Первые шаги

Введение

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 интерпретатора

Откройте окно терминала (как было описано в главе *Установка*) и запустите интерпретатор Python, введя команду `python3` и нажав `Enter`.

Пользователи Windows могут запустить интерпретатор в командной строке, если установили переменную *PATH* надлежащим образом. Чтобы открыть командную строку в Windows, зайдите в меню «Пуск» и нажмите «Выполнить...». В появившемся диалоговом 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 неудивительно [1]) 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 командной 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 каждый 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 редактора 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, которая доступна для Windows, Mac OS X и GNU/Linux.

Если вы пользуетесь Windows, **Не используйте Блокнот** – это плохой выбор, поскольку он не обладает функцией подсветки синтаксиса, а также не позволяет автоматически вставлять отступы, что очень важно в нашем случае, как мы увидим позже. Хорошие редакторы, как Komodo Edit, позволяют делать это автоматически.

Опытные программисты, должно быть, уже используют Vim или Emacs. Не стоит даже и говорить, что это два наиболее мощных редактора, и вы только выиграете от их ис-

[1] “print” - *англ.* «Печатать» (*прим. перев.*)

пользования для написания программ на Python. Лично я пользуюсь ими обоими для большинства своих программ, и даже написал книгу о Vim. Я настоятельно рекомендую вам решиться и потратить время на изучение Vim или Emacs, поскольку это будет приносить 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 редактора.

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

Для пользователей Vim

> Существует хорошее введение в использование Vim как мощного IDE для Python, автор – John M Anderson. Также я рекомендую плагин jedi-vim и мой собственный конфигурационный файл.

Для пользователей Emacs

> Существует хорошее введение в использование Emacs как мощного IDE для Python, автор – Ryan McGuire. Также я рекомендую Конфигурацию dotemacs от BG.

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 «Привет, Мир!». Как сказал Simon Cozens[2], это «традиционное заклинание богов программирования, 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», введите строку:

```
print('Привет, Мир!')
```

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

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

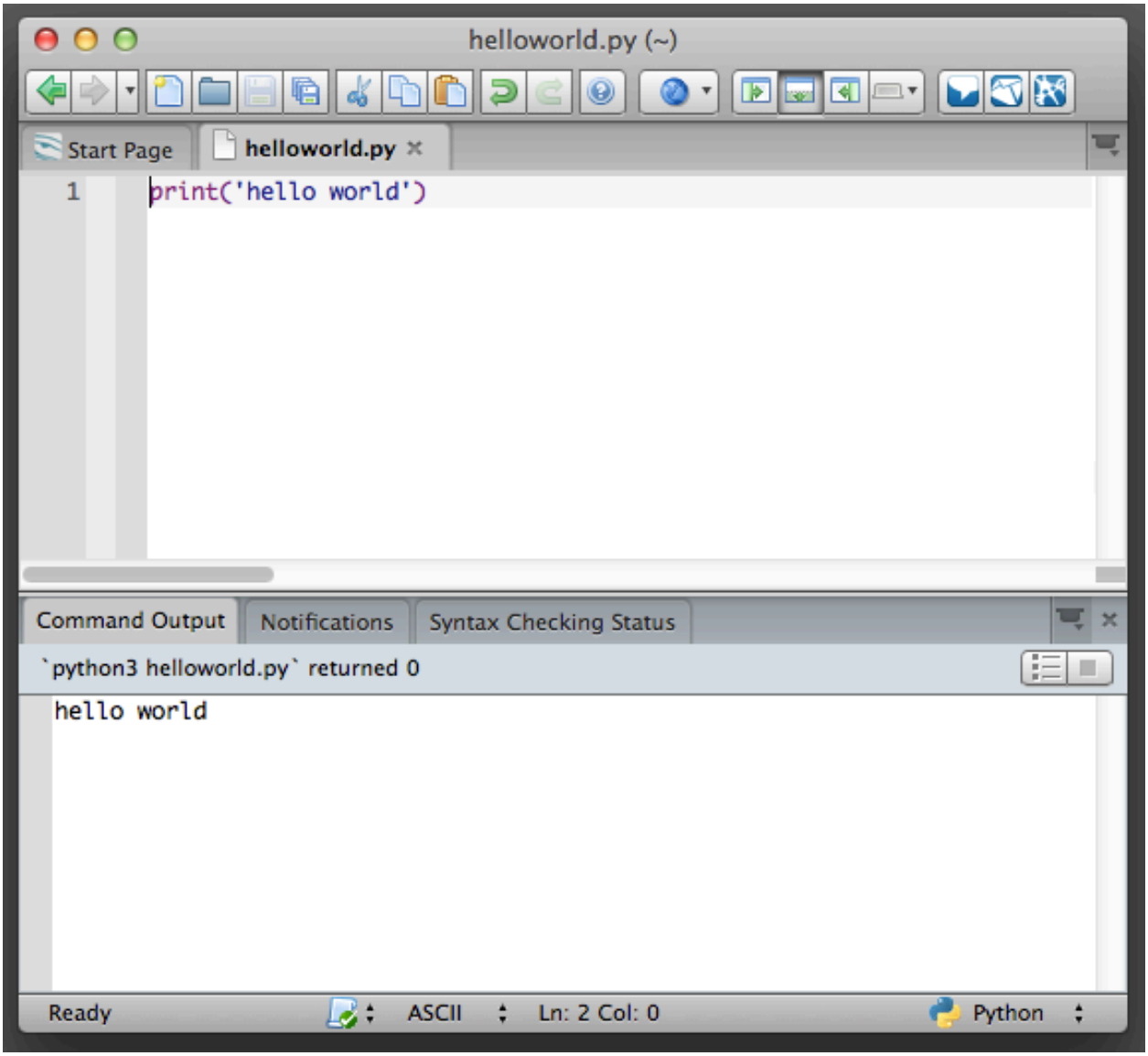
- `C:\\py` в Windows
- `/tmp/py` в GNU/Linux
- `/tmp/py` в Mac OS X

[2] Автор восхитительной книги “Beginning Perl”

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*.
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 точности* 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 имеется всего лишь одно выражение. В этом выражении мы вызываем функцию `print`, которая просто выводит текст `'Привет, Мир!'`. О функциях мы узнаем в *одной из последующих глав*, а пока вам достаточно понять, что всё, что вы укажете в скобках, будет выведено на экран. В данном примере мы указали `'Привет, Мир!'`.

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] добавлением атрибута исполнимости для всех пользователей в системе[4].

