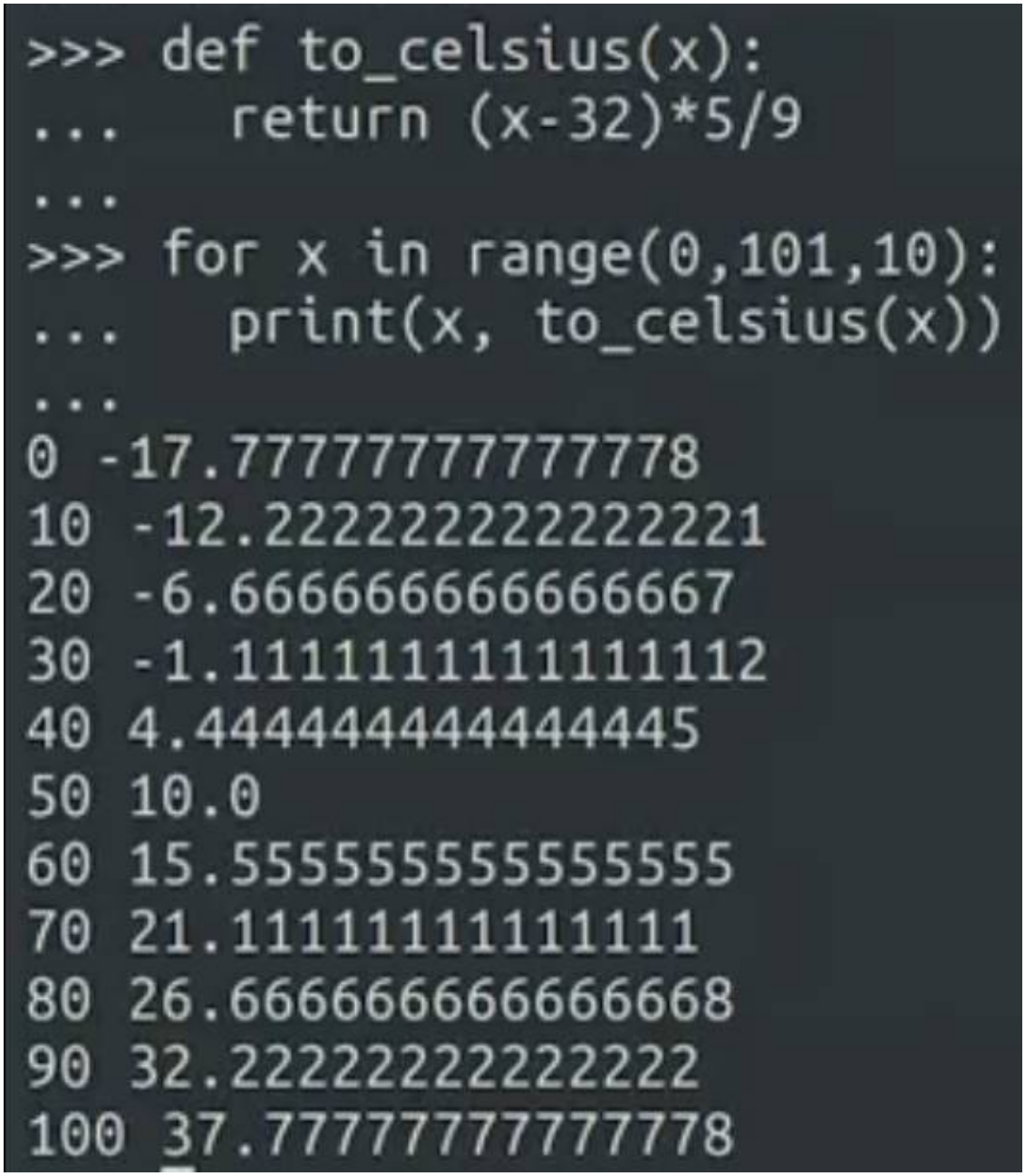
Здесь мы определяем список значений. После этого мы инициализируем две переменные, sum и length, которые будут обновляться в теле цикла for. В цикле for мы перебираем каждое из значений в списке, добавляя текущее значение к сумме значений, а затем также добавляя единицу к длине, которая вычисляет, сколько элементов в списке. После того, как мы прошли весь список, мы выводим сумму и среднее значение.

Некоторые примеры последовательностей, которые мы можем перебирать, — это файлы в каталоге, строки в файле, процессы, запущенные на машине, и множество других. Как IT-специалист, вы можете использовать циклы for для автоматизации множества вещей. Например, вы можете использовать их для копирования файлов, обработки содержимого файлов, автоматической установки программного обеспечения и многого другого.

Если вы задаетесь вопросом, когда следует использовать циклы for, а когда — циклы while, есть способ это определить. Используйте циклы for, когда есть последовательность элементов, которые вы хотите перебирать. Используйте циклы while, когда вы хотите повторять действие до тех пор, пока не изменится условие. Если то, что вы пытаетесь сделать, можно сделать и с помощью цикла for, и с помощью цикла while, просто используйте тот, который вам больше нравится.



Мы говорили о функции range и о том, как она генерирует последовательность чисел, начиная с нуля. Однако иногда нам не нужно начинать с нуля. В таких ситуациях функция range также позволяет указать первый элемент списка, который нужно сгенерировать. Мы делаем это, передавая функции два параметра вместо одного. В этом примере мы вычисляем произведение всех чисел от 1 до 9. Для этой операции важно начать с единицы, а не с нуля. Если бы мы начали с нуля, все произведение было бы равно нулю. Кроме того, мы можем указать третий параметр для изменения размера каждого шага. Это означает, что вместо того, чтобы идти по одному элементу, мы могли бы использовать шаг между ними.



Давайте рассмотрим этот пример. Здесь сначала мы определяем функцию, которая преобразует значение температуры из градусов Фаренгейта в градусы Цельсия, и для этого мы просто используем формулу преобразования. Затем у нас есть цикл for, который начинается с нуля и доходит до 100 с шагом 10. Обратите внимание, что мы используем 101 в качестве верхнего предела вместо 100. Мы делаем это, потому что диапазон никогда не включает последний элемент, а мы хотим включить 100 в наш диапазон.

Тело цикла for выводит значение в градусах Фаренгейта и значение в градусах Цельсия, создавая таблицу преобразования.

Теперь давайте посмотрим, что происходит, когда циклы находятся внутри циклов.

Вы когда-нибудь играли в домино? На всякий случай, если вы не знакомы, каждая фишка домино имеет два числа, представленные набором точек, вырезанных на каждой половине фишки. Числа от 0 до 6.

Фишки можно поворачивать так, чтобы каждая комбинация чисел была представлена только один раз в наборе фишек домино. Другими словами, фишка 2,3 та же самая, что и фишка 3,2, и в наборе только одна такая фишка.

Теперь представьте, что мы хотим написать программу, которая выводит каждую фишку домино в наборе. Если мы возьмём все фишки, у которых слева ноль, мы можем вывести фишки с числами от 0 до 6 справа. Это можно легко сделать с помощью цикла for.

А как насчёт плиток, у которых слева единица? Нам нужно пропустить плитку с 1,0, потому что она уже была учтена как 0,1. Мы могли бы вывести список плиток с единицей слева и числами от 1 до 6 справа.

Когда мы посмотрим на двойку, нам нужно будет пропустить и ноль, и единицу, и так далее.

Превращение этого в код означает, что нам нужно написать два цикла, один внутри другого. Это то, что мы называем вложенными циклами for.

```
>>> for left in range(7):
...     for right in range(left, 7):
...         print("[" + str(left) + "|" + str(right) + "]", end=" ")
...     print()
...
[0|0] [0|1] [0|2] [0|3] [0|4] [0|5] [0|6]
[1|1] [1|2] [1|3] [1|4] [1|5] [1|6]
[2|2] [2|3] [2|4] [2|5] [2|6]
[3|3] [3|4] [3|5] [3|6]
[4|4] [4|5] [4|6]
[5|5] [5|6]
[6|6]
>>>
```

Посмотрите, как это выглядит в коде Python.

В этом коде мы используем новый параметр, который мы передаем функции print. Этот параметр называется end. Обычно, после того как функция print получает переданное содержимое и выводит его на экран, записывается специальный символ, создающий новую строку – символ новой строки.

Если мы хотим, чтобы функция print записала что-то другое вместо символа новой строки, мы используем параметр end, как показано в этом примере.

Обратите внимание, как второй цикл for каждый раз перебирает разное количество элементов при каждом вызове по мере изменения значения параметра left.

В зависимости от того, чего вы хотите добиться с помощью вложенных циклов, вы можете захотеть, чтобы оба цикла всегда проходили одно и то же количество элементов, или вы можете захотеть, чтобы второй цикл соединялся с первым.

```
>>> teams = [ 'Dragons', 'Wolves', 'Pandas', 'Unicorns']
>>> for home_team in teams:
...   for away_team in teams:
...     if home_team != away_team:
...       print(home_team + " vs " + away_team)
...
Dragons vs Wolves
Dragons vs Pandas
Dragons vs Unicorns
Wolves vs Dragons
Wolves vs Pandas
Wolves vs Unicorns
Pandas vs Dragons
Pandas vs Wolves
Pandas vs Unicorns
Unicorns vs Dragons
Unicorns vs Wolves
Unicorns vs Pandas
>>>
```

Давайте рассмотрим другой пример. Допустим, вы руководите местной женской баскетбольной лигой в вашем городе. У вас есть четыре команды, которые будут играть друг против друга в лиге, как дома, так и на выезде. Вы сохранили названия команд в списке, подобном этому. И мы хотим написать скрипт, который будет выводить все возможные пары команд для игр.

Для этой задачи имеет значение порядок названий, потому что для каждой игры первое название будет домашней командой, а второе название — гостевой командой. И конечно, чего мы не хотим, так это чтобы команда играла сама с собой.

Какое выражение нам нужно использовать, чтобы этого избежать? Для этого нам нужно использовать условное выражение, которое гарантирует, что мы будем выводить пару только тогда, когда названия различаются.

Как видите, вложенные циклы очень полезны для решения определенных задач. Вложенные циклы — это удобный инструмент, но нам нужно быть осторожными и не применять их вслепую к любой задаче. Почему? Потому что чем длиннее списки, которые ваш код должен перебирать, тем больше времени потребуется вашему компьютеру для выполнения задачи.

Допустим, вам нужно выполнить операцию, которая будет проходить по списку из 10 000 элементов. Если операция занимает одну миллисекунду на элемент, то весь цикл займет одну миллисекунду, умноженную на 10 000, что составляет 10 секунд. Теперь представьте, что мы добавляем вложенный цикл, который должен пройти по тем же 10 000 элементам. Это означает, что каждая итерация внешнего цикла будет выполнять полную итерацию внутреннего цикла, что, опять же, займет 10 секунд, что позволит пройти весь список. Теперь вся итерация занимает 10 000 раз по 10 секунд, что составляет 100 000 секунд. Это более 27 часов.

Давайте рассмотрим некоторые распространенные ошибки, с которыми вы можете столкнуться. Как мы уже отмечали, циклы for итерируют последовательности. И интерпретатор откажется итерировать по одному элементу.

```
>>> for x in 25:
...     print(x)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> for x in range(25):
...     print(x)
```

Как вы видите здесь, мы пытаемся для x в диапазоне 25 вывести x. В этом примере мы пытаемся итерировать по числу 25. И Python выводит ошибку типа, сообщая нам, что целые числа не являются итерируемыми. Есть два решения этой проблемы, в зависимости от того, что мы пытаемся сделать. Если мы хотим итерировать от 0 до 25, то мы используем функцию range.

```
>>> for x in [25]:
...     print(x)
...
25
>>>
```

Но если мы пытаемся перебрать список, в котором 25 является единственным элементом, то это должен быть список, а это значит, что его нужно записать в квадратных скобках.

Возможно, вы задаетесь вопросом, зачем вообще нужно перебирать список из одного элемента, и это хороший вопрос. Эта задача обычно возникает, когда у вас есть функция с циклом for внутри, которая перебирает элементы списка, полученного в качестве параметра. Допустим, у вас есть функция, которая исправляет права доступа к списку файлов, полученных в качестве параметра, и вы хотите вызвать эту функцию, чтобы исправить права доступа только к одному конкретному файлу. Для этого вам нужно передать файл в качестве единственного элемента списка.

Рекурсия

Рекурсия — это повторное применение одной и той же процедуры к более простой задаче. Вы когда-нибудь играли с матрешками? Это отличный наглядный пример рекурсии. В каждой матрешке находится меньшая матрешка. Когда вы открываете матрешку, чтобы найти меньшую внутри, вы продолжаете, пока не дойдете до самой маленькой матрешки, которую нельзя открыть.

Рекурсия позволяет нам решать сложные задачи, сводя задачу к более простой. Возьмите матрешки, вложенные друг в друга, и представьте, что мы хотим узнать, сколько всего матрешек. Нам нужно было бы открывать каждую куклу по очереди, пока не дойдем до последней, а затем посчитать, сколько кукол мы открыли. Это рекурсия в действии.

Вот еще один пример с более сложной задачей. Представьте, что вы стоите в очереди и хотите узнать, сколько людей перед вами. Скажу вам честно, я терпеть не могу длинные очереди. В любом случае, если очередь длинная, может быть трудно посчитать людей, не сойдя с очереди, и не потеряв свое место. Вместо этого вы могли бы спросить человека перед вами, сколько людей перед ним? Поскольку этот человек будет находиться в той же ситуации, что и вы, ему придется задать тот же вопрос человеку перед ним, и так далее, пока вопрос не дойдет до первого человека в очереди. Этот человек может уверенно ответить, что перед ним нет людей. Затем второй человек в очереди может ответить «один», человек за ним отвечает «два» и так далее, пока ответ не дойдет до вас. Я понимаю, что шансы довольно малы, что все эти люди будут подыгрывать только для того, чтобы вы знали, где вы находитесь в очереди, но это полезный способ визуализировать, как работает рекурсия.

Как это переводится на язык программирования? В программировании рекурсия — это способ выполнения повторяющейся задачи путем вызова функции самой себя. Рекурсивная функция обычно вызывает саму себя с измененным параметром до тех пор, пока не будет достигнуто определенное условие. Это условие называется базовым случаем.

В наших предыдущих примерах базовым случаем была бы самая маленькая русская матрешка или человек в начале очереди.

```
def factorial(n):
    if n < 2:
        return 1
    return n * factorial(n-1)
```

Давайте рассмотрим пример рекурсивной функции, чтобы понять, о чем мы говорим. Здесь мы определяем функцию, называемую факториалом. В начале функции у нас есть условный блок, определяющий базовый случай, когда n меньше 2, и она просто возвращает значение 1.

После базового случая у нас есть строка, где функция факториала вызывает саму себя с n-1. Это называется рекурсивным случаем, и он создает цикл. Каждый раз, когда выполняется функция, она вызывает саму себя с меньшим числом, пока не достигнет базового случая. Достигнув базового случая, она возвращает значение 1, затем ранее вызванная функция умножает это значение на два, а ранее вызванная функция умножает его на три и так далее.

Этот цикл будет продолжаться до тех пор, пока первая вызванная функция факториала не вернет желаемый результат.

```
>>> def factorial(n):
...     print("Factorial called with " + str(n))
...     if n < 2:
...         print("Returning 1")
...         return 1
...     result = n * factorial(n-1)
...     print("Returning " + str(result) + " for factorial of " + str(n))
...     return result
...
>>> factorial(4)
Factorial called with 4
Factorial called with 3
Factorial called with 2
Factorial called with 1
Returning 1
Returning 2 for factorial of 2
Returning 6 for factorial of 3
Returning 24 for factorial of 4
24
>>>
```

Давайте добавим несколько операторов print, чтобы точно увидеть, как это работает. Здесь мы видим, что функция продолжала вызывать саму себя, пока не достигла базового случая. После этого каждая функция возвращала значение предыдущей функции, умноженное на n, пока исходная функция не вернула результат.

Возможно, вы задаетесь вопросом, зачем нам нужны рекурсивные функции, если можно просто использовать цикл for или while? Что ж, решения некоторых конкретных задач проще писать и понимать с использованием рекурсивных функций. Многие математические функции, например, факториал или сумма всех предыдущих чисел, являются хорошими примерами этого. Если математическая функция уже определена в рекурсивном виде, то легко просто написать код как рекурсивную функцию.

Но дело не только в математических функциях. Давайте рассмотрим пару примеров того, как это может помочь IT-специалисту, пытающемуся автоматизировать задачи.

Допустим, вам нужно написать инструмент, который проходит по множеству каталогов на вашем компьютере и вычисляет, сколько файлов содержится в каждом из них. При перечислении файлов внутри каталога, вы можете обнаружить внутри него подкаталоги, и вам также потребуется подсчитать количество файлов, в этих подкаталогах. Это отличный случай для использования рекурсии. Здесь базовый случай — это каталог, без подкаталогов. В этом случае функция просто вернет количество файлов. Рекурсивный случай — это вызов рекурсивной функции для каждого из содержащихся подкаталогов. Возвращаемое значение вызова данной функции будет суммой всех файлов в этом каталоге плюс всех файлов в содержащихся подкаталогах.

Каталог файлов, который может содержать другие каталоги, является примером рекурсивной структуры, поскольку каталоги могут содержать подкаталоги, которые содержат подкаталоги, которые содержат подкаталоги, и так далее.

При работе с рекурсивной структурой обычно проще использовать рекурсивные функции, чем циклы for или while.

Еще один пример рекурсивной структуры, ориентированный на ИТ, — это все, что работает с группами пользователей, которые могут содержать другие группы. Мы часто сталкиваемся с подобной ситуацией при использовании административных инструментов, таких как Active Directory или LDAP.

Допустим, ваше программное обеспечение для управления группами позволяет создавать группы, в состав которых входят как пользователи, так и другие группы, и вы хотите вывести список всех пользователей, входящих в данную группу. В этом случае вы бы использовали рекурсивную функцию для перебора групп.

Здесь базовый случай — это группа, которая включает только пользователей, и в этом случае будет выведен список всех пользователей. Рекурсивный случай будет означать перебор всех содержащихся групп, вывод списка всех пользователей в них, а затем вывод списка всех пользователей, входящих в текущую группу.

Важно отметить, что в некоторых языках существует максимальное количество рекурсивных вызовов. В Python по умолчанию можно вызвать рекурсивную функцию 1000 раз, пока не будет достигнут лимит. Это подходит для таких вещей, как подкаталоги или группы пользователей, которые не имеют тысяч уровней вложенности, но этого может быть недостаточно для математических функций.

```
>>> def factorial(n):
...     if n < 2:
...         return 1
...     return n * factorial(n-1)
...
>>> factorial(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  [Previous line repeated 995 more times]
  File "<stdin>", line 2, in factorial
RecursionError: maximum recursion depth exceeded in comparison
>>>
```

Вернемся к нашему примеру с факториалом и попробуем вызвать его, если n равно 1000, то есть факториал 1000. Видите ошибку? Она говорит нам, что мы достигли максимального предела для рекурсивных вызовов. Поэтому, хотя вы можете использовать рекурсию во множестве различных сценариев, мы рекомендуем использовать ее только тогда, когда вам нужно пройти через рекурсивную структуру, которая не достигнет 1000 уровней вложенности.

Обработка исключений

Давайте посмотрим, как работают исключения в Python.

```
# exception from Python itself
14 / 0
```

⊗ 0.3s

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/Users/alfredodeza/code/python-bootcamp/introduction-to-python/notebooks/week1/
line: 2>()
      1 # exception from Python itself
----> 2 14 / 0

ZeroDivisionError: division by zero
```

Исключениями являются случаи, когда Python прерывает выполнение кода и объявляет об ошибках в вашей программе.

Если мы запустим здесь этот код, в котором мы делим 14 на 0, мы получим очень красиво описанную ошибку, которая называется ошибкой деления на ноль.

```
# generating an exception with your own code
raise RuntimeError("this is a problem!")
```

⊗ 0.3s

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
/Users/alfredodeza/code/python-bootcamp/introduction-to-python/notebooks/week1/
line: 2>()
      1 # generating an exception with your own code
----> 2 raise RuntimeError("this is a problem!")

RuntimeError: this is a problem!
```

Вы также можете создавать исключения самостоятельно. В этом случае мы генерируем ошибку времени выполнения, которая является встроенным исключением. В скобках мы добавляем здесь текстовую строку, в которой говорится, что это ошибка. Вот как вы можете генерировать исключение. У вас может быть функция или другой фрагмент кода, и вы хотите прервать работу и вызвать исключение, используя ключевое слово raise.

```
try:
    # some intense operation that causes an error
    result = 14 / 0
except ZeroDivisionError:
    # do some other intense operation
    result = 14 / 2

print(result)
```

✓ 0.2s

```
# don't get tempted to catch all exceptions
try:
    # some intense operation that causes an error
    result = 14 / 0
    raise RuntimeError("error!")
except Exception:
    # do some other intense operation
    result = 14 / 2

print(result)
```

✓ 0.1s

И вы также можете перехватить исключение.

Что тут происходит? Ключевое слово try переведет выполнение кода на следующую строку, выполнит операцию деления 14 на 0, и эта попытка приведет к ошибке ZeroDivisionError.

Но вы сможете перехватить это исключение с помощью блока исключений except. Если в этом случае есть исключение, соответствующее тому, что вы ловите, в данном случае ZeroDivisionError, вы получите не исключение, а выполнение кода в блоке except.

Не поддавайтесь искушению перехватить здесь все исключения. Вы можете видеть, что в следующей ячейке я использую исключение Exception. Это очень общий и очень широкий набор исключений. Это означает, что будет перехватываться каждое исключение, которое только можно, почти каждое исключение, которое есть в Python.

Но проблема в том, что в этом случае в результате у нас есть ошибка деления на ноль. И у нас также есть ошибка времени выполнения. Вы можете получать различные типы исключений, но вы должны знать, откуда они берутся. Вы должны знать, о чем идет речь. Способ сделать это — перехватить несколько исключений.

```
try:
    # some intense operation that causes an error
    result = 14 / 0
    result + "100"
except (ZeroDivisionError, TypeError):
    # do some other intense operation
    result = 14 / 2

print(result)
```

✓ 0.2s

Здесь мы делим на ноль и прибавляем к числу строку. Поэтому мы ловим исключение ZeroDivisionError и исключение TypeError (неподдерживаемые типы операндов). Это потому, что мы используем результат, который представляет собой число с плавающей запятой, которое пытаемся добавить к строке.

Вы можете добавить столько типов исключений, сколько захотите. И если я сделаю это здесь, я поймаю обе или одну из этих ошибок ZeroDivisionError или TypeError.

```
# assign the resulting exception to a variable
try:
    # some intense operation that causes an error
    result = 14 / 0
except ZeroDivisionError as error:
    # do some other intense operation
    print(f"got an error --> {error}")
    result = 14 / 2

print(result)
```

✓ 0.2s

```
got an error --> division by zero
```

Наконец, вы также можете присвоить полученное исключение переменной. И способ присвоения исключения переменной – это использование ключевого слова as и далее имя переменной. Затем вы можете использовать эту переменную с оператором печати.

Строки

К настоящему моменту мы уже использовали строки во многих примерах, но еще не уделили им достаточно времени для детального изучения.

Во-первых, краткое напоминание. Строка — это тип данных в Python, который используется для представления фрагмента текста. Она записывается в кавычки, двойные или одинарные — на ваш выбор. Неважно, какой тип кавычек вы используете, главное, чтобы они совпадали. Если мы перепутаем двойные и одинарные кавычки, Python будет недоволен и вернет синтаксическую ошибку, сообщив, что он не смог найти конец строки.

```
>>> name = "Sasha"
>>> color = 'gold'
>>> place = "Cambridge'
  File "<stdin>", line 1
    place = "Cambridge'
                      ^
SyntaxError: EOL while scanning string literal
>>> pet = ""
>>> pet = "looooooooooooooooooooooooooooong cat"
>>> print("Name: " + name + ", Favorite color: " + color)
Name: Sasha, Favorite color: gold
>>> "example" * 3
'exampleexampleexample'
>>> len(pet)
38
>>>
```

Строка может быть как короткой, до нуля символов, обычно называемой пустой строкой, также очень длинной. Мы также можем использовать строки для создания более длинных строк с помощью знака плюс, действие, называемое конкатенацией.

Менее распространенная операция — это умножение строки на число, которое умножает содержимое строки на определенное количество раз.

Если мы хотим узнать длину строки, мы можем использовать функцию len. Функция len сообщает нам количество символов, содержащихся в строке.

Мы можем использовать строки для представления множества различных вещей. Они могут содержать имя пользователя, имя компьютера, адрес электронной почты, имя файла и любой другой текст. Большая часть данных, с которыми мы будем взаимодействовать, будет храниться в строках, поэтому важно знать, как их использовать.

Существует множество вещей, которые мы можем делать со строками в наших скриптах. Например, мы можем проверить, названы ли файлы определенным образом, посмотрев на имя файла и проверив, соответствуют ли они нашим критериям. Или мы можем создать список адресов электронной почты, проверив пользователей системы и объединив в домен.

Когда мы впервые столкнулись с циклами, мы отметили, что можем перебирать символы строки по одному. Но что, если нам нужен доступ только к определенному символу или символам? Например, если у нас есть текст, который слишком длинный для отображения, и мы хотим показать только его часть, или если мы хотим составить акроним, взяв первую букву каждого слова во фразе.

```
>>> name = "Jaylen"
>>> print(name[1])
a
>>> print(name[0])
J
>>> print(name[5])
n
>>> print(name[6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

Мы можем сделать это с помощью операции, называемой индексацией строки. Эта операция позволяет получить доступ к символу в заданной позиции или индексе, используя квадратные скобки и номер нужной позиции.

На первый взгляд это может показаться запутанным, так как мы запрашиваем первый символ, а получаем второй. В чем дело? Дело в том, что Python начинает отсчет индексов с нуля, а не с единицы, как это происходит с функцией range.

Итак, если нам нужен первый символ, нам нужно получить доступ к нулевому индексу. Зная, что индексы начинаются с нуля, как вы думаете, какой индекс будет последним в строке? Он всегда будет на единицу меньше, длины строки. В данном случае наша строка содержит шесть символов, поэтому последний индекс будет пятым.

И если мы попытаемся получить доступ к шестому индексу, мы получим ошибку индекса, сообщающую, что он выходит за пределы диапазона. Мы можем получить доступ только до длины минус один.

```
>>> text = "Random string with a lot of characters"
>>> print(text[-1])
s
>>> print(text[-2])
r
>>>
```

Что делать, если вы хотите вывести последний символ строки, но не знаете, какова длина? Это можно сделать, используя отрицательные индексы.

Использование отрицательных индексов позволяет нам получать доступ к позициям в строке, начиная с последнего символа.

```
>>> color = "Orange"
>>> color[1:4]
'ran'
>>>
```

Помимо доступа к отдельным символам, мы также можем получить доступ к фрагменту строки. Фрагмент — это часть строки, которая может содержать более одного символа, иногда также называемая подстрокой. Мы делаем это, создавая диапазон с помощью двоеточия в качестве разделителя.

Диапазон, который мы используем при доступе к фрагменту строки, работает так же, как и диапазон, создаваемый функцией range. Он включает первое число, но доходит до единицы меньше, чем последнее число. В этом случае мы начинаем с индекса 1, второй буквы строки, и доходим до индекса три, четвертой буквы строки.

```
>>> fruit = "Pineapple"
>>> print(fruit[:4])
Pine
>>> print(fruit[4:])
apple
>>>
```

Другой вариант для диапазона — включить только один из двух индексов. В этом случае предполагается, что другой индекс равен либо нулю для первого значения, либо длине строки для второго значения. Доступ к фрагменту от нуля до четырех берет первые четыре символа строки. Индексы от нуля до трех. Доступ к фрагменту от четвертого до нуля берет все данные, начиная с индекса четыре.

Итак, мы рассмотрели, как получить доступ к определенным символам внутри строки. А что, если мы захотим их изменить? Представьте, что у вас есть строка с неправильным символом, и вы хотите ее исправить. Учитывая то, что вы узнали об индексации строк, у вас может

возникнуть соблазн исправить ее, обратившись к соответствующему индексу и изменив символ.

```
>>> message = "A kong string with a silly typo"
>>> message[2] = "l"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Давайте посмотрим, что произойдет, если мы попробуем это сделать. В этом случае нам говорят, что строки не поддерживают присваивание элементов. Это означает, что мы не можем изменять отдельные символы, потому что строки в Python неизменяемы. Что мы можем сделать, так это создать новую строку на основе старой.

```
>>> new_message = message[0:2] + "l" + message[3:]
>>> print(new_message)
A long string with a silly typo
>>>
```

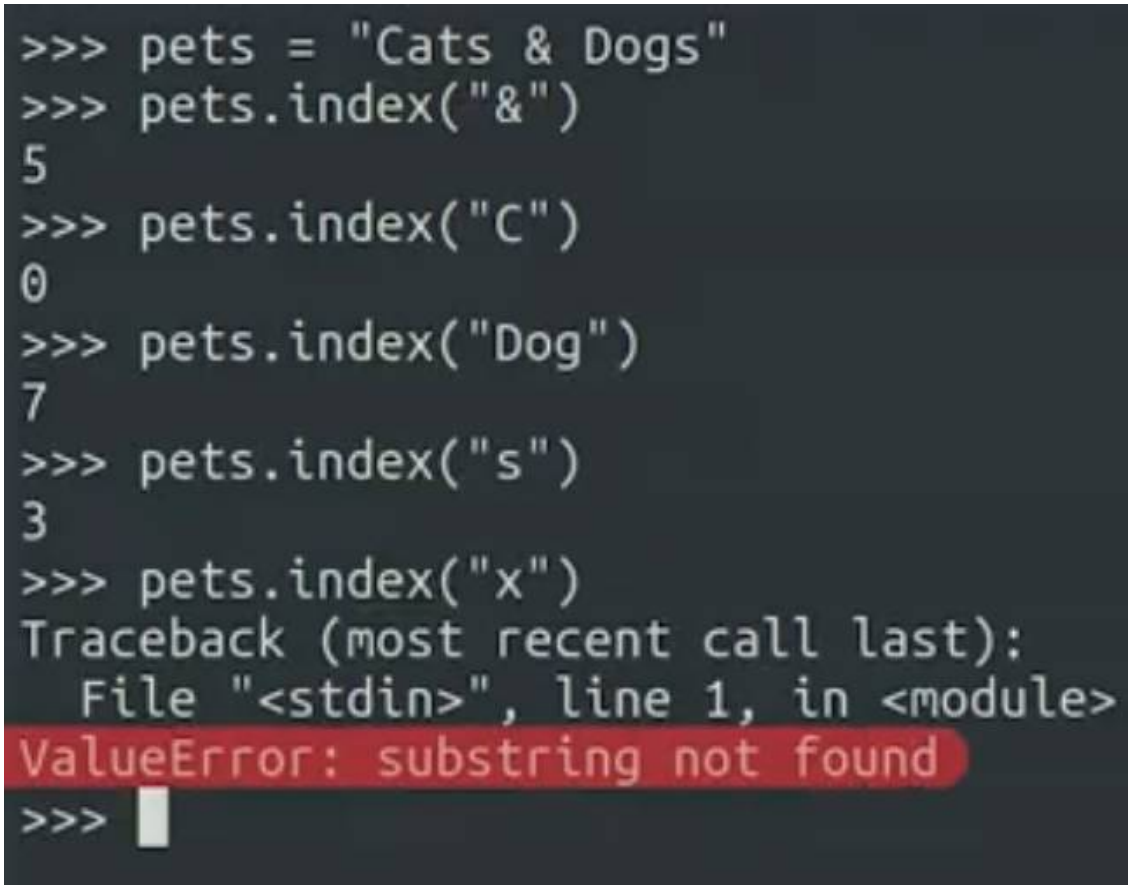
Отлично. Мы исправили опечатку. Но означает ли это, что переменная message никогда не может измениться? Не совсем. Мы можем присвоить новое значение той же переменной.

```
>>> message = "And another one"
>>> print(message)
And another one
>>>
```

Здесь мы присваиваем переменной `message` совершенно новое значение. Мы не меняем исходную строку, которая была ей присвоена ранее. Мы присваиваем совершенно новую строку с другим содержимым.

Мы выяснили, как создать новое сообщение из старого, но как нам узнать, какой символ нужно изменить?

В этом случае мы используем метод для получения индекса определенного символа.



Метод — это функция, связанная с определенным классом. Мы еще поговорим о классах и методах позже. А пока вам нужно знать, что это функция, которая применяется к переменной, и мы можем вызвать ее, поставив точку после переменной.

Метод «index» возвращает индекс заданной подстроки внутри строки. Подстрока, которую мы передаем, может быть любой длины.

Что если подстрок больше одной? Здесь мы знаем, что есть два символа «s», но получаем только одно значение. Это потому, что метод index возвращает только первую позицию, которая соответствует запросу.

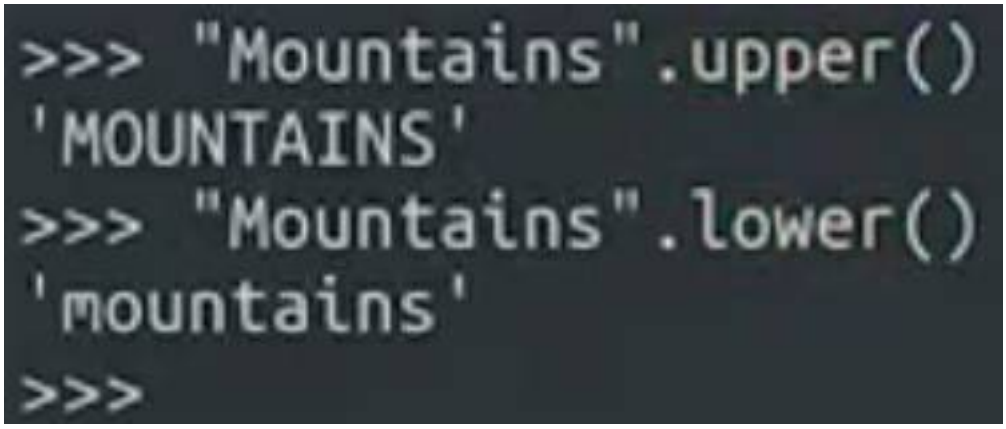
Что произойдет, если в строке нет искомой подстроки? Метод index не может вернуть число, потому что подстроки нет, поэтому вместо этого мы получим ошибку значения.

