

# **Python 3** **for beginners** **with the** **speed of light**

## Содержание

1	Возможности языка Python3	1	.....	00
				3
2	Загрузка и установка Python	2	.....	00
				3
3	Знакомство со средой разработки <b>IDLE</b>	3	.....	00
				8
4	Синтаксис	4	.....	01
				0
5	Почему моя программа не работает?	5	.....	01
				1
6	Условный оператор <b>if</b>	6	.....	01
				7
7	Циклы	7	.....	01
				9
8	Ключевые слова	8	.....	02
				1
9	Встроенные функции	9	.....	02
				3
1	Числа	10	.....	02
0				7
1	Строки (форматирование)	11	.....	03
1				4
1	Списки (массивы)	12	.....	04
2				7
1	Индексы и срезы	13	.....	05
3				0
1	Кортежи	14	.....	05
4				2
1	Словари	15	.....	05
5				4
1	Множества	16	.....	05
6				7
1	Функции	17	.....	05
7				9
1	Исключения и их обработка	18	.....	06
8				2
1	Байтовые строки	19	.....	06
9				7
2	Файлы	20	.....	06

0			8
2	<b>With ... as</b> — менеджеры контекста	21	07
1			1
2	<b>PEP 8</b> — руководство по написанию кода на	22	07
2	Python		3
2	Документирование кода	23	09
3			6
2	Создание и подключение модулей	24	09
4			9
2	Объектно-ориентированное		
5	программирование. Общее представление.	25	10
			3
2	Инкапсуляция, наследование, полиморфизм	26	10
6			5
2	Перегрузка операторов	27	10
7			7
2	Декораторы	28	11
8			4
2	Python2 vs Python3: различия синтаксиса	29	12
9			6
3	Введение в Python с PyCharm Educational	30	13
0	Edition		3
3	Компиляция программы на python 3 в exe с		
1	помощью программы cx_Freeze	31	14
			3
3	NumPy: начало работы	32	14
2			7
3	Что нового в Python 3,3?	33	16
3			7
3	Вышел Python 3.4.0	34	17
4			6
3	pythondigest.ru - самые свежие новости из мира		
5	Python	35	17
			7
3	Модуль fractions	36	17
6			8
3	Модуль cmath	37	18
7			0
3	Модуль glob	38	18
8			1
3	Модуль сору - поверхностное и глубокое		

9	копирование объектов	39	.....	18
				2
4	Модуль functools	40	.....	18
0				4
4	Модуль os.path	41	.....	18
1				8
4	Модуль json	42	.....	19
2				0
4	Модуль calendar	43	.....	19
3				4
4	Модуль os	44	.....	19
4				7
4	Модуль pickle	45	.....	20
5				0
4	Модуль datetime	46	.....	20
6				1
4	Модуль bisect	47	.....	20
7				4
4	Модуль collections	48	.....	20
8				5
4	Модуль array. Массивы в python	49	.....	20
9				9
5	Модуль itertools	50	.....	21
0				1
5	Модуль time	51	.....	21
1				4
5	Модуль sys	52	.....	21
2				6
5	Модуль random	53	.....	21
3				9
5	Модуль math	54	.....	22
4				1
5	Интерпретатор hq9+	55	.....	22
5				3
5	Задача про словарь	56	.....	22
6				5
5	Пишем блэкджек	57	.....	22
7				7
5	Интерпретатор brainfuck	58	.....	22
8				8

## Возможности языка Python3

Вот лишь некоторые вещи, которые умеет делать python:

- Работа с xml/html файлами
- Работа с http запросами
- GUI (графический интерфейс)
- Создание веб-сценариев
- Работа с FTP
- Работа с изображениями, аудио и видео файлами
- Робототехника
- Программирование математических и научных вычислений

И многое, многое другое...

Таким образом, python подходит для решения львиной доли повседневных задач, будь то резервное копирование, чтение электронной почты, либо же какая-нибудь игрушка. Язык программирования Python практически ничем не ограничен, поэтому также может использоваться в крупных проектах. К примеру, python интенсивно применяется IT-гигантами, такими, как, например, Google и Yandex. К тому же простота и универсальность python делают его одним из лучших языков программирования.

## Загрузка и установка Python

### Установка Python на Windows

Скачивать python будем с [официального сайта](#). Кстати, не рекомендую скачивать интерпретатор python с других сайтов или через торрент, в них могут быть вирусы. Программа бесплатная. Заходим

на <https://python.org/downloads/windows/>, выбираем "latest python release" и **python 3**.

На python 2 могут не работать некоторые мои примеры программ.

На момент написания материала это python 3.4.1.



Появляется страница с описанием данной версии Python (на английском). Если интересно - можете почитать. Затем крутим в самый низ страницы, а затем открываем "download page".

## More resources

- [Change log for this release.](#)
- [Online Documentation](#)
- [What's new in 3.4?](#)
- [3.4 Release Schedule](#)
- Report bugs at <http://bugs.python.org>.
- [Help fund Python and its community.](#)

## Download

Please proceed to the [download page](#) for the download.

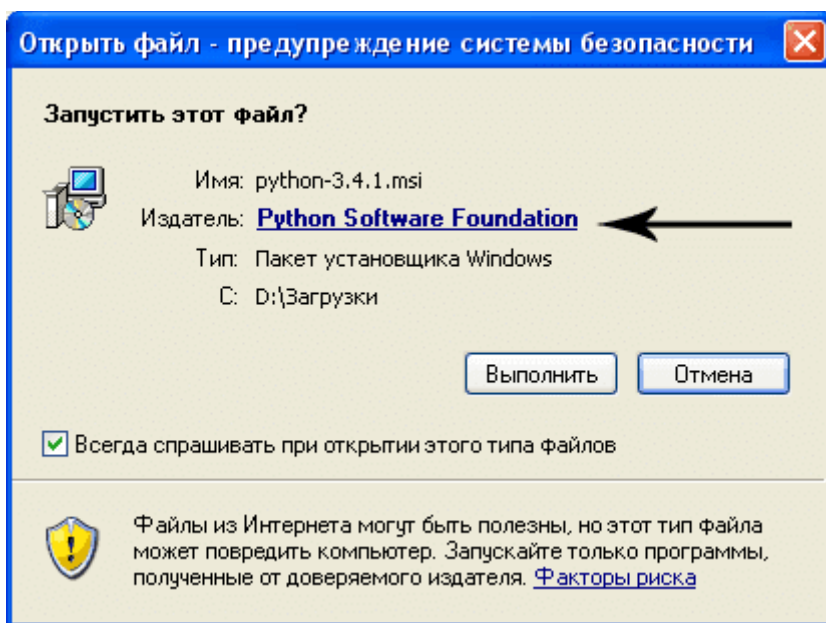
Notes on this release:

- The binaries for AMD64 will also work on processors that implement the Intel 64 architecture. (Also known as the "x64" architecture, and formerly known as both "EM64T" and "x86-64".) They will not work on Intel Itanium Processors (formerly "IA-64").
- There is [important information about IDLE, Tkinter, and Tcl/Tk on Mac OS X here.](#)

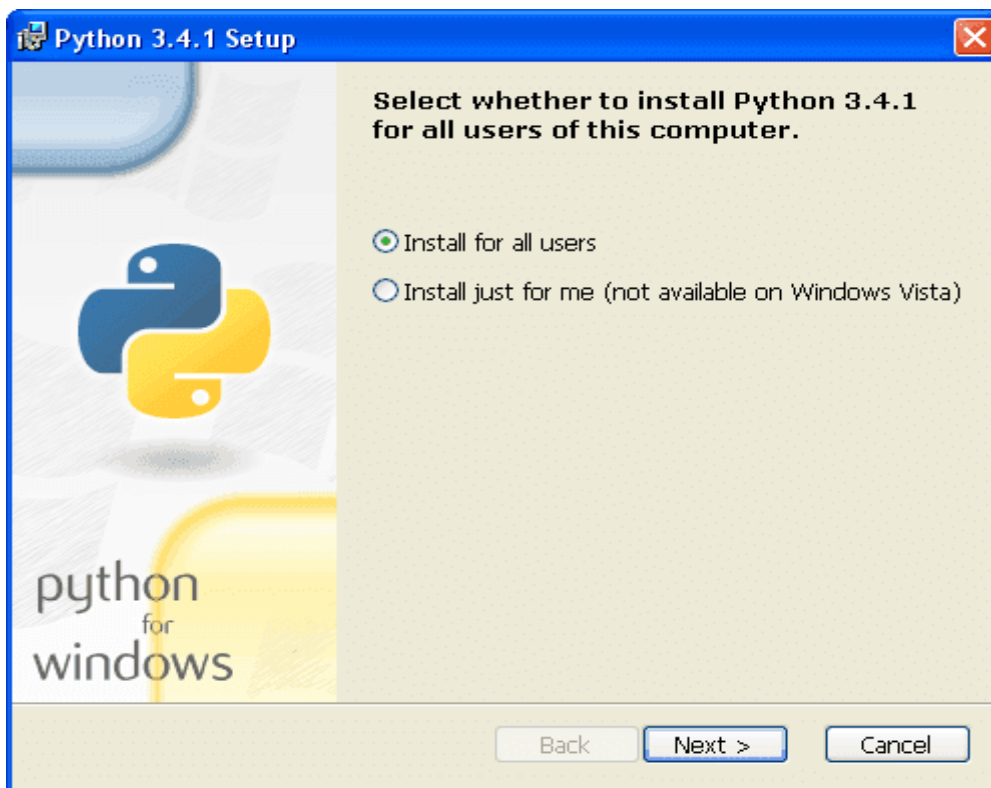
Вы увидите список файлов, которые можно загрузить. Нам нужен Windows x86 MSI installer (если система 32-х битная), или Windows x86-64 MSI installer (если система 64-х битная). Больше из файлов нам ничего не нужно.

Version	Operating System	Description	Date	MD5 Sum	File Size
<a href="#">Mac OS X 64-bit/32-bit installer</a>	Mac OS X	for Mac OS X 10.6 and later		316a2f83edff73bbcb2c84390bee2db	22776248
<a href="#">Mac OS X 32-bit i386/PPC installer</a>	Mac OS X	for Mac OS X 10.5 and later		534f8ec2f5ad5539f9165b3125b5e959	22692757
<a href="#">XZ compressed source tarball</a>	Source release			6cafc183b4106476dd73d5738d7f616a	14125788
<a href="#">Gzipped source tarball</a>	Source release			26695450087f8587b26d0b6a63844af5	19113124
<a href="#">Windows debug information files</a>	Windows			9ce29e8356cf13f88e41f7595c2d7399	36744364
<a href="#">Windows x86 MSI installer</a>	Windows			4940c3fad01ffa2ca7f9cc43a005b89a	24408064
<a href="#">Windows debug information files for 64-bit binaries</a>	Windows			44a2d4d3c62a147f5a9f733b030490d1	24129218
<a href="#">Windows help file</a>	Windows			6ff47ff938b15d2900f3c7311ab629e5	7297786
<a href="#">Windows x86-64 MSI installer</a>	Windows	for AMD64/EM64T/x64, not Itanium processors		25440653f27ee1597fd6b3e15eee155f	25104384

Ждём, пока python загрузится. Затем открываем загрузившийся файл. Файл подписан *Python Software Foundation*, значит, все в порядке.