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

[4] В указанной команде буква “**a**” взята из слова “all” (*англ.* «все»), а буква “**x**” – из слова “e**x**ecute” (*англ.* «исполнять») – *прим. перев.*

```
$ 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, сможет 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 программу нужно интерпретатором, положение которого указано в первой строке файла программы.

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 переменной окружения `PATH`.

При попытке запуска какой-либо программы система ищет её в каталогах, перечисленных в переменной окружения `PATH`, и запускает. Таким образом, мы можем сделать программу 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!
```

Мы можем вывести на экран значение переменной `PATH` при помощи команды `echo`, добавив перед именем переменной символ `$`, чтобы указать оболочке, что мы хотим получить значение этой переменной. Мы видим, что `/home/swaroop/bin` – один из каталогов 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 каталог, который я хочу добавить к переменной `PATH`.

Этот метод полезен для написания сценариев, которые будут доступны для запуска в любой момент из любого места. По сути, это равносильно созданию собственных команд, как `cd` или любой другой, которые часто используются в терминале GNU/Linux или приглашении 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, объясняющих:

- 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.

Это будет полезно для тех, кто будет читать вашу программу, так как им легче будет понять, 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 литеральными, потому что они «буквальны»[1] – вы используете их значение буквально. Число `2` всегда 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 $52.3 \cdot 10^{-4}$.
- 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.

Замечание для опытных программистов

В Python 3 нет ASCII-строк, потому что Unicode является надмножеством (включает в себя) ASCII. Если необходимо получить строку строго в кодировке ASCII, используйте `str.encode("ascii")`. Подробнее смотрите в обсуждении этого вопроса на StackOverflow. По умолчанию все строки в Unicode.

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

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 двойные или тройные 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. В последствии 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 пригождается 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 соответствующими 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 начинает отсчёт с 0, поэтому первая позиция – номер 0, вторая – номер 1 и т.д.

Заметьте, мы ведь могли добиться того же самого результата и объединением 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**.

Переменные

Использование одних лишь литеральных констант может скоро наскучить – нам ведь нужен способ хранения любой информации и манипулирования ею. Вот здесь на сцену 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, поэтому переменным даются имена.

Имена идентификаторов

Переменные – это частный случай идентификаторов. *Идентификаторы* – это имена, присвоенные *чему-то* для его обозначения. При выборе имён для идентификаторов необходимо 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_δξЖђёўЩдΞέά`.
- 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**. Основными типами являются числа и строки, о которых мы уже говорили. В дальнейших главах мы увидим, как создавать свои собственные типы при помощи *классов*.

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 следующий пример и запустите программу.

Как писать программы на Python

Впредь стандартная процедура сохранения и запуска программы на Python будет выглядеть 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*, как объяснялось ранее.

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 константы `5` переменной `i`, используя оператор присваивания (`=`). Эта строка называется предложением и указывает, что должно быть произведено некоторое действие, и в данном случае мы связываем имя переменной `i` со значением `5`. 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. После 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 предварительного объявления или определения типа данных не требуется/применяется.

Логические и физические строки

Физическая строка – это то, что вы *видите*, когда набираете программу. Логическая строка – 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 соответствует физической строке.

Python неявно стимулирует использование по одному предложению на строку, что облегчает чтение кода.

Чтобы записать более одной логической строки на одной физической строке, вам придётся 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 точки с запятой. Кстати, я *никогда* не использовал и даже не встречал точки с запятой в программах 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 логической 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 черты не обязательно. Это относится к случаям, когда в логической строке есть открывающаяся круглая, квадратная или фигурная скобка, но нет закрывающейся. Это называется **неявным объединением строк**. Вы сможете увидеть это в действии в программах с использованием *списков* в дальнейших главах.

Отступы

В Python пробелы важны. Точнее, **пробелы в начале строки важны**. Это называется **отступами**. Передние отступы (пробелы и табуляции) в начале логической строки исполь-

зуются для определения уровня отступа логической строки, который, в свою очередь, используется для группировки предложений.

Это означает, что предложения, идущие вместе, **должны** иметь одинаковый отступ. Каждый такой набор предложений называется **блоком**. В дальнейших главах мы увидим примеры 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, отображённая 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 используется на протяжении всей программы, конечно). Случаи, в которых вы можете использовать новые блоки, будут подробно описаны в дальнейших главах, как например, в *главе «Поток команд»*.

Как отступать

Не смешивайте пробелы и символы табуляции в отступах, поскольку не на всех платформах 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 использовать 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 табуляции, то и вы используйте в этом файле символы табуляции для отступов.

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

Замечание для программистов на статических языках программирования

Python всегда будет использовать отступы для выделения блоков и никогда не будет использовать 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*. Простой пример выражения: `2 + 3`. Выражение можно разделить на операторы и операнды.

Операторы – это некий функционал, производящий какие-либо действия, который может быть представлен в виде символов, как например `+`, или специальных зарезервированных слов. Операторы могут производить некоторые действия над данными, и эти данные 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 также 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'`
`-`	Вычитание	Даёт разность двух чисел; если первый операнд отсутствует, он считается равным нулю	`-5.2` даст отрицательное число, а `50 - 24` даст `26`.
`*`	Умножение	Даёт произведение двух чисел или возвращает строку, повторённую заданное число раз.	`2 * 3` даст `6`. `'la' * 3` даст `'lalala'`.

Оператор	Название	Объяснение	Примеры
`**`	Возведение в степень	Возвращает число `x`, 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`.
`//`	Целочисленное деление	Возвращает неполное частное от деления	`4 // 3` даст `1`.
`%`	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, ни что иное как десятичное `5`
`&`	Побитовое И	Побитовая операция И над числами	`5 & 3` даёт `1`.
`|`	Побитовое ИЛИ	Побитовая операция ИЛИ над числами	`5 | 3` даёт `7`
`^`	Побитовое ИСКЛЮЧИТЕЛЬНО ИЛИ	Побитовая операция ИСКЛЮЧИТЕЛЬНО ИЛИ	`5 ^ 3` даёт `6`
`~`	Побитовое НЕ	Побитовая операция НЕ для числа `x` соответствует `-(x+1)`	`~5` даёт `-6`.

Оператор	Название	Объяснение	Примеры
`<`	Меньше	Определяет, верно ли, что `x` меньше `y`. Все операторы сравнения возвращают `True` или `False` [1]. 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`.
`==`	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`	Логическое НЕ	Если `x` равно `True`, оператор вернёт `False`. Если же `x` равно `False`, получим `True`.	`x = True; not x` даёт `False`.
`and`	Логическое 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.

[1] "True" - *англ.* «Верно (Правда)»; "False" - *англ.* "«ибочно (Ложь)». (*прим. перев.*)

Оператор	Название	Объяснение	Примеры
`or`	Логическое 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.

