# Первые шаги

Давайте посмотрим, как создать традиционную программу «Hello World» на Python. Это научит вас писать, сохранять и выполнять программы на Python.

Существует два способа запуска программ на Python: использование интерактивного приглашения интерпретатора и использование файла с текстом программы. Сейчас мы увидим, как пользоваться обоими методами.

## Использование командной строки интерпрета- тора

Откройте окно терминала (как было описано в главе [*Установка*](#_bookmark27)) и запустите интерпрета- тор Python, введя команду python3 и нажав Enter.

Пользователи Windows могут запустить интерпретатор в командной строке, если уста- новили переменную PATH надлежащим образом. Чтобы открыть командную строку в Windows, зайдите в меню «Пуск» и нажмите «Выполнить…». В появившемся диалого- вом окне введите «cmd» и нажмите Enter; теперь у вас будет всё необходимое для начала работы с python в командной строке DOS.

Если вы используете IDLE, нажмите «Пуск» «Программы» «Python 3.0» «IDLE (Python GUI)».

*→ → →*

Как только вы запустили python3, вы должны увидеть >>> в начале строки, где вы можете что-то набирать. Это и называется *командной строкой интерпретатора Python*.

Теперь введите print('Hello World') и нажмите клавишу Enter. В результате должны появиться слова «Hello World».

Вот пример того, что вы можете увидеть на экране, если будете использовать компьютер с Mac OS X. Информация о версии Python может отличаться в зависимости от компьютера, но часть, начинающаяся с приглашения (т.е. от >>> и далее) должна быть одинаковой на всех операционных системах.

$ python3

Python 3.3.0 (default, Oct 22 2012, 12:20:36)

[GCC 4.2.1 Compatible Apple Clang 4.0 ((tags/Apple/clang-421.0.60))] on darwin Type "help", "copyright", "credits" or "license" for more information.

>>> print('hello world') hello world

>>>

Обратите внимание, что Python выдаёт результат работы строки немедленно! Вы только что ввели одиночный «оператор» Python. print используется для того, чтобы (что неуди- вительно[1](#_bookmark36) ) напечатать любое переданное в него значение. В данном случае мы передали в него текст «Hello World», который и был напечатан на экране.

**Совет:** Как выйти из командной строки интерпретатора

Если вы используете IDLE или оболочку GNU/Linux или BSD, вы можете выйти из команд- ной строки интерпретатора нажатием Ctrl-D или введя команду exit() (примечание: не забудьте написать скобки, «()»), а затем нажав клавишу Enter. Если вы используете командную строку Windows, нажмите Ctrl-Z, а затем нажмите клавишу Enter.

## Выбор редактора

Поскольку мы не можем набирать программу в командной строке интерпретатора каж- дый раз, когда нам нужно что-то запустить, нам понадобится сохранять программы в файлах, чтобы потом иметь возможность запускать их сколько угодно раз.

Прежде чем приступить к написанию программ на Python в файлах, нам нужен редактор для работы с файлами программ. Выбор редактора крайне важен. Подходить к выбору ре- дактора следует так же, как и к выбору личного автомобиля. Хороший редактор поможет вам легко писать программы на Python, делая ваше путешествие более комфортным, а также позволяя быстрее и безопаснее достичь вашей цели.

Одно из самых основных требований – это *подсветка синтаксиса*, когда разные элементы программы на Python раскрашены так, чтобы вы могли легко *видеть* вашу программу и ход её выполнения.

Если вы не знаете, с чего начать, я бы порекомендовал воспользоваться программой [Komodo Edit](http://www.activestate.com/komodo-edit/downloads), которая доступна для Windows, Mac OS X и GNU/Linux.

Если вы пользуетесь Windows, **Не используйте Блокнот** – это плохой выбор, посколь- ку он не обладает функцией подсветки синтаксиса, а также не позволяет автоматически

1 «print» – *англ.* «Печатать» (*прим. перев.*)

вставлять отступы, что очень важно в нашем случае, как мы увидим позже. Хорошие ре- дакторы, как Komodo Edit, позволяют делать это автоматически.

Опытные программисты, должно быть, уже используют [Vim](http://www.vim.org/) или [Emacs](http://www.gnu.org/software/emacs/). Не стоит даже и говорить, что это два наиболее мощных редактора, и вы только выиграете от их ис- пользования для написания программ на Python. Лично я пользуюсь ими обоими для большинства своих программ, и даже написал [книгу о Vim](http://www.swaroopch.com/notes/vim). Я настоятельно рекомендую вам решиться и потратить время на изучение Vim или Emacs, поскольку это будет прино- сить вам пользу долгие годы. Однако, как я уже писал выше, новички могут пока просто остановиться на Komodo Edit и сосредоточиться на изучении Python, а не текстового ре- дактора.

Я повторюсь ещё раз: обязательно выберите подходящий редактор – это сделает написа- ние программ на Python более простым и занимательным.

Для пользователей Vim

Существует хорошее введение в [использование Vim как мощного IDE для](http://blog.sontek.net/turning-vim-into-a-modern-python-ide) [Python, автор – John M Anderson](http://blog.sontek.net/turning-vim-into-a-modern-python-ide). Также я рекомендую [плагин jedi-vim](https://github.com/davidhalter/jedi-vim) и [мой](https://github.com/swaroopch/dotvim) [собственный конфигурационный файл](https://github.com/swaroopch/dotvim).

Для пользователей Emacs

Существует хорошее введение в [использование Emacs как мощного IDE для](http://www.enigmacurry.com/2008/05/09/emacs-as-a-powerful-python-ide/) [Python, автор – Ryan McGuire](http://www.enigmacurry.com/2008/05/09/emacs-as-a-powerful-python-ide/). Также я рекомендую [Конфигурацию dotemacs](https://github.com/ghoseb/dotemacs) [от BG](https://github.com/ghoseb/dotemacs).

## Использование программных файлов

А теперь давайте вернёмся к программированию. Существует такая традиция, что какой бы язык программирования вы ни начинали учить, первой вашей программой должна быть программа «Привет, Мир!». Это программа, которая просто выводит надпись «При- вет, Мир!». Как сказал Simon Cozens[2](#_bookmark38), это «традиционное заклинание богов программи- рования, которое поможет вам лучше изучить язык».

Запустите выбранный вами редактор, введите следующую программу и сохраните её под именем helloworld.py .

Если вы пользуетесь Komodo Edit, нажмите «Файл» «Новый» «Новый файл», вве- дите строку:

*→ →*

print('Привет, Мир!')

В Komodo Edit нажмите «Файл» *→* «Сохранить» для сохранения файла.

2 Автор восхитительной книги «Beginning Perl»

Куда сохранить файл? В любую папку, расположение которой вы знаете. Если вы не пони- маете, что это значит, то создайте новую папку и используйте её для всех ваших программ на Python:

* C:\py в Windows
* /tmp/py в GNU/Linux
* /tmp/py в Mac OS X

Чтобы создать папку, воспользуйтесь командой mkdir в терминале. Например, mkdir / tmp/py.

**Важно:** Не забывайте указывать расширение файла .py. Например, «file.py».

В Komodo Edit нажмите «Инструменты» «Запуск команды», наберите python3 helloworld.py и нажмите «Выполнить». Вы должны увидеть вывод, показанный на скриншоте ниже.

*→*

Но всё-таки лучше редактировать программу в Komodo Edit, а запускать в терминале:

* + 1. Откройте терминал, как описано в главе [*Установка*](#_bookmark27).
    2. Перейдите в каталог, в котором вы сохранили файл. Например, cd /tmp/py.
    3. Запустите программу, введя команду python3 helloworld.py. Вывод программы показан ниже.

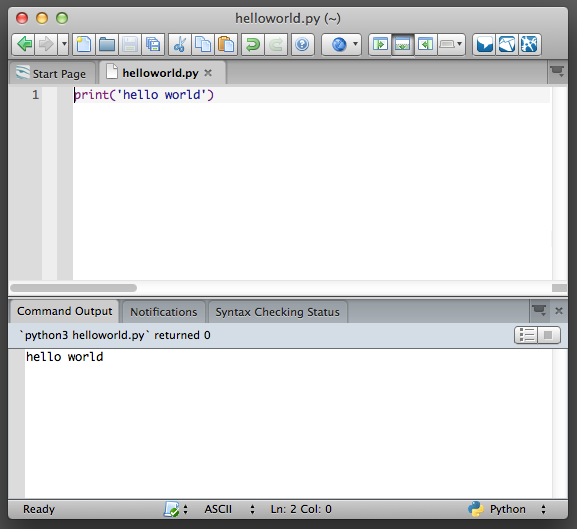
$ python3 helloworld.py Привет, Мир!

Если у вас получился такой же вывод, поздравляю! – вы успешно выполнили вашу первую программу на Python. Вы только что совершили самый сложный шаг в обучении программированию, заключающийся в написании своей первой программы!

Если вы получите сообщение об ошибке, введите вышеуказанную программу *в точно- сти* так, как показано здесь, и запустите снова. Обратите внимание, что Python различает регистр букв, то есть print – это не то же самое, что Print (обратите внимание на букву p в нижнем регистре в первом случае и на букву P в верхнем регистре во втором). Также убедитесь, что перед первым символом в строке нет пробелов или символов табуляции – позже мы увидим, почему это важно.

#### Как это работает

Программа на Python состоит из *выражений*. В нашей первой программе име- ется всего лишь одно выражение. В этом выражении мы вызываем функцию print, которая просто выводит текст 'Привет, Мир!'. О функциях мы узна- ем в [*одной из последующих глав*](#_bookmark83), а пока вам достаточно понять, что всё, что вы



укажете в скобках, будет выведено на экран. В данном примере мы указали

'Привет, Мир!'.

6.3.1 Исполнимые программы на Python

Это касается только пользователей GNU/Linux и Unix, но пользователям Windows тоже будет полезно об этом знать.

Каждый раз, когда нам нужно запустить программу на Python, нам приходится в явном виде запускать python3 foo.py. Но почему бы нам не запускать её точно так же, как и все другие программы? Этого можно достичь при помощи так называемого *hashbang*.

Добавьте строку, указанную ниже, в *самое начало* вашей программы:

*#!/usr/bin/env python3*

Теперь ваша программа должна выглядеть так:

*#!/usr/bin/env python3*

print('Привет, Мир!')

Теперь необходимо установить программе атрибут исполнимости, используя команду

*chmod*, а затем *выполнить* программу.

Команда chmod здесь используется для изменения режима файла[3](#_bookmark40) добавлением атрибута исполнимости для всех пользователей в системе[4](#_bookmark41).

$ chmod a+x helloworld.py

После этого мы можем запускать программу напрямую, потому что наша операционная система запустит /usr/bin/env, который, в свою очередь, найдёт Python 3, а значит, смо- жет запустить наш файл.

$ ./helloworld.py Привет, Мир!

Здесь «./» обозначает, что программа находится в текущем каталоге.

Ради интереса можете даже переименовать файл в просто «helloworld» и запустить его как ./helloworld, и это также сработает, поскольку система знает, что запускать про- грамму нужно интерпретатором, положение которого указано в первой строке файла про- граммы.

Но до сих пор мы могли выполнять свою программу только если знали полный путь к ней. А что, если нам нужно запускать эту программу из любого каталога? Это можно организовать, расположив свою программу в одном из каталогов, перечисленных в пере- менной окружения PATH.

При попытке запуска какой-либо программы система ищет её в каталогах, перечислен- ных в переменной окружения PATH, и запускает. Таким образом, мы можем сделать про- грамму доступной из любого места, скопировав её в один из каталогов, перечисленных в PATH.

$ echo $PATH

/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin

$ cp helloworld.py /home/swaroop/bin/helloworld

$ helloworld Привет, Мир!

3 **ch**ange **mod**e – англ. «изменить режим» (*прим. перев.*)

4 В указанной команде буква «**a**» взята из слова «**a**ll» (*англ.* «все»), а буква «**x**» – из слова «e**x**ecute» (*англ.*

«исполнять») – *прим. перев.*

Мы можем вывести на экран значение переменной PATH при помощи команды echo, добавив перед именем переменной символ $, чтобы указать оболочке, что мы хотим по- лучить значение этой переменной. Мы видим, что /home/swaroop/bin – один из ка- талогов в переменной PATH, где *swaroop* – это имя пользователя, которое я использую в своей системе. В вашей системе, скорее всего, будет аналогичный каталог для вашего пользователя.

Вы также можете добавить какой-либо каталог к переменной PATH – это можно сделать, выполнив PATH=$PATH:/home/swaroop/mydir, где '/home/swaroop/mydir' – это ка- талог, который я хочу добавить к переменной PATH.

Этот метод полезен для написания сценариев, которые будут доступны для запуска в лю- бой момент из любого места. По сути, это равносильно созданию собственных команд, как cd или любой другой, которые часто используются в терминале GNU/Linux или пригла- шении DOS.

**Примечание:** Когда речь идёт о Python, слова «программа» или «сценарий (скрипт)» обозначают одно и то же.

## Получение помощи

Для быстрого получения информации о любой функции или операторе Python служит встроенная функция help. Это особенно удобно при использовании командной строки интерпретатора. К примеру, выполните help(print) – это покажет справку по функции print, которая используется для вывода на экран.

**Примечание:** Для выхода из справки нажмите q.

Аналогичным образом можно получить информацию почти о чём угодно в Python. При помощи функции help() можно даже получить описание самой функции help!

Если вас интересует информация об операторах, как например, return, их необходимо указывать в кавычках (например, help('return')), чтобы Python понял, чего мы хотим.

## Резюме

Теперь вы умеете с лёгкостью писать, сохранять и запускать программы на Python.

И поскольку сейчас вы уже используете Python, давайте узнаем больше о его основных принципах.

# Основы

Просто напечатать «Привет, Мир!» недостаточно, верно? Вы хотите сделать больше – вы хотите ввести что-то в программу, обработать и получить нечто на выходе. В Python это можно организовать при помощи констант и переменных, а также некоторыми другими способами, которые будут рассмотрены в этой главе.

## Комментарии

*Комментарии* – это то, что пишется после символа #, и представляет интерес лишь как заметка для читающего программу.

Например:

print('Привет, Мир!) # print -- это функция

или:

*# print -- это функция*

print('Привет, Мир!)

Старайтесь в своих программах писать как можно больше полезных комментариев, объ- ясняющих:

* предположения;
* важные решения;
* важные детали;
* проблемы, которые вы пытаетесь решить;
* проблемы, которых вы пытаетесь избежать и т.д.

[Текст программы говорит о том, КАК, а комментарии должны объяснять, ПОЧЕМУ](http://www.codinghorror.com/blog/2006/12/code-tells-you-how-comments-tell-you-why.html).

Это будет полезно для тех, кто будет читать вашу программу, так как им легче будет по- нять, что программа делает. Помните, что таким человеком можете оказаться вы сами через полгода!

## Литеральные константы

Примером литеральной константы может быть число, например, 5, 1.23, 9.25e-3 или что-нибудь вроде 'Это строка' или "It's a string!". Они называются литеральны- ми, потому что они «буквальны»[1](#_bookmark49) – вы используете их значение буквально. Число 2 все- гда представляет само себя и ничего другого – это «константа», потому что её значение нельзя изменить. Поэтому всё это называется литеральными константами.

## Числа

Числа в Python бывают трёх типов: целые, с плавающей точкой и комплексные.

* Примером целого числа может служить 2.
* Примерами чисел с плавающей точкой (или «плавающих» для краткости) могут быть 3.23 и 52.3E-4. Обозначение E показывает степени числа 10. В данном случае 52.3E-4 означает 52.3 \* 10-4.
* Примеры комплексных чисел: (-5+4j) и (2.3 - 4.6j)

#### Замечание для опытных программистов

Нет отдельного типа „long int“ (длинное целое). Целые числа по умолчанию могут быть произвольной длины.

## Строки

Строка – это *последовательность символов*. Чаще всего строки – это просто некоторые наборы слов.

Слова могут быть как на английском языке, так и на любом другом, поддерживаемом стандартом Unicode, что означает [почти на любом языке мира](http://www.unicode.org/cldr/data/charts/supplemental/languages_and_scripts.html).

#### Замечание для опытных программистов

В Python 3 нет ASCII-строк, потому что Unicode является надмножеством (включает в се- бя) ASCII. Если необходимо получить строку строго в кодировке ASCII, используйте str. encode("ascii"). Подробнее смотрите в [обсуждении этого вопроса на StackOverflow](http://stackoverflow.com/questions/175240/how-do-i-convert-a-files-format-from-unicode-to-ascii-using-python#175270). По умолчанию все строки в Unicode.

1 «literal» – *англ.* «буквальный»; вспомните «литера» (*син.* «буква»). (*прим. перев.*)

Я могу с уверенностью сказать, что вы будете использовать строки почти в каждой вашей программе на Python. Поэтому уделите внимание тому, как работать со строками в Python.

* + 1. Одинарные кавычки

Строку можно указать, используя одинарные кавычки, как например, 'Фраза в кавычках'. Все пробелы и знаки табуляции сохранятся, как есть.

### Двойные кавычки

Строки в двойных кавычках работают точно так же, как и в одинарных. Например,

"What's your name?".

### Тройные кавычки

Можно указывать «многострочные» строки с использованием тройных кавычек (""" или '''). В пределах тройных кавычек можно свободно использовать одинарные и двойные кавычки. Например:

*'''Это многострочная строка. Это её первая строка. Это её вторая строка.*

*"What's your name?", - спросил я. Он ответил: "Bond, James Bond." '''*

### Строки неизменяемы

Это означает, что после создания строки её больше нельзя изменять. На первый взгляд это может показаться недостатком, но на самом деле это не так. Впоследствии на примере разных программ мы увидим, почему это не является ограничением.

### Объединение строковых констант

Если расположить рядом две строковых константы, Python автоматически их объединит. Например, 'What\'s ' 'your name?' автоматически преобразуется в "What's your name?".

#### Замечание для программистов на C/C++

В Python нет отдельного типа данных char (символ). В нём нет нужды, и я уверен, что вы не будете по нему скучать.

#### Замечание для программистов на Perl/PHP

Помните, что строки в двойных кавычках и в одинарных эквивалентны, и ничем друг от друга не отличаются.

### Метод format

Иногда бывает нужно составить строку на основе каких-либо данных. Вот здесь-то и при- гождается метод format().

Сохраните следующие строки в файл str\_format.py:

age = 26

name = 'Swaroop'

print('Возраст *{0}* -- *{1}* лет.'.format(name, age)) print('Почему *{0}* забавляется с этим Python?'.format(name))

Вывод:

$ python str\_format.py Возраст Swaroop -- 26 лет.

Почему Swaroop забавляется с этим Python?

#### Как это работает:

В строку могут быть включены определённые обозначения, а впоследствии может быть вызван метод format для замещения этих обозначений соответ- ствующими аргументами.

Взгляните на первый случай применения обозначений, где мы пишем {0}, и это соответствует переменной name, являющейся первым аргументом метода format. Аналогично, второе обозначение {1} соответствует переменной age, являющейся вторым аргументом метода format. Заметьте, что Python начи- нает отсчёт с 0, поэтому первая позиция – номер 0, вторая – номер 1 и т.д.

Заметьте, мы ведь могли добиться того же самого результата и объединени- ем строк: 'Возраст' + name + ' -- ' + str(age) + ' лет.', однако вы сами видите, как это некрасиво, и как легко в таком случае допустить ошибку.

Во-вторых, преобразование в строку делается методом format автоматически, в отличие от явного преобразования в нашем примере. В-третьих, используя метод format, мы можем изменить сообщение, не затрагивая используемых переменных, и наоборот.

На всякий случай имейте в виду, что цифры здесь не обязательны. Можно было бы просто написать:

age = 26

name = 'Swaroop'

print('Возраст *{}* -- *{}* лет.'.format(name, age)) print('Почему *{}* забавляется с этим Python?'.format(name))

и получить такой же результат, как и ранее.

В методе format Python помещает значение каждого аргумента в обозначенное место. Могут быть и более детальные обозначения, как то:

**>>>** *# десятичное число (.) с точностью в 3 знака для плавающих:*

**>>>** '*{0:.3}*'.format(1/3) '0.333'

**>>>** *# заполнить подчёркиваниями (\_) с центровкой текста (^) по ширине 11:*

**>>>** '*{0:\_^11}*'.format('hello') ' hello '

**>>>** *# по ключевым словам:*

**>>>** '*{name}* написал *{book}*'.format(name='Swaroop', book='A Byte of Python') 'Swaroop написал A Byte of Python'

Детально такие обозначения форматов описаны в Предложении по расширению Python

[**PEP 3101**](https://www.python.org/dev/peps/pep-3101).

## Переменные

Использование одних лишь литеральных констант может скоро наскучить – нам ведь нужен способ хранения любой информации и манипулирования ею. Вот здесь на сце- ну выходят *переменные*. Слово «переменные» говорит само за себя – их значение может меняться, а значит, вы можете хранить в переменной всё, что угодно. Переменные – это просто области памяти компьютера, в которых вы храните некоторую информацию. В отличие от констант, к такой информации нужно каким-то образом получать доступ, по- этому переменным даются имена.

## Имена идентификаторов

Переменные – это частный случай идентификаторов. *Идентификаторы* – это имена, при- своенные *чему-то* для его обозначения. При выборе имён для идентификаторов необхо- димо соблюдать следующие правила:

* Первым символом идентификатора должна быть буква из алфавита (символ ASCII в верхнем или нижнем регистре, или символ Unicode), а также символ подчёркивания («\_»).
* Остальная часть идентификатора может состоять из букв (символы ASCII в верхнем или нижнем регистре, а также символы Unicode), знаков подчёркивания («\_») или цифр (0-9).
* Имена идентификаторов чувствительны к регистру. Например, myname и myName – это **не** одно и то же. Обратите внимание на «n» в нижнем регистре в первом случае и «N» в верхнем во втором.
* Примеры *допустимых* имён идентификаторов: i, my\_name, name\_23, a1b2\_c3 и

любые\_символы\_utf8\_δξѪђёўЩӆΞέά.

* Примеры *недопустимых* имён идентификаторов: 2things, здесь есть пробелы, my-name, >a1b2\_c3 и "это\_в\_кавычках".

## Типы данных

Переменные могут хранить значения разных типов, называемых **типами данных**. Ос- новными типами являются числа и строки, о которых мы уже говорили. В дальнейших главах мы увидим, как создавать свои собственные типы при помощи [*классов*](#_bookmark144).

## Объекты

Помните, Python рассматривает всё, что есть в программе, как *объекты*. Имеется в виду, в самом общем смысле. Вместо того, чтобы говорить «*нечто*», мы говорим «*объект*».

#### Замечание для программистов в объектно-ориентированном стиле

Python строго объектно ориентирован в том смысле, что объектом является всё, включая числа, строки и функции.

Сейчас мы увидим, как использовать переменные наряду с константами. Сохраните сле- дующий пример и запустите программу.

#### Как писать программы на Python

Впредь стандартная процедура сохранения и запуска программы на Python будет выгля- деть так:

1. Откройте ваш любимый редактор, например Komodo Edit.
2. Введите текст программы из примера.
3. Сохраните его в файл, указав его имя в комментарии. Я следую правилу сохранять все программы на Python с расширением .py.
4. Запустите интерпретатор командой python3 program.py. Кроме того, вы можете сделать программу [*исполнимой*](#_bookmark39), как объяснялось ранее.
   * 1. Пример: Использование переменных и констант

*# Имя файла : var.py*

i = 5 print(i) i = i + 1 print(i)

s = '''Это многострочная строка. Это вторая её строчка.''' print(s)

#### Вывод:

$ python var.py 5

6

Это многострочная строка. Это вторая её строчка.

**Как это работает:**

Вот как эта программа работает. Сперва мы присваиваем значение констан- ты 5 переменной i, используя оператор присваивания (=). Эта строка назы- вается предложением и указывает, что должно быть произведено некоторое действие, и в данном случае мы связываем имя переменной i со значением

1. Затем мы печатаем значение i, используя функцию print, которая просто печатает значение переменной на экране.

Далее мы добавляем 1 к значению, хранящемуся в i и сохраняем его там. По- сле этого мы печатаем его и получаем значение 6, что неудивительно.

Аналогичным образом мы присваиваем строковую константу переменной s, после чего печатаем её.

#### Замечание для программистов на статических языках программирования

Переменные используются простым присваиванием им значений. Никакого предвари- тельного объявления или определения типа данных не требуется/применяется.

## Логические и физические строки

Физическая строка – это то, что вы *видите*, когда набираете программу. Логическая стро- ка – это то, что *Python видит* как единое предложение. Python неявно предполагает, что каждой *физической строке* соответствует *логическая строка*.

Примером логической строки может служить предложение print('Привет, Мир!') – если оно на одной строке (как вы видите это в редакторе), то эта строка также соответ- ствует физической строке.

Python неявно стимулирует использование по одному предложению на строку, что об- легчает чтение кода.

Чтобы записать более одной логической строки на одной физической строке, вам придёт- ся явно указать это при помощи точки с запятой (;), которая отмечает конец логической строки/предложения. Например,

i = 5 print(i)

то же самое, что

i = 5;

print(i);

и то же самое может быть записано в виде

i = 5; print(i);

или даже

i = 5; print(i)

Однако я **настоятельно рекомендую** вам придерживаться **написания одной логической строки в каждой физической строке**. Таким образом вы можете обойтись совсем без точ- ки с запятой. Кстати, я *никогда* не использовал и даже не встречал точки с запятой в про- граммах на Python.