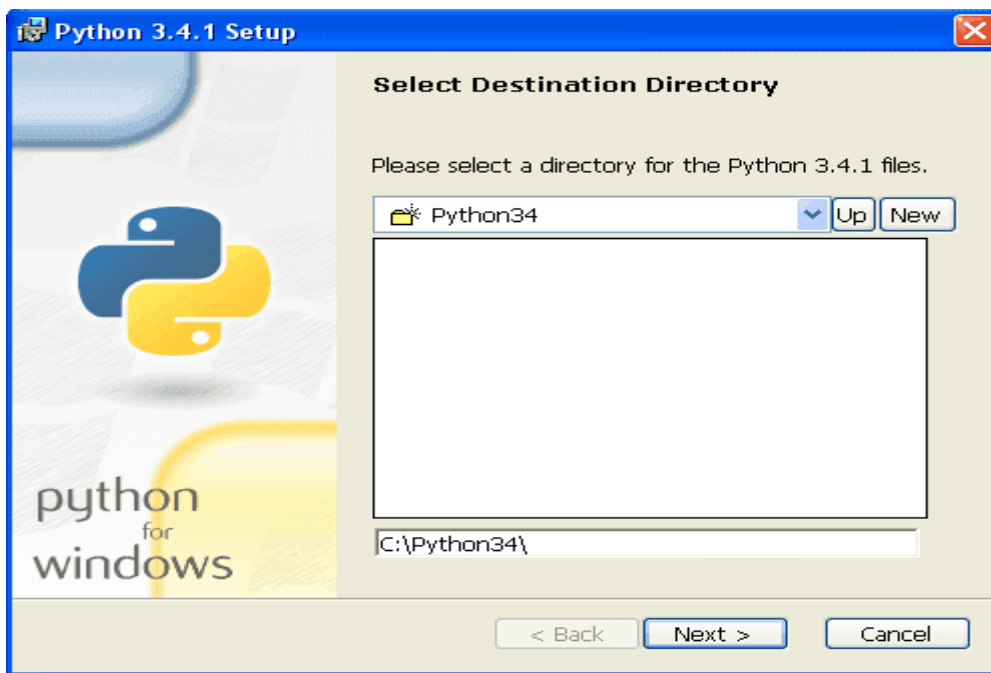


Устанавливаем для всех пользователей или только для одного (на ваше усмотрение).



Выбираем папку для установки. Я оставляю папку по умолчанию. Вы можете выбрать любую папку на своем диске.





Выбираем компоненты, которые будут установлены. Оставьте компоненты по умолчанию, если не уверены.



Ждем установки python...

Finish. Поздравляю, вы установили Python! Также в установщик python для windows встроена среда разработки IDLE. Прямо сейчас вы можете написать свою первую программу!

## Установка Python на linux системы (ubuntu, linux mint и другие)

Откройте консоль (обычно ctrl+alt+t). Введите в консоли:

```
python3
```

Скорее всего, вас любезно поприветствует python 3:

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Если это так, то можно вас поздравить: у вас уже стоит python 3. В противном случае нужно установить пакет **\*python3\***:

```
sudo apt-get install python3
```

Либо через mintinstaller / synaptic / центр приложений ubuntu / что вам больше нравится.

В python для linux нет предустановленной среды IDLE. Если хотите, её можно установить отдельно. Пакет называется **\*python3-idle\***.

Однако, её установка не является обязательной. Вы можете писать в своём любимом текстовом редакторе (gedit, vim, emacs...) и запускать программы через консоль:

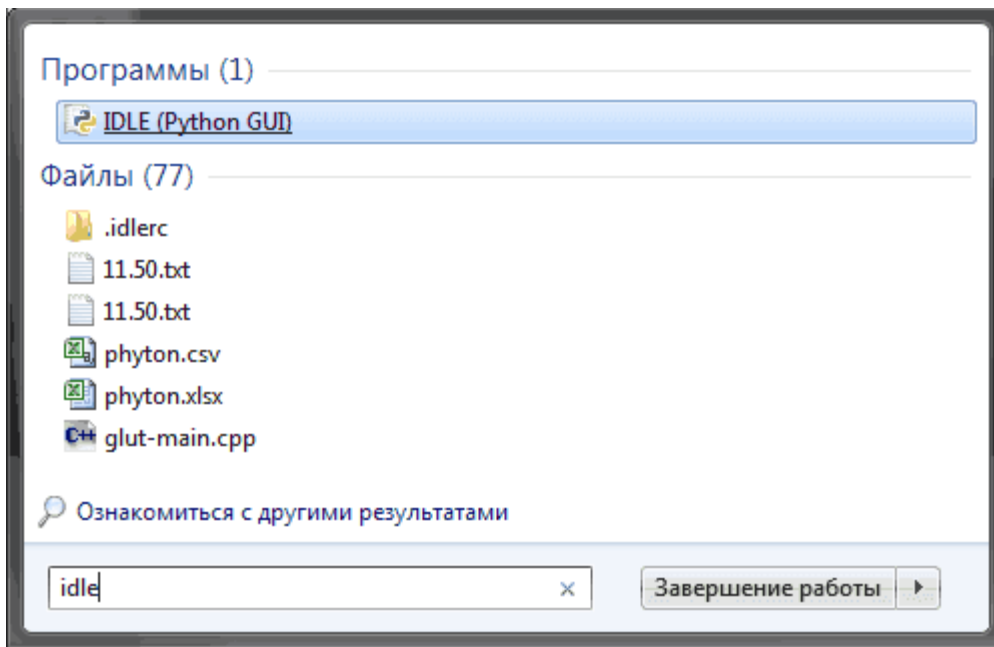
```
python3 path_to_file.py
```

Теперь вы можете написать первую программу (хотите, пишите в IDLE, хотите - в своём любимом текстовом редакторе).

## Знакомство со средой разработки IDLE

После загрузки и установки python открываем IDLE (среда разработки на языке Python, поставляемая вместе с дистрибутивом).

Здесь и далее буду приводить примеры под ОС Windows.

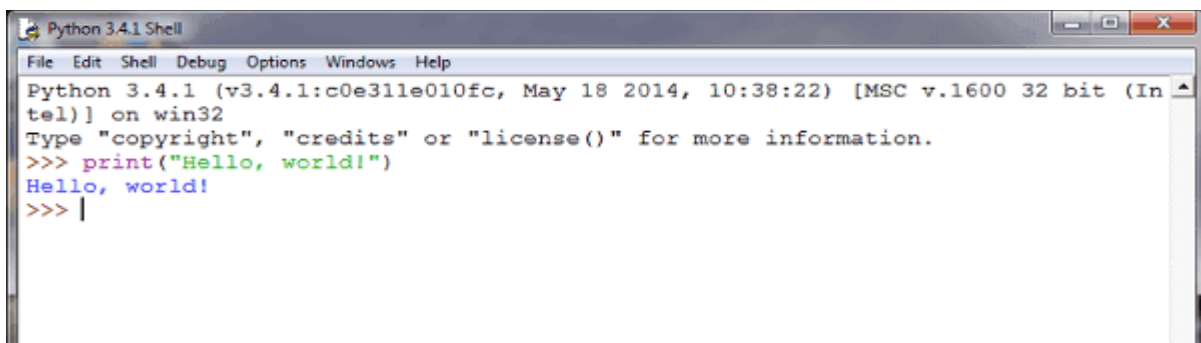


Запускаем IDLE (изначально запускается в интерактивном режиме), после чего уже можно начинать писать первую программу. Традиционно, первой программой у нас будет "hello world".

Чтобы написать "hello world" на python, достаточно всего одной строки:

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

Вводим этот код в IDLE и нажимаем Enter. Результат виден на картинке:



Поздравляю! Вы написали свою **первую программу на python!**

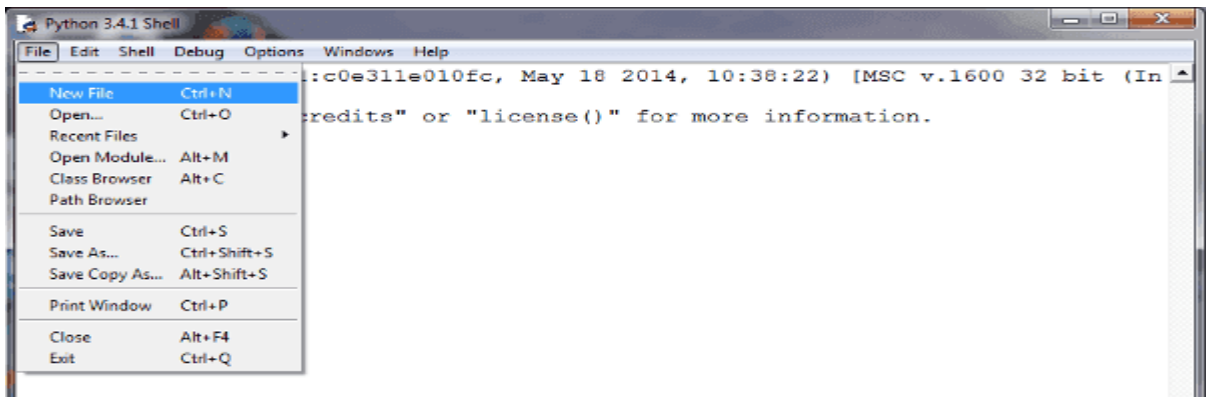
С интерактивным режимом мы немного познакомились, можете с ним ещё поиграться, например, написать

```
print(3 + 4)
```

```
print(3 * 5)
print(3 ** 2)
```

Но, всё-таки, интерактивный режим не будет являться основным. В основном, вы будете сохранять программный код в файл и запускать уже файл.

Для того, чтобы создать новое окно, в интерактивном режиме IDLE выберите File → New File (или нажмите Ctrl + N).



В открывшемся окне введите следующий код:

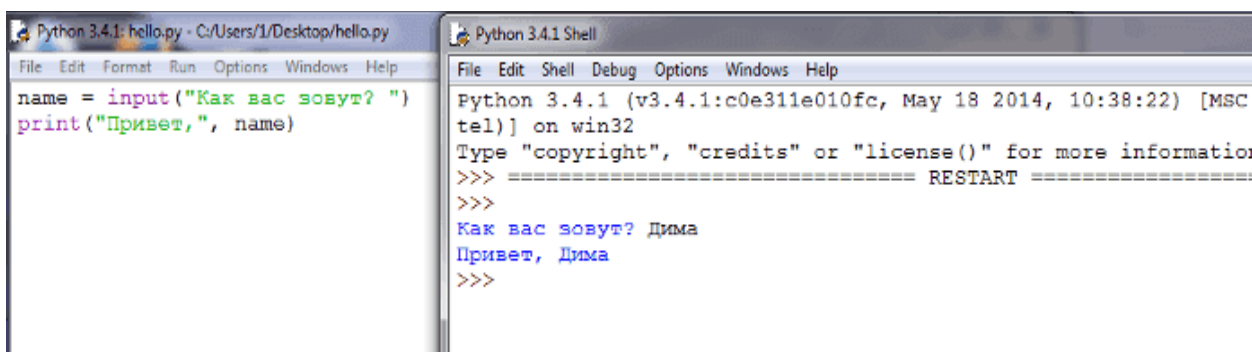
```
name = input("Как Вас зовут? ")
print("Привет, ", name)
```

Первая строка печатает вопрос ("Как Вас зовут? "), ожидает, пока вы не напечатаете что-нибудь и не нажмёте Enter и сохраняет введённое значение в переменной name.