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 умножение? Школьный курс математики говорит нам, что умножение должно производиться в первую очередь. Это означает, что оператор умножения имеет более высокий приоритет, чем оператор сложения.

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

Эта таблица взята из Справочника по языку Python (англ.) и приводится здесь для полноты описания. На практике лучше использовать скобки для группировки операторов и операндов, чтобы в явном виде указать порядок вычисления выражений. Заодно это облегчит чтение программы. Более подробно см. в разделе *Изменение порядка вычисления* 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 деления
`+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]
`[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. Например, `+` и `-` имеют равный приоритет.

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 операторов. Как и всё остальное, скобки нужно использовать разумно (не перестарайтесь) и избегать излишних, как в `(2 + (3 * 4))`.

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

[2] “tuple” - *англ.* «кортеж» (*прим. перев.*)

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`. Некоторые операторы, как, например, оператор присваивания, имеют ассоциативность справа налево, т.е. `a = b = c` рассматривается как `a = (b = c)`.

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 выражения `2 * (length + breadth)` в функцию `print`.

Также обратите внимание, как Python «красиво печатает» результат. Несмотря на то, что мы не указали пробела между `'Площадь равна'` и переменной `area`, Python подставляет его за нас, чтобы получить красивый и понятный 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 практике.

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; скажем, печатала «Доброе утро» или «Добрый вечер» в зависимости от времени суток.

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

Оператор if

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

Пример: (сохраните как `if.py`)

```
number = 23
guess = int(input('Введите целое число : '))

if guess == number:
    print('Поздравляю, вы угадали,') # Здесь начинается новый блок
    print('(хотя и не выиграли никакого приза!)') # Здесь заканчивается новый блок
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

print('Завершено')
# Это последнее выражение выполняется всегда после выполнения оператора if
```

[1] if – *англ.* «если» (*прим.перев.*)

[2] Соответствует булевому значению `True` (*прим.перев.*)

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

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, совпадают 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 фрагменты программы. Мы узнаем о них больше в *следующей главе*.

Мы передаём встроенной функции `input` строку, которую она выводит на экран и ожидает ввода от пользователя. Как только мы ввели что-нибудь и нажали клавишу `Enter`, функция `input()` возвращает строку, которую мы ввели. 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. Обратите внимание, что мы используем соответствующие уровни отступа, чтобы указать Python, какие выражения относятся к какому блоку. Вот почему отступы 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 показываем, 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 числом отступов, конечно).

Внутри if-блока оператора `if` может быть другой оператор `if` и так далее – это называется вложенным[4] 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` можно достичь того же самого (а в некоторых случаях можно даже использовать *словарь*, чтобы сделать это быстро).

Оператор while

Оператор `while` позволяет многократно выполнять блок команд до тех пор, пока выполняется 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
```

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

```
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, число несколько меньше.
Введите целое число : 22
Нет, число несколько больше.
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 преимущество 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 переменную `running` в значение `True` перед запуском цикла. Прежде всего проверяется, равно ли значение переменной `running` `True`, а затем происходит переход к соответствующему *while-блоку*. После выполнения этого блока команд условие, которым в данном случае является переменная `running`, проверяется снова. Если оно истинно, while-блок запускается снова, в противном случае происходит переход к дополнительному else-блоку, а затем – к следующему 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 эквивалентными 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* 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 генерируем эту последовательность, используя встроенную функцию `range`[5].

Мы задаём два числа, и `range` возвращает последовательность чисел от первого 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* второго 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 следующий элемент. Чтобы увидеть всю последовательность чисел сразу, используйте `list(range())`. Списки[6] подробно рассматриваются в главе *Структуры*

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

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

данных.

Затем цикл `for` осуществляет итерацию по этому диапазону - `for i in range(1,5)` эквивалентно `for i in [1, 2, 3, 4]`, что напоминает присваивание 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*.

Помните также, что цикл `for..in` работает для любой последовательности. В нашем случае 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 соответствовало бы выражение `for i in range(0,5)`. Как видно, в Python цикл `for` проще, более выразителен и менее подвержен ошибкам.

Оператор break

Оператор `break` служит для *прерывания*[7] цикла, т.е. остановки выполнения команд даже если условие выполнения цикла ещё не приняло значения `False` или последовательность элементов не закончилась.

Важно отметить, что если циклы `for` или `while` прервать оператором *break*, соответствующие 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:

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

```
$ 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 выводим 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 функции `len`.

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

Поэтический Python Swaroop'a

Для ввода строк здесь я использовал мини-стишок, который сам сочинил. Он называется *Поэтический Python Swaroop'a*[8]

Программировать весело.
Если работа скучна,
Чтобы придать ей весёлый тон -
    используй Python!

[8] **Swaroop's Poetic Python**:
Programming is fun.
When the work is done,
if you wanna make your work also fun:
    use Python!

Оператор continue

Оператор `continue` используется для указания Python, что необходимо пропустить все оставшиеся команды в текущем блоке цикла и *продолжить*[9] со следующей итерации цикла.

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
```

Как это работает:

> В этой программе мы запрашиваем ввод со стороны пользователя, но обрабатываем 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 помощи 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 используемые конструкции Python, поэтому овладеть ими очень важно.

Далее мы увидим, как создавать и использовать функции.

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

Функции

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

Функция – это, пожалуй, *наиболее* важный строительный блок любой нетривиальной программы (на любом языке программирования), поэтому в этой главе мы рассмотрим различные 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 следует 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 объявлены какие-либо переменные. Параметры функции – это некие входные данные, которые мы можем передать функции, чтобы получить соответствующий 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*, тогда 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 качестве аргументов. Во втором случае мы вызываем функцию с переменными в качестве аргументов. `printMax(x, y)` назначает значение аргумента `x` параметру `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 связаны с другими переменными с таким же именем за пределами функции – т.е. имена переменных являются *локальными* в функции. Это называется *областью видимости* переменной. Область видимости всех переменных ограничена блоком, в котором они объявлены, 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. Поэтому когда мы заменяем значение `x` в функции, `x`, объявленный в основном блоке, остаётся незатронутым.
>
> Последним вызовом функции `print` мы выводим значение `x`, указанное в основном блоке, подтверждая таким образом, что оно не изменилось при локальном присваивании значения в ранее вызванной функции.

Зарезервированное слово «global»

Чтобы присвоить некоторое значение переменной, определённой на высшем уровне программы (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 зарезервированного слова `global`. Без применения зарезервированного слова `global` невозможно присвоить значение переменной, определённой за пределами функции.