Можно использовать более одной физической строки для логической строки, но к этому следует прибегать лишь в случае очень длинных строк. Пример написания одной логиче- ской строки, занимающей несколько физических строк, приведён ниже. Это называется **явным объединением строк**.

s = 'Это строка. **\**

Это строка продолжается.' print(s)

Это даст результат:

Это строка. Это строка продолжается.

Аналогично,

print\ (i)

то же самое, что и

print(i)

Иногда имеет место неявное подразумевание, когда использование обратной косой чер- ты не обязательно. Это относится к случаям, когда в логической строке есть открываю- щаяся круглая, квадратная или фигурная скобка, но нет закрывающейся. Это называется **неявным объединением строк**. Вы сможете увидеть это в действии в программах с ис- пользованием [*списков*](#_bookmark115) в дальнейших главах.

## Отступы

В Python пробелы важны. Точнее, **пробелы в начале строки важны**. Это называется **от- ступами**. Передние отступы (пробелы и табуляции) в начале логической строки исполь- зуются для определения уровня отступа логической строки, который, в свою очередь, ис- пользуется для группировки предложений.

Это означает, что предложения, идущие вместе, **должны** иметь одинаковый отступ. Каж- дый такой набор предложений называется **блоком**. В дальнейших главах мы увидим при- меры того, насколько важны блоки.

Вы должны запомнить, что неправильные отступы могут приводить к возникновению ошибок. Например:

i = 5

print('Значение составляет ', i) *# Ошибка! Пробел в начале строки*

print('Я повторяю, значение составляет ', i)

Когда вы запустите это, вы получите следующую ошибку:

File "whitespace.py", line 4

print('Значение составляет ', i) # Ошибка! Пробел в начале строки

^

IndentationError: unexpected indent

Обратите внимание на то, что в начале второй строки есть один пробел. Ошибка, отобра- жённая Python, говорит нам о том, что синтаксис программы неверен, т.е. программа не была написана по правилам. Для вас же это означает, что *вы не можете начинать новые блоки предложений где попало* (кроме основного блока по умолчанию, который использу- ется на протяжении всей программы, конечно). Случаи, в которых вы можете использо- вать новые блоки, будут подробно описаны в дальнейших главах, как например, в [*главе*](#_bookmark67)

[*«Поток команд»*](#_bookmark67).

#### Как отступать

**Не** смешивайте пробелы и символы табуляции в отступах, поскольку не на всех плат- формах это работает корректно. Я *настоятельно рекомендую* вам использовать *одиночную табуляцию* или *четыре пробела* для каждого уровня отступа.

Выберите какой-нибудь один из этих стилей отступа. Но что ещё более важно, это ис- пользовать выбранный стиль **постоянно**, а также соблюдать стиль редактируемых вами файлов. Т.е. когда вы пишете новый файл, используйте *только* один ваш любимый стиль, а если в редактируемом вами файле для отступов уже используются, скажем, символы та- буляции, то и вы используйте в этом файле символы табуляции для отступов.

Хорошие редакторы, такие как Komodo Edit, будут делать это автоматически.

#### Замечание для программистов на статических языках программирования

Python всегда будет использовать отступы для выделения блоков и никогда не будет ис- пользовать скобки. Введите from future import braces, чтобы узнать больше.

## Резюме

Теперь, когда мы прошли через множество жизненно важных мелочей, можно перейти к более интересным вещам – таким как управляющие конструкции. Но сначала как следует освойтесь с прочитанным в настоящей главе.

# Операторы и выражения

Большинство предложений (логических строк) в программах содержат *выражения*. Про- стой пример выражения: 2 + 3. Выражение можно разделить на операторы и операнды.

*Операторы* – это некий функционал, производящий какие-либо действия, который мо- жет быть представлен в виде символов, как например +, или специальных зарезервиро- ванных слов. Операторы могут производить некоторые действия над данными, и эти дан- ные называются *операндами*. В нашем случае 2 и 3 – это операнды.

## Операторы

Кратко рассмотрим операторы и их применение:

Обратите внимание, вычислить значения выражений, данных в примерах, можно так- же используя интерпретатор интерактивно. Например, для проверки выражения 2 + 3 воспользуйтесь интерактивной командной строкой интерпретатора Python:

**>>>** 2 + 3

5

**>>>** 3 \* 5

15

#### Операторы и их применение

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Название | Объяснение | Примеры |
| + | Сложение | Суммирует два  объекта | 3 + 5 даст 8; 'a' + 'b' даст 'ab' |
| - | Вычита-  ние | Даёт разность  двух чисел; если первый операнд отсутствует, он считается равным нулю | -5.2 даст отрицательное число, а 50 -  24 даст 26. |

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Название | Объяснение | Примеры |
| \* | Умноже-  ние | Даёт  произведение двух чисел или возвращает строку, повторённую заданное число раз. | 2 \* 3 даст 6. 'la' \* 3 даст 'lalala'. |
| \*\* | Возведе-  ние в степень | Возвращает  число х, возведённое в степень y | 3 \*\* 4 даст 81 (т.е. 3 \* 3 \* 3 \* 3) |
| / | Деление | Возвращает  частное от деления x на y | 4 / 3 даст 1.3333333333333333. |
| // | Целочис-  ленное деление | Возвращает  неполное частное от деления | 4 // 3 даст 1. -4 // 3 даст -2. |
| % | Деление  по модулю | Возвращает  остаток от деления | 8 % 3 даст 2. -25.5 % 2.25 даст 1.5. |
| << | Сдвиг  влево | Сдвигает биты  числа влево на заданное количество позиций. (Любое число в памяти компьютера представлено в виде битов - или двоичных чисел, т.е. 0 и 1) | 2 << 2 даст 8. В двоичном виде 2  представляет собой 10. Сдвиг влево на 2 бита даёт 1000, что в десятичном виде означает 8. |
| >> | Сдвиг  вправо | Сдвигает биты  числа вправо на заданное число позиций. | 11 >> 1 даст 5. В двоичном виде 11  представляется как 1011, что будучи смещённым на 1 бит вправо, даёт 101, а это, в свою очередь, не что иное как  десятичное 5 |
| & | Побито-  вое И | Побитовая  операция И над числами | 5 & 3 даёт 1. |

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Название | Объяснение | Примеры |
| | | Побито-  вое ИЛИ | Побитовая  операция ИЛИ над числами | 5 | 3 даёт 7 |
| ^ | Побито-  вое ИСКЛЮ- ЧИТЕЛЬ- НО  ИЛИ | Побитовая  операция ИС- КЛЮЧИТЕЛЬНО ИЛИ | 5 ^ 3 даёт 6 |
| ~ | Побито-  вое НЕ | Побитовая  операция НЕ для числа x соответствует  -(x+1) | ~5 даёт -6. |
| < | Меньше | Определяет,  верно ли, что x меньше y. Все операторы сравнения возвращают True или False[1](#_bookmark60).  Обратите внимание на заглавные буквы в этих  словах. | 5 < 3 даст False,  а 3 < 5 даст True.  Можно составлять произвольные цепочки сравнений: 3 < 5 < 7 даёт True. |
| > | Больше | Определяет,  верно ли, что x  больше y | 5 > 3 даёт True. Если оба операнда -  числа, то перед сравнением они оба преобразуются к одинаковому типу. В противном случае всегда возвращается  False. |
| <= | Меньше  или равно | Определяет,  верно ли, что x меньше или равно y | x = 3; y = 6; x <= y даёт True. |
| >= | Больше  или равно | Определяет,  верно ли, что x больше или равно y | x = 4; y = 3; x >= 3 даёт True. |

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Название | Объяснение | Примеры |
| == | Равно | Проверяет,  одинаковы ли объекты | x = 2; y = 2; x == y даёт True. x =  'str'; y = 'stR'; x == y даёт False. x = 'str'; y = 'str'; x == y даёт True. |
| != | Не равно | Проверяет,  верно ли, что объекты не равны | x = 2; y = 3; x != y даёт True. |
| not | Логиче-  ское НЕ | Если x равно  True, оператор вернёт False. Если же x равно False, получим True. | x = True; not x даёт False. |
| and | Логиче-  ское И | x and y даёт  False, если x равно False , в противном случае возвращает значение y | x = False; y = True; x and y возвращает False, поскольку x равно False. В этом случае Python не станет проверять значение y, так как уже знает, что левая часть выражения „and“ равняется False, что подразумевает, что и всё выражение в целом будет равно False, независимо от значений всех остальных операндов. Это называется укороченной оценкой булевых (логических) выражений. |
| or | Логиче-  ское ИЛИ | Если x равно  True, в результате получим True, в противном случае получим  значение y | x = True; y = False; x or y даёт  True. Здесь также может производиться укороченная оценка выражений. |

1 «True» - *англ.* «Верно (Правда)»; «False» - *англ.* «Ошибочно (Ложь)». (*прим. перев.*)

* + 1. Краткая запись мат. операций и присваивания

Зачастую результат проведения некой математической операции необходимо присвоить переменной, над которой эта операция производилась. Для этого существуют краткие формы записи выражений:

Вы можете записать:

a = 2; a = a \* 3

в виде:

a = 2; a \*= 3

Обратите внимание, что выражения вида «переменная = переменная операция выражение» принимает вид «переменная операция = выражение».

## Порядок вычисления

Если имеется выражение вида 2 + 3 \* 4, что производится раньше: сложение или умно- жение? Школьный курс математики говорит нам, что умножение должно производиться в первую очередь. Это означает, что оператор умножения имеет более высокий приори- тет, чем оператор сложения.

Следующая таблица показывает приоритет операторов в Python, начиная с самого низ- кого (самое слабое связывание) и до самого высокого (самое сильное связывание). Это означает, что в любом выражении Python сперва вычисляет операторы и выражения, рас- положенные внизу таблицы, а затем операторы выше по таблице.

Эта таблица взята из [Справочника по языку Python (англ.)](http://docs.python.org/py3k/reference/expressions.html#summary) и приводится здесь для пол- ноты описания. На практике лучше использовать скобки для группировки операторов и операндов, чтобы в явном виде указать порядок вычисления выражений. Заодно это об- легчит чтение программы. Более подробно см. в разделе [*Изменение порядка вычисления*](#_bookmark63)ниже.

#### Приоритет операторов

|  |  |
| --- | --- |
| Оператор | Описание |
| lambda | лямбда-выражение |
| or | Логическое «ИЛИ» |
| and | Логическое «И» |
| not x | Логическое «НЕ» |
| in, not in | Проверка принадлежности |
| is, is not | Проверка тождественности |
| <, <=, >, >=, !=, == | Сравнения |
| | | Побитовое «ИЛИ» |
| ^ | Побитовое «ИСКЛЮЧИТЕЛЬНО ИЛИ» |
| & | Побитовое «И» |
| <<, >> | Сдвиги |
| +, - | Сложение и вычитание |
| \*, /, //, % | Умножение, деление, целочисленное деление и остаток от де-  ления |
| +x, -x | Положительное, отрицательное |
| ~x | Побитовое НЕ |
| \*\* | Возведение в степень |
| x.attribute | Ссылка на атрибут |
| x[индекс] | Обращение по индексу |
| x[индекс1:индекс2] | Вырезка |
| f(аргументы ...) | Вызов функции |
| (выражения, ...) | Связка или кортеж[2](#_bookmark62) |
| [выражения, ...] | Список |
| {ключ:данные,  ...} | Словарь |

Операторы, о которых мы не упомянули, будут объяснены в дальнейших главах.

В этой таблице операторы с *равным приоритетом* расположены в одной строке. Напри- мер, + и - имеют равный приоритет.

2 «tuple» - *англ.* «кортеж» (*прим. перев.*)

## Изменение порядка вычисления

Для облегчения чтения выражений можно использовать скобки. Например, 2 + (3 \* 4) определённо легче понять, чем 2 + 3 \* 4, которое требует знания приоритета операто- ров. Как и всё остальное, скобки нужно использовать разумно (не перестарайтесь) и избе- гать излишних, как в (2 + (3 \* 4)).

Есть ещё одно преимущество в использовании скобок – они дают возможность изме- нить порядок вычисления выражений. Например, если сложение необходимо произвести прежде умножения, можно записать нечто вроде (2 + 3) \* 4.

## Ассоциативность

Операторы обычно обрабатываются слева направо. Это означает, что операторы с равным приоритетом будут обработаны по порядку от левого до правого. Например, 2 + 3 + 4 обрабатывается как (2 + 3) + 4.

## Выражения

Пример (сохраните как expression.py):

length = 5

breadth = 2

area = length \* breadth print('Площадь равна', area)

print('Периметр равен', 2 \* (length + breadth))

#### Вывод:

$ python expression.py Площадь равна 10

Периметр равен 14

**Как это работает:**

Длина и ширина прямоугольника хранятся в переменных length и breadth соответственно. Мы используем их для вычисления периметра и площади прямоугольника при помощи выражений. Результат выражения length \* breadth сохраняется в переменной area, после чего выводится на экран функцией print. Во втором случае мы напрямую подставляем значение вы- ражения 2 \* (length + breadth) в функцию print.

Также обратите внимание, как Python «красиво печатает» результат. Несмот- ря на то, что мы не указали пробела между 'Площадь равна' и перемен- ной area, Python подставляет его за нас, чтобы получить красивый и понят- ный вывод. Программа же остаётся при этом легкочитаемой (поскольку нам не нужно заботиться о пробелах между строками, которые мы выводим). Это пример того, как Python облегчает жизнь программисту.

## Резюме

Мы увидели, как пользоваться операторами, операндами и выражениями. Это основные строительные блоки любой программы. Далее мы увидим, как это применить на практи- ке.

# Поток команд

В программах, которые мы до сих пор рассматривали, последовательность команд всегда выполнялась Python по порядку строго сверху вниз. А что, если нам необходимо изменить поток выполняющихся команд? Например, если требуется, чтобы программа принимала некоторое решение и выполняла различные действия в зависимости от ситуации; ска- жем, печатала «Доброе утро» или «Добрый вечер» в зависимости от времени суток.

Как вы уже, наверное, догадались, этого можно достичь при помощи операторов управ- ления потоком. В Python есть три оператора управления потоком: if, for и while.

## Оператор if

Оператор if используется для проверки условий: *если*[1](#_bookmark69) условие верно[2](#_bookmark70), выполняется блок выражений (называемый «if-блок»), *иначе*[3](#_bookmark71) выполняется другой блок выражений (назы- ваемый «else-блок»). Блок «else» является необязательным.

**Пример:** (сохраните как if.py)

(продолжение на следующей странице)

number = 23

guess = int(input('Введите целое число : '))

**if** guess == number:

print('Поздравляю, вы угадали,') *# Начало нового блока*

print('(хотя и не выиграли никакого приза!)') *# Конец нового блока*

**elif** guess < number:

print('Нет, загаданное число немного больше этого.') *# Ещё один блок # Внутри блока вы можете выполнять всё, что угодно ...*

**else**:

print('Нет, загаданное число немного меньше этого.')

*# чтобы попасть сюда, guess должно быть больше, чем number*

1 if – *англ.* «если» (*прим.перев.*)

2 Соответствует булевому значению True (*прим.перев.*)

3 else – *англ.* «иначе», «в противном случае» (*прим.перев.*)

(продолжение с предыдущей страницы)

print('Завершено')

*# Это последнее выражение выполняется всегда после выполнения оператора if*

#### Вывод:

$ python if.py

Введите целое число : 50

Нет, загаданное число немного меньше этого. Завершено

$ python if.py

Введите целое число : 22

Нет, загаданное число немного больше этого. Завершено

$ python if.py

Введите целое число : 23 Поздравляю, вы угадали,

(хотя и не выиграли никакого приза!) Завершено

**Как это работает:**

В этой программе мы принимаем варианты от пользователя и проверяем, сов- падают ли они с заранее заданным числом. Мы устанавливаем переменной number значение любого целого числа, какого хотим. Например, 23. После этого мы принимаем вариант числа от пользователя при помощи функции input(). Функции – это всего-навсего многократно используемые фрагмен- ты программы. Мы узнаем о них больше в [*следующей главе*](#_bookmark83).

Мы передаём встроенной функции input строку, которую она выводит на экран и ожидает ввода от пользователя. Как только мы ввели что-нибудь и на- жали клавишу Enter, функция input() возвращает строку, которую мы вве- ли. Затем мы преобразуем полученную строку в число при помощи int(), и сохраняем это значение в переменную guess. Вообще-то, int – это класс, но на данном этапе вам достаточно знать лишь, что при помощи него можно преобразовать строку в целое число (предполагая, что строка содержит целое число).

Далее мы сравниваем число, введённое пользователем, с числом, которое мы выбрали заранее. Если они равны, мы печатаем сообщение об успехе. Обра- тите внимание, что мы используем соответствующие уровни отступа, чтобы указать Python, какие выражения относятся к какому блоку. Вот почему отсту- пы так важны в Python. Я надеюсь, вы придерживаетесь правила «постоянных отступов», не так ли?

Обратите внимание, что в конце оператора if стоит двоеточие – этим мы по-

казываем, что далее следует блок выражений.

После этого мы проверяем, верно ли, что пользовательский вариант числа меньше загаданного, и если это так, мы информируем пользователя о том, что ему следует выбирать числа немного больше этого. Здесь мы использо- вали выражение elif, которое попросту объединяет в себе два связанных if else-if else выражения в одно выражение if-elif-else. Это облегчает чтение программы, а также не требует дополнительных отступов.

Выражения elif и else также имеют двоеточие в конце логической строки, за которым следуют соответствующие блоки команд (с соответствующим чис- лом отступов, конечно).

Внутри if-блока оператора if может быть другой оператор if и так далее – это называется вложенным[4](#_bookmark72) оператором if.

Помните, что части elif и else не обязательны. Минимальная корректная запись оператора if такова:

**if True**:

print('Да, это верно.')

После того, как Python заканчивает выполнение всего оператора if вместе с его частями elif и else, он переходит к следующему выражению в блоке, содержащем этот оператор if. В нашем случае это основной блок программы (в котором начинается выполнение программы), а следующее выражение – это print('Завершено'). После этого Python доходит до конца программы и просто выходит из неё.

Хотя это и чрезвычайно простая программа, я указал вам на целый ряд вещей, которые стоит взять на заметку. Всё это довольно легко (даже удивительно легко для тех из вас, кто пришёл из мира C/C++). Поначалу вам придётся держать все эти вещи в памяти, но после некоторой практики вы привыкнете, и они вам покажутся вполне «естественными».

#### Замечание для программистов на C/C++

В Python нет оператора switch. Однако, при помощи конструкции if..elif..else мож- но достичь того же самого (а в некоторых случаях можно даже использовать [*словарь*](#_bookmark119), что- бы сделать это быстро).

4 nested – *англ.* «вложенный» (*прим.перев.*)

## Оператор while

Оператор while позволяет многократно выполнять блок команд до тех пор, пока выпол- няется некоторое условие. Это один из так называемых *операторов цикла*. Он также может иметь необязательный пункт else.

**Пример:** (сохраните как while.py)

number = 23 running = **True**

**while** running:

guess = int(input('Введите целое число : '))

**if** guess == number: print('Поздравляю, вы угадали.')

running = **False** *# это останавливает цикл while*

**elif** guess < number:

print('Нет, загаданное число немного больше этого.')

**else**:

print('Нет, загаданное число немного меньше этого.')

**else**:

print('Цикл while закончен.')

*# Здесь можете выполнить всё что вам ещё нужно*

print('Завершение.')

#### Вывод:

$ python while.py Введите целое число : 50

Нет, загаданное число немного меньше этого. Введите целое число : 22

Нет, загаданное число немного больше этого. Введите целое число : 23

Поздравляю, вы угадали. Цикл while закончен.

Завершение.

**Как это работает:**

В этой программе мы продолжаем играть в игру с угадыванием, но преиму- щество состоит в том, что теперь пользователь может угадывать до тех пор, пока не угадает правильное число, и ему не придётся запускать программу заново для каждой попытки, как это происходило до сих пор. Это наглядно демонстрирует применение оператора while.

Мы переместили операторы input и if внутрь цикла while и установили

#### 9.2. Оператор while 58

переменную running в значение True перед запуском цикла. Прежде всего проверяется, равно ли значение переменной running True, а затем происхо- дит переход к соответствующему *while-блоку*. После выполнения этого блока команд условие, которым в данном случае является переменная running, про- веряется снова. Если оно истинно, while-блок запускается снова, в противном случае происходит переход к дополнительному else-блоку, а затем – к следу- ющему оператору.

Блок else выполняется тогда, когда условие цикла while становится ложным (False)

– это может случиться даже при самой первой проверке условия. Если у цикла while имеется дополнительный блок else, он всегда выполняется, если только цикл не будет прерван оператором break.

True и False называются булевым типом данных, и вы можете считать их эквивалент- ными значениям 1 и 0 соответственно.

#### Примечание для программистов на C/C++

Помните, что у цикла while может быть блок else.

## Цикл for

Оператор for..in также является оператором цикла, который осуществляет *итерацию* по последовательности объектов, т.е. проходит через каждый элемент в последовательности. Мы узнаем больше о [*последовательностях*](#_bookmark121) в дальнейших главах, а пока просто запомните, что последовательность – это упорядоченный набор элементов.

**Пример:** (сохраните как for.py)

**for** i **in** range(1, 5): print(i)

**else**:

print('Цикл for закончен')

#### Вывод:

$ python for.py 1

2

3

4

Цикл for закончен

**Как это работает:**

В этой программе мы выводим на экран *последовательность* чисел. Мы гене- рируем эту последовательность, используя встроенную функцию range[5](#_bookmark75).

Мы задаём два числа, и range возвращает последовательность чисел от перво- го числа до второго. Например, range(1,5) даёт последовательность [1, 2, 3, 4]. По умолчанию range принимает значение шага, равное 1. Если мы зададим также и третье число range, оно будет служить шагом. Например, range(1,5,2) даст [1,3]. Помните, интервал простирается только *до* второ- го числа, т.е. **не** включает его в себя.

Обратите внимание, что range() генерирует последовательность чисел, но только по одному числу за раз – когда оператор for запрашивает следую- щий элемент. Чтобы увидеть всю последовательность чисел сразу, используй- те list(range()). Списки[6](#_bookmark76) подробно рассматриваются в главе [*Структуры*](#_bookmark115)[*данных*](#_bookmark115).

Затем цикл for осуществляет итерацию по этому диапазону - for i in range(1,5) эквивалентно for i in [1, 2, 3, 4], что напоминает при- сваивание переменной i по одному числу (или объекту) за раз, выполняя блок команд для каждого значения i. В данном случае в блоке команд мы просто выводим значение на экран.

Помните, что блок else не обязателен. Если он присутствует, он всегда выполняется один раз после окончания цикла for, если только не указан оператор [*break*](#_bookmark77).

Помните также, что цикл for..in работает для любой последовательности. В нашем слу- чае это список чисел, сгенерированный встроенной функцией range, но в общем случае можно использовать любую последовательность любых объектов! В следующих разделах мы познакомимся с этим поближе.

#### Примечание для программистов на C/C++/Java/C#

Цикл for в Python радикально отличается от цикла for в C/C++. Программисты на C# заметят, что цикл for в Python похож на цикл foreach в C#. Программистам на Java это может напомнить конструкцию for (int i : IntArray) в Java 1.5.

Если в C/C++ записать for (int i = 0; i < 5; i++), то в Python этому соответство- вало бы выражение for i in range(0,5). Как видно, в Python цикл for проще, более выразителен и менее подвержен ошибкам.

5 range – *англ.* «диапазон», «интервал» (*прим.перев.*)

6 list – *англ.* «список» (*прим.перев.*)

## Оператор break

Оператор break служит для *прерывания*[7](#_bookmark78) цикла, т.е. остановки выполнения команд даже если условие выполнения цикла ещё не приняло значения False или последователь- ность элементов не закончилась.

Важно отметить, что если циклы for или while прервать оператором *break*, соответству- ющие им блоки else выполняться **не** будут.

**Пример:** (сохраните как break.py)

**while True**:

s = input('Введите что-нибудь : ')

**if** s == 'выход':

**break**

print('Длина строки:', len(s)) print('Завершение')

#### Вывод:

$ python break.py

Введите что-нибудь : Программировать весело. Длина строки: 23

Введите что-нибудь : Если работа скучна, Длина строки: 19

Введите что-нибудь : Чтобы придать ей весёлый тон - Длина строки: 30

Введите что-нибудь : используй Python! Длина строки: 23

Введите что-нибудь : выход Завершение

**Как это работает:**

В этой программе мы многократно считываем пользовательский ввод и выво- дим на экран длину каждой введённой строки. Для остановки программы мы вводим специальное условие, проверяющее, совпадает ли пользовательский ввод со строкой 'выход'. Мы останавливаем программу *прерыванием* цикла оператором break и достигаем её конца.

Длина введённой строки может быть найдена при помощи встроенной функ- ции len.

Помните также, что оператор break может применяться и в цикле for.

7 break – *англ.* «разбивать», «разрывать» (*прим.перев.*)

* + 1. Поэтический Python Swaroop’а

Для ввода строк здесь я использовал мини-стишок, который сам сочинил. Он называется

*Поэтический Python Swaroop’а*[8](#_bookmark80)

Программировать весело. Если работа скучна,

Чтобы придать ей весёлый тон - используй Python!

## Оператор continue

Оператор continue используется для указания Python, что необходимо пропустить все оставшиеся команды в текущем блоке цикла и *продолжить*[9](#_bookmark81) со следующей итерации цик- ла.