Как узнать, содержится ли подстрока в строке, чтобы избежать ошибки?



Давайте разберемся. Мы можем использовать ключевое слово in, чтобы проверить, содержится ли подстрока в строке. Мы можем встретить ключевое слово in при использовании циклов for. В том случае оно используется для итерации. Здесь это условие, которое может быть либо истинным, либо ложным.

До сих пор мы рассматривали способы доступа к частям строк с помощью индексации, создания новых строк путем нарезки и конкатенации, поиска символов в строках с помощью метода индекса и даже проверки наличия одной строки в другой. В дополнение ко всей этой мощи обработки строк, класс String предоставляет множество других методов для работы с текстом.



Строковые методы позволяют выполнять преобразования или форматирование текстовой строки, например, преобразование в верхний регистр и его противоположность — в нижний. Эти методы очень полезны при обработке пользовательского ввода.

```
>>> answer = 'YES'
>>> if answer.lower() == "yes":
...     print("User said yes")
...
User said yes
>>>
```

Допустим, вы хотите проверить, ответил ли пользователь да на вопрос. Как узнать, ввел ли пользователь ответ в верхнем или нижнем регистре? Вам это не нужно. Вы просто преобразуете ответ в нужный вам регистр.

```
>>> " yes ".strip()
'yes'
>>> " yes ".lstrip()
'yes '
>>> " yes ".rstrip()
' yes'
>>>
```

Еще один полезный метод при работе с пользовательским вводом — это метод strip. Этот метод удаляет окружающие пробелы в строке. Если мы запрашиваем у пользователя ответ, нас обычно не волнуют окружающие пробелы. Рекомендуется использовать метод strip для удаления любых пробелов. Это означает, что strip удаляет не только пробелы, но и символы табуляции и символы новой строки, которые обычно не нужны в строках, предоставленных пользователем.

Существуют еще две версии этого метода: L strip и R strip, которые удаляют пробелы слева или справа от строки, а не с обеих сторон.

```
>>> "The number of times e occurs in this string is 4".count("e")
4
>>>
```

```
>>> "Forest".endswith("rest")
True
>>>
```

```
>>> "Forest".isnumeric()
False
>>> "12345".isnumeric()
True
>>> int("12345") + int("54321")
66666
>>>
```

Другие методы предоставляют информацию о самой строке.

Метод count возвращает количество раз, когда заданная подстрока встречается в строке.

Метод ends возвращает, заканчивается ли строка определенной подстрокой.

Метод is numeric возвращает, состоит ли строка только из чисел.

Кроме того, если у нас есть строка, содержащая числа, мы можем использовать функцию int для преобразования ее в числовое значение.

```
>>> " ".join(["This", "is", "a", "phrase", "joined", "by", "spaces"])
'This is a phrase joined by spaces'
>>> "...".join(["This", "is", "a", "phrase", "joined", "by", "triple", "dots"])
'This...is...a...phrase...joined...by...triple...dots'
>>>
```

Ранее мы показывали, что можем объединять строки, используя знак плюс. Метод join также можно использовать для объединения. Чтобы использовать метод join, необходимо вызвать его для строки, которая будет использоваться для объединения.

```
>>> "This is another example".split()
['This', 'is', 'another', 'example']
>>>
```

Наконец, мы также можем разделить строку на список строк. Метод «split» возвращает список всех слов в исходной строке и автоматически разделяет ее по любому пробелу. Он может дополнительно принимать параметр и разделять строки по другому символу, например, по запятой или точке.

До сих пор мы создавали строки, используя знак плюса для объединения только тех частей строки, которые хотели создать, и использовали функцию str(),

для преобразования чисел в строки, чтобы мы могли их тоже объединять. Это работает, но это не идеально, особенно когда операции, которые вы хотите выполнить со строкой, довольно сложные.

Есть лучший способ сделать это, используя метод format. Давайте рассмотрим пару примеров.

```
>>> name = "Manny"
>>> number = len(name) * 3
>>> print("Hello {}, your lucky number is {}".format(name, number))
Hello Manny, your lucky number is 15
>>>
```

В этом примере у нас есть две переменные: name и number. Мы генерируем строку, содержащую эти переменные, используя фигурные скобки в качестве заполнителя, чтобы показать, где должны быть записаны переменные. Затем мы передаем переменные в качестве параметра методу format. Обратите внимание, что не имеет значения, что name — это строка, а number — целое число. Метод format обрабатывает это, поэтому нам не нужно этого делать. Довольно удобно.

```
>>> print("Your lucky number is {number}, {name}.".format(name=name, number=len(name)*3))
Your lucky number is 15, Manny.
>>>
```

Фигурные скобки не всегда пустые. Используя определённые выражения внутри этих скобок, мы можем воспользоваться всей мощью форматирования.

Одна из вещей, которые мы можем поместить внутрь фигурных скобок, это имя переменной, которую мы хотим разместить в этой позиции, чтобы сделать всю строку более читабельной. Это особенно актуально, когда текст может быть переписан или переведён, и переменные могут поменяться местами.

```
>>> price = 7.5
>>> with_tax = price * 1.09
>>> print(price, with_tax)
7.5 8.175
>>> print("Base price: ${:.2f}. With Tax: ${:.2f}".format(price, with_tax))
Base price: $7.50. With Tax: $8.18
>>>
```

Рассмотрим другой пример. Допустим, вы хотите вывести цену товара с налогом и без налога. В зависимости от ставки налога, число может быть длинным числом с множеством десятичных знаков.

Чтобы это исправить, мы можем заставить функцию format выводить только два десятичных знака. В этом случае между фигурными скобками мы пишем выражение форматирования.

Существует множество различных выражений, которые мы можем написать. Эти выражения необходимы, когда мы хотим указать Python форматировать наши значения иначе, чем это делается по умолчанию.

Выражение начинается с двоеточия, отделяющего его от имени поля. После двоеточия мы пишем .2f. Это означает, что мы собираемся отформатировать число с плавающей запятой, и после десятичной точки должно быть две цифры. Независимо от цены, наша функция будет всегда выводить два десятичных знака.

```
>>> def to_celsius(x):
...     return (x-32)*5/9
...
>>> for x in range(0,101,10):
...     print("{:>3} F | {:>6.2f} C".format(x, to_celsius(x)))
...
  0 F | -17.78 C
 10 F | -12.22 C
 20 F |  -6.67 C
 30 F |  -1.11 C
 40 F |   4.44 C
 50 F |  10.00 C
 60 F |  15.56 C
 70 F |  21.11 C
 80 F |  26.67 C
 90 F |  32.22 C
100 F |  37.78 C
>>>
```

В этих двух выражениях мы используем оператор «больше», чтобы выровнять текст по правому краю. В первом выражении мы хотим, чтобы числа были выровнены по правому краю на три пробела. Во втором выражении мы хотим, чтобы число всегда имело ровно два десятичных знака и было выровнено по правому краю на отметке в шесть пробелов.

Мы можем использовать форматирование строк таким образом, чтобы вывод программы выглядел красиво, а также для генерации полезных сообщений в логах и отладочных сообщений.

Существует множество других параметров форматирования, которые вы можете использовать, когда они вам понадобятся.

Введение в структуры данных Python

Ключевые термины

Список (List) - упорядоченная коллекция значений, заключенных в квадратные скобки []. Используется для хранения последовательностей элементов.

Индекс (Index) - числовая позиция элемента в списке. Начинается с 0 для первого элемента. Используется для доступа к элементам по позиции.

Итерация - повторное выполнение кода на последовательных элементах списка. В Python выполняется с помощью цикла for-in.

Словарь (Dictionary) - неупорядоченная коллекция пар ключ-значение, обозначаемых фигурными скобками {}. Ключи сопоставляются с соответствующими значениями.

Ключ (Key) - уникальный идентификатор, который используется для поиска значений в словаре. Поиск осуществляется очень быстро.

Значение (Value) - данные, связанные с данным ключом в словаре. Значения могут быть любого типа данных Python.

Кортеж (Tuple) - неизменяемая упорядоченная коллекция фиксированного размера, похожая на список. Обозначается символом (). Пригодится, когда данные не должны меняться.

Набор (Set) - неупорядоченная коллекция уникальных объектов. Полезно для удаления дубликатов и операций с наборами.

Членство (Membership) - возможность проверить, содержится ли значение в коллекции, такой как списки, словари, кортежи или наборы.

Методы (Method) - встроенные функции, позволяющие манипулировать и взаимодействовать со структурами данных.

Итерация - процесс многократного выполнения кода над каждым элементом коллекции по очереди.

```
# LIST
nums = [1, 2, 3] # Ordered collection in square brackets
print(nums[0]) # Access first item using INDEX 0

for n in nums: # ITERATE over list
  print(n)

# DICTIONARY
airports = {"SFO": "San Francisco", "LAX": "Los Angeles"} # Key-value pairs in {}
print(airports["SFO"]) # Use KEY to access VALUE

for code in airports:
  print(airports[code]) # VALUE for each KEY

```

```
15  # TUPLE
16  colors = ("red", "green", "blue") # Immutable ordered collection
17  print(colors[2])
18
19  # SET
20  unique_codes = {"SFO", "LAX", "SFO"} # Only 1 SFO
21  print(len(unique_codes)) # Length is 2 (no duplicates)
22  unique_codes.pop() # Remove item (no index needed)
23
24  # MEMBERSHIP check with IN
25  print("SFO" in airports) # True, has key
26
27  # METHOD to add items
28  unique_codes.add("ORD")
```

```
1
1
2
3
San Francisco
San Francisco
Los Angeles
blue
2
True
```

Структуры данных Python являются основой для записи, хранения и извлечения данных.

Список – это одна из наиболее распространенных структур данных в Python. По сути, она представляет собой просто контейнер с элементами.

```
# lists are containers of items identified by square brackets
[1, 2, 3, 4, 5]
```

```
# lists can contain different types of items in them
[1, "two", False, 12.5]
```

Как вы можете видеть, здесь у нас есть целые числа. Но на самом деле внутри этих квадратных скобок может быть любой тип. Списки определяются квадратными скобками, и когда вы их определяете, вы можете вносить любой тип данных - целое число или строку, логическое значение и число с плавающей запятой, и все это будет работать. Хорошо, что нет никаких ограничений на то, что вы можете добавить в эти контейнеры. Это список, и мы можем рассматривать его скорее как контейнер, куда мы можем помещать элементы, помещать данные и затем работать с ними.

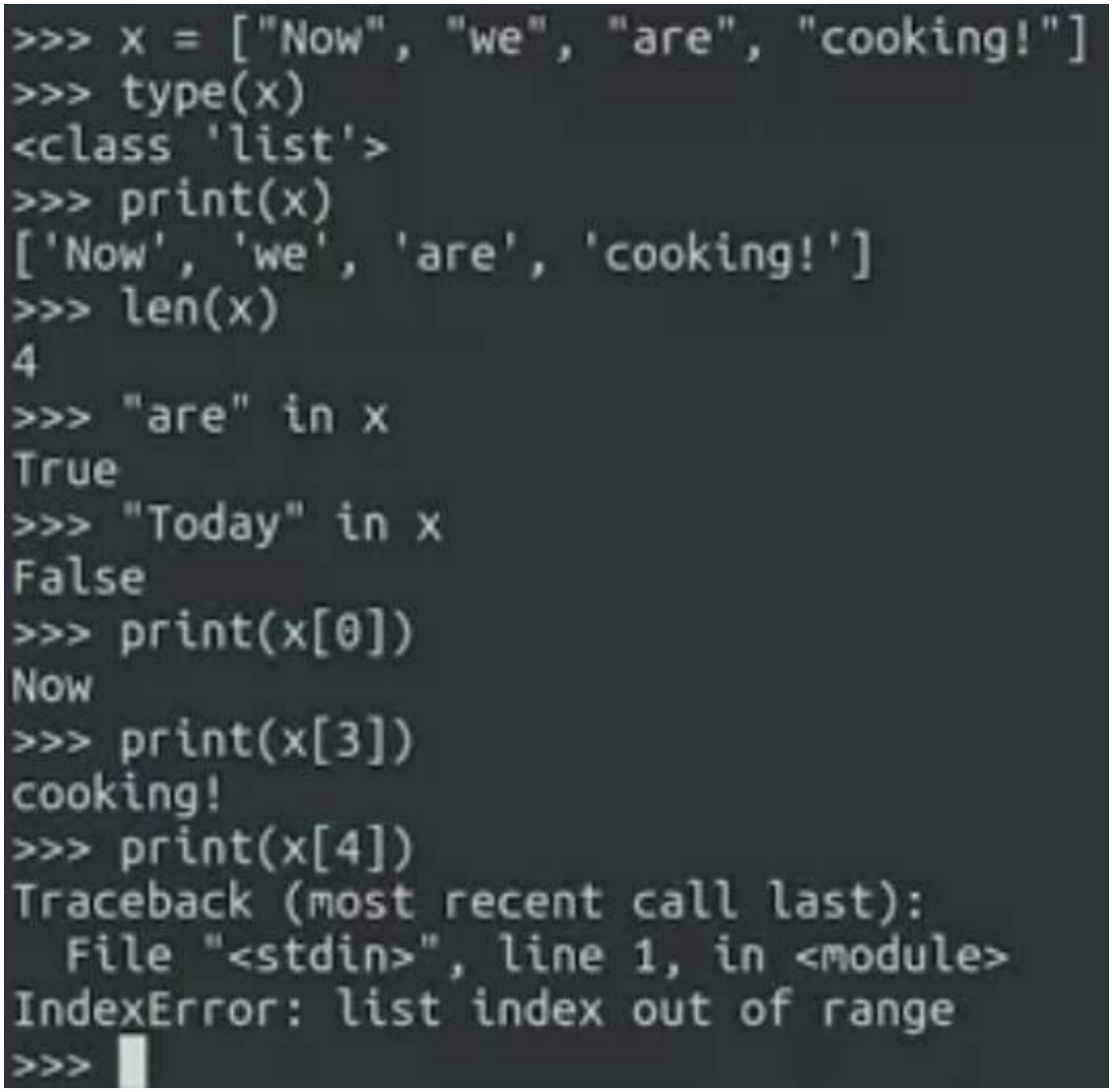
```
# you can use a built-in to count items in a list
len([3, 4, "red", "car"])
```

✓ 0.2s

```
4
```

Как вы уже знаете, Python поставляется с множеством готовых к использованию типов данных. Мы рассматривали целые числа, числа с плавающей запятой, логические значения и строки, но этих типов данных недостаточно. В конечном итоге, в ваших скриптах вам понадобится код, который будет манипулировать коллекциями элементов, например, списком строк, представляющих все имена файлов в каталоге, или списком целых чисел, представляющих размер сетевых пакетов. Вот где пригодится тип данных «список».

Вы можете представить списки как длинные коробки, пространство внутри которых разделено на разные ячейки. Каждая ячейка может содержать различное значение. И мы используем квадратные скобки, чтобы указать, где список начинается и заканчивается.



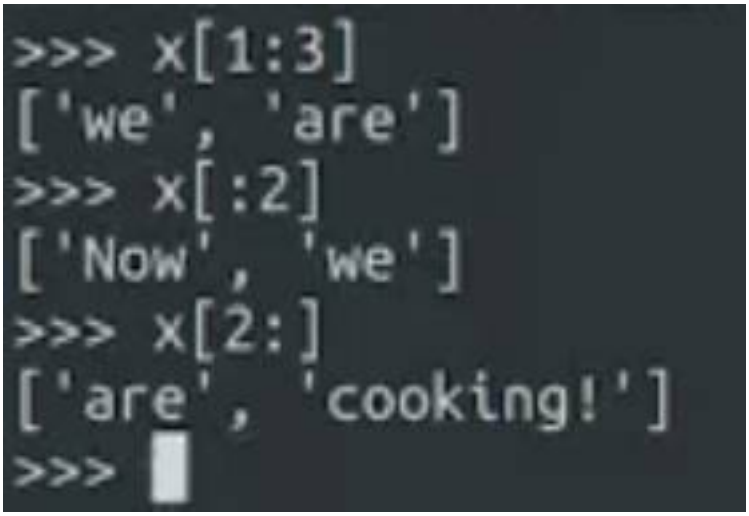
Давайте рассмотрим пример. Здесь мы создали новую переменную с именем x и установили ее содержимое в виде списка строк. Мы можем проверить тип x, используя функцию type.

И Python сообщает нам, что это список так же, как мы это делали с другими переменными.

Мы можем показать содержимое всего списка, используя функцию print.

Длина списка — это количество его элементов. Чтобы получить это значение, мы будем использовать ту же функцию len, которую использовали для строк.

Чтобы проверить, содержит ли список определенный элемент, вы можете использовать ключевое слово in, как в этих примерах.



Мы также можем использовать индексацию для доступа к отдельным элементам в зависимости от их положения в списке. Для этого мы используем квадратные скобки и индекс, к которому хотим получить доступ точно так же, как мы делали со строками. Помните, что первому элементу присваивается индекс 0. Это означает, что последний индекс списка будет равен длине списка минус 1.

Что произойдет, если мы попытаемся получить доступ к элементу после конца списка? Мы получим ошибку индекса, мы не можем перейти за конец списка.

В Python строки и списки являются примерами последовательностей. Существуют и другие последовательности, и все они имеют ряд общих операций, таких как итерация по ним, использование циклов for, индексирование, использование функции len для определения длины последовательности, использование оператора + для конкатенации двух последовательностей и использование оператора in для проверки, содержит ли последовательность элемент.

```
# each item has a position called "index". Count starts with 0
items = ["carrots", "peas", "celery"]

# retrieving an item is done with the index
items[0]
```

✓ 0.2s

```
'carrots'
```

Таким образом, вы также можете определить список, присвоив его переменной, а затем получать данные списка по индексу.

Что это значит? Каждый элемент в списке имеет индекс, позицию, и в Python первый индекс начинается с нулевого номера, поэтому элемент с индексом 0 и является первым элементом.

Когда вы хотите получить элемент списка, вы используете квадратные скобки и целое число индекса, которое даст элемент в этом списке.

```
# you can also retrieve the last item
items[-1]
```

✓ 0.3s

```
'celery'
```

```
# but also use negative indexes to count from the end
items[-2]
```

✓ 0.2s

```
'peas'
```

Однако со списками можно делать и другие хитрые приемы. Вы можете использовать отрицательные целые числа, которые будут давать элементы не с начала, а с конца списка.

Индексация может сбивать с толку, потому что она начинается с нуля. Некоторые люди ожидают, что это будет единица.

```
# create one by defining an empty list with bracket
items = []
items
```

✓ 0.2s

```
[]
```

```
# but you can also use the built-in `list()` in Python is more common to use brackets
items = list()
items
```

0.2s

```
[]
```

Одно из отличий списков от строк заключается в том, что списки являются изменяемыми, что означает, что они могут изменяться. Это значит, что мы можем добавлять, удалять или изменять элементы в списке.

```
>>> fruits = ["Pineapple", "Banana", "Apple", "Melon"]
>>> fruits.append("Kiwi")
>>> print(fruits)
['Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi']
>>>
```

В этом примере метод «append» добавляет новый элемент в конец списка. Неважно, какой длины список, элемент всегда добавляется в конец. Можно начать с пустого списка и добавить все его элементы с помощью «append».

```
>>> fruits.insert(0, "Orange")
>>> print(fruits)
['Orange', 'Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi']
>>>
```

Если вы хотите вставить элемент в другую позицию, а не в конец, можно использовать метод «insert». Метод «insert» принимает индекс, в качестве первого параметра, и элемент в качестве второго параметра. Он добавляет элемент по этому индексу в списке. Чтобы добавить его первым элементом, мы используем индекс ноль, и мы можем использовать любое другое число.

```
>>> fruits.insert(25, "Peach")
>>> print(fruits)
['Orange', 'Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi', 'Peach']
>>>
```

Что произойдет, если мы используем число больше длины списка? Ура, никаких ошибок. Если мы используем индекс больше, чем текущая длина, элемент просто добавляется в конец.

```
>>> fruits.remove("Melon")
>>> print(fruits)
['Orange', 'Pineapple', 'Banana', 'Apple', 'Kiwi', 'Peach']
>>>
```

Мы также можем удалять элементы из списка. Мы можем сделать это, используя значение элемента, который хотим удалить. Метод remove удаляет из списка первое вхождение переданного нам элемента.

```
>>> fruits.remove("Pear")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>
```

Что произойдет, если элемента нет в списке? Мы получим ошибку значения, сообщающую нам, что элемент отсутствует в списке.

```
>>> fruits.pop(3)
'Apple'
>>> print(fruits)
['Orange', 'Pineapple', 'Banana', 'Kiwi', 'Peach']
>>> fruits[2] = "Strawberry"
>>> print(fruits)
['Orange', 'Pineapple', 'Strawberry', 'Kiwi', 'Peach']
```

Другой способ удаления элементов — использование метода pop, который принимает индекс. Метод pop возвращает элемент, который был удален по переданному индексу. И последний способ изменить содержимое списка — изменить элемент, присвоив ему другое значение.

Теперь давайте посмотрим, как мы можем создавать списки и перебирать элементы списка.

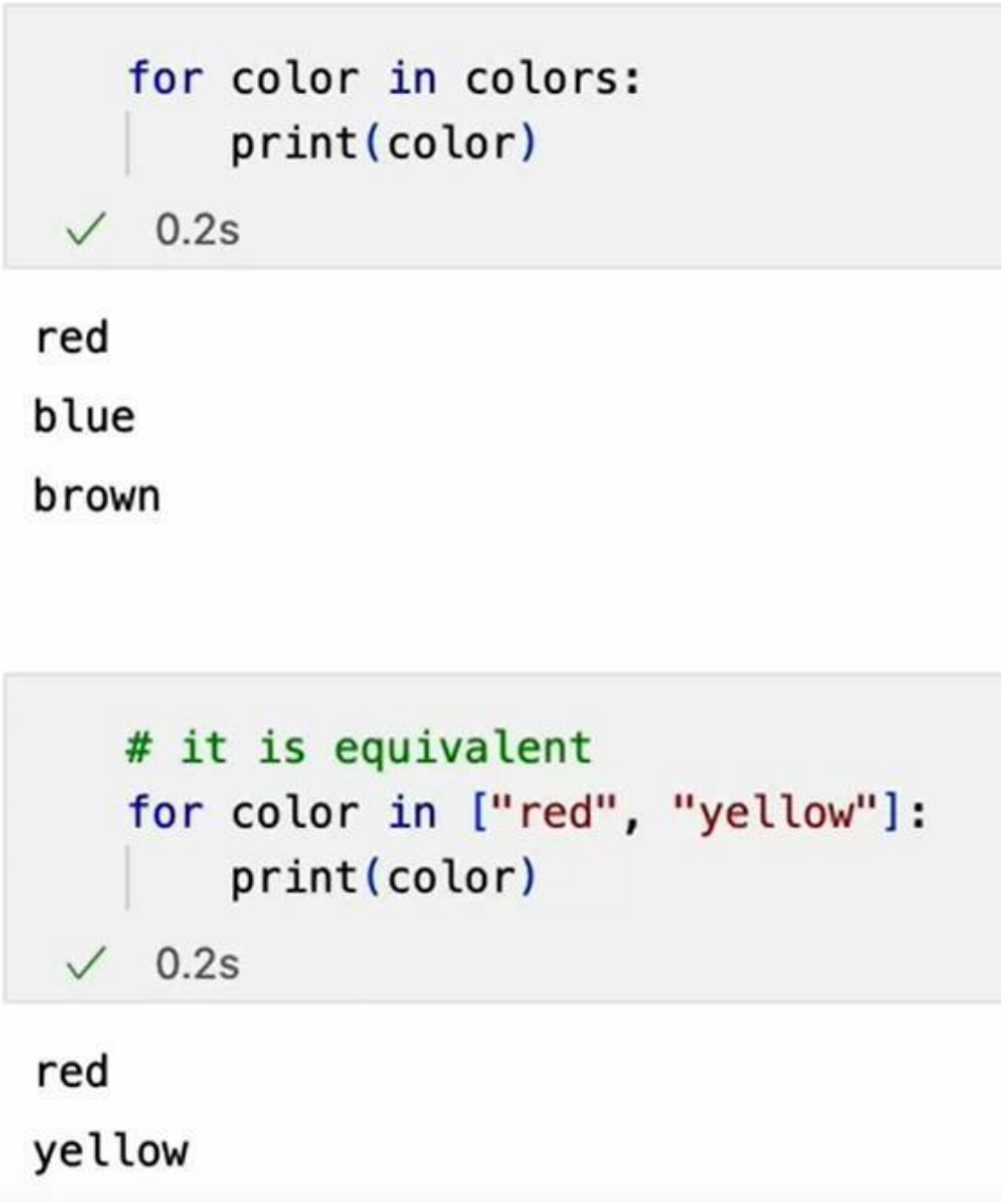
В основном вы всегда используете квадратные скобки, но вы также можете использовать встроенную функцию под названием list.

```
# data can be pre-seeded
colors = ["red", "blue", "brown"]
colors
```

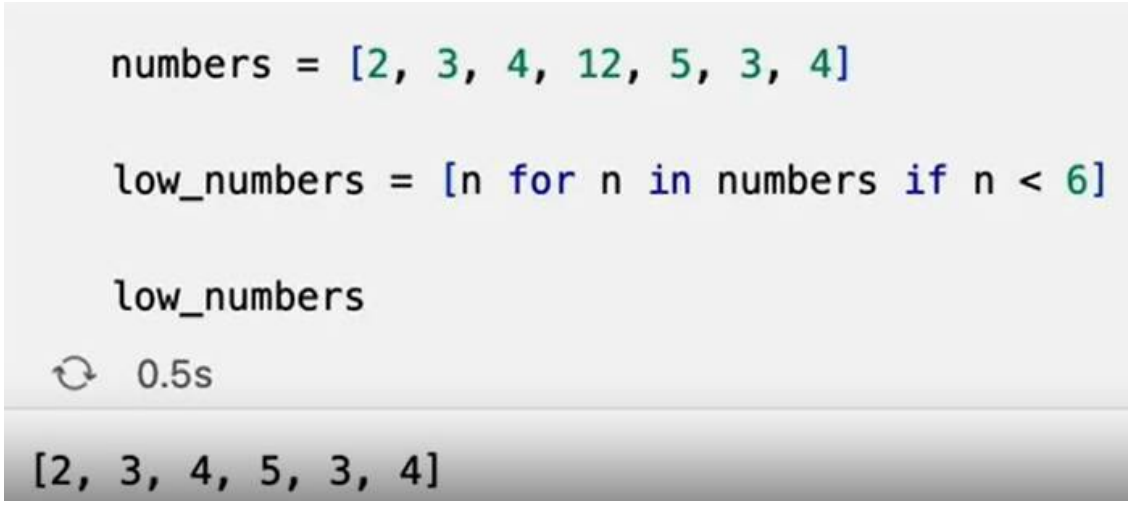
✓ 0.2s

```
['red', 'blue', 'brown']
```

Внутри скобок вы сразу можете указать элементы списка через запятую.



Теперь, вы можете перебирать список в цикле, что является одной из наиболее распространенных операций в Python. Это можно сделать с помощью ключевого слова for, после которого вы указываете временную переменную для сохранения элемента из списка, и дальше говорите in и название списка.



Далее давайте рассмотрим списковое включение или list comprehensions — это упрощенный подход к созданию списка.

Здесь вы должны быть осторожны, потому что list comprehensions — это довольно круто, и вы можете добавлять туда условные выражения, но в результате код может стать немного сложно читаемым – если они слишком длинные, их может быть трудно читать, и иногда с условными выражениями легко попасть в неприятности.

Здесь у меня есть список чисел, и далее на его основе я создаю другой список, используя цикл for и условное выражение.

Это, по сути, цикл внутри списка в квадратных скобках – n для любых чисел, если n меньше шести.

Это выражение выполнит цикл по всем числам, и, если число меньше шести, оно динамически добавится в этот список.

```
# curly brackets are required to create one
{}
```

```
# you always map a key to a value
{"key": "value"}
```

```
# the values can be other types
{"key": True}
```

Теперь давайте поговорим о словарях. Словари Python — одна из моих любимых структур данных в Python. В отличие от списка, который представляет собой контейнер элементов, словарь — это отображение элементов. Один из примеров, который мне обычно нравится приводить, — это телефонная книга.

Вы знаете, что здесь есть имя, и это имя соответствует номеру телефона. Телефонная книга — прекрасный пример словаря, представляя собой отображение имени в номер. В словаре у вас всегда будет ключ и значение.

Для создания словаря необходимы фигурные скобки, но вы можете создать словарь и другими способами, о которых мы поговорим дальше. Это всего лишь введение, и всякий раз, когда вы видите подобные фигурные скобки, вам следует подумать о словаре.

Здесь ключ — это строка, и значение также является строкой, но вы также можете сопоставить другие структуры данных и другие типы, например логическое значение.

```
# but the key has to be unique, there can't be duplicates
{"name": "Alfredo", "name": "Alfredo"}
```

✓ 0.2s

```
{'name': 'Alfredo'}
```

Если вы хотите указать в словаре более одного отображения ключ-значение, вы должны разделить их запятой, как в списках, когда у вас много разных элементов. И единственное ограничение заключается в том, что они должны быть уникальными. Здесь вы можете видеть,

что имя Альфредо повторяется. И если я запущу этот код, вы увидите, что я получу только одно отображение.

Python требует, чтобы ключи всегда были уникальными. В данном случае они не уникальны, и здесь у меня будет только одно сопоставление.

```
# value can be other dictionaries or lists
{"items": ["lumber", "concrete", "nails"]}
```

✓ 0.2s

```
{'items': ['lumber', 'concrete', 'nails']}
```

```
# you can't have a list or a dict as a key however!
{[1,2]: False}
```

⊗ 0.2s

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/alfredodeza/code/python-bootcamp/introduction-to-python/notebooks/week2
ll 7' in <cell line: 2>()
      1 # you can't have a list or a dict as a key however!
----> 2 {[1,2]: False}

TypeError: unhashable type: 'list'
```

Мы также можем использовать другие типы структур данных. Здесь у меня есть список предметов, который сопоставлен с ключом items. Но вы должны быть осторожны, потому что это не означает, что вы можете использовать список в качестве ключа, который сопоставляется со значением, у вас возникнут проблемы – вы получите ошибку типа.

```
# with curly brackets
contact_information = {}
contact_information
```

✓ 0.2s

```
{}
```

```
# with curly brackets and pre-seeded data
contact_information = {"name": "Alfredo", "lastname": "Deza"}
contact_information
```

0.2s

```
{'name': 'Alfredo', 'lastname': 'Deza'}
```

```
# with the dict() built-in
contact_information = dict()
contact_information
```

✓ 0.2s

```
{}
```

```
# with dict() and a list of tuple pairs
data = [("name", "alfredo"), ("lastname", "deza")]
dict(data)
```

Вы можете создать словарь с помощью фигурных скобок с предварительно заполненными данными. Но вы также можете создать его с помощью встроенной функцией dict.

При этом вы можете передать словарь как список разделенных пар, например кортежей, и создать на его основе словарь.

```
# with dict() and keyword arguments
dict(first="alfredo", lastname="deza")
```

И еще один способ создания словаря — использование в качестве аргументов функции dict ключевых слов.

```
contact_information = {
    "name": "Alfredo",
    "lastname": "Deza",
    "age": 49,
    "height": 1.90,
}
```

✓ 0.3s

```
# Retrieve only keys by default or explicitly
#for key in contact_information:
#    print(key)

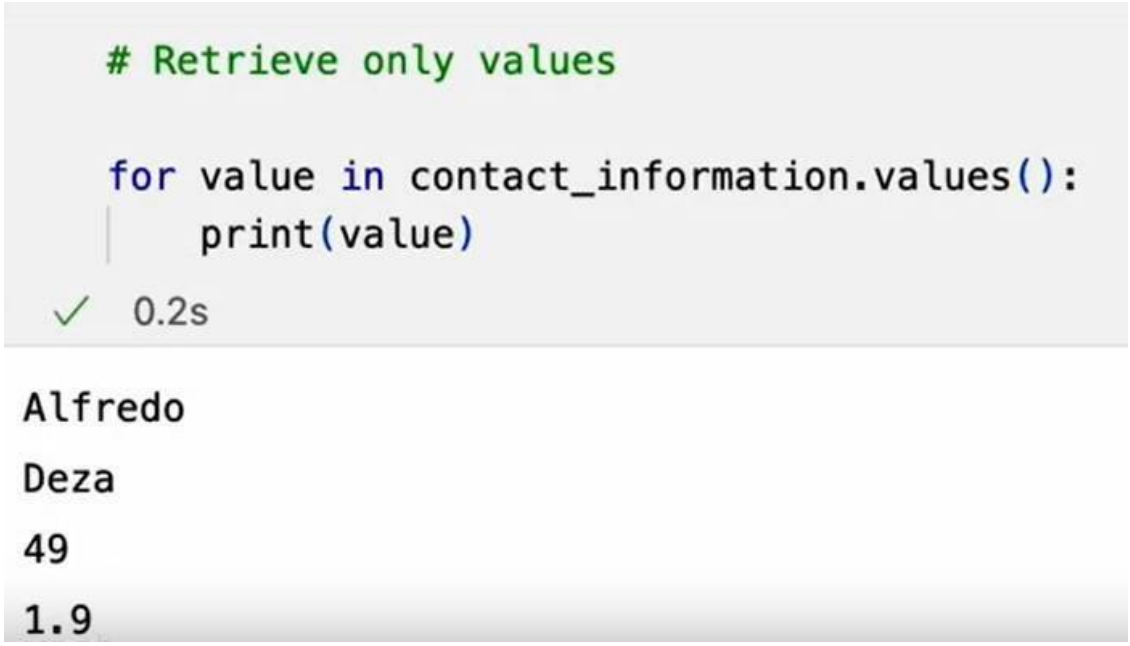
# explicit
for key in contact_information.keys():
    print(key)
```

0.1s

Теперь давайте посмотрим, как можно перебирать элементы в словаре. Здесь мы создали словарь contact_information. Если вы хотите получить только ключи, вы можете сделать это с помощью функции keys словаря в цикле for.

```
# Retrieve only keys by default or explicitly
for key in contact_information:
    print(key)
```

По умолчанию для перебора только ключей словаря вы можете опустить функцию keys и получите тот же самый результат. Здесь вместо key вы можете использовать любое другое название переменной. Это может быть любое слово.