Можно использовать уже существующие значения переменных, определённых за пределами 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 переменная 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
Знчение 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 имени `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 видимости. Есть ещё один тип области видимости, называемый «нелокальной» (`nonlocal`) областью видимости, который представляет собой нечто среднее между первыми двумя. 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 первой строке `func_outer` находится ни в локальной области видимости (определение переменной не входит в блок `func_inner`), ни в глобальной области видимости (она также и не в основном блоке программы). Мы объявляем, что хотим использовать именно эту переменную `x`, следующим образом: `nonlocal x`.
>
> Попробуйте заменить «`nonlocal x`» на «`global x`», а затем удалить это зарезервированное слово, и пронаблюдайте за разницей между этими двумя случаями.

Значения аргументов по умолчанию

Зачастую часть параметров функций могут быть *необязательными*, и для них будут использоваться 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 присваивания (=) с последующим значением.

Обратите внимание, что значение по умолчанию должно быть константой. Или точнее говоря, оно должно быть неизменным[1] – это объясняется подробнее в последующих главах. А пока запомните это.

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

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].
>
> 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] фразу 5 раз.

Важно: Значениями по умолчанию могут быть снабжены только параметры, находящиеся в конце списка параметров. Таким образом, в списке параметров функции параметр со значением по умолчанию не может предшествовать параметру без значения по умолчанию. 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.*

Ключевые аргументы

Если имеется некоторая функция с большим числом параметров, и при её вызове требуется 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 остальные параметры имеют значения аргумента по умолчанию.

Пример: (сохраните как `func_key.py`)

[2] times – *англ.* «раз» (*прим. перев.*)
[3] say – *англ.* «сказать» (*прим. перев.*)

```
def func(a, b=5, c=10):
    print('a равно', a, ', b равно', b, ', a с 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, a с равно 10
a равно 25, b равно 5, a с равно 24
a равно 100, b равно 5, a с 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 умолчанию, равное `5`.

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

Переменное число параметров [4]

Иногда бывает нужно определить функцию, способную принимать *любое* число параметров. Этого можно достичь при помощи звёздочек (сохраните как `total.py`):

```
def total(initial=5, *numbers, **keywords):
    count = initial
    for number in numbers:
        count += number
    for key in keywords:
        count += keywords[key]
    return count

print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

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

Вывод:

```
$ python total.py
166
```

Как это работает:

> Когда мы объявляем параметр со звёздочкой (например, `*param`), все позиционные 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*.

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 сокращённый оператор, позволяющий вместо `x = x + y` просто написать `x += y`.

Если вам нужны аргументы, передаваемые только по ключу, но не нужен параметр 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] 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 эквивалентен выражению `return None`. `None` – это специальный тип данных в Python, обозначающий 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

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

`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.

Примечание: Существует встроенная функция `max`, в которой уже реализован функционал «поиск максимума», так что пользуйтесь этой встроенной функцией, где это возможно.

Строки документации [6]

Python имеет остроумную особенность, называемую *строками документации*, обычно обозначаемую сокращённо *docstrings*. Это очень важный инструмент, которым вы обязательно должны пользоваться, поскольку он помогает лучше документировать программу и облегчает её понимание. Поразительно, но строку документации можно получить, например, 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.
```

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

Как это работает:

> Строка в первой логической строке функции является *строкой документации* для этой функции. Обратите внимание на то, что строки документации применимы также к *модулям* и *классам*, о которых мы узнаем в соответствующих главах.
>
> Строки документации принято записывать в форме многострочной[7] строки, где первая строка начинается с заглавной буквы и заканчивается точкой. Вторая 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 помощью атрибута этой функции (т.е. имени, принадлежащего ей) `__doc__` (обратите внимание на *двойное подчёркивание*). Просто помните, что Python представляет *всё* в виде объектов, включая функции. Мы узнаем больше об объектах в главе *о классах*.

Если вы пользовались функцией `help()` в Python, значит вы уже видели строки документации. 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 нажать клавишу `q` для выхода из справки (`help`).

Точно так же автоматические инструменты могут получать документацию из программы. 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, которые 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 посторонним библиотекам), мы опустим эту возможность из нашего обсуждения. Если вам интересно почитать об аннотациях, просмотрите **PEP 3107**.

Резюме

Мы рассмотрели достаточно много аспектов функций, но тем не менее, вы должны понимать, 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.

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

Далее мы увидим, как использовать и создавать модули Python.

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 программах? 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 программирования C, 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\nПеременная PYTHONPATH содержит', sys.path, '\n')
```

Вывод:

```
$ 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',
'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] 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 конфликтует с именем переменной `argv`, которая может использоваться в вашей программе.

Переменная `sys.argv` является *списком* строк (списки будут детально обсуждаться в *одной из последующих глав*). Она содержит список *аргументов командной строки*, т.е. аргументов, переданных программе из командной строки.

Если вы используете среду разработки[2] для написания и запуска программ, поищите где-нибудь в её меню возможность передавать параметры командной 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]. Python сохраняет аргументы командной строки в переменной `sys.argv` для дальнейшего использования.

Помните, что имя запускаемого сценария[4] всегда является первым аргументом в списке `sys.argv`. Так что в приведённом примере `'using_sys.py'` будет элементом `sys.argv[0]`, `'we'` – `sys.argv[1]`, `'are'` – `sys.argv[2]`, а `'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 модули,

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

[2] IDE - *от англ.* "Integrated Development Environment" – «интегрированная среда разработки» (*прим. перев.*)

[3] "we are arguments" - *англ.* «мы аргументы» (*прим. перев.*)

[4] Программу на интерпретируемом языке программирования также называют *сценарием* или *скриптом* (*прим. перев.*)

расположенные в текущем каталоге, можно импортировать напрямую. В противном случае придётся поместить свой модуль в один из каталогов, перечисленных в `sys.path`.

Помните, что текущий каталог – это каталог, в котором была запущена программа. Выполните `import os; print(os.getcwd())`, чтобы узнать текущий каталог программы.

Файлы байткода .pyc

Импорт модуля – относительно дорогостоящее мероприятие, поэтому Python предпринимает 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 *"Введение"* о том, как работает Python?). Такой файл `.pyc` полезен при импорте модуля в следующий раз в другую программу – это произойдёт намного быстрее, поскольку значительная 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 соответствующие 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 вместо этого оператор `import`, чтобы предотвратить конфликты имён и не затруднять чтение программы.

Пример:

```
from math import *
n = 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 neprost",a)
                b = 1
            else:
                pass

    if (b != 1):
        print("a prost",a)
        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 импортирован. 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 импортируется 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:

```
$ python3 using_name.py
Эта программа запущена сама по себе.

$ python3
>>> import using_name
Меня импортировали в другой модуль.
>>>
```

Как это работает:

> В каждом модуле Python определено его имя – `__name__`[5] . Если оно равно `'__main__'`, это означает, что модуль запущен самостоятельно пользователем, и мы можем выполнить соответствующие действия.