**Пример:** (сохраните как continue.py)

**while True**:

s = input('Введите что-нибудь : ')

**if** s == 'выход':

**break**

**if** len(s) < 3: print('Слишком мало') **continue**

print('Введённая строка достаточной длины')

*# Разные другие действия здесь...*

#### Вывод:

$ python continue.py Введите что-нибудь : a Слишком мало

Введите что-нибудь : 12 Слишком мало

Введите что-нибудь : абв

Введённая строка достаточной длины Введите что-нибудь : выход

8 **Swaroop’s Poetic Python**:

Programming is fun. When the work is done,

if you wanna make your work also fun: use Python!

9 continue – *англ.* «продолжать» (*прим.перев.*)

#### Как это работает:

В этой программе мы запрашиваем ввод со стороны пользователя, но обраба- тываем введённую строку только если она имеет длину хотя бы в 3 символа. Итак, мы используем встроенную функцию len для получения длины строки, и если длина менее 3, мы пропускаем остальные действия в блоке при помо- щи оператора continue. В противном случае все остальные команды в цикле выполняются, производя любые манипуляции, которые нам нужны.

Заметьте, что оператор continue также работает и с циклом for.

## Резюме

Мы увидели, как использовать три оператора для управления потоком команд: if, while и for, а также связанные с ними операторы break и continue. Это наиболее часто ис- пользуемые конструкции Python, поэтому овладеть ими очень важно.

Далее мы увидим, как создавать и использовать функции.

# Функции

Функции – это многократно используемые фрагменты программы. Они позволяют дать имя определённому блоку команд с тем, чтобы впоследствии запускать этот блок по ука- занному имени в любом месте программы и сколь угодно много раз. Это называется *вы- зовом* функции. Мы уже использовали много встроенных функций, как то len и range.

Функция – это, пожалуй, *наиболее* важный строительный блок любой нетривиальной про- граммы (на любом языке программирования), поэтому в этой главе мы рассмотрим раз- личные аспекты функций.

Функции определяются при помощи зарезервированного слова def. После этого слова указывается *имя* функции, за которым следует пара скобок, в которых можно указать имена некоторых переменных, и заключительное двоеточие в конце строки. Далее сле- дует блок команд, составляющих функцию. На примере можно видеть, что на самом деле это очень просто:

**Пример:** (сохраните как function1.py)

**def** sayHello():

print('Привет, Мир!') *# блок, принадлежащий функции # Конец функции*

sayHello() *# вызов функции*

sayHello() *# ещё один вызов функции*

#### Вывод:

$ python function1.py Привет, Мир!

Привет, Мир!

**Как это работает:**

Мы определили функцию с именем sayHello, используя описанный выше синтаксис. Эта функция не принимает параметров, поэтому в скобках не объ- явлены какие-либо переменные. Параметры функции – это некие входные данные, которые мы можем передать функции, чтобы получить соответству- ющий им результат.

Обратите внимание, что мы можем вызывать одну и ту же функцию много раз, а значит нет необходимости писать один и тот же код снова и снова.

## Параметры функций

Функции могут принимать параметры, т.е. некоторые значения, передаваемые функции для того, чтобы она что-либо *сделала* с ними. Эти параметры похожи на переменные, за исключением того, что значение этих переменных указывается при вызове функции, и во время работы функции им уже присвоены их значения.

Параметры указываются в скобках при объявлении функции и разделяются запятыми. Аналогично мы передаём значения, когда вызываем функцию. Обратите внимание на терминологию: имена, указанные в объявлении функции, называются *параметрами*, то- гда как значения, которые вы передаёте в функцию при её вызове, – *аргументами*.

**Пример:** (сохраните как func\_param.py)

**def** printMax(a, b):

**if** a > b:

print(a, 'максимально')

**elif** a == b:

print(a, 'равно', b)

**else**:

print(b, 'максимально') printMax(3, 4) *# прямая передача значений*

x = 5

y = 7

printMax(x, y) *# передача переменных в качестве аргументов*

#### Вывод:

$ python func\_param.py

4 максимально

7 максимально

**Как это работает:**

Здесь мы определили функцию с именем printMax, которая использует два параметра с именами a и b. Мы находим наибольшее число с применением простого оператора if..else и выводим это число.

При первом вызове функции printMax мы напрямую передаём числа в ка- честве аргументов. Во втором случае мы вызываем функцию с переменными

в качестве аргументов. printMax(x, y) назначает значение аргумента x па- раметру a, а значение аргумента y – параметру b. В обоих случаях функция printMax работает одинаково.

## Локальные переменные

При объявлении переменных внутри определения функции, они никоим образом не свя- заны с другими переменными с таким же именем за пределами функции – т.е. имена переменных являются *локальными* в функции. Это называется *областью видимости* пере- менной. Область видимости всех переменных ограничена блоком, в котором они объяв- лены, начиная с точки объявления имени.

**Пример:** (сохраните как func\_local.py)

x = 50

**def** func(x):

print('x равен', x) x = 2

print('Замена локального x на', x)

func(x)

print('x по-прежнему', x)

#### Вывод:

$ python func\_local.py x равен 50

Замена локального x на 2 x по-прежнему 50

**Как это работает:**

При первом выводе *значения*, присвоенного имени x, в первой строке функции Python использует значение параметра, объявленного в основном блоке, выше определения функции.

Далее мы назначаем x значение 2. Имя x локально для нашей функции. По- этому когда мы заменяем значение x в функции, x, объявленный в основном блоке, остаётся незатронутым.

Последним вызовом функции print мы выводим значение x, указанное в ос- новном блоке, подтверждая таким образом, что оно не изменилось при ло- кальном присваивании значения в ранее вызванной функции.

## Зарезервированное слово «global»

Чтобы присвоить некоторое значение переменной, определённой на высшем уровне про- граммы (т.е. не в какой-либо области видимости, как то функции или классы), необходимо указать Python, что её имя не локально, а *глобально* (*global*). Сделаем это при помощи заре- зервированного слова global. Без применения зарезервированного слова global невоз- можно присвоить значение переменной, определённой за пределами функции.

Можно использовать уже существующие значения переменных, определённых за преде- лами функции (при условии, что внутри функции не было объявлено переменной с таким же именем). Однако, это не приветствуется, и его следует избегать, поскольку человеку, читающему текст программы, будет непонятно, где находится объявление переменной. Использование зарезервированного слова global достаточно ясно показывает, что пере- менная объявлена в самом внешнем блоке.

**Пример:** (сохраните как func\_global.py)

x = 50

**def** func():

**global** x

print('x равно', x) x = 2

print('Заменяем глобальное значение x на', x)

func()

print('Значение x составляет', x)

#### Вывод:

$ python func\_global.py x равно 50

Заменяем глобальное значение x на 2 Значение x составляет 2

**Как это работает:**

Зарезервированное слово global используется для того, чтобы объявить, что x – это глобальная переменная, а значит, когда мы присваиваем значение име- ни x внутри функции, это изменение отразится на значении переменной x в основном блоке программы.

Используя одно зарезервированное слово global, можно объявить сразу несколько переменных: global x, y, z.

## Зарезервированное слово «nonlocal»

Мы увидели, как получать доступ к переменным в локальной и глобальной области ви- димости. Есть ещё один тип области видимости, называемый «нелокальной» (nonlocal) областью видимости, который представляет собой нечто среднее между первыми дву- мя. Нелокальные области видимости встречаются, когда вы определяете функции внутри функций.

Поскольку в Python всё является выполнимым кодом, вы можете определять функции где угодно.

#### Давайте рассмотрим пример:

*# Filename: func\_nonlocal.py*

**def** func\_outer(): x = 2

print('x равно', x)

**def** func\_inner():

**nonlocal** x x = 5

func\_inner()

print('Локальное x сменилось на', x)

func\_outer()

**Вывод:**

$ python func\_nonlocal.py x равно 2

Локальное x сменилось на 5

#### Как это работает:

Когда мы находимся внутри func\_inner, переменная x, определённая в пер- вой строке func\_outer находится ни в локальной области видимости (опре- деление переменной не входит в блок func\_inner), ни в глобальной обла- сти видимости (она также и не в основном блоке программы). Мы объявля- ем, что хотим использовать именно эту переменную x, следующим образом: nonlocal x.

Попробуйте заменить «nonlocal x» на «global x», а затем удалить это за- резервированное слово, и пронаблюдайте за разницей между этими двумя случаями.

## Значения аргументов по умолчанию

Зачастую часть параметров функций могут быть *необязательными*, и для них будут ис- пользоваться некоторые заданные значения по умолчанию, если пользователь не укажет собственных. Этого можно достичь с помощью значений аргументов по умолчанию. Их можно указать, добавив к имени параметра в определении функции оператор присваи- вания (=) с последующим значением.

Обратите внимание, что значение по умолчанию должно быть константой. Или точнее говоря, оно должно быть неизменным[1](#_bookmark89) – это объясняется подробнее в последующих гла- вах. А пока запомните это.

**Пример:** (сохраните как func\_default.py)

**def** say(message, times = 1): print(message \* times)

say('Привет')

say('Мир', 5)

#### Вывод:

$ python func\_default.py Привет

МирМирМирМирМир

**Как это работает:**

Функция под именем say используется для вывода на экран строки указанное число раз. Если мы не указываем значения, по умолчанию строка выводится один раз. Мы достигаем этого указанием значения аргумента по умолчанию, равного 1 для параметра times[2](#_bookmark90).

При первом вызове say мы указываем только строку, и функция выводит её один раз. При втором вызове say мы указываем также и аргумент 5, обозначая таким образом, что мы хотим *сказать*[3](#_bookmark91) фразу 5 раз.

**Важно:** Значениями по умолчанию могут быть снабжены только параметры, находящи- еся в конце списка параметров. Таким образом, в списке параметров функции параметр со значением по умолчанию не может предшествовать параметру без значения по умол- чанию. Это связано с тем, что значения присваиваются параметрам в соответствии с их положением. Например, def func(a, b=5) допустимо, а def func(a=5, b) – *не допустимо*.

1 «immutable» в терминологии Python (*прим. перев.*)

2 times – *англ.* «раз» (*прим. перев.*)

3 say – *англ.* «сказать» (*прим. перев.*)

## Ключевые аргументы

Если имеется некоторая функция с большим числом параметров, и при её вызове тре- буется указать только некоторые из них, значения этих параметров могут задаваться по их имени – это называется *ключевые параметры*. В этом случае для передачи аргументов функции используется имя (ключ) вместо позиции (как было до сих пор).

Есть два *преимущества* такого подхода: во-первых, использование функции становится легче, поскольку нет необходимости отслеживать порядок аргументов; во-вторых, можно задавать значения только некоторым избранным аргументам, при условии, что осталь- ные параметры имеют значения аргумента по умолчанию.

**Пример:** (сохраните как func\_key.py)

**def** func(a, b=5, c=10):

print('a равно', a, ', b равно', b, ', а c равно', c)

func(3, 7) func(25, c=24) func(c=50, a=100)

#### Вывод:

$ python func\_key.py

a равно 3, b равно 7, а c равно 10 a равно 25, b равно 5, а c равно 24

a равно 100, b равно 5, а c равно 50

**Как это работает:**

Функция с именем func имеет один параметр без значения по умолчанию, за которым следуют два параметра со значениями по умолчанию.

При первом вызове, func(3, 7), параметр a получает значение 3, параметр

b получает значение 7, а c получает своё значение по умолчанию, равное 10.

При втором вызове func(25, c=24) переменная a получает значение 25 в силу позиции аргумента. После этого параметр c получает значение 24 по имени, т.е. как ключевой параметр. Переменная b получает значение по умол- чанию, равное 5.

При третьем обращении func(c=50, a=100) мы используем ключевые аргу- менты для всех указанных значений. Обратите внимание на то, что мы ука- зываем значение для параметра c перед значением для a, даже несмотря на то, что в определении функции параметр a указан раньше c.

## Переменное число параметров4

Иногда бывает нужно определить функцию, способную принимать *любое* число парамет- ров. Этого можно достичь при помощи звёздочек (сохраните как function\_varargs.py):

**def** total(a=5, \*numbers, \*\*phonebook): print('a', a)

*#проход по всем элементам кортежа*

**for** single\_item **in** numbers: print('single\_item', single\_item)

*#проход по всем элементам словаря*

**for** first\_part, second\_part **in** phonebook.items(): print(first\_part,second\_part)

print(total(10,1,2,3,Jack=1123,John=2231,Inge=1560))

#### Вывод:

$ python function\_varargs.py a 10

single\_item 1

single\_item 2

single\_item 3

|  |  |
| --- | --- |
| Inge | 1560 |
| John | 2231 |
| Jack | 1123 |
| None |  |

**Как это работает:**

Когда мы объявляем параметр со звёздочкой (например, \*param), все позици- онные аргументы начиная с этой позиции и до конца будут собраны в кортеж под именем param.

Аналогично, когда мы объявляем параметры с двумя звёздочками (\*\*param), все ключевые аргументы начиная с этой позиции и до конца будут собраны в словарь под именем param.

Мы изучим кортежи и словари в [*одной из последующих глав*](#_bookmark114).

4 VarArgs – *от англ.* «**Var**iable number of **Arg**ument**s**» – «переменное число аргументов» (*прим. перев.*)

## Только ключевые параметры

Если некоторые ключевые параметры должны быть доступны только по ключу, а *не* как позиционные аргументы, их можно объявить после параметра со звёздочкой (сохраните как keyword\_only.py):

**def** total(initial=5, \*numbers, extra\_number): count = initial

**for** number **in** numbers: count += number

count += extra\_number print(count)

total(10, 1, 2, 3, extra\_number=50)

total(10, 1, 2, 3)

*# Вызовет ошибку, поскольку мы не указали значение # аргумента по умолчанию для 'extra\_number'.*

#### Вывод:

$ python keyword\_only.py 66

Traceback (most recent call last):

File "keyword\_only.py", line 12, in <module> total(10, 1, 2, 3)

TypeError: total() needs keyword-only argument extra\_number

**Как это работает:**

Объявление параметров после параметра со звёздочкой даёт только ключевые аргументы. Если для таких аргументов не указано значение по умолчанию, и оно не передано при вызове, обращение к функции вызовет ошибку, в чём мы только что убедились.

Обратите внимание на использование +=, который представляет собой сокра- щённый оператор, позволяющий вместо x = x + y просто написать x += y.

Если вам нужны аргументы, передаваемые только по ключу, но не нужен па- раметр со звёздочкой, то можно просто указать одну звёздочку без указания имени: def total(initial=5, \*, extra\_number).

## Оператор «return»

Оператор return используется для возврата[5](#_bookmark96) из функции, т.е. для прекращения её работы и выхода из неё. При этом можно также *вернуть некоторое значение* из функции.

**Пример:** (сохраните как func\_return.py)

*#!/usr/bin/python*

*# Filename: func\_return.py*

**def** maximum(x, y):

**if** x > y:

**return** x

**elif** x == y:

**return** 'Числа равны.'

**else**:

**return** y

print(maximum(2, 3))

#### Вывод:

$ python func\_return.py 3

**Как это работает:**

Функция maximum возвращает максимальный из двух параметров, которые в данном случае передаются ей при вызове. Она использует обычный условный оператор if..else для определения наибольшего числа, а затем *возвращает* это число.

Обратите внимание, что оператор return без указания возвращаемого значения эквива- лентен выражению return None. None – это специальный тип данных в Python, обозна- чающий ничего. К примеру, если значение переменной установлено в None, это означает, что ей не присвоено никакого значения.

Каждая функция содержит в неявной форме оператор return None в конце, если вы не указали своего собственного оператора return. В этом можно убедиться, запустив print(someFunction()), где функция someFunction – это какая-нибудь функция, не имеющая оператора return в явном виде. Например:

**def** someFunction():

**pass**

Оператор pass используется в Python для обозначения пустого блока команд.

5 return – *англ.* «возврат» (*прим. перев.*)

**Примечание:** Существует встроенная функция max, в которой уже реализован функци- онал «поиск максимума», так что пользуйтесь этой встроенной функцией, где это воз- можно.

## Строки документации6

Python имеет остроумную особенность, называемую *строками документации*, обычно обозначаемую сокращённо *docstrings*. Это очень важный инструмент, которым вы обяза- тельно должны пользоваться, поскольку он помогает лучше документировать программу и облегчает её понимание. Поразительно, но строку документации можно получить, на- пример, из функции, даже во время выполнения программы!

**Пример:** (сохраните как func\_doc.py)

**def** printMax(x, y):

*'''Выводит максимальное из двух чисел.*

*Оба значения должны быть целыми числами.'''*

x = int(x) *# конвертируем в целые, если возможно*

y = int(y)

**if** x > y:

print(x, 'наибольшее')

**else**:

print(y, 'наибольшее')

printMax(3, 5) print(printMax. doc )

#### Вывод:

$ python func\_doc.py

5 наибольшее

Выводит максимальное из двух чисел.

Оба значения должны быть целыми числами.

**Как это работает:**

Строка в первой логической строке функции является *строкой документации* для этой функции. Обратите внимание на то, что строки документации при- менимы также к [*модулям*](#_bookmark101) и [*классам*](#_bookmark144), о которых мы узнаем в соответствующих

6 DocString - *от англ.* «**Doc**umentation **String**» – «строка документации» (*прим. перев.*)

главах.

Строки документации принято записывать в форме многострочной[7](#_bookmark100) строки, где первая строка начинается с заглавной буквы и заканчивается точкой. Вто- рая строка оставляется пустой, а подробное описание начинается с третьей. Вам *настоятельно рекомендуется* следовать такому формату для всех строк документации всех ваших нетривиальных функций.

Доступ к строке документации функции printMax можно получить с помо- щью атрибута этой функции (т.е. имени, принадлежащего ей) doc (обра- тите внимание на *двойное подчёркивание*). Просто помните, что Python пред- ставляет *всё* в виде объектов, включая функции. Мы узнаем больше об объек- тах в главе [*о классах*](#_bookmark144).

Если вы пользовались функцией help() в Python, значит вы уже видели строки доку- ментации. Эта функция просто-напросто считывает атрибут doc соответствующей функции и аккуратно выводит его на экран. Вы можете проверить её на рассмотренной выше функции: просто включите help(printMax) в текст программы. Не забудьте на- жать клавишу q для выхода из справки (help).

Точно так же автоматические инструменты могут получать документацию из програм- мы. Именно поэтому я *настоятельно рекомендую* вам использовать строки документации для любой нетривиальной функции, которую вы пишете. Команда pydoc, поставляемая вместе с пакетом Python, работает аналогично функции help().

## Аннотации

Функции имеют ещё одну дополнительную возможность, называемую аннотациями, ко- торые предоставляют отличный способ сопровождения каждого параметра, равно как и возвращаемого значения дополнительной информацией. Поскольку сам язык Python не интерпретирует эти аннотации каким-либо способом (этот функционал отводится посто- ронним библиотекам), мы опустим эту возможность из нашего обсуждения. Если вам интересно почитать об аннотациях, просмотрите [**PEP 3107**](https://www.python.org/dev/peps/pep-3107).

## Резюме

Мы рассмотрели достаточно много аспектов функций, но тем не менее, вы должны по- нимать, что это далеко не все их аспекты. В то же время, мы охватили большинство того, с чем вы будете сталкиваться при повседневном использовании функций в Python.

Далее мы увидим, как использовать и создавать модули Python.

7 т.е. строки, содержащей символы перевода строки. (*прим. перев*)

**10.11. Аннотации** 75

# Модули

Как можно использовать код повторно, помещая его в функции, мы уже видели. А что, если нам понадобится повторно использовать различные функции в других наших про- граммах? Как вы уже, наверное, догадались, ответ – модули.

Существуют разные способы составления модулей, но самый простой – это создать файл с расширением .py, содержащий функции и переменные.

Другой способ – написать модуль на том языке программирования, на котором написан сам интерпретатор Python. Например, можно писать модули на [языке программирова-](http://docs.python.org/py3k/extending/index.html) [ния C](http://docs.python.org/py3k/extending/index.html), которые после компиляции могут использоваться стандартным интерпретатором Python.

Модуль можно *импортировать* в другую программу, чтобы использовать функции из него. Точно так же мы используем стандартную библиотеку Python. Сперва посмотрим, как использовать модули стандартной библиотеки.

**Пример:** (сохраните как using\_sys.py)

**import sys**

print('Аргументы командной строки:')

**for** i **in** sys.argv: print(i)

print('**\n\n**Переменная PYTHONPATH содержит', sys.path, '**\n**')

#### Вывод:

(продолжение на следующей странице)

$ python3 using\_sys.py we are arguments Аргументы командной строки:

using\_sys.py we

are arguments

Переменная PYTHONPATH содержит ['', 'C:\\Windows\\system32\\python30.zip', 'C:\\Python30\\DLLs', 'C:\\Python30\\lib',

(продолжение с предыдущей страницы)

'C:\\Python30\\lib\\plat-win', 'C:\\Python30', 'C:\\Python30\\lib\\site-packages']

#### Как это работает:

В начале мы *импортируем* модуль sys командой import. Этим мы говорим Python, что хотим использовать этот модуль. Модуль sys содержит функции, относящиеся к интерпретатору Python и его среде, т.е. к **сис**теме (**sys**tem).

Когда Python выполняет команду import sys, он ищет модуль sys. В данном случае это один из встроенных модулей, и Python знает, где его искать.

Если бы это был не скомпилированный модуль, т.е. модуль, написанный на Python, тогда интерпретатор Python искал бы его в каталогах, перечисленных в переменной sys.path. Если модуль найден, выполняются команды в теле модуля, и он становится *доступным*. Обратите внимание, что инициализация[1](#_bookmark102) происходит только при *первом* импорте модуля.

Доступ к переменной argv в модуле sys предоставляется при помощи точки, т.е. sys.argv. Это явно показывает, что это имя является частью модуля sys. Ещё одним преимуществом такого обозначения является то, что имя не кон- фликтует с именем переменной argv, которая может использоваться в вашей программе.

Переменная sys.argv является *списком* строк (списки будут детально обсуж- даться в [*одной из последующих глав*](#_bookmark115)). Она содержит список *аргументов команд- ной строки*, т.е. аргументов, переданных программе из командной строки.

Если вы используете среду разработки[2](#_bookmark103) для написания и запуска программ, поищите где-нибудь в её меню возможность передавать параметры команд- ной строки.

В нашем примере, когда мы запускаем «python using\_sys.py we are arguments», мы запускаем модуль using\_sys.py командой python, а всё, что следует далее – аргументы, передаваемые программе[3](#_bookmark104). Python сохраняет аргументы командной строки в переменной sys.argv для дальнейшего ис- пользования.

Помните, что имя запускаемого сценария[4](#_bookmark105) всегда является первым аргумен- том в списке sys.argv. Так что в приведённом примере 'using\_sys.py' бу- дет элементом sys.argv[0], 'we' – sys.argv[1], 'are' – sys.argv[2], а

1 Инициализация – ряд действий, производимых при начальной загрузке (*прим. перев.*)

2 IDE – *от англ.* «Integrated Development Environment» – «интегрированная среда разработки» (*прим. пе- рев.*)

3 «we are arguments» – *англ.* «мы аргументы» (*прим. перев.*)

4 Программу на интерпретируемом языке программирования также называют *сценарием* или *скриптом*

(*прим. перев.*)

'arguments' – sys.argv[3]. Помните, что в Python нумерация начинается с 0, а не с 1.

sys.path содержит список имён каталогов, откуда импортируются модули. Заметьте, что первая строка в sys.path пуста; эта пустая строка показывает, что текущая директория также является частью sys.path, которая совпадает со значением переменной окружения PYTHONPATH. Это означает, что моду- ли, расположенные в текущем каталоге, можно импортировать напрямую. В противном случае придётся поместить свой модуль в один из каталогов, пе- речисленных в sys.path.

Помните, что текущий каталог – это каталог, в котором была запущена про- грамма. Выполните «import os; print(os.getcwd())», чтобы узнать те- кущий каталог программы.

## Файлы байткода .pyc

Импорт модуля – относительно дорогостоящее мероприятие, поэтому Python предприни- мает некоторые трюки для ускорения этого процесса. Один из способов – создать *байт- компилированные* файлы (или *байткод*) с расширением .pyc, которые являются некой промежуточной формой, в которую Python переводит программу (помните раздел [*«Введе-*](#_bookmark22)[*ние»*](#_bookmark22) о том, как работает Python?). Такой файл .pyc полезен при импорте модуля в следу- ющий раз в другую программу – это произойдёт намного быстрее, поскольку значитель- ная часть обработки, требуемой при импорте модуля, будет уже проделана. Этот байткод также является платформо-независимым.

**Примечание:** Обычно файлы .pyc создаются в том же каталоге, где расположены и со- ответствующие им файлы .py. Если Python не может получить доступ для записи файлов в этот каталог, файлы .pyc созданы не будут.

## Оператор from … import …

Чтобы импортировать переменную argv прямо в программу и не писать всякий раз sys.

при обращении к ней, можно воспользоваться выражением «from sys import argv».

Для импорта всех имён, использующихся в модуле sys, можно выполнить команду

«from sys import \*». Это работает для любых модулей.

В общем случае вам *следует избегать* использования этого оператора и использовать вме- сто этого оператор import, чтобы предотвратить конфликты имён и не затруднять чтение программы.