Во второй строке мы используем функцию print для вывода текста на экран, в данном случае для вывода "Привет, " и того, что хранится в переменной "name".

Теперь нажмём F5 (или выберем в меню IDLE Run → Run Module и убедимся, что то, что мы написали, работает. Перед запуском IDLE предложит нам сохранить файл. Сохраним туда, куда вам будет удобно, после чего программа запустится.

Вы должны увидеть что-то наподобие этого (на скриншоте слева - файл с написанной вами программой, справа - результат её работы):



Поздравляю! Вы научились писать простейшие программы, а также познакомились со средой разработки IDLE.

## Синтаксис

- Конец строки является концом инструкции (точка с запятой не требуется).
- Вложенные инструкции объединяются в блоки по величине отступов. Отступ может быть любым, главное, чтобы в пределах одного вложенного блока отступ был одинаков. И про читаемость кода не забывайте. Отступ в 1 пробел, к примеру, не лучшее решение. Используйте 4 пробела (или знак табуляции, на худой конец).
- Вложенные инструкции в Python записываются в соответствии с одним и тем же шаблоном, когда основная инструкция завершается двоеточием, вслед за которым располагается вложенный блок кода, обычно с отступом под строкой основной инструкции.
- Основная инструкция:
- Вложенный блок инструкций

### Несколько специальных случаев

- Иногда возможно записать несколько инструкций в одной строке, разделяя их точкой с запятой:
- `a = 1; b = 2; print(a, b)`

Но не делайте это слишком часто! Помните об удобочитаемости. А лучше вообще так не делайте.

- Допустимо записывать одну инструкцию в нескольких строках. Достаточно ее заключить в пару круглых, квадратных или фигурных скобок:
- ```
if (a == 1 and b == 2 and
    c == 3 and d == 4): # Не забываем про двоеточие
    print('spam' * 3)
```
- Тело составной инструкции может располагаться в той же строке, что и тело основной, если тело составной инструкции не содержит составных инструкций. Ну я думаю, вы поняли :). Давайте лучше пример приведу:
- ```
if x > y: print(x)
```

## Почему моя программа не работает?

Моя программа не работает! Что делать? В данном материале я постараюсь собрать наиболее частые ошибки начинающих программировать на python 3, а также расскажу, как их исправлять.

**Проблема:** Моя программа не запускается. На доли секунды появляется чёрное окошко, а затем исчезает.

**Причина:** после окончания выполнения программы (после выполнения всего кода или при возникновении исключения программа закрывается. И если вы её вызвали двойным кликом по иконке (а вы, скорее всего, вызвали её именно так), то она закроется вместе с окошком, в котором находится вывод программы.

**Решение:** запускать программу через IDLE или через консоль.

**Проблема:** Не работает функция input. Пишет SyntaxError.

**Пример кода:**

```
>>> a = input()
hello world
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    hello world
    ^
SyntaxError: unexpected EOF while parsing
```

**Причина:** Вы запустили Python 2.

**Решение:** Установить Python 3.

**Проблема:** Где-то увидел простую программу, а она не работает.

**Пример кода:**

```
name = raw_input()
print name
```

**Ошибка:**

```
File "a.py", line 3
    print name
          ^
SyntaxError: invalid syntax
```

**Причина:** Вам подсунули программу на Python 2.

**Решение:** Прочитать об отличиях Python 2 от Python 3. Переписать её на Python 3. Например, данная программа на Python 3 будет выглядеть так:

```
name = input()
print(name)
```

**Проблема:** TypeError: Can't convert 'int' object to str implicitly.

**Пример кода:**

```
>>> a = input() + 5
8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

**Причина:** Нельзя складывать строку с числом.

**Решение:** Привести строку к числу с помощью функции `int()`. Кстати, заметьте, что функция `input()` всегда возвращает строку!

```
>>> a = int(input()) + 5
8
>>> a
13
```

**Проблема:** `SyntaxError: invalid syntax.`

**Пример кода:**

```
a = 5
if a == 5
    print('Ура!')
```

**Ошибка:**

```
File "a.py", line 3
    if a == 5
           ^
SyntaxError: invalid syntax
```

**Причина:** Забыто двоеточие.

**Решение:**

```
a = 5
if a == 5:
    print('Ура!')
```

**Проблема:** `SyntaxError: invalid syntax.`

**Пример кода:**

```
a = 5
if a = 5:
    print('Ура!')
```

**Ошибка:**

```
File "a.py", line 3
    if a = 5
```



^

**SyntaxError:** invalid syntax

**Причина:** Забыто равно.

**Решение:**

```
a = 5
if a == 5:
    print('Ура!')
```

**Проблема:** NameError: name 'a' is not defined.

**Пример кода:**

```
print(a)
```

**Причина:** Переменная "a" не существует. Возможно, вы опечатались в названии или забыли инициализировать её.

**Решение:** Исправить опечатку.

```
a = 10
print(a)
```

**Проблема:** IndentationError: expected an indented block.

**Пример кода:**

```
a = 10
if a > 0:
print(a)
```

**Причина:** Нужен отступ.

**Решение:**

```
a = 10
if a > 0:
    print(a)
```

**Проблема:** TabError: inconsistent use of tabs and spaces in indentation.

**Пример кода:**

```
a = 10
if a > 0:
    print(a)
    print('Ура!')
```

**Ошибка:**

```
File "a.py", line 5
    print('Ура!')
          ^
TabError: inconsistent use of tabs and spaces in
indentation
```

**Причина:** Смешение пробелов и табуляций в отступах.

**Решение:** Исправить отступы.

```
a = 10
if a > 0:
    print(a)
    print('Ура!')
```

**Проблема:** UnboundLocalError: local variable 'a' referenced before assignment.

**Пример кода:**

```
def f():
    a += 1
    print(a)

a = 10
f()
```

**Ошибка:**

```
Traceback (most recent call last):
  File "a.py", line 7, in <module>
    f()
  File "a.py", line 3, in f
    a += 1
```

**UnboundLocalError:** local variable 'a' referenced before assignment

**Причина:** Попытка обратиться к локальной переменной, которая ещё не создана.

**Решение:**

```
def f():
    global a
    a += 1
    print(a)

a = 10
f()
```

**Проблема:** Программа выполнялась, но в файл ничего не записалось / записалось не всё.

**Пример кода:**

```
>>> f = open('output.txt', 'w', encoding='utf-8')
>>> f.write('bla')
3
>>>
```

**Причина:** Не закрыт файл, часть данных могла остаться в буфере.

**Решение:**

```
>>> f = open('output.txt', 'w', encoding='utf-8')
>>> f.write('bla')
3
>>> f.close()
>>>
```

## Условный оператор if

**Условная инструкция if-elif-else** (её ещё иногда называют оператором ветвления) - основной инструмент выбора в Python. Проще говоря, она выбирает, какое действие следует выполнить, в зависимости от значения переменных в момент проверки условия.

### Синтаксис инструкции if

Сначала записывается часть if с условным выражением, далее могут следовать одна или более необязательных частей elif, и, наконец, необязательная часть else. Общая форма записи условной инструкции if выглядит следующим образом:

```
if test1:
    state1
elif test2:
    state2
else:
    state3
```

Простой пример (напечатает 'true', так как 1 - истина):

```
if 1:
    print('true')
else:
    print('false')

true
```

Чуть более сложный пример (его результат будет зависеть от того, что ввёл пользователь):

```
a = int(input())
if a < -5:
```

```
    print('Low')
elif -5 <= a <= 5:
    print('Mid')
else:
    print('High')
```

Конструкция с несколькими elif может также служить отличной заменой конструкции switch - case в других языках программирования.

## Проверка истинности в Python

- Любое число, не равное 0, или непустой объект - истина.
- Числа, равные 0, пустые объекты и значение None - ложь
- Операции сравнения применяются к структурам данных рекурсивно
- Операции сравнения возвращают True или False
- Логические операторы and и or возвращают истинный или ложный объект-операнд

Логические операторы:

X and Y

Истина, если оба значения X и Y истинны.

X or Y

Истина, если хотя бы одно из значений X или Y истинно.

not X

Истина, если X ложно.

## Трехместное выражение if/else

Следующая инструкция:

```
if X:
    A = Y
else:
    A = Z
```

довольно короткая, но, тем не менее, занимает целых 4 строки. Специально для таких случаев и было придумано выражение if/else:

```
A = Y if X else Z
```

В данной инструкции интерпретатор выполнит выражение Y, если X истинно, в противном случае выполнится выражение Z.

```
>>> A = 't' if 'spam' else 'f'
>>> A
't'
```

## Циклы

### Цикл while

While - один из самых универсальных циклов в Python, поэтому довольно медленный. Выполняет тело цикла до тех пор, пока условие цикла истинно.

```
i = 5
while i < 15:
    print(i)
    i = i + 2
```

```
5
7
9
11
13
```

### Цикл for

Цикл for уже чуточку сложнее, чуть менее универсальный, но выполняется гораздо быстрее цикла while. Этот цикл проходится по любому итерируемому объекту (строке, списку,...), и во время каждого прохода выполняет тело цикла.

```
for i in 'hello world':
    print(i * 2, end='')
```

```
hheellllloo  wwoorrlldd
```

### Оператор continue

Оператор continue начинает следующий проход цикла, минуя оставшееся тело цикла (for или while)

```
>>> for i in 'hello world':  
    if i == 'o':  
        continue  
    print(i * 2, end='')
```

```
hheel1111  wwrr11dd
```

## Оператор break

Оператор break досрочно прерывает цикл.

```
for i in 'hello world':  
    if i == 'o':  
        break  
    print(i * 2, end='')
```

```
hheel1111
```

## Волшебное слово else

Слово else, примененное в цикле for или while, проверяет, был ли произведен выход из цикла инструкцией break, или же "естественным" образом. Блок инструкций внутри else выполнится только в том случае, если выход из цикла произошел без помощи break.