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

Создание собственных модулей

Создать собственный модуль очень легко. Да вы всё время делали это! Ведь каждая программа на Python также является и модулем. Необходимо лишь убедиться, что у неё установлено 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 которую 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__)
```

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 доступа к элементам модуля. Python повсеместно использует одно и то же обозначение 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 объявлено имя `__version__`, возникнет конфликт. Это весьма вероятно, так как объявлять версию любого модуля при помощи этого имени – общепринятая практика. Поэтому всегда 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». Выполните команду "`import this`", чтобы узнать больше, а также просмотрите это обсуждение, в котором приводятся примеры по каждому из принципов.

Функция dir

Вы можете использовать встроенную функцию `dir`, чтобы получить список идентификаторов, которые объект определяет. Так в число идентификаторов модуля входят функции, классы и переменные, определённые в этом модуле.

Когда вы передаёте функции `dir()` имя модуля, 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__', '__s
tderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_compact_freelists',
'_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', 'getdefaultencoding', 'getfil
esystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
'gettrace', 'getwindowsversion', 'hexversion', 'intern', 'maxsize', 'maxunicode
', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platfor
m', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile', 'setrecursionlimit
', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_in
```

```
fo', '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 `sys`. Видим огромный список атрибутов, содержащихся в нём.

Затем мы вызываем функцию `dir`, не передавая ей параметров. По умолчанию, 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 переменную/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* переменной/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. Например, 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 организации 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:

```
| - <некоторый каталог из sys.path>/
| |---- world/
|       |---- __init__.py
|       |---- asia/
|       |     |---- __init__.py
|       |     |---- india/
|       |           |---- __init__.py
|       |           |---- foo.py
|       |---- africa/
|             |---- __init__.py
|             |---- madagascar/
|                   |---- __init__.py
|                   |---- bar.py
```

Пакеты – это удобный способ иерархически организовать модули. Такое часто встречается в *стандартной библиотеке*.

Резюме

Точно так же, как функции являются многократно используемыми фрагментами программ, модули являются многократно используемыми программами. Пакеты – это способ иерархической организации модулей. Стандартная библиотека Python является примером 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] – это структура данных, которая содержит упорядоченный набор элементов, т.е. хранит *последовательность* элементов. Это легко представить, если вспомнить список покупок, 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*.

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 объект данного класса. Например, Python предоставляет метод `append` для класса `list`, который позволяет добавлять элемент к концу списка. Так `mylist.append('and item')`

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

добавит эту строку к списку `mylist`. Обратите внимание на обозначение точкой для доступа 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 переменные/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), 'покупок.')

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

print('\nТакже 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:

```
$ python3 using_list.py
Я должен сделать  4 покупок.

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']
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*.

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 – *неизменяемы*.

Далее после совершения покупки мы хотим удалить её из списка. Это достигается 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 **неизменяемы**, 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 функция 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*.

Теперь мы перемещаем этих животных в новый зоопарк, поскольку старый зоопарк закрывается. Поэтому кортеж `new_zoo` содержит тех животных, которые 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 списков. 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, содержащий 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 контактную 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 полностью 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 обращением 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'a:", ab['Swaroop'])

# Удаление пары ключ-значение
```

```
del ab['Spammer']

print('\nВ адресной книге {0} контактов\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("\nАдрес Guido:", ab['Guido'])
```

Вывод:

```
$ python3 using_dict.py
Адрес Swaroop'a: swaroop@swaroopch.com

В адресной книге 3 контактов

Контакт Swaroop с адресом swaroop@swaroopch.com
Контакт Matsumoto с адресом matz@ruby-lang.org
Контакт Larry с адресом larry@wall.org

Адрес Guido: guido@python.org
```

Как это работает:

Мы создаём словарь `ab`[2] при помощи обозначений, описанных ранее. Затем мы обращаемся к парам ключ-значение, указывая ключ в операторе индексирования, 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 операции нет необходимости знать, какое значение соответствует данному ключу.

Далее мы обращаемся ко всем парам ключ-значение нашего словаря, используя метод `items`, который возвращает список кортежей, каждый из которых содержит пару элементов: ключ и значение. Мы получаем эту пару и присваиваем 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 некоторого значения, как мы сделали для Guido в примере выше.

[2] address book – *англ.* «адресная книга» (*прим. перев.*)

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 использовали 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 компиляторов называется *таблицей имён*).

Последовательности

Списки, кортежи и строки являются примерами последовательностей. Но что такое последовательности 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 элемент последовательности.

Все три типа последовательностей, упоминавшиеся выше (списки, кортежи и строки), также предоставляют операцию получения **вырезки**, которая позволяет получить вырезку 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:

```
$ python3 seq.py
Элемент 0 - яблоки
Элемент 1 - манго
Элемент 2 - морковь
Элемент 3 - бананы
Элемент -1 - бананы
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*. Когда мы указываем число в квадратных скобках после последовательности, как показано выше, Python извлекает элемент, соответствующий указанной позиции 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 отсчитывается от конца последовательности. Поэтому `shoplist[-1]` указывает на последний элемент последовательности `shoplist`, а `shoplist[-2]` – на предпоследний.

Операция вырезки производится при помощи указания имени последовательности, 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) указывает, 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. Отрицательные числа обозначают позицию с конца последовательности. Например, `shoplist[:-1]` вернёт вырезку из последовательности, исключающую последний 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, находящиеся 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 оболочку интерпретатора Python, т.е. его командную строку, чтобы сразу видеть результат. Последовательности 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, когда присутствие объекта в наборе важнее порядка или того, сколько раз данный объект там встречается.

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

```
>>> bri = set(['Бразилия', 'Россия', 'Индия'])
>>> 'Индия' in bri
True
>>> 'США' in bri
False
>>> bric = bri.copy()
>>> bric.add('Китай')
>>> bric.issuperset(bri)
True
>>> bri.remove('Россия')
>>> bri & bric # OR bri.intersection(bric)
{'Бразилия', 'Индия'}
```

Как это работает:

> Этот пример достаточно нагляден, так как использует основы теории множеств 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 эффект, 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 последовательности, или другого сложного объекта (не такого простого *объекта*, как целое число), вам следует воспользоваться операцией вырезки. Если вы просто присвоите имя переменной другому имени, оба они будут *ссылаться* на один и тот же объект, а это может привести к проблемам, если вы не осторожны.