**Пример:**

**from math import** \*

n = int(input("Введите диапазон:- ")) p = [2, 3]

count = 2

a = 5

**while** (count < n): b=0

**for** i **in** range(2,a):

**if** ( i <= sqrt(a)):

**if** (a % i == 0): print(a,"непростое") b = 1

**else**:

**pass**

**if** (b != 1):

print(a,"простое") p = p + [a]

count = count + 1 a = a + 2

print(p)

## Имя модуля – name

У каждого модуля есть имя, и команды в модуле могут узнать имя их модуля. Это полезно, когда нужно знать, запущен ли модуль как самостоятельная программа или импортиро- ван. Как уже упоминалось выше, когда модуль импортируется впервые, содержащийся в нём код исполняется. Мы можем воспользоваться этим для того, чтобы заставить модуль вести себя по-разному в зависимости от того, используется ли он сам по себе или импор- тируется в другую программа. Этого можно достичь с применением атрибута модуля под названием name .

**Пример:** (сохраните как using\_name.py)

**if** name == ' main ':

print('Эта программа запущена сама по себе.')

**else**:

print('Меня импортировали в другой модуль.')

#### Вывод:

(продолжение на следующей странице)

$ python3 using\_name.py

Эта программа запущена сама по себе.

#### 11.3. Имя модуля – name 79

(продолжение с предыдущей страницы)

$ python3

>>> import using\_name

Меня импортировали в другой модуль.

>>>

#### Как это работает:

В каждом модуле Python определено его имя – name [5](#_bookmark110) . Если оно равно ' main ', это означает, что модуль запущен самостоятельно пользовате- лем, и мы можем выполнить соответствующие действия.

## Создание собственных модулей

Создать собственный модуль очень легко. Да вы всё время делали это! Ведь каждая про- грамма на Python также является и модулем. Необходимо лишь убедиться, что у неё уста- новлено расширение .py. Следующий пример объяснит это.

**Пример:** (сохраните как mymodule.py)

**def** sayhi():

print('Привет! Это говорит мой модуль.')

version = '0.1'

*# Конец модуля mymodule.py*

Выше приведён простой *модуль*. Как видно, в нём нет ничего особенного по сравнению с обычной программой на Python. Далее посмотрим, как использовать этот модуль в других наших программах.

Помните, что модуль должен находиться либо в том же каталоге, что и программа, в ко- торую мы импортируем его, либо в одном из каталогов, указанных в sys.path.

Ещё один модуль (сохраните как mymodule\_demo.py):

**import mymodule**

mymodule.sayhi()

print ('Версия', mymodule. version )

5 name - *англ.* «имя» (*прим. перев.*)

#### Вывод:

$ python mymodule\_demo.py Привет! Это говорит мой модуль. Версия 0.1

**Как это работает:**

Обратите внимание, что мы используем всё то же обозначение точкой для до- ступа к элементам модуля. Python повсеместно использует одно и то же обо- значение точкой, придавая ему таким образом характерный «Python-овый» вид и не вынуждая нас изучать всё новые и новые способы делать что-либо.

Вот версия, использующая синтаксис from..import (сохраните как mymodule\_demo2. py):

**from mymodule import** sayhi, version

sayhi()

print('Версия', version )

Вывод mymodule\_demo2.py такой же, как и mymodule\_demo.py.

Обратите внимание, что если в модуле, импортирующем данный модуль, уже было объ- явлено имя version , возникнет конфликт. Это весьма вероятно, так как объявлять версию любого модуля при помощи этого имени – общепринятая практика. Поэтому все- гда рекомендуется отдавать предпочтение оператору import, хотя это и сделает вашу программу немного длиннее.

Вы могли бы также использовать:

**from mymodule import** \*

Это импортирует все публичные имена, такие как sayhi, но не импортирует

version , потому что оно начинается с двойного подчёркивания

#### Дзэн Python

Одним из руководящих принципов в Python является «Явное лучше Неявного». Выпол- ните команду «import this», чтобы узнать больше, а также просмотрите это обсужде- ние, в котором приводятся примеры по каждому из принципов.

## Функция dir

Встроенная функция dir() возвращает список имён, определяемых объектом. Например, для модуля в этот список входят функции, классы и переменные, определённые в этом модуле.

Эта функция может принимать аргументы. Если в качестве аргумента указано имя моду- ля, она возвращает список имён, определённых в этом модуле. Если никакого аргумента не передавать, она вернёт список имён, определённых в текущем модуле.

#### Пример:

$ python3

>>> import sys # получим список атрибутов модуля 'sys'

>>> dir(sys)

[' displayhook ', ' doc ', ' excepthook ', ' name ', ' package ', ' stderr ', ' stdin ', ' stdout ', '\_clear\_type\_cache', '\_compact\_free

lists', '\_current\_frames', '\_getframe', 'api\_version', 'argv', 'builtin\_module

\_names', 'byteorder', 'call\_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle', 'dont\_write\_bytecode', 'exc\_info', 'excepthook', 'exec\_prefix', 'executable', 'exit', 'flags', 'float\_info', 'getcheckinterval', 'getdefault encoding', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getref count', 'getsizeof', 'gettrace', 'getwindowsversion', 'hexversion', 'intern', 'maxsize', 'maxunicode', 'meta\_path', 'modules', 'path', 'path\_hooks', 'path\_importer\_cache', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version\_info', 'warnoptions', 'winver']

>>> dir() # получим список атрибутов текущего модуля

[' builtins ', ' doc ', ' name ', ' package ', 'sys']

>>> a = 5 # создадим новую переменную 'a'

>>> dir()

[' builtins ', ' doc ', ' name ', ' package ', 'a', 'sys']

>>> del a # удалим имя 'a'

>>> dir()

[' builtins ', ' doc ', ' name ', ' package ', 'sys']

>>>

#### Как это работает:

Сперва мы видим результат применения dir к импортированному модулю

#### 11.5. Функция dir 82

sys. Видим огромный список атрибутов, содержащихся в нём.

Затем мы вызываем функцию dir, не передавая ей параметров. По умолча- нию, она возвращает список атрибутов текущего модуля. Обратите внимание, что список импортированных модулей также входит туда.

Чтобы пронаблюдать за действием dir, мы определяем новую переменную a и присваиваем ей значение, а затем снова вызываем dir. Видим, что в полученном списке появилось дополнительное значение. Удалим перемен- ную/атрибут из текущего модуля при помощи оператора del, и изменения вновь отобразятся на выводе функции dir.

Замечание по поводу del: этот оператор используется для *удаления* перемен- ной/имени, и после его выполнения, в данном случае – del a, к переменной a больше невозможно обратиться – её как будто никогда и не было.

Обратите внимание, что функция dir() работает для *любого* объекта. Напри- мер, выполните «dir('print')», чтобы увидеть атрибуты функции print, или «dir(str)», чтобы увидеть атрибуты класса str.

## Пакеты

К настоящему времени вы, вероятно, начали наблюдать некоторую иерархию в организа- ции ваших программ. Переменные обычно находятся в функциях. Функции и глобальные переменные обычно находятся в модулях. А что, если возникнет необходимость как-то организовать модули? Вот здесь-то и выходят на сцену пакеты.

Пакеты – это просто каталоги с модулями и специальным файлом init .py, который показывает Python, что этот каталог особый, так как содержит модули Python.

Представим, что мы хотим создать пакет под названием «world» с субпакетами «asia»,

«africa» и т.д., которые, в свою очередь, будут содержать модули «india», «madagascar» и т.д.

Для этого следовало бы создать следующую структуру каталогов:

(продолжение на следующей странице)

| - <некоторый каталог из sys.path>/

| | world/

|

|

|

|

|

|

|

|

| init .py

| asia/

|

|

|

|

| init .py

| india/

|---- init py

| foo.py

| africa/

|---- init py

(продолжение с предыдущей страницы)

|

|

|

| madagascar/

|---- init py

| bar.py

Пакеты – это удобный способ иерархически организовать модули. Такое часто встреча- ется в [*стандартной библиотеке*](#_bookmark177).

## Резюме

Точно так же, как функции являются многократно используемыми фрагментами про- грамм, модули являются многократно используемыми программами. Пакеты – это спо- соб иерархической организации модулей. Стандартная библиотека Python является при- мером такого набора пакетов и модулей.

Мы увидели, как пользоваться этими модулями и создавать свои.

Далее мы познакомимся с некоторыми интересными концепциями, называемыми

«структуры данных».

# Структуры данных

Структуры данных – это, по сути, и есть *структуры*, которые могут хранить некоторые

*данные* вместе. Другими словами, они используются для хранения связанных данных.

В Python существуют четыре встроенных структуры данных: список, кортеж, словарь и множество. Посмотрим, как ими пользоваться, и как они могут облегчить нам жизнь.

## Список

Список[1](#_bookmark117) – это структура данных, которая содержит упорядоченный набор элементов, т.е. хранит *последовательность* элементов. Это легко представить, если вспомнить список по- купок, в котором перечисляется, что нужно купить, с тем лишь исключением, что в списке покупок каждый элемент обычно размещается на отдельной строке, тогда как в Python они разделяются запятыми.

Список элементов должен быть заключён в квадратные скобки, чтобы Python понял, что это список. Как только список создан, можно добавлять, удалять или искать элементы в нём. Поскольку элементы можно добавлять и удалять, мы говорим, что список – это *изменяемый* тип данных, т.е. его можно модифицировать.

## Краткое введение в объекты и классы

Хотя я и старался до сих пор оттянуть обсуждение объектов и классов, на данном этапе всё же необходимо некоторое пояснение, чтобы вы лучше поняли идею списков. Мы изучим эту тему детально в [*её собственной главе*](#_bookmark144).

Список – это один из примеров использования объектов и классов. Когда мы назначаем некоторой переменной i значение, скажем, целое число 5, это можно представить себе как создание **объекта** (т.е. экземпляра) i **класса** (т.е. типа) int. Чтобы лучше понять это, прочитайте help(int).

Класс может также иметь **методы**, т.е. функции, определённые для использования только применительно к данному классу. Этот функционал будет доступен только когда имеется

1 list – *англ.* «список» (*прим.перев.*)

85

объект данного класса. Например, Python предоставляет метод append для класса list, который позволяет добавлять элемент к концу списка. Так mylist.append('and item') добавит эту строку к списку mylist. Обратите внимание на обозначение точкой для до- ступа к методам объектов.

Класс также может иметь **поля**, которые представляют собой не что иное, как переменные, определённые для использования только применительно к данному классу. Эти перемен- ные/имена можно использовать только тогда, когда имеется объект этого класса. Доступ к полям также осуществляется при помощи точки. Например, mylist.field.

**Пример:** (сохраните как using\_list.py)

*# Это мой список покупок*

shoplist = ['яблоки', 'манго', 'морковь', 'бананы']

print('Я должен сделать', len(shoplist), 'покупки.') print('Покупки:', end=' ')

**for** item **in** shoplist: print(item, end=' ')

print('**\n**Также нужно купить риса.') shoplist.append('рис')

print('Теперь мой список покупок таков:', shoplist)

print('Отсортирую-ка я свой список') shoplist.sort()

print('Отсортированный список покупок выглядит так:', shoplist)

print('Первое, что мне нужно купить, это', shoplist[0]) olditem = shoplist[0]

**del** shoplist[0]

print('Я купил', olditem)

print('Теперь мой список покупок:', shoplist)

#### Вывод:

(продолжение на следующей странице)

$ python3 using\_list.py

Я должен сделать 4 покупки.

Покупки: яблоки манго морковь бананы Также нужно купить риса.

Теперь мой список покупок таков: ['яблоки', 'манго', 'морковь', 'бананы', 'рис']

Отсортирую-ка я свой список

Отсортированный список покупок выглядит так: ['бананы', 'манго', 'морковь', 'рис', 'яблоки']

(продолжение с предыдущей страницы)

Первое, что мне нужно купить, это бананы Я купил бананы

Теперь мой список покупок: ['манго', 'морковь', 'рис', 'яблоки']

#### Как это работает:

Переменная shoplist – это список покупок человека, идущего на рынок. В shoplist мы храним только строки с названиями того, что нужно купить, однако в список можно добавлять *любые объекты*, включая числа или даже другие списки.

Мы также использовали цикл for..in для итерации по элементам списка. Вы уже, наверное, поняли, что список является также и последовательностью. Особенности последовательностей будут рассмотрены [*ниже*](#_bookmark121).

Обратите внимание на использование ключевого аргумента end в функции print, который показывает, что мы хотим закончить вывод пробелом вместо обычного перевода строки.

Далее мы добавляем элемент к списку при помощи append – метода объекта списка, который уже обсуждался ранее. Затем мы проверяем, действительно ли элемент был добавлен к списку, выводя содержимое списка на экран при помощи простой передачи этого списка функции print, которая аккуратно его печатает.

Затем мы сортируем список, используя метод sort объекта списка. Имейте в виду, что этот метод действует на сам список, а не возвращает изменённую его версию. В этом отличие от того, как происходит работа со строками. Именно это имеется в виду, когда мы говорим, что списки *изменяемы*, а строки – *неиз- меняемы*.

Далее после совершения покупки мы хотим удалить её из списка. Это дости- гается применением оператора del. Мы указываем, какой элемент списка мы хотим удалить, и оператор del удаляет его. Мы указываем, что хотим удалить первый элемент списка, и поэтому пишем «del shoplist[0]» (помните, что Python начинает отсчёт с 0).

Чтобы узнать более детально обо всех методах объекта списка, просмотрите help(list).

## Кортеж

Кортежи служат для хранения нескольких объектов вместе. Их можно рассматривать как аналог списков, но без такой обширной функциональности, которую предоставляет класс списка. Одна из важнейших особенностей кортежей заключается в том, что они **неизме- няемы**, так же, как и строки. Т.е. модифицировать кортежи невозможно.

Кортежи обозначаются указанием элементов, разделённых запятыми; по желанию их можно ещё заключить в круглые скобки.

Кортежи обычно используются в тех случаях, когда оператор или пользовательская функ- ция должны наверняка знать, что набор значений, т.е. кортеж значений, не изменится.

**Пример:** (сохраните как using\_tuple.py)

zoo = ('питон', 'слон', 'пингвин') *# помните, что скобки не обязательны*

print('Количество животных в зоопарке -', len(zoo))

new\_zoo = 'обезьяна', 'верблюд', zoo print('Количество клеток в зоопарке -', len(new\_zoo)) print('Все животные в новом зоопарке:', new\_zoo)

print('Животные, привезённые из старого зоопарка:', new\_zoo[2]) print('Последнее животное, привезённое из старого зоопарка -', new\_zoo[2][2]) print('Количество животных в новом зоопарке -', len(new\_zoo)-1 + **\**

len(new\_zoo[2]))

#### Вывод:

$ python3 using\_tuple.py Количество животных в зоопарке - 3 Количество клеток в зоопарке - 3

Все животные в новом зоопарке: ('обезьяна', 'верблюд', ('питон', 'слон', 'пингвин'))

Животные, привезённые из старого зоопарка: ('питон', 'слон', 'пингвин') Последнее животное, привезённое из старого зоопарка - пингвин Количество животных в новом зоопарке - 5

**Как это работает:**

Переменная zoo обозначает кортеж элементов. Как мы видим, функция len позволяет получить длину кортежа. Это также указывает на то, что кортеж является [*последовательностью*](#_bookmark121).

Теперь мы перемещаем этих животных в новый зоопарк, поскольку старый зоопарк закрывается. Поэтому кортеж new\_zoo содержит тех животных, ко- торые уже там, наряду с привезёнными из старого зоопарка. Возвращаясь к реальности, обратите внимание на то, что кортеж внутри кортежа не теряет своей индивидуальности.

Доступ к элементам кортежа осуществляется указанием позиции элемента, заключённой в квадратные скобки – точно так же, как мы это делали для спис- ков. Это называется оператором *индексирования*. Доступ к третьему элементу в new\_zoo мы получаем, указывая new\_zoo[2], а доступ к третьему элементу внутри третьего элемента в кортеже new\_zoo – указывая new\_zoo[2][2]. Это достаточно просто, как только вы поймёте принцип.

#### Скобки

Хотя скобки и не являются обязательными, я предпочитаю всегда указывать их, чтобы было очевидно, что это кортеж, особенно в двусмысленных случаях. Например, print(1, 2,3) и print( (1,2,3) ) делают разные вещи: первое выражение выводит три числа, тогда как второе – кортеж, содержащий эти три числа.

#### Кортеж, содержащий 0 или 1 элемент

Пустой кортеж создаётся при помощи пустой пары скобок – «myempty = ()». Однако, с кортежем из одного элемента не всё так просто. Его нужно указывать при помощи за- пятой после первого (и единственного) элемента, чтобы Python мог отличить кортеж от скобок, окружающих объект в выражении. Таким образом, чтобы получить кортеж, со- держащий элемент 2, вам потребуется указать «singleton = (2,)».

#### Замечание для программистов на Perl

Список внутри списка не теряет своей индивидуальности, т.е. списки не развёртываются, как в Perl. Это же относится к кортежу внутри кортежа, или кортежу внутри списка, или списку внутри кортежа и т.д. В Python все они рассматриваются как объекты, хранящиеся внутри другого объекта – только и всего.

## Словарь

Словарь – это некий аналог адресной книги, в которой можно найти адрес или контакт- ную информацию о человеке, зная лишь его имя; т.е. некоторые **ключи** (имена) связаны со **значениями** (информацией). Заметьте, что ключ должен быть уникальным – вы ведь не сможете получить корректную информацию, если у вас записаны два человека с пол- ностью одинаковыми именами.

Обратите также внимание на то, что в словарях в качестве ключей могут использоваться только неизменяемые объекты (как строки), а в качестве значений можно использовать как неизменяемые, так и изменяемые объекты. Точнее говоря, в качестве ключей должны использоваться только простые объекты.

Пары ключ-значение указываются в словаре следующим образом: «d = {key1 : value1, key2 : value2 }». Обратите внимание, что ключ и значение разделяются двоеточием, а пары друг от друга отделяются запятыми, а затем всё это заключается в фигурные скобки.

Помните, что пары ключ-значение никоим образом не упорядочены в словаре. Если вам необходим некоторый порядок, вам придётся отдельно отсортировать словарь перед об- ращением к нему.

Словари являются экземплярами/объектами класса dict.

**Пример:** (сохраните как using\_dict.py)

*# 'ab' - сокращение от 'a'ddress'b'ook*

ab = { 'Swaroop' : 'swaroop@swaroopch.com', 'Larry' : 'larry@wall.org', 'Matsumoto' : 'matz@ruby-lang.org', 'Spammer' : 'spammer@hotmail.com'

}

print("Адрес Swaroop'а:", ab['Swaroop'])

*# Удаление пары ключ-значение*

**del** ab['Spammer']

print('**\n**В адресной книге *{0}* контакта**\n**'.format(len(ab)))

**for** name, address **in** ab.items():

print('Контакт *{0}* с адресом *{1}*'.format(name, address))

*# Добавление пары ключ-значение*

ab['Guido'] = 'guido@python.org'

**if** 'Guido' **in** ab:

print("**\n**Адрес Guido:", ab['Guido'])

#### Вывод:

(продолжение на следующей странице)

$ python3 using\_dict.py

Адрес Swaroop'а: [swaroop@swaroopch.com](mailto:swaroop@swaroopch.com) В адресной книге 3 контакта

Контакт Swaroop с адресом [swaroop@swaroopch.com](mailto:swaroop@swaroopch.com) Контакт Matsumoto с адресом [matz@ruby-lang.org](mailto:matz@ruby-lang.org) Контакт Larry с адресом [larry@wall.org](mailto:larry@wall.org)

(продолжение с предыдущей страницы)

Адрес Guido: [guido@python.org](mailto:guido@python.org)

#### Как это работает:

Мы создаём словарь ab[2](#_bookmark120) при помощи обозначений, описанных ранее. Затем мы обращаемся к парам ключ-значение, указывая ключ в операторе индек- сирования, которым мы пользовались для списков и кортежей. Как видите, синтаксис прост.

Удалять пары ключ-значение можно при помощи нашего старого доброго оператора del. Мы просто указываем имя словаря и оператор индексирования для удаляемого ключа, после чего передаём это оператору del. Для этой опе- рации нет необходимости знать, какое значение соответствует данному клю- чу.

Далее мы обращаемся ко всем парам ключ-значение нашего словаря, исполь- зуя метод items, который возвращает список кортежей, каждый из которых содержит пару элементов: ключ и значение. Мы получаем эту пару и при- сваиваем её значение переменным name и address соответственно в цикле for..in, а затем выводим эти значения на экран в блоке for.

Новые пары ключ-значение добавляются простым обращением к нужному ключу при помощи оператора индексирования и присваиванием ему неко- торого значения, как мы сделали для Guido в примере выше.

Проверить, существует ли пара ключ-значение, можно при помощи оператора

in.

Чтобы просмотреть список всех методов класса dict смотрите help(dict).

#### Ключевые Аргументы и Словари

К слову, если вы использовали ключевые аргументы в ваших функциях, вы уже исполь- зовали словари! Только подумайте: вы указали пару ключ-значение среди параметров функции при её определении, а когда обращаетесь к переменным внутри функции, то это, фактически, обращение по ключу к словарю (который в терминах разработчиков ком- пиляторов называется *таблицей имён*).

2 **a**ddress **b**ook – *англ.* «адресная книга» (*прим. перев.*)

## Последовательности

Списки, кортежи и строки являются примерами последовательностей. Но что такое по- следовательности и что в них такого особенного?

Основные возможности – это **проверка принадлежности** (т.е. выражения «in» и «not in») и **оператор индексирования**, позволяющий получить напрямую некоторый эле- мент последовательности.

Все три типа последовательностей, упоминавшиеся выше (списки, кортежи и строки), также предоставляют операцию получения **вырезки**, которая позволяет получить вырез- ку последовательности, т.е. её фрагмент.

**Пример:** (сохраните как seq.py)

shoplist = ['яблоки', 'манго', 'морковь', 'бананы'] name = 'swaroop'

*# Операция индексирования* print('Элемент 0 -', shoplist[0]) print('Элемент 1 -', shoplist[1]) print('Элемент 2 -', shoplist[2]) print('Элемент 3 -', shoplist[3]) print('Элемент -1 -', shoplist[-1])

print('Элемент -2 -', shoplist[-2])

print('Символ 0 -', name[0])

*# Вырезка из списка*

print('Элементы с 1 по 3:', shoplist[1:3]) print('Элементы с 2 до конца:', shoplist[2:]) print('Элементы с 1 по -1:', shoplist[1:-1]) print('Элементы от начала до конца:', shoplist[:])

*# Вырезка из строки*

print('Символы с 1 по 3:', name[1:3]) print('Символы с 2 до конца:', name[2:]) print('Символы с 1 до -1:', name[1:-1]) print('Символы от начала до конца:', name[:])

#### Вывод:

(продолжение на следующей странице)

$ python3 seq.py Элемент 0 - яблоки Элемент 1 - манго Элемент 2 - морковь Элемент 3 - бананы Элемент -1 - бананы

(продолжение с предыдущей страницы)

Элемент -2 - морковь Символ 0 - s

Элементы с 1 по 3: ['манго', 'морковь'] Элементы с 2 до конца: ['морковь', 'бананы'] Элементы с 1 по -1: ['манго', 'морковь']

Элементы от начала до конца: ['яблоки', 'манго', 'морковь', 'бананы'] Символы с 1 по 3: wa

Символы с 2 до конца: aroop Символы с 1 до -1: waroo

Символы от начала до конца: swaroop

#### Как это работает:

Прежде всего, мы видим, как использовать индексы для получения отдельных элементов последовательности. Это ещё называют *приписыванием индекса*. Ко- гда мы указываем число в квадратных скобках после последовательности, как показано выше, Python извлекает элемент, соответствующий указанной пози- ции в последовательности. Помните, что Python начинает отсчёт с 0. Поэтому shoplist[0] извлекает первый элемент, а shoplist[3] – четвёртый элемент последовательности shoplist.

Индекс также может быть отрицательным числом. В этом случае позиция от- считывается от конца последовательности. Поэтому shoplist[-1] указывает на последний элемент последовательности shoplist, а shoplist[-2] – на предпоследний.

Операция вырезки производится при помощи указания имени последова- тельности, за которым может следовать пара чисел, разделённых двоеточием и заключённых в квадратные скобки. Заметьте, как это похоже на операцию индексирования, которой мы пользовались до сих пор. Помните, что числа в скобках необязательны, тогда как двоеточие – обязательно.

Первое число (перед двоеточием) в операции вырезки указывает позицию, с которой вырезка должна начинаться, а второе число (после двоеточия) указы- вает, где вырезка должна закончиться. Если первое число не указано, Python начнёт вырезку с начала последовательности. Если пропущено второе число, Python закончит вырезку у конца последовательности. Обратите внимание, что полученная вырезка будет *начинаться* с указанной начальной позиции, а *заканчиваться* прямо перед указанной конечной позицией, т.е. начальная позиция будет включена в вырезку, а конечная – нет.

Таким образом, shoplist[1:3] возвращает вырезку из последовательности, начинающуюся с позиции 1, включает позицию 2, но останавливается на позиции 3, и поэтому возвращает *вырезку* из двух элементов. Аналогично, shoplist[:] возвращает копию всей последовательности.

Вырезка может осуществляться и с отрицательными значениями. Отрица-

тельные числа обозначают позицию с конца последовательности. Например, shoplist[:-1] вернёт вырезку из последовательности, исключающую по- следний элемент, но содержащую все остальные.