```
for i in 'hello world':  
    if i == 'a':  
        break  
else:  
    print('Буквы а в строке нету')
```

```
Буквы а в строке нету
```

## Ключевые слова, встроенные функции

### Ключевые слова

**False** - ложь.

**True** - правда.

**None** - "пустой" объект.

**and** - логическое И.

**with / as** – менеджер контекста.

**assert** условие - возбуждает исключение, если условие ложно.

**break** - выход из цикла.

**class** – пользовательский тип, состоящий из методов и атрибутов.

**continue** - переход на следующую итерацию цикла.

**def** - определение функции.

**del** - удаление объекта.

**elif** - в противном случае, если.

**else** - см. for/else или if/else.

**except** - перехватить исключение.

**finally** - вкупе с инструкцией try, выполняет инструкции независимо от того, было ли исключение или нет.

**for** - цикл for.

**from** - импорт нескольких функций из модуля.

**global** - позволяет сделать значение переменной, присвоенное ей внутри функции, доступным и за пределами этой функции.

**if** - если.

**import** - импорт модуля.

**in** - проверка на вхождение.

**is** - ссылаются ли 2 объекта на одно и то же место в памяти.

**lambda** - определение анонимной функции.



**nonlocal** - позволяет сделать значение переменной, присвоенное ей внутри функции, доступным в объемлющей инструкции.

**not** - логическое НЕ.

**or** - логическое ИЛИ.

**pass** - ничего не делающая конструкция.

**raise** - возбудить исключение.

**return** - вернуть результат.

**try** - выполнить инструкции, перехватывая исключения.

**while** - цикл while.

**yield** - определение функции-генератора.

## Модуль keyword

В общем-то, keyword - не такой уж и модуль, но все же.

**keyword.kwlist** - список всех доступных ключевых слов (почему не кортеж, непонятно).

**keyword.iskeyword(строка)** - является ли строка ключевым словом.

## Встроенные функции

### Краткий обзор встроенных функций в Python 3

### Встроенные функции, выполняющие преобразование типов

**bool(x)** - преобразование к типу bool, использующая стандартную процедуру проверки истинности. Если x является ложным или опущен, возвращает значение False, в противном случае она возвращает True.

**bytearray([источник [, кодировка [ошибки]])** - преобразование к bytearray. Bytearray - изменяемая последовательность целых чисел в диапазоне  $0 \leq X < 256$ . Вызванная без аргументов, возвращает пустой массив байт.

**bytes**([источник [, кодировка [ошибки]]) - возвращает объект типа bytes, который является неизменяемой последовательностью целых чисел в диапазоне  $0 \leq X < 256$ . Аргументы конструктора интерпретируются как для bytearray().

**complex**([real[, imag]]) - преобразование к комплексному числу.

**dict**([object]) - преобразование к словарю.

**float**([X]) - преобразование к числу с плавающей точкой. Если аргумент не указан, возвращается 0.0.

**frozenset**([последовательность]) - возвращает неизменяемое множество.

**int**([object], [основание системы счисления]) - преобразование к целому числу.

**list**([object]) - создает список.

**memoryview**([object]) - создает объект memoryview.

**object**() - возвращает безликий объект, являющийся базовым для всех объектов.

**range**([start=0], stop, [step=1]) - арифметическая прогрессия от start до stop с шагом step.

**set**([object]) - создает множество.

**slice**([start=0], stop, [step=1]) - объект среза от start до stop с шагом step.

**str**([object], [кодировка], [ошибки]) - строковое представление объекта. Использует метод `__str__`.

**tuple**(obj) - преобразование к кортежу.

**Встроенные функции, использующие методы класса  
(логика может отличаться в зависимости от типа  
данных)**

Функция	Используемый метод класса	Основное предназначение
<b>abs(X)</b>	<code>__abs__</code>	Модуль (абсолютное значение) X.
<b>len(X)</b>	<code>__len__</code>	Число элементов объекта
<b>hash(object)</b>	<code>__hash__</code>	Хэш-сумма (если есть)

<b>iter</b> (object)	<code>__iter__</code>	Возвращает объект-итератор
<b>next</b> (iterator)	<code>__next__</code>	Следующий элемент итератора
<b>format</b> (value[,format_spec])	<code>__format__</code>	Форматирование (обычно форматирование строки)
<b>pow</b> (x, y, [r])	<code>__pow__</code>	( x в степени y ) % r
<b>reversed</b> (object)	<code>__reversed__</code>	Итератор из развернутого объекта
<b>repr</b> (obj)	<code>__repr__</code>	Представление объекта

### **Встроенные функции, использующиеся в качестве декораторов**

**classmethod**(function) - возвращает метод класса для функции.

**property**(fget=None, fset=None, fdel=None, doc=None)

**staticmethod**(function) - статический метод для функции.

### **Встроенные функции, для перевода между системами счисления**

**bin**(x) - преобразование целого числа в двоичную строку.

**hex**(x) - преобразование целого числа в шестнадцатеричную строку.

**oct**(x) - преобразование целого числа в восьмеричную строку.

### **Другие встроенные функции**

**all**(последовательность) - Возвращает True, если все элементы истинные (или, если последовательность пуста).

**any**(последовательность) - Возвращает True, если хотя бы один элемент - истина. Для пустой последовательности возвращает False.

**ascii(object)** - Как repr(), возвращает строку, содержащую представление объекта, но заменяет не-ASCII символы на экранированные последовательности.

**chr(X)** - Возвращает односимвольную строку, код символа которой равен X.

**compile(source, filename, mode, flags=0, dont\_inherit=False)** - Компиляция в программный код, который впоследствии может выполняться функцией eval или exec. Строка не должна содержать символов возврата каретки или нулевые байты.

**delattr(object, name)** - Удаляет атрибут с именем 'name'.

**dir([object])** - Список имен объекта, а если объект не указан, список имен в текущей локальной области видимости.

**divmod(a, b)** - Возвращает частное и остаток от деления a на b.

**enumerate(iterable, start=0)** - Возвращает итератор, при каждом проходе предоставляющем кортеж из номера и соответствующего члена последовательности.

**eval(expression, globals=None, locals=None)** - Выполняет строку программного кода.

**exec(object[, globals[, locals]])** - Выполняет программный код на Python.

**filter(function, iterable)** - Возвращает итератор из тех элементов, для которых function возвращает истину.

**getattr(object, name [,default])** - извлекает атрибут объекта или default.

**globals()** - Словарь глобальных имен.

**hasattr(object, name)** - Имеет ли объект атрибут с именем 'name'.

**help([object])** - Вызов встроенной справочной системы.

**id(object)** - Возвращает "адрес" объекта. Это целое число, которое гарантированно будет уникальным и постоянным для данного объекта в течение срока его существования.

**input**([prompt]) - Возвращает введенную пользователем строку. Prompt - подсказка пользователю.

**isinstance**(object, ClassInfo) - Истина, если объект является экземпляром ClassInfo или его подклассом. Если объект не является объектом данного типа, функция всегда возвращает ложь.

**issubclass**(класс, ClassInfo) - Истина, если класс является подклассом ClassInfo. Класс считается подклассом себя.

**locals**() - словарь локальных имен.

**map**(function, iterator) - итератор, получившийся после применения к каждому элементу последовательности функции function.

**max**(iter, [args ...] \* [, key]) - максимальный элемент последовательности.

**min**(iter, [args ...] \* [, key]) - минимальный элемент последовательности.

**open**(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True) - открывает файл и возвращает соответствующий поток.

**ord**(c) - код символа.

**print**([object, ...], \*, sep=" ", end="\n", file=sys.stdout) - печать.

**round**(X [, N]) - округление до N знаков после запятой.

**setattr**(объект, имя, значение) - устанавливает атрибут объекта.

**sorted**(iterable[, key][, reverse]) - отсортированный список.

**sum**(iter, start=0) - сумма членов последовательности. Для более точного значения суммы последовательности из чисел с плавающей точкой используйте **math.fsum()** из модуля **math**.

**super**([тип [, объект или тип]]) - доступ к родительскому классу.

**type**(object) - тип объекта.

**type**(name, bases, dict) - возвращает новый экземпляр класса name.

**vars**([object]) - словарь из атрибутов объекта. По умолчанию - словарь локальных имен.

**zip(\*iters)** - итератор, возвращающий кортежи, состоящие из соответствующих элементов аргументов-последовательностей.

## Числа

### Целые числа (int)

Числа в Python 3 ничем не отличаются от обычных чисел. Они поддерживают набор самых обычных математических операций:

$x + y$	Сложение
$x - y$	Вычитание
$x * y$	Умножение
$x / y$	Деление
$x // y$	Получение целой части от деления
$x \% y$	Остаток от деления
$-x$	Смена знака числа
$\text{abs}(x)$	Модуль числа
$\text{divmod}(x, y)$	Пара ( $x // y$ , $x \% y$ )
$x ** y$	Возведение в степень
$\text{pow}(x, y[, z])$	$x^y$ по модулю (если модуль задан)

Также нужно отметить, что числа в python 3, в отличие от многих других языков, поддерживают длинную арифметику (однако, это требует больше памяти).

```
>>> 255 + 34
289
>>> 5 * 2
10
>>> 20 / 3
6.666666666666667
>>> 20 // 3
6
>>> 20 % 3
2
>>> 3 ** 4
81
>>> pow(3, 4)
81
```

```
>>> pow(3, 4, 27)
0
>>> 3 ** 150
369988485035126972924700782451696644186473100389722973815
184405301748249
```

## Битовые операции

Над целыми числами также можно производить битовые операции

$x   y$	Побитовое <i>или</i>
$x \wedge y$	Побитовое <i>исключающее или</i>
$x \& y$	Побитовое <i>и</i>
$x \ll n$	Битовый сдвиг влево
$x \gg y$	Битовый сдвиг вправо
$\sim x$	Инверсия битов

## Дополнительные методы

**int.bit\_length()** - количество бит, необходимых для представления числа в двоичном виде, без учёта знака и лидирующих нулей.

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

**int.to\_bytes(length, byteorder, \*, signed=False)** - возвращает строку байтов, представляющих это число.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() // 8) + 1,
byteorder='little')
b'\xe8\x03'
```

classmethod **int.from\_bytes**(bytes, byteorder, \*, signed=False) - возвращает число из данной строки байтов.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big',
signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big',
signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

## Системы счисления

Те, у кого в школе была информатика, знают, что числа могут быть представлены не только в десятичной системе счисления. К примеру, в компьютере используется двоичный код, и, к примеру, число 19 в двоичной системе счисления будет выглядеть как 10011. Также иногда нужно переводить числа из одной системы счисления в другую. Python для этого предоставляет несколько функций:

- **int**([object], [основание системы счисления]) - преобразование к целому числу в десятичной системе счисления. По умолчанию система счисления десятичная, но можно задать любое основание от 2 до 36 включительно.
- **bin**(x) - преобразование целого числа в двоичную строку.
- **hex**(x) - преобразование целого числа в шестнадцатеричную строку.
- **oct**(x) - преобразование целого числа в восьмеричную строку.

Примеры:

```
>>> a = int('19') # Переводим строку в число
>>> b = int('19.5') # Строка не является целым числом
Traceback (most recent call last):
  File "", line 1, in
ValueError: invalid literal for int() with base 10:
'19.5'
```



```

>>> c = int(19.5) # Применённая к числу с плавающей
точкой, отсекает дробную часть
>>> print(a, c)
19 19
>>> bin(19)
'0b10011'
>>> oct(19)
'0o23'
>>> hex(19)
'0x13'
>>> 0b10011 # Так тоже можно записывать числовые
константы
19
>>> int('10011', 2)
19
>>> int('0b10011', 2)
19

```

## Вещественные числа (float)

Вещественные числа поддерживают те же операции, что и целые. Однако (из-за представления чисел в компьютере) вещественные числа неточны, и это может привести к ошибкам:

```

>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 +
0.1
0.9999999999999999

```

Для высокой точности используют другие объекты (например decimal и fraction).