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`)

```
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 = '_*_'
```

```
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. Оператор `in` используется для проверки, является ли некоторая строка частью данной строки.

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

Резюме

Мы детально рассмотрели различные встроенные структуры данных Python. Эти структуры 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.*)

Решение задач

Мы рассмотрели различные части языка Python, и теперь посмотрим, как все эти части работают 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* файлы необходимо копировать? *Как* их хранить? *Где* их хранить?

После надлежащего анализа мы **проектируем** нашу программу. Мы создаём список, описывающий 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 получиться 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 её со страницы проекта GnuWin32 и добавить "`C:\Program Files\GnuWin32\bin`" к системной переменной окружения PATH, аналогично тому, как мы это *делали* для самой команды "python". Обратите внимание, что для этого подойдёт любая команда 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 кода, 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 тот путь, который вы будете использовать.

# 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], 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 выведенную 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

[1] debug – применительно к компьютерным программам обозначает отладку (обнаружение и устранение ошибок, которые при этом принято называть «bug», т.е. «жук»). По всей видимости, это берёт своё начало 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.*)

команде “zip”, чтобы выяснить, в чём может быть проблема. Если команда успешно выполняется, проверьте, совпадает ли ваша программа на Python в точности с программой, приведённой выше.

Как это рабтает:

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]. Каталог назначения – это каталог, в котором мы сохраняем все резервные копии, и он указывается в переменной `target_dir`. Именем zip-архива, который мы создаём, будет текущая дата и время, которые генерируются при помощи функции `time.strftime()`. У него будет расширение `.zip`, и храниться он будет в каталоге `target_dir`.

Обратите внимание на употребление переменной `os.sep` – она содержит разделитель пути для конкретной операционной системы, т.е. он будет `'/'` в GNU/Linux и Unix[3], `'\\'` в Windows и `':'` в Mac OS. Использование `os.sep` вместо этих символов напрямую делает программу переносимой, и она сможет 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 формата `%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.

Имя конечного zip-файла мы создаём при помощи оператора, который *соединяет* строки, т.е. объединяет две строки и возвращает новую. После этого мы создаём строку `zip_command`, которая содержит команду, которую мы намерены выполнить. Проверить, работает ли эта команда, можно запустив её отдельно 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]. Параметр “`-r`” обозначает, что команда архивации должна работать рекурсивно[5] для каталогов, т.е. должна включать все подкаталоги и файлы. Оба параметра объединены и указаны в краткой форме “`-qr`”. За параметрами следует имя создаваемого zip-архива, за которым указывается список файлов и каталогов для резервного копирования. Мы превращаем список `source` в строку, используя уже знакомый нам метод `join`.

[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.*)

Затем мы, наконец, *выполняем* команду при помощи функции `os.system`, которая 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 важных файлов!

Замечание для пользователей Windows

Вместо управляющей последовательности для обратной наклонной черты могут использоваться «сырые»[6] строки. Например, можно писать "`C:\\Documents`" или "`r'C:\Documents'`". Однако, **не** используйте "`'C:\Documents'`", так как в этом случае окажется, что вы пытаетесь применить неизвестную управляющую последовательность `\D`.

Теперь, когда у нас есть рабочий сценарий резервного копирования, мы можем использовать его для создания копий наших файлов. Пользователям GNU/Linux и Unix рекомендуется сделать этот программный файл *исполнимым*, чтобы иметь возможность запускать его в любое время из любого места. Это называется **операционной** фазой или **развёртыванием** программы.

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

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 текущей датой в качестве имени, который в свою очередь, расположен в главном каталоге для хранения резервных копий. Первое достоинство этого состоит в том, что копии хранятся в иерархической структуре, которой легче управлять. Второе достоинство – в том, что имена файлов намного короче. Третье достоинство состоит в том, что по именам каталогов 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`:

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

```
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 тот путь, который вы будете использовать.

# 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:

```
$ python3 backup_ver2.py
Каталог успешно создан E:\\Backup\\20080702
Резервная копия успешно создана в E:\\Backup\\20080702\\202311.zip

$ 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 другой стороны, когда их много, становится трудно отличить, какая копия для чего. Например, 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 достичь добавлением пользовательского комментария к имени zip-архива.

Примечание: Следующая программа не работает, так что не переживайте, просто проследуйте по ней, так как в ней содержится урок.

Сохраните как `backup_ver3.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 тот путь, который вы будете использовать.

# 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')
```

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 синтаксиса, что означает, что сценарий не удовлетворяет структуре, которую ожидает увидеть Python. Когда Python выдаёт сообщение об ошибке, он также указывает нам на место ошибки. Так что мы начинаем *отладку* программы с этой строки.
>
> При внимательном рассмотрении, мы видим, что одна логическая строка была разбита на две физические строки, но мы не указали, что эти две физические 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 исправление в нашу программу. Коррекция программы при обнаружении ошибок и называется **отладкой**[7].

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
```

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

```
# двойные кавычки внутри строки.

# 2. Резервные копии должны храниться в основном каталоге резерва.
target_dir = 'E:\\Backup' # Подставьте тот путь, который вы будете использовать.

# 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')
```

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, сделанные 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 создание 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 комментарии 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 добавить уровень *подробности*[8] 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.

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 полезное.

Попробуйте написать пятую версию с использованием модуля zipfile вместо вызова `os.system`.

Процесс разработки программного обеспечения

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

- Что (Анализ)
- Как (Проектирование)

[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*)

- Создание (Реализация)
- Тестирование (Тестирование и Отладка)
- Использование (Развёртывание и Оперирование)
- Поддержка (Усовершенствование)

Процедура, которую мы прошли при написании сценария создания резервных копий рекомендуется и для других программ: Проведите анализ и проектирование. Начните реализацию 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 собственную 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 манипулируют данными. Это называется *процедурно-ориентированным* стилем программирования. Существует и другой способ организации программ: объединять данные и функционал внутри некоего объекта. Это называется *объектно-ориентированной* парадигмой программирования. В большинстве случаев можно ограничиться процедурным программированием, а при написании большой программы или если решение конкретной задачи того требует, можно переходить к техникам объектно-ориентированного программирования.

Два основных аспекта объектно-ориентированного программирования – классы и объекты. **Класс** создаёт новый *тип*, а **объекты** являются *экземплярами* класса. Аналогично, когда мы говорим о «переменных типа `int`», это означает, что переменные, которые хранят 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].

Объекты могут хранить данные в обычных переменных, которые *принадлежат* объекту. Переменные, принадлежащие объекту или классу, называют **полями**. Объекты могут также обладать функционалом, т.е. иметь функции, *принадлежащие* классу. Такие функции 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 объекта класса или всему классу. Они называются **переменными экземпляра** и **переменными класса** соответственно.

Класс создаётся ключевым словом `class`. Поля и методы класса записываются в блоке

[1] boxing and unboxing

кода с отступом.

self

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

Хотя этому параметру можно дать любое имя, *настоятельно рекомендуется* использовать только имя `self`; использование любого другого имени не приветствуется. Есть много достоинств использования стандартного имени: во-первых, любой человек, просматривающий вашу программу, легко узнает его; во-вторых, некоторые специализированные Интегрированные 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 автоматически 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:

```
$ python3 simplestclass.py
<__main__.Person object at 0x019F85F0>
```

Как это работает:

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

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 оператора `pass`.

Далее мы создаём объект-экземпляр класса, записывая имя класса со скобками. (Мы узнаем больше о *реализации* 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 рассмотрим 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.

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. Обратите внимание на двойные подчёркивания в начале и в конце имени.

Пример: (сохраните как `class_init.py`)

```
class Person:
    def __init__(self, name):
        self.name = name
    def sayHi(self):
        print('Привет! Меня зовут', self.name)

p = Person('Swaroop')
p.sayHi()

# Этот короткий пример можно также записать как Person('Swaroop').sayHi()
```

Вывод:

```
$ python3 class_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`. Обратите 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.

Важно отметить, что при создании нового экземпляра класса мы не вызываем метод `__init__` явным образом, а передаём аргументы в скобках после имени этого класса. В этом и заключается специальная роль данного метода.

После этого мы получаем возможность использовать поле `self.name` в наших методах, что и продемонстрировано в методе `sayHi`.

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 обычными переменными, *заключёнными* в **пространствах имён** классов и объектов. Это означает, 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 различаются в зависимости от того, *принадлежит* ли переменная классу или объекту соответственно.

Переменные класса разделяемы – доступ к ним могут получать все экземпляры этого класса. Переменная класса существует только одна, поэтому когда любой из объектов изменяет 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`):

```
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(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("\nЗдесь роботы могут проделать какую-то работу.\n")

print("Роботы закончили свою работу. Давайте уничтожим их.")
del droid1
del droid2

Robot.howMany()
```

Вывод:

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

Как это работает:

Это длинный пример, но он помогает продемонстрировать природу переменных класса и объекта. Здесь `population` принадлежит классу `Robot`, и поэтому 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. Также 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 можем 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 :

```
@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 добавляем 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, равно 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 использования. В этом методе мы просто уменьшаем счётчик `Robot.population` на 1.

Метод `__del__` запускается лишь тогда, когда объект перестаёт использоваться, а поэтому 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 использоваться 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 заключается в **многократном использовании** одного и того же кода, и один из способов этого достичь – при помощи механизма *наследования*. Легче всего представить себе наследование 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 подтипам.

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 студентов, 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 обращаться к объекту преподавателя или студента можно как к объекту `SchoolMember`, что может быть полезно в ряде случаев, например, для подсчёта количества человек в школе. Когда 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 необходимости копировать его во всех классы, как пришлось бы в случае использования независимых классов.

Класс `SchoolMember` в этой ситуации называют *базовым классом* или *надклассом*[3]. Классы `Teacher` и `Student` называют *производными классами* или *подклассами*[4].

Рассмотрим теперь этот пример в виде программы (сохраните как `inherit.py`).

```
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):
```

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

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

```
        '''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)

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 указы-

ваем имена его базовых классов в виде кортежа, следующего за сразу за его именем. Далее мы видим, что метод `__init__` базового класса вызывается явно при помощи переменной `self`, чтобы инициализировать часть объекта, относящуюся к базовому классу. Это очень важно запомнить: Python не вызывает конструктор базового класса автоматически – его необходимо вызывать самостоятельно в явном виде.

Здесь же мы видим, как можно вызывать методы базового класса, предваряя запись имени метода именем класса, а затем передавая переменную `self` вместе 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 базовым классам по очереди, в порядке, в котором они перечислены в кортеже при определении класса.

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

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

Метаклассы [5]

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 метаклассы 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 «абстрактные базовые классы». Это означает, что такой класс *абстрактен*, т.е. является лишь некой концепцией, не предназначенной для использования в качестве реального класса.

[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.*)

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

```
#!/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))

    @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:

```
$ 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:

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 вывести 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. Возможность 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:

```
$ python3 user_input.py
Введите текст: сэр
Нет, это не палиндром

$ 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* при помощи кода "`seq[a:b]`", начиная с позиции `a` до позиции `b`. Но ведь мы также можем указать 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 обратном порядке.

Функция `input()` принимает строку в качестве аргумента и показывает её пользователю. Затем она ждёт, чтобы пользователь набрал что-нибудь и нажал клавишу ввода. Как только пользователь это сделал, функция `input()` возвращает введённый пользователем текст.

Мы считываем этот текст и выстраиваем его в обратном порядке. Если перевёрнутый и исходный текст одинаковы, значит введённый текст является палиндромом.

Домашнее задание

Проверка, является ли текст палиндромом должна также игнорировать знаки пунктуации, пробелы и регистр букв. Например, «А роза упала на лапу Азора» также является палиндромом, но наша текущая программа так не считает. Попробуйте улучить её так, чтобы она распознала этот палиндром.

Подсказка: (не читайте)

Воспользуйтесь кортежем (список *всех* знаков пунктуации можно найти здесь), содержащим все запрещённые символы, и примените тест на принадлежность, чтобы обнаружить 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], чтобы указать Python, что файл больше не используется.

[1] close – *англ.* «закрывать» (*прим.перев*)

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 указанием имени файла и режима, в котором мы хотим его открыть. Режим может быть для чтения (`'r'`), записи (`'w'`) или добавления (`'a'`)[2]. Можно также указать, в каком виде мы будем считывать, записывать или добавлять данные: в текстовом (`'t'`) или бинарном (`'b'`). На самом деле существует много других 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 используем метод `write` файлового объекта для записи в файл, после чего закрываем 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

[2] **r**ead, **w**rite и **a**ppend соответственно (*прим.перев.*)

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 символ 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 программа действительно записала текст в него и считала из него.

Pickle

Python предоставляет стандартный модуль с именем `pickle`[3], при помощи которого можно сохранять **любой** объект Python в файле, а затем извлекать его обратно. Это называется *длительным* хранением объекта.

Пример: (сохраните как `pickling.py`):

```
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

# 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:

[3] pickle – *англ.* «мариновать», «солить» (*прим.перев.*)

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 «расконсервацией» («unpickling»).