Кроме того, можно также указать третий аргумент для вырезки, который будет обозначать *шаг* вырезки (по умолчанию шаг вырезки равен 1):

**>>>** shoplist = ['яблоки', 'манго', 'морковь', 'бананы']

**>>>** shoplist[::1]

['яблоки', 'манго', 'морковь', 'бананы']

**>>>** shoplist[::2] ['яблоки', 'морковь']

**>>>** shoplist[::3] ['яблоки', 'бананы']

**>>>** shoplist[::-1]

['бананы', 'морковь', 'манго', 'яблоки']

Обратите внимание на то, что когда шаг равен 2, мы получаем элементы, на- ходящиеся на позициях 0, 2, … Когда шаг равен 3, мы получаем элементы с позиций 0, 3, … и т.д.

Попробуйте разные комбинации параметров вырезки, используя интерактивную оболоч- ку интерпретатора Python, т.е. его командную строку, чтобы сразу видеть результат. По- следовательности замечательны тем, что они дают возможность обращаться к кортежам, спискам и строкам одним и тем же способом!

## Множество

Множества – это *неупорядоченные* наборы простых объектов. Они необходимы тогда, ко- гда присутствие объекта в наборе важнее порядка или того, сколько раз данный объект там встречается.

Используя множества, можно осуществлять проверку принадлежности, определять, яв- ляется ли данное множество подмножеством другого множества, находить пересечения множеств и так далее.

(продолжение на следующей странице)

**>>>** bri = set(['Бразилия', 'Россия', 'Индия'])

**>>>** 'Индия' **in** bri True

**>>>** 'США' **in** bri False

**>>>** bric = bri.copy()

**>>>** bric.add('Китай')

**>>>** bric.issuperset(bri) True

(продолжение с предыдущей страницы)

**>>>** bri.remove('Россия')

**>>>** bri & bric *# OR bri.intersection(bric)*

{'Бразилия', 'Индия'}

#### Как это работает:

Этот пример достаточно нагляден, так как использует основы теории мно- жеств из школьного курса математики.

## Ссылки

Когда мы создаём объект и присваиваем его переменной, переменная только *ссылается* на объект, а не представляет собой этот объект! То есть имя переменной указывает на ту часть памяти компьютера, где хранится объект. Это называется **привязкой** имени к объекту.

Обычно вам не следует об этом беспокоиться, однако есть некоторый неочевидный эф- фект, о котором нужно помнить:

**Пример:** (сохраните как reference.py)

print('Простое присваивание')

shoplist = ['яблоки', 'манго', 'морковь', 'бананы']

mylist = shoplist *# mylist - лишь ещё одно имя, указывающее на тот же объект!*

**del** shoplist[0] *# Я сделал первую покупку, поэтому удаляю её из списка*

print('shoplist:', shoplist) print('mylist:', mylist)

*# Обратите внимание, что и shoplist, и mylist выводят один и тот же список # без пункта "яблоко", подтверждая тем самым, что они указывают на один*

*# объект.*

print('Копирование при помощи полной вырезки')

mylist = shoplist[:] *# создаём копию путём полной вырезки*

**del** mylist[0] *# удаляем первый элемент*

print('shoplist:', shoplist) print('mylist:', mylist)

*# Обратите внимание, что теперь списки разные*

#### Вывод:

$ python3 reference.py Простое присваивание

shoplist: ['манго', 'морковь', 'бананы'] mylist: ['манго', 'морковь', 'бананы'] Копирование при помощи полной вырезки shoplist: ['манго', 'морковь', 'бананы'] mylist: ['морковь', 'бананы']

**Как это работает:**

Большая часть объяснения содержится в комментариях.

Помните, что если вам нужно сделать копию списка или подобной последо- вательности, или другого сложного объекта (не такого простого *объекта*, как целое число), вам следует воспользоваться операцией вырезки. Если вы про- сто присвоите имя переменной другому имени, оба они будут *ссылаться* на один и тот же объект, а это может привести к проблемам, если вы не осторож- ны.

## Ещё о строках

Мы уже детально обсуждали строки ранее. Что же ещё можно о них узнать? Что ж, вы знали, например, что строки также являются объектами и имеют методы, при помощи которых можно делать практически всё: от проверки части строки до удаления краевых пробелов?

Все строки, используемые вами в программах, являются объектами класса str. Некоторые полезные методы этого класса продемонстрированы на примере ниже. Чтобы посмотреть весь список методов, выполните help(str).

**Пример:** (сохраните как str\_methods.py)

(продолжение на следующей странице)

name = 'Swaroop' *# Это объект строки*

**if** name.startswith('Swa'):

print('Да, строка начинается на "Swa"')

**if** 'a' **in** name:

print('Да, она содержит строку "a"')

**if** name.find('war') != -1:

print('Да, она содержит строку "war"')

delimiter = '\_\*\_'

(продолжение с предыдущей страницы)

mylist = ['Бразилия', 'Россия', 'Индия', 'Китай'] print(delimiter.join(mylist))

#### Вывод:

$ python3 str\_methods.py

Да, строка начинается на "Swa" Да, она содержит строку "a" Да, она содержит строку "war"

Бразилия\_\*\_Россия\_\*\_Индия\_\*\_Китай

**Как это работает:**

Здесь мы видим сразу несколько методов строк в действии. Метод startswith служит для того, чтобы определять, начинается ли строка с некоторой заданной подстроки. Опе- ратор in используется для проверки, является ли некоторая строка частью данной строки.

Метод find используется для определения позиции данной подстроки в строке; find воз- вращает -1, если подстрока не обнаружена. В классе str также имеется отличный метод для объединения (join)[3](#_bookmark126) элементов последовательности с указанной строкой в качестве разделителя между элементами, возвращающий большую строку, сгенерированную та- ким образом.

## Резюме

Мы детально рассмотрели различные встроенные структуры данных Python. Эти струк- туры данных будут крайне важны для написания программ существенного размера.

Теперь, когда мы накопили достаточно базовых знаний о Python, далее посмотрим, как проектировать и писать настоящую программу на Python.

3 join – *англ.* «объединять» (*прим.перев.*)

**12.9. Резюме** 97

# Решение задач

Мы рассмотрели различные части языка Python, и теперь посмотрим, как все эти части ра- ботают вместе, проектируя и составляя программу, которая *делает* что-то полезное. Цель состоит в том, чтобы научиться писать сценарии на языке Python самостоятельно.

## Задача

Перед нами стоит следующая задача: *Составить программу, которая создаёт резервные копии всех наших важных файлов*.

Хотя задача и проста, информации явно недостаточно, чтобы приступать к её решению. Необходим некоторый дополнительный **анализ**. Например, как мы выберем, *какие* фай- лы необходимо копировать? *Как* их хранить? *Где* их хранить?

После надлежащего анализа мы **проектируем** нашу программу. Мы создаём список, опи- сывающий то, как наша программа должна работать. В данном случае я создал список того, как *я* себе представляю её работу. Когда вы проектируете программу, у вас может по- лучиться другой результат, поскольку каждый человек представляет себе это по-своему, так что это в порядке вещей.

* + 1. Файлы и каталоги, которые необходимо скопировать, собираются в список.
    2. Резервные копии должны храниться в основном каталоге резерва.
    3. Файлы помещаются в zip-архив.
    4. Именем для zip-архива служит текущая дата и время.
    5. Будем использовать стандартную команду zip, имеющуюся по умолчанию в любом стандартном дистрибутиве GNU/Linux. Пользователи Windows могут [установить](http://gnuwin32.sourceforge.net/downlinks/zip.php) её со [страницы проекта GnuWin32](http://gnuwin32.sourceforge.net/packages/zip.htm) и добавить «C:\Program Files\GnuWin32\bin» к системной переменной окружения PATH, аналогично тому, как мы это [*делали*](#_bookmark30) для самой команды «python». Обратите внимание, что для этого подойдёт любая коман- да архивации, если у неё есть интерфейс командной строки, чтобы ей можно было передавать аргументы из нашего сценария.

## Решение

Как только проект программы более-менее устоялся, можно приступать к написанию ко- да, который и будет являться **реализацией** нашего решения.

Сохраните как backup\_ver1.py:

**import os import time**

*# 1. Файлы и каталоги, которые необходимо скопировать, собираются в список.*

source = ['"C:**\\**My Documents"', 'C:**\\**Code']

*# Заметьте, что для имён, содержащих пробелы, необходимо использовать # двойные кавычки внутри строки.*

*# 2. Резервные копии должны храниться в основном каталоге резерва.*

target\_dir = 'E:**\\**Backup' *# Подставьте ваш путь.*

*# 3. Файлы помещаются в zip-архив.*

*# 4. Именем для zip-архива служит текущая дата и время.*

target = target\_dir + os.sep + time.strftime('%Y%m*%d*%H%M%S') + '.zip'

*# 5. Используем команду "zip" для помещения файлов в zip-архив*

zip\_command = "zip -qr *{0} {1}*".format(target, ' '.join(source))

*# Запускаем создание резервной копии*

**if** os.system(zip\_command) == 0:

print('Резервная копия успешно создана в', target)

**else**:

print('Создание резервной копии НЕ УДАЛОСЬ')

#### Вывод:

$ python3 backup\_ver1.py

Резервная копия успешно создана в E:\\Backup\\20080702185040.zip

Теперь наступает стадия **тестирования**, когда мы проверяем, правильно ли работает наша программа. Если она работает не так, как ожидалось, нам придётся заняться её **отладкой** (дебагом)[1](#_bookmark130), т.е. устранением **багов** (ошибок) в программе.

Если приведённая выше программа у вас не заработает, допишите print(zip\_command)

прямо перед вызовом os.system и запустите программу. После этого скопируйте вы-

1 debug – применительно к компьютерным программам обозначает [отладку](https://ru.wikipedia.org/wiki/Отладка_программы) (обнаружение и устранение ошибок, которые при этом принято называть «bug», т.е. «жук»). По всей видимости, это [берёт своё начало](https://ru.wikipedia.org/wiki/Баг#Этимология) с процедуры изгнания насекомых из схем больших ЭВМ, хотя само понятие «bug» в смысле маленькой неисправности встречается и в более ранней литературе, например, в записях Томаса Эдисона 1878 года. (*прим. перев.*)

веденную команду «zip\_command» и вставьте её в командную строку, чтобы проверить, работает ли она корректно сама по себе. Если она не срабатывает, проверьте справку по команде «zip», чтобы выяснить, в чём может быть проблема. Если команда успешно вы- полняется, проверьте, совпадает ли ваша программа на Python в точности с программой, приведённой выше.

#### Как это работает:

Вы заметили, как мы превратили наш *проект* в *код* шаг за шагом.

Мы использовали модули os и time, предварительно импортировав их. Далее мы указали файлы и каталоги для резервного копирования в списке source[2](#_bookmark131). Каталог назначения – это каталог, в котором мы сохраняем все резервные ко- пии, и он указывается в переменной target\_dir. Именем zip-архива, кото- рый мы создаём, будет текущая дата и время, которые генерируются при по- мощи функции time.strftime(). У него будет расширение .zip, и хранить- ся он будет в каталоге target\_dir.

Обратите внимание на употребление переменной os.sep – она содержит раз- делитель пути для конкретной операционной системы, т.е. он будет '/' в GNU/Linux и Unix[3](#_bookmark132), '\\' в Windows и ':' в Mac OS. Использование os.sep вместо этих символов напрямую делает программу переносимой, и она смо- жет работать на всех этих операционных системах.

Функция time.strftime() принимает в качестве аргумента формат вывода времени, например, такой, как мы указали в программе выше. Символ форма- та %Y будет замещён годом и столетием. Символ %m будет замещён месяцем в форме числа от 01 до 12, и так далее. Полный список таких символов формата можно найти в [справочнике по Python](http://docs.python.org/py3k/library/time.html#time.strftime).

Имя конечного zip-файла мы создаём при помощи оператора, который *соеди- няет* строки, т.е. объединяет две строки и возвращает новую. После этого мы создаём строку zip\_command, которая содержит команду, которую мы наме- рены выполнить. Проверить, работает ли эта команда, можно запустив её от- дельно в командной оболочке (терминал в GNU/Linux или командная строка DOS).

Команда zip, которую мы используем, имеет некоторые параметры. Параметр

«-q» используется для указания, что команда должна сработать тихо[4](#_bookmark133). Пара- метр «-r» обозначает, что команда архивации должна работать рекурсивно[5](#_bookmark134) для каталогов, т.е. должна включать все подкаталоги и файлы. Оба парамет- ра объединены и указаны в краткой форме «-qr». За параметрами следует

2 source – *англ.* «источник» (*прим.перев.*)

3 Под словом «Unix» здесь подразумеваются все операционные системы, построенные по принципам ОС Unix, а не только она сама по себе. Примерами таких операционных систем являются все дистрибутивы GNU/Linux, семейство ОС \*BSD, Android, Solaris и т.д. (*прим.перев.*)

4 **q**uietly – *англ.* «тихо» (*прим.перев.*)

5 **r**ecursive – *англ.* «рекурсивно» (*прим.перев.*)

имя создаваемого zip-архива, за которым указывается список файлов и ката- логов для резервного копирования. Мы превращаем список source в строку, используя уже знакомый нам метод join.

Затем мы, наконец, *выполняем* команду при помощи функции os.system, ко- торая запускает команду так, как будто она была запущена из *системы*, т.е. из командной оболочки. Она возвращает 0, если команда выполнена успешно, в противном случае она возвращает код ошибки.

В зависимости от вывода команды, мы печатаем соответствующее сообщение о том, успешным было создание резервных копий или нет.

Вот и всё, мы создали сценарий для сохранения резервных копий наших важ- ных файлов!

#### Замечание для пользователей Windows

Вместо управляющей последовательности для обратной наклонной черты могут ис- пользоваться «сырые»[6](#_bookmark136) строки. Например, можно писать «C:\\Documents» или «r'C:\ Documents'». Однако, **не** используйте «'C:\Documents'», так как в этом случае окажет- ся, что вы пытаетесь применить неизвестную управляющую последовательность \D.

Теперь, когда у нас есть рабочий сценарий резервного копирования, мы можем использо- вать его для создания копий наших файлов. Пользователям GNU/Linux и Unix рекоменду- ется сделать этот программный файл [*исполнимым*](#_bookmark39), чтобы иметь возможность запускать его в любое время из любого места. Это называется **операционной** фазой или **развёрты- ванием** программы.

Программа, приведённая выше, работает корректно, но (обычно) поначалу программы не работают так, как вы того ожидаете. Проблемы могут возникать вследствие неправиль- ного проектирования программы, допущения ошибки при наборе программного кода и т.д. В таких случаях приходится возвращаться к стадии проектирования или отладки про- граммы.

## Вторая версия

Первая версия нашего сценария работает. Тем не менее, его можно улучшить так, чтобы было удобнее пользоваться в повседневной работе. Это называется стадией **поддержки** программы.

Одно из улучшений, показавшееся мне полезным, – это лучший механизм именования файлов: использование *времени* в качестве имени файла, сохраняющегося в каталог с те- кущей датой в качестве имени, который в свою очередь, расположен в главном каталоге

6 raw – *англ.* «сырой», «необработанный» (*прим.перев*)

для хранения резервных копий. Первое достоинство этого состоит в том, что копии хра- нятся в иерархической структуре, которой легче управлять. Второе достоинство – в том, что имена файлов намного короче. Третье достоинство состоит в том, что по именам ката- логов можно легко определить, в какие дни создавались резервные копии, так как каталог создаётся только в случае резервного копирования данных в этот день.

Сохраните как backup\_ver2.py:

**import os import time**

*# 1. Файлы и каталоги, которые необходимо скопировать, собираются в список.*

source = ['"C:**\\**My Documents"', 'C:**\\**Code']

*# Заметьте, что для имён, содержащих пробелы, необходимо использовать # двойные кавычки внутри строки.*

*# 2. Резервные копии должны храниться в основном каталоге резерва.*

target\_dir = 'E:**\\**Backup' *# Подставьте ваш путь.*

*# 3. Файлы помещаются в zip-архив.*

*# 4. Текущая дата служит именем подкаталога в основном каталоге*

today = target\_dir + os.sep + time.strftime('%Y%m*%d*')

*# Текущее время служит именем zip-архива*

now = time.strftime('%H%M%S')

*# Создаём каталог, если его ещё нет*

**if not** os.path.exists(today): os.mkdir(today) *# создание каталога* print('Каталог успешно создан', today)

*# Имя zip-файла*

target = today + os.sep + now + '.zip'

*# 5. Используем команду "zip" для помещения файлов в zip-архив*

zip\_command = "zip -qr *{0} {1}*".format(target, ' '.join(source))

*# Запускаем создание резервной копии*

**if** os.system(zip\_command) == 0:

print('Резервная копия успешно создана в', target)

**else**:

print('Создание резервной копии НЕ УДАЛОСЬ')

#### Вывод:

(продолжение на следующей странице)

$ python3 backup\_ver2.py

Каталог успешно создан E:\\Backup\\20080702

Резервная копия успешно создана в E:\\Backup\\20080702\\202311.zip

(продолжение с предыдущей страницы)

$ python3 backup\_ver2.py

Резервная копия успешно создана в E:\\Backup\\20080702\\202325.zip

#### Как это работает:

Большая часть программы осталась прежней. Разница в том, что теперь мы проверяем, существует ли каталог с именем, соответствующем текущей дате, внутри главного каталога для хранения резервных копий. Для этого мы ис- пользуем функцию os.path.exists. Если он не существует, мы создаём его функцией os.mkdir.

## Третья версия

Вторая версия уже удобнее для работы с большим количеством резервных копий. С дру- гой стороны, когда их много, становится трудно отличить, какая копия для чего. Напри- мер, мы могли внести значительные изменения в какую-то программу или презентацию, и теперь хотим указать суть этих изменений в имени zip-архива. Этого легко можно до- стичь добавлением пользовательского комментария к имени zip-архива.

**Примечание:** Следующая программа не работает, так что не переживайте, просто про- следуйте по ней, так как в ней содержится урок.

Сохраните как backup\_ver3.py

(продолжение на следующей странице)

**import os import time**

*# 1. Файлы и каталоги, которые необходимо скопировать, собираются в список.*

source = ['"C:**\\**My Documents"', 'C:**\\**Code']

*# Заметьте, что для имён, содержащих пробелы, необходимо использовать # двойные кавычки внутри строки.*

*# 2. Резервные копии должны храниться в основном каталоге резерва.*

target\_dir = 'E:**\\**Backup' *# Подставьте ваш путь.*

*# 3. Файлы помещаются в zip-архив.*

*# 4. Текущая дата служит именем подкаталога в основном каталоге*

today = target\_dir + os.sep + time.strftime('%Y%m*%d*')

*# Текущее время служит именем zip-архива*

now = time.strftime('%H%M%S')

(продолжение с предыдущей страницы)

*# Запрашиваем комментарий пользователя для имени файла*

comment = input('Введите комментарий --> ')

**if** len(comment) == 0: *# проверяем, введён ли комментарий*

target = today + os.sep + now + '.zip'

**else**:

target = today + os.sep + now + '\_' + comment.replace(' ', '\_') + '.zip'

*# Создаём каталог, если его ещё нет*

**if not** os.path.exists(today): os.mkdir(today) *# создание каталога* print('Каталог успешно создан', today)

*# 5. Используем команду "zip" для помещения файлов в zip-архив*

zip\_command = "zip -qr *{0} {1}*".format(target, ' '.join(source))

*# Запускаем создание резервной копии*

**if** os.system(zip\_command) == 0:

print('Резервная копия успешно создана в', target)

**else**:

print('Создание резервной копии НЕ УДАЛОСЬ')

#### Вывод:

$ python3 backup\_ver3.py

File "backup\_ver3.py", line 25

target = today + os.sep + now + '\_' +

^

SyntaxError: invalid syntax

**Как это (не) работает:**

*Эта программа не работает!* Python сообщает об обнаружении ошибки син- таксиса, что означает, что сценарий не удовлетворяет структуре, которую ожи- дает увидеть Python. Когда Python выдаёт сообщение об ошибке, он также ука- зывает нам на место ошибки. Так что мы начинаем *отладку* программы с этой строки.

При внимательном рассмотрении, мы видим, что одна логическая строка бы- ла разбита на две физические строки, но мы не указали, что эти две физи- ческие строки являются частью одной. На деле же Python просто обнаружил оператор сложения (+) без соответствующего операнда в той же логической строке, а поэтому не знает, как продолжать. Помните, что мы можем указать, что логическая строка продолжается на следующей физической при помощи обратной наклонной черты в конце физической строки. Внесём это исправ- ление в нашу программу. Коррекция программы при обнаружении ошибок и

называется **отладкой**[7](#_bookmark139).

## Четвёртая версия

Сохраните как backup\_ver4.py

**import os import time**

*# 1. Файлы и каталоги, которые необходимо скопировать, собираются в список.*

source = ['"C:**\\**My Documents"', 'C:**\\**Code']

*# Заметьте, что для имён, содержащих пробелы, необходимо использовать # двойные кавычки внутри строки.*

*# 2. Резервные копии должны храниться в основном каталоге резерва.*

target\_dir = 'E:**\\**Backup' *# Подставьте ваш путь.*

*# 3. Файлы помещаются в zip-архив.*

*# 4. Текущая дата служит именем подкаталога в основном каталоге*

today = target\_dir + os.sep + time.strftime('%Y%m*%d*')

*# Текущее время служит именем zip-архива*

now = time.strftime('%H%M%S')

*# Запрашиваем комментарий пользователя для имени файла*

comment = input('Введите комментарий --> ')

**if** len(comment) == 0: *# проверяем, введён ли комментарий*

target = today + os.sep + now + '.zip'

**else**:

target = today + os.sep + now + '\_' + \ comment.replace(' ', '\_') + '.zip'

*# Создаём каталог, если его ещё нет*

**if not** os.path.exists(today): os.mkdir(today) *# создание каталога* print('Каталог успешно создан', today)

*# 5. Используем команду "zip" для помещения файлов в zip-архив*

zip\_command = "zip -qr *{0} {1}*".format(target, ' '.join(source))

*# Запускаем создание резервной копии*

**if** os.system(zip\_command) == 0:

print('Резервная копия успешно создана в', target)

**else**:

print('Создание резервной копии НЕ УДАЛОСЬ')

7 bug fixing – устранение «багов», исправление ошибок (*прим.перев*)

#### 13.5. Четвёртая версия 105

**Вывод:**

$ python3 backup\_ver4.py

Введите комментарий --> added new examples

Резервная копия успешно создана в E:\Backup\20080702\202836\_added\_new\_

*‹→*examples.zip

$ python3 backup\_ver4.py Введите комментарий -->

Резервная копия успешно создана в E:\Backup\20080702\202839.zip

#### Как это работает:

Теперь эта программа работает! Давайте просмотрим все улучшения, сделан- ные нами для версии 3. Мы запрашиваем пользовательский комментарий при помощи функции input, а затем проверяем, ввёл ли пользователь что-либо, определяя длину введённой строки функцией len. Если пользователь просто нажал ENTER, не вводя никакого текста (может быть, это было регулярное со- здание резервной копии, или никаких особых изменений внесено не было), мы продолжаем так же, как делали до сих пор.

Если же комментарий был введён, он добавляется к имени zip-архива перед расширением .zip. Обратите внимание, что мы заменяем пробелы в коммен- тарии подчёркиваниями: управлять файлами без пробелов в именах намного легче.

## Дополнительные усовершенствования

Четвёртая версия – вполне удовлетворительный рабочий сценарий для большинства пользователей, однако нет пределов совершенства. Например, в программу можно доба- вить уровень *подробности*[8](#_bookmark141) вывода, чтобы при указании параметра «-v» она становилась более «разговорчивой».

Ещё одним возможным улучшением была бы возможность передавать сценарию другие файлы и каталоги прямо в командной строке. Эти имена можно получать из списка sys. argv и добавлять к нашему списку source при помощи метода extend класса list.

Наиболее важным усовершенствованием было бы прекращение использования os. system для создания архивов, а применение вместо него встроенных модулей zipfile или tarfile. Они являются частью стандартной библиотеки, поэтому всегда доступны для использования без зависимости от внешней программы zip на компьютере.

8 verbosity – *англ.* «многословность». Применительно к компьютерным программам обозначает степень подробности выводимых программой сообщений, т.е. степень «разговорчивости» программы. Отсюда и название этого параметра (*прим.перев*)

В приведённых примерах мы использовали способ с os.system для создания резервных копий исключительно в педагогических целях, чтобы пример был достаточно прост для понимания любым читателем, но достаточно реален для того, чтобы делать что-то полез- ное.

Попробуйте написать пятую версию с использованием модуля [zipfile](http://docs.python.org/py3k/library/zipfile.html) вместо вызова os. system.

## Процесс разработки программного обеспече- ния

В процессе создания программы мы прошли через несколько **стадий**. Эти стадии можно свести примерно в такой список:

* Что (Анализ)
* Как (Проектирование)
* Создание (Реализация)
* Тестирование (Тестирование и Отладка)
* Использование (Развёртывание и Оперирование)
* Поддержка (Усовершенствование)

Процедура, которую мы прошли при написании сценария создания резервных копий ре- комендуется и для других программ: Проведите анализ и проектирование. Начните ре- ализацию с простейшей версии. Протестируйте и отладьте её. Попользуйтесь ею, чтобы убедиться, что она работает, как ожидалось. После этого добавляйте любые необходимые функции, повторяя цикл «Создание-Тестирование-Использование» столько раз, сколько потребуется. Помните, **Программы выращиваются, а не строятся**.