Также вещественные числа не поддерживают длинную арифметику:

```

>>> a = 3 ** 1000
>>> a + 0.1
Traceback (most recent call last):
  File "", line 1, in
OverflowError: int too large to convert to float

```

Простенькие примеры работы с числами:

```

>>> c = 150
>>> d = 12.9
>>> c + d

```

```

162.9
>>> p = abs(d - c) # Модуль числа
>>> print(p)
137.1
>>> round(p) # Округление
137

```

## Дополнительные методы

**float.as\_integer\_ratio()** - пара целых чисел, чьё отношение равно этому числу.

**float.is\_integer()** - является ли значение целым числом.

**float.hex()** - переводит float в hex (шестнадцатеричную систему счисления).

classmethod **float.fromhex(s)** - float из шестнадцатеричной строки.

```

>>> (10.5).hex()
'0x1.5000000000000p+3'
>>> float.fromhex('0x1.5000000000000p+3')
10.5

```

Помимо стандартных выражений для работы с числами (а в Python их не так уж и много), в составе Python есть несколько полезных модулей.

Модуль **math** предоставляет более сложные математические функции.

```

>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887

```

Модуль **random** реализует генератор случайных чисел и функции случайного выбора.

```

>>> import random
>>> random.random()
0.15651968855132303

```

## Комплексные числа (complex)

В Python встроены также и комплексные числа:

```

>>> x = complex(1, 2)
>>> print(x)
(1+2j)
>>> y = complex(3, 4)
>>> print(y)
(3+4j)
>>> z = x + y
>>> print(x)
(1+2j)
>>> print(z)
(4+6j)
>>> z = x * y
>>> print(z)
(-5+10j)
>>> z = x / y
>>> print(z)
(0.44+0.08j)
>>> print(x.conjugate()) # Сопряжённое число
(1-2j)
>>> print(x.imag) # Мнимая часть
2.0
>>> print(x.real) # Действительная часть
1.0
>>> print(x > y) # Комплексные числа нельзя сравнить
Traceback (most recent call last):
  File "", line 1, in
TypeError: unorderable types: complex() > complex()
>>> print(x == y) # Но можно проверить на равенство
False
>>> abs(3 + 4j) # Модуль комплексного числа
5.0
>>> pow(3 + 4j, 2) # Возведение в степень
(-7+24j)

```

Также для работы с комплексными числами используется также модуль `cmath`.

## Округление чисел и его особенности

Округлением чисел в Python занимается встроенная функция `round`.

**round**(number[, ndigits]) - округляет число `number` до `ndigits` знаков после запятой (по умолчанию, до нуля знаков, то есть, до ближайшего целого)

```
>>> round(1.5)
2
>>> round(2.5)
2
>>> round(2.65, 1)
2.6
>>> round(2.75, 1)
2.8
```

Тут есть одна особенность, о которой нужно знать, и о которой часто забывают.

Со школы многие привыкли, что, когда  $(N + 1)$  знак = 5, а последующие знаки равны нулю, округление производится всегда в большую по модулю сторону.

Однако, как видно из примеров, в Python это не так. Здесь используется так называемое "Банковское округление", то есть округление к ближайшему чётному.

На практике это оказывается не так уж и важно, например:

```
>>> round(2.85, 1)
2.9
```

Что-то не так, правда? На самом деле, всё именно так, как и задумывалось. Просто из-за проблем с точностью чисел с плавающей точкой это число чуть больше, чем 2.85, а потому округляется до 2.9.

```
>>> from fractions import Fraction
>>> a = Fraction(2.85)
>>> b = Fraction('2.85')
>>> a == b
False
>>> a > b
True
```

# Строки

Строки в Python - упорядоченные последовательности символов, используемые для хранения и представления текстовой информации, поэтому с помощью строк можно работать со всем, что может быть представлено в текстовой форме.

## Литералы строк

Работа со строками в Python очень удобна. Существует несколько литералов строк, которые мы сейчас и рассмотрим.

### Строки в апострофах и в кавычках

```
S = 'spam"s'  
S = "spam's"
```

Строки в апострофах и в кавычках - одно и то же. Причина наличия двух вариантов в том, чтобы позволить вставлять в литералы строк символы кавычек или апострофов, не используя экранирование.

### Экранированные последовательности - служебные символы

Экранированные последовательности позволяют вставить символы, которые сложно ввести с клавиатуры.

Экранированная последовательность	Назначение
\n	Перевод строки
\a	Звонок
\b	Забой
\f	Перевод страницы
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\N{id}	Идентификатор ID базы данных Юникода
\uhhhh	16-битовый символ Юникода в 16-ричном представлении
\Uhhhh...	32-битовый символ Юникода в 32-

Экранированная последовательность	Назначение
	ричном представлении
\xhh	16-ричное значение символа
\ooo	8-ричное значение символа
\0	Символ Null (не является признаком конца строки)

## "Сырые" строки - подавляют экранирование

Если перед открывающей кавычкой стоит символ 'r' (в любом регистре), то механизм экранирования отключается.

```
S = r'C:\newt.txt'
```

Но, несмотря на назначение, "сырая" строка не может заканчиваться символом обратного слэша. Пути решения:

```
S = r'\n\n\'[: -1]
S = r'\n\n' + '\\\'
S = '\\n\n'
```

## Строки в тройных апострофах или кавычках

Главное достоинство строк в тройных кавычках в том, что их можно использовать для записи многострочных блоков текста. Внутри такой строки возможно присутствие кавычек и апострофов, главное, чтобы не было трех кавычек подряд.

```
>>> c = '''это очень большая
строка, многострочный
блок текста'''
>>> c
'это очень большая\nстрока, многострочный\nблок текста'
>>> print(c)
это очень большая
строка, многострочный
блок текста
```

Это все о литералах строк и работе с ними. О функциях и методах строк я расскажу в следующей статье.

## Функции и методы строк

### Базовые операции

- Конкатенация (сложение)

```
>>> S1 = 'spam'
>>> S2 = 'eggs'
>>> print(S1 + S2)
'spameggs'
```

- Дублирование строки

```
>>> print('spam' * 3)
spamspamspam
```

- Длина строки (функция len)

```
>>> len('spam')
4
```

- Доступ по индексу

```
>>> S = 'spam'
>>> S[0]
's'
>>> S[2]
'a'
>>> S[-2]
'a'
```

Как видно из примера, в Python возможен и доступ по отрицательному индексу, при этом отсчет идет от конца строки.

- Извлечение среза

Оператор извлечения среза: [X:Y]. X – начало среза, а Y – окончание;

символ с номером Y в срез не входит. По умолчанию первый индекс равен 0, а второй - длине строки.

```
>>> s = 'spameggs'
```

```
>>> s[3:5]
'me'
>>> s[2:-2]
'ameg'
>>> s[:6]
'spameg'
>>> s[1:]
'pameggs'
>>> s[:]
'spameggs'
```

Кроме того, можно задать шаг, с которым нужно извлекать срез.

```
>>> s[::-1]
'sggemaps'
>>> s[3:5:-1]
''
>>> s[2::2]
'aeg'
```

## Другие функции и методы строк

При вызове методов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку.

```
>>> s = 'spam'
>>> s[1] = 'b'
Traceback (most recent call last):
  File "", line 1, in
    s[1] = 'b'
TypeError: 'str' object does not support item assignment
>>> s = s[0] + 'b' + s[2:]
>>> s
'sbam'
```



Поэтому все строковые методы возвращают новую строку, которую потом следует присвоить переменной.

**Таблица "Функции и методы строк"**

<b>Функция или метод</b>	<b>Назначение</b>
<b>S = 'str'; S = "str"; S = '''str'''; S = """"str""""</b>	Литералы строк
<b>S = "\n\r\t\n\b\b\b"</b>	Экранированные последовательности
<b>S = r"C:\temp\new"</b>	Неформатированные строки (подавляют экранирование)
<b>S = b"byte"</b>	Строка байтов
<b>S1 + S2</b>	Конкатенация (сложение строк)
<b>S1 * 3</b>	Повторение строки
<b>S[i]</b>	Обращение по индексу
<b>S[i:j:step]</b>	Извлечение среза
<b>len(S)</b>	Длина строки
<b>str in S</b>	Проверка на вхождение подстроки в строку
<b>S.find(str, [start],[end])</b>	Поиск подстроки в строке. Возвращает номер первого вхождения или -1
<b>S.rfind(str, [start],[end])</b>	Поиск подстроки в строке. Возвращает номер последнего вхождения или -1
<b>S.index(str, [start],[end])</b>	Поиск подстроки в строке. Возвращает номер первого вхождения или вызывает ValueError
<b>S.rindex(str, [start],[end])</b>	Поиск подстроки в строке. Возвращает номер последнего вхождения или вызывает ValueError
<b>S.replace(шаблон, замена)</b>	Замена шаблона
<b>S.split(символ)</b>	Разбиение строки по разделителю
<b>S.isdigit()</b>	Состоит ли строка из цифр
<b>S.isalpha()</b>	Состоит ли строка из букв
<b>S.isalnum()</b>	Состоит ли строка из цифр или букв
<b>S.islower()</b>	Состоит ли строка из символов в нижнем регистре
<b>S.isupper()</b>	Состоит ли строка из символов в верхнем регистре
<b>S.isspace()</b>	Состоит ли строка из неотображаемых

Функция или метод	Назначение
	символов (пробел, символ перевода страницы ('\\f'), "новая строка" ('\\n'), "перевод каретки" ('\\r'), "горизонтальная табуляция" ('\\t') и "вертикальная табуляция" ('\\v'))
<b>S.istitle()</b>	Начинаются ли слова в строке с заглавной буквы
<b>S.upper()</b>	Преобразование строки к верхнему регистру
<b>S.lower()</b>	Преобразование строки к нижнему регистру
<b>S.startswith(str)</b>	Начинается ли строка S с шаблона str
<b>S.endswith(str)</b>	Заканчивается ли строка S шаблоном str
<b>S.join(список)</b>	Сборка строки из списка с разделителем S
<b>ord(символ)</b>	Символ в его код ASCII
<b>chr(число)</b>	Код ASCII в символ
<b>S.capitalize()</b>	Переводит первый символ строки в верхний регистр, а все остальные - в нижний
<b>S.center(width, [fill])</b>	возвращает отцентрированную строку, по краям которой символ fill (пробел по умолчанию)
<b>S.count(str, [start],[end])</b>	Возвращает количество непересекающихся вхождений подстроки в диапазоне [начало, конец] (0 и длина строки по умолчанию)
<b>S.expandtabs([tabsize])</b>	Возвращает копию строки, в которой все символы табуляции заменяются одним или несколькими пробелами, в зависимости от текущего столбца. Если TabSize не указан, размер табуляции полагается равным 8 пробелов
<b>S.lstrip([chars])</b>	Удаление пробельных символов в начале строки
<b>S.rstrip([chars])</b>	Удаление пробельных символов в конце строки
<b>S.strip([chars])</b>	Удаление пробельных символов в начале и в конце строки
<b>S.partition(шаблон)</b>	Возвращает кортеж, содержащий часть перед первым шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден,

Функция или метод	Назначение
	возвращается кортеж, содержащий саму строку, а затем две пустых строки
<b>S.rpartition(sep)</b>	Возвращает кортеж, содержащий часть перед последним шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий две пустых строки, а затем саму строку
<b>S.swapcase()</b>	Переводит символы нижнего регистра в верхний, а верхнего – в нижний
<b>S.title()</b>	Первую букву каждого слова переводит в верхний регистр, а все остальные - в нижний
<b>S.zfill(width)</b>	Делает длину строки не меньшей width, по необходимости заполняя первые символы нулями
<b>S.ljust(width, fillchar=" ")</b>	Делает длину строки не меньшей width, по необходимости заполняя последние символы символом fillchar
<b>S.rjust(width, fillchar=" ")</b>	Делает длину строки не меньшей width, по необходимости заполняя первые символы символом fillchar
<b>S.format(*args, **kwargs)</b>	Форматирование строки

### Форматирование строк.

Иногда (а точнее, довольно часто) возникают ситуации, когда нужно сделать строку, подставив в неё некоторые данные, полученные в процессе выполнения программы (пользовательский ввод, данные из файлов и т. д.). Подстановку данных можно сделать с помощью форматирования строк. Форматирование можно сделать с помощью оператора %, и метода format.