```
# Retrieve only values

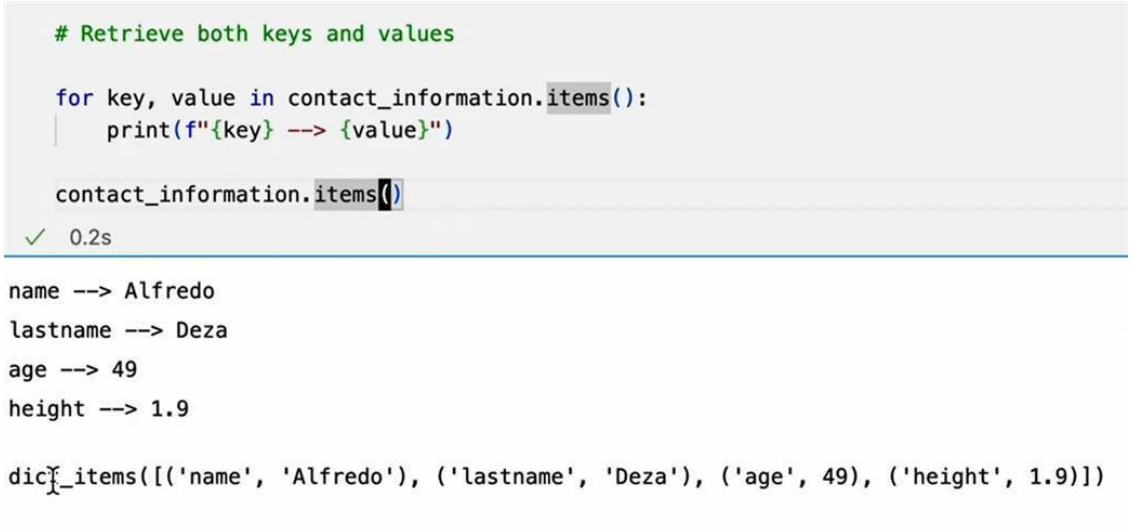
for value in contact_information.values():
    print(value)
```

✓ 0.2s

```
Alfredo
Deza
49
1.9
```

Но что, если вы захотите получить значения ключей словаря? Вы можете получить к ним доступ с помощью функции values.

Но что, если вам нужны и ключи, и значения?



```
# Retrieve both keys and values

for key, value in contact_information.items():
    print(f"{key} --> {value}")

contact_information.items()
```

✓ 0.2s

```
name --> Alfredo
lastname --> Deza
age --> 49
height --> 1.9

dict_items([('name', 'Alfredo'), ('lastname', 'Deza'), ('age', 49), ('height', 1.9)])
```

Для этого вы можете использовать функцию items для словаря. И в Python это называется распаковкой, вам придется использовать здесь две переменные, ключ и значение, и Python будет выполнять цикл и выдавать два значения за раз. И сам этот объект выглядит как список пар элементов.

Есть еще две структуры данных, о которых я хочу поговорить. В Python имеется множество структур данных, и вы также можете создавать свои собственные структуры данных со своими правилами. Но есть еще две структуры, которые, на мой взгляд, часто используются.

```
ro_items = ('first', 'second', 'third')
print("first item in the tuple is: %s" % ro_items.index('first'))
print(ro_items[-1])
for item in ro_items:
    print(item)
```

✓ 0.2s

```
first item in the tuple is: 0
third
first
second
third
```

Первая структура данных — это кортеж, и кортеж похож на список, но его нужно рассматривать как список, доступный только для чтения. И вы получаете доступ к индексам в кортеже точно так же, как и к списку.

Кортеж сохраняет порядок, но не допускает никаких изменений, и давайте посмотрим, что произойдет, если мы запросим несуществующий индекс.

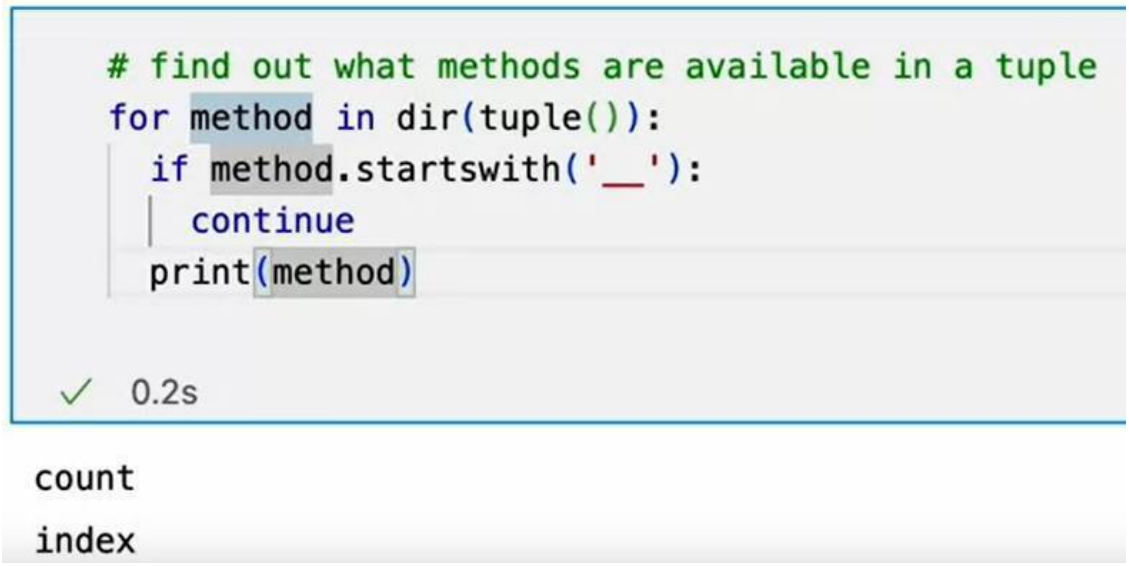
```
# expect an error here, just like a list!
ro_items[9]
```

⊗ 0.2s

```
---------------------------------------------------------
IndexError                                Traceback
/Users/alfredodeza/code/python-bootcamp/introduction
2' in <cell line: 2>()
      1 # expect an error here, just like a list!
----> 2 ro_items[9]

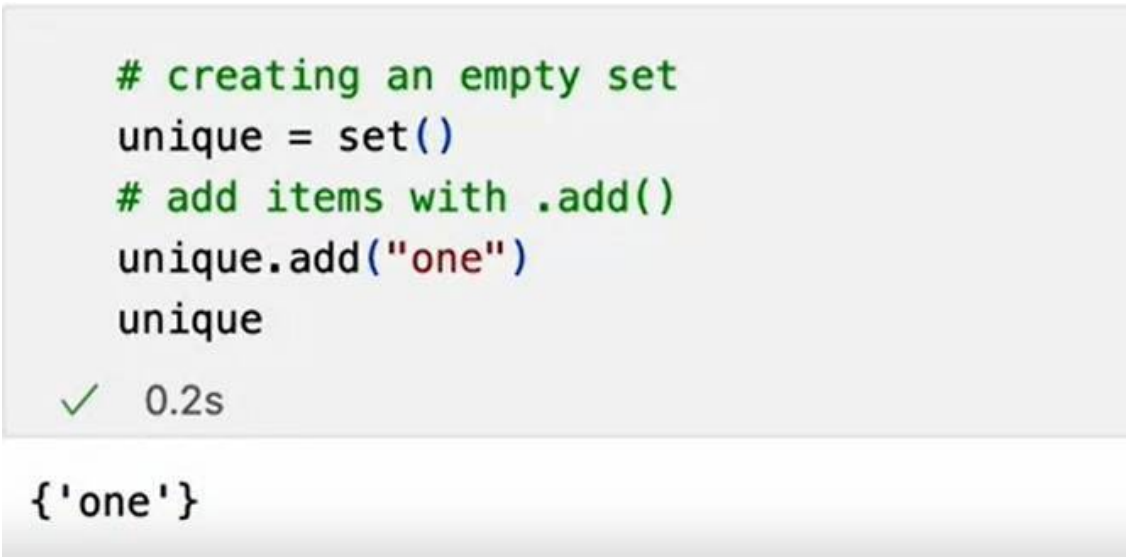
IndexError: tuple index out of range
```

Мы получим ошибку индекса, поэтому здесь нет 10 элементов. И если мы попытаемся получить индекс элемента, который не существует, мы получим ошибку значения.



Теперь, какие методы доступны для кортежа. У нас есть очень изящный цикл for, с помощью которого мы можем немного покопаться в объекте кортежа и найти все доступные методы для него. И у нас есть только метод count и метод index.

Создав кортеж, вы сможете только подсчитывать элементы и получать индексы. Вы не можете ничего сделать с кортежем. Он только для чтения.



В Python существует ряд типов данных, которые являются последовательностями. Строки — это последовательности символов и являются неизменяемыми. Списки — это последовательности элементов любого типа и являются изменяемыми. Кортежи — это последовательности элементов любого типа, которые являются неизменяемыми. Мы пишем кортежи в скобках вместо квадратных скобок.

Вы можете задаться вопросом: зачем нам вообще еще один тип последовательности? Разве списки не были замечательными? Да, списки замечательны, они могут содержать любое количество элементов, и мы можем добавлять, удалять и изменять их содержимое сколько угодно раз. Но бывают случаи, когда мы хотим убедиться, что элемент в определенной позиции

или индексе ссылается на что-то конкретное и не изменится. В таких ситуациях списки нам не помогут.

```
>>> fullname = ('Grace', 'M', 'Hopper')
>>>
```

В этом примере у нас есть кортеж, представляющий полное имя человека. Первый элемент кортежа — это имя. Второй элемент — это инициал отчества, а третий элемент — это фамилия. Если мы добавим еще один элемент где-нибудь внутри, что будет представлять этот элемент? Это просто запутает, и наш код не будет знать, что с ним делать. Именно поэтому модификация не допускается. Другими словами, при использовании кортежей положение элементов внутри кортежа имеет значение.

Кортежи используются для множества различных вещей в Python. Один из распространенных примеров — это возвращаемое значение функций. Когда функция возвращает более одного значения, она фактически возвращает кортеж.

```
>>> def convert_seconds(seconds):
...     hours = seconds // 3600
...     minutes = (seconds - hours * 3600) // 60
...     remaining_seconds = seconds - hours * 3600 - minutes * 60
...     return hours, minutes, remaining_seconds
...
>>> result = convert_seconds(5000)
>>> type(result)
<class 'tuple'>
>>> print(result)
(1, 23, 20)
>>>
```

Например, функция для преобразования секунд в часы, минуты и секунды. Эта функция возвращает три значения, другими словами, она возвращает кортеж из трех элементов.

Помните, поскольку это кортеж, порядок имеет значение. Первый элемент представляет часы, второй представляет минуты, а третий представляет секунды.

```
>>> hours, minutes, seconds = result
>>> print(hours, minutes, seconds)
1 23 20
>>> hours, minutes, seconds = convert_seconds(1000)
>>> print(hours, minutes, seconds)
0 16 40
```

Одна из интересных вещей, которые мы можем делать с кортежами, — это распаковывать их. Это означает, что мы можем превратить кортеж из трех элементов в три отдельные переменные. Поскольку порядок не изменится, мы знаем, что представляют эти переменные.

В Python очень часто используют кортежи для представления данных, которые имеют более одного значения и которые необходимо сохранить вместе.

Например, вы можете использовать кортеж для хранения имени файла и его размера, или вы можете хранить имя и адрес электронной почты человека, или дату и время, а также общее состояние системы в любой момент времени.

Далее есть наборы, которые тоже похожи на списки. Здесь мы можем добавлять элементы, в отличие от кортежа, мы можем делать много чего с наборами, но особенность наборов заключается в том, что они сохраняют уникальность элементов в этой коллекции элементов.

```
# adding more items works as long as they are unique
unique.add("one")
unique.add("one")
unique.add("one")
unique.add("two")
print(unique)
```

✓ 0.2s

```
{'one', 'two'}
```

Что произойдет, если я буду добавлять один и тот же элемент в набор? Объект набора по-прежнему будет давать два элемента. Особенность наборов в том, что они помогают сохранить уникальность элементов в них.

```
# you can pop items too, like in lists, but it takes no arguments
unique.pop()
unique
```

✓ 0.2s

```
{'two'}
```

И вы можете извлекать элементы из набора с помощью метода pop, но в отличие от списков, метод pop здесь не принимает аргументов, а просто возвращает случайный элемент, удаляя его из набора.

Добавление и извлечение данных из структур данных

Давайте теперь посмотрим, как можно работать как со списками, так и со словарями Python для добавления и извлечения данных. Эти две очень распространенные структуры данных в Python ведут себя по-разному, и нам нужно понимать, как можно добавлять в них данные, извлекать данные и различные способы, которыми можно манипулировать данными из этих структур данных.

И сначала посмотрим, как можно добавлять данные в списки.

```
# define an empty list of fruits
fruits = []
fruits
```

✓ 0.2s

```
[]
```

```
# appending items will insert them by the end
fruits.append("orange")
fruits.append("apple")
fruits
```

✓ 0.2s

```
['orange', 'apple']
```

Здесь мы добавляем элементы в этот список фруктов, и мы видим, что при этом список сохраняет порядок.

```
# similar to append, you can insert, but that requires an index (position)
fruits.insert(0, "melon")
fruits
```

✓ 0.2s

```
['melon', 'orange', 'apple']
```

Также вы можете делать вставку в список, используя индекс или позицию элемента в списке. В этом случае мы используем индекс 0 и вставляем дыню в первую позицию списка.

Таким образом, если вы хотите изменить порядок в списке, вы можете использовать вставку, потому что она позволяет использовать индекс.

```
# you can add one list to another one
vegetables = ["cucumber", "carrots"]
fruits + vegetables
```

✓ 0.2s

```
['melon', 'orange', 'apple', 'cucumber', 'carrots']
```

```
# watch out when appending a list to an existing list
shopping_list = fruits + vegetables
shopping_list.append(["sugar", "salt"])
shopping_list
```

✓ 0.2s

```
['melon', 'orange', 'apple', 'cucumber', 'carrots', ['sugar', 'salt']]
```

Также вы можете добавить один список к другому, с помощью простого оператора +. И вы можете добавить список к существующему списку с помощью метода append. При этом у вас будет список внутри списка.

```
# you can extend a list instead (similar to adding)
shopping_list = fruits + vegetables
shopping_list.extend(["sugar", "salt"])
shopping_list
```

✓ 0.2s

```
['melon', 'orange', 'apple', 'cucumber', 'carrots', 'sugar', 'salt']
```

Но если это не то, что вы хотите, то вы можете расширить список новым списком. Расширение списка похоже на добавление к списку, но при этом у вас не будет вложенного списка.

```
colors = ["red", "yellow", "green", "blue"]

# by index
colors[0]
```

✓ 0.2s

```
'red'
```

```
# slicing for the first three items
colors[:2]
```

✓ 0.3s

```
['red', 'yellow']
```

```
# slicing for the first three items
colors[:3]
```

✓ 0.2s

```
['red', 'yellow', 'green']
```

Теперь давайте посмотрим, как можно извлечь данные из списка. Мы можем извлекать значения из списка с помощью индекса.

Здесь у нас есть список цветов: красный, желтый, зеленый и синий. И мы можем извлечь элемент с помощью индекса. В этом случае мы используем индекс 0 и получаем элемент «красный».

Но что, если мы захотим извлечь диапазон элементов? Мы не сможем сделать это с помощью только одних индексов. Вот тут-то и вступает в игру операция нарезки, которая используется с двоеточием.



Это очень изящный способ создания подмножества списка, если вы знаете диапазон, с которым хотите работать.

Вы также можете извлечь элемент из списка, но элемент с данным номером индекса должен существовать.

```
# popping can retrieve an item and removing it from the list
# but the index must exist!
colors.pop(100)
```

⊗ 0.2s

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/Users/alfredodeza/code/python-bootcamp/introduction-to-python/notebooks/wee
6' in <cell line: 3>()
      1 # popping can retrieve an item and removing it from the list
      2 # but the index must exist!
----> 3 colors.pop(100)

IndexError: pop index out of range
```

```
# use an existing index to pop correctly
popped_item = colors.pop(1)
popped_item
```

✓ 0.2s

```
'yellow'
```

Здесь я хочу извлечь элемент с индексом 100, который не существует. И я получу ошибку индекса за пределами диапазона.

И такое извлечение элемента изменяет списки.

```
# popping alters the list however!
colors
```

✓ 0.2s

```
['red', 'green', 'blue']
```

Здесь вы можете видеть, что желтый больше не является частью этого списка.

```
# if you know the item you can remove it
colors.remove("blue")
colors
```

✓ 0.2s

```
['red', 'green']
```

Еще одна вещь, которую вы можете сделать, это просто удалить элемент из списка. Операция удаления найдет, где находится элемент и удалит его из списка.

Теперь давайте рассмотрим несколько способов, с помощью которых можно извлекать данные из словарей.

```
contact_information = {}
```

✓ 0.2s

```
# normal retrieval (and possible exception)
contact_information["height"]
```

⊗ 0.2s

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/Users/alfredodeza/code/python-bootcamp/introduction-to-python/notebooks/week3
Cell 3' in <cell line: 2>()
      1 # normal retrieval (and possible exception)
----> 2 contact_information["height"]

KeyError: 'height'
```

Здесь мы создаем экземпляр словаря contact_information, и мы создаем экземпляр пустого словаря. Что произойдет, если я извлеку из словаря ключ, которого не существует? – Я получу ошибку ключа.



И есть несколько способов это обойти, которые я вам покажу. Первый способ — использование блока try-except. Мы пока не рассматривали работу с исключениями, и мы рассмотрим это позже.

Здесь я использую try с извлечением из словаря ключа, которого не существует. И если я получу ошибку ключа, в блоке except я выведу на печать сообщение, потому что ошибка ключа перехватывается, и я делаю что-то другое вместо этого исключения. Это один из способов, которым вы можете справиться с этой неприятной ошибкой ключа.

Другой способ, который мне нравится, это использование метода get. Метод get позволяет передать ключ, значение которого вы хотите извлечь, и он выдает это значение. И если ключа не существует, метод выдаст специальное значение None для этого ключа. И у меня больше не будет ошибки ключа. Это полезный способ, потому что по умолчанию, если ключ не найден, метод get выдаст None.

```
# falling back when there is no key
result = contact_information.get("height", "5ft 9in.")
print("Height of contact is", result)
```

✓ 0.2s

```
Height of contact is 5ft 9in.
```

Но что произойдет, если вам не нужно None, а вы хотите получить какое-то другое значение? Метод get позволяет указать второй аргумент, значение которого будет выведено, если ключ не существует.

```
contact_information["age"] = 31
print("Age is", contact_information.pop("age"))
print(contact_information)
```

✓ 0.2s

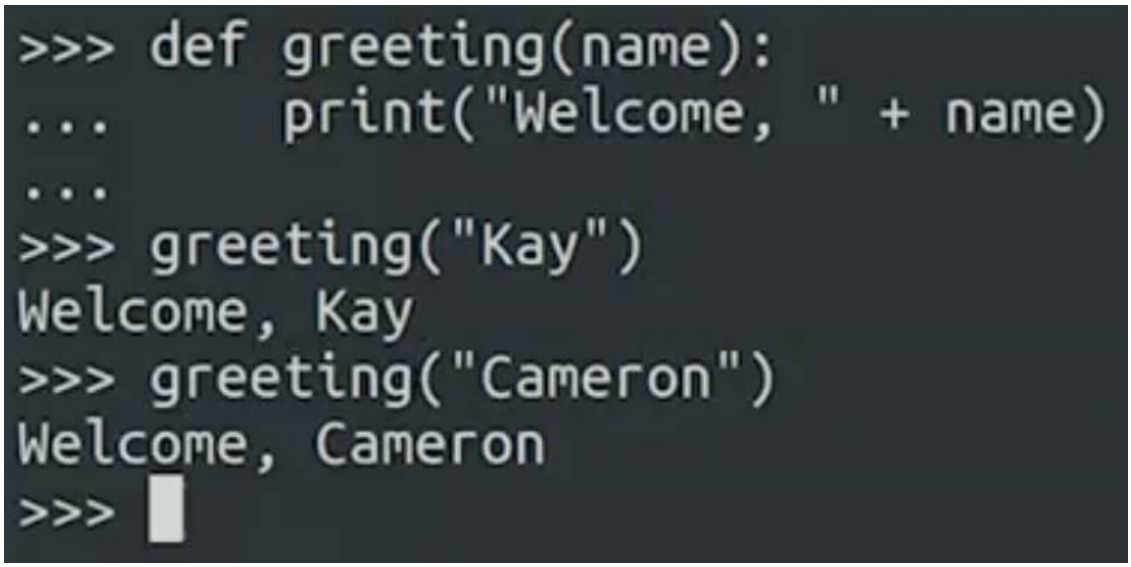
```
Age is 31
{}
```

Вы также можете использовать метод pop в словарях, который извлекает и удаляет ключ из словаря. Поэтому, когда я здесь извлеку ключ, словарь должен быть пустым.

Функции

До сих пор мы рассматривали переменные, выражения и операции, которые являются самыми маленькими компонентами скриптов. Далее мы рассмотрим функции, которые являются ещё одним важным строительным блоком программирования.

В наших примерах мы уже встречали функции Python. Функция print, которая выводит текст на экран, функция type, которая сообщает нам тип определённого значения, функция str, которая преобразует число в строку. Все эти функции являются частью языка, и существует множество других встроенных функций Python. Но мы посмотрим, как определять собственные функции, чтобы указать компьютеру делать то, чего не делают встроенные функции языка.



Давайте начнём с простого примера. В этом фрагменте кода мы определяем функцию. Наша функция принимает параметр. Здесь этот параметр — имя, и функция выводит приветствие для этого имени. Этот фрагмент кода невелик, но он уже демонстрирует много важных моментов о том, как мы определяем функции в Python.

Давайте разберем это шаг за шагом. Чтобы определить функцию, мы используем ключевое слово «def». Имя функции — это то, что идет после ключевого слова. В этом примере имя функции — «greeting». Поэтому, чтобы вызвать функцию позже в скрипте, мы будем использовать слово «greeting». После имени у нас есть параметры функции, которые записываются в скобках. В этом примере у нас только один параметр, «name», за которым следует двоеточие, в конце строки. После двоеточия у нас есть тело функции. Именно здесь мы указываем, что мы хотим, чтобы наша функция делала. Обратите внимание, как тело смещено вправо. Это ключевая характеристика Python, и мы будем часто с ней сталкиваться. Пока просто помните, что тело функции должно находиться справа от определения.

В этом примере тело содержит всего одну строку, которая вызывает функцию «print». Выглядит просто, правда? Но создание функций может быть на самом деле очень мощным инструментом. Тело функции может содержать столько строк, сколько нам нужно, и выполнять множество интересных действий.

```
>>> def greeting(name, department):
...     print("Welcome, " + name)
...     print("You are part of " + department)
...
>>> greeting("Blake", "IT Support")
Welcome, Blake
You are part of IT Support
>>> greeting("Ellis", "Software engineering")
Welcome, Ellis
You are part of Software engineering
>>>
```

Теперь наша функция принимает два параметра вместо одного: имя и отдел, и выводит два отдельных сообщения. Снова обратите внимание на отступы. Мы можем добавить в тело функции столько строк, сколько захотим, но каждая строка должна быть с отступом в одинаковую величину справа. В этом примере мы используем четыре пробела. Мы могли бы использовать два, восемь или любое другое число, главное, чтобы они были одинаковыми.

Мы видели, как можно передавать значения в функцию в качестве параметров, передавая такие значения, как имя или отдел в примере ранее. Но как насчет получения значений из функции? Вот здесь и вступает в игру концепция возвращаемых значений. Работа, которую выполняют функции, может давать новые результаты.

Конечно, мы можем вывести результаты на экран, но что, если мы хотим использовать эти результаты позже в нашем скрипте или не хотим выводить их вообще? Мы можем сделать это, возвращая значения из функций, которые мы определяем сами.

```
>>> def area_triangle(base, height):
...     return base*height/2
...
>>> area_a = area_triangle(5,4)
>>> area_b = area_triangle(7,3)
>>> sum = area_a + area_b
>>> print("The sum of both areas is: " + str(sum))
The sum of both areas is: 20.5
>>>
```

Давайте вернемся к вычислению площади треугольника. Площадь треугольника вычисляется как основание, умноженное на высоту, деленное на 2. Представьте, что нам нужно вычислить это значение несколько раз в нашем коде. Было бы полезно иметь функцию, которая делает это за нас.

Посмотрите, как это будет выглядеть. Мы используем ключевое слово «return», чтобы сообщить Python, что это возвращаемое значение функции. Когда мы вызываем функцию, мы сохраняем это значение в переменной.

Допустим, у нас есть два треугольника, и мы хотим сложить сумму площадей обоих треугольников. Сначала мы вычисляем площади двух треугольников по отдельности. Затем мы складываем суммы обеих площадей вместе. Наконец, мы выводим результат, преобразуя его в строку.

Как вы можете видеть в этом примере, функция area triangle возвращает значение, которое, как и следовало ожидать, является площадью треугольника. Мы сохраняем это значение в отдельной переменной для каждого вызова функции. В данном случае это area_a и area_b. Затем мы работаем с этими значениями, добавляя их в переменную с именем sum и выводя только этот конечный результат. Это демонстрирует мощь оператора return. Он позволяет нам комбинировать вызовы функций и более сложных операций, что делает ваш код более многоразовым.

```
>>> def convert_seconds(seconds):
...     hours = seconds // 3600
...     minutes = (seconds - hours * 3600) // 60
...     remaining_seconds = seconds - hours * 3600 - minutes * 60
...     return hours, minutes, remaining_seconds
```

Операторы return в Python еще интереснее, потому что мы можем использовать их для возврата более чем одного значения. Допустим, у вас есть продолжительность времени в секундах, и вы хотите преобразовать её в эквивалентное количество часов, минут и секунд.

Вот как это сделать в Python. Вы заметили новый оператор в этой функции? Этот оператор с двойной косой чертой называется целочисленным делением. Целочисленное деление делит число и берёт целую часть результата деления. Например, 5 // 2 — это два вместо 2,5. В нашем примере первая операция вычисляет, сколько часов в заданном количестве секунд, а вторая вычисляет, сколько минут осталось после вычитания часов, а затем сколько секунд осталось после вычитания минут. В результате мы получаем три числа, поэтому функция возвращает все три значения.

```
>>> hours, minutes, seconds = convert_seconds(5000)
>>> print(hours, minutes, seconds)
1 23 20
>>>
```

Поскольку мы знаем, что функция возвращает три значения, мы присваиваем результат функции трём разным переменным.

Есть ещё один момент, который следует отметить о возвращаемых значениях. Можно ничего не возвращать, и это совершенно нормально.

```
>>> def greeting(name):
...     print("Welcome, " + name)
...
>>> result = greeting("Christine")
Welcome, Christine
>>> print(result)
None
>>>
```

Здесь функция просто вывела сообщение и ничего не вернула. Как вы думаете, что произойдет, если мы попробуем присвоить значение этой функции переменной?

Здесь, когда мы вызвали функцию, она вывела сообщение, как мы и ожидали. Мы сохранили возвращаемое значение в переменной result, но в функции не было оператора return.

Поэтому значение result равно None. None — это очень специфический тип данных в Python, используемый для обозначения того, что переменные пусты или ничего не возвращают.

Пример функции:

```
1  def double(x):
2      """Doubles a number"""
3      return x * 2
4
5  print(double(5)) # Prints 10
```

```
10
```

Функция с аргументами:

```
def full_name(first, last):
    return first + " " + last

print(full_name("John", "Doe")) # Prints John Doe
```

```
John Doe
```

Переменные аргументы:

```
def sum_all(*numbers):
    sum = 0
    for n in numbers:
        sum += n
    return sum

print(sum_all(1, 2, 3)) # Prints 6
```

```
6
```

Аргументы ключи:

```
def greet(name, greeting="Hello"):
    print(greeting + ", " + name)

greet("John") # Prints Hello, John
greet("Mary", greeting="Hi") # Prints Hi, Mary
```

```
Hello, John
Hi, Mary
```

Генератор:

```
1    #counter generator
2    def counter(start=0):
3        n = start
4        while True:
5            yield n
6            n += 1
7
8    for i in counter(5):
9        if i > 10:
10           break
11       print(i)
```

```
5
6
7
8
9
10
```

```
#Infinite Fibonacci sequence generator
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

for n in fib():
    if n > 10:
        break
    print(n)
```

```
0
1
1
2
3
5
8
```

Выражение генератора:

```
#Generator expression to get squares
nums = (x**2 for x in range(10))
print(list(nums))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Бесконечная последовательность:

```python
# Infinite random attack sequence
import random

attacks = ["kimura", "armbar", "triangle"]

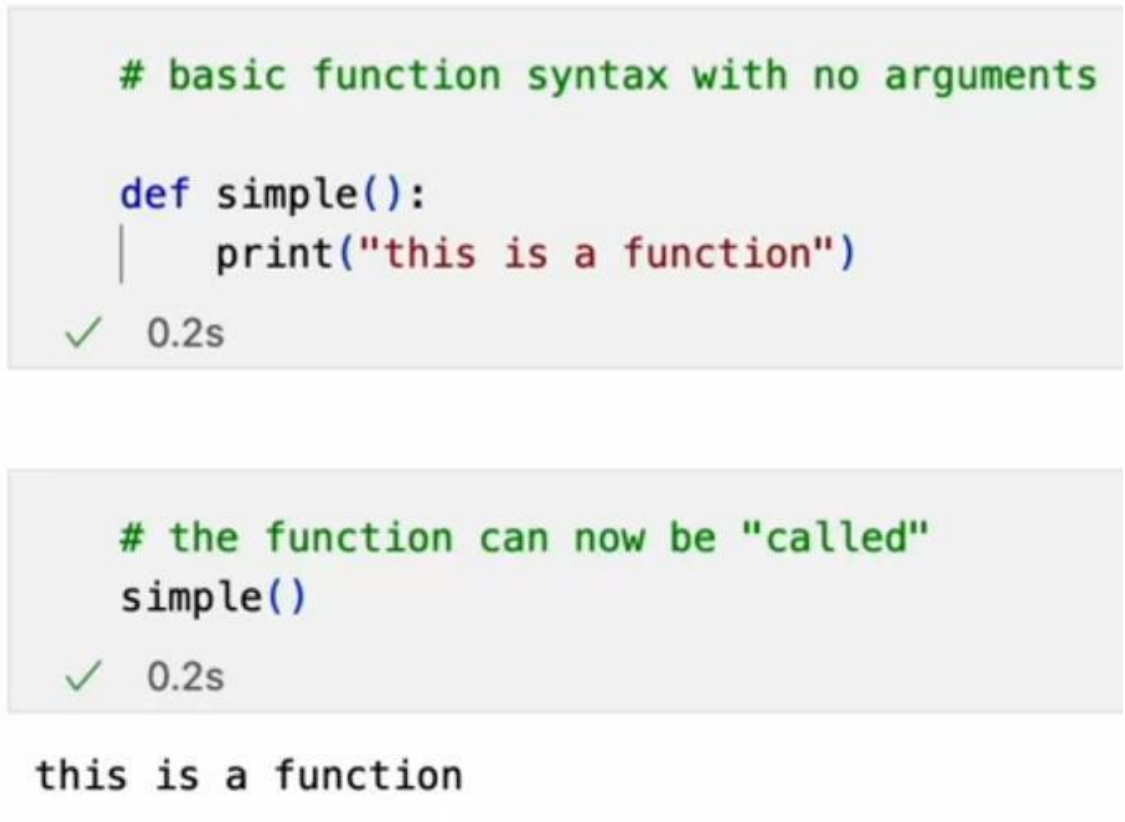
def lazy_random_attacks():
    """Lazily yield random attacks forever"""

    while True:
        attack = random.choice(attacks)
        print("Yielding attack")
        yield attack

generator = lazy_random_attacks()

for _ in range(5):
    print(next(generator))
```

Функции — это первый способ, с помощью которого вы можете написать блок повторно используемого кода.



Теперь, чтобы написать функцию, вам нужно будет использовать ключевое слово def. Затем вы продолжаете с имени, без пробелов, и далее следуют скобки. И, наконец, двоеточие. А затем следующая строка, которая должна быть с отступом, так что в Python это будет означать, - эта следующая строка является частью этой функции. В каждой отдельной строке, которая входит как часть функции, должен быть отступ не менее четырех пробелов.

Если я запущу этот код, Python создаст функцию с именем simple, и всякий раз, когда мы будем вызывать функцию по имени, она будет выводить эти выходные данные.

```
# watch out for functions that don't return values

result = simple()
print("The value of result is:", result)
```

✓ 0.2s

```
this is a function
The value of result is: None
```

```
# if you need a value from a function, the function must return

def produce():
    return "This is a return value!"

# calling it allows capturing the resulting product
result = produce()
print("The result value is:", result)
```

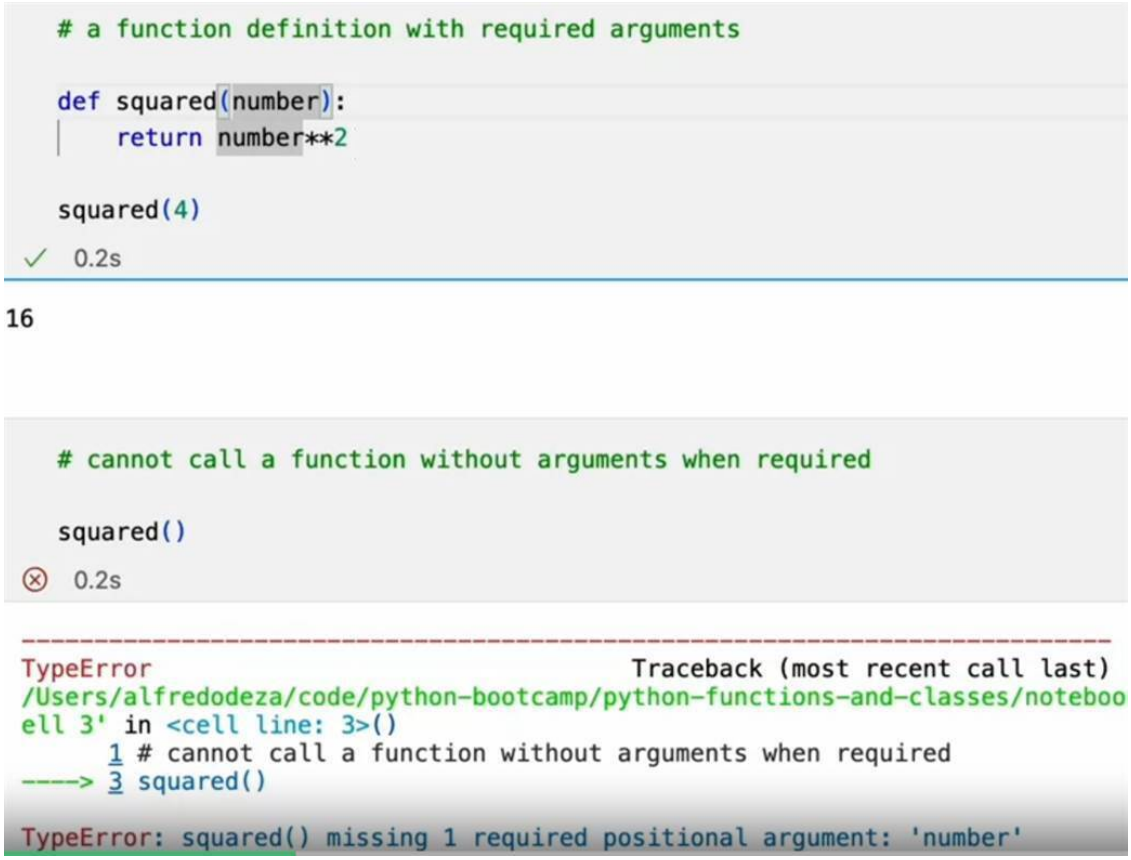
✓ 0.2s

```
The result value is: This is a return value!
```

Не все функции, которые что-то делают, будут возвращать значение. Что произойдет, если я захочу присвоить такую функцию переменной result?

Вы увидите, что это значение result равно None, и это потому, что в Python вы всегда получите неявное возвращаемое значение None. Если вы явно возвращаете значение в функции, это хорошо, но, если вы этого не сделаете, Python по умолчанию присвоит возвращаемое значение None.

Но если вы хотите вернуть значение в функции, вы делаете это с помощью ключевого слова return.



Также функции могут принимать аргументы, и в этом случае мы посмотрим, как работают эти аргументы. Определение функции с обязательными аргументами будет иметь перечень аргументов в скобках.

Здесь, как вы видите, у нас есть функция и эта функция возвращает квадрат произведения любого числа, которое мы передаем. Так что, если я запущу эту функцию и передам ей четыре в качестве аргумента, то я получу 16.

Но что произойдет, вы не передадите значение аргумента в функцию? - Вы получите ошибку типа.

```
# there are functions with optional arguments
# this one will give you a 0, defaulting to no input
int()
```

✓ 0.2s

```
0
```

```
# but you can use an argument too, making it flexible
int("10")
```

✓ 0.2s

```
10
```

Также есть функции с необязательными аргументами. Здесь показана встроенная функция int, - вам не нужно ее импортировать. Я не создавал эту функцию, она поставляется с Python.

И без аргумента эта функция даст вам ноль. Но с аргументом вы можете преобразовать строку в целое значение.

```
# an optional argument uses a "keyword argument"

def greetings(full_name="John Doe"):
    print("Greetings ", full_name)
```

✓ 0.2s

```
# you can call the function with or without the argument because it has a default
greetings()
greetings("Superman!")
```

✓ 0.2s

```
Greetings  John Doe
Greetings  Superman!
```

Далее давайте посмотрим, что такое аргумент ключ. И аргумент ключ — это возможность сделать аргумент необязательным и иметь значение по умолчанию.

И если вы определите функцию с аргументом ключом, и вызовете такую функцию без аргумента, тогда аргументу присвоится значение ключа по умолчанию.

```
# arguments always go before keyword arguments

def formal_greeting(name, title="Dr."):
    print("Greetings", title, name)

formal_greeting("Jenkins", title="Sr.")
```

✓ 0.2s

```
Greetings Sr. Jenkins
```

И, наконец, одна вещь, с которой вы должны быть осторожны, - если вы смешиваете аргументы, аргумент и аргумент ключ, вы должны соблюдать определенный порядок.

Вы должны сначала передать значение аргумента в функцию, а затем значение аргумент ключ, иначе Python выдаст вам исключение.

Еще одна вещь, которую следует иметь в виду при работе с функциями и аргументами функций, это то, что вы можете столкнуться с переменными аргументами.

```
# this function can take 0 or more arguments

def family_members(*args):
    for name in args:
        print(name)

family_members("Lucy", "Matt", "Bob")
```

✓ 0.2s

```
Lucy
Matt
Bob
```

Переменные аргументы означают, что функция может определять ноль или более аргументов, и ей все равно, передаете ли вы одно значение аргумента, передаете ли вы ноль или передаете 100 значений аргументов.

```
# it can also be called with no arguments at all making it flexible
family_members()
```
✓ 0.2s

```
# there is no need to call it *args or *a (although common)

def family_members(*names):
    for name in names:
        print(name)

family_members("John", "Sam")
```
✓ 0.2s

```
John
Sam
```

Почему это может быть полезно, - потому что иногда вы не знаете сколько будет аргументов у функции при использовании.

```
# define a function that takes 0 or more keyword arguments
def stats(**kwargs):
    # kwargs is now a dictionary
    for key, value in kwargs.items():
        print(key, "-->", value)

stats(speed="slow", active=False, weight=210)
```
✓ 0.5s

```
speed --> slow
active --> False
weight --> 210
```

```
# because it's variable, it means that we can pass 0 items
result = stats()
print("Result is", result)
```
✓ 0.2s

```
Result is None
```

Также есть переменные аргументы ключи. Вы можете указать их с помощью двойной звезды. Внутри это словарь. Вы можете работать с этой функцией, как если бы это был словарь. При вызове такой функции вы можете передать в качестве аргументов сколько угодно пар ключ-значение.

Как мы уже отмечали ранее, функции мощны, потому что вы можете создавать свои собственные. Вы можете использовать их для организации кода в ваших скриптах в логические блоки, что делает написанный вами код более удобным для использования и повторного использования.

```
name = "Kay"
number = len(name) * 9

print("Hello " + name + ". Your lucky number is " + str(number))

name = "Cameron"
number = len(name) * 9

print("Hello " + name + ". Your lucky number is " + str(number))
```

Run

Output:

```
Hello Kay. Your lucky number is 27
Hello Cameron. Your lucky number is 63
```

Посмотрите этот пример. Этот скрипт использует функцию «len», которая возвращает длину строки. В этом примере скрипт затем использует эту длину для вычисления числа. И наконец, он выводит сообщение, с именем и числом. Каждый раз, когда мы хотим выполнить вычисление, мы изменяем значения переменных и пишем формулу. Затем выводим приветствие, за которым следует число.

Видите, что в первой и второй части кода есть ровно две одинаковые строки? Когда вы обнаруживаете дублирование кода в ваших скриптах, полезно проверить, можно ли немного очистить код, используя функцию. Как насчет того, чтобы переписать этот код, создав функцию для группировки всего дублированного кода в один код?

```
def lucky_number(name):
    number = len(name) * 9
    print("Hello " + name + ". Your lucky number is " + str(number))

lucky_number("Kay")
lucky_number("Cameron")
```

Run

Output:

```
Hello Kay. Your lucky number is 27
Hello Cameron. Your lucky number is 63
```

Обновленный скрипт дает нам точно такой же результат, как и исходный, но выглядит намного чище. Во-первых, мы определили функцию, названную lucky number, которая выполняет наши вычисления и выводит их на экран. Затем мы вызываем эту функцию дважды, по одному разу с разным именем.

Поскольку мы сгруппировали вычисления и вывод в функцию, наш код не только легче читается, но и теперь может быть использован повторно. Мы можем выполнять код внутри функции lucky number столько раз, сколько нам нужно, просто вызывая ее с другим именем, чтобы нам не приходилось писать код снова и снова для каждого нового значения.

Вы заметили, что мы передаем информацию в функции через их параметры? Это один из многих способов, которые мы можем использовать для ввода данных в код. Значения этих параметров могут поступать из разных мест, например, из файла на нашем компьютере или через форму на веб-сайте, но это не влияет на код. Результат функции остается тем же, независимо от того, откуда поступают параметры.

Хороший или плохой стиль при написании кода не сильно влияет на успех или провал скрипта, но он может иметь большое значение для тех, кто его использует и вносит в него свой вклад. Плохой стиль программирования может осложнить жизнь ИТ-специалистам или системным администраторам, которым приходится читать скрипт после его написания или вносить в него изменения, чтобы он работал с новой системой. Плохой стиль может даже вызвать головную боль у автора скрипта, если прошло много времени с момента его написания.

Представьте, что вам приходится переписывать свой собственный код, потому что он слишком запутанный, чтобы его понять. С другой стороны, хороший стиль может сделать скрипт почти похожим на естественный человеческий язык. Хороший стиль позволяет сразу же сделать замысел и структуру скрипта ясными для пользователя. Хороший стиль облегчает жизнь людям, которым приходится поддерживать код, и помогает им понять, что он делает и как он это делает. Он также может уменьшить количество ошибок, поскольку упрощает обновление кода.

Но что делает стиль фрагмента кода хорошим или плохим? Хотя нет жестких правил, которые применимы к каждому языку программирования и ситуации, знание нескольких принципов значительно поможет создать хороший, хорошо стилизованный код.

Во-первых, вы хотите, чтобы ваш код был максимально самодокументирующимся. Самодокументирующийся код написан таким образом, чтобы он был читаемым и не скрывал своего предназначения. Этот принцип можно применять ко всем аспектам написания кода, от выбора имен переменных до написания четких, лаконичных выражений.

```
>>> def calculate(d):
...     q = 3.14
...     z = q * (d ** 2)
...     print(z)
...
>>> calculate(5)
78.5
>>>
```

Возьмем, к примеру, этот фрагмент кода. Трудно определить назначение этого кода, просто взглянув на него. Названия переменных не дают читателю много информации, и хотя вы, вероятно, сможете вычислить результат, нет никаких подсказок о том, для чего этот результат может быть использован. На языке программирования, когда мы переписываем код, чтобы сделать его более самодокументируемым, мы называем этот процесс рефакторингом. Итак, как бы выглядел рефакторинг этого кода?

```
>>> def circle_area(radius):
...     pi = 3.14
...     area = pi * (radius ** 2)
...     print(area)
...
>>> circle_area(5)
78.5
>>>
```

В этом переписанном коде названия переменных и функции отражают их назначение, что помогает пользователю быстрее понять код.

Когда хорошее именование не может сделать код понятным, вы можете добавить немного пояснительного текста к коду. Вы делаете это, добавляя то, что мы называем комментарием в Python.

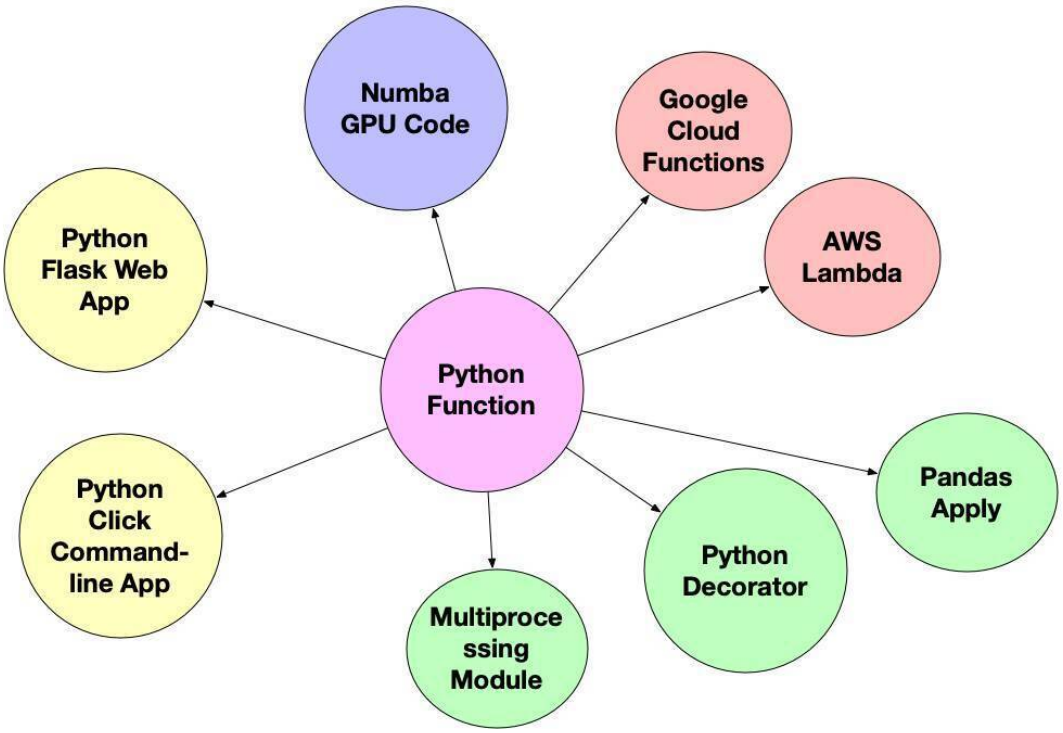
```
# This is how you write a comment in Python!
```

Комментарии обозначаются символом решетки (#). Когда ваш компьютер видит символ решетки, он понимает, что должен игнорировать все, что идет после этого символа в этой строке.

Использование комментариев позволяет объяснить, почему функция делает что-то определенным образом. Это также позволяет оставлять заметки для себя в будущем или для других программистов, чтобы напомнить себе, что нужно улучшить и почему.

Создание функций

Что можно сделать с помощью функций Python? - Всего несколько строк кода — и множество разнообразных и сложных проблем имеют свои решения.



Давайте рассмотрим простую функцию, которая возвращает «Марко», если вы передадите «Поло».

```
def marco(name):
if name == "Marco":
return "Polo"
marco("Marco")
'Polo'
```

Эта функция принимает аргумент имени. Если имя совпадает с Марко, тогда функция вернет Поло.

Обратите внимание, что если функции будет передано другое имя, в данном случае Боб, тогда функция ничего не вернет. Причина такого поведения в том, что по умолчанию функция Python возвращает None.

```
marco("Bob")
```

Как бы проста эта функция ни была, она может работать в различных средах. Ее можно «применить» к Pandas DataFrame. Она может работать в облаке через AWS Lambda. Возможно, она могла бы стать веб-сервисом?

Если вы понимаете, как работает функция, вы можете запрограммировать практически что угодно с помощью языка Python! Для невероятных вещей требуется всего несколько строк кода. Более того, это отличный дизайн для написания небольших фрагментов логики, даже при построении более сложной системы.

Задача: написать простую функцию, которая принимает переменную и возвращает что-то еще, если фраза совпадает. Например, что-то вроде этого:

```
if dollars == 1000:
return "rich"
```

Получилось? Если нет, давайте обсудим некоторые критические компоненты того, что делает функцию работающей. У функции должно быть несколько вещей, чтобы она была рабочей. Функция Python использует слово def, за которым следует имя. Этот синтаксис — это имя функции. Функция также имеет скобки (), куда поступают входные данные, а затем следует двоеточие «:». И внутри функция должна что-то делать.

В приведенном примере функция проверяет, равно ли имя Marco. И наконец, функция возвращает некоторый результат. Функция не обязана принимать входные данные и не обязана возвращать что-либо явно, но большинство полезных функций допускают ввод данных и возвращают некоторое значение.

```
def somename(somevar):
#some logic
return someresult
```

А как насчет функции, которая делает еще меньше? Пример ниже — функция, которая не принимает никаких входных данных и ничего не возвращает. Она просто выводит «Марко Поло».

```
def simple_marco():
print("Marco Polo")
simple_marco()
Marco Polo
```

Один из способов проверить, что функция возвращает — это присвоить вывод функции переменной. Затем вы можете вывести эту переменную. В следующем примере переменная return_value хранит вывод функции, который равен None, значению по умолчанию функции, которой нечего возвращать.

```
return_value = simple_marco()
print(return_value)
None
```

Также существует много случаев, когда в функции выполняется некоторая логика, но функции не нужно принимать данные. Ниже показан пример, где две строки, Marco и Polo, складываются вместе, а затем возвращаются функцией.

```
def medium_marco():
statement = "Marco " + "Polo"
return statement
return_medium_marco = medium_marco()
print(return_medium_marco)
Marco Polo
```

Переменная return_medium_marco показывает, что возвращает функция. Когда переменная return_medium_marco печатается, она возвращает Marco Polo.

Задача: написать функцию без ввода или возврата.

Давайте рассмотрим другой сценарий. Ниже показан пример — функция, которая принимает данные, но ничего не возвращает.

```
def input_marco(name):
if name == "Marco":
print("Polo")
else:
print("Nice try. No Cigar!")
```

Если передается Marco, выводится Polo.
input_marco("Marco")
Polo
Если передается другое имя, выполняется другой оператор печати.
input_marco("Bob")
Nice try Bob. No Cigar!
Обратите внимание, что при печати сама функция все равно возвращает None.
result = input_marco("Jenny")
print(result)
Nice try Jenny. No Cigar!
None
Задача: написать функцию с входными данными и без возврата.

Создание классов и методов

Класс - проект для создания объектов. Определяет атрибуты и методы.

Экземпляр - объект, созданный из класса. Имеет доступ к атрибутам и методам класса.

Метод- функция, принадлежащая классу. Доступ через имя экземпляра или класса.

Конструктор - специальный метод, который запускается при создании экземпляра. Используется для инициализации атрибутов.

Наследование - создание дочернего класса из родительского класса. Дочерний класс наследует атрибуты и методы.

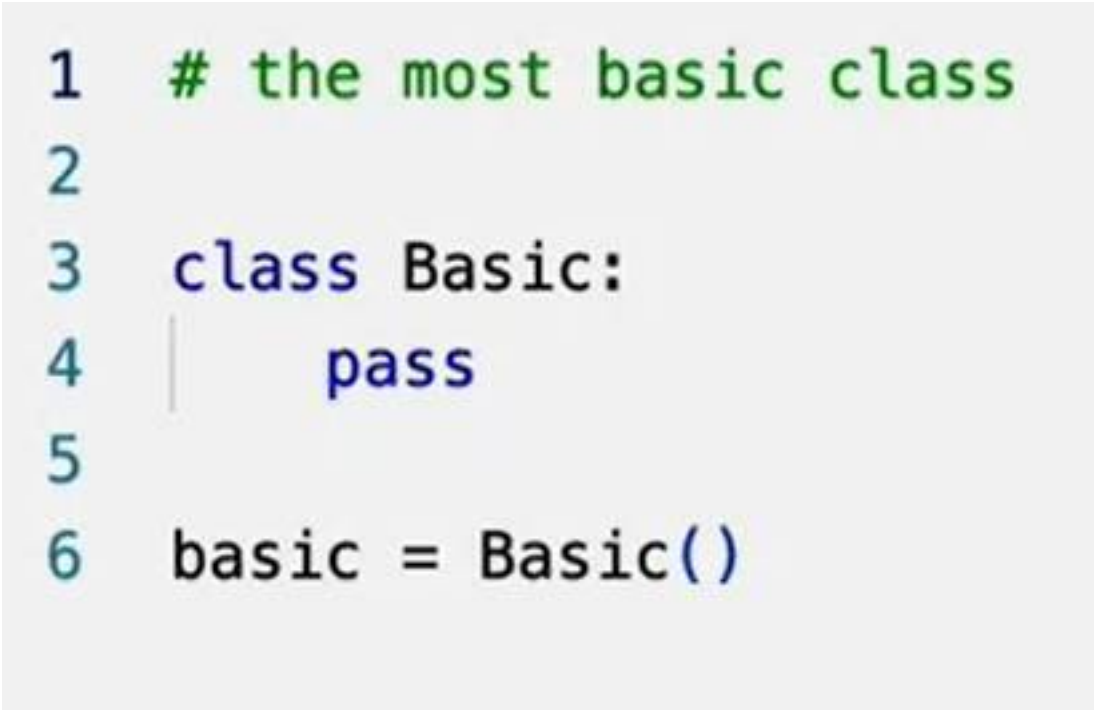
Пример класса:

```
class Vehicle:
wheels = 4 # Class attribute
def __init__(self, make, model):
self.make = make # Instance attribute
self.model = model
def description(self): # Method
return f"The {self.make} {self.model}"
car = Vehicle("BMW", "i3") # Instance created
print(car.wheels) # Access class attribute
print(car.description()) # Call instance method 4
The BMW i3
```

Пример наследования:

```
class Pet:
def eat(self):
print("Chomp")
class Dog(Pet):
def bark(self):
print("Bark!")
dog = Dog()
dog.eat() # Inherited method
Chomp
Bark!
```

Помимо функций, классы — это то, с чем вы будете часто сталкиваться в Python. В основном, конечно, используются функции, но классы также есть, и у них есть свое собственное место и своя собственная причина для использования и реализации, чтобы вы могли создавать мощные программы.



```
1  # the most basic class
2
3  class Basic:
4      pass
5
6  basic = Basic()
```

Итак, для создания класса вы будете использовать ключевое слово class в Python. И затем у вас будет имя, похожее на имя в функциях, только там у вас есть def, затем имя, а затем двоеточие, — вот как вы создаете или определяете новый класс.

Я использую pass здесь, потому что этот класс пустой, он бессмысленный. Здесь строка 4 ничего не делает. А затем в строке 6 я создаю экземпляр класса. Как только я вызываю этот класс, он становится экземпляром класса. Он создается, он конструируется.

Теперь давайте посмотрим, что мы можем найти, когда попробуем покопаться в атрибутах, которые у нас здесь есть.

```
1  # use dir() to find about what is available in a class
2  dir(basic)
```

✓ 0.2s

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
```

Здесь, безусловно, много разных функций, и все они начинаются с двойного подчеркивания. И мы видим по крайней мере одну из них, которая является _init_ — это специальная функция, которая вызывается при создании нового объекта класса или конкретного экземпляра класса.

Теперь, все это - специальные методы, которые принадлежат классу, и вы, вероятно, не будете использовать большинство из них в своей повседневной работе, но полезно знать, что они там есть.

```
1  # in the past, Python classes could have this odd "object" in them
2
3  class Basic(object):
4      pass
5
6  # this is not common or needed in Python anymore
```

Теперь давайте создадим другой тип конструкции класса, и это может показаться немного странным. В строке 3 у нас снова есть класс, но здесь мы используем объект.

Что это за объект? - Старые версии Python передавали этот объект классу при его определении. Но это больше не используется в современном Python, так что вы этого не увидите в

современном коде. Некоторые люди склонны использовать это до сих пор, но в современном Python это не так.

```
1  # classes in general are good when grouping code and behavior that can be reused
2  # That type of grouping and behavior is not possible with functions for example
3
4  class Dog:
5      is_animal = True
6
7      def bark():
8          print("woof!")
9
10 dog = Dog()
```

0.2s

```
1  # with the Dog() class instantiated and ready to be used you can interact with its parts
2  dog.bark()
3  dog.is_animal
```

Здесь мы создаем класс с именем Dog и используем то, что называется атрибутами класса.

В строке 5 мы определяем атрибут is_animal со значением True. Это называется атрибутом класса. А затем мы определяем метод. Здесь есть ошибка в этом методе, и этот метод пропускает что-то важное, но давайте продолжим. Если вы не знакомы с классами, вы не сможете увидеть, что не так с этим классом. Но если вы хорошо знакомы с классами, вы увидите, что здесь, в строке 7, есть проблема.

Создадим экземпляр класса, объект собаки, в строке 10. А теперь давайте заставим его лаять.

```
1  # with the Dog() class instantiated and ready to be used yo
2  dog.bark()
3  dog.is_animal
```

⊗ 0.2s

```
---------------------------------------------------------------------
TypeError                                 Traceback (most recent
/Users/alfredodeza/code/python-bootcamp/python-functions-and-clas
ynb Cell 6' in <cell line: 2>()
      1 # with the Dog() class instantiated and ready to be used
----> 2 dog.bark()
      3 dog.is_animal

TypeError: bark() takes 0 positional arguments but 1 was given
```

И когда мы запустим этот код, мы получим ошибку TypeError.

Вы можете видеть, что метод bark принимает 0 аргументов, но в ошибке пишут, что был задан один аргумент. Неявно Python за кулисами передает аргумент, и этот аргумент по соглашению называется self. Мы рассмотрим методы позже. Просто помните, что каждый раз, когда вы создаете метод, у вас должен быть аргумент, и первым аргументом должен быть self.

```
1  # classes in general are good when grouping code and behavior that can be reused
2  # That type of grouping and behavior is not possible with functions for example
3
4  class Dog:
5      is_animal = True
6
7      def bark(self):
8          print("woof!")
9
10 dog = Dog()
```

✓ 0.2s

Если я снова запущу этот код, метод bark выведет woof.

Вы можете создать столько экземпляров класса, сколько вам нужно. Это одна из мощных вещей классов, - они позволяют создавать экземпляры классов там, где вам нужно.

Но следите за атрибутами класса.

```
1  # watch out for class attributes that can change "state" from every instance
2  Dog.is_animal = False
3  print("Is rufus an animal?", rufus.is_animal)
4  print("Is dog an animal?", dog.is_animal)
```

✓ 0.2s

```
Is rufus an animal? False
Is dog an animal? False
```

```
1  # that change will also affect any future object
2  sparky = Dog()
3  sparky.is_animal
```

✓ 0.2s

```
False
```

Dog.is_animal является атрибутом класса. Это значит, что если вы не создаете экземпляр класса и вносите изменения, как мы делаем это в строке 2, то мы испортим каждый экземпляр класса, который будет затем создан.

Теперь, давайте посмотрим, что такое конструктор класса.

```
1  # the constructor in Python is with the special method called __init__
2
3  class Dog:
4
5      def __init__(self):
6          self.is_animal = True
7
8  dog = Dog()
9  dog.is_animal
```

✓ 0.2s

```
True
```

Конструктор — это возможность устанавливать атрибуты, когда создается класс. В этом случае мы будем использовать метод init. Здесь мы берем аргументы и определяем некоторое поведение.

В строке 6 мы используем self и определяем, что атрибут self.is_animal равен true. Это немного похоже на то, что у нас было раньше с атрибутом класса, за исключением того, что теперь мы используем self.

Почему self? Потому что переменная self указывает на объект класса.

Разница в том, что теперь я не могу изменить этот атрибут is_animal и повлиять на каждый отдельный экземпляр класса, как раньше с атрибутом класса. Это более безопасно.

```
1  rufus.is_animal = False
2  print("Rufus is an animal?", rufus.is_animal)
3  print("Sparky is an animal?", sparky.is_animal)
```

✓ 0.2s

```
Rufus is an animal? False
Sparky is an animal? True
```

Но давайте не будем отвлекаться на это. Просто помните, что у вас есть self, и вы можете с помощью self установить значение атрибута класса.

```
1  # pass arguments and keyword arguments to a class
2
3  class Animal:
4
5      def __init__(self, name, legs=4, barks=True):
6          self.name = name
7          self.legs = legs
8          self.barks = barks
9
10     # there is a bug in here!
11     def info(self):
12         print(f"This is an animal named {name}, has {legs} legs")
13         if barks:
14             print("And this one barks!")
15         else:
16             print("It doesn't bark at all")
```

У нас есть состояние класса, и мы можем изменить это состояние, разрешив передачу аргументов в конструктор.

В этом классе Animal конструктор использует self, и мы передаем в конструктор обязательный аргумент name, и два аргумента-ключа, которые являются необязательными, legs и barks. И с помощью self в конструкторе класса устанавливаются значения этих переменных.

В методе класса info мы используем эти переменные, но здесь есть ошибка. Посмотрите на комментарий в строке номер 10.

```
1  bunny = Animal("buster", barks=False)
2  bunny.info()
3  # if the bug is not fixed in the class, this will cause a NameError
⊗ 0.2s
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/Users/alfredodeza/code/python-bootcamp/python-functions-and-classes/notebooks/
Cell 8' in <cell line: 2>()
      1 bunny = Animal("buster", barks=False)
----> 2 bunny.info()

/Users/alfredodeza/code/python-bootcamp/python-functions-and-classes/notebooks/
Cell 7' in Animal.info(self)
     11 def info(self):
---> 12     print(f"This is an animal named {name}, has {legs} legs")
     13     if barks:
     14         print("And this one barks!")

NameError: name 'name' is not defined
```

Видите, в чем заключается ошибка здесь в строке 12?

Переменная name не определена. Откуда взялась эта ошибка?

Потому что эта переменная name не является обычной переменной. Это атрибут self. У нас нет прямого доступа к этой переменной, которая была передана конструктору. У нас есть доступ к этой переменной, которая находится внутри класса, с помощью self.

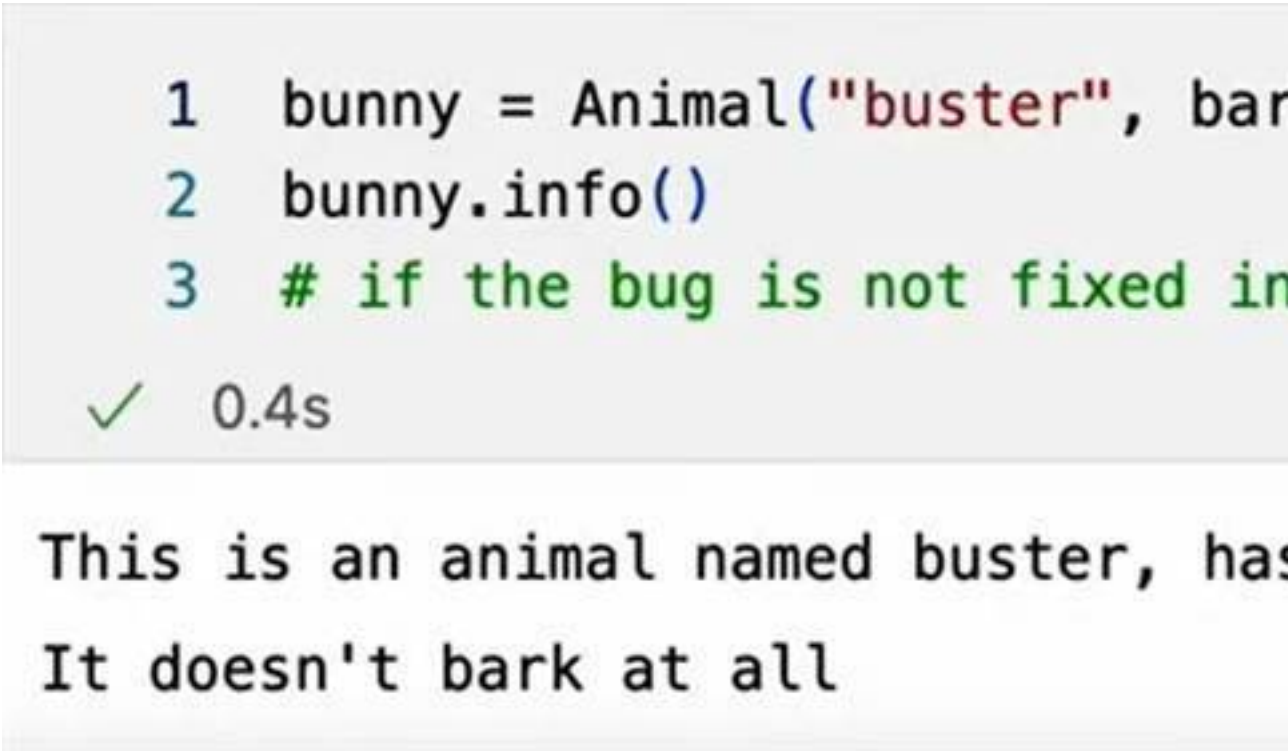
```python
# pass arguments and keyword arguments to a class

class Animal:

    def __init__(self, name, legs=4, barks=True):
        self.name = name
        self.legs = legs
        self.barks = barks

    # there is a bug in here!
    def info(self):
        print(f"This is an animal named {self.name}, has {self.legs} legs")
        if self.barks:
            print("And this one barks!")
        else:
            print("It doesn't bark at all")
```

Если мы исправим эту ошибку, тогда этот код будет работать.



Теперь давайте поговорим о методах класса, добавлении методов и работе с методами класса.

```
3  class Budget:
4
5      def __init__(self, budget):
6          self.budget = budget
7
8      def expense(self, amount):
9          self.budget = self.budget - amount
10         print(f"Budget left: {self.budget}")
11
12 budget = Budget(100)
13
14
```

✓ 0.2s

```
1  # spend a few and get a report
2  budget.expense(23)
3  budget.expense(45)
```

✓ 0.2s

```
Budget left: 77
Budget left: 32
```

Здесь показан класс с двумя методами. И здесь используется конструктор в строке 5, который также является методом. И мы создаем еще один метод в строке 8.

Это метод расхода бюджета, в который мы передаем сумму расхода. Из self.budget мы вычитаем эту сумму, которую мы передаем в метод. И затем мы распечатываем остаток self.budget.

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

    def expense(self, amount):
        self.budget = self.budget - amount
        self.report()

    def report(self):
        print(f"Budget left: {self.budget}")

✓ 0.2s
```

```
# code works just like before, but it is more organized!
april_budget = Budget(400)
april_budget.expense(34)

✓ 0.4s
```

```
Budget left: 366
```

Модифицируем этот класс и разделим задачи, добавив еще один метод, называемый report. Причем новый метод будет вызываться внутри метода expense.

В предыдущем методе мы будет расходовать бюджет, а в новом методе выводить остаток.

```
# method can take arguments and keyword arguments as well
# a method can call another method. Let's add another method to do the reporting
class Budget:

    def __init__(self, budget):
        self.budget = budget

    def expense(self, amount):
        self.budget = self.budget - amount
        self.report()

    def report(self, currency="$"):
        print(f"Budget left: {currency}{self.budget}")

✓ 0.2s
```

```
may_budget = Budget(23)
may_budget.report()
may_budget.report("PER $")
```

Помимо self и обязательных аргументов, в методы также можно передавать и необязательные аргументы. Здесь мы передаем в метод значение валюты. Мы сделали этот аргумент необязательным, и по умолчанию он будет иметь значение в долларах.

Наследование классов — одна из тех вещей, которую необходимо понять. Предпосылка наличия наследования и работы с наследованием — это родительский класс. А затем дочерние классы будут наследовать этот родительский класс, будут иметь определенные атрибуты поведения и все, что определяет родительский класс.

```
1  # create a base class for house pets
2
3  class Pet:
4
5      def eat(self):
6          self.food = self.food - self.appetite
7          print(f"Ate {self.appetite} of food, have {self.food} left")
8
9  pet = Pet()
10 pet.eat()
```

⊗ 0.2s

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/Users/alfredodeza/code/python-bootcamp/python-functions-and-classes/notebooks
ll 2' in <cell line: 10>()
      7         print(f"Ate {self.appetite} of food, have {self.food} left")
      9 pet = Pet()
---> 10 pet.eat()

/Users/alfredodeza/code/python-bootcamp/python-functions-and-classes/notebooks
ll 2' in Pet.eat(self)
      5 def eat(self):
----> 6     self.food = self.food - self.appetite
      7     print(f"Ate {self.appetite} of food, have {self.food} left")

AttributeError: 'Pet' object has no attribute 'food'
```

Здесь у нас есть класс Pet, и он определяет один метод, который называется eat().

И если мы создадим экземпляр этого класса и вызовем метод eat(), мы получим ошибку атрибута, - у класса нет атрибута food.

```
# create to child classes for other house pets like cat and dog and parakeet

class Parakeet(Pet):

    def __init__(self):
        self.food = 100
        self.appetite = 1

class Dog(Pet):

    def __init__(self):
        self.food = 400
        self.appetite = 7

perry = Parakeet()
rufus = Dog()

```

Но это родительский класс, определяющий метод, который мы хотим, чтобы был у всех других дочерних классов. И способ наследования с дочерними классами заключается в передаче родительского класса дочернему классу, как показано на рисунке.

В дочернем классе мы указываем конструктор, в котором определяем атрибуты, используемые в родительском классе.

```
perry.eat()
rufus.eat()
```

✓ 0.2s

```
Ate 1 of food, have 99 left
Ate 7 of food, have 393 left
```

Так как у дочерних классов уже есть этот метод eat, унаследованный от родительского класса, мы можем его вызвать с определенными в дочернем классе атрибутами, и ошибки теперь не будет.

```
1  # demonstrate how other classes have methods that automatically appear
2  for attribute in dir(rufus):
3      if attribute.startswith('_'):
4          continue
5      print(attribute)
```

✓ 0.2s

```
appetite
eat
food
```

Здесь показан способ, как распечатать все атрибуты экземпляра класса.

Модули

Модуль — файл Python, содержащий повторно используемый код, такой как функции или классы.

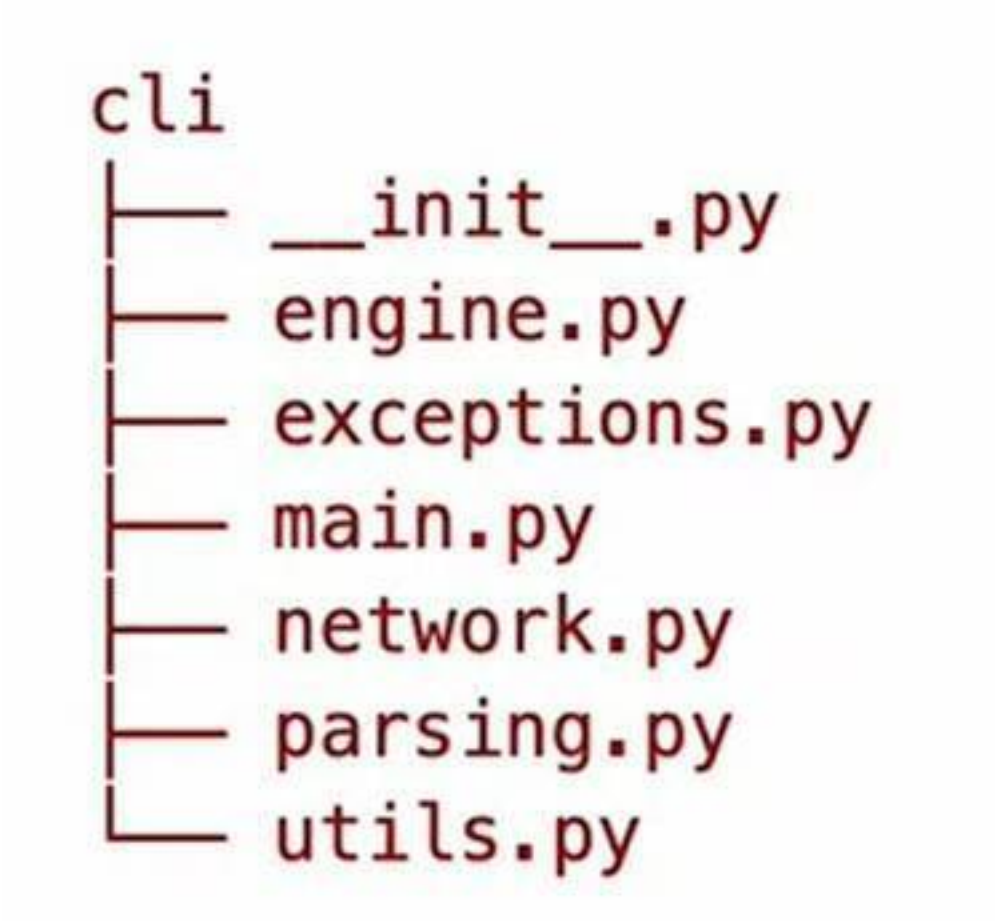
Операция импорт import — извлекает модули, делая их содержимое доступным в текущем пространстве имен.

Каталог virtualenv — автономный каталог, содержащий изолированные пакеты/зависимости Python.

Активация — настраивает оболочку для использования выделенного интерпретатора Python virtualenv.

Программа pip — инструмент Python для установки/управления пакетами и зависимостями.

```
# Import module
import utils
# Access function
print(utils.format_name("Wyclef"))
# Virtualenv creation
python3 -m venv env
# Activate virtualenv
source env/bin/activate
# Install dependency
pip install pandas
# Deactivate when done
deactivate
```



Модуль Python — это не что иное, как файл Python, имеющий расширение py. Это все, что вам нужно знать. Вы можете думать об этом файле, как о способе организации вашего кода. Если у вас есть что-то вроде очень длинного файла Python с несколькими классами, возможно, множеством функций, и это больше, чем, скажем, 200 строк, — это не очень хорошо организованный код.

Представьте, что у вас есть один ящик для всей вашей одежды. И лучше разделить вещи на несколько ящиков и поддерживать все аккуратно.

Здесь показан пример проекта CLI, как пример структуры. Вы можете видеть, что код разделен в нем на несколько файлов py.

У самого Python также есть свои собственные модули.

```
# all you need to create a module is to create a Python file so that
# for example, this is the `os` module in Python (actually os.py)
import os

dir(os)

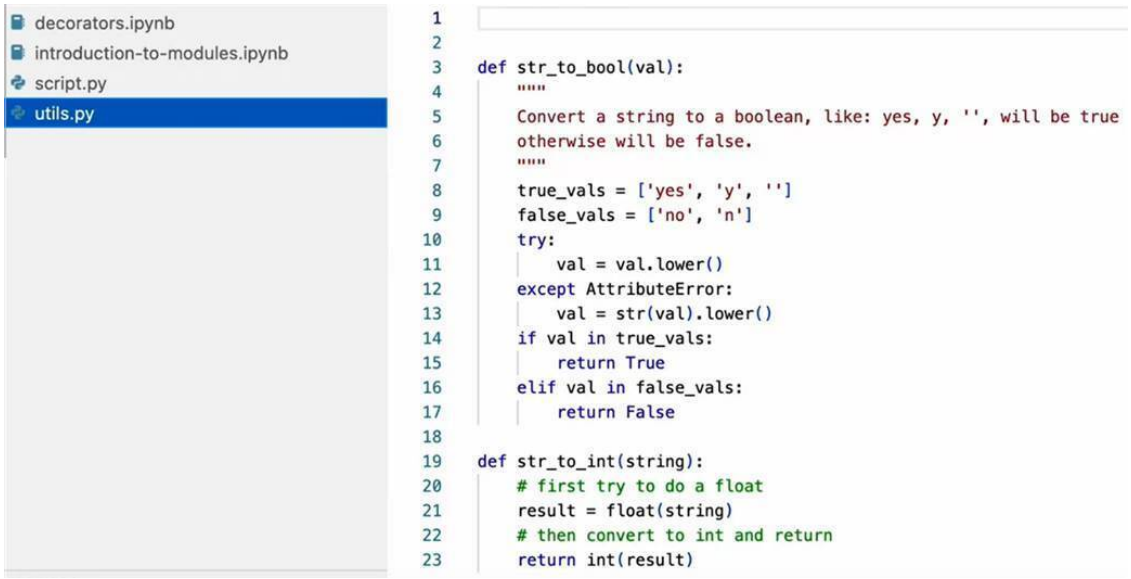
✓ 0.3s
```

```
Output exceeds the size limit. Open the full output data in a text editor
['CLD_CONTINUED',
 'CLD_DUMPED',
 'CLD_EXITED',
 'CLD_TRAPPED',
 'DirEntry',
 'EX_CANTCREAT',
```

Здесь мы импортируем Python модуль OS, и вы видите, что в этом модуле много всего.

По сути, вам нужно знать, что все, что находится в файле .py, считается модулем, и это может помочь вам лучше организовать свой код.

Давайте теперь разберемся с импортом.



Здесь у нас есть utils.py, это модуль Python. Теперь, как вы работаете с импортом.

```
1  # this path has a `utils.py`
   imported
2  import utils
3  dir(utils)
```

✓ 0.3s

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'str_to_bool',
 'str_to_int']
```

Здесь мы импортируем utils, и с помощью команды dir вы можете увидеть все, что доступно в utils, - вы видите, что здесь есть str_to_bool и str_to_int.

```
def str_to_bool(val):
    """
    Convert a string to a boolean, like: yes, y, '', will be true
    otherwise will be false.
    """
    true_vals = ['yes', 'y', '']
    false_vals = ['no', 'n']
    try:
        val = val.lower()
    except AttributeError:
        val = str(val).lower()
    if val in true_vals:
        return True
    elif val in false_vals:
        return False

def str_to_int(string):
    # first try to do a float
    result = float(string)
    # then convert to int and return
    return int(result)
```

Если мы откроем utils, вы увидите, что это две функции в утилите.
Вы можете работать различными способами доступа к этим вспомогательным функциям.

```
import utils
```

```
1  # it is useful to import utils
   functions
2  utils.str_to_bool("yes")
```

✓ 0.2s

```
True
```

```
1  utils.str_to_int("45.3")
```

✓ 0.2s

```
45
```

И самый базовый способ импорта, это использование ключевого слова import и utils, если файл находится в том же каталоге, что и ваш код. Если скажем, utils находился в другом каталоге, то вам придется беспокоиться об установке зависимостей и должным образом предоставления их Python.

Далее вы указываете utils., а затем название функции, с которой вы хотите работать. Это также может быть класс или переменная.

В любом случае, вы получаете доступ с помощью точки, - именно так вы получите доступ к тому, что находится в этом файле или модуле utils.py.

```
1  # you can also import them separately
2  from utils import str_to_int, str_to_bool
3
4  print(str_to_int("11.1"))
5  print(str_to_bool("y"))
```

✓ 0.2s

```
11
True
```

Также вы можете импортировать эти функции отдельно, указывая их в импорте с помощью конструкции from import. И теперь префикс util с точкой не требуется для использования этих функций.

```
1  # sometimes code has several levels deep with directories
2  import program.items
3
4  program.items.some_function()
```

✓ 0.2s

```
'this is a function in a module'
```

```
1  import program
2
3  program.items.some_function
```

И у вас может быть каталог. Обычно каталоги в Python называются пакетами, но это сбивает с толку.

Каталоги используются, чтобы организовать модули по наборам. Здесь у нас есть пакет program, и мы импортируем его, чтобы использовать функцию в модуле items данного пакета.

Есть пара трюков, которые вам следует знать о работе со скриптами Python, и я попытаюсь обобщить некоторые из них.

```
"""
This is a Python script meant to be run from the terminal (or command line)

It uses the `if __name__` construct at the end of the script to hook into any function.
In this case, it connect to the main() function

This script doesn't do much, it will just report the arguments being passed into the
script. It should serve as a foundation to create a more powerful script
"""
import sys


def main(arguments):
    print("This is the main function, and it has access to the following variables:")
    for argument in arguments:
        print(argument)


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

Начнем с этого небольшого текста документации здесь в вверху кода, — это называется строкой документации, когда у вас есть тройные кавычки в самом верху. И разработчики Python обычно используют этот прием, чтобы обеспечить хорошую документацию кода, потому что это выглядит лучше, чем просто использование комментариев.

Итак, что делает этот скрипт Python?

Он определяет основную функцию main, и эта основная функция принимает некоторые аргументы.

Теперь, способ выполнить некоторый код только при запуске скрипта из командной строки, заключается в использовании этого небольшого фрагмента здесь, в самом низу.

if __name__ == "__main__":

Если же модуль будет импортирован в другой скрипт, это код выполнен не будет.

if __name__ == "__main__":

print("Script is run directly")

else:

print("Script is imported")

Таким образом все, что это означает, это - эй, Python, если я выполняюсь на терминале, в командной строке, выполни этот код.

Если же вы импортируете этот модуль, вы просто сможете использовать функцию main, как, скажем, утилиту, и код после if __name__ == "__main__": выполняться не будет.

И еще мы здесь видим функцию sys.argv модуля sys, которая возвращает список параметров командной строки, передаваемых скрипту Python.

```
python script.py argument --help -f key=value
This is the main function, and it has access to the following variables:
script.py
argument
--help
-f
key=value
```

То есть все что вы передадите скрипту в качестве аргументов в командной строке, может быть использовано в коде с помощью sys.argv.

Прежде чем мы перейдем к другим вещам, очень важно понять, что у Python есть проблема с пакетированием. В целом, создание пакетов Python, установка зависимостей и работа с пакетами довольно проблематичны. Поэтому мы посмотрим здесь, как работать с виртуальными средами, чтобы правильно установить зависимости.

```
 ls
python3 -m venv venv
ls

venv
```

Виртуальное окружение или среда позволяет создавать изолированные среды для проектов Python, чтобы между ними не было конфликтов.

Мы можем создать виртуальное окружение с помощью команды python3. Здесь флаг -m — это параметр командной строки, который позволяет запускать модуль как скрипт. Используя флаг -m, вы можете вызывать модули Python как отдельные программы.

Далее мы запускаем как скрипт модуль venv, который поддерживает создание виртуальных сред, каждая из которых может иметь собственный независимый набор пакетов Python, установленных в их каталогах.

В результате будет создана виртуальная среда с именем venv. Команда ls используется для вывода списка файлов и каталогов, содержимого вашего текущего рабочего каталога. После команды python3 мы видим с помощью ls, что появился каталог venv. И командой tree мы выводим дерево каталога виртуальной среды.

```
venv
├── bin
│   ├── Activate.ps1
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── easy_install
│   ├── easy_install-3.8
│   ├── pip
│   ├── pip3
│   ├── pip3.8
│   ├── python -> python3
│   └── python3 -> /Library/Developer/CommandLineTools/usr/bin/python3
├── include
├── lib
│   └── python3.8
│       └── site-packages
│           ├── easy_install.py
│           ├── pip
│           │   ├── __init__.py
│           │   ├── __main__.py
│           │   ├── _internal
│           │   │   ├── __init__.py
│           │   │   ├── build_env.py
│           │   │   ├── cache.py
│           │   │   ├── cli
```

Все, что вам нужно здесь знать, это то, что виртуальная среда дает вам возможность устанавливать пакеты самодостаточным образом в этом каталоге, и у вас не возникнет проблем, если вы установите что-то неправильно. Единственное, что вам нужно будет сделать в этом случае, это удалить каталог виртуальной среды и создать новый.

Итак, мы создали каталог, но не активировали виртуальную среду.

```
ls
venv
source venv/bin/activate
pip install flask

Collecting flask
  Downloading Flask-2.1.2-py3-none-any.whl (95 kB)
     |████████████████████████████████| 95 kB 6.4 MB/s
Collecting click>=8.0
  Downloading click-8.1.3-py3-none-any.whl (96 kB)
     |████████████████████████████████| 96 kB 22.7 MB/s
```

Чтобы активировать виртуальную среду, мы выполним команду source, а затем venv/bin/activate. Теперь вы можете использовать python, который находится в каталоге venv. То же самое и с pip. Это означает, что, если я хочу установить, скажем, Flask, я наберу команду pip install flask, и Flask установится в каталог venv вместе со всеми зависимостями.

```
deactivate
# If your virtual environment is in a directory called 'venv':
rm -r venv
```

Теперь, чтобы деактивировать, удалить виртуальную среду, вы должны выполнить команду deactivate.

Управление файлами с помощью Python

Как IT-специалисту, вам, вероятно, часто придется работать с файлами и каталогами на компьютере. Когда вам нужно работать с большим количеством файлов и каталогов, автоматизация может оказать огромную помощь.

Как системный администратор, в своей работе я постоянно взаимодействую с файлами и каталогами, иногда даже если у меня нет намерения их изменять. Например, я использую Python для проверки существования определенного файла в файловой системе перед выполнением каких-либо других операций.

Как вы, возможно, помните, операционные системы, такие как Mac OS, Windows и Linux, используют файловые системы для организации и управления хранением и доступом к данным. Данные обычно хранятся на диске и сохраняются в файлах, которые находятся в контейнерах, называемых каталогами или папками.

Файловые системы обычно организованы в виде древовидной структуры, где каталоги и файлы вложены под своими родителями. И мы знаем, где находится ресурс, такой как каталог или файл, в этой древовидной структуре по его пути.

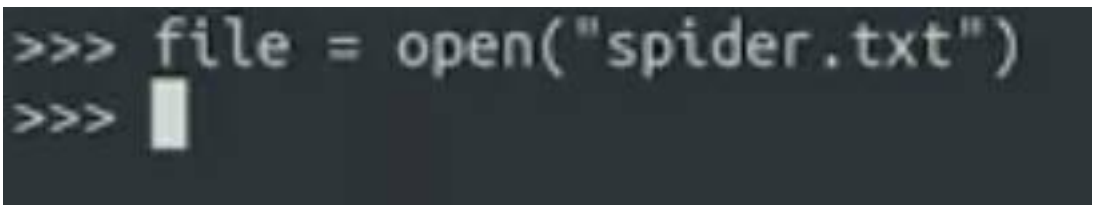
Абсолютный путь — это полный путь к ресурсу в файловой системе. Например, на компьютере с Windows абсолютный путь к папке пользователя Jordan будет C:\Users\Jordan. На компьютере с Linux абсолютный путь к соответствующему каталогу будет /home/jordan. Мы называем это абсолютным путем, потому что не имеет значения, где в файловой системе выполняется наш скрипт, абсолютный путь всегда приведет нас к ресурсу.

С другой стороны, относительные пути используют только часть пути, чтобы показать, где находится ресурс, относительно текущего рабочего каталога. Относительные пути — это сокращенный способ, который можно использовать, чтобы не указывать полный путь к файлу. Но имейте в виду, что они имеют смысл только относительно текущего местоположения.

Например, если мы выведем содержимое каталога examples, мы получим разные результаты в зависимости от того, что является текущим каталогом. Если наш текущий каталог — /home/jordan, мы получим содержимое /home/jordan/examples. Но если текущий каталог — /user/share/doc/python3, мы получим содержимое /user/share/doc/python3/examples.

Ранее мы предоставляли информацию нашим скриптам, создавая переменные и передавая параметры нашим функциям с нужными нам значениями. Это хорошо работает для небольших скриптов, но обычно не идеально для больших задач. При обработке больших объемов данных хорошо иметь возможность считывать эти данные из файлов.

В программировании мы постоянно работаем с файлами. Это настолько полезная задача, что большинство языков программирования имеют встроенную функцию работы с файлами, являющуюся частью основного набора возможностей. Python не является исключением. Он предоставляет нам объекты файлов, которые мы можем использовать для чтения и записи в файлы.



Чтобы открыть файл на компьютере с именем spider.txt, мы можем написать следующий код. Здесь мы создаем новый объект файла и присваиваем его переменной с именем file. Пара-

метр, который мы передали функции open, это имя файла, который мы хотим открыть. В данном случае мы предполагаем, что файл, который мы хотим прочитать, находится в том же каталоге, что и скрипт, который мы запускаем, но мы можем с таким же успехом передать абсолютный путь, чтобы открыть файл в другом каталоге.

Когда мы открываем файл, как мы делаем в этом примере, операционная система проверяет, есть ли у нас разрешения на доступ к этому файлу, а затем предоставляет нашему коду дескриптор файла. Это токен, генерируемый ОС, который позволяет программам выполнять дополнительные операции с файлом.

В Python этот дескриптор файла хранится как атрибут объекта files.

Объект file предоставляет нам набор методов, которые мы можем использовать для работы с файлом. Теперь, с помощью этого объекта file, мы можем прочитать содержимое файла и вывести его на экран.

```
>>> file = open("spider.txt")
>>> print(file.readline())
The itsy bitsy spider climbed up the waterspout.

>>> print(file.readline())
Down came the rain

>>>
```

Здесь мы использовали метод readline. Он позволяет нам прочитать одну строку из файла. Давайте вызовем его еще раз, посмотрим, что произойдет. Отлично. На этот раз мы получили вторую строку файла.

Итак, как это работает? Каждый раз, когда мы вызываем метод readline, объект файла обновляет текущую позицию в файле. Таким образом, он продолжает двигаться вперед.

```
>>> print(file.read())
and washed the spider out.
Out came the sun
and dried up all the rain
and the itsy bitsy spider climbed up the spout again.

>>>
```

Мы также можем вызвать метод read, которыйтчитает с текущей позиции до конца файла, а не только одну строку.

В этом файле всего шесть строк. Мы читаем первые две с помощью вызовов readline, а затем остальные четыре вместе с помощью вызова read.

Как и readline, метод read начинает чтение с того места, где мы сейчас находимся в файле. Но вместо одной строки он читает весь файл до конца.

```
>>> file.close()
>>>
```

Наконец, мы закрываем файл с помощью метода close.

Этот шаблон «открытие-использование-закрытие» — типичный способ работы с файлами в большинстве языков программирования. Закрывать файлы после того, как вы их открыли, — хорошая идея по нескольким причинам.

Во-первых, когда файл открывает ваш скрипт, файловая система обычно блокирует его, и поэтому никакие другие программы или скрипты не могут использовать его, пока вы не закончите.

Во-вторых, существует ограниченное количество файловых дескрипторов, которые вы можете создать, прежде чем файловая система их исчерпает. Хотя это число может быть большим, можно открыть много файлов и истощить ресурсы файловой системы. Это может произойти, если мы открываем файлы, например, в цикле.

В-третьих, оставление открытых файлов без присмотра может привести к состояниям гонки, которые возникают, когда несколько процессов пытаются одновременно изменять и читать один и тот же ресурс, и могут вызывать всевозможные неожиданные проблемы. В состоянии гонки никто не выигрывает.

Теперь я открою вам небольшой секрет. Я забываю закрывать файлы, и вы, возможно, согласитесь со мной, что бывает довольно сложно отслеживать, какие файлы вы открыли, а затем помнить о том, чтобы их закрыть. К счастью, создатели Python со мной согласны. Чтобы помочь нам не забывать закрывать файл после завершения работы с ним, Python позволяет создавать блоки кода, используя ключевое слово «with». Давайте посмотрим, как это выглядит.

```
>>> with open("spider.txt") as file:
...     print(file.readline())
...
The itsy bitsy spider climbed up the waterspout.

>>>
```

Как видите, ключевое слово «with» позволяет создать блок кода, содержащий действия, которые мы хотим выполнить с файлом. В данном случае мы хотим взять строку данных из файла и вывести её на экран.

При использовании блока «with» Python автоматически закроет файл. Поэтому нам не нужно помнить об этом самостоятельно.

И подход с открытием-использованием-закрытием, и подход с использованием «with» каждый имеют свои преимущества.

Использование блока «with» — это хороший способ открыть и работать с одним файлом, а затем автоматически закрыть его в конце блока.

С другой стороны, использование «open» вне блока означает, что мы можем использовать объект файла в других местах нашего кода. Таким образом, мы не ограничены одним-

единственным блоком. Но при таком подходе нам нужно помнить о том, чтобы закрыть его после завершения работы.

Наряду с этим с файловыми объектами можно выполнять итерацию, также как и с другии последовательностями Python, такими как список или строки. Это действительно полезно, когда вы хотите обрабатывать файл построчно. Скажем, например, вы хотите сделать всю строку заглавными буквами перед печатью.

```
>>> with open("spider.txt") as file:
...     for line in file:
...             print(line.upper())
...
THE ITSY BITSY SPIDER CLIMBED UP THE WATERSPOUT.

DOWN CAME THE RAIN

AND WASHED THE SPIDER OUT.

OUT CAME THE SUN

AND DRIED UP ALL THE RAIN

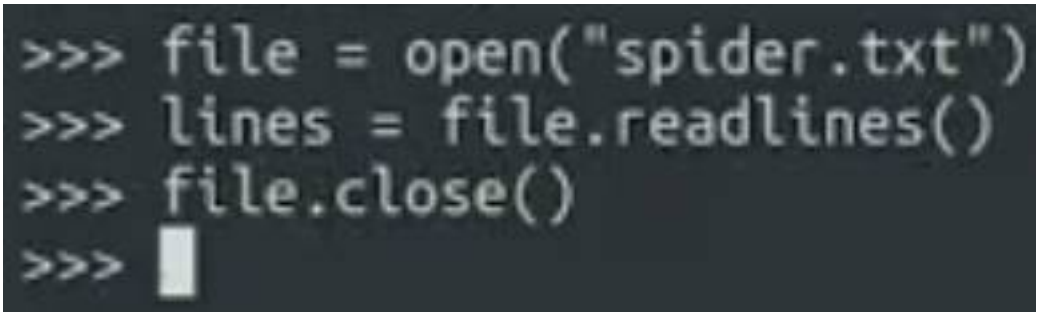
AND THE ITSY BITSY SPIDER CLIMBED UP THE SPOUT AGAIN.

>>>
```

Но что это за странные пустые строки между содержимым? Происходит то, что в конце каждой строки файла появляется новый символ строки. Поэтому, когда Python читает строку файла с помощью line, строковая переменная всегда будет иметь в конце новый символ строки. Другими словами, символ новой строки не удаляется при вызове строки чтения. Когда мы просим Python напечатать строку, функция печати добавляет еще один символ новой строки, создавая пустую строку. Что мы можем сделать, чтобы не получить пустые строки?

```
>>> with open("spider.txt") as file:
...     for line in file:
...             print(line.strip().upper())
...
THE ITSY BITSY SPIDER CLIMBED UP THE WATERSPOUT.
DOWN CAME THE RAIN
AND WASHED THE SPIDER OUT.
OUT CAME THE SUN
AND DRIED UP ALL THE RAIN
AND THE ITSY BITSY SPIDER CLIMBED UP THE SPOUT AGAIN.
>>>
```

Мы можем использовать строковый метод, удаляя все окружающие пробелы, включая табуляции и новые строки. Теперь, когда мы используем метод strip для удаления символа новой строки, мы получим вывод без пустых строк, как и мы хотели.



Еще один способ работы с содержимым файла заключается в чтении строк файла в список. Затем мы можем что-то делать со списком, например, сортировка содержимого. Для этого мы открываем файл и используем метод readlines.

Несмотря на то, что файловый объект теперь закрыт, переменная lines содержит список строк в файле, поэтому мы можем работать с ней.

```
>>> file = open("spider.txt")
>>> lines = file.readlines()
>>> file.close()
>>> lines.sort()
>>> print(lines)
['Down came the rain\n', 'Out came the sun\n', 'The itsy bitsy spider climbed up the waterspout.\n'
, 'and dried up all the rain\n', 'and the itsy bitsy spider climbed up the spout again.\n', 'and wa
shed the spider out.\n']
>>>
```

Например, давайте отсортируем его и распечатаем. Есть две вещи, на которые стоит обратить внимание в этом коде.

Сначала строки были отсортированы в алфавитном порядке, поэтому они больше не располагаются в том порядке, в котором они были в файле. Во-вторых, мы видим, что Python отображает символ новой строки «\n» при печати списка строк. Это способ явно показать, что в этих строках есть новый символ строки.

Такие методы, как read или readlines, которые читают весь файл сразу, полезны, но мы должны быть осторожны при чтении всего содержимого файла в переменную программы. Если файл очень большой, для его хранения требуется много памяти компьютера, что может привести к снижению производительности.

Если размер файла составляет всего несколько килобайт, его можно прочитать и полностью обработать в памяти. Но для больших файлов, более эффективно обработать его построчно.

Теперь, когда мы знаем, как читать файлы в Python, синтаксис записи в них, вероятно, не покажется таким уж странным.

```
>>> with open("novel.txt", "w") as file:
...     file.write("It was a dark and stormy night")
...
30
>>>
```

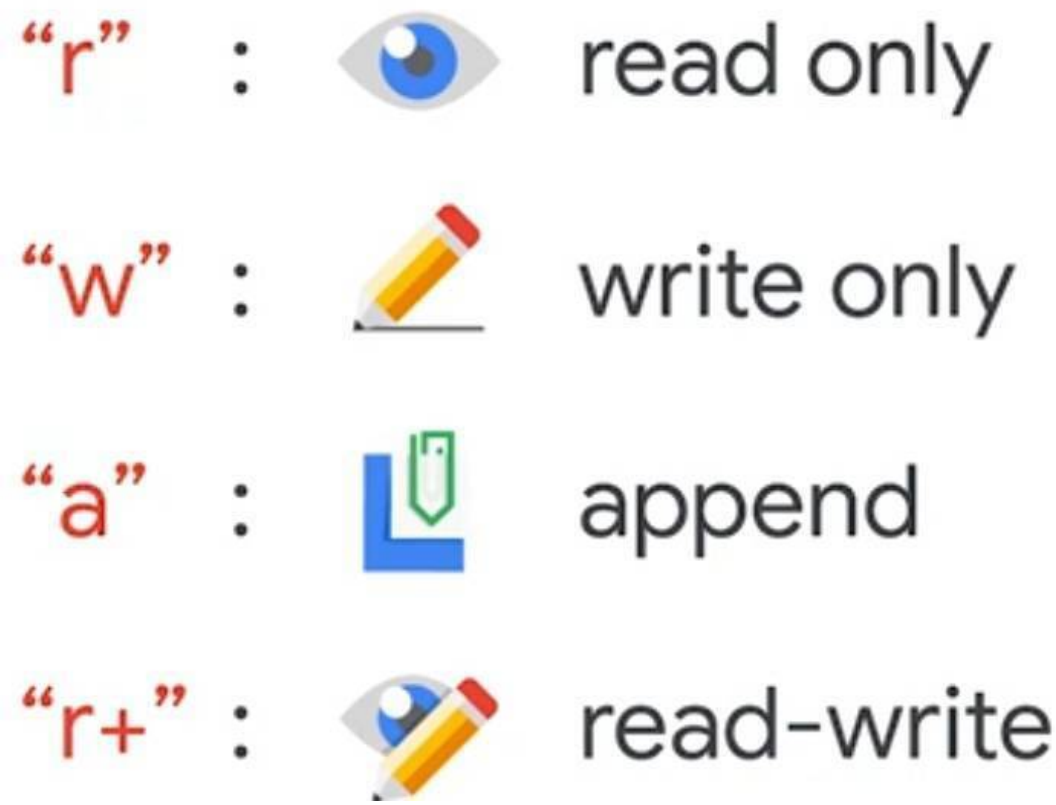
Вы можете догадаться, что использование метода write файлового объекта записывает в него содержимое, а не читает его. Однако второй аргумент метода open является новым. Так что же означает буква w?

Файловые объекты можно открывать в нескольких различных режимах. Режим аналогичен разрешению файла. Он определяет, что вы можете делать с файлом, который вы только что открыли.

По умолчанию функция открытия использует режим r, который означает только чтение. Вы получите сообщение об ошибке, если попытаетесь записать файл, открытый в режиме только для чтения. Поскольку значением по умолчанию является только чтение, нам не нужно передавать r в качестве второго аргумента, когда мы просто хотим прочитать файл.

Однако запись в файл – это совсем другая история. Символ w сообщает функции open, что мы хотим открыть файл только для записи. Если файл не существует, Python создаст его. Если файл существует, то его текущее содержимое будет перезаписано тем, что мы решим записать с помощью нашего скрипта.

Важно помнить, что при открытии файла в режиме только записи вы не сможете прочитать его содержимое. Если вы попытаетесь это сделать, интерпретатор выдаст ошибку.



Если вы хотите добавить содержимое в уже существующий файл, вы можете сделать это, используя другие режимы, например a для добавления содержимого в конец существующего файла или r+ для режима чтения и записи, в котором вы можете как читать содержимое, так и перезаписывать его.

Если вы открываете файл для записи и файл уже существует, старое содержимое будет удалено, как только файл будет открыт. Представьте себе, что вы случайно удалили важный контент в файле. Поэтому дважды проверьте, что вы открываете правильный файл в правиль-

ном режиме. Открытие в режиме записи w означает, что вы будете записывать поверх любых предыдущих записей в этом файле.

И наконец, что делает эта одинокая запись 30, болтающаяся в конце нашего кода? Это возвращаемое значение метода записи. В случае успеха этот метод возвращает количество записанных символов. В данном случае — 30.

Наряду с режимами «только чтение», «запись», «добавление» и «чтение и запись», функция open поддерживает множество других режимов. Вы найдете всю документацию по этому вопросу в официальном справочнике.

Чтобы программы, включая программы Python, работали правильно, приложение должно вызывать один или несколько файлов, которые хранятся на локальном компьютере или находятся в облаке, используя пути к файлам. Путь к файлу – это конкретное расположение файла на компьютере или веб-сервере. В коде Python это записывается как строка, повторяющая путь к файлу на локальном компьютере или к файлу на удаленном сервере, который получен по его URL-адресу.

Вы также можете использовать пути к файлам для вызова переменных среды такие как библиотеки, приложения и даже другие языки, такие как C+ или JavaScript.

Есть два типа пути, относительный и абсолютный.

Относительные пути к файлам по умолчанию относятся к определенному каталогу, в котором находится команда Python, которая изначально была запущена. Обычно им отдают предпочтение поскольку они достаточно гибки, чтобы вызывать необходимые файл на любом компьютере.



Абсолютные пути к файлам, с другой стороны, называют точное местоположение файла, - имя диска, затем каталог, затем имя файла. Программисты Python склонны избегать абсолютных путей к файлам, поскольку имена дисков могут измениться.

Абсолютные пути к файлам также изменяются в зависимости от операционной системы.

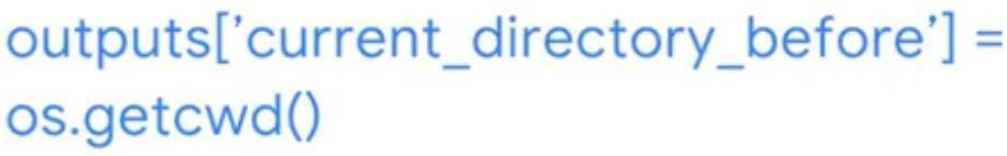
Хотя сценарии Python могут выполняться на любом компьютере, сценарии, основанные на абсолютных путях, могут не работать.

Специфическая структура пути к вашему файлу зависит от операционной системы или ОС, которую вы используете для своего кода. Например, пути к файлам Windows обычно начинаются с имени диска, например C или D. В Mac и Linux путь начинается с косой черты, что является корнем.

Каталоги в пути к файлу также разделяются косой чертой. Windows обычно использует обратную косую черту.

Корневые косые черты и буквы дисков важны, когда вы используете абсолютные пути к файлам.

Чтобы получить текущий каталог, используйте команду CWD в своем коде. os.getcwd, за которой следует имя файла, к которому вы хотите получить доступ.



Чтобы ссылаться на внешние файлы, вам также потребуется команда CWD.

Получение и установка текущего каталога — это только один из способов использования путей к файлам в Python для ссылки на внешние файлы.

```
outputs['files_and_directories'] = os.listdir()
```

Вы также можете перечислить файлы и каталоги, чтобы найти нужный вам путь к файлу, используя os.listdir().

```
outputs['path_value'] = os.environ.get('PATH')
```

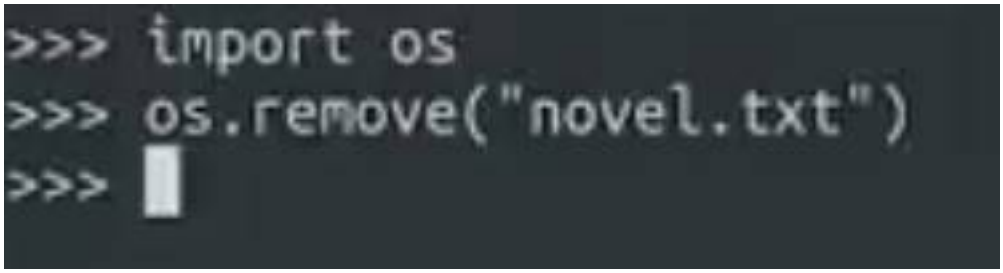
Вы также можете получить доступ к путям к файлам для переменных среды, используя os.environ.get.

Таким образом, мы можем читать файлы, перебирать содержимое файлов, а также записывать содержимое в файлы. Это самые распространенные операции, но есть еще много вещей, которые нам, возможно, придется делать при работе с файлами в наших скриптах. Возможно, придется удалять, переименовать или перемещать файлы, или может понадобиться информация о файле, например, когда он был последний раз изменялся или его текущий размер.

Для этих операций мы будем использовать функции, предоставляемые модулем ОС. Этот модуль обеспечивает уровень абстракция между Python и операционной системой. Это позволяет нам взаимодействовать с операционной системой, при этом мы не знаем, работаем ли мы в Windows, Mac, Linux или любой другой операционной системе, поддерживаемой Python. Это означает, что вы можете писать и тестировать сценарий в одной операционной системе, например Windows, а затем запустить его в другой операционной системе, такой как Linux.

Но есть одна вещь, на которую следует обратить внимание: пути могут быть разными в разных операционных системах. Поэтому всякий раз, когда мы используем абсолютный путь в нашем коде, нам нужно убедиться, что мы можем предоставить альтернативы платформам, которые мы хотим поддерживать.

Модуль ОС позволяет нам делать почти все те же задачи, которые мы обычно можем выполнять при работе с файлами из командной строки. Мы можем изменить права доступа к файлам и удалять или переименовывать файлы с помощью кода.



Давайте посмотрим на это в действии. Итак, сначала мы импортируем модуль ОС. Затем мы вызываем функцию удаления. Теперь файл удален.

```
>>> import os
>>> os.remove("novel.txt")
>>> os.remove("novel.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'novel.txt'
>>>
```

Посмотрим, что произойдет, если мы попытаемся удалить его еще раз. Если мы попытаемся удалить файл, который не существует, функция выдаст ошибку.

```
>>> os.rename("first_draft.txt", "finished_masterpiece.txt")
>>>
```

Мы можем легко переименовать файл с помощью функции переименования. Первый параметр для переименования функции — старое имя файла, а второе — новое имя.

Если бы мы попытались сделать это для файла, который не существует, мы снова получим ошибку «файл не найден».

Итак, как мы проверим, что файл существует или нет? Внутри ОС есть субмодуль, связанными с информацией о файлах, например, существуют они или нет. Это называется подмодулем пути ОС.

```
>>> os.path.exists("finished_masterpiece.txt")
True
>>> os.path.exists("userlist.txt")
False
>>>
```

И мы можем использовать функцию существования в этом модуле, чтобы проверить, существует ли файл.

Мы можем получить гораздо больше информации о файлах с помощью функций в модуле os.path.

```
>>> os.path.getsize("spider.txt")
192
>>> os.path.getmtime("spider.txt")
1578322923.8999994
>>>
```

Например, чтобы проверить размер файла, мы можем использовать функцию getsize, которая возвращает размер файла в байтах.

Чтобы проверить, когда файл был модифицирован, можно использовать функцию getmtime. Что это за длинный номер? Это не похоже на время, не так ли? Это потому, что это временная метка. В данном случае это Unix временная метка. Она представляет собой количество секунд с 1 января 1970 года. Так было принято много лет назад для хранения времени, связанного с файлами на компьютерах.

```
>>> import datetime
>>> timestamp = os.path.getmtime("spider.txt")
>>> datetime.datetime.fromtimestamp(timestamp)
datetime.datetime(2020, 1, 6, 7, 2, 3, 899999)
>>>
```

Мы можем использовать модуль datetime, чтобы сделать это легче читаемым.

Здесь мы используем метод fromtimestamp класса datetime внутри модуля datetime.

```
>>> os.path.abspath("spider.txt")
'/home/user/spider.txt'
>>>
```

В наших примерах мы использовали относительные пути файлов без необходимости указывать их полные пути.

В некоторых случаях нам может потребоваться указать, где именно файл должен находится в нашем скрипте. Именно здесь может помочь функция abspath.

Функция abspath принимает имя файла и превращает его в абсолютный путь.

Python использует текущий рабочий каталог, в котором запускается сценарий, чтобы начать поиск файла и создает полный путь. Это полезно, если вы хотите сохранить полный путь к файлу или получить доступ к файлу независимо от того, какой текущий каталог.

Если вам нужно работать с файлами в ваших скриптах, скорее всего, вам придётся работать с каталогами. Например, вам может потребоваться обработать все файлы, находящиеся в определённом каталоге, или создать каталог с результатами анализа данных.

Как и в случае с файлами, Python предоставляет множество различных функций, позволяющих создавать, удалять и просматривать содержимое каталогов.

```
>>> print(os.getcwd())
/home/user
>>>
```

Чтобы проверить, в каком текущем каталоге выполняется ваша программа на Python, вы можете использовать метод «getcwd».

Если вы используете систему, подобную Unix, вы, возможно, помните, что команда, которая выводит рабочий каталог, называется «pwd». В данном случае текущий каталог — это домашний каталог пользователя, который является каталогом по умолчанию, с которого мы начинаем работу при открытии терминала в Linux.

```
>>> os.mkdir("new_dir")
>>>
```

Для создания каталога мы используем функцию «mkdir». Эта функция имеет то же имя, что и команды Windows и Linux, которые делают то же самое.

```
>>> os.chdir("new_dir")
>>> os.getcwd()
'/home/user/new_dir'
>>>
```

Вы также можете менять каталоги в своей программе, используя функцию chdir и передавая в качестве параметра каталог, в который вы хотите перейти.

Как и в других функциях, которые мы рассматривали, мы можем использовать относительные или абсолютные пути для этого. Как видите, мы изменили текущий рабочий каталог на новый каталог, который мы создали внутри домашнего каталога пользователя.

```
>>> os.mkdir("newer_dir")
>>> os.rmdir("newer_dir")
>>>
```

Мы используем mkdir для создания каталогов, и мы можем использовать функцию rmdir для их удаления, но функция rmdir будет работать только если каталог пуст. Если мы попы-

таемся удалить каталог, который содержит файлы, мы получим ошибку. Нам нужно сначала удалить все файлы, и подкаталоги в этом каталоге, прежде чем мы сможем его удалить, но как мы можем узнать, что находится в этом каталоге?

```
>>> import os
>>> os.listdir("website")
['images', 'index.html', 'favicon.ico']
>>>
```

Есть несколько методов, которые мы можем использовать для этого. Функция os.listdir возвращает список всех файлов и подкаталогов в заданном каталоге.

```
>>> dir = "website"
>>> for name in os.listdir(dir):
...     fullname = os.path.join(dir, name)
...     if os.path.isdir(fullname):
...         print("{} is a directory".format(fullname))
...     else:
...         print("{} is a file".format(fullname))
...
website/images is a directory
website/index.html is a file
website/favicon.ico is a file
>>>
```

Итак, у нас есть список строк. Поскольку это просто строки, мы не знаем, являются ли они каталогами или файлами. Чтобы узнать, что это такое, мы можем использовать функции, такие как os.path.isdir, но обратите внимание, что список содержит только имена файлов. Если мы хотим узнать, является ли один из этих файлов каталогом, нам нужно использовать os.path.join для создания полного пути.

Во-первых, мы определяем переменную dir с именем каталога, который мы хотим проверить. Это делает наш код более читаемым и более удобным в использовании.

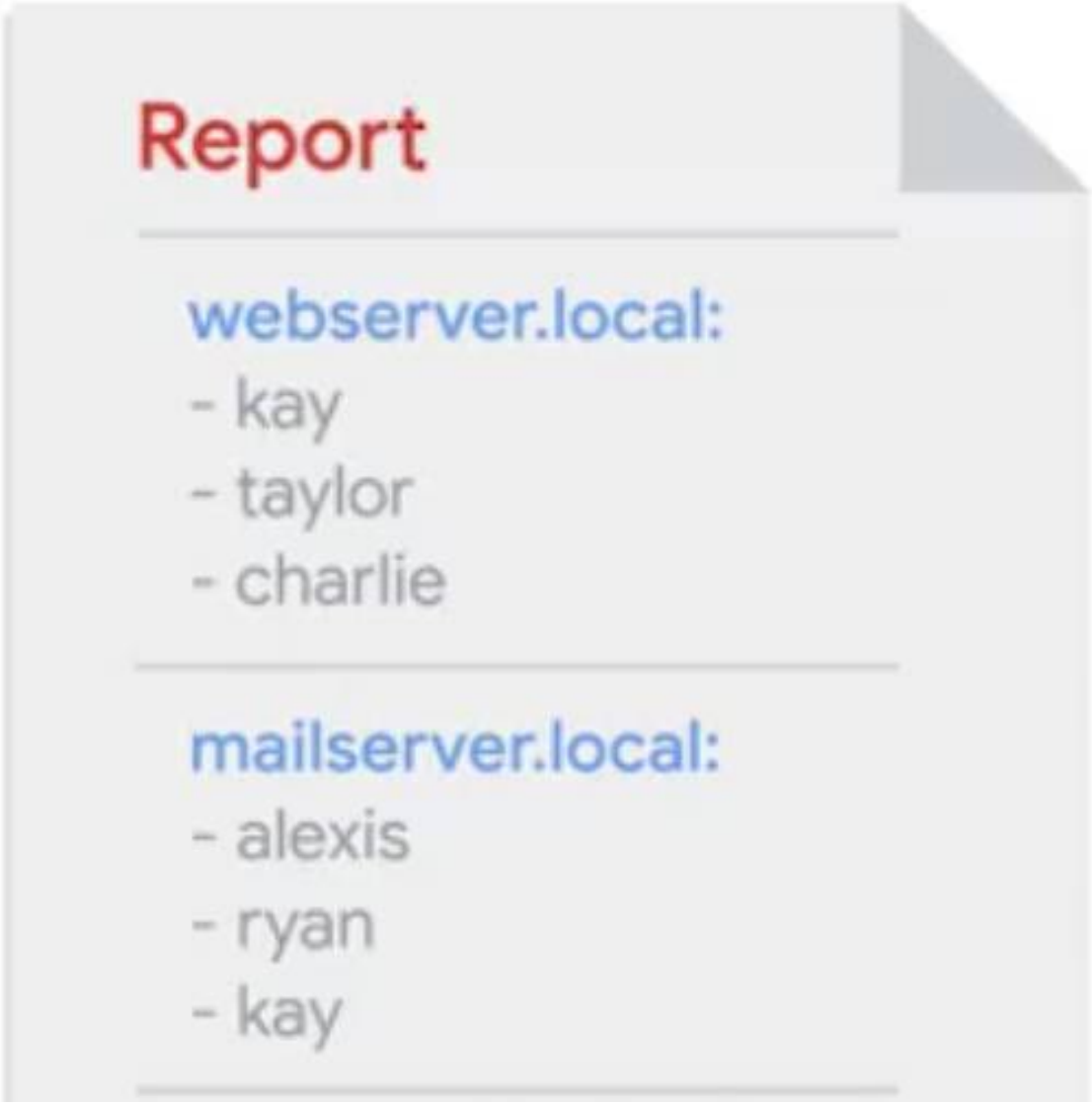
Затем мы перебираем имена файлов, возвращаемые функцией os.listdir. Из предыдущего выполнения этой функции мы знаем, что это просто имена файлов без указания каталога. Поэтому, используя os.path.join, мы объединяем каталог с каждым из этих имен файлов и создаем строку.

Наконец, мы используем эту строку, чтобы вызвать os.path.isdir и проверить, является ли это каталогом или файлом.

Возможно, вы задаетесь вопросом, что это за функция join? Кажется, она просто добавляет косую черту между двумя строками. Функция join позволяет нам снова быть независимыми от операционной системы. В Linux и MacOS части файла разделяются с помощью прямой косой черты. В Windows они разделяются с помощью обратной косой черты. Используя функцию os.path.join вместо явного добавления косой черты, мы можем убедиться, что наши скрипты работают

со всеми операционными системами. Это еще один из тех удобных маленьких инструментов, которые помогают нам избегать ошибок при работе на разных платформах.

Когда мы говорили о том, как читать файлы, мы рассматривали файлы с текстовыми строками, последовательно расположенными. Но данные поступают во множестве различных форматов, помимо текста. И вам может потребоваться работать с некоторыми из них в ваших скриптах. Форматы придают данным структуру.



Если у нас есть файл, который подробно описывает, какие пользователи вошли в систему на этой машине, то при чтении файла мы знаем, как его разобрать, чтобы получить нужную информацию. Разбор файла означает анализ его содержимого для правильной структуризации данных.

Мы используем множество различных форматов файлов, для структурирования, хранения и передачи данных. Возможно, некоторые из них вам уже знакомы. Например, HTML — это формат разметки, определяющий содержимое веб-страницы. JSON — это формат обмена данными, обычно используемый для передачи данных между компьютерами в сетях, особенно в интернете. CSV, или значения, разделенные запятыми, — это очень распространенный формат данных, используемый для хранения данных в виде сегментов текста, разделенных запятыми.

В стандартной библиотеке Python вы найдете классы и модули для работы со многими из этих форматов данных, включая CSV и HTML. Для менее распространенных форматов файлов или более сложных методов обработки данных вы найдете больше библиотек, доступных в качестве дополнительных модулей Python.

Мы рассмотрим, как можно использовать модуль CSV для обработки CSV-файлов. Это не только покажет, как мы можем использовать Python, для работы с определенным форматом данных. Умение работать с CSV-файлами — очень полезный навык. Этот формат позволяет нам легко хранить и извлекать информацию, которая может понадобиться для наших скриптов, например, данные о сотрудниках нашей компании или внутренней сети компьютера.

Многие программы способны экспортировать данные в CSV-файлы, например, табличные редакторы, такие как Microsoft Excel или Google Sheets.

CSV расшифровывается как Comma Separated Values (значения, разделённые запятыми). CSV — довольно простой формат. Эти файлы хранятся в виде обычного текста. И каждая строка в CSV-файле обычно представляет собой отдельную запись данных. Каждое поле в этой записи разделяется запятой, а содержимое поля хранится между запятыми.

Стандартная библиотека Python включает в себя модуль, который позволяет нам читать, создавать и обрабатывать CSV-файлы.

```
>>> import csv
>>> f = open("csv_file.txt")
>>> csv_f = csv.reader(f)
```

Итак, мы будем использовать модуль CSV. Для этого нам нужно импортировать его так же, как мы это делали с другими модулями. Теперь, прежде чем мы сможем разобрать CSV-файл, нам нужно открыть его тем же способом, что и раньше. И теперь мы можем разобрать этот файл, используя модуль CSV.

```
>>> import csv
>>> f = open("csv_file.txt")
>>> csv_f = csv.reader(f)
>>> for row in csv_f:
...     name, phone, role = row
...     print("Name: {}, Phone: {}, Role: {}".format(name, phone, role))
...
Name: Sabrina Green, Phone: 802-867-5309, Role: System Administrator
Name: Eli Jones, Phone: 684-3481127, Role: IT specialist
Name: Melody Daniels, Phone: 846-687-7436, Role: Programmer
Name: Charlie Rivera, Phone: 698-746-3357, Role: Web Developer
>>> f.close()
>>>
```

Мы можем перебирать его содержимое и получать доступ к информации.

Переменная row содержит каждую строку в CSV-файле. Эта переменная представляет собой список, где каждому полю в CSV соответствует один элемент списка. Первое поле — это имя, второе — номер телефона, а третье — должность.

Таким образом, мы можем распаковать значения, чтобы использовать переменные для ссылки на них. Помните, что для этого нам нужно, чтобы количество переменных слева от знака равенства было точно таким же, как и длина последовательности справа.

Когда мы распаковали эти значения, давайте выведем их на экран. Как мы уже упоминали, мы распаковываем строку, чтобы нам не приходилось использовать индексы для доступа к каждому элементу в этом списке. Например, мы могли бы использовать row[0] для доступа к имени сотрудника. Это допустимо, но может быть сложно понять при чтении большого количества кода. Распаковка списка в именованные переменные упрощает понимание кода в дальнейшем.

Также мы можем использовать функцию записи для генерации содержимого в файл. Это может быть очень полезно, если вы обрабатываете данные в своем скрипте и вам необходимо сохранить их в файл. Возможно, вы хотите импортировать их в электронную таблицу или использовать позже в своем скрипте.

```
>>> hosts = [["workstation.local", "192.168.25.46"], ["webserver.cloud", "10.2.5.6"]]
>>> with open('hosts.csv', 'w') as hosts_csv:
...     writer = csv.writer(hosts_csv)
...     writer.writerows(hosts)
...
>>>
user@ubuntu:~$ cat hosts.csv
workstation.local,192.168.25.46
webserver.cloud,10.2.5.6
user@ubuntu:~$
```

Начнем с сохранения данных, которые мы хотим записать, в список. Мы создали список списков. Это данные, которые мы хотим сохранить в CSV-файле, представляющие имена машин в нашей сети и их IP-адреса.

Хорошо, теперь, когда эти данные готовы к записи, давайте откроем файл в режиме записи. Мы будем использовать блок «with», чтобы не забыть закрыть файл.

Теперь, когда у нас есть открытый для записи файл, давайте вызовем функцию записи модуля CSV с этим файлом в качестве параметра.

Переменная writer является экземпляром класса CSV writer. Есть две функции, которые мы можем использовать: writerow, которая будет записывать по одной строке за раз; и writerows, которая будет записывать все строки вместе. В данном случае у нас уже есть все данные, которые мы хотим записать. Поэтому мы вызовем writerows.

Таким образом, мы записали наши данные в CSV-файл.

Мы увидели, как мы можем читать и записывать CSV-файлы, используя тип данных список на стороне Python. Это работает, когда мы знаем, какими будут поля, но может быть довольно громоздким, когда у нас много столбцов, и нам нужно помнить, какой из них что собой представляет.

Представьте, что ваши списки сотрудников содержат не только имя, номер телефона и должность, но и дату начала работы, имя пользователя, местоположение офиса, отдел, предпочтительные местоимения и так далее. Вскоре станет трудно отслеживать, какой столбец соответствует какой позиции в строке.

В таких случаях обычно в CSV-файлах имена столбцов указываются в первой строке. Мы можем извлечь выгоду из этой дополнительной информации, используя DictReader, немного другой считыватель, который также предоставляется модулем CSV.

```
>>> with open('software.csv') as software:
...     reader = csv.DictReader(software)
...     for row in reader:
...         print(("{} has {} users").format(row["name"], row["users"]))
...
MailTree has 324 users
CalDoor has 22 users
Chatty Chicken has 4 users
>>>
```

Этот считыватель преобразует каждую строку данных в CSV-файле в словарь. Затем мы можем получить доступ к данным, используя имена столбцов вместо позиции в строке.

Итак, здесь мы открываем файл и создаем DictReader для обработки наших CSV-данных, затем, проходя по строкам, мы можем получить доступ к информации в каждой строке, используя ключи так же, как и при доступе к данным в словаре.

```
>>> users = [ {"name": "Sol Mansi", "username": "solm", "department": "IT infrastructure"},
...  {"name": "Lio Nelson", "username": "lion", "department": "User Experience Research"},
...  {"name": "Charlie Grey", "username": "greyc", "department": "Development"}]
>>> keys = ["name", "username", "department"]
>>> with open('by_department.csv', 'w') as by_department:
...     writer = csv.DictWriter(by_department, fieldnames=keys)
...     writer.writeheader()
...     writer.writerows(users)
...
>>>
user@ubuntu:~$ cat by_department.csv
name,username,department
Sol Mansi,solm,IT infrastructure
Lio Nelson,lion,User Experience Research
Charlie Grey,greyc,Development
user@ubuntu:~$
```

Мы можем использовать DictWriter аналогичным образом для генерации CSV-файла из содержимого списка словарей. Это означает, что каждый элемент в списке будет строкой в файле, а значения каждого поля будут взяты из каждого словаря.

Для этого нам также потребуется передать список ключей, которые мы хотим сохранить в файле при создании объекта записи.

Сначала нам нужен список словарей с данными, которые мы хотим сохранить. В этом примере мы хотим сохранить данные о пользователях в нашей компании и отделах, в которых они работают.

Итак, у нас есть список словарей, и каждый из них содержит ключи: имя, имя пользователя и отдел. Теперь мы хотим записать этот HTML-файл, и код будет выглядеть так. Сначала мы определяем список ключей, которые хотим записать в файл, затем открываем файл для записи. Далее мы создали DictWriter, передав ключи, которые мы определили ранее, а затем вызываем два разных метода на объекте writer.

Метод rightheader создаст первую строку CSV-файла на основе переданных нами ключей, а метод rightrows превратит список словарей в строки этого файла.

Регулярные выражения

Теперь давайте поговорим о регулярных выражениях.

Так что же это за выражения и почему они регулярные? Регулярное выражение, также известное как regex или regexp, по сути, представляет собой поисковый запрос для текста, выраженный строковым шаблоном.

Когда вы выполняете поиск по определенному фрагменту текста, все, что соответствует заданному вами шаблону регулярного выражения, возвращается в качестве результата поиска.

Регулярные выражения позволяют ответить на такие вопросы, как: какие четырехбуквенные слова есть в файле? Или сколько различных типов ошибок содержится в этом журнале ошибок?

Другими словами, регулярные выражения позволяют нам искать в тексте строки, соответствующие определенному шаблону. Знание регулярных выражений может быть полезно любому, кому необходимо обрабатывать текст. От ИТ-специалистов до инженеров-программистов, системных администраторов и аналитиков данных, знание регулярных выражений — это удобный инструмент, который полезно иметь под рукой.

```
>>> log = "July 31 07:51:48 mycomputer bad_process[12345]: ERROR Performing package upgrade"
>>> index = log.index("[")
>>> print(log[index+1:index+6])
12345
>>>
```

Например, предположим, что у нас есть журнал. И мы хотим извлечь идентификатор процесса из этой строки, который представляет собой число в квадратных скобках 12345.

В журнале много лишнего текста: строка, которая нам не нужна, например дата, имя компьютера и другая информация.

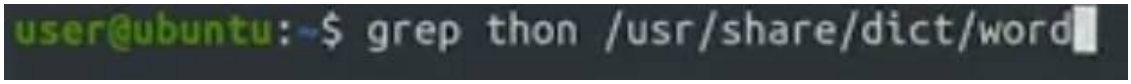
Мы могли бы извлечь идентификатор процесса, используя метод индекса, чтобы найти первую квадратную скобку в строке. Помните, что при доступе к строкам индекс символа — это позиция этого символа в строке, начинающаяся с 0.

Хотя мы получаем текст, который мы хотели, мы могли бы наткнуться на несколько проблем. Одна из проблем в том, что мы не знаем наверняка, какой длины будет строка идентификатора процесса во всех случаях. В этом примере мы видим, что она состоит из пяти символов. Но это может измениться в будущем, если компьютеры перезагружены или количество процессов увеличивается.

```
>>> import re
>>> log = "July 31 07:51:48 mycomputer bad_process[12345]: ERROR Performing package upgrade"
>>> regex = r"\[(\d+)\]"
>>> result = re.search(regex, log)
>>> print(result[1])
12345
>>>
```

Вместо этого мы могли бы использовать регулярное выражение для более надежного извлечения идентификатора процесса. Для этого мы импортируем RE модуль, который позволяет нам использовать функцию поиска регулярных выражений внутри строк.

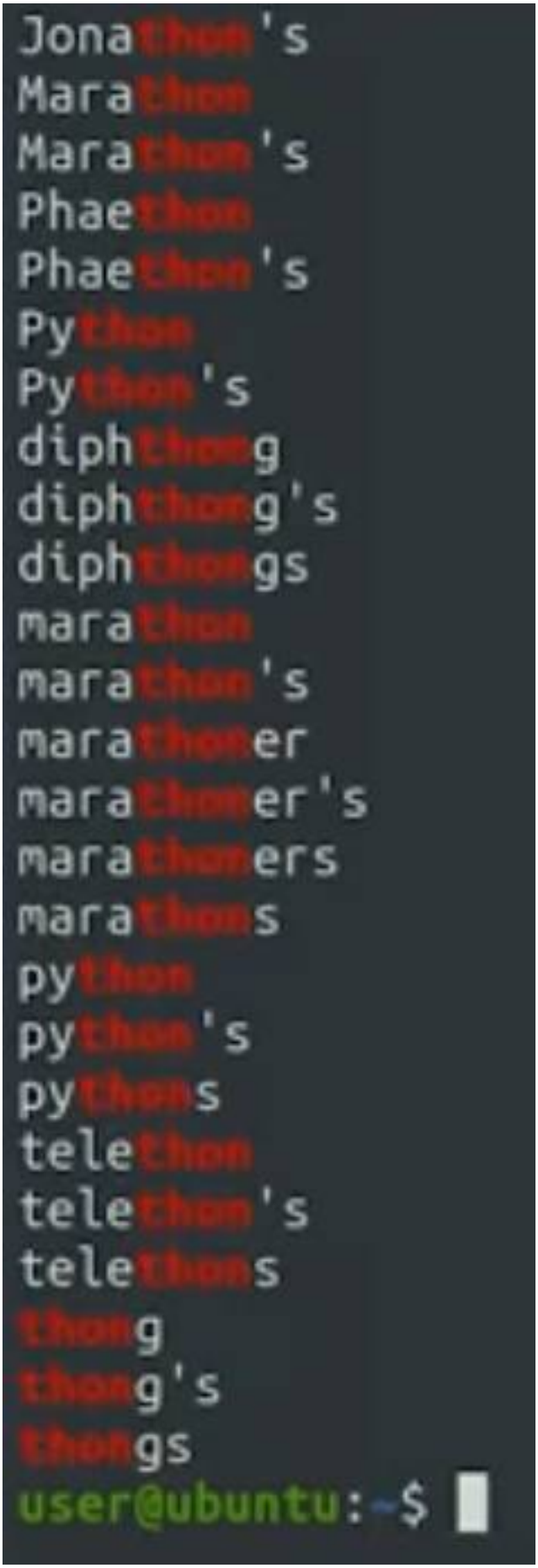
Это регулярное выражение будет работать независимо от того, где отображается наш идентификатор процесса или насколько он длинный или короткий.



Мы также можем использовать регулярные выражения с инструментами командной строки. Например, grep – чрезвычайно мощный инструмент для применения регулярных выражений.

grep работает по распечатке любой строки, соответствующей запросу, который мы ей передаем.

Давайте попробуем использовать grep для поиска слов внутри файла /usr/share/dict/words, представляющий собой файл, который используют некоторые программы проверки орфографии.


Jonathon's
Marathon
Marathon's
Phaethon
Phaethon's
Python
Python's
diphthong
diphthong's
diphthongs
marathon
marathon's
marathoner
marathoner's
marathoners
marathons
python
python's
pythons
telethon
telethon's
telethons
thong
thong's
thongs
user@ubuntu:~$

Этот файл содержит одно слово в строке. Начнем с рассмотрения слов, которые содержат частицу thon.

Все эти слова содержат строку thon где-то внутри них, поэтому они появляются в наших результатах. Мы также видим, что результат выделен, показывая нам соответствующую часть строки.

Стоит отметить, что строка, которую мы передаем grep, чувствительна к регистру.

```
user@ubuntu:~$ grep -i python /usr/share/dict/words
Python
Python's
```

Если мы хотим сопоставить строку независимо от регистра нам придется передать параметр -i команде grep.

Чтобы получить максимальную отдачу от регулярных выражений, нам нужно больше узнать об их синтаксисе, который может быть сложный, но мощный. В частности, нам необходимо знать зарезервированные символы, которые придают дополнительный смысл создаваемым нами шаблонам. Именно эти символы позволяют нам выполнять более сложное сопоставление.

```
user@ubuntu:~$ grep l.rts /usr/share/dict/words
alerts
blurts
flirts
user@ubuntu:~$
```

Например, точка соответствует любому символу. Это означает, что, если мы добавим точку в нашем выражении, эта точка является подстановочным знаком, который можно заменить любым другим символом.

Давайте рассмотрим пример. Посмотрите, как для каждого слова точка в нашем шаблоне была заменена другой буквой.

```
user@ubuntu:~$ grep ^fruit /usr/share/dict/words
fruit
fruit's
fruitcake
fruitcake's
fruitcakes
fruited
fruitful
fruitfully
```

```
user@ubuntu:~$ grep cat$ /usr/share/dict/words
Muscat
bobcat
cat
copycat
ducat
lolcat
```

Другой простой пример специальных символов, которые мы можем использовать в выражениях: каретка или циркумфлекс и знак доллара. Они сообщают, где в строке регулярное выражение должно соответствовать from. Циркумфлекс указывает на начало, а знак доллара указывает на конец строки.

Давайте посмотрим, как мы можем использовать модуль re для базового сопоставления.

```
>>> import re
>>> result = re.search(r"aza", "plaza")
>>> print(result)
<re.Match object; span=(2, 5), match='aza'>
>>>
```

Здесь мы вызываем функцию поиска re модуля с использованием шаблона aza для строки. Затем мы сохранили возвращаемое значение этой функции в переменной результата. r в начале указывает, что это необработанная строка. Это означает, что интерпретатору Python не следует пытаться интерпретировать какие-либо специальные символы, а вместо этого следует просто передать строку в функцию как есть. В этом примере нет особых символов.

Вывод, который мы получаем при вызове print показывает некоторую интересную информацию, например, положение в строке.

```
>>> result = re.search(r"aza", "bazaar")
>>> print(result)
<re.Match object; span=(1, 4), match='aza'>
>>>
```

Давайте попробуем это еще раз с другим словом. В этом случае мы видим, что диапазон другой. Это потому, что подстрока соответствия в другом положении внутри строки.

Как вы думаете, что произойдет, если мы передим строку, которая не соответствует выражению?

```
>>> result = re.search(r"aza", "maze")
>>> print(result)
None
>>>
```

Если выражение не соответствует передаваемой нами строке, в результате мы не получаем ничего. Помните, что None — это особое значение, которое Python использует, чтобы показать, что фактически здесь нет ничего.

```
>>> print(re.search(r"^x", "xenon"))
<re.Match object; span=(0, 1), match='x'>
>>>
```

Давайте попрактикуемся в использовании специальных символов на нескольких примерах.

Здесь мы видим, что x совпадает с началом строки, как мы и ожидали.

```
>>> print(re.search(r"p.ng", "penguin"))
<re.Match object; span=(0, 4), match='peng'>
>>>
```

Что произойдет, если мы используем точку, которая может соответствовать любому символу?

В объекте сопоставления мы видим, что соответствующая строка — peng.

Точка в регулярных выражениях называется подстановочным знаком, потому что она может соответствовать более чем одному символу. Точка — это самый широкий возможный подстановочный знак, поскольку она соответствует абсолютно любому символу. Но что, если нам нужно что-то более строгое, например, проверить, содержит ли ответ пользователя допустимый символ, или найти все имена пользователей в CSV-файле, начинающиеся с гласной?

Для этого нам нужно ограничить наши подстановочные знаки диапазоном символов. Для этой задачи мы используем другую функцию регулярных выражений, называемую классами символов.

```
>>> print(re.search(r"[Pp]ython", "Python"))
<re.Match object; span=(0, 6), match='Python'>
>>>
```

Классы символов записываются внутри квадратных скобок и позволяют нам перечислять символы, которые мы хотим сопоставить, внутри этих скобок. Например, предположим, мы хотим сопоставить слово Python, но разрешить как строчные, так и заглавные буквы p.

```
>>> print(re.search(r"[a-z]way", "The end of the highway"))
<re.Match object; span=(18, 22), match='hway'>
>>>
```

Внутри квадратных скобок мы также можем определить диапазон символов с помощью дефиса. Например, мы могли бы использовать строчные буквы от a до z для обозначения любой строчной буквы.

```
>>> print(re.search(r"[a-z]way", "What a way to go"))
None
>>>
```

Давайте попробуем другую строку. В этом случае мы не нашли совпадения. Это потому, что перед строкой стоит пробел, и это не соответствует определенному нами диапазону.

```
>>> print(re.search("cloud[a-zA-Z0-9]", "cloudy"))
<re.Match object; span=(0, 6), match='cloudy'>
>>> print(re.search("cloud[a-zA-Z0-9]", "cloud9"))
<re.Match object; span=(0, 6), match='cloud9'>
>>>
```

Мы можем определить больше диапазонов, например, от A до Z для всех заглавных букв или от 0 до 9 для всех цифр. Мы можем комбинировать столько диапазонов и символов, сколько захотим. И как видите, мы можем найти совпадение со всем, что определено между квадратными скобками.

```
>>> print(re.search(r"[^a-zA-Z]", "This is a sentence with spaces."))
<re.Match object; span=(4, 5), match=' '>
>>> print(re.search(r"[^a-zA-Z ]", "This is a sentence with spaces."))
<re.Match object; span=(30, 31), match='.'>
>>>
```

Иногда нам может потребоваться найти совпадение с любыми символами, которые не входят в группу. Для этого мы используем циркумфлекс внутри квадратных скобок. Например, давайте создадим шаблон поиска, который ищет любые символы, кроме букв.

Что, если мы добавим пробел в список символов, которые мы не хотим сопоставлять? Вы, возможно, уже догадались. Поскольку мы добавили пробел внутри класса символов, наш пример теперь совпал с последней точкой в предложении, которой нет в списке символов для исключения.

```
>>> print(re.search(r"cat|dog", "I like cats."))
<re.Match object; span=(7, 10), match='cat'>
>>> print(re.search(r"cat|dog", "I like dogs."))
<re.Match object; span=(7, 10), match='dog'>
>>> print(re.search(r"cat|dog", "I like both dogs and cats."))
<re.Match object; span=(12, 15), match='dog'>
>>>
```

Если мы хотим сопоставить либо одно, либо другое выражение, мы можем использовать символ вертикальной черты (|). Это позволяет нам перечислить альтернативные варианты, которые могут быть сопоставлены. Например, у нас может быть выражение, которое соответствует либо слову «кот», либо слову «собака». Таким образом, поскольку строка содержит подстроку «кот», функция поиска может ее найти.

Давайте попробуем предложение, содержащее как собак, так и кошек. В этой строке у нас есть два возможных совпадения для нашего поиска. Но мы получаем только первое. Так работает функция поиска.

```
>>> print(re.findall(r"cat|dog", "I like both dogs and cats."))
['dog', 'cat']
>>>
```

Если мы хотим получить все возможные совпадения, мы можем сделать это с помощью функции findall, которая также предоставляется модулем re.

До сих пор мы рассматривали, как сопоставить один конкретный символ, группу символов или даже любой символ. Теперь давайте посмотрим, как сопоставить эти символы несколько раз.

```
>>> print(re.search(r"Py.*n", "Pygmalion"))
<re.Match object; span=(0, 9), match='Pygmalion'>
>>> print(re.search(r"Py.*n", "Python Programming"))
<re.Match object; span=(0, 17), match='Python Programmin'>
>>>
```

Довольно часто встречаются выражения, которые включают точку, за которой следует звездочка. Это означает, что оно соответствует любому символу, повторяющемуся как можно больше раз, включая ноль.

Это регулярное выражение можно прочитать как «сопоставление с пи, за которым следует любое количество других символов, а затем n».

Давайте попробуем другую строку, Python Programming. Как вы думаете, чему будет соответствовать этот шаблон? Помните, что звездочка принимает как можно больше символов. В терминах программирования мы говорим, что такое поведение является жадным.

```
>>> print(re.search(r"Py[a-z]*n", "Python Programming"))
<re.Match object; span=(0, 6), match='Python'>
>>> print(re.search(r"Py[a-z]*n", "Pyn"))
<re.Match object; span=(0, 3), match='Pyn'>
>>>
```

Если бы мы хотели, чтобы наши шаблоны соответствовали только буквам, нам следовало бы использовать класс символов, как показано.

Помните, как мы говорили, что ноль повторений также возможен? Это позволит строке Pyn соответствовать нашему шаблону.

Давайте рассмотрим еще квалификаторы повторения: плюс и вопросительный знак, которые могут помочь нам создавать более сложные выражения.

```
>>> print(re.search(r"o+l+", "goldfish"))
<re.Match object; span=(1, 3), match='ol'>
>>> print(re.search(r"o+l+", "woolly"))
<re.Match object; span=(1, 5), match='ooll'>
>>> print(re.search(r"o+l+", "boil"))
None
>>>
```

Символ плюс соответствует одному или нескольким вхождениям символа, который стоит перед ним.

```
>>> print(re.search(r"p?each", "To each their own"))
<re.Match object; span=(3, 7), match='each'>
>>>
```

Символ вопросительного знака — это еще один множитель. Он означает либо ноль, либо одно вхождение символа перед ним.

Мы уже рассмотрели множество специальных символов, которые можно использовать в регулярных выражениях для сопоставления различных типов строк: точка, звездочка, плюс, вопросительный знак, циркумфлекс, знак доллара и квадратные скобки.

Но что делать, если шаблон должен соответствовать одному из специальных символов? Допустим, мы хотим проверить, содержит ли определенная строка точку. Если мы просто поставим точку, шаблон будет соответствовать любому символу.

```
>>> print(re.search(r".com", "welcome"))
<re.Match object; span=(2, 6), match='lcom'>
>>> print(re.search(r"\.com", "welcome"))
None
>>>
```

Для сопоставления точки нам необходимо использовать экранирующий символ, который в случае регулярных выражений представляет собой обратную косую черту.

Экранировав точку, шаблон перестал соответствовать слову welcome, а поскольку в строке нет .com, он вернул значение none.

```
>>> print(re.search(r"\.com", "mydomain.com"))
<re.Match object; span=(8, 12), match='.com'>
>>>
```

Давайте попробуем что-нибудь, что должно действительно совпадать. Отлично. Добавив обратную косую черту, мы добились того, что это правильно соответствует тому, что мы хотели.

Мы можем использовать обратную косую черту для экранирования любых специальных символов, включая те, о которых мы еще даже не говорили.

На что следует обратить внимание. С обратными косыми чертами может возникнуть путаница, поскольку они также используются для определения некоторых специальных строковых символов. Мы, например, говорили, что \n это последовательность, используемая в Python для обозначения новой строки, а \t делает то же самое для табуляции.

Когда мы видим шаблон, включающий обратную косую черту, это может быть экранирование специального символа регулярного выражения или специального строкового символа.

Использование необработанных строк, как мы это делали, помогает избежать некоторой возможной путаницы, потому что специальные символы не будут интерпретироваться при генерации строки. Они будут интерпретироваться только при разборе регулярного выражения.

Помимо этого, Python также использует обратную косую черту для нескольких специальных последовательностей, которые мы можем использовать для представления предопределенных наборов символов.

```
>>> print(re.search(r"\w*", "This is an example"))
<re.Match object; span=(0, 4), match='This'>
>>> print(re.search(r"\w*", "And_this_is_another"))
<re.Match object; span=(0, 19), match='And_this_is_another'>
>>>
```

Например, \w соответствует любому буквенно-цифровому символу, включая буквы, цифры и подчеркивания.

В первом примере наш шаблон соответствует первым четырем буквам до пробела, потому что пробелы не входят в этот набор символов. Во втором примере он соответствует всей строке.

Также есть \d для соответствия цифрам, \s для соответствия символам пробел, табуляция или новая строка, \b для границ слов и еще несколько других.

Мы изучили множество правил для использования регулярных выражений в Python. Вооружившись всеми этими знаниями, мы можем начать комбинировать эти специальные символы для создания шаблонов, соответствующих нужному тексту.

Например, предположим, у вас есть список всех стран мира, и вы хотите проверить, какие из этих названий начинаются и заканчиваются на букву «a». Как будет выглядеть шаблон?

```
>>> print(re.search(r"A.*a", "Argentina"))
<re.Match object; span=(0, 9), match='Argentina'>
>>> print(re.search(r"A.*a", "Azerbaijan"))
<re.Match object; span=(0, 9), match='Azerbaija'>
>>> print(re.search(r"^A.*a$", "Azerbaijan"))
None
>>> print(re.search(r"^A.*a$", "Australia"))
<re.Match object; span=(0, 9), match='Australia'>
>>>
```

Возможно, что-то вроде A.*a. Давайте попробуем. Отлично. Кажется, это работает.

Давайте попробуем с другой страной. Азербайджан. Возможно, это удивительно, но это произошло потому, что мы не указали, что хотим, чтобы наши шаблоны соответствовали всей строке. Поэтому, хотя Азербайджан не заканчивается на букву «a», в его названии есть буква «a». Таким образом, он соответствует нашему шаблону.

Нам нужно сделать наши шаблоны более строгими, добавив символы начала и конца строки. Добавив знак доллара к нашему шаблону, мы ясно дали понять, что хотим сопоставлять только строки, которые начинаются и заканчиваются на букву «a». Поэтому Азербайджан больше не соответствует шаблону.

Давайте проверим, работает ли это по-прежнему для страны, которая должна соответствовать условию, например, Австралия.

```
>>> pattern = r"^[a-zA-Z_][a-zA-Z0-9_]*$"
>>> print(re.search(pattern, "_this_is_a_valid_variable_name"))
<re.Match object; span=(0, 30), match='_this_is_a_valid_variable_name'>
>>> print(re.search(pattern, "this isn't a valid variable"))
None
>>> print(re.search(pattern, "my_variable1"))
<re.Match object; span=(0, 12), match='my_variable1'>
>>> print(re.search(pattern, "2my_variable1"))
None
>>>
```

Используя регулярные выражения, мы также можем построить шаблон, который будет проверять, является ли строка допустимым именем переменной в Python. Помните, какие правила? Она может содержать любое количество букв, цифр или подчеркиваний, но не может начинаться с цифры.

Итак, как вы думаете, как будет выглядеть шаблон проверки?

Мы начнем с циркумфлекса, чтобы указать, что мы хотим начать с начала, и далее символьный класс со всеми строчными и прописными буквами плюс подчеркивание. Остальной

ная часть переменной может содержать столько чисел, букв или подчеркиваний, сколько нам нужно. Поэтому нам нужен еще один символьный класс, на этот раз содержащий цифры со звездочкой в конце.

И последнее: мы хотим, чтобы это был конец строки, которую мы сопоставляем. В противном случае мы могли бы сопоставить что-то, что может быть переменной, но также содержит дополнительные элементы после себя. Таким образом, мы получаем знак доллара.

Теперь применим этот шаблон к нескольким примерам.

Правильно, мы можем использовать подчеркивания в любом месте строки, это допустимое имя переменной. Оно соответствует нашему шаблону проверки, потому что мы включили в него подчеркивания.

Давайте попробуем что-нибудь немного другое. Как только мы используем пробел, это перестает быть допустимым именем переменной. Оно не соответствует шаблону, потому что пробелы не входят в число возможных символов.

Давайте проверим переменную с числом. Конечно, мы можем использовать числа внутри имени переменной, но что, если мы начнем с числа? Переменная с числом в начале не является допустимым именем переменной.

До сих пор мы использовали функцию поиска, чтобы проверить, соответствует ли строка определенному шаблону. Но единственное, что мы делали с результатом, это выводили его на экран. Но в большинстве случаев мы хотим взять соответствующую информацию и использовать ее для чего-то другого. Например, мы можем захотеть извлечь имя хоста или идентификатор процесса из строки журнала и использовать это значение для другой операции. Для этого нам нужно использовать концепцию регулярных выражений, называемую захватывающими группами. Захватывающие группы — это части шаблона, заключенные в скобки.

```
>>> result = re.search(r"^(\w*), (\w*)$", "Lovelace, Ada")
>>> print(result)
<re.Match object; span=(0, 13), match='Lovelace, Ada'>
>>> print(result.groups())
('Lovelace', 'Ada')
>>> print(result[0])
Lovelace, Ada
>>> print(result[1])
Lovelace
>>> print(result[2])
Ada
>>> "{} {}".format(result[2], result[1])
'Ada Lovelace'
>>>
```

Допустим, у нас есть список имен людей. Эти имена хранятся в формате фамилия, запятая, имя. Мы хотим перевернуть это и создать строку, которая начинается с имени, за которым следует фамилия. Мы можем сделать это с помощью регулярного выражения с захватывающими группами.

Сначала мы создадим шаблон соответствия, который соответствует группе букв, за которой следует запятая, пробел, а затем еще одна группа букв.

Давайте посмотрим на вывод метода groups. Поскольку мы определили две отдельные группы, метод group возвращает кортеж из двух элементов. Мы также можем использовать

индексацию для доступа к этим группам. Итак, мы можем сформировать нужное нам имя, используя эти индексы.

```
>>> def rearrange_name(name):
...     result = re.search(r"^(\w*), (\w*)$", name)
...     if result is None:
...             return name
...     return "{} {}".format(result[2], result[1])
...
>>> rearrange_name("Lovelace, Ada")
'Ada Lovelace'
>>> rearrange_name("Ritchie, Dennis")
'Dennis Ritchie'
>>>
```

Давайте поместим это в функцию, которая будет выполнять перестановку. Начнем с определения функции, которая принимает имя в качестве параметра. Теперь мы будем искать строку с тем же шаблоном, который мы видели раньше. Если результат равен «нет», это означает, что он не соответствует ожидаемому формату. В этом случае функция просто вернет имя как есть.

Давайте вернем переставленную версию этой строки. Отлично, мы определили нашу функцию, давайте протестируем ее с несколькими именами. Здорово, кажется, это работает.

```
>>> rearrange_name("Hopper, Grace M.")
'Hopper, Grace M.'
>>>
```

Но что, если мы дадим ей что-то немного более сложное? Теперь регулярное выражение не сработало, потому что мы использовали символ \w, который соответствует только буквам. Поэтому оно не распознало среднюю начальную букву как часть заданного имени. Можете ли вы придумать, как это исправить? Нам нужно добавить дополнительные символы, которые мы хотим разрешить в именах.

```
>>> def rearrange_name(name):
...     result = re.search(r"^([\w .-]*), ([\w .-]*)$", name)
...     if result == None:
...         return name
...     return "{} {}".format(result[2], result[1])
...
>>> rearrange_name("Hopper, Grace M.")
'Grace M. Hopper'
>>>
```

В этом примере мы хотим добавить пробелы и точки. Для других имен мы также можем включить тире.

Как видите, добавив дополнительные символы между квадратными скобками, мы обработали имя, которое стало немного сложнее.

До сих пор мы использовали квалификаторы повторения «*», «+» и «?». А что, если нам нужен шаблон, который повторяется определенное количество раз? Это может произойти, если мы обрабатываем строку, которая, как нам известно, содержит определенные данные в столбце, или нам нужна строка определенной длины.

В таких случаях мы бы вручную записывали один и тот же шаблон столько раз, сколько необходимо. Но это было бы сложно читать и сложно поддерживать. Именно поэтому Python также предлагает числовые квалификаторы повторения.

```
>>> print(re.search(r"[a-zA-Z]{5}", "a ghost"))
<re.Match object; span=(2, 7), match='ghost'>
>>> print(re.search(r"[a-zA-Z]{5}", "a scary ghost appeared"))
<re.Match object; span=(2, 7), match='scary'>
>>> print(re.findall(r"[a-zA-Z]{5}", "a scary ghost appeared"))
['scary', 'ghost', 'appea']
>>> print(re.findall(r"\b[a-zA-Z]{5}\b", "A scary ghost appeared"))
['scary', 'ghost']
>>>
```

Они записываются в фигурные скобки и могут представлять собой одно или два числа, указывающие диапазон. Например, чтобы найти любую строку, состоящую ровно из пяти букв, мы можем использовать выражение, подобное этому.

В данном случае мы ищем буквы, которые повторяются пять раз, а у слова ghost пять букв, поэтому оно соответствует нашему шаблону.

В следующей строке у нас на самом деле больше совпадений для нашего поиска, но мы получаем только первое. Помните, что мы можем сделать, чтобы найти больше совпадений? Правильно, использовать функцию findall.

Что, если бы мы хотели найти все слова, которые состоят ровно из пяти букв? Мы можем сделать это с помощью \b, что соответствует ограничениям по количеству слов в начале и конце шаблона, чтобы указать, что мы хотим найти целые слова.

```
>>> print(re.findall(r"\w{5,10}", "I really like strawberries"))
['really', 'strawberri']
>>> print(re.findall(r"\w{5,}", "I really like strawberries"))
['really', 'strawberries']
>>> print(re.search(r"s\w{,20}", "I really like strawberries"))
<re.Match object; span=(14, 26), match='strawberries'>
>>>
```

Мы говорили, что мы также можем использовать два числа в диапазоне. Например, если мы хотим найти диапазон от пяти до десяти букв или цифр, мы можем использовать выражение, подобное этому.

Эти диапазоны также могут быть открытыми. Число, за которым следует запятая, означает как минимум повторений без верхней границы, ограниченной только максимальным количеством повторений в исходном тексте.

Теперь, в нашем последнем примере, запятая, за которой следует число, означает от нуля до указанного количества повторений.

```
>>> import re
>>> log = "July 31 07:51:48 mycomputer bad_process[12345]: ERROR Performing package upgrade"
>>> regex = r"\[(\d+)\]"
>>> result = re.search(regex, log)
>>> print(result[1])
12345
>>>
```

Давайте рассмотрим пример. Давайте пройдемся по нему шаг за шагом.

Первый символ шаблона обратная косая черта, которая используется в качестве escape-символа. Это означает, что следующий символ, который здесь представляет собой квадратную скобку, трактуется буквально.

После квадратной скобки идет символ скобки. Поскольку он не экранирован, мы знаем, что он будет использоваться в качестве группы захвата.

Из нашего обсуждения особых символов и квалификаторов повторения, мы знаем, что это выражение будет соответствовать одному или нескольким числовых символов.

После закрывающих скобок захватывающей группы, у нас есть символ закрывающей квадратной скобки, которому также предшествует escape-символ.

После вызова функции поиска мы знаем, что, поскольку мы фиксируем группы в выражении, мы можем получить доступ к данным результата с помощью индекса.

```
>>> result = re.search(regex, "99 elephants in a [cage]")
>>> print(result[1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not subscriptable
>>>
```

Но что, если наша строка на самом деле не блок цифр в квадратных скобках? Блин, ошибка, что случилось? Мы попытались получить доступ к индексу 1 переменной, которой нет.

Так что же нам делать? У нас должна быть функция, которая извлекает идентификатор процесса или PID, если это возможно, а в противном случае делает что-то еще.

```
>>> def extract_pid(log_line):
...     regex = r"\[(\d+)\]"
...     result = re.search(regex, log_line)
...     if result is None:
...             return ""
...     return result[1]
...
```

Мы начнем с определения функции. И мы будем использовать то же самое регулярное выражение, как и раньше, и мы сохраним результат функции поиска в значении результата.

Теперь мы знаем, что, если поиск не будет успешным, результат будет нулевым.

Поэтому, если мы не сможем найти PID, мы вернем пустую строку.

До сих пор мы использовали две функции из модуля «re»: search и findall. На самом деле есть еще несколько функций, которые могут быть очень полезны в зависимости от того, что мы пытаемся сделать. Одна из этих функций называется split. Она работает аналогично функции split, которую мы использовали ранее со строками. Но здесь в качестве разделителя вы можете использовать любое регулярное выражение.

```
>>> re.split(r"[.?!]", "One sentence. Another one? And the last one!")
['One sentence', ' Another one', ' And the last one', '']
>>> re.split(r"([.?!])", "One sentence. Another one? And the last one!")
['One sentence', '.', ' Another one', '?', ' And the last one', '!', '']
>>>
```

Например, мы можем захотеть разделить текст на отдельные предложения. Для этого нам нужно проверить не только точки, но и вопросительные или восклицательные знаки, поскольку они также являются допустимыми окончаниями предложений.

Обратите внимание, что мы не экранируем символы, которые написали внутри квадратных скобок. Это потому, что все, что находится внутри квадратных скобок, воспринимается как буквальный символ, а не как его особое значение. Также обратите внимание, что в результирующем списке отсутствуют символы.

Если мы хотим, чтобы наш список для разделения содержал элементы, которые мы используем для разделения значений, мы можем использовать скобки. Это позволит нам получить как предложения, так и знаки обозначения в качестве элементов списка.

```
>>> re.sub(r"[\w.%+-]+@[\w.-]+", "[REDACTED]", "Received an email for go_nuts95@my.example.com")
'Received an email for [REDACTED]'
>>>
```

Еще одна интересная функция, предоставляемая модулем «re», называется sub. Она используется для создания новых строк путем замены всей или части строки другой строкой, аналогично методу замены строк, но с использованием регулярных выражений как для сопоставления, так и для замены.

В этом примере у нас есть журналы в системе, которые содержат адреса электронной почты пользователей, и мы хотим анонимизировать данные, удалив все адреса. Мы можем это сделать с помощью регулярного выражения.

Выражение, которое мы используем для идентификации адресов электронной почты, состоит из двух частей: части до знака «@» и части после него.

Обратите внимание на часть, которая идет перед знаком «@». Мы включаем буквенно-цифровые символы, представленные \w, включая буквы, цифры и знак подчеркивания, а также точку, знак процента, плюс и тире. После знака «@» мы разрешаем только буквенно-цифровые символы точку и тире. Это будет соответствовать всем адресам электронной почты, а также некоторым строкам, которые на самом деле не являются действительными адресами электронной почты, например, адресу, с двумя точками. Здесь мы хотим перестраховаться.

Поэтому мы будем удалять все, что выглядит как адрес. Если бы мы хотели проверить, является ли адрес настоящим адресом электронной почты, нам нужно было бы быть намного строже.

Давайте теперь рассмотрим пример использования подпрограммы, где мы используем регулярные выражения для замены.

```
>>> re.sub(r"^([\w .-]*), ([\w .-]*)$", r"\2 \1", "Lovelace, Ada")
'Ada Lovelace'
>>>
```

Здесь в первом параметре у нас есть выражение, содержащее две группы, которые мы хотим сопоставить: одну до запятой и одну после запятой. И мы хотим использовать второй параметр для замены совпадающей строки.

Мы используем \2 для обозначения второй группы, за которой следует пробел, и \1 для обозначения первой группы. При обращении к группам обратная косая черта, за которой следует число, указывает на соответствующую группу. Это общее обозначение для регулярных выражений, и оно используется многими инструментами, поддерживающими регулярные выражения, а не только Python.

Мы также можем использовать их для сопоставления шаблонов, которые повторяются и используют группы в качестве обратных ссылок.

Управление данными и процессами

Мы уже говорили о чтении и записи файлов. Использование файлов для хранения информации с последующей обработкой данных с помощью скрипта — отличный способ автоматизации.

Однако иногда нам нужно взаимодействовать с пользователем и запрашивать у него информацию, которую невозможно хранить в файле. Для этого в Python предусмотрена функция «input». Она позволяет запросить у пользователя определенное значение, которое затем можно использовать в скрипте.

```
user@ubuntu:~$ cat hello.py
#!/usr/bin/env python3

name = input("Please enter your name: ")
print("Hello, " + name)
user@ubuntu:~$ ./hello.py
Please enter your name: Roger
Hello, Roger
user@ubuntu:~$
```

Давайте посмотрим, как это работает. Вот скрипт под названием «hello.py». Он очень простой: запрашивает у пользователя имя, а затем выводит приветствие, включающее это имя.

Запустим скрипт и посмотрим на него в действии. Важно помнить: функция «input» всегда возвращает строку. Если нам нужны данные другого типа — например, число или дата, — строку необходимо преобразовать в нужный формат.

```
def to_seconds(hours, minutes, seconds):
    return hours*3600+minutes*60+seconds

print("Welcome to this time converter")

cont = "y"
while(cont.lower() == "y"):
    hours = int(input("Enter the number of hours: "))
    minutes = int(input("Enter the number of minutes: "))
    seconds = int(input("Enter the number of seconds: "))

    print("That's {} seconds".format(to_seconds(hours, minutes, seconds)))
    print()
    cont = input("Do you want to do another conversion? [y to continue] ")

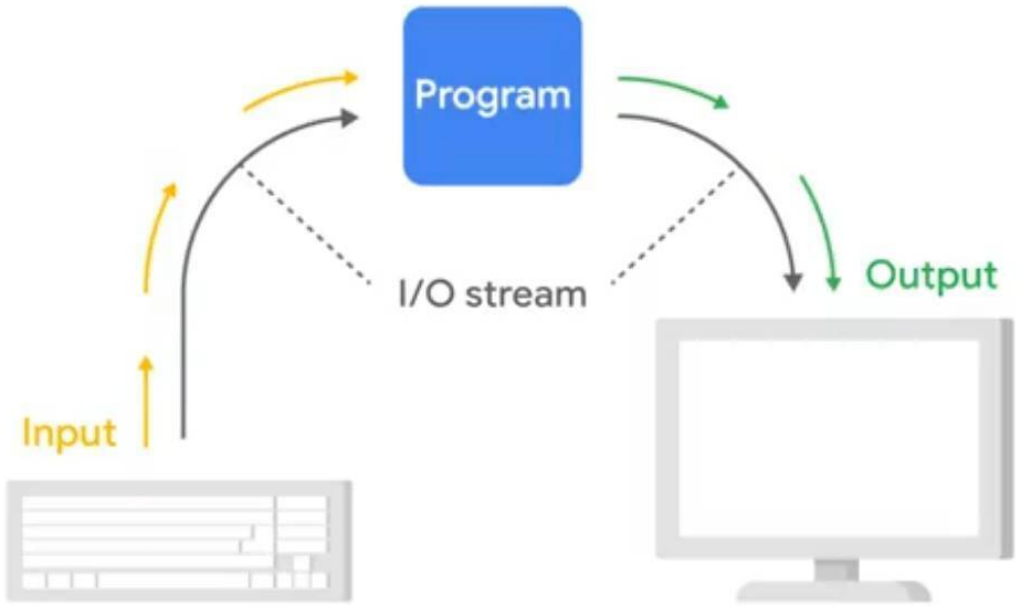
print("Good bye!")
```

Рассмотрим другой пример. Этот скрипт выполняет несколько задач, поэтому разберем его по частям.

Сначала мы определяем функцию, которая переводит часы, минуты и секунды в общее количество секунд. Основная часть программы начинается с вывода приветствия и перехода к циклу «while». Обратите внимание: мы заранее инициализируем переменную «cont», которая будет использоваться для проверки того, хочет ли пользователь продолжить работу.

В теле цикла «while» мы просим пользователя ввести количество часов, минут и секунд для преобразования. Поскольку нашей функции перевода в секунды требуются целые числа, мы преобразуем значение, полученное от «input», с помощью функции «int». Получив эти три значения, мы вызываем функцию и выводим результат. Затем мы спрашиваем пользователя, хочет ли он выполнить еще одно преобразование.

Мы уже рассмотрели несколько способов передачи данных в скрипты и получения информации из них. Мы умеем читать файлы и записывать в них данные, а также принимать ввод с клавиатуры и выводить информацию на экран. Но что именно происходит «за кулисами» в этих случаях? Как программа на Python взаимодействует с экраном и клавиатурой?



Для этого используются потоки ввода-вывода. Потоки ввода-вывода — это основной механизм выполнения операций ввода и вывода в программах. Их можно представить как каналы связи между программой и источниками ввода (например, клавиатурой) или устройствами вывода (например, экраном).

Когда мы выполняем системную команду в терминале, также используются потоки ввода-вывода, связывающие эту команду с вводом и выводом терминала. Благодаря этому мы можем видеть результаты выполнения команды или вводить данные в интерактивном режиме, если это предусмотрено логикой программы.

Мы называем их «потоками», поскольку данные передаются непрерывно: программа может считывать входные данные и формировать вывод столько раз, сколько потребуется для выполнения поставленной задачи.

Что же это означает на практике? Большинство операционных систем по умолчанию предоставляют три различных потока ввода-вывода, каждый из которых имеет свое назначение.



Стандартный поток ввода (STDIN) — это канал между программой и источником входных данных; обычно это текстовые данные, поступающие с клавиатуры. Когда мы используем функцию «input» в скрипте на Python для получения данных от пользователя, мы задействуем именно поток STDIN.

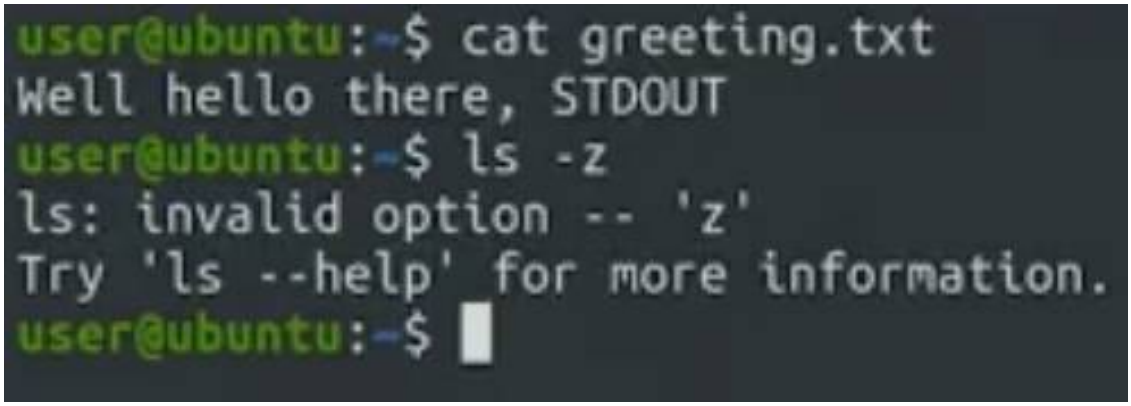
Стандартный поток вывода (STDOUT) — это канал между программой и устройством вывода, например дисплеем. Как правило, данные из STDOUT отображаются в виде текста в терминале — именно это происходит, когда мы используем функцию «print» для вывода информации на экран.

Последний тип стандартного потока ввода-вывода называется «стандартный поток ошибок» (STDERR). Поток STDERR, подобно стандартному потоку вывода, отображает информацию на экране, но используется специально для вывода сообщений об ошибках и диагностических данных программы. Если вам когда-либо доводилось запускать код на Python и получать сообщение об ошибке, то, скорее всего, оно выводилось через стандартный поток ошибок.

```
user@ubuntu:~$ cat streams.py
#!/usr/bin/env python3

data = input("This will come from STDIN: ")
print("Now we write it to STDOUT: " + data)
print("Now we generate an error to STDERR: " + data + 1)
user@ubuntu:~$
```

Давайте рассмотрим это на конкретном примере. Наш скрипт состоит из трех строк, и каждая из них взаимодействует с отдельным потоком. В первой строке мы считываем данные из стандартного потока ввода, во второй — записываем их в стандартный поток вывода. В последней строке мы намеренно вызываем ошибку, пытаясь объединить строку с целым числом. Эта ошибка будет выведена в стандартный поток ошибок.



Эти потоки ввода-вывода используются не только в программах на Python. Например, мы можем использовать системную команду «cat» для вывода содержимого файла. В этом случае содержимое файла отображается в терминале через стандартный поток вывода. А если такая команда вызывает ошибку, то сообщение об ошибке направляется в стандартный поток ошибок. Например, если использовать неподдерживаемый флаг с командой «ls» (которая обычно показывает содержимое каталогов), мы получим сообщение об ошибке.

Может показаться, что стандартный поток вывода и стандартный поток ошибок здесь ничем не отличаются. Это ощущение возникает потому, что оба потока выводят данные на экран, но на самом деле они существенно различаются. Подробнее об этих различиях мы поговорим позже — пока сохраним интригу.

Главное, что стоит запомнить на данном этапе: потоки ввода-вывода — это механизмы, с помощью которых программы получают и передают информацию.

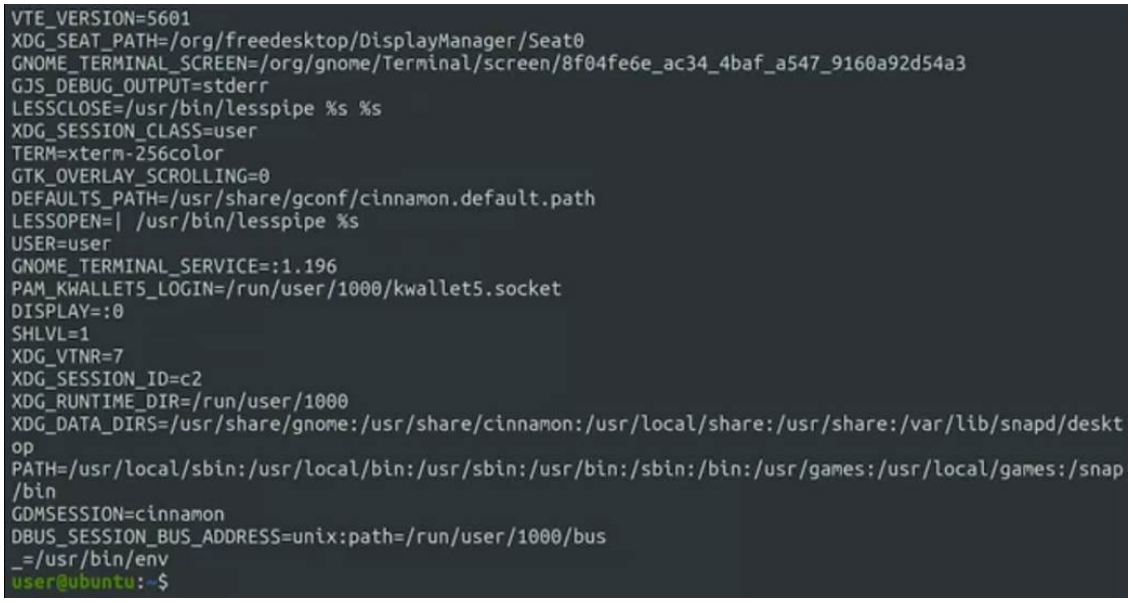
Когда мы запускаем терминал на компьютере с Linux — будь то локальная или удаленная машина, — программа, которая считывает и выполняет наши команды, называется «оболочкой» (shell). Оболочка представляет собой интерфейс командной строки для взаимодействия с операционной системой.

Самая распространенная оболочка в Linux — это bash. Другие популярные варианты — Zsh и Fish. Хотя принципы их работы схожи, далее под термином «оболочка Linux» мы будем подразумевать именно bash.

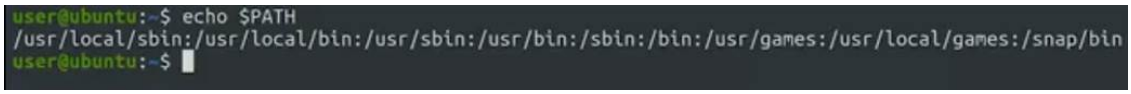
Программы на Python выполняются в среде командной строки, предоставляемой оболочкой. Переменные, заданные в этой среде — их называют (как нетрудно догадаться) переменными окружения, — служат еще одним источником информации, которую можно использовать в скриптах.

Умение работать с переменными окружения и изменять их очень полезно, когда нужно быстро скорректировать поведение программы. Обычно для этого достаточно внести небольшие изменения в среду, в которой выполняется программа.

Проверить эти переменные можно прямо в командной строке с помощью команды «env».



Давайте посмотрим. Здесь представлено множество различных переменных, но для чего они нужны? Все зависит от конкретной переменной. Некоторые из них важнее других. Например, переменная «PATH» имеет огромное значение.



Давайте выведем ее содержимое с помощью команды «echo». Напомню: «echo» — это команда, используемая в оболочке Linux для вывода текста. Чтобы получить доступ к значению переменной в оболочке, перед ее именем нужно поставить знак доллара.

Вот мы и вывели содержимое переменной `PATH`. Оболочка использует эту переменную окружения, чтобы определить, где искать исполняемые файлы, если мы запускаем их без указания конкретного пути к директории. Именно в перечисленных здесь директориях оболочка будет искать программы.

При запуске программы Python 3 оболочка последовательно проверяет каждую директорию, указанную в переменной «PATH». Обнаружив программу с именем «python3», она запускает ее.

Как мы уже говорили, значения этих переменных можно считать с помощью Python. Давайте проверим это на практике, написав скрипт.

```
user@ubuntu:~$ cat variables.py
#!/usr/bin/env python3

import os

print("HOME: " + os.environ.get("HOME", ""))
print("SHELL: " + os.environ.get("SHELL", ""))
print("FRUIT: " + os.environ.get("FRUIT", ""))
user@ubuntu:~$ ./variables.py
HOME: /home/user
SHELL: /bin/bash
FRUIT:
user@ubuntu:~$ export FRUIT=Pineapple
user@ubuntu:~$ ./variables.py
HOME: /home/user
SHELL: /bin/bash
FRUIT: Pineapple
user@ubuntu:~$
```

Для доступа к переменным окружения мы используем словарь «environ» из модуля «os». В данном случае мы применим метод «get».

Метод «get» отчасти напоминает привычный нам способ обращения к элементам словаря, но есть и отличие: реакция на отсутствие ключа. Если мы пытаемся получить значение по ключу (например, «os.environ['FRUIT']») и такого ключа в словаре нет, возникает ошибка. Но метод «get» позволяет указать значение, которое будет возвращено, если искомый ключ отсутствует. Иными словами, с помощью «get» можно задать значение по умолчанию для случаев, когда нужного ключа в словаре нет.

Здесь мы просим Python попытаться получить значение, связанное с ключом, но если ключ не определен — вернуть пустую строку. Здесь мы проделываем это для трех переменных: «HOME», «SHELL» и «FRUIT».

Мы определяем переменную, просто присваивая ей значение через знак равенства (без пробелов). Кроме того, ключевое слово «export» сообщает оболочке (shell), что установленное нами значение должно быть доступно для всех вызываемых команд.

Еще один распространенный способ передачи информации программам — использование аргументов командной строки. Это параметры, которые передаются программе при ее запуске. Очень часто скрипты настраивают так, чтобы они получали определенные значения именно через аргументы командной строки. Это позволяет сделать код скрипта универсальным и запускать его автоматически, не требуя интерактивного ввода от пользователя. Благодаря этому такие аргументы крайне полезны при решении задач системного администрирования: мы можем указать нужные данные еще до запуска программы. Это открывает широкие возможности для автоматизации — и с этим трудно поспорить.

```
user@ubuntu:~$ cat parameters.py
#!/usr/bin/env python3
import sys
print(sys.argv)
user@ubuntu:~$ ./parameters.py
['./parameters.py']
user@ubuntu:~$ ./parameters.py  one two three
['./parameters.py', 'one', 'two', 'three']
user@ubuntu:~$
```

Получить доступ к этим значениям можно с помощью переменной «argv» из модуля «sys».

Здесь мы просто импортируем модуль «sys» и выводим список «sys.argv». Теперь посмотрим, что произойдет при вызове программы. В данном случае мы запустили скрипт без каких-либо параметров. Список содержит всего один элемент — имя только что запущенной программы.

Попробуем передать несколько параметров. Теперь мы видим, что каждый переданный параметр стал отдельным элементом списка.

```
user@ubuntu:~$ wc variables.py
  7  19 174 variables.py
user@ubuntu:~$ echo $?
0
user@ubuntu:~$ wc notpresent.py
wc: notpresent.py: No such file or directory
user@ubuntu:~$ echo $?
1
```

И наконец, перейдем к понятию статуса завершения (или кода возврата) — это еще один канал обмена информацией между командной оболочкой и выполняемыми в ней программами.

Статус завершения — это значение, которое программа возвращает оболочке. Во всех Unix-подобных операционных системах статус завершения процесса равен нулю, если выполнение прошло успешно, и отличен от нуля в случае ошибки.

Фактическое возвращаемое значение предоставляет дополнительную информацию о том, с какой именно ошибкой столкнулась программа. Сведения о том, успешно ли завершилась команда, полезны для программы, которая эту команду вызывает; например, она может использовать эту информацию для повторного запуска команды в случае сбоя.

Для проверки кода завершения программы можно использовать специальную переменную, позволяющую узнать статус выхода последней выполненной команды. Это переменная, обозначаемая знаком вопроса; чтобы увидеть ее содержимое, мы используем конструкцию «знак доллара — знак вопроса» («$?»).

Давайте проверим это на примере команды «wc», которая подсчитывает количество строк, слов и символов в файле. Сначала передадим ей наш скрипт «variables.py» и проверим код завершения. Итак, мы выполнили команду «wc», и она вывела количество строк, слов и символов для нашего Python-скрипта. Затем мы вывели содержимое переменной «$?» и увидели, что код завершения равен нулю. Это произошло потому, что команда «wc» выполнилась успешно.

В следующем примере «wc» не смогла обработать переданный файл, так как его не существует. Команда выдала сообщение об ошибке, и при выводе содержимого переменной «$?» мы видим, что она завершилась с кодом 1.

Так обстоит дело с системными командами, но как насчет скриптов на Python? Когда скрипт Python завершается успешно, он возвращает код 0. Если же он завершается с ошибкой (например, «TypeError» или «ValueError»), возвращаемый код отличен от нуля. Мы можем настроить скрипт так, чтобы он возвращал любой нужный нам код.

```
import os
import sys

filename=sys.argv[1]

if not os.path.exists(filename):
    with open(filename, "w") as f:
        f.write("New file created\n")
else:
    print("Error, the file {} already exists!".format(filename))
    sys.exit(1)
```

Давайте рассмотрим пример. Этот скрипт принимает имя файла в качестве аргумента командной строки. Сначала он проверяет, существует ли такой файл. Если файла нет, скрипт создает его и записывает в него строку. Если же файл существует, скрипт выводит сообщение об ошибке и завершается с кодом 1.

```
user@ubuntu:~$ ./create_file.py example
user@ubuntu:~$ echo $?
0
user@ubuntu:~$ cat example
New file created
user@ubuntu:~$ ./create_file.py example
Error, the file example already exists!
user@ubuntu:~$ echo $?
1
user@ubuntu:~$
```

Чтобы проверить это, давайте запустим скрипт, передав ему имя несуществующего файла. Отлично, похоже, всё прошло успешно. Обратите внимание: программа завершилась с кодом 0, хотя мы не указывали это в коде явно. Это стандартное поведение.

Хорошо, а как вы думаете, что произойдёт, если мы снова запустим команду? Вы угадали: возникнет ошибка, так как файл уже существует, и программа завершится с кодом 1.

До сих пор мы взаимодействовали с операционной системой, используя встроенные возможности Python. Например, мы применяли файловые объекты для чтения содержимого файлов, модуль «shutil» — для проверки заполненности диска, а модуль «sys» — для обработки стандартного ввода, получения параметров или формирования кода завершения.

Но как быть, если нам нужно запустить системную программу из скрипта на Python? Допустим, в рамках скрипта нам требуется отправить ICMP-пакеты на хост, чтобы проверить, отвечает ли он. Можно попытаться найти внешний модуль, предоставляющий такую функциональность. А можно просто запустить команду «ping», которая сама отправит пакеты.

Иногда для выполнения задачи проще или быстрее воспользоваться системной командой внутри скрипта Python либо задействовать функциональность, отсутствующую как во встроенных, так и во внешних модулях Python. Для таких случаев в Python предусмотрена возможность выполнения системных команд с помощью функций модуля «subprocess».

```
>>> import subprocess
>>> subprocess.run(["date"])
Tue 07 Jan 2020 02:34:44 PM PST
CompletedProcess(args=['date'], returncode=0)
>>>
```

Давайте рассмотрим пример. Сначала импортируем модуль «subprocess», а затем вызовем команду «date» (отображающую текущую дату) с помощью функции «subprocess.run».

Функция «run» возвращает объект типа «CompletedProcess». Этот объект содержит информацию о выполнении команды. Из выведенных данных видно, что код завершения команды равен 0.

Для запуска внешней команды создается отдельная среда для дочернего процесса (или подпроцесса), в котором и выполняется эта команда. Пока родительский процесс — наш скрипт — ожидает завершения работы подпроцесса, он находится в состоянии блокировки; это означает, что родительский процесс не может выполнять никаких действий, пока дочерний процесс не завершит свою работу.

После того как внешняя команда завершает работу, дочерний процесс прекращает выполнение, и управление возвращается к родительскому процессу. Затем скрипт может продолжить работу в обычном режиме.

```
>>> subprocess.run(["sleep", "2"])
CompletedProcess(args=['sleep', '2'], returncode=0)
>>> result = subprocess.run(["ls", "this_file_does_not_exist"])
ls: cannot access 'this_file_does_not_exist': No such file or directory
>>> print(result.returncode)
2
>>>
```

Давайте посмотрим, как это работает, на примере команды «sleep», которая приостанавливает выполнение на заданное количество секунд. Вы, вероятно, заметили, что во время работы команды «sleep» интерпретатор был заблокирован, и мы не могли с ним взаимодействовать. Именно это и означает блокировку родительского процесса до завершения дочернего.

Обратите внимание на то, как мы вызываем команду. Функция «run» принимает список, первым элементом которого является имя вызываемой команды, а за ним следуют любые другие параметры, которые мы хотим ей передать. Таким образом, все элементы, идущие после имени программы, — это аргументы командной строки для нее. В данном случае мы даем команде «sleep» указание подождать две секунды.

В двух предыдущих примерах команды выполнились успешно, поэтому код возврата («returncode») в объекте завершившегося процесса был равен 0. Давайте рассмотрим пример, когда код завершения отличен от нуля. Если вызвать команду «ls» с именем несуществующего файла, она выведет сообщение об ошибке и вернет код завершения, отличный от нуля. Это значение будет сохранено в атрибуте «returncode» объекта завершившегося процесса, и мы сможем использовать его в своем коде. Мы видим, что команда завершилась неудачей, а сохраненный код возврата равен 2, что указывает на наличие ошибки. Эту информацию можно использовать в скрипте, чтобы выполнить иные действия в случае сбоя.

Такой способ использования функции «run» удобен, если нам нужно просто запустить команду и нас интересует лишь то, успешно ли она выполнилась. Вывод команды будет отображен на экране, а значит, наш скрипт не сможет его контролировать. Это может пригодиться для системных команд,

которые либо не выдают полезного вывода (например, cp, chmod, sleep и многие другие), либо когда нам не требуется дальнейшая обработка этого вывода. Иными словами, когда вполне допустимо просто вывести результат выполнения команды на экран. Например, если мы пишем скрипт, который меняет права доступа для множества файлов в дереве каталогов, нас не интересует вывод команды chmod. Нам важно лишь знать, успешно ли она выполнилась. Если же нам нужно перехватить вывод внешней команды, чтобы затем использовать полученные данные, потребуется другой подход.

```
>>> result = subprocess.run(["host", "8.8.8.8"], capture_output=True)
>>> print(result.returncode)
0
>>> print(result.stdout)
b'8.8.8.8.in-addr.arpa domain name pointer dns.google.\n'
>>>
```

Если мы хотим, чтобы наши скрипты на Python могли обрабатывать вывод выполняемой системной команды, нам нужно указать функции «run», чтобы она перехватывала этот вывод. Это может пригодиться, когда необходимо извлечь информацию из команды и затем использовать её для других целей в скрипте.

Например, представьте, что вы хотите собрать статистику о том, какие пользователи входят в систему на сервере в течение дня. Это можно сделать с помощью скрипта, вызывающего команду «who», которая выводит список пользователей, работающих в данный момент на компьютере. Скрипт может анализировать вывод команды, сохраняя список активных пользователей раз в час, а в конце дня — формировать итоговый отчет. Чтобы иметь возможность обрабатывать вывод команд, при вызове функции «run» мы установим параметр захвата вывода («capture_output») в значение «True».

В этом примере мы вызовем команду «host», которая позволяет преобразовать имя хоста в IP-адрес и наоборот. При вызове мы передадим параметр «capture_output=True» и сохраним

результат в переменной, чтобы иметь к нему доступ. Давайте попробуем. Мы знаем, что переменная с результатом представляет собой экземпляр класса «CompletedProcess», к которому мы можем обратиться. Мы можем проверить атрибут кода возврата (return code), как делали это ранее. Также мы можем вывести на экран и использовать данные, полученные в результате выполнения команды; они хранятся в атрибуте «stdout».

Что это за буква «b» в начале строки? Она указывает на то, что данная строка не является обычной строкой Python. На самом деле это массив байтов. Массивы байтов — тема непростая. Данные в компьютерах хранятся и передаются в виде байтов, и каждый байт может представлять до 256 различных символов. Однако существует огромное количество символов, используемых в разных языках. Например, для китайского языка требуется более 10 000 различных символов. Чтобы можно было писать на этих языках, со временем были разработаны специальные стандарты — кодировки, определяющие, каким последовательностям байтов соответствуют те или иные символы. В наши дни большинство людей используют кодировку UTF-8 — часть стандарта Unicode, содержащего перечень всех возможных символов.

```
>>> print(result.stdout.decode().split())
['8.8.8.8.in-addr.arpa', 'domain', 'name', 'pointer', 'dns.google.']
>>>
```

Возвращаясь к нашему примеру: при выполнении команды через «run» Python не знает, какую кодировку использовать для обработки вывода команды, поэтому просто представляет его как последовательность байтов. Чтобы получить полноценную строку, можно вызвать метод «decode». Этот метод применяет кодировку для преобразования байтов в строку; по умолчанию используется UTF-8, что нам и нужно. Итак, давайте преобразуем наш массив байтов в строку, а затем разделим её на несколько частей. Таким образом, мы сможем работать с выводом выполненной команды и делать с ним всё, что нам нужно. Например, можно оставить последний элемент списка — имя, соответствующее искомому IP-адресу.

Отлично. Мы только что рассмотрели перехваченный стандартный вывод. А как насчет стандартного вывода ошибок (stderr)? Если использовать параметр для захвата вывода, а команда запишет что-либо в стандартный вывод ошибок, эти данные сохранятся в атрибуте «stderr» объекта завершенного процесса.

```
>>> result = subprocess.run(["rm", "does_not_exist"], capture_output=True)
>>> print(result.returncode)
1
>>> print(result.stdout)
b''
>>> print(result.stderr)
b"rm: cannot remove 'does_not_exist': No such file or directory\n"
>>>
```

Давайте посмотрим на пример. Попробуем выполнить команду «rm» (используемую для удаления файлов), передав ей имя несуществующего файла. Проверим код возврата команды. Она завершилась с ошибкой — как мы и ожидали. Теперь проверим содержимое атрибутов «stdout» и «stderr». Мы видим, что в данном случае стандартный вывод был пустым. Однако возникла ошибка (вывод в стандартный поток ошибок), к которой можно обратиться через атрибут «stderr». Это отличный пример того, что стандартный вывод и стандартный вывод ошибок — это разные потоки, поэтому Python захватывает их по отдельности.

Модуль «subprocess» предоставляет множество дополнительных возможностей для использования в скриптах. Например, ранее мы упоминали, что один из способов передачи информации процессам — это изменение переменных окружения. С помощью этого механизма можно управлять тем, где процесс ищет исполняемые файлы, какие команды использует для взаимодействия с компонентами системы, какой вывод генерирует, и многим другим.

Обычно при изменении окружения дочернего процесса поступают так: сначала копируют текущее окружение родительского процесса, вносят необходимые изменения, а затем передают полученный набор переменных дочернему процессу.

```
import os
import subprocess

my_env = os.environ.copy()
my_env["PATH"] = os.pathsep.join(["/opt/myapp/", my_env["PATH"]])

result = subprocess.run(["myapp"], env=my_env)
```

Давайте посмотрим, как это делается. В данном примере мы начинаем с вызова метода «copy()» для словаря «os.environ», содержащего текущие переменные окружения. Это создает новый словарь, который можно изменять по своему усмотрению, не затрагивая исходное окружение.

В этом скрипте мы добавляем еще одну директорию в переменную «PATH». Напомним, что переменная «PATH» определяет, где операционная система ищет исполняемые программы. Добавляя запись в «PATH», мы указываем ОС искать программы в дополнительном месте.

Для формирования нового значения мы вызываем метод «join()» модуля «os.path». Он объединяет элементы переданного списка, используя разделитель путей, соответствующий текущей операционной системе. В данном случае мы объединяем путь «/opt/myapp» и старое значение переменной «PATH» с помощью этого разделителя. Наконец, мы запускаем команду «myapp», передавая подготовленное окружение в соответствующий параметр.

Подводя итог: этот скрипт изменяет содержимое переменной окружения «PATH», добавляя в нее новую директорию, а затем запускает команду «myapp» с использованием этой измененной переменной. При таком подходе команда будет выполнена в измененной среде с обновленным значением переменной PATH.

С функцией «run» можно использовать и множество других параметров. Например, параметр «cwd» позволяет изменить текущую рабочую директорию, в которой будет выполняться команда. Это очень удобно при работе с набором директорий, когда команду нужно запустить в каждой из них.

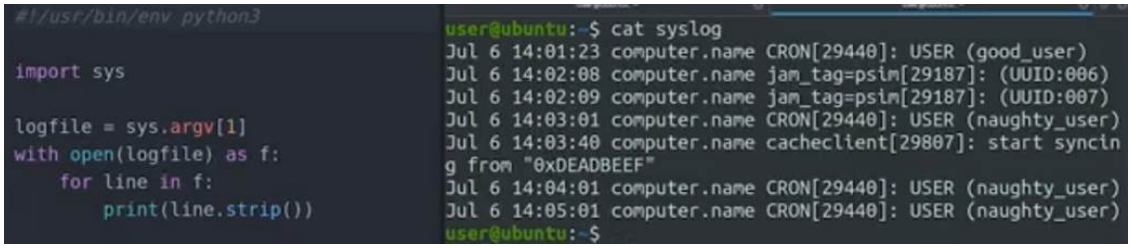
Также можно задать параметр «timeout». В этом случае функция «run» принудительно завершит процесс, если его выполнение займет больше времени, чем указано (в секундах). Это полезно, если вы запускаете команду, которая потенциально может «зависнуть» — например, при попытке подключения к сети, когда компьютер находится в офлайн-режиме.

Кроме того, можно использовать параметр «shell». Если установить его значение «True», Python сначала запустит экземпляр системной оболочки по умолчанию, а затем выполнит в ней заданную команду. Это позволяет использовать в командной строке подстановку переменных и другие операции, свойственные оболочке; без параметра «shell» это было бы невозможно.

Взаимодействие с базовой системой напрямую через скрипты Python (с помощью подпроцессов и системных команд) может быть полезным, особенно когда нужно быстро выполнить конкретную задачу. Но у этого подхода есть и недостатки. Использование команд системного уровня подразумевает, что в скрипты закладываются определенные допущения относительно инфраструктуры, в которой будет работать автоматизация. Если эти условия изменятся, это может привести к неожиданным последствиям или сбоям. Подобные допущения могут нарушиться по разным причинам.

Что произойдет с нашей автоматизацией, если изменятся флаги (аргументы) терминальной команды, а скрипт продолжит использовать старые? Что произойдет, если мы сменим операционную систему с Linux на Windows? Перестанут ли наши скрипты работать вовсе или же их выполнение приведет к неожиданным и, возможно, нежелательным последствиям? Любые изменения в системе или во внешних командах, используемых скриптами, повышают риск сбоев.

Иногда такой сбой очевиден, а иногда его трудно обнаружить. Если мы автоматизируем разовую, четко определенную задачу и главным требованием является скорость разработки, то использование системных команд и подпроцессов может очень помочь. Однако для более сложных или длительных процессов лучше воспользоваться встроенными или сторонними модулями Python. Поэтому, прежде чем прибегать к запуску подпроцесса, стоит заглянуть в стандартную библиотеку или репозиторий PyPI: возможно, задачу удастся решить средствами самого Python или кто-то уже создал нужный нам инструмент автоматизации. Не забывайте: не стоит изобретать велосипед.



```
#!/usr/bin/env python3

import sys

logfile = sys.argv[1]
with open(logfile) as f:
    for line in f:
        print(line.strip())
```

```
user@ubuntu:~$ cat syslog
Jul 6 14:01:23 computer.name CRON[29440]: USER (good_user)
Jul 6 14:02:08 computer.name jam_tag=psim[29187]: (UUID:006)
Jul 6 14:02:09 computer.name jam_tag=psim[29187]: (UUID:007)
Jul 6 14:03:01 computer.name CRON[29440]: USER (naughty_user)
Jul 6 14:03:40 computer.name cacheclient[29807]: start syncin
g from "0xDEADBEEF"
Jul 6 14:04:01 computer.name CRON[29440]: USER (naughty_user)
Jul 6 14:05:01 computer.name CRON[29440]: USER (naughty_user)
user@ubuntu:~$
```

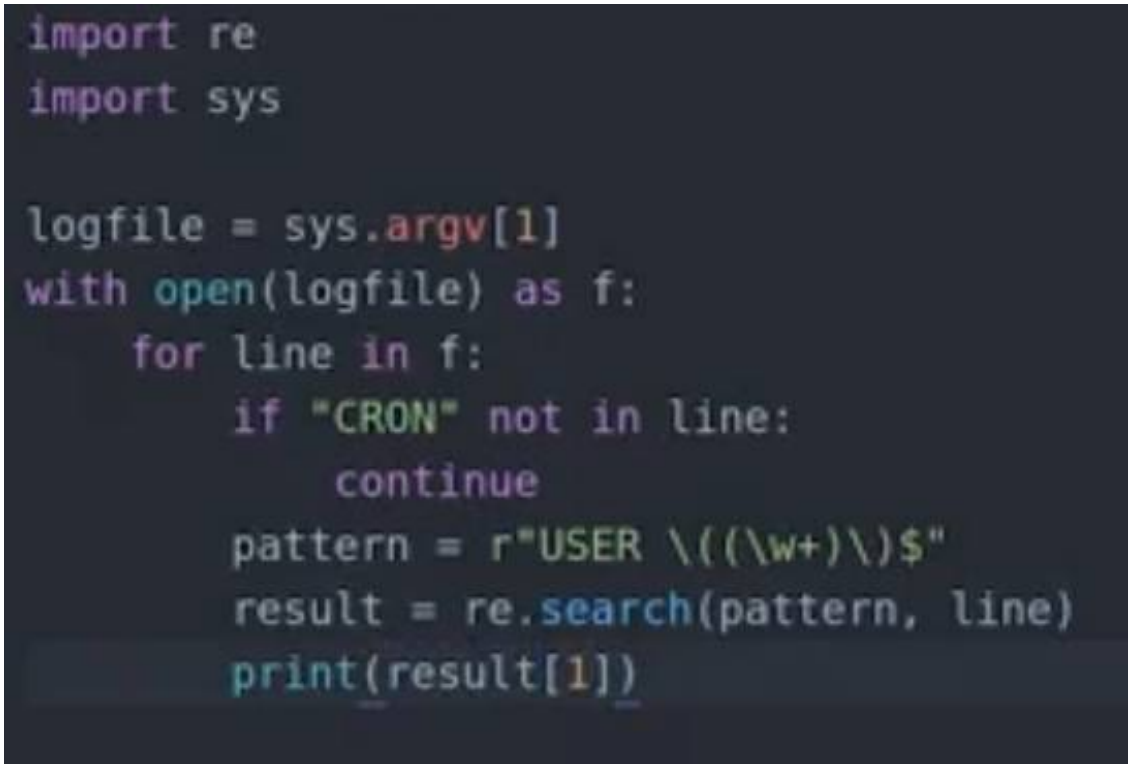
Мы уже обсуждали различные способы работы с файлами. Стандартный подход — вызвать функцию «open», которая возвращает файловый объект, а затем перебрать его строки с помощью цикла «for».

Помните: из соображений производительности при работе с большими файлами считается хорошей практикой читать их построчно, а не загружать всё содержимое в память.

Предположим, в нашем примере лог-файл содержит следующие сообщения. Сервер, генерирующий этот файл, ведет себя странно, и мы подозреваем, что причина — в задаче CRON, запущенной одним из системных администраторов.

Напомним, что задачи CRON используются для планирования запуска скриптов в операционных системах семейства UNIX. Чтобы выяснить, что происходит с сервером, мы хотим проанализировать лог-файлы и точно узнать, кто запускал задачи CRON.

Глядя на образец лога, мы видим, что наибольший интерес для нас представляют строки, содержащие подстроку «CRON». В этих строках также указано имя пользователя, запустившего задачу CRON (оно заключено в скобки).



```
import re
import sys

logfile = sys.argv[1]
with open(logfile) as f:
    for line in f:
        if "CRON" not in line:
            continue
        pattern = r"USER \((\w+)\)$"
        result = re.search(pattern, line)
        print(result[1])
```

Зная это, мы можем игнорировать любые строки, не содержащие подстроку «CRON». Проверить это можно с помощью ключевого слова «in». Здесь мы используем ключевое слово «continue», которое предписывает циклу перейти к следующему элементу.

Таким образом, если строка не содержит искомую подстроку, мы пропускаем её и переходим к следующей. Определив, что мы обрабатываем нужную строку лога, мы можем использовать регулярные выражения для извлечения имени пользователя. Это можно сделать разными способами. В данном примере мы воспользуемся экранирующими символами, группами захвата и якорем конца строки.

Поскольку имя пользователя находится в конце строки лога, мы используем якорь «знак доллара» ($), чтобы находить текст только в конце строки. Чтобы найти имя пользователя, мы ищем слово «user», за которым следует строка в скобках — именно так устроены эти записи. Это означает, что нам нужно экранировать скобки с помощью обратной косой черты. Поскольку мы хотим извлечь само имя пользователя, мы используем еще одну пару скобок для создания захватывающей группы. Для самого имени пользователя мы ищем любые буквенно-цифровые символы с помощью конструкции «\w+».

Это очень полезно, но нам может понадобиться и дополнительная информация. Чтобы улучшить результат, было бы неплохо подсчитать, сколько раз каждое имя пользователя встречается в логе. Как вы думаете, что можно для этого использовать? Если вы подумали о словаре, то вы правы. Как мы уже видели в предыдущих примерах, словари отлично подходят для подсчета количества вхождений строк.

```
>>> usernames = {}
>>> name = "good_user"
>>> usernames[name] = usernames.get(name, 0) +1
>>> print(usernames)
{'good_user': 1}
>>> usernames[name] = usernames.get(name, 0) +1
>>> print(usernames)
{'good_user': 2}
>>>
```

Мы будем использовать имена пользователей в качестве ключей словаря, а значения — для подсчета того, сколько раз каждое имя встречается в файле. Чтобы сделать это более лаконично, воспользуемся методом «get», о котором мы говорили.

Давайте опробуем этот подход в интерпретаторе, прежде чем добавлять его в код. Сначала создадим пустой словарь с помощью фигурных скобок. Затем установим значение для ключа равным текущему значению плюс единица, используя метод «get» для получения текущего значения. Мы получаем текущее значение из словаря, передавая ноль в качестве значения по умолчанию (на случай, если ключа в словаре еще нет). Затем мы прибавляем единицу и сохраняем результат как новое значение для этого ключа.

Посмотрим, что сейчас находится в словаре. Отлично. А что произойдет, если мы повторим эту операцию? Замечательно. Мы видим, что операция успешно выполняется как в случае, когда ключ уже есть в словаре, так и тогда, когда его там нет. Теперь добавим это в наш скрипт.

```
import re
import sys

logfile = sys.argv[1]
usernames = {}
with open(logfile) as f:
    for line in f:
        if "CRON" not in line:
            continue
        pattern = r"USER \((\w+)\)$"
        result = re.search(pattern, line)
        if result is None:
            continue
        name = result[1]
        usernames[name] = usernames.get(name, 0) + 1
print(usernames)
```

Нам нужно будет инициализировать пустой словарь в начале кода. Прежде чем добавлять значения в словарь, нужно убедиться, что регулярное выражение действительно нашло совпадение. Помните, как это сделать? Можно проверить, не равна ли переменная результата «None».

Мы воспользуемся тем же приемом, что и раньше: если результат равен «Non», будем использовать ключевое слово «continue». Теперь, вместо того чтобы просто выводить значения на экран, мы будем добавлять их в словарь по мере обработки файла. Для этого определим переменную «name», в которой будет храниться захваченная группа, и используем её в качестве ключа для словаря. И наконец, выведем содержимое словаря после завершения обработки файла.

Тестирование

Когда вы пишете совсем простой код — скажем, складываете две переменные, — легко понять, что именно он делает, и убедиться в правильности его работы. Однако по мере усложнения операций (использования циклов, условных конструкций и вызова множества функций) становится труднее быть уверенным в том, что код выполняет свою задачу именно так, как задумано.

Здесь на помощь приходит тестирование программного обеспечения. Тестирование — это процесс проверки кода для выяснения того, работает ли он в соответствии с ожиданиями. Тестируя программу, вы стремитесь обнаружить ошибки и дефекты, а также выявить сбои в работе. Во многом этот процесс напоминает испытания, проводимые при производстве новой техники. Когда новый автомобиль сходит с конвейера, важно убедиться, что при нажатии на газ он едет, а при нажатии на тормоз — останавливается. С программным обеспечением всё так же: нужно быть уверенным, что запущенная программа ведет себя должным образом — и тогда можно смело «давить на газ».

Скрипты и программы могут давать сбои самыми неожиданными способами, особенно по мере усложнения их логики. За исключением самых простых программ, практически невозможно предусмотреть и протестировать все возможные варианты сбоев. И хотя это означает, что в ваших скриптах могут оставаться незамеченные ошибки, не стоит волноваться. Написание тестов помогает устранить множество багов, повышая надежность и качество автоматизации. Тесты помогают превратить хороший код в отличный.

Одной из задач, стоящих перед программистами при написании кода, является его тестирование для проверки того, что он работает так, как ожидалось. Наличие качественных тестов помогает выявлять ошибки и сбои еще до того, как скрипты будут запущены для выполнения реальных задач автоматизации. Самый простой способ тестирования скрипта — запустить его с различными параметрами и проверить, возвращает ли он ожидаемые значения. Мы уже выполняли подобное ручное тестирование для некоторых фрагментов кода. Запуск скрипта с разными аргументами командной строки для наблюдения за изменениями в его поведении — это пример ручного тестирования.

Еще одной формой ручного тестирования является использование интерпретатора для проверки кода перед его включением в скрипт. Формальное тестирование программного обеспечения выводит этот процесс на новый уровень, оформляя тесты в виде отдельного программного кода, который можно запускать для проверки корректности работы наших программ. Это называется автоматическим тестированием. Цель автоматического тестирования — автоматизировать процесс проверки соответствия возвращаемого значения ожидаемому результату. Вместо того чтобы вручную многократно запускать функцию с разными параметрами и проверять результаты, мы поручаем эту задачу компьютеру.

Автоматическое тестирование подразумевает написание кода для выполнения самих тестов. Зачем писать дополнительный код для проверки уже имеющегося? Потому что при тестировании необходимо убедиться, что код выполняет свои функции для множества различных входных данных. Следует проверить поведение программы для широкого набора возможных значений, называемых тестовыми примерами (test cases).

Допустим, вы пишете скрипт для обновления домена в списке адресов электронной почты — подобный тому, который мы создавали ранее. Вам нужно будет проверить, как программа работает, если список содержит один, два или десять адресов. Также стоит проверить поведение скрипта, если новый домен состоит из одного или двадцати символов либо представляет собой пустую строку. Также стоит проверить, как система поведет себя, если список

содержит только адреса, требующие обновления, только те, что не требуют обновления, или их комбинацию.

Как видите, список сценариев для проверки может очень быстро стать весьма внушительным. Чем больше тестовых случаев вы охватите, тем надежнее будет проверен код и тем увереннее вы сможете утверждать, что он работает именно так, как ожидается. При ручном тестировании вряд ли удастся каждый раз проверять все возможные варианты после внесения изменений в код. Скорее всего, мы ограничимся лишь несколькими тестами, рискуя пропустить ошибку. Это не лучший подход. Именно поэтому мы не хотим выполнять такие проверки вручную, а предпочитаем поручить эту задачу компьютеру.

Как и в любом другом примере автоматизации, преимущество автоматических тестов заключается в том, что их можно запускать сколько угодно раз, неизменно получая одни и те же результаты. Компьютер будет раз за разом выполнять одни и те же проверки, позволяя нам убедиться, что возвращаемое значение соответствует ожиданиям. Если же по какой-то причине результат не совпадет с ожидаемым, возникнет ошибка, и мы сможем проанализировать код, чтобы разобраться в ситуации.

Существует множество различных типов тестов для автоматизированного тестирования. Самый распространенный из них — модульный тест (unit test). Модульные тесты используются для проверки корректности работы небольших изолированных частей программы. Обычно их пишут параллельно с основным кодом, чтобы проверить поведение отдельных фрагментов или модулей — например, функций или методов. Такие тесты позволяют разработчику убедиться, что каждый участок кода выполняет свою задачу именно так, как задумано.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.