## Резюме

Мы увидели, как создавать свои собственные программы/сценарии на Python, а также различные стадии написания программ. На данном этапе вам будет полезно создать соб- ственную программу по такому рецепту, как мы это делали в настоящей главе, чтобы лучше привыкнуть к Python, равно как и к решению задач.

Далее мы обсудим объектно-ориентированное программирование.

# Объектно-ориентированное программирование

До сих пор наши программы состояли из функций, т.е. блоков выражений, которые мани- пулируют данными. Это называется *процедурно-ориентированным* стилем программиро- вания. Существует и другой способ организации программ: объединять данные и функ- ционал внутри некоего объекта. Это называется *объектно-ориентированной* парадигмой программирования. В большинстве случаев можно ограничиться процедурным програм- мированием, а при написании большой программы или если решение конкретной зада- чи того требует, можно переходить к техникам объектно-ориентированного программи- рования.

Два основных аспекта объектно-ориентированного программирования – классы и объ- екты. **Класс** создаёт новый *тип*, а **объекты** являются *экземплярами* класса. Аналогично, когда мы говорим о «переменных типа int», это означает, что переменные, которые хра- нят целочисленные значения, являются экземплярами (объектами) класса int.

#### Замечание для программистов на статических языках

Обратите внимание, что даже целые числа рассматриваются как объекты (класса int), в отличие от C++ и Java (до версии 1.5), где целые числа являются примитивами. См. help(int) для более детального описания этого класса. Программисты на C# и Java 1.5 могут заметить сходство с концепцией *упаковки и распаковки*[1](#_bookmark145).

Объекты могут хранить данные в обычных переменных, которые *принадлежат* объек- ту. Переменные, принадлежащие объекту или классу, называют **полями**. Объекты могут также обладать функционалом, т.е. иметь функции, *принадлежащие* классу. Такие функ- ции принято называть **методами** класса. Эта терминология важна, так как она помогает нам отличать независимые функции и переменные от тех, что принадлежат классу или объекту. Всё вместе (поля и методы) принято называть **атрибутами** класса.

Поля бывают двух типов: они могут принадлежать каждому отдельному экземпляру объ- екта класса или всему классу. Они называются **переменными экземпляра** и **перемен- ными класса** соответственно.

1 boxing and unboxing

Класс создаётся ключевым словом class. Поля и методы класса записываются в блоке кода с отступом.

## self

Методы класса имеют одно отличие от обычных функций: они должны иметь дополни- тельно имя, добавляемое к началу списка параметров. Однако, при вызове метода никако- го значения этому параметру присваивать **не** нужно – его укажет Python. Эта переменная указывает на **сам** объект экземпляра класса, и по традиции она называется self[2](#_bookmark148).

Хотя этому параметру можно дать любое имя, *настоятельно рекомендуется* использовать только имя self; использование любого другого имени не приветствуется. Есть много до- стоинств использования стандартного имени: во-первых, любой человек, просматриваю- щий вашу программу, легко узнает его; во-вторых, некоторые специализированные Ин- тегрированные среды разработки (IDE) изначально рассчитаны на использование self.

#### Замечание для программистов на C++, Java и C#

self в Python эквивалентно указателю this в C++ и ссылке this в Java и C#.

Вы, должно быть, удивляетесь, как Python присваивает значение self и почему вам не нужно указывать это значение самостоятельно. Поясним это на примере. Предположим, у нас есть класс с именем MyClass и экземпляр этого класса с именем myobject. При вызове метода этого объекта, например, «myobject.method(arg1, arg2)», Python ав- томатически превращает это в «MyClass.method(myobject, arg1, arg2)» – в этом и состоит смысл self.

Это также означает, что если какой-либо метод не принимает аргументов, у него всё равно будет один аргумент – self.

## Классы

Простейший класс показан в следующем примере (сохраните как simplestclass.py).

**class Person**:

**pass** *# Пустой блок*

p = Person() print(p)

#### Вывод:

2 self – *англ.* «сам» (*прим.перев.*)

$ python3 simplestclass.py

< main .Person object at 0x019F85F0>

#### Как это работает:

Мы создаём новый класс при помощи оператора class и имени класса. За этим следует блок выражений, формирующих тело класса. В данном случае блок у нас пуст, на что указывает оператор pass.

Далее мы создаём объект-экземпляр класса, записывая имя класса со скобка- ми. (Мы узнаем больше о [*реализации*](#_bookmark150) в следующем разделе). Для проверки мы выясняем тип переменной, просто выводя её на экран. Так мы видим, что у нас есть экземпляр класса Person в модуле main .

Обратите внимание, что выводится также и адрес в памяти компьютера, где хранится ваш объект. На вашем компьютере адрес будет другим, так как Python хранит объекты там, где имеется свободное место.

## Методы объектов

Итак, мы выяснили что классы/объекты могут иметь методы, представляющие собой функции, за исключением дополнительной переменной self. А теперь давайте рассмот- рим пример (сохраните как method.py).

**class Person**:

**def** sayHi(self): print('Привет! Как дела?')

p = Person() p.sayHi()

*# Этот короткий пример можно также записать как Person().sayHi()*

#### Вывод:

$ python3 method.py Привет! Как дела?

**Как это работает:**

Здесь мы видим self в действии. Обратите внимание, что метод sayHi не принимает параметров, но тем не менее, имеет self в определении функции.

* 1. Метод **init**

Существует много методов, играющих специальную роль в классах Python. Сейчас мы увидим значительность метода init .

Метод init запускается, как только объект класса реализуется. Этот метод полезен для осуществления разного рода *инициализации*, необходимой для данного объекта. Об- ратите внимание на двойные подчёркивания в начале и в конце имени.

**Пример:** (сохраните как oop\_init.py)

**class Person**:

**def** init (self, name): self.name = name

**def** say\_hi(self):

print('Привет! Меня зовут', self.name)

p = Person('Swaroop') p.say\_hi()

*# Предыдущие 2 строки можно # Person('Swaroop').say\_hi()*

#### Вывод:

$ python oop\_init.py Привет! Меня зовут Swaroop

**Как это работает:**

Здесь мы определяем метод init так, чтобы он принимал параметр name (наряду с обычным self). Далее мы создаём новое поле с именем name. Об- ратите внимание, что это две разные переменные, даже несмотря на то, что они обе названы name. Это не проблема, так как точка в выражении self. name обозначает, что существует нечто с именем «name», являющееся частью объекта «self», и другое name – локальная переменная. Поскольку мы в явном виде указываем, к которому имени мы обращаемся, путаницы не возникнет.

Для создания нового экземпляра p класса Person мы указываем имя класса, после которого – аргументы в скобках: p = Person('Swaroop').

Метод init мы при этом не вызываем явным образом. В этом и заклю- чается специальная роль данного метода.

После этого мы получаем возможность использовать поле self.name в наших методах, что и продемонстрировано в методе say\_hi.

## Переменные класса и объекта

Функциональную часть классов и объектов (т.е. методы) мы обсудили, теперь давайте ознакомимся с частью данных. Данные, т.е. поля, являются не чем иным, как обычны- ми переменными, *заключёнными* в **пространствах имён** классов и объектов. Это означа- ет, что их имена действительны только в контексте этих классов или объектов. Отсюда и название «*пространство имён*».

Существует два типа *полей*: переменные класса и переменные объекта, которые разли- чаются в зависимости от того, *принадлежит* ли переменная классу или объекту соответ- ственно.

*Переменные класса* разделяемы – доступ к ним могут получать все экземпляры этого клас- са. Переменная класса существует только одна, поэтому когда любой из объектов изме- няет переменную класса, это изменение отразится и во всех остальных экземплярах того же класса.

*Переменные объекта* принадлежат каждому отдельному экземпляру класса. В этом случае у каждого объекта есть своя собственная копия поля, т.е. не разделяемая и никоим образом не связанная с другими такими же полями в других экземплярах. Это легко понять на примере (сохраните как objvar.py):

(продолжение на следующей странице)

**class Robot**:

*'''Представляет робота с именем.'''*

*# Переменная класса, содержащая количество роботов*

population = 0

**def** init (self, name): *'''Инициализация данных.'''* self.name = name

print('(Инициализация *{0}*)'.format(self.name))

*# При создании этой личности, робот добавляется # к переменной 'population'*

Robot.population += 1

**def** del (self):

*'''Я умираю.'''*

print('*{0}* уничтожается!'.format(self.name))

Robot.population -= 1

**if** Robot.population == 0:

print('*{0}* был последним.'.format(self.name))

**else**:

print('Осталось *{0:d}* работающих роботов.'.format( **\**

(продолжение с предыдущей страницы)

Robot.population))

**def** sayHi(self):

*'''Приветствие робота.*

*Да, они это могут.'''*

print('Приветствую! Мои хозяева называют меня *{0}*.'.format(self.name))

**def** howMany():

*'''Выводит численность роботов.'''*

print('У нас *{0:d}* роботов.'.format(Robot.population)) howMany = staticmethod(howMany)

droid1 = Robot('R2-D2') droid1.sayHi() Robot.howMany()

droid2 = Robot('C-3PO') droid2.sayHi() Robot.howMany()

print("**\n**Здесь роботы могут проделать какую-то работу.**\n**") print("Роботы закончили свою работу. Давайте уничтожим их.")

**del** droid1

**del** droid2

Robot.howMany()

#### Вывод:

(продолжение на следующей странице)

$ python3 objvar.py (Инициализация R2-D2)

Приветствую! Мои хозяева называют меня R2-D2. У нас 1 роботов.

(Инициализация C-3PO)

Приветствую! Мои хозяева называют меня C-3PO. У нас 2 роботов.

Здесь роботы могут проделать какую-то работу.

Роботы закончили свою работу. Давайте уничтожим их. R2-D2 уничтожается!

Осталось 1 работающих роботов.

(продолжение с предыдущей страницы)

C-3PO уничтожается! C-3PO был последним. У нас 0 роботов.

#### Как это работает:

Это длинный пример, но он помогает продемонстрировать природу перемен- ных класса и объекта. Здесь population принадлежит классу Robot, и поэто- му является переменной класса. Переменная name принадлежит объекту (ей присваивается значение при помощи self), и поэтому является переменной объекта.

Таким образом, мы обращаемся к переменной класса population как Robot. population, а не self.population. К переменной же объекта name во всех методах этого объекта мы обращаемся при помощи обозначения self.name. Помните об этой простой разнице между переменными класса и объекта. Так- же имейте в виду, что переменная объекта с тем же именем, что и переменная класса, сделает недоступной («спрячет») переменную класса!

Метод howMany принадлежит классу, а не объекту. Это означает, что мы мо- жем определить его как classmethod или staticmethod, в зависимости от того, нужно ли нам знать, в каком классе мы находимся. Поскольку нам не нужна такая информация, мы воспользуемся staticmethod.

Мы могли достичь того же самого, используя [декораторы](http://www.ibm.com/developerworks/linux/library/l-cpdecor.html) :

**@staticmethod def** howMany():

*'''Выводит численность роботов.'''*

print('У нас *{0:d}* роботов.'.format(Robot.population))

Декораторы можно считать неким упрощённым способом вызова явного оператора, как мы видели в этом примере.

Пронаблюдайте, как метод init используется для инициализации экземпляра Robot с именем. В этом методе мы увеличиваем счётчик population на 1, так как добав- ляем ещё одного робота. Также заметьте, что значения self.name для каждого объекта свои, что указывает на природу переменных объекта.

Помните, что к переменным и методам самого объекта нужно обращаться, пользуясь

**только** self. Это называется *доступом к атрибутам*.

В этом примере мы также наблюдали применение **строк документации** для классов, рав- но как и для методов. Во время выполнения мы можем обращаться к строке документации класса при помощи «Robot. doc », а к строке документации метода – при помощи

«Robot.sayHi. doc ».

Наряду с методом init , существует и другой специальный метод del , который

вызывается тогда, когда объект собирается умереть, т.е. когда он больше не используется, и занимаемая им память возвращается операционной системе для другого использова- ния. В этом методе мы просто уменьшаем счётчик Robot.population на 1.

Метод del запускается лишь тогда, когда объект перестаёт использоваться, а поэто- му заранее неизвестно, *когда* именно этот момент наступит. Чтобы увидеть его в действии явно, придётся воспользоваться оператором del, что мы и сделали выше.

#### Примечание для программистов на C++/Java/C#

В Python все члены класса (включая данные) являются *публичными* (public), а все методы

– *виртуальными* (virtual).

Исключение: Если имя переменной начинается с *двойного подчёркивания*, как, например,

privatevar, Python делает эту переменную приватной (private). Поэтому принято имя любой переменной, которая должна использоваться только внутри класса или объекта, начинать с подчёркивания; все же остальные имена являются публичными, и могут ис- пользоваться в других классах/объектах. Помните, что это лишь традиция, и Python вовсе не обязывает делать именно так (кроме двойного подчёркивания).

## Наследование

Одно из главных достоинств объектно-ориентированного программирования заключа- ется в **многократном использовании** одного и того же кода, и один из способов этого достичь – при помощи механизма *наследования*. Легче всего представить себе наследова- ние в виде отношения между классами как *тип и подтип*.

Представим, что нам нужно написать программу, которая отслеживает информацию о преподавателях и студентах в колледже. У них есть некоторые общие характеристики: имя, возраст и адрес. Есть также и специфические характеристики, такие как зарплата, курсы и отпуск для преподавателей, а также оценки и оплата за обучение для студентов.

Можно создать для них независимые классы и работать с ними, но тогда добавление какой-либо новой общей характеристики потребует добавления её к каждому из этих независимых классов в отдельности, что делает программу неповоротливой.

Лучше создать общий класс с именем SchoolMember, а затем сделать так, чтобы классы преподавателя и студента *наследовали* этот класс, т.е. чтобы они стали подтипами этого типа (класса), после чего добавить любые специфические характеристики к этим подти- пам.

У такого подхода есть множество достоинств. Если мы добавим/изменим какую-либо функциональность в SchoolMember, это автоматически отобразится и во всех подтипах. Например, мы можем добавить новое поле удостоверения для преподавателей и студен- тов, просто добавив его к классу SchoolMember. С другой стороны, изменения в подтипах

никак не влияют на другие подтипы. Ещё одно достоинство состоит в том, что обращать- ся к объекту преподавателя или студента можно как к объекту SchoolMember, что может быть полезно в ряде случаев, например, для подсчёта количества человек в школе. Ко- гда подтип может быть подставлен в любом месте, где ожидается родительский тип, т.е. объект считается экземпляром родительского класса, это называется **полиморфизмом**.

Заметьте также, что код родительского класса *используется многократно*, и нет необходи- мости копировать его во все классы, как пришлось бы в случае использования независи- мых классов.

Класс SchoolMember в этой ситуации называют *базовым классом* или *надклассом*[3](#_bookmark153). Классы

Teacher и Student называют *производными классами* или *подклассами*[4](#_bookmark154). Рассмотрим теперь этот пример в виде программы (сохраните как inherit.py).

(продолжение на следующей странице)

**class SchoolMember**:

*'''Представляет любого человека в школе.'''*

**def** init (self, name, age): self.name = name

self.age = age

print('(Создан SchoolMember: *{0}*)'.format(self.name))

**def** tell(self):

*'''Вывести информацию.'''*

print('Имя:"*{0}*" Возраст:"*{1}*"'.format(self.name, self.age), end=" ")

**class Teacher**(SchoolMember):

*'''Представляет преподавателя.'''*

**def** init (self, name, age, salary): SchoolMember. init (self, name, age) self.salary = salary

print('(Создан Teacher: *{0}*)'.format(self.name))

**def** tell(self): SchoolMember.tell(self)

print('Зарплата: "*{0:d}*"'.format(self.salary))

**class Student**(SchoolMember):

*'''Представляет студента.'''*

**def** init (self, name, age, marks): SchoolMember. init (self, name, age) self.marks = marks

print('(Создан Student: *{0}*)'.format(self.name))

**def** tell(self): SchoolMember.tell(self)

3 также «суперкласс», «родительский класс» (*прим.перев.*)

4 также «субкласс», «класс-наследник» (*прим.перев.*)

(продолжение с предыдущей страницы)

print('Оценки: "*{0:d}*"'.format(self.marks))

t = Teacher('Mrs. Shrividya', 40, 30000)

s = Student('Swaroop', 25, 75) print() *# печатает пустую строку*

members = [t, s]

**for** member **in** members:

member.tell() *# работает как для преподавателя, так и для студента*

#### Вывод:

$ python3 inherit.py

(Создан SchoolMember: Mrs. Shrividya) (Создан Teacher: Mrs. Shrividya) (Создан SchoolMember: Swaroop) (Создан Student: Swaroop)

Имя:"Mrs. Shrividya" Возраст:"40" Зарплата: "30000" Имя:"Swaroop" Возраст:"25" Оценки: "75"

**Как это работает:**

Чтобы воспользоваться наследованием, при определении класса мы указыва- ем имена его базовых классов в виде кортежа, следующего сразу за его име- нем. Далее мы видим, что метод init базового класса вызывается явно при помощи переменной self, чтобы инициализировать часть объекта, отно- сящуюся к базовому классу. Это очень важно запомнить: поскольку мы опре- деляем метод init в подклассах Teacher и Student, Python не вызывает конструктор базового класса SchoolMember автоматически – его необходимо вызывать самостоятельно в явном виде.

Напротив, если мы *не* определим метод init в подклассе, Python вызовет конструктор базового класса автоматически.

Здесь же мы видим, как можно вызывать методы базового класса, предваряя запись имени метода именем класса, а затем передавая переменную self вме- сте с другими аргументами.

Обратите внимание, что при вызове метода tell из класса SchoolMember экземпляры Teacher или Student можно использовать как экземпляры SchoolMember.

Заметьте также, что вызывается метод tell из подкласса, а не метод tell из класса SchoolMember. Это можно понять следующим образом: Python *всегда* начинает поиск методов в самом классе, что он и делает в данном случае. Если

же он не находит метода, он начинает искать методы, принадлежащие базо- вым классам по очереди, в порядке, в котором они перечислены в кортеже при определении класса.

Замечание по терминологии: если при наследовании перечислено более од- ного класса, это называется *множественным наследованием*.

Параметр end используется в методе tell() для того, чтобы новая строка на- чиналась через пробел после вызова print().

## Метаклассы5

В обширной теме объектно-ориентированного программирования существует ещё много всего, но мы лишь слегка коснёмся некоторых концепций, чтобы вы просто знали об их существовании.

Точно так же, как классы используются для создания объектов, можно использовать ме- таклассы для создания классов. Метаклассы существуют для изменения или добавления нового поведения в классы.

Давайте рассмотрим пример. Допустим, мы хотим быть уверены, что мы всегда создаём исключительно экземпляры подклассов класса SchoolMember, и не создаём экземпляры самого класса SchoolMember.

Для достижения этой цели мы можем использовать концепцию под названием «абстракт- ные базовые классы». Это означает, что такой класс *абстрактен*, т.е. является лишь некой концепцией, не предназначенной для использования в качестве реального класса.

Мы можем объявить наш класс как абстрактный базовый класс при помощи встроенного метакласса по имени ABCMeta.

(продолжение на следующей странице)

*#!/usr/bin/env python*

*# Filename: inherit\_abc.py*

**from abc import** \*

**class SchoolMember**(metaclass=ABCMeta): *'''Представляет любого человека в школе.'''* **def** init (self, name, age):

self.name = name self.age = age

print('(Создан SchoolMember: *{0}*)'.format(self.name))

5 в оригинальной версии книги этот параграф невидим для читателей, так как находится в комментарии с пометкой автора «It is too sudden to introduce this concept here.», что означает «Слишком неожиданно представление этой концепции здесь.» (*прим.перев.*)

(продолжение с предыдущей страницы)

**@abstractmethod def** tell(self):

*'''Вывести информацию.'''*

print('Имя:"*{0}*" Возраст:"*{1}*"'.format(self.name, self.age), end=" ")

**class Teacher**(SchoolMember):

*'''Представляет преподавателя.'''*

**def** init (self, name, age, salary): SchoolMember. init (self, name, age) self.salary = salary

print('(Создан Teacher: *{0}*)'.format(self.name))

**def** tell(self): SchoolMember.tell(self)

print('Зарплата: "*{0:d}*"'.format(self.salary))

**class Student**(SchoolMember):

*'''Представляет студента.'''*

**def** init (self, name, age, marks): SchoolMember. init (self, name, age) self.marks = marks

print('(Создан Student: *{0}*)'.format(self.name))

**def** tell(self): SchoolMember.tell(self)

print('Оценки: "*{0:d}*"'.format(self.marks))

t = Teacher('Mrs. Shrividya', 40, 30000)

s = Student('Swaroop', 25, 75)

*#m = SchoolMember('abc', 10)*

*# Это приведёт к ошибке: "TypeError: Can't instantiate abstract class # SchoolMember with abstract methods tell"*

print() *# печатает пустую строку*

members = [t, s]

**for** member **in** members:

member.tell() *# работает как для преподавателя, так и для студента*

#### Вывод:

(продолжение на следующей странице)

$ python3 inherit.py

(Создан SchoolMember: Mrs. Shrividya) (Создан Teacher: Mrs. Shrividya)

(продолжение с предыдущей страницы)

(Создан SchoolMember: Swaroop) (Создан Student: Swaroop)

Имя:"Mrs. Shrividya" Возраст:"40" Зарплата: "30000" Имя:"Swaroop" Возраст:"25" Оценки: "75"

#### Как это работает:

Мы можем объявить метод tell класса SchoolMember абстрактным, и таким образом автоматически запретим создавать экземпляры класса SchoolMember.

Тем не менее, мы можем работать с экземплярами Teacher и Student так, как будто они экземпляры SchoolMember, поскольку они являются подклассами.

## Резюме

Мы изучили различные аспекты классов и объектов, равно как и терминологию, связанную с ними. Мы также увидели ряд достоинств и «подводных камней» объектно-ориентированного программирования. Python – в высокой степени объектно- ориентирован, поэтому понимание этих принципов очень поможет вам в дальнейшем.

Далее мы узнаем, как работать с вводом/выводом и получать доступ к файлам в Python.

# Ввод-вывод

Рано или поздно возникают ситуации, когда программа должна взаимодействовать с пользователем. Например, принять какие-нибудь данные от пользователя, а затем вы- вести результаты. Для этого применяются функции input() и print() соответственно.

Для вывода можно также использовать различные методы класса str (строка). К примеру, при помощи метода rjust можно получить строку, выравненную по правому краю к указанной ширине. См. help(str) для более подробного описания.

Ещё одним распространённым типом ввода/вывода является работа с файлами. Возмож- ность создавать, читать и записывать в файлы является ключевой для многих программ, поэтому в настоящей главе и мы рассмотрим этот аспект.

## Ввод от пользователя

Сохраните эту программу как user\_input.py:

**def** reverse(text):

**return** text[::-1]

**def** is\_palindrome(text):

**return** text == reverse(text)

something = input('Введите текст: ')

**if** (is\_palindrome(something)): print("Да, это палиндром")

**else**:

print("Нет, это не палиндром")

#### Вывод:

(продолжение на следующей странице)

$ python3 user\_input.py Введите текст: сэр

Нет, это не палиндром

(продолжение с предыдущей страницы)

$ python3 user\_input.py Введите текст: мадам Да, это палиндром

$ python3 user\_input.py Введите текст: топот Да, это палиндром

#### Как это работает:

Мы применяем операцию вырезки для переворачивания текста. Мы уже видели, как создаются [*вырезки из последовательностей*](#_bookmark121)при помощи кода

«seq[a:b]», начиная с позиции a до позиции b. Но ведь мы также можем ука- зать и третий аргумент, определяющий *шаг*, с которым производится вырезка. По умолчанию шаг равен 1, поэтому и возвращается непрерывный фрагмент текста. Указание отрицательного шага, т.е. -1 приведёт к выводу текста в об- ратном порядке.

Функция input() принимает строку в качестве аргумента и показывает её пользователю. Затем она ждёт, чтобы пользователь набрал что-нибудь и на- жал клавишу ввода. Как только пользователь это сделал, функция input() возвращает введённый пользователем текст.

Мы считываем этот текст и выстраиваем его в обратном порядке. Если пе- ревёрнутый и исходный текст одинаковы, значит введённый текст является [палиндромом](http://ru.wikipedia.org/wiki/Палиндром).

#### Домашнее задание

Проверка, является ли текст палиндромом должна также игнорировать знаки пунктуа- ции, пробелы и регистр букв. Например, «А роза упала на лапу Азора» также является палиндромом, но наша текущая программа так не считает. Попробуйте улучшить её так, чтобы она распознала этот палиндром.

#### Подсказка: (не читайте)

Воспользуйтесь кортежем (список *всех* знаков пунктуации можно найти [здесь](http://grammar.ccc.commnet.edu/grammar/marks/marks.htm)), содержа- щим все запрещённые символы, и примените тест на принадлежность, чтобы обнару- жить символы, подлежащие удалению, т.е. forbidden = („!“, „?“, „.“, …).

## Файлы

Открывать и использовать файлы для чтения или записи можно путём создания объекта класса file, а читать/записывать в файл – при помощи его методов read, readline или write соответственно. Возможность читать или записывать в файл зависит от режима, указанного при открытии файла. По окончании работы с файлом, нужно вызвать метод close[1](#_bookmark160), чтобы указать Python, что файл больше не используется.