### Форматирование строк с помощью метода format

Если для подстановки требуется только один аргумент, то значение - сам аргумент:

```
>>> 'Hello, {}!'.format('Vasya')
'Hello, Vasya!'
```

А если несколько, то значениями будут являться все аргументы со строками подстановки (обычных или именованных):

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{} , {} , {}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
>>> 'Coordinates: {latitude},
{longitude}'.format(latitude='37.24N', longitude='-
115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-
115.81W'}
>>> 'Coordinates: {latitude},
{longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Однако метод format умеет больше. Вот его синтаксис:

```
поле замены      ::= "{" [имя поля] ["!" преобразование]
[":" спецификация] "}"
имя поля         ::= arg_name ( "." имя атрибута | "["
индекс "]" ) *
преобразование   ::= "r" (внутреннее представление) | "s"
(человеческое представление)
спецификация     ::= см. ниже
```

Например:

```
>>> "Units destroyed: {players[0]}".format(players = [1,
2, 3])
'Units destroyed: 1'
>>> "Units destroyed: {players[0]!r}".format(players =
['1', '2', '3'])
'Units destroyed: '1''
```

Теперь спецификация формата:

```

спецификация ::= [[fill]align][sign][#][0][width][,]
[.precision][type]
заполнитель ::= символ кроме '{' или '}'
выравнивание ::= "<" | ">" | "=" | "^"
знак ::= "+" | "-" | " "
ширина ::= integer
точность ::= integer
тип ::= "b" | "c" | "d" | "e" | "E" | "f" | "F"
| "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"

```

Выравнивание производится при помощи символа-заполнителя. Доступны следующие варианты выравнивания:

Флаг	Значение
'<'	Символы-заполнители будут справа (выравнивание объекта по левому краю) (по умолчанию).
'>'	выравнивание объекта по правому краю.
'='	Заполнитель будет после знака, но перед цифрами. Работает только с числовыми типами.
'^'	Выравнивание по центру.

Опция "знак" используется только для чисел и может принимать следующие значения:

Флаг	Значение
'+'	Знак должен быть использован для всех чисел.
'-'	'-' для отрицательных, ничего для положительных.
'Пробел'	'-' для отрицательных, пробел для положительных.

Поле "тип" может принимать следующие значения:

Тип	Значение
'd', 'i', 'u'	Десятичное число.
'o'	Число в восьмеричной системе счисления.
'x'	Число в шестнадцатеричной системе счисления (буквы в нижнем регистре).

'X'	Число в шестнадцатеричной системе счисления (буквы в верхнем регистре).
'e'	Число с плавающей точкой с экспонентой (экспонента в нижнем регистре).
'E'	Число с плавающей точкой с экспонентой (экспонента в верхнем регистре).
'f', 'F'	Число с плавающей точкой (обычный формат).
'g'	Число с плавающей точкой. с экспонентой (экспонента в нижнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
'G'	Число с плавающей точкой. с экспонентой (экспонента в верхнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
'c'	Символ (строка из одного символа или число - код символа).
's'	Строка.
'%'	Число умножается на 100, отображается число с плавающей точкой, а за ним знак %.

И напоследок, несколько примеров:

```
>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'

>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"

>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
'right aligned'
>>> '{:^30}'.format('centered')
'centered'
>>> '{:*^30}'.format('centered') # use '*' as a fill char
'*****centered*****'

>>> '{:+f}; {:+f}'.format(3.14, -3.14) # show it always
```

```

'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14) # show a space
for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14) # show only the
minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'

>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin:
{0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin:
{0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'

>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'

```

Метод `format` является наиболее правильным, но часто можно встретить программный код с форматированием строк в форме оператора `%`.

### Форматирование строк с помощью оператора `%`

Если для подстановки требуется только один аргумент, то значение - сам аргумент:

```

>>> 'Hello, %s!' % 'Vasya'
'Hello, Vasya!'

```

А если несколько, то значением будет являться кортеж со строками подстановки:

```

>>> '%d %s, %d %s' % (6, 'bananas', 10, 'lemons')
'6 bananas, 10 lemons'

```

Теперь, а почему я пишу то `%d`, то `%s`? А всё зависит от того, что мы используем в качестве подстановки и что мы хотим получить в итоге.

Формат	Что получится
'%d', '%i', '%u'	Десятичное число.
'%o'	Число в восьмеричной системе счисления.
'%x'	Число в шестнадцатеричной системе счисления (буквы в нижнем регистре).
'%X'	Число в шестнадцатеричной системе счисления (буквы в верхнем регистре).
'%e'	Число с плавающей точкой с экспонентой (экспонента в нижнем регистре).
'%E'	Число с плавающей точкой с экспонентой (экспонента в верхнем регистре).
'%f', '%F'	Число с плавающей точкой (обычный формат).
'%g'	Число с плавающей точкой. с экспонентой (экспонента в нижнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
'%G'	Число с плавающей точкой. с экспонентой (экспонента в верхнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
'%c'	Символ (строка из одного символа или число - код символа).
'%r'	Строка (литерал python).
'%s'	Строка (как обычно воспринимается пользователем).
'%%'	Знак '%'

Спецификаторы преобразования записываются в следующем порядке:

1. %.
2. Ключ (опционально), определяет, какой аргумент из значения будет подставляться.
3. Флаги преобразования.
4. Минимальная ширина поля. Если \*, значение берётся из кортежа.
5. Точность, начинается с '.', затем - желаемая точность.
6. Модификатор длины (опционально).
7. Тип (см. таблицу выше).

```
>>> print ('%(language)s has %(number)03d quote types.' %
{"language": "Python", "number": 2})
```



Python has 002 quote types.

Флаги преобразования:

Флаг	Значение
"#"	Значение будет использовать альтернативную форму.
"0"	Свободное место будет заполнено нулями.
"-"	Свободное место будет заполнено пробелами справа.
" "	Свободное место будет заполнено пробелами справа.
"+"	Свободное место будет заполнено пробелами слева.

```
>>> '%.2s' % 'Hello!'
'He'

>>> '%.*s' % ( 2, 'Hello!' )
'He'

>>> '%-10d' % 25
'25          '

>>> '%+10f' % 25
'+25.000000'

>>> '%+10s' % 'Hello'
'      Hello'
```

## Списки (list)

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов (почти как массив, но типы могут отличаться).

Чтобы использовать списки, их нужно создать. Создать список можно несколькими способами. Например, можно обработать любой итерируемый объект (например, строку) встроенной функцией **list**:

```
>>> list('список')
['с', 'п', 'и', 'с', 'о', 'к']
```

Список можно создать и при помощи литерала:

```
>>> s = [] # Пустой список
>>> l = ['s', 'p', ['isok'], 2]
```

```
>>> s
[]
>>> l
['s', 'p', ['isok'], 2]
```

Как видно из примера, список может содержать любое количество любых объектов (в том числе и вложенные списки), или не содержать ничего.

И еще один способ создать список - это **генераторы списков**. Генератор списков - способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списков очень похожи на цикл for.

```
>>> c = [c * 3 for c in 'list']
>>> c
['lll', 'iii', 'sss', 'ttt']
```

Возможна и более сложная конструкция генератора списков:

```
>>> c = [c * 3 for c in 'list' if c != 'i']
>>> c
['lll', 'sss', 'ttt']
>>> c = [c + d for c in 'list' if c != 'i' for d in
'spam' if d != 'a']
>>> c
['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```

Но в сложных случаях лучше пользоваться обычным циклом for для генерации списков.

## Функции и методы списков

Создать создали, теперь нужно со списком что-то делать. Для списков доступны основные встроенные функции, а также методы списков.

**Таблица "методы списков"**

Метод	Что делает
<b>list.append(x)</b>	Добавляет элемент в конец списка
<b>list.extend(L)</b>	Расширяет список list, добавляя в конец все элементы списка L
<b>list.insert(i, x)</b>	Вставляет на i-ый элемент значение x
<b>list.remove(x)</b>	Удаляет первый элемент в списке, имеющий значение

Метод	Что делает
	x
<b>list.pop([i])</b>	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
<b>list.index(x, [start [, end]])</b>	Возвращает положение первого элемента от start до end со значением x
<b>list.count(x)</b>	Возвращает количество элементов со значением x
<b>list.sort([key = функция])</b>	Сортирует список на основе функции
<b>list.reverse()</b>	Разворачивает список
<b>list.copy()</b>	Поверхностная копия списка (новое в python 3.3)
<b>list.clear()</b>	Очищает список (новое в python 3.3)

Нужно отметить, что методы списков, в отличие от строковых методов, изменяют сам список, а потому результат выполнения не нужно записывать в эту переменную.

```
>>> l = [1, 2, 3, 5, 7]
>>> l.sort()
>>> l
[1, 2, 3, 5, 7]
>>> l = l.sort()
>>> print(l)
None
```

И, напоследок, примеры работы со списками:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
```

```
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Иногда, для увеличения производительности, списки заменяют гораздо менее гибкими массивами.

## Индексы и срезы

### Взятие элемента по индексу

Как и в других языках программирования, взятие по индексу:

```
>>> a = [1, 3, 8, 7]
>>> a[0]
1
>>> a[3]
7
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Как и во многих других языках, нумерация элементов начинается с нуля. При попытке доступа к несуществующему индексу возникает исключение `IndexError`.

В данном примере переменная `a` являлась списком, однако взять элемент по индексу можно и у других типов: строк, кортежей.

В Python также поддерживаются отрицательные индексы, при этом нумерация идёт с конца, например:

```
>>> a = [1, 3, 8, 7]
>>> a[-1]
7
>>> a[-4]
1
>>> a[-5]
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>  
IndexError: list index out of range
```

## Срезы

В Python, кроме индексов, существуют ещё и **срезы**.

item[START:STOP:STEP] - берёт срез от номера START, до STOP (не включая его), с шагом STEP. По умолчанию START = 0, STOP = длине объекта, STEP = 1. Соответственно, какие-нибудь (а возможно, и все) параметры могут быть опущены.

```
>>> a = [1, 3, 8, 7]  
>>> a[:]  
[1, 3, 8, 7]  
>>> a[1:]  
[3, 8, 7]  
>>> a[:3]  
[1, 3, 8]  
>>> a[::2]  
[1, 8]
```

Также все эти параметры могут быть и отрицательными:

```
>>> a = [1, 3, 8, 7]  
>>> a[::-1]  
[7, 8, 3, 1]  
>>> a[:-2]  
[1, 3]  
>>> a[-2::-1]  
[8, 3, 1]  
>>> a[1:4:-1]  
[]
```

В последнем примере получился пустой список, так как START < STOP, а STEP отрицательный. То же самое произойдёт, если диапазон значений окажется за пределами объекта:

```
>>> a = [1, 3, 8, 7]  
>>> a[10:20]  
[]
```

Также с помощью срезов можно не только извлекать элементы, но и добавлять и удалять элементы (разумеется, только для изменяемых последовательностей).

```
>>> a = [1, 3, 8, 7]
>>> a[1:3] = [0, 0, 0]
>>> a
[1, 0, 0, 0, 7]
>>> del a[:-3]
>>> a
[0, 0, 7]
```

## Кортежи (tuple)

Кортеж, по сути - неизменяемый список.

### Зачем нужны кортежи, если есть списки?

1. Защита от дурака. То есть кортеж защищен от изменений, как намеренных (что плохо), так и случайных (что хорошо).
2. Меньший размер. Дабы не быть голословным:

```
3.>>> a = (1, 2, 3, 4, 5, 6)
4.>>> b = [1, 2, 3, 4, 5, 6]
5.>>> a.__sizeof__()
6.36
7.>>> b.__sizeof__()
8.44
```

9. Возможность использовать кортежи в качестве ключей словаря:

```
10.>>> d = {(1, 1, 1) : 1}
11.>>> d
12.{(1, 1, 1): 1}
13.>>> d = {[1, 1, 1] : 1}
14.Traceback (most recent call last):
15.  File "<stdin>", line 1, in
16.      d = {[1, 1, 1] : 1}
17.TypeError: unhashable type: 'list'
```

## Как работать с кортежами?

С преимуществами кортежей разобрались, теперь встает вопрос - а как с ними работать. Примерно так же, как и со списками.

Создаем пустой кортеж:

```
>>> a = tuple() #С помощью встроенной функции tuple()
>>> a
()
>>> a = () #С помощью литерала кортежа
>>> a
()
>>>
```

Создаем кортеж из одного элемента:

```
>>> a = ('s')
>>> a
's'
```

Стоп. Получилась строка. Но как же так? Мы же кортеж хотели! Как же нам кортеж получить?

```
>>> a = ('s', )
>>> a
('s', )
```

Ура! Заработало! Все дело - в запятой. Сами по себе скобки ничего не значат, точнее, значат то, что внутри них находится одна инструкция, которая может быть отделена пробелами, переносом строк и прочим мусором. Кстати, кортеж можно создать и так:

```
>>> a = 's',
>>> a
('s', )
```

Но все же не увлекайтесь, и ставьте скобки, тем более, что бывают случаи, когда скобки необходимы.

Ну и создать кортеж из итерируемого объекта можно с помощью все той же пресловутой функции tuple()

```
>>> a = tuple('hello, world!')
>>> a
('h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!')
```

## Операции с кортежами

Все операции над списками, не изменяющие список (сложение, умножение на число, методы `index()` и `count()` и некоторые другие операции). Можно также по-разному менять элементы местами и так далее.

Например, гордость программистов на python - поменять местами значения двух переменных:

```
a, b = b, a
```

## Словари (dict)

**Словари в Python** - неупорядоченные коллекции произвольных объектов с доступом по ключу. Их иногда ещё называют ассоциативными массивами или хеш-таблицами.

Чтобы работать со словарём, его нужно создать. Создать его можно несколькими способами. Во-первых, с помощью литерала:

```
>>> d = {}
>>> d
{}
>>> d = {'dict': 1, 'dictionary': 2}
>>> d
{'dict': 1, 'dictionary': 2}
```

Во-вторых, с помощью функции **dict**:

```
>>> d = dict(short='dict', long='dictionary')
>>> d
{'short': 'dict', 'long': 'dictionary'}
>>> d = dict([(1, 1), (2, 4)])
>>> d
{1: 1, 2: 4}
```



В-третьих, с помощью метода fromkeys:

```
>>> d = dict.fromkeys(['a', 'b'])
>>> d
{'a': None, 'b': None}
>>> d = dict.fromkeys(['a', 'b'], 100)
>>> d
{'a': 100, 'b': 100}
```

В-четвертых, с помощью генераторов словарей, которые очень похожи на генераторы списков.

```
>>> d = {a: a ** 2 for a in range(7)}
>>> d
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

Теперь попробуем добавить записей в словарь и извлечь значения ключей:

```
>>> d = {1: 2, 2: 4, 3: 9}
>>> d[1]
2
>>> d[4] = 4 ** 2
>>> d
{1: 2, 2: 4, 3: 9, 4: 16}
>>> d['1']
Traceback (most recent call last):
  File "", line 1, in
    d['1']
KeyError: '1'
```

Как видно из примера, присвоение по новому ключу расширяет словарь, присвоение по существующему ключу перезаписывает его, а попытка извлечения несуществующего ключа порождает исключение. Для избежания исключения есть специальный метод (см. ниже), или можно перехватывать исключение.

Что же можно еще делать со словарями? Да то же самое, что и с другими объектами: встроенные функции, ключевые слова (например, циклы for и while), а также специальные методы словарей.

## Методы словарей

**dict.clear()** - очищает словарь.

**dict.copy()** - возвращает копию словаря.

classmethod **dict.fromkeys**(seq[, value]) - создает словарь с ключами из seq и значением value (по умолчанию None).

**dict.get**(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а возвращает default (по умолчанию None).

**dict.items()** - возвращает пары (ключ, значение).

**dict.keys()** - возвращает ключи в словаре.

**dict.pop**(key[, default]) - удаляет ключ и возвращает значение. Если ключа нет, возвращает default (по умолчанию бросает исключение).

**dict.popitem()** - удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение `KeyError`. Помните, что словари неупорядочены.

**dict.setdefault**(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ с значением default (по умолчанию None).

**dict.update**([other]) - обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются. Возвращает None (не новый словарь!).

**dict.values()** - возвращает значения в словаре.

# Множества (set и frozenset)

Множество в python - "контейнер", содержащий не повторяющиеся элементы в случайном порядке.

Создаём множества:

```
>>> a = set()
>>> a
set()
>>> a = set('hello')
>>> a
{'h', 'o', 'l', 'e'}
>>> a = {'a', 'b', 'c', 'd'}
>>> a
{'b', 'c', 'a', 'd'}
>>> a = {i ** 2 for i in range(10)} # генератор множеств
>>> a
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> a = {} # А так нельзя!
>>> type(a)
<class 'dict'>
```

Как видно из примера, множества имеет тот же литерал, что и словарь, но пустое множество с помощью литерала создать нельзя.

Множества удобно использовать для удаления повторяющихся элементов:

```
>>> words = ['hello', 'daddy', 'hello', 'mum']
>>> set(words)
{'hello', 'daddy', 'mum'}
```

С множествами можно выполнять множество операций: находить объединение, пересечение...

- `len(s)` - число элементов в множестве (размер множества).
- `x in s` - принадлежит ли `x` множеству `s`.
- `set.isdisjoint(other)` - истина, если `set` и `other` не имеют общих элементов.
- `set == other` - все элементы `set` принадлежат `other`, все элементы `other` принадлежат `set`.

- **set.issubset(other)** или **set <= other** - все элементы set принадлежат other.
- **set.issuperset(other)** или **set >= other** - аналогично.
- **set.union(other, ...)** или **set | other | ...** - объединение нескольких множеств.
- **set.intersection(other, ...)** или **set & other & ...** - пересечение.
- **set.difference(other, ...)** или **set - other - ...** - множество из всех элементов set, не принадлежащие ни одному из other.
- **set.symmetric\_difference(other)**; **set ^ other** - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
- **set.copy()** - копия множества.

И операции, непосредственно изменяющие множество:

- **set.update(other, ...)**; **set |= other | ...** - объединение.
- **set.intersection\_update(other, ...)**; **set &= other & ...** - пересечение.
- **set.difference\_update(other, ...)**; **set -= other | ...** - вычитание.
- **set.symmetric\_difference\_update(other)**; **set ^= other**
- **set.add(elem)** - добавляет элемент в множество.
- **set.remove(elem)** - удаляет элемент из множества. `KeyError`, если такого элемента не существует.
- **set.discard(elem)** - удаляет элемент, если он находится в множестве.
- **set.pop()** - удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым.
- **set.clear()** - очистка множества.

## frozenset

Единственное отличие set от frozenset заключается в том, что set - изменяемый тип данных, а frozenset - нет. Примерно похожая ситуация с списками и кортежами.

```
>>> a = set('qwerty')
>>> b = frozenset('qwerty')
>>> a == b
True
>>> type(a - b)

>>> type(a | b)

>>> a.add(1)
>>> b.add(1)
Traceback (most recent call last):
  File "", line 1, in
    b.add(1)
AttributeError: 'frozenset' object has no attribute 'add'
```

## Функции

### Именные функции, инструкция def

Функция в python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции **def**.

Определим простейшую функцию:

```
def add(x, y):
    return x + y
```

Инструкция **return** говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y.

Теперь мы ее можем вызвать:

```
>>> add(1, 10)
11
>>> add('abc', 'def')
'abcdef'
```

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

```
>>> def newfunc(n):  
    def myfunc(x):  
        return x + n  
    return myfunc  
  
>>> new = newfunc(100) # new - это функция  
>>> new(200)  
300  
>>>
```

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

```
>>> def func():  
    pass  
  
>>> print(func())  
None  
>>>
```

### Аргументы функции

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

```
>>> def func(a, b, c=2): # c - необязательный аргумент  
    return a + b + c  
  
>>> func(1, 2) # a = 1, b = 2, c = 2 (по умолчанию)  
5  
>>> func(1, 2, 3) # a = 1, b = 2, c = 3  
6  
>>> func(a=1, b=3) # a = 1, b = 3, c = 2  
6  
>>> func(a=3, c=6) # a = 3, c = 6, b не определен  
Traceback (most recent call last):  
  File "", line 1, in  
    func(a=3, c=6)  
TypeError: func() takes at least 2 arguments (2 given)
```

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится \*:

```
>>> def func(*args):  
    return args  
  
>>> func(1, 2, 3, 'abc')  
(1, 2, 3, 'abc')  
>>> func()  
( )  
>>> func(1)  
(1, )
```

Как видно из примера, args - это кортеж из всех переданных аргументов функции, и с переменной можно работать также, как и с кортежем.

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится \*\*:

```
>>> def func(**kwargs):  
    return kwargs  
  
>>> func(a=1, b=2, c=3)  
{'a': 1, 'c': 3, 'b': 2}  
>>> func()  
{}  
>>> func(a='python')  
{'a': 'python'}  
>>>
```

В переменной kwargs у нас хранится словарь, с которым мы, опять-таки, можем делать все, что нам заблагорассудится.

## Анонимные функции, инструкция lambda

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией `def func()`:

```
>>> func = lambda x, y: x + y  
>>> func(1, 2)  
3
```

```
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
>>>
```

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

```
>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
>>>
```

## Исключения и их обработка

Исключения (exceptions) - ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках.

Самый простейший пример исключения - деление на ноль:

```
>>> 100 / 0
Traceback (most recent call last):
  File "", line 1, in
    100 / 0
ZeroDivisionError: division by zero
```

В данном случае интерпретатор сообщил нам об исключении ZeroDivisionError, то есть делении на ноль. Также возможны и другие исключения:

```
>>> 2 + '1'
Traceback (most recent call last):
  File "", line 1, in
    2 + '1'
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> int('qwerty')
Traceback (most recent call last):
  File "", line 1, in
    int('qwerty')
ValueError: invalid literal for int() with base 10: 'qwerty'
```



В этих двух примерах генерируются исключения `TypeError` и `ValueError` соответственно. Подсказки дают нам полную информацию о том, где порождено исключение, и с чем оно связано.

Рассмотрим иерархию встроенных в python исключений, хотя иногда вам могут встретиться и другие, так как программисты могут создавать собственные исключения. Данный список актуален для python 3.3, в более ранних версиях есть незначительные изменения.

- **BaseException** - базовое исключение, от которого берут начало все остальные.
  - **SystemExit** - исключение, порождается функцией **sys.exit** при выходе из программы.
  - **KeyboardInterrupt** - порождается при прерывании программы пользователем (обычно сочетанием клавиш **Ctrl+C**).
  - **GeneratorExit** - порождается при вызове метода **close** объекта **generator**.
  - **Exception** - а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать.
    - **StopIteration** - порождается встроенной функцией **next**, если в итераторе больше нет элементов.
    - **ArithmeticError** - арифметическая ошибка.
      - **FloatingPointError** - порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
      - **OverflowError** - возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как python поддерживает длинные числа), но может возникать в некоторых других случаях.
      - **ZeroDivisionError** - деление на ноль.
    - **AssertionError** - выражение в функции **assert** ложно.
    - **AttributeError** - объект не имеет данного атрибута (значения или метода).
    - **BufferError** - операция, связанная с буфером, не может быть выполнена.
    - **EOFError** - функция наткнулась на конец файла и не смогла прочитать то, что хотела.

- **ImportError** - не удалось импортирование модуля или его атрибута.
- **LookupError** - некорректный индекс или ключ.
  - **IndexError** - индекс не входит в диапазон элементов.
  - **KeyError** - несуществующий ключ (в словаре, множестве или другом объекте).
- **MemoryError** - недостаточно памяти.
- **NameError** - не найдено переменной с таким именем.
  - **UnboundLocalError** - сделана ссылка на локальную переменную в функции, но переменная не определена ранее.
- **OSError** - ошибка, связанная с системой.
  - **BlockingIOError**
  - **ChildProcessError** - неудача при операции с дочерним процессом.
  - **ConnectionError** - базовый класс для исключений, связанных с подключениями.
    - **BrokenPipeError**
    - **ConnectionAbortedError**
    - **ConnectionRefusedError**
    - **ConnectionResetError**
  - **FileExistsError** - попытка создания файла или директории, которая уже существует.
  - **FileNotFoundError** - файл или директория не существует.
  - **InterruptedError** - системный вызов прерван входящим сигналом.
  - **IsADirectoryError** - ожидался файл, но это директория.
  - **NotADirectoryError** - ожидалась директория, но это файл.
  - **PermissionError** - не хватает прав доступа.
  - **ProcessLookupError** - указанного процесса не существует.
  - **TimeoutError** - закончилось время ожидания.
- **ReferenceError** - попытка доступа к атрибуту со слабой ссылкой.

- **RuntimeError** - возникает, когда исключение не попадает ни под одну из других категорий.
- **NotImplementedError** - возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
- **SyntaxError** - синтаксическая ошибка.
  - **IndentationError** - неправильные отступы.
    - **TabError** - смешивание в отступах табуляции и пробелов.
- **SystemError** - внутренняя ошибка.
- **TypeError** - операция применена к объекту несоответствующего типа.
- **ValueError** - функция получает аргумент правильного типа, но некорректного значения.
- **UnicodeError** - ошибка, связанная с кодированием / декодированием **unicode** в строках.
  - **UnicodeEncodeError** - исключение, связанное с кодированием **unicode**.
  - **UnicodeDecodeError** - исключение, связанное с декодированием **unicode**.
  - **UnicodeTranslateError** - исключение, связанное с переводом **unicode**.
- **Warning** - предупреждение.

Теперь, зная, когда и при каких обстоятельствах могут возникнуть исключения, мы можем их обрабатывать. Для обработки исключений используется конструкция **try - except**.

Первый пример применения этой конструкции:

```
>>> try:
    k = 1 / 0
except ZeroDivisionError:
    k = 0

>>> print(k)
0
```

В блоке **try** мы выполняем инструкцию, которая может породить исключение, а в блоке **except** мы перехватываем их. При этом перехватываются как само

исключение, так и его потомки. Например, перехватывая `ArithmeticError`, мы также перехватываем `FloatingPointError`, `OverflowError` и `ZeroDivisionError`.

```
>>> try:
    k = 1 / 0
except ArithmeticError:
    k = 100

>>> print(k)
100
```

Также возможна инструкция `except` без аргументов, которая перехватывает вообще всё (и прерывание с клавиатуры, и системный выход и т. д.). Поэтому в такой форме инструкция `except` практически не используется, а используется `except Exception`. Однако чаще всего перехватывают исключения по одному, для упрощения отладки (вдруг вы ещё другую ошибку сделаете, а `except` её перехватит).

Ещё две инструкции, относящиеся к нашей проблеме, это **`finally`** и **`else`**. `Finally` выполняет блок инструкций в любом случае, было ли исключение, или нет (применима, когда нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция `else` выполняется в том случае, если исключения не было.

```
f = open('1.txt')
ints = []
try:
    for line in f:
        ints.append(int(line))
except ValueError:
    print('Это не число. Выходим.')
except Exception:
    print('Это что ещё такое?')
else:
    print('Всё хорошо.')
finally:
    f.close()
    print('Я закрыл файл.')
```

# Именно в таком порядке: try, группа except, затем else, и только потом finally.

Это не число. Выходим.  
Я закрыл файл.

## Байтовые строки (bytes и bytearray)

Байтовые строки в Python - что это такое и с чем это едят? Байтовые строки очень похожи на обычные строки, но с некоторыми отличиями. Попробуем выяснить, с какими.

Что такое байты? Байт - минимальная единица хранения и обработки цифровой информации. Последовательность байт представляет собой какую-либо информацию (текст, картинку, мелодию...).

Создаём байтовую строку:

```
>>> b'bytes'
b'bytes'
>>> 'Байты'.encode('utf-8')
b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd1\x8b'
>>> bytes('bytes', encoding = 'utf-8')
b'bytes'
>>> bytes([50, 100, 76, 72, 41])
b'2dLN')'
```

Если первые три способа нам уже известны, то последний нужно пояснить. Функция bytes принимает список чисел от 0 до 255, и возвращает байты, получающиеся применением функции chr.

```
>>> chr(50)
'2'
>>> chr(100)
'd'
>>> chr(76)
'L'
```

Что делать с байтами? Хотя байтовые строки поддерживают практически все строковые методы, с ними мало что нужно делать. Обычно их надо записать в файл / прочесть из файла и преобразовать во что-либо другое (конечно, если

очень хочется, то можно и распечатать). Для преобразования в строку используется метод `decode`:

```
>>> b'xd0x91xd0xb0xd0xb9xd1x82xd1x8b'.decode('utf-8')
'Байты'
```

## Bytearray

Bytearray в python - массив байт. От типа **bytes** отличается только тем, что является изменяемым. Про него, в общем-то, больше рассказать нечего.

```
>>> b = bytearray(b'hello world!')
>>> b
bytearray(b'hello world!')
>>> b[0]
104
>>> b[0] = b'h'
Traceback (most recent call last):
  File "", line 1, in
    b[0] = b'h'
TypeError: an integer is required
>>> b[0] = 105
>>> b
bytearray(b'iello world!')
>>> for i in range(len(b)):
    b[i] += i

>>> b
bytearray(b'ifnos}vzun,')
```

## Файлы

В данном материале мы рассмотрим встроенные средства python для работы с файлами: открытие / закрытие, чтение и запись.

Итак, начнем. Прежде, чем работать с файлом, его надо открыть. С этим замечательно справится встроенная функция `open`:

```
f = open('text.txt', 'r')
```

У функции `open` много параметров, они указаны в статье "Встроенные функции", нам пока важны 3 аргумента: первый, это имя файла. Путь к файлу может быть относительным или абсолютным. Второй аргумент, это режим, в котором мы будем открывать файл.

Режим	Обозначение
'r'	открытие на чтение (является значением по умолчанию).
'w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.
'x'	открытие на запись, если файла не существует, иначе исключение.
'a'	открытие на дозапись, информация добавляется в конец файла.
'b'	открытие в двоичном режиме.
't'	открытие в текстовом режиме (является значением по умолчанию).
'+'	открытие на чтение и запись

Режимы могут быть объединены, то есть, к примеру, 'rb' - чтение в двоичном режиме. По умолчанию режим равен 'rt'.

И последний аргумент, `encoding`, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку.

## Чтение из файла

Открыли мы файл, а теперь мы хотим прочитать из него информацию. Для этого есть несколько способов, но большого интереса заслуживают лишь два из них.

Первый - метод `read`, читающий весь файл целиком, если был вызван без аргументов, и `n` символов, если был вызван с аргументом (целым числом `n`).

```
>>> f = open('text.txt')
>>> f.read(1)
'H'
>>> f.read()
'ello world!\nThe end.\n\n'
```

Ещё один способ сделать это - прочитать файл построчно, воспользовавшись циклом for:

```
>>> f = open('text.txt')
>>> for line in f:
    line

'Hello world!\n'
'\n'
'The end.\n'
'\n'
```

## Запись в файл

Теперь рассмотрим запись в файл. Попробуем записать в файл вот такой вот список:

```
>>> l = [str(i)+str(i-1) for i in range(20)]
>>> l
['0-1', '10', '21', '32', '43', '54', '65', '76', '87',
'98', '109', '1110', '1211', '1312', '1413', '1514',
'1615', '1716', '1817', '1918']
```

Откроем файл на запись:

```
>>> f = open('text.txt', 'w')
```

Запись в файл осуществляется с помощью метода write:

```
>>> for index in l