Резюме

Мы обсудили разные типы ввода/вывода, а также работу с файлами и использование модуля `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 если файл был случайно удалён, пока программа работала? Такие ситуации обрабатываются 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* синтаксическую 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 `Ctrl-D` и посмотрите, что произойдёт.

```
>>> s = input('Введите что-нибудь --> ')
Введите что-нибудь -->
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]. 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 -->     # Нажмите ctrl-c
Вы отменили операцию.

$ python3 try_except.py
Введите что-нибудь --> без ошибок
Вы ввели без ошибок
```

Как это работает:

> Здесь мы поместили все команды, которые могут вызвать исключения/ошибки, внутрь блока `try`, а затем поместили обработчики соответствующих ошибок/исключений в блок `except`. Выражение `except` может обрабатывать как одиночную ошибку или исключение, так и список ошибок/исключений в скобках. Если не указано имя ошибки или исключения, обрабатываться будут *все* ошибки и исключения.

[1] try – *англ.* «пытаться» (*прим.перев.*)

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 умолчанию, 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 дальнейшей работы с ним.

16.4 Вызов исключения

Исключение можно *поднять* при помощи оператора `raise`[2], передав ему имя ошибки/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`)

```
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))
else:
    print('Не было исключений.')
```

Вывод:

```
$ python3 raising.py
Введите что-нибудь --> a
```

[2] raise – *англ.* “поднимать” (*прим.перев.*)

```
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 исключения называется `ShortInputException`. Он содержит два поля: `length`, хранящее длину введённого текста, и `atleast`, указывающее, какую минимальную длину текста ожидала программа.

В пункте `except` мы указываем класс ошибки `ShortInputException`, который будет сохранён как[3] переменная `ex`, содержащая соответствующий объект ошибки/исключения. Это аналогично параметрам и аргументам при вызове функции. Внутри этого пункта `except` мы используем поля `length` и `atleast` объекта исключения для вывода необходимых сообщений пользователю.

Try .. Finally

Представим, что программе происходит чтение файла и необходимо убедиться, что объект файла был корректно закрыт и что не возникло никакого исключения. Этого можно достичь с применением блока `finally`.

Сохраните как `finally.py`:

```
import time

try:
    f = open('poem.txt')
    while True: # наш обычный способ читать файлы
        line = f.readline()
        if len(line) == 0:
            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:

[3] as – *англ.* «как» (*прим.перев.*)

```
$ 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 случае 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 освобождением этого ресурса в блоке `finally`. Для того, чтобы сделать это более «чисто», существует оператор `with`[4]:

Сохраните как `using_with.py`:

```
with open("poem.txt") as f:
    for line in f:
        print(line, end='')
```

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, используемый оператором `with`. Он считывает объект, возвращаемый оператором `open`. Назовём его в данном случае «thefile».

Перед запуском блока кода, содержащегося в нём, оператор `with` *всегда* вызывает 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`.

[4] with – *англ.* «с» (*прим.перев.*)

Более обширное рассмотрение этой темы выходит за рамки настоящей книги, поэтому для более исчерпывающего объяснения см. **PEP 343**.

16.7 Резюме

Мы обсудили использование операторов `try..except` и `try..finally`. Мы также увидели, 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 является 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. Детальное описание всех модулей стандартной библиотеки Python можно найти в разделе «Library Reference» документации, входящей в комплект поставки Python.

Давайте изучим несколько полезных модулей.

Примечание

Если темы в настоящей главе покажутся вам слишком сложными, вы можете её пропустить. Однако я настоятельно рекомендую вернуться к этой главе, когда вы будете чувствовать 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` предоставляет 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 называется `warnings` и служит для отображения предупреждений пользователю. Если 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`:

```
import os, platform, logging

if platform.platform().startswith('Windows'):
    logging_file = os.path.join(os.getenv('HOMEDRIVE'), \
                                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 взаимодействия с операционной системой, модуль `platform` для получения информации о платформе (т.е. операционной системе) и модуль `logging` для сохранения лога[1].

Прежде всего, при помощи строки, возвращаемой функцией `platform.platform()` мы проверяем, какая операционная система используется (для более подробной информации см. `import platform; help(platform)`). Если это Windows, то мы определяем диск, содержащий домашний каталог, путь к домашнему каталогу на нём и имя файла, в котором хотим сохранить информацию. 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.

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 формате, ожидаемом операционной системой.

Далее мы конфигурируем модуль `logging` таким образом, чтобы он записывал все сообщения в определённом формате в указанный файл.

Наконец, мы можем выводить сообщения, предназначенные для отладки, информирования, предупреждения и даже критические сообщения. После выполнения программы можно просмотреть этот файл и узнать, что происхо-

[1] log – *англ.* «журнал», «вести журнал» (*прим.перев.*)

дило в программе, хотя пользователю, запустившему программу, ничего не было показано.

Серия «Модуль недели»

В стандартной библиотеке можно найти ещё много полезного. Например, отладка, обработка параметров командной строки, регулярные выражения и так далее.

Лучший способ дальнейшего изучения стандартной библиотеки – читать замечательную серию Дуга Хелмана «Модуль недели» или официальную документацию Python.

Резюме

Мы изучили лишь некоторые возможности некоторых модулей стандартной библиотеки Python. Я настоятельно рекомендую просмотреть документацию по стандартной библиотеке Python, 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 использовать 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 поставить 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 методы можно посмотреть в документации.

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 собственным уровнем отступа. Однако, существует и исключение. Если блок команд содержит 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 отдельным 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 возвращается 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 параметр `key`, определяющий способ сортировки списка (обычно мы думаем только 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 отдельный 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. Представьте, 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 больше 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 необходимо 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 сокращает объёмы стандартного кода, необходимого для циклической обработки каждого элемента списка и сохранения его в новом списке.

Передача кортежей и словарей в функции

Для получения параметров, переданных функции, в виде кортежа или словаря, существуют специальные приставки "*" или "**" соответственно. Это особенно полезно в случаях, 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 аргументы, переданные функции, сохранятся в `args` в виде кортежа. В случае использования 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 файле, 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, содержащиеся в строке. Вот простой пример.

```
>>> eval('2*3')
6
```

Оператор assert

Оператор `assert` существует для того, чтобы указать, что нечто является истиной. Например, если требуется гарантировать, что в списке будет хотя бы один элемент, и вызвать ошибку, если это не так, то оператор `assert` идеально подойдёт для такой задачи. Когда заявленное выражение ложно, вызывается ошибка `AssertionError`.

```
>>> 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 случаев гораздо лучше «отлавливать» исключения и либо решать соответствующую проблему 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 *управляющей 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 использованием двойных кавычек. Аналогично следует использовать управляющую последовательность для вставки двойной кавычки в строку, ограниченную двойными кавычками. 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*. Но есть и другой – использовать управляющую последовательность для символа перевода строки \n. Например: "Это первая строка\nЭто вторая строка". Полезно знать ещё одну управляющую последовательность – 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]. 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'.

---

[1] "r" – *от англ.* «raw» – «сырой, необработанный» (*прим. перев.*)

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.