**Пример:** (сохраните как using\_file.py)

poem = '''**\** Программировать весело. Если работа скучна,

Чтобы придать ей весёлый тон - используй Python!

'''

f = open('poem.txt', 'w') *# открываем для записи (writing)*

f.write(poem) *# записываем текст в файл*

f.close() *# закрываем файл*

f = open('poem.txt') *# если не указан режим, по умолчанию подразумевается # режим чтения ('r'eading)*

**while True**:

line = f.readline()

**if** len(line) == 0: *# Нулевая длина обозначает конец файла (EOF)*

**break**

print(line, end='')

f.close() *# закрываем файл*

#### Вывод:

$ python3 using\_file.py Программировать весело. Если работа скучна,

Чтобы придать ей весёлый тон - используй Python!

**Как это работает:**

Сперва мы открываем файл при помощи встроенной функции open с указа- нием имени файла и режима, в котором мы хотим его открыть. Режим может быть для чтения ('r'), записи ('w') или добавления ('a')[2](#_bookmark161). Можно также ука- зать, в каком виде мы будем считывать, записывать или добавлять данные:

1 close – *англ.* «закрывать» (*прим.перев*)

2 **r**ead, **w**rite и **a**ppend соответственно (*прим.перев.*)

#### 15.2. Файлы 123

в текстовом ('t') или бинарном ('b'). На самом деле существует много дру- гих режимов, и help(open) даст вам их детальное описание. По умолчанию open() открывает файл как текст в режиме для чтения.

В нашем примере мы сначала открываем файл в режиме записи текста и ис- пользуем метод write файлового объекта для записи в файл, после чего за- крываем файл при помощи close.

Далее мы открываем тот же самый файл для чтения. В этом случае нет нужды указывать режим, так как режим «чтения текстового файла» применяется по умолчанию. Мы считываем файл построчно методом readline в цикле. Этот метод возвращает полную строку, включая символ перевода строки в конце. Когда же он возвращает пустую строку, это означает, что мы достигли конца файла, и мы прерываем цикл при помощи break.

По умолчанию функция print() выводит текст, автоматически добавляя сим- вол перевода строки в конце. Мы подавляем этот символ, указывая end='', поскольку строки, считанные из файла, и без того оканчиваются символом перевода строки. И, наконец, мы закрываем файл с помощью close.

Теперь проверяем содержимое файла poem.txt, чтобы убедиться, что про- грамма действительно записала текст в него и считала из него.

## Pickle

Python предоставляет стандартный модуль с именем pickle[3](#_bookmark163), при помощи которого мож- но сохранять **любой** объект Python в файле, а затем извлекать его обратно. Это называется *длительным* хранением объекта.

**Пример:** (сохраните как pickling.py):

(продолжение на следующей странице)

**import pickle**

*# имя файла, в котором мы сохраним объект*

shoplistfile = 'shoplist.data'

*# список покупок*

shoplist = ['яблоки', 'манго', 'морковь']

*# Запись в файл*

f = open(shoplistfile, 'wb') pickle.dump(shoplist, f) *# помещаем объект в файл* f.close()

**del** shoplist *# уничтожаем переменную shoplist*

3 pickle – *англ.* «мариновать», «солить» (*прим.перев.*)

(продолжение с предыдущей страницы)

*# Считываем из хранилища*

f = open(shoplistfile, 'rb')

storedlist = pickle.load(f) *# загружаем объект из файла*

print(storedlist)

#### Вывод:

$ python3 pickling.py ['яблоки', 'манго', 'морковь']

**Как это работает:**

Чтобы сохранить объект в файле, нам нужно сперва открыть файл с помощью open в режиме бинарной записи ('wb'), после чего вызвать функцию dump из модуля pickle. Этот процесс называется «консервацией» («pickling»).

После этого мы извлекаем объект при помощи функции load из модуля pickle, которая возвращает объект. Этот процесс называется «расконсерва- цией» («unpickling»).

## Резюме

Мы обсудили разные типы ввода/вывода, а также работу с файлами и использование мо- дуля pickle.

Далее мы познакомимся с концепцией исключений.

# Исключения

Исключения возникают тогда, когда в программе возникает некоторая *исключительная* ситуация. Например, к чему приведёт попытка чтения несуществующего файла? Или ес- ли файл был случайно удалён, пока программа работала? Такие ситуации обрабатывают- ся при помощи **исключений**.

Это касается и программ, содержащих недействительные команды. В этом случае Python

**поднимает** руки и сообщает, что обнаружил **ошибку**.

## Ошибки

Рассмотрим простой вызов функции print. Что, если мы ошибочно напишем print как Print? Обратите внимание на заглавную букву. В этом случае Python *поднимает* синтак- сическую ошибку.

**>>>** Print('Привет, Мир!') Traceback (most recent call last):

File "<pyshell#0>", line 1, in <module> Print('Привет, Мир!')

NameError: name 'Print' is not defined

**>>>** print('Привет, Мир!') Привет, Мир!

Обратите внимание, что была поднята ошибка NameError, а также указано место, где была обнаружена ошибка. Так в данном случае действует *обработчик ошибок*.

## Исключения

**Попытаемся** считать что-либо от пользователя. Нажмите Сtrl-D (или Ctrl+Z в Windows) и посмотрите, что произойдёт.

(продолжение на следующей странице)

**>>>** s = input('Введите что-нибудь --> ') Введите что-нибудь -->

126

(продолжение с предыдущей страницы)

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module> s = input('Введите что-нибудь --> ')

EOFError: EOF when reading a line

Python поднимает ошибку с именем EOFError, что означает, что он обнаружил символ

*конца файла* (который вводится при помощи Ctrl-D) там, где не ожидал.

## Обработка исключений

Обрабатывать исключения можно при помощи оператора try..except[1](#_bookmark169). При этом все обычные команды помещаются внутрь try-блока, а все обработчики исключений – в except-блок.

**Пример:** (сохраните как try\_except.py)

**try**:

text = input('Введите что-нибудь --> ')

**except** EOFError:

print('Ну зачем вы сделали мне EOF?')

**except** KeyboardInterrupt: print('Вы отменили операцию.')

**else**:

print('Вы ввели *{0}*'.format(text))

#### Вывод:

$ python3 try\_except.py

Введите что-нибудь -->

# Нажмите ctrl-d

Ну зачем вы сделали мне EOF?

$ python3 try\_except.py Введите что-нибудь -->

Вы отменили операцию.

# Нажмите ctrl-c

$ python3 try\_except.py

Введите что-нибудь --> без ошибок Вы ввели без ошибок

**Как это работает:**

Здесь мы поместили все команды, которые могут вызвать исключе- ния/ошибки, внутрь блока try, а затем поместили обработчики соответству-

1 try – *англ.* «пытаться» (*прим.перев.*)

#### Обработка исключений 127

ющих ошибок/исключений в блок except. Выражение except может об- рабатывать как одиночную ошибку или исключение, так и список оши- бок/исключений в скобках. Если не указано имя ошибки или исключения, об- рабатываться будут *все* ошибки и исключения.

Помните, что для каждого выражения try должно быть хотя бы одно соответствующее выражение except. Иначе какой смысл был бы в блоке try?

Если ошибка или исключение не обработано, будет вызван обработчик Python по умол- чанию, который останавливает выполнение программы и выводит на экран сообщение об ошибке. Выше мы уже видели это в действии.

Можно также добавить пункт else к соответствующему блоку try..except. Этот пункт будет выполнен тогда, когда исключений не возникает.

В следующем примере мы увидим, как можно получить объект исключения для дальней- шей работы с ним.

## Вызов исключения

Исключение можно *поднять* при помощи оператора raise[2](#_bookmark171), передав ему имя ошиб- ки/исключения, а также объект исключения, который нужно *выбросить*.

Вызываемая ошибка или исключение должна быть классом, который прямо или непрямо является производным от класса Exception.

**Пример:** (сохраните как raising.py)

(продолжение на следующей странице)

**class ShortInputException**(Exception): *'''Пользовательский класс исключения.'''* **def** init (self, length, atleast):

Exception. init (self) self.length = length self.atleast = atleast

**try**:

text = input('Введите что-нибудь --> ')

**if** len(text) < 3:

**raise** ShortInputException(len(text), 3)

*# Здесь может происходить обычная работа*

**except** EOFError:

print('Ну зачем вы сделали мне EOF?')

**except** ShortInputException **as** ex:

print('ShortInputException: Длина введённой строки -- *{0}*; **\**

ожидалось, как минимум, *{1}*'.format(ex.length, ex.atleast))

2 raise – *англ.* «поднимать» (*прим.перев.*)

#### Вызов исключения 128

(продолжение с предыдущей страницы)

**else**:

print('Не было исключений.')

#### Вывод:

$ python3 raising.py Введите что-нибудь --> а

ShortInputException: Длина введённой строки -- 1; ожидалось, как минимум, 3

$ python3 raising.py Введите что-нибудь --> абв Не было исключений.

**Как это работает:**

Здесь мы создаём наш собственный тип исключения. Этот новый тип исклю- чения называется ShortInputException. Он содержит два поля: length, хра- нящее длину введённого текста, и atleast, указывающее, какую минималь- ную длину текста ожидала программа.

В пункте except мы указываем класс ошибки ShortInputException, кото- рый будет сохранён как[3](#_bookmark173) переменная ex, содержащая соответствующий объ- ект ошибки/исключения. Это аналогично параметрам и аргументам при вы- зове функции. Внутри этого пункта except мы используем поля length и atleast объекта исключения для вывода необходимых сообщений пользова- телю.

## Try .. Finally

Представим, что в программе происходит чтение файла и необходимо убедиться, что объ- ект файла был корректно закрыт и что не возникло никакого исключения. Этого можно достичь с применением блока finally.

Сохраните как finally.py:

(продолжение на следующей странице)

**import time**

**try**:

f = open('poem.txt')

**while True**: *# наш обычный способ читать файлы*

line = f.readline()

**if** len(line) == 0:

3 as – *англ.* «как» (*прим.перев.*)

#### Try .. Finally 129

(продолжение с предыдущей страницы)

**break**

print(line, end='')

time.sleep(2) *# Пусть подождёт некоторое время*

**except** KeyboardInterrupt:

print('!! Вы отменили чтение файла.')

**finally**:

f.close()

print('(Очистка: Закрытие файла)')

#### Вывод:

$ python3 finally.py Программировать весело Если работа скучна,

Чтобы придать ей весёлый тон -

!! Вы отменили чтение файла. (Очистка: Закрытие файла)

**Как это работает:**

Здесь мы производим обычные операции чтения из файла, но в данном слу- чае добавляем двухсекундный сон после вывода каждой строки при помощи функции time.sleep, чтобы программа выполнялась медленно (ведь Python очень быстр от природы). Во время выполнения программы нажмите ctrl-c, чтобы прервать/отменить выполнение программы.

Пронаблюдайте, как при этом выдаётся исключение KeyboardInterrupt, и программа выходит. Однако, прежде чем программа выйдет, выполняется пункт finally, и файловый объект будет всегда закрыт.

## Оператор with

Типичной схемой является запрос некоторого ресурса в блоке try с последующим осво- бождением этого ресурса в блоке finally. Для того, чтобы сделать это более «чисто», существует оператор with[4](#_bookmark175):

Сохраните как using\_with.py:

**with** open("poem.txt") **as** f:

**for** line **in** f: print(line, end='')

4 with – *англ.* «с» (*прим.перев.*)

#### Оператор with 130

**Как это работает:**

Вывод должен быть таким же, как и в предыдущем примере. Разница лишь в том, что здесь мы используем функцию open с оператором with – этим мы оставляем автоматическое закрытие файла под ответственность with open.

За кулисами происходит следующее. Существует некий протокол, используе- мый оператором with. Он считывает объект, возвращаемый оператором open. Назовём его в данном случае «thefile».

Перед запуском блока кода, содержащегося в нём, оператор with *всегда* вы- зывает функцию thefile. enter , а также *всегда* вызывает thefile.

exit после завершения выполнения этого блока кода.

Так что код, который мы бы написали в блоке finally, будет автоматически обработан методом exit . Это избавляет нас от необходимости повторно в явном виде указывать операторы try..finally.

Более обширное рассмотрение этой темы выходит за рамки настоящей книги, поэтому для более исчерпывающего объяснения см. [**PEP 343**](https://www.python.org/dev/peps/pep-0343).

## Резюме

Мы обсудили использование операторов try..except и try..finally. Мы также уви- дели, как создавать наши собственные типы исключений и как их вызывать.

Далее мы ознакомимся со стандартной библиотекой Python.

# Стандартная библиотека

Стандартная библиотека Python содержит огромное количество полезных модулей и яв- ляется частью стандартного комплекта поставки Python. Ознакомиться со стандартной библиотекой Python очень важно, так как множество задач можно решить очень быстро, если вы знакомы с возможностями этих библиотек.

Рассмотрим некоторые наиболее часто используемые модули этой библиотеки. Деталь- ное описание всех модулей стандартной библиотеки Python можно найти в [разделе](http://docs.python.org/py3k/library/index.html)

[«Library Reference»](http://docs.python.org/py3k/library/index.html) документации, входящей в комплект поставки Python. Давайте изучим несколько полезных модулей.

#### Примечание

Если темы в настоящей главе покажутся вам слишком сложными, вы можете её пропу- стить. Однако я настоятельно рекомендую вернуться к этой главе, когда вы будете чув- ствовать себя более уверенно с Python.

## Модуль sys

Модуль sys содержит функциональность, характерную для системы. Так мы видели, что список sys.argv содержит аргументы командной строки.

Предположим, нам нужно узнать версию используемой команды Python с тем, чтобы, к примеру, убедиться в том, что мы используем как минимум версию 3. Модуль sys предо- ставляет такую возможность.

**>>> import sys**

**>>>** sys.version\_info (3, 0, 0, 'beta', 2)

**>>>** sys.version\_info[0] >= 3 True

#### Как это работает:

Модуль sys содержит кортеж version\_info, который хранит информацию о версии. Первый элемент этого кортежа обозначает старшую версию. Мы мо- жем использовать его, например, для того, чтобы убедиться, что программа будет выполняться только в Python 3.0:

Сохраните как versioncheck.py:

**import sys**, **warnings**

**if** sys.version\_info[0] < 3:

warnings.warn("Для выполнения этой программы необходима как минимум **\**

версия Python 3.0", RuntimeWarning)

**else**:

print('Нормальное продолжение')

#### Вывод:

$ python2.7 versioncheck.py

versioncheck.py:6: Для выполнения этой программы необходима как минимум

*‹→*версия Python 3.0 RuntimeWarning)

$ python3 versioncheck.py Нормальное продолжение

**Как это работает:**

Мы используем один из модулей стандартной библиотеки, который называ- ется warnings и служит для отображения предупреждений пользователю. Ес- ли версия Python менее 3, мы показываем соответствующее предупреждение.

## Модуль logging

Представьте ситуацию, когда необходимо сохранить некоторые отладочные или другие важные сообщения где-нибудь, чтобы иметь возможность позже проверить, отработала ли программа, как ожидалось. Как мы «сохраним где-нибудь» эти сообщения? Сделать это можно при помощи модуля logging.

Сохраните как use\_logging.py:

(продолжение на следующей странице)

**import os**, **platform**, **logging**

**if** platform.platform().startswith('Windows'): logging\_file = os.path.join(os.getenv('HOMEDRIVE'), \

(продолжение с предыдущей страницы)

os.getenv('HOMEPATH'), \ 'test.log')

**else**:

logging\_file = os.path.join(os.getenv('HOME'), 'test.log') print("Сохраняем лог в", logging\_file)

logging.basicConfig( level=logging.DEBUG,

format='*%(asctime)s* : *%(levelname)s* : *%(message)s*', filename = logging\_file,

filemode = 'w',

)

logging.debug("Начало программы") logging.info("Какие-то действия") logging.warning("Программа умирает")

#### Вывод:

$ python3 use\_logging.py

Сохраняем лог в C:\Users\swaroop\test.log

Если открыть файл test.log, он будет выглядеть примерно так:

|  |  |  |
| --- | --- | --- |
| 2012-10-26 | 16:52:41,457 : | DEBUG : Начало программы |
| 2012-10-26 | 16:52:41,474 : | INFO : Какие-то действия |
| 2012-10-26 | 16:52:41,475 : | WARNING : Программа умирает |

#### Как это работает:

Мы использовали три модуля из стандартной библиотеки: модуль os для вза- имодействия с операционной системой, модуль platform для получения ин- формации о платформе (т.е. операционной системе) и модуль logging для сохранения лога[1](#_bookmark180).

Прежде всего, при помощи строки, возвращаемой функцией platform. platform() мы проверяем, какая операционная система используется (для более подробной информации см. import platform; help(platform)). Ес- ли это Windows, то мы определяем диск, содержащий домашний каталог, путь к домашнему каталогу на нём и имя файла, в котором хотим сохранить ин- формацию. Сложив все эти три части, мы получаем полный путь к файлу. Для других платформ нам нужно знать только путь к домашнему каталогу пользователя, и мы получим полный путь к файлу.

1 log – *англ.* «журнал», «вести журнал» (*прим.перев.*)

При помощи функции os.path.join() мы объединяем три части пути к файлу вместе. Мы используем эту функцию вместо простого объединения строк для того, чтобы гарантировать, что полный путь к файлу записан в фор- мате, ожидаемом операционной системой.

Далее мы конфигурируем модуль logging таким образом, чтобы он записы- вал все сообщения в определённом формате в указанный файл.

Наконец, мы можем выводить сообщения, предназначенные для отладки, ин- формирования, предупреждения и даже критические сообщения. После вы- полнения программы можно просмотреть этот файл и узнать, что происхо- дило в программе, хотя пользователю, запустившему программу, ничего не было показано.

## Серия «Модуль недели»

В стандартной библиотеке можно найти ещё много полезного. Например, [отладка](http://docs.python.org/py3k/library/pdb.html), [обра-](http://docs.python.org/py3k/library/argparse.html) [ботка параметров командной строки](http://docs.python.org/py3k/library/argparse.html), [регулярные выражения](http://docs.python.org/py3k/library/re.html) и так далее.

Лучший способ дальнейшего изучения стандартной библиотеки – читать замечательную серию Дуга Хелмана [«Модуль недели»](http://www.doughellmann.com/projects/PyMOTW/) или официальную [документацию Python](http://docs.python.org/py3k/).

## Резюме

Мы изучили лишь некоторые возможности некоторых модулей стандартной библиотеки Python. Я настоятельно рекомендую просмотреть [документацию по стандартной библио-](http://docs.python.org/py3k/library/index.html) [теке Python](http://docs.python.org/py3k/library/index.html), чтобы увидеть все доступные модули.

Далее мы обратимся к некоторым аспектам, которые сделают вашу экскурсию по Python более «завершённой».

# Дополнительно

К настоящему моменту мы уже рассмотрели большую часть того, что вам придётся ис- пользовать при работе с Python. В этой главе мы охватим некоторые дополнительные аспекты, которые помогут отшлифовать ваши знания.

## Передача кортежей

Хотелось ли вам когда-нибудь, чтобы функция возвращала не один результат, а два? Это возможно. Всё, что для этого нужно, – использовать кортеж.

**>>> def** get\_error\_details():

**... return** (2, 'описание ошибки No2')

**...**

**>>>** errnum, errstr = get\_error\_details()

**>>>** errnum 2

**>>>** errstr

'описание ошибки No2'

Обратите внимание, что использование выражения «a, b = <некоторое выражение>» интерпретирует результат как кортеж из двух значений.

Чтобы интерпретировать результат как «(a, <всё остальное>)», нужно просто поста- вить звёздочку, как это делалось для параметров функций:

**>>>** a, \*b = [1, 2, 3, 4]

**>>>** a 1

**>>>** b [2, 3, 4]

Это также подразумевает, что поменять местами два значения в Python быстрее всего можно так:

**>>>** a = 5; b = 8

**>>>** a, b = b, a

**>>>** a, b (8, 5)

## Специальные методы

Есть ряд методов, играющих особую роль для классов. Например, init и del .

Специальные методы служат для того, чтобы имитировать поведение встроенных типов данных. Например, всё, что потребуется для использования операции индексирования x[индекс] применительно к своему классу (в таком виде, как это делалось для списков и кортежей), это реализовать метод getitem (). Кстати, именно этот метод Python использует для самого класса list!

Некоторые полезные специальные методы перечислены в таблице ниже. Все другие ме- тоды можно посмотреть в [документации](http://docs.python.org/py3k/reference/datamodel.html#specialnames).

|  |  |
| --- | --- |
| Имя | Описание |
| init (self, …) | Этот метод вызывается прямо перед тем, как вновь созданный  объект возвращается для использования. |
| del (self) | Вызывается перед уничтожением объекта |
| str (self) | Вызывается при использовании функции print или str(). |
| lt (self, other) | Вызывается, когда используется оператор «меньше» (<).  Существуют и аналогичные методы для всех операторов (+, >, и т.д.) |
| getitem (self, key) | Вызывается при использовании оператора индексирования  x[индекс] |
| len (self) | Вызывается при обращении к встроенной функции len() для  объекта-последовательности. |

## Блоки в одно выражение

Мы неоднократно говорили, что каждый блок команд отделяется от других своим соб- ственным уровнем отступа. Однако, существует и исключение. Если блок команд содер- жит только одно выражение, его можно указывать в одной строке с условным оператором или, скажем, оператором цикла. Рассмотрим это на примере:

**>>>** flag = **True**

**>>> if** flag: print('Да') Да

Обратите внимание, что единственный оператор расположен в той же строке, а не отдель- ным блоком. Этот способ может подкупить тем, что якобы «сокращает» программу, но я настоятельно рекомендую избегать его во всех случаях, кроме проверки ошибок. Прежде всего, потому что гораздо легче добавлять команды, когда уже есть необходимый уровень отступа.

## Lambda-формы

Ключевое слово lambda используется для создания функций и возврата их значения во время выполнения программы. lambda принимает параметр, за которым следует одно выражение, которое становится телом функции, а значение этого выражения возвраща- ется новой функцией.

**Пример:** (сохраните как lambda.py)

points = [ { 'x' : 2, 'y' : 3 }, { 'x' : 4, 'y' : 1 } ]

points.sort(key=**lambda** i : i['y']) print(points)

#### Вывод:

$ python3 lambda.py

[{'x': 4, 'y': 1}, {'x': 2, 'y': 3}]

**Как это работает:**

Обратите внимание на то, что метод sort класса list может принимать пара- метр key, определяющий способ сортировки списка (обычно мы думаем толь- ко о сортировке по возрастанию или по убыванию). В данном случае мы хотим провести сортировку по собственному принципу, для чего нам необходимо написать соответствующую функцию. Но вместо того, чтобы создавать отдель- ный блок def для описания функции, которая будет использоваться только в этом месте, мы применяем лямбда-выражение.

## Генераторы списков

Генераторы списков служат для создания новых списков на основе существующих. Пред- ставьте, что имеется список чисел, на основе которого требуется получить новый список, состоящий из всех чисел, умноженных на 2, но только при условии, что само число боль- ше 2. Генераторы списков подходят для таких задач как нельзя лучше.

**Пример:** (сохраните как list\_comprehension.py)

listone = [2, 3, 4]

listtwo = [2\*i **for** i **in** listone **if** i > 2] print(listtwo)

#### Вывод:

$ python3 list\_comprehension.py [6, 8]

**Как это работает:**

В этом примере мы создаём новый список, указав операцию, которую необхо- димо произвести (2 \* i), когда выполняется некоторое условие (if i > 2). Обратите внимание, что исходный список при этом не изменяется.

Преимущество использования генераторов списков состоит в том, что это заметно сокра- щает объёмы стандартного кода, необходимого для циклической обработки каждого эле- мента списка и сохранения его в новом списке.

## Передача кортежей и словарей в функции

Для получения параметров, переданных функции, в виде кортежа или словаря, существу- ют специальные приставки «\*» или «\*\*» соответственно. Это особенно полезно в случа- ях, когда функция может принимать переменное число параметров.

**>>> def** powersum(power, \*args):

**...**

**...**

**...**

**...**

**...**

**...**

*'''Возвращает сумму аргументов, возведённых в указанную степень.'''*

total = 0

**for** i **in** args:

total += pow(i, power)

**return** total

**>>>** powersum(2, 3, 4)

25

**>>>** powersum(2, 10)

100

Поскольку перед переменной args указана приставка «\*», все дополнительные аргу- менты, переданные функции, сохранятся в args в виде кортежа. В случае использова- ния приставки «\*\*» все дополнительные параметры будут рассматриваться как пары ключ/значение в словаре.

## exec и eval

Функция exec служит для выполнения команд Python, содержащихся в строке или фай- ле, в отличие от самого текста программы. Например, во время выполнения программы можно сформировать строку, содержащую текст программы на Python, и запустить его при помощи exec:

**>>>** exec('print("Здравствуй, Мир!")') Здравствуй, Мир!

Аналогично, функция eval позволяет вычислять корректные выражения Python, содер- жащиеся в строке. Вот простой пример.

**>>>** eval('2\*3') 6

## Оператор assert

Оператор assert существует для того, чтобы указать, что нечто является истиной. На- пример, если требуется гарантировать, что в списке будет хотя бы один элемент, и вы- звать ошибку, если это не так, то оператор assert идеально подойдёт для такой задачи. Когда заявленное выражение ложно, вызывается ошибка AssertionError. Метод pop() возвращает последний элемент списка, одновременно удаляя его оттуда.

