Часто задаваемые вопросы по Python

Выпуск 3.8.8

Гвидо ван Россум
и команда разработчиков Python

августа 12, 2021

https://Digitology.tech
Email: tweakit@bk.ru

Оглавление

ГЛАВА 1

Общие часто задаваемые вопросы по Python

1.1 Общие сведения

1.1.1 Что такое Python?

Python — это интерпретируемый интерактивный объектно-ориентированный язык программирования. Он включает модули, исключения, динамическую типизацию, динамические типы данных очень высокого уровня и классы. Он поддерживает несколько парадигм программирования, помимо объектно-ориентированного программирования, таких как процедурное и функциональное программирование. Python сочетает в себе замечательную мощь с очень понятным синтаксисом. Он имеет интерфейсы ко многим системным вызовам и библиотекам, а также к различным оконным системам и может быть расширен на C или C++. Его также можно использовать в качестве языка расширения для приложений, которым требуется программируемый интерфейс. Наконец, Python переносим: он работает во многих вариантах Unix, включая Linux и macOS, а также в Windows.

Чтобы узнать больше, начните с tutorial-index. В Руководство по Python для начинающих содержатся ссылки на другие вводные учебники и ресурсы для обучения Python.

1.1.2 Что такое фонд программного обеспечения Python?

Фонд программного обеспечения Python (PSF, Python Software Foundation) - независимая некоммерческая организация, обладающая авторским правом на Python версии 2.1 и более поздние. Миссия PSF заключается в продвижении технологии с открытым исходным кодом, связанной с языком программирования Python, и в распространении информации об использовании Python. Домашняя страница PSF находится на https://www.python.org/psf/. Пожертвования в PSF не облагаются налогом в США. Если вы используете Python и находите это полезным, пожалуйста, внесите свой вклад через страницу пожертвований PSF.

1.1.3 Существуют ли ограничения авторских прав на использование Python?

Вы можете делать все, что хотите, с исходниками, пока оставляете авторские права и отображать эти авторские права в любой документации о Python, которые вы производите. Если вы соблюдаете правила авторского права, можно использовать Python для коммерческого использования, продавать копии

Python в исходной или двоичной форме (модифицированные или немодифицированные) или продавать продукты, включающие Python в той или иной форме. Конечно, мы все еще хотели бы знать обо всем коммерческом использовании Python.

Для получения дополнительных пояснений и ссылки на полный текст лицензии см. раздел страница лицензии PSF.

Логотип Python имеет торговую марку, и в некоторых случаях для его использования требуется разрешение. Для получения дополнительной информации обратитесь к Политика использования товарных знаков.

1.1.4 Почему вообще был создан Python?

Вот *очень* краткое изложение того, с чего все началось, написане Гвидо ван Россум:

> У меня был большой опыт внедрения интерпретируемого языка в группе ABC в CWI и из работы с этой группой я много узнал о дизайне языка. Это является источником многих Python особенностей, включая использование отступов для группировки инструкция и включение типов данных очень высокого уровня (хотя все детали отличаются в Python).
>
> У меня был ряд претензий к языку ABC, но мне также понравились многие его функции. Невозможно было расширить язык ABC (или его реализация), чтобы исправить мои жалобы - на самом деле его отсутствие расширяемости была одной из самых больших проблем. У меня был опыт использования Модула-2 + пообщался с разработчиками Модулы-3 и прочитал отчет Модула-3. Modula-3 является источником синтаксиса и семантики, используемых для исключений, и некоторых других функций Python.
>
> Я работал в группе распределенных операционных систем Amoeba в CWI. Нам нужен был лучший способ управления системой, чем написание сценариев C программ или оболочки Борна, так как Amoeba имела собственный интерфейс системного вызова, который не был легко доступен из оболочки Борна. Мой опыт работы с ошибками в Amoeba убедил меня в важности исключений как возможности языка программирования.
>
> Мне пришло в голову, что скриптовый язык с таким синтаксисом, как ABC, но с доступом к вызовам системы Amoeba, заполнит потребность. Я понял, что было бы глупо писать язык, специфичный для Amoeba, поэтому я решил, что мне нужен язык, который был бы в целом расширяемым.
>
> Во время рождественских праздников 1989 года у меня было много времени, поэтому я решил попробовать. В течение следующего года, еще в основном работая над ним в свободное время, Python был использован в проекте Amoeba с возрастающим успехом, и отзывы коллег заставили меня добавить много ранних улучшений.
>
> В феврале 1991 года, после чуть более года разработки, я решил опубликовать в USENET. Остальное находится в файле `Misc/HISTORY`.

1.1.5 Для чего Python хорош?

Python - высокоуровневый язык программирования общего назначения, который может применяться ко многим различным классам задач.

Язык поставляется с большой стандартной библиотекой, которая охватывает такие области, как обработка строк (регулярные выражения, Юникод, вычисление различий между файлами), интернет-протоколы (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI программирование), программная инженерия (модульное тестирование, логирование, профилирование, парсинг Python кода) и интерфейсы операционной системы (системные вызовы, файловые системы, TCP/IP, сокеты). Ознакомьтесь с содержанием library-index, чтобы получить представление о доступных возможностях. Также доступны разнообраз-

ные расширения сторонних производителей. Проконсультируйтесь с Пакетным индексом Python, чтобы найти интересующие вас пакеты.

1.1.6 Как работает схема нумерации версий Python?

Версии Python имеют нумерацию A.B.C или A.B. A является основным номером версии - он увеличивается только для действительно серьезных изменений в языке. B - это второстепенный номер версии, увеличенный для менее значимых изменений. C - микроуровень - он увеличивается для каждого bugfix релиза. Дополнительные сведения о версиях bugfix см. в разделе **PEP 6**.

Не все выпуски содержат bugfix. В преддверии нового основного выпуска выпускаются серии разрабатываемых выпусков, обозначаемых как альфа, бета или релиз кандидат. Альфа-версии - это ранние версии, в которых интерфейсы еще не доработаны; Неудивительно, что интерфейс между двумя альфа-версиями изменился. Бета-версии более стабильны, сохраняя существующие интерфейсы, но, возможно, добавляя новые модули и релиз-кандидаты заморожены, без изменений, кроме как при необходимости исправить критические ошибки.

Альфа-версии, бета-версии и версии-кандидаты имеют дополнительный суффикс. Суффикс для альфа-версии - «aN» для некоторого небольшого числа N, суффикс для бета-версии - «bN» для некоторого небольшого числа N, а суффикс для версии- кандидата выпуска равен «cN» для небольшого числа N. Другими словами, все версии, помеченные 2.0aN, предшествуют версиям, обозначенным 2.0bN, которые предшествуют версиям, обозначенным 2.0cN, и *me* предшествуют 2.0.

Также можно найти номера версий с суффиксом «+», например «2.2+». Это неизданные версии, созданные непосредственно из репозитория разработки CPython. На практике после того, как сделан окончательный минорный релиз, версия увеличивается до следующей минорной версии, которая становится версией «a0», например «2.4a0».

См. также документацию по `sys.version`, `sys.hexversion` и `sys.version_info`.

1.1.7 Как мне получить копию исходного кода Python?

Новейший дистрибутив исходников Python всегда доступен в python.org по адресу https://www.python.org/downloads/. Последние разрабатываемые исходники можно получить по адресу https://github.com/python/cpython/.

Исходный дистрибутив представляет собой gzipped tar файл, содержащий полностью все исходники C файлов, документацию в формате Sphinx, Python библиотечные модули, примерные программы и несколько полезных частей свободно распространяемого программного обеспечения. Исходный файл будет компилироваться и заканчиваться на большинстве платформ UNIX.

Проконсультируйтесь с разделом «Начало работы» Руководства разработчика Python для большей информации о получении исходного кода и его компиляции.

1.1.8 Как получить документацию по Python?

Стандартная документация для текущей стабильной версии Python доступна по адресу https://docs.python.org/3/. PDF, обычный текст и загружаемые версии HTML также доступны по адресу https://docs.python.org/3/download.html.

Документация написана в reStructuredText и обрабатывается инструментом документации Sphinx. Источник reStructuredText для документации является частью Python дистрибутива исходников.

1.1.9 Я никогда раньше не программировал. Есть ли учебник по Python?

Имеется множество учебных пособий и книг. Стандартная документация включает tutorial-index.

Обратитесь к Руководстве для начинающих, чтобы найти информацию для начинающих Python программистов, включая списки учебных пособий.

1.1.10 Существует ли группа новостей или список рассылки, посвященный Python?

Есть группа новостей, `comp.lang.python`, и список рассылки, python-list. Группа новостей и список рассылки соединены между собой - если вы можете читать новости, нет необходимости подписываться на список рассылки. `comp.lang.python` - это большой трафик, получающий сотни сообщений каждый день, и читатели Usenet часто более способны справиться с этим объемом.

Объявления о новых выпусках программного обеспечения и событиях можно найти в comp.lang.python.announce, списке с низким трафиком, который получает около пяти сообщений в день. Он доступен как список рассылки python-announce.

Более подробную информацию о других списках рассылки и группах новостей можно найти по адресу https://www.python.org/community/lists/.

1.1.11 Как получить бета-версию Python?

Альфа и бета-версии доступны в https://www.python.org/downloads/. Все выпуски анонсируются в группах новостей comp.lang.python и comp.lang.python.announce и на главной странице Python по адресу https://www.python.org/; доступна RSS- ленте новостей.

Также можно получить доступ к версии разработки Python через Git. Дополнительные сведения см. в разделе Руководство разработчика Python.

1.1.12 Как мне отправлять отчеты об ошибках и исправления для Python?

Чтобы сообщить об ошибке или отправить патч, используйте установку Roundup по адресу https://bugs.python.org/.

Для сообщения об ошибках необходимо иметь учетную запись Roundup; это позволяет нам связаться с вами, если у нас есть последующие вопросы. Это также позволит Roundup отправлять вам обновления во время работы с вашей ошибкой. Если вы ранее используемый SourceForge, чтобы сообщить об ошибках Python, вы можете получить пароль Roundup через процедуру сброса пароля Roundup.

Для получения дополнительной информации о том, как разрабатывается Python, обратитесь к Руководство разработчика Python.

1.1.13 Есть ли опубликованные статьи о Python, на которые я могу сослаться?

Наверное, лучше привести вашу любимую книгу о Python.

Самая первая статья о Python была написана в 1991 году и сейчас довольно устарела.

> Гвидо ван Россум и Джелк де Бур, «Интерактивное тестирование удаленных серверов с использованием языка программирования Python», CWI ежеквартально, том 4, выпуск 4 (декабрь 1991), Амстердам, стр. 283-303.

1.1.14 Есть ли книги по Python?

Да, их много, и публикуется еще больше. Список см. в python.org wiki в https://wiki.python.org/moin/PythonBooks.

Вы также можете искать «Python» в книжных магазинах в Интернете и отфильтровывать ссылки на Монти Пайтон; или возможно, искать «Python» и «язык».

1.1.15 Где в мире находится www.python.org?

Инфраструктура проекта Python расположена по всему миру и управляется Командой инфраструктуры Python. Подробности здесь.

1.1.16 Почему он называется Python?

Когда он начал реализовывать Python, Гвидо ван Россум также читал опубликованные сценарии из «Летающий цирк Монти Пайтона», комедийного сериала BBC 1970-х годов. Ван Россум думал, что ему нужно имя, которое было бы коротким, уникальным и немного загадочным, поэтому он решил назвать язык Python.

1.1.17 Должен ли я полюбить «Летающий цирк Монти Пайтона»?

Нет, но это помогает.:)

1.2 Python в реальном мире

1.2.1 Насколько стабилен Python?

Очень стабильный. Новые стабильные версии выходили примерно каждые 6–18 месяцев с 1991 года, и, похоже, это будет продолжаться. Начиная с версии 3.9, каждые 12 месяцев будет выпускаться новый крупный выпуск Python (**PEP 602**).

Разработчики выпускают «bugfix» выпуски старых версий, поэтому стабильность существующих выпусков постепенно улучшается. Выпуски Bugfix, обозначенные третьим компонентом номера версии (например, 3.5.3, 3.6.2), управляются для обеспечения стабильности; в выпуске bugfix включены только исправления известных проблем, и гарантировано, что интерфейсы останутся прежними на протяжении всего ряда выпусков bugfix.

Последние стабильные версии всегда можно найти на странице загрузки Python. Существует две готовые к промышленному использованию версии Python: 2.x и 3.x. Рекомендуемая версия - 3.x, которая поддерживается наиболее широко используемыми библиотеками. Хотя 2.x по-прежнему широко используется, он не будет поддерживаться после 1 января 2020 г..

1.2.2 Сколько людей используют Python?

Наверное, есть миллионы пользователей, хотя точно подсчитать сложно.

Python доступен для бесплатной загрузки, поэтому нет данных о продажах, и он доступен со многих различных сайтов и упакован со многими дистрибутивами Linux, так что статистика загрузок тоже не рассказывает всю историю.

Группа новостей comp.lang.python очень активна, но не все Python пользователи выкладывают в группу или даже читают её.

1.2.3 Были ли реализованы какие-либо значимые проекты на Python?

См. https://www.python.org/about/success для получения списка проектов, использующих Python. Консультации по материалам прошлые конференций Python покажет вклад многих различных компаний и организаций.

Известные Python проекты включают в себя менеджер списков рассылки Mailman и сервер приложений Zope. Несколько дистрибутивов Linux, особенно Red Hat, написали часть или все свои программы установки и системного администрирования на Python. Компании, которые используют Python внутри себя, включают Google, Yahoo и Lucasfilm Ltd.

1.2.4 Какие новые разработки ожидаются в Python в будущем?

См. предложения по совершенствованию Python https://www.python.org/dev/peps/ (PEP). PEP представляют собой проектную документацию, описывающую предлагаемую новую функцию Python и содержащую краткую техническую спецификацию и обоснование. Найдите PEP под названием «График выпуска Python X.Y», где X.Y - версия, которая еще не была публично выпущена.

Новая разработка обсуждается на списке рассылки python-dev.

1.2.5 Разумно ли предлагать несовместимые изменения в Python?

В общем, нет. Уже существуют миллионы строк кода Python по всему миру. Поэтому любое изменение в языке, которое делает недействительным более чем очень малую часть существующих программ должна быть осуждена. Даже если вы можете предоставить программу преобразования, есть еще проблема обновления всей документации; о Python было написано много книг, и мы не хотим их аннулировать и все это одним махом.

Если необходимо изменить функцию, необходимо обеспечить постепенное обновление. : pep: *5* описывает процедуру, которую использовали для введения обратно несовместимых изменения, сводя к минимуму неудобства для пользователей.

1.2.6 Является ли Python хорошим языком для начинающих программистов?

Да.

По-прежнему принято начинать обучение с процедурного и статически типизированного языка, такого как Pascal, C или подмножество C++ или Java. Студенты могут лучше обучаться, изучая Python как свой первый язык. Python имеет очень простой и согласованный синтаксис и большую стандартную библиотеку, и что самое главное, использование Python в начальном курсе программирования позволяет студентам сосредоточиться на важных навыках программирования, таких как разложение проблем и дизайн типа данных. С помощью Python студенты могут быстро знакомиться с основными понятиями, такими как циклы и процедуры. Вероятно, они могут даже работать с пользовательскими объектами в их самом первом курсе.

Для студента, который никогда раньше не программировал, использование статически типизированного языка кажется неестественным. Это представляет дополнительную сложность, которую студент должен освоить и замедляет темп курса. Студенты пытаются научиться мыслить как компьютер, разлагать проблемы, разрабатывать согласованные интерфейсы и инкапсулировать данные. Хотя обучение использованию статически типизированного языка важно в долгосрочной перспективе, это не обязательно лучшая тема для изучения в первом курсе программирования студентов.

Многие другие аспекты Python делают его хорошим первым языком. Как и Java, Python имеет большую стандартную библиотеку, чтобы студентам можно было назначать программные проекты в самом начале курса, которые что-то *делают*. Задания не ограничиваются стандартным калькулятором с четырьмя функциями и программами балансирования чеков. Используя стандартную библиотеку, студенты могут получить удовлетворение от работы над реалистичными приложениями по мере изучения основ программирования. Использование стандартной библиотеки также обучает учащихся код повторному использованию. Сторонние модули, такие как PyGame, также помогают расширить охват учащихся.

Интерактивный интерпретатор Python позволяет студентам тестировать языковые функции пока они программируют. Они могут держать окно с запущенным интерпретатором пока они вводят исходный код своей программы в другом окне. Если они не могут запомнить методы для списка, они могут делать что-то вроде этого:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
↪'remove', 'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

Также как и интерпретатор, документация всегда рядом с учащимся, так как они программируют.

Также есть хорошие IDE для Python. IDLE - кроссплатформенная среда IDE для Python, которая написана на Python с использованием Tkinter. PythonWin - это специфичная для Windows среда IDE. Пользователи Emacs будут рады узнать, что для Emacs существует очень хороший режим Python. Все эти среды программирования обеспечивают подсветку синтаксиса, автоматическое выделение отступов и доступ к интерактивному интерпретатору при кодировании. Полный список сред редактирования Python можно найти в вики Python.

Если вы хотите обсудить использование Python в образовании, вам может быть интересно присоединиться к список рассылки edu-sig.

ГЛАВА 2

Часто задаваемые вопросы по программированию

2.1 Общие вопросы

2.1.1 Есть ли отладчик уровня исходного кода с точками останова, пошаговым режимом и т.д.?

Да.

Ниже описано несколько отладчиков для Python, и встроенная функция `breakpoint()` позволяет попасть в любой из них.

Модуль pdb является простым, но адекватным отладчиком консольного режима для Python. Является частью стандартной библиотеки Python и `задокументирован в Справочном руководстве библиотеки`. Можно также написать собственный отладчик, используя в качестве примера код для pdb.

Интерактивная среда разработки IDLE, которая является частью стандартного дистрибутива Python (обычно доступен как Tools/scripts/idle), включает графический отладчик.

PythonWin - это Python IDE, включающая GUI-отладчик на основе pdb. Отладчик Pythonwin окрашивает точки останова и имеет довольно много классных функций, таких как отладка программ, не относящихся к Pythonwin. Pythonwin доступен как часть проекта Python для расширений Windows и как часть дистрибутива ActivePython (см. https://www.activestate.com/activepython).

Eric - это среда, построенная на основе PyQt и компонента редактирования Scintilla.

Pydb - версия стандартного Python дебаггера pdb, модифицированного для использования с DDD (Отладчик отображаемых данных, Data Display Debugger), популярным графическим отладочным интерфейсом. Pydb можно найти в http://bashdb.sourceforge.net/pydb/, DDD - в https://www.gnu.org/software/ddd.

Есть много коммерческих IDE Python, которые включают графические отладчики. К ним относятся:

- Wing IDE (https://wingware.com/)
- Komodo IDE (https://komodoide.com/)
- PyCharm (https://www.jetbrains.com/pycharm/)

2.1.2 Существуют ли инструменты, помогающие находить ошибки или выполнять статический анализ?

Да.

Pylint и Pyflakes выполняют базовую проверку, которая поможет вам быстрее обнаруживать ошибки.

Статические проверяльщики типа, такие как Mypy, Pyre и Pytype, могут проверять подсказки типа в исходном коде Python.

2.1.3 Как создать автономный двоичный файл из сценария Python?

Вам не нужна возможность компиляции Python в C код, если все, что вам нужно, это автономная программа, которую пользователи могут загрузить и запустить без необходимости установки Python дистрибутива. Существует ряд инструментов, которые определяют набор модулей, необходимых программе, и связывают эти модули вместе с двоичным файлом Python для создания одного исполняемого файла.

Одним из них является использование инструмента freeze (замораживания), который включен в дерево исходников Python в качестве `Tools/freeze`. Преобразует Python байт код в массивы C; компилятор C, который позволяет встроить все модули в новую программу, связанную со стандартными модулями Python.

Она работает путем рекурсивного сканирования исходного кода для инструкции import (в обеих формах) и поиска модулей в стандартном пути Python, а также в исходном каталоге (для встроенных модулей). Затем он преобразует байт-код для модулей, записанных в Python, в C код (инициализаторы массива, которые можно превратить в кодовые объекты с помощью модуля marshal) и создает пользовательский конфигурационный файл, содержащий только те встроенные модули, которые фактически используются в программе. Затем он компилирует сгенерированный код C и связывает его с остальной частью Python интерпретатора, чтобы сформировать автономный двоичный файл, который действует точно так же, как ваш скрипт.

Очевидно, что для замораживания требуется компилятор C. Есть несколько других утилит, которые этого не делают. Один из них - py2exe Томаса Хеллера (только для Windows)

> http://www.py2exe.org/

Другой инструмент - cx_Freeze Энтони Туининга.

2.1.4 Существуют ли стандарты кодирования или руководство по стилю для программ Python?

Да. Стиль кодирования, необходимый для стандартных библиотечных модулей, документируется как **PEP 8**.

2.2 Основной язык

2.2.1 Почему я получаю UnboundLocalError, когда переменная имеет значение?

Неожиданностью может быть получение UnboundLocalError в предыдущем рабочем коде при его изменении путем добавления инструкция назначения где-либо в теле функции.

Следующий код:

```
>>> x = 10
>>> def bar():
...     print(x)
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
>>> bar()
10
```

работает, но этот код:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

приводит к ошибке UnboundLocalError:

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

Это связано с тем, что при назначении переменной в область видимости эта переменная становится локальной к этой области видимости и затеняет любую переменную с аналогичным именем во внешней области видимости. Поскольку последняя инструкция в foo присваивает значение новый `x`, компилятор распознает его как локальная переменная. Следовательно, когда ранее `print(x)` пытается распечатать неинициализированную локальную переменную и возникает ошибка.

В приведенном выше примере можно получить доступ к переменной внешней области видимости, объявив ее глобальной:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
>>> foobar()
10
```

Это явное объявление требуется для того, чтобы напомнить, что (в отличие от поверхностно аналогичной ситуации с переменными класса и сущности) вы фактически изменяете значение переменной во внешнем область видимости:

```
>>> print(x)
11
```

Подобное можно сделать во вложенном область видимости с помощью ключевого `nonlocal`:

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
>>> foo()
10
11
```

2.2.2 Каковы правила для локальных и глобальных переменных в Python?

В Python переменные, на которые ссылаются только внутри функции, неявно глобальны. Если переменной назначается значение в любом месте тела функции, она считается локальной, если она явно не объявлена как глобальная.

Хотя сначала это немного удивительно, мгновение рассмотрения объясняет это. С одной стороны, требование `global` для назначенных переменных обеспечивает планку против непреднамеренных побочных эффектов. С другой стороны, если бы `global` требовалось для всех глобальных ссылок, вы бы использовали `global` постоянно. Необходимо объявить глобальной каждую ссылку на встроенную функцию или на компонент импортированного модуля. Этот беспорядок победил бы полезность декларации `global` для выявления побочных эффектов.

2.2.3 Почему лямбда-выражения, определенные в цикле с разными значениями, возвращают один и тот же результат?

Предположим, что используется цикл for для определения нескольких различных лямбд (или даже простых функций), например:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

Этот код создает список, содержащий 5 лямбд, которые вычисляют `x**2`. Можно ожидать, что при вызове они будут возвращает соответственно `0`, `1`, `4`, `9` и `16`. Однако, когда вы на самом деле проверите, вы увидите, что все они возвращают `16`:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

Это происходит потому, что `x` не локальная к лямбдам, но определяется во внешнем область видимости, и доступ к ней осуществляется при вызове лямбды, — не при ее определении. В конце цикла значение `x` `4`, поэтому все функции теперь возвращают `4**2`, т.е. `16`. Это также можно проверить, изменив значение `x`, и посмотреть, как изменяются результаты лямбд:

```
>>> x = 8
>>> squares[2]()
64
```

Чтобы избежать этого, необходимо сохранить значения в переменных, локальных к лямбдам, чтобы они не полагались на значение глобального `x`:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Здесь, `n=x` создает новую переменную `n` локальную к лямбде и вычисленный, когда лямбда определена так, чтобы у нее было тот же значение, которое `x` имел в той точке в цикле. Это означает, что значение `n` будет `0` в первой лямбде, `1` во второй, `2` в третьей и так далее. Поэтому каждая лямбда теперь возвращает правильный результат:

```
>>> squares[2]()
4
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
>>> squares[4]()
16
```

Обратите внимание, что это поведение не свойственно лямбдам, но относится и к обычным функциям.

2.2.4 Как передать глобальные переменные в модули?

Каноническим способом обмена информацией между модулями в рамках одной программы является создание специального модуля (часто называемого config или cfg). Просто импортируйте конфигурационный модуль во все модули приложения; модуль становится доступным как глобальное имя. Поскольку существует только одна сущность каждого модуля, любые изменения, внесенные в объект модуля, отражаются везде. Например:

config.py:

```
x = 0   # значение по умолчанию параметра конфигурации "x"
```

mod.py:

```
import config
config.x = 1
```

main.py:

```
import config
import mod
print(config.x)
```

Следует отметить, что использование модуля также является основой для реализации шаблона конструкции Синглетон по той же причине.

2.2.5 Каковы «рекомендации» по использованию import в модуле?

В общем, не используй `from modulename import *`. Это загромождает пространство имен импортера и значительно усложняет для линтеров обнаружение неопределенных имен.

Импортируйте модули в начале файла. Так будет понятно, какие еще модули требуются вашему коду и позволяет избежать вопросов о том, входит ли имя модуля в область видимости. Использование одного импорта на строку упрощает добавление и удаление импорта модуля, но при использовании нескольких операций импорта на строку используется меньше места на экране.

Рекомендуется импортировать модули в следующем порядке:

1. стандартные библиотечные модули - например, `sys`, `os`, `getopt`, `re`
2. сторонние библиотечные модули (какие-либо установленные в каталоге site-packages Python'a) - например, mx.DateTime, ZODB, PIL.Image и т.д.
3. локально разработанные модули

Иногда необходимо переместить импорт в функцию или класс, чтобы избежать проблем с циклическим импортом. Гордон МакМиллан говорит:

> Циклический импорт подходит для тех случаев, когда оба модуля используют форму импорта «import <module>». Они завершаются неудачей, когда второй модуль хочет получить имя

из первого («from module import name»), а импорт выполняется на верхнем уровне. Это происходит потому, что имена в 1-м модуле еще недоступны, потому что первый модуль занят импортом 2-го.

В этом случае, если второй модуль используется только в одной функции, то импорт можно легко переместить в эту функцию. К моменту вызова импорта первый модуль завершит инициализацию, а второй модуль сможет выполнить импорт.

Также может потребоваться переместить импорт с верхнего уровня кода, если некоторые модули специфичны для платформы. В этом случае может оказаться невозможным даже импортировать все модули в верхней части файла. В этом случае импорт правильных модулей в соответствующие специфичные для платформы кодом является хорошим вариантом.

Перемещение импорта в локальную область видимости, например внутри определения функции, только в том случае, если необходимо решить проблему, например для избежания циклического импорта, или попытаться сократить время инициализации модуля. Этот метод особенно полезен, если многие операции импорта не требуются в зависимости от способа выполнения программы. Можно также переместить импорт в функцию, если модули используются только в этой функции. Следует отметить, что загрузка модуля в первый раз может быть дорогостоящей из-за однократной инициализации модуля, но загрузка модуля несколько раз является фактически свободной, что обходится только к паре поисков по словарю. Даже если имя модуля вышло из области видимости, модуль, вероятно, доступен в `sys.modules`.

2.2.6 Почему значения по умолчанию разделяются между объектами?

Этот тип ошибок обычно смущает начинающих программистов. Рассмотрим функцию:

```
def foo(mydict={}):  # Опасность: общая ссылка на один словарь для всех
↪вызовов
    ... compute something ...
    mydict[key] = value
    return mydict
```

При первом вызове функции `mydict` содержит один элемент. Во-втором, `mydict` содержит два элемента, потому что когда `foo()` начинает выполняться, `mydict` начинается с элемента, уже находящегося в нем.

Часто ожидается, что вызов функции создает новые объекты для значения по умолчанию. Это не то, что происходит. значения по умолчанию создаются ровно один раз при определении функции. Если этот объект изменен, как и словарь в этом примере, последующие вызовы функции будут ссылаться на измененный объект.

По определению, неизменяемые объекты, такие как числа, строки, кортежи и `None`, безопасны для изменения. Изменения в изменяемых объектах, таких как словари, списки и сущности классов, могут привести к путанице.

Благодаря этой функции рекомендуется не использовать изменяемые объекты в качестве значений по умолчанию. Вместо этого используйте `None` в качестве значение по умолчанию и внутри функции, проверьте, является ли параметр `None` и создайте новый список/словарь/что угодно, если он нужен. Например, не пиши:

```
def foo(mydict={}):
    ...
```

но:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # создать новый словарь для локального пространства имен
```

Эта функция может быть полезной. При наличии функции, которая требует много времени для вычисления, обычным методом является кэширование параметров и результирующего значение каждого вызова функции и возвращает кэшированное значение, если такое же самое значение запрашивается снова. Это называется «запоминание» и может быть реализовано так:

```
# Вызывающие могут предоставить только два параметра и дополнительно передать
↪_cache по ключю
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Рассчитать значение
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result           # Сохранить результат в кэше
    return result
```

Вместо значения по умолчанию можно использовать глобальную переменную, содержащую словарь; это вопрос вкуса.

2.2.7 Как передать необязательные параметры или ключевые параметры из одной функции в другую?

Собирайте аргументы, используя спецификаторы `*` и `**` в списке параметров функции; это дает позиционные аргументы как кортеж и ключевые аргументы как словарь. Эти аргументы можно передать при вызове другой функции с помощью `*` и `**`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

2.2.8 В чем разница между аргументами и параметрами?

Параметры определяются именами, которые появляются в определении функции, в то время как *аргументы* являются значениями, фактически передаваемыми функции при ее вызове. Параметры определяют, какие типы аргументов может принимать функция. Например, учитывая определение функции:

```
def func(foo, bar=None, **kwargs):
    pass
```

foo, *bar* и *kwargs* являются параметрами `func`. Однако при вызове `func`, например:

```
func(42, bar=314, extra=somevar)
```

значения `42`, `314` и `somevar` являются аргументами.

2.2.9 Почему измененный список „y“ также изменил список „x“?

Если ты напишешь код:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

Возможно, вам станет интересно, почему добавление элемента к `y` также изменило `x`.

Есть два фактора, которые дают этот результат:

1) переменные - это просто имена, которые ссылаются на объекты. Выполнение `y = x` не создает копию списка - создает новую переменную `y`, которая ссылается на тот же объект `x` на который он ссылается. Это означает, что существует только один объект (список), и `x` и `y` ссылаются на него.

2) списки являются *изменяемыми*, что означает, что вы можете изменить их содержимое.

После вызова `append()` содержимое изменяемого объекта изменилось с `[]` на `[10]`. Так как обе переменные ссылаются на один и тот же объект, то при использовании любого имени происходит доступ к измененному значению `[10]`.

Если вместо этого назначить неизменяемый объект `x`:

```
>>> x = 5  # ints являются неизменяемыми
>>> y = x
>>> x = x + 1  # 5 нельзя изменить, мы создаем здесь новый объект
>>> x
6
>>> y
5
```

мы видим, что в данном случае `x` и `y` уже не равны. Это потому, что целые числа *неизменны*, и когда мы выполняем `x = x + 1` мы не изменяем int `5`, увеличивая его значение; вместо этого мы создаем новый объект (`6` int) и присваиваем его `x` (то есть изменяем, на какой объект ссылается `x`). После этого назначения мы имеем два объекта (ints `6` и `5`) и две переменные, которые ссылаются на них (`x` теперь относится к `6`, но `y` все еще относится к `5`).

Некоторые операции (например `y.append(10)` и `y.sort()`) мутируют объект, тогда как поверхностно похожие операции (например `y = y + [10]` и `sorted(y)`) создают новый объект. Как правило, в Python (и во всех случаях в стандартной библиотеке) метод, мутирующий объект, будет возвращать `None`, чтобы избежать путаницы двух типов операций. Так что если вы ошибочно напишете `y.sort()` думая, что это вернет вам отсортированную копию `y`, вы вместо этого получите `None`, что, вероятно, вызовет в вашей программе легко диагностируемую ошибку.

Однако существует один класс операций, где одна и та же операция иногда имеет разное поведение с различными типами: дополненные операторы назначения. Например, `+=` изменяет списки, но не кортежи или инты (`a_list += [1, 2, 3]` эквивалентен `a_list.extend([1, 2, 3])` и изменяет `a_list`, тогда как `some_tuple += (1, 2, 3)` и `some_int += 1` создают новые объекты).

Другими словами:

- Если у нас есть изменяемый объект (`list`, `dict`, `set` и т. д.), мы можем использовать некоторые конкретные операции, чтобы изменить его, и все переменные, которые ссылаются на него, увидят изменение.
- Если у нас есть неизменяемый объект (`str`, `int`, `tuple` и т. д.), все переменные, которые ссылаются на него, всегда будут видеть одно и тот же значение, но операции, которые преобразуют это значение в новое значение, всегда возвращают новый объект.