**>>>** mylist = ['item']

**>>> assert** len(mylist) >= 1

**>>>** mylist.pop() 'item'

**>>>** mylist []

**>>> assert** len(mylist) >= 1 Traceback (most recent call last):

File "<stdin>", line 1, in <module> AssertionError

Тем не менее, оператор assert следует использовать благоразумно. В большинстве слу- чаев гораздо лучше «отлавливать» исключения и либо решать соответствующую про- блему автоматически, либо выдавать пользователю сообщение об ошибке и завершать работу программы.

## Функция repr

Функция repr используется для получения канонического строкового представления объекта. Любопытно, что в большинстве случаев eval(repr(object)) == object.

**>>>** i = []

**>>>** i.append('item')

**>>>** repr(i) "['item']"

**>>>** eval(repr(i)) ['item']

**>>>** eval(repr(i)) == i True

По большому счёту, функция repr служит для получения печатаемого представления объекта. Определив метод repr в собственном классе, можно указать, что он будет возвращать по вызову функции repr.

## Управляющие последовательности

Попробуйте ответить на вопрос: Как указать строку, содержащую одинарную кавычку (')? Например, строку «What's your name?». Её ведь нельзя записать просто как «'What's your name?'», потому что тогда Python не сможет определить, где начало строки, и где конец. В таком случае придётся каким-то образом указать, что данная одинарная кавычка не обозначает конца строки. Это можно сделать при помощи так называемой *управляю- щей последовательности*. Укажите одинарную кавычку как \' – через обратную косую черту. Теперь наша строка будет выглядеть так: 'What\'s your name?'.

Другой способ записи такой специфической строки – "What's your name?", т.е. с ис- пользованием двойных кавычек. Аналогично следует использовать управляющую после- довательность для вставки двойной кавычки в строку, ограниченную двойными кавычка- ми. Сама же обратная наклонная черта указывается управляющей последовательностью

\\.

А как записать двустрочную строку? Один из вариантов нам уже знаком – заключить строку в тройные кавычки, как было показано [*ранее*](#_bookmark50). Но есть и другой – использовать управляющую последовательность для символа перевода строки \n. Например: «Это первая строка\nЭто вторая строка». Полезно знать ещё одну управляющую после- довательность – табуляцию (\t). Управляющих последовательностей существует намного больше, но здесь упомянуты только наиболее важные.

Следует отметить, что одинарная наклонная черта в конце строки лишь указывает на то, что продолжение идёт строкой ниже, но не вставляет перевода строки. Например:

"Это первое предложение. **\**

Это второе предложение."

эквивалентно записи "Это первое предложение. Это второе предложение.".

## Необрабатываемые строки

Для записи строки, в которой не будет проводиться никакой специальной обработки, как, например, управляющих последовательностей, перед строкой указывается приставка «r» или «R»[1](#_bookmark196). Например, r"Перевод строки обозначается \n".

#### Замечание для пользователей регулярных выражений

Для работы с регулярными выражениями всегда используйте необрабатываемые строки. В противном случае вас ждёт много возни с обратными косыми чёрточками. Например, обратные ссылки можно обозначать как '\\1' или r'\1'.

## Резюме

Итак, в настоящей главе мы рассмотрели некоторые дополнительные возможности Python, хотя по-прежнему, не охватили всего. Тем не менее, к настоящему моменту мы уже прошли почти всё, что вам когда-либо понадобится использовать на практике. Этого вполне достаточно для начала работы над любыми программами.

Далее мы обсудим, как продолжать исследовать Python.

1 «r» – *от англ.* «raw» – «сырой, необработанный» (*прим. перев.*)

**18.11. Необрабатываемые строки** 142

# Что дальше

Добросовестный читатель, дочитавший книгу до сих пор, а также много практиковав- шийся в написании программ, наверняка уже освоился с Python. И конечно, вы уже по- пробовали реализовать какие-нибудь собственные идеи на Python, чтобы потренировать- ся. Нет? Тогда стоит начать! Но в таком случае возникает вопрос: «Что же делать дальше?».

Я предлагаю вам справиться со следующей задачей.

#### Задача

Создайте собственную программу «Адресная книга», работающую из командной стро- ки и позволяющую просматривать, добавлять, изменять, удалять или искать контактные данные ваших знакомых. Кроме того, эта информация также должна сохраняться на дис- ке для последующего доступа.

Это достаточно простая задача, если думать о ней в терминах, которые мы до сих пор проходили. Если же вы всё-таки нуждаетесь в подсказке, как действовать, вот она.

#### Подсказка (не читать!)

Создайте класс для хранения персональных данных. Объекты визитных карточек храни- те в словаре, в котором имена контактов будут служить ключами. Для длительного хра- нения этих объектов на жёстком диске воспользуйтесь модулем pickle. Для добавления, изменения или удаления контактов применяйте встроенные методы словаря.

Как только вы справитесь с этим, вы смело можете называться программистом на Python. А теперь немедленно [напишите мне письмо](http://www.swaroopch.com/contact/) ;-). Это не обязательный шаг, но весьма ре- комендуемый. Также подумайте о [покупке бумажной версии книги](http://www.swaroopch.com/buybook/), чтобы поддержать её дальнейшую разработку.

Если эта задачка показалась вам слишком лёгкой, попробуйте ещё одну:

#### Задача

Реализуйте команду [replace](http://unixhelp.ed.ac.uk/CGI/man-cgi?replace). Эта команда заменяет одну строку другой в списке передан- ных ей файлов.

Команда replace может быть любой желаемой сложности: от простой замены строк до поиска по шаблону (регулярному выражению).

А вот ещё некоторые возможные направления вашего дальнейшего путешествия по миру Python:

## Упражнения

На Stack Overflow есть неплохое обсуждение [упражнений на Python, помогающих отто-](http://stackoverflow.com/questions/29578/python-exercises-to-hone-your-skills) [чить ваше мастерство](http://stackoverflow.com/questions/29578/python-exercises-to-hone-your-skills).

## Примеры программ

Лучший способ овладеть языком программирования – это писать много программ и чи- тать много программ:

* + - [Проект PLEAC](http://pleac.sourceforge.net/pleac_python/index.html)
    - Хранилище кода Rosetta
    - [Примеры для Python на java2s](http://www.java2s.com/Code/Python/CatalogPython.htm)
    - [Книга рецептов Python](http://code.activestate.com/recipes/langs/python/) – ценнейшая коллекция рецептов и подсказок, как решать те или иные проблемы при помощи Python. Обязательна к прочтению каждым про- граммистом на Python.
    - [Модуль недели](http://www.doughellmann.com/PyMOTW/contents.html) – ещё один замечательный путеводитель по [*стандартной библио-*](#_bookmark177)[*теке Python*](#_bookmark177).

## Вопросы и ответы

* + - [Официальные «Можно и нельзя» в Python](http://docs.python.org/dev/howto/doanddont.html)
    - [Официальные ЧаВо Python](http://www.python.org/doc/faq/general/)
    - [Список НеЧасто задаваемых Вопросов от Norvig](http://norvig.com/python-iaq.html)
    - [Вопросы и ответы из интервью о Python](http://dev.fyicenter.com/Interview-Questions/Python/index.html)
    - [Вопросы на StackOverflow с пометкой «python»](http://beta.stackoverflow.com/questions/tagged/python)

## Советы и рекомендации

* + - [Трюки и уловки Python](http://www.siafoo.net/article/52)
    - [Сколачиваем программы при помощи Python](http://ivory.idyll.org/articles/advanced-swc/)
    - [Очаровательный Python](http://gnosis.cx/publish/tech_index_cp.html) – замечательная серия статей о Python Дэвида Мерца.

## Учебники

* + - [Всеобъемлющий список учебников по Python от Awaretek](http://www.awaretek.com/tutorials.html)

## Видео

* + - [PyVideo](http://www.pyvideo.org/category)

## Обсуждение

Если вы застряли на какой-то проблеме в Python и не знаете, кого спросить, тогда [список](http://mail.python.org/mailman/listinfo/tutor) [рассылки python-tutor](http://mail.python.org/mailman/listinfo/tutor) подойдёт наилучшим образом.

Тем не менее, сначала проделайте всю домашнюю работу и попытайтесь решить пробле- му самостоятельно.

## Новости

Если вас интересуют последние новости мира Python, отслеживайте их на официальной [планете Python](http://planet.python.org/).

## Установка библиотек

В [Каталоге пакетов Python](http://pypi.python.org/pypi) существует колоссальное количество открытых библиотек, ко- торые вы можете использовать в своих программах. Для их установки можно воспользо- ваться [pip](http://www.pip-installer.org/en/latest/).

## Графические программы

Для создания собственной графической программы на Python понадобится какая-нибудь библиотека ГИП (графического интерфейса пользователя) со своими привязками к Python. Привязки позволяют писать программу на Python, используя библиотеки, кото- рые сами по себе написаны на C, C++ или других языках.

Выбор ГИП для Python достаточно обширен:

#### Kivy

[http://kivy.org](http://kivy.org/)

#### PyGTK

Это привязки Python к инструментарию GTK+, на основе которого построен GNOME. У GTK+ есть много своих особенностей, но как только вы освоитесь, вы сможете создавать ГИП очень быстро. Что касается дизайнера графического интер- фейса Glade, то он просто незаменим. Документация же всё ещё требует некоторых улучшений. GTK+ хорошо работает в GNU/Linux, но его порт на Windows пока не закончен. При помощи GTK+ можно создавать как свободные, так и проприетарные программы. Для начала прочитайте [Учебник по PyGTK](http://www.pygtk.org/tutorial.html).

#### PyQt

Это привязки Python к инструментарию Qt, на основе которого построена KDE. Qt чрезвычайно прост в использовании, особенно благодаря Qt Designer и изумитель- ной документации Qt. PyQt бесплатно, если используется для создания свободных программ (с лицензией GPL). Для создания закрытых проприетарных программ вам придётся его купить. Начиная с Qt 4.5, разрешается создавать при помощи него не только GPL’ные программы. Для начала прочитайте [краткое пособие по PyQt](http://zetcode.com/tutorials/pyqt4/) или [Книгу PyQt](http://www.qtrac.eu/pyqtbook.html).

#### wxPython

Это привязки Python к инструментарию wxWidgets. wxPython не так прост в освое- нии, но зато он переносим и работает на GNU/Linux, Windows, Mac и даже на встраи- ваемых платформах. Многие среды разработки для wxPython, такие как [SPE (Stani’s](http://spe.pycs.net/) [Python Editor)](http://spe.pycs.net/) и [wxGlade](http://wxglade.sourceforge.net/) включают дизайнеры графического интерфейса. При по- мощи wxPython можно создавать как свободные, так и проприетарные программы. Для начала прочитайте [учебник по wxPython](http://zetcode.com/wxpython/).

**19.9. Установка библиотек** 146

## Резюме по инструментам ГИП

Другие варианты можно найти на [wiki-странице GuiProgramming официального сайта](http://www.python.org/cgi-bin/moinmoin/GuiProgramming) [Python](http://www.python.org/cgi-bin/moinmoin/GuiProgramming).

К сожалению, не существует некоего единого стандартного инструмента для создания графических программ на Python. Я бы рекомендовал выбирать один из инструментов, описанных выше, наиболее подходящий для конкретной ситуации. Во-первых, опреде- литесь, согласны ли вы платить за использование этого инструмента. Во-вторых, опре- делитесь, на каких платформах должна работать ваша программа: только на Windows, на Mac и GNU/Linux или на всех сразу. И в-третьих, если вы выбрали платформу GNU/Linux, то в какой среде вы работаете: в KDE или GNOME.

Для более подробного анализа см. страницу 26 [Статей о Python, Том 3, Выпуск 1](http://archive.pythonpapers.org/ThePythonPapersVolume3Issue1.pdf).

## Различные реализации

Язык программирования обычно состоит из двух частей: собственно языка и программ- ного обеспечения. Язык – это то, *как* мы что-либо пишем, а программное обеспечение – это то, *что* запускает наши программы.

До сих пор для выполнения наших программ мы использовали *CPython*. Он называется

«CPython», потому что написан на языке C и является *Классическим интерпретатором Python*[1](#_bookmark210).

Но существует и другое программное обеспечение, способное выполнять программы на Python:

* + - [Jython](http://www.jython.org/)

Реализация Python, работающая на платформе Java. Это означает, что можно ис- пользовать библиотеки и классы Java в программе на Python и наоборот.

* + - [IronPython](http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython)

Реализация Python, работающая на платформе .NET, что означает возможность ис- пользования библиотек и классов .NET в программах на Python и наоборот.

* + - [PyPy](http://codespeak.net/pypy/dist/pypy/doc/home.html)

Реализация Python, написанная на Python! Это исследовательский проект, суще- ствующий для ускорения и облегчения разработки интерпретатора, поскольку сам интерпретатор написан на динамическом языке (в отличие от статических языков, как C, Java и C# в вышеупомянутых реализациях)

* + - [Stackless Python](http://www.stackless.com/)

1 **C**lassical **Python** interpreter (*прим.перев.*)

#### 19.11. Резюме по инструментам ГИП 147

Реализация Python, специализирующаяся на высокой производительности много- поточных программ.

Существуют и другие реализации, такие как [CLPython](http://common-lisp.net/project/clpython/) – написанная на Common Lisp и IronMonkey – порт IronPython на JavaScript, что даёт возможность писать браузерные при- ложения («Ajax») на Python вместо JavaScript.

Каждая из перечисленных реализаций имеет свою область применения.

## Функциональное программирование (для хо- рошо подготовленных читателей)

Для написания больших программ определённо необходимо изучить функциональный подход к программированию, в отличие от объектно-ориентированного подхода, кото- рый мы проходили в [*разделе «Объектно-ориентированное программирование»*](#_bookmark144):

* + - [Практические советы по функциональному программированию, A.M. Kuchling](http://docs.python.org/dev/howto/functional.html)
    - [Глава «Функциональное программирование» книги «Dive Into Python»](http://www.diveintopython.net/functional_programming/index.html)
    - [Презентация «Функциональное программирование в Python»](http://ua.pycon.org/static/talks/kachayev/index.html)

## Резюме

Вот мы и подошли к концу нашей книги, но как говорится, это только *начало конца*! Те- перь вы заядлый программист на Python, и несомненно, готовы решать множество задач при помощи Python. Теперь вы можете начать автоматизировать на своём компьютере всё подряд, можете писать собственные игры и многое-многое другое. Так что вперёд!

#### 19.13. Функциональное программирование (для хорошо подготовленных читателей)

148

# Приложение: FLOSS

## Free/Libre and Open Source Software (FLOSS)1

В основу идеи [FLOSS](http://en.wikipedia.org/wiki/FLOSS) положена концепция сообщества, в котором принято делиться, и особенно делиться знаниями. Свободные программы можно свободно использовать, из- менять и распространять.

Если вы уже прочли эту книгу, то вы уже знакомы со свободным ПО, так как вы изучали

**Python** всё это время, а Python является свободным программным обеспечением!

Вот несколько примеров свободного ПО, по которым можно составить некоторое пред- ставление о том, что способно создать такое сообщество:

* + - **Linux**. Это свободное ядро операционной системы, используемое, например, в опе- рационной системе GNU/Linux. Разработку ядра «Linux» начал Линус Торвальдс ещё в свою бытность студентом. [[Linux Kernel](http://www.kernel.org/)]
    - **GNU**. Свободная операционная система, основанная Ричардом Столлманом в 1983 году. [[GNU](http://www.gnu.orgGNU/)]
    - **GNU/Linux**. Сочетание операционной системы GNU с ядром «Linux», в наше вре- мя серьёзный конкурент Microsoft Windows. В принципе, GNU/Linux является сво- бодным программным обеспечением, но разные дистрибутивы GNU/Linux могут включать и несвободные программы. [[Free GNU/Linux distributions](http://www.gnu.org/distros)]
    - **Ubuntu**. Это дистрибутив, разрабатываемый сообществом и спонсируемый фир- мой Canonical. В настоящее время это, пожалуй, самый популярный дистрибутив GNU/Linux. Он позволяет легко устанавливать и использовать множество свобод- ных программ. Больше того, он позволяет просто перезагрузить ваш компьютер и запустить GNU/Linux с CD! Это даёт возможность в полной мере опробовать новую операционную систему до установки на жёсткий диск. Однако, Ubuntu не полно- стью свободна, так как включает некоторые проприетарные драйверы, микрокод и приложения. [[Ubuntu](http://www.ubuntu.com/)]
    - **LibreOffice**. Это превосходный офисный пакет, разрабатываемый сообществом и включающий в себя компоненты для создания текста, презентаций, электронных

1 Свободное и открытое программное обеспечение (*прим.перев.*)

таблиц, рисунков и многого другого. Он также позволяет открывать и редактиро- вать файлы MS Word и MS Powerpoint. Он работает практически на всех платфор- мах и является полностью свободным и открытым программным обеспечением. [[LibreOffice](http://www.libreoffice.org/)]

* + - **Mozilla Firefox**. Это веб-браузер нового поколения, являющийся сильнейшим сопер- ником Internet Explorer. Он огненно быстр, и заработал широкое признание своими удобными и впечатляющими возможностями. А концепция расширений позволяет дополнять его самыми разнообразными плагинами. [[Mozilla Firefox](http://www.mozilla.org/products/firefox)]
    - Его «напарник» **Thunderbird** – отличный клиент электронной почты, делающий её чтение чрезвычайно простым. [[Mozilla Thunderbird](http://www.mozilla.org/products/thunderbird)]
    - **Mono**. Это свободная реализация платформы Microsoft .NET. Она позволяет созда- вать и запускать .NET-приложения в GNU/Linux, Windows, FreeBSD, Mac OS и на мно- гих других платформах. [[Mono](http://www.mono-project.com/)], [[ECMA](http://www.ecma-international.org/)], [[.NET](http://www.microsoft.com/net)]
    - **Apache web server**. Это популярный открытый веб-сервер. По сути, это *самый* по- пулярный веб-сервер на планете. На нём работает более половины всех сайтов. Да, Apache действительно обслуживает больше сайтов, чем все остальные веб-серверы (включая Microsoft IIS) вместе взятые. [[Apache](http://httpd.apache.org/)]
    - **MySQL**. Это чрезвычайно популярный открытый сервер баз данных. Он наиболее известен своей ошеломляющей скоростью. Это его обозначают буквой «M» в соче- тании «LAMP», на котором работает большинство сайтов в Интернете. [[MySQL](http://www.mysql.comMySQL/)]
    - **VLC Player**. Это проигрыватель, который может воспроизводить всё, начиная от DivX и до MP3, до Ogg, до VCD, до DVD, до … и кто сказал, что это не забавно?

;-) [[VLC media player](http://www.videolan.org/vlc/)]

* + - **GeexBox** – дистрибутив GNU/Linux, созданный для проигрывания фильмов сразу после загрузки с CD! [[GeexBox](http://geexbox.org/en/start.html)]

Этот список предназначен только лишь для того, чтобы вкратце передать вам мысль, на самом же деле существует ещё огромное множество свободного ПО, такого как язык Perl, язык PHP, система управления содержимым веб-сайтов Drupal, сервер баз данных PostgreSQL, игра TORCS, среда разработки KDevelop, проигрыватель Xine, текстовый ре- дактор VIM, редактор Quanta+, аудио-плеер Banshee, графический редактор GIMP, … этот список можно продолжать вечно.

Чтобы отследить свежие слухи в мире свободного ПО, посетите следующие сайты:

* + - [linux.com](http://www.linux.com/)
    - [LinuxToday](http://www.linuxtoday.com/)
    - [NewsForge](http://newsforge.com/)
    - [DistroWatch](http://www.distrowatch.com/)

Узнать больше о свободном ПО можно на следующих сайтах:

* + - [SourceForge](http://www.sourceforge.net/)
    - [FreshMeat](http://www.freshmeat.net/)

Так что вперёд – осваивать бескрайний, свободный и открытый мир СПО!

# Приложение: о книге

## Колофон

Практически все программы, которые я использовал при написании этой книги, являются

[*свободными*](#_bookmark213).

### Рождение книги

При написании первого черновика этой книги в основе моей системы была Red Hat 9.0 Linux, но уже шестую версию черновика я писал на Fedora Core 3 Linux.

В начале для написания книги я использовал KWord (как я и описывал в [*Предисловии*](#_bookmark18)).

### Отрочество

Позже я перешёл на формат DocBook XML и использовал Kate, но это показалось мне слишком скучным. Поэтому я перешёл на OpenOffice, который замечательно подходил со своим уровнем управления форматированием и возможностью генерировать PDF, но он выдавал слишком небрежные HTML-страницы.

Наконец, я открыл для себя XEmacs и переписал всю книгу с нуля в формате DocBook XML (снова), так как решил, что этот формат – надолго.

Для шестой версии черновика я решил использовать Quanta+. При этом я использовал стандартные таблицы стилей XSL, которые шли в комплекте с Fedora Core 3 Linux. Потом я написал CSS документ, чтобы придать цвет и стиль HTML-страницам. Я также на ско- рую руку написал лексический анализатор (конечно, на Python), который осуществлял автоматическую подсветку синтаксиса в примерах программ.

Для седьмой редакции я использовал [MediaWiki](http://www.mediawiki.org/) в качестве основы для своего [сай-](http://www.swaroopch.com/notes/) [та](http://www.swaroopch.com/notes/). Теперь я всё редактирую прямо на сайте, а читатели могут сразу же чи- тать/редактировать/обсуждать содержимое на вики-странице.

Благодаря [расширению ViewSourceWith для Firefox](https://addons.mozilla.org/en-US/firefox/addon/394), которое интегрируется с Vim, я про- должал использовать Vim для редактирования.

A Byte of Python (Russian), Версия 2.02

### Сейчас

Использую [Vim](http://www.swaroopch.com/notes/vim), [Pandoc](http://johnmacfarlane.net/pandoc/README.html), и Mac OS X.

## Об авторе

<http://www.swaroopch.com/about/>

# Приложение: История версий

* **2.0** (20/10/2012)
  + Переписано в формате [Pandoc](http://johnmacfarlane.net/pandoc/README.html) Спасибо моей жене, которая перевела большую часть текста из формата Mediawiki
  + Упрощение текста, удаление таких необязательных разделов, как nonlocal и метаклассы
* **1.90** (04/09/2008)
  + Возобновление после перерыва в 3.5 года!
  + Обновление для Python 3.0
  + Переписано в формате [MediaWiki](http://www.mediawiki.org/) (снова)
* **1.20** (13/01/2005)
  + Полностью переписано в [Quanta+](https://en.wikipedia.org/wiki/Quanta_Plus) на [Fedora](http://fedoraproject.org/) Core 3 со множеством исправлений и дополнений. Много новых примеров. Заново переписан формат DocBook.
* **1.15** (28/03/2004)
  + Незначительные исправления.
* **1.12** (16/03/2004)
  + Дополнения и исправления.
* **1.10** (09/03/2004)
  + Исправления опечаток, благодаря множеству отзывов заинтересованных чита- телей.
* **1.00** (08/03/2004)
  + После колоссального числа отзывов и предложений от читателей я произвёл значительную переработку текста наряду с исправлением опечаток.
* 0.99 (22/02/2004)
  + Добавлена новая глава о модулях. Также добавлен фрагмент о переменном чис- ле аргументов в функциях.
* 0.98 (16/02/2004)
  + Написан скрипт на Python и таблица стилей CSS для улучшения вывода в XHTML, включая недоделанный-но-функциональный лексический анализа- тор для подсветки синтаксиса в примерах программ
* 0.97 (13/02/2004)
  + Ещё один заново переписанный черновик в формате DocBook XML (снова). Книга существенно улучшена – она стала значительно более связанной и чи- табельной.
* 0.93 (25/01/2004)
  + Добавлено описание IDLE и другие вещи, относящиеся к Windows.
* 0.92 (05/01/2004)
  + Изменения в нескольких примерах.
* 0.91 (30/12/2003)
  + Исправлены опечатки. Сделаны наброски многих разделов.
* 0.90 (18/12/2003)
  + Добавлены 2 главы. Формат [OpenOffice](https://en.wikipedia.org/wiki/OpenOffice) с исправлениями.
* 0.60 (21/11/2003)
  + Полностью переписано и расширено.
* 0.20 (20/11/2003)
  + Исправлены некоторые опечатки и ошибки.
* 0.15 (20/11/2003)
  + Переведено в формат [DocBook XML](https://en.wikipedia.org/wiki/DocBook) при помощи XEmacs.
* 0.10 (14/11/2003)
  + Самый первый набросок в редакторе [KWord](https://en.wikipedia.org/wiki/Kword).

# Приложение: Инструкция по переводу

Полный исходный текст книги доступен в Git-репозитории [https://github.com/swaroopch/](https://github.com/swaroopch/byte_of_python) [byte\_of\_python](https://github.com/swaroopch/byte_of_python) .

[Создайте ответвление репозитория](https://help.github.com/articles/fork-a-repo).

Затем скачайте репозиторий на свой компьютер. Для этого нужно быть знакомым с [Git](http://www.git-scm.com/).

Отредактируйте файлы .pd на своём родном языке. Прочитайте [Pandoc README](http://johnmacfarlane.net/pandoc/README.html#pandocs-markdown), чтобы познакомиться с форматированием текста

Затем следуйте указаниям в [README](http://github.com/swaroopch/byte_of_python#README) для установки программ, необходимых для конвер- тирования исходных файлов в PDF и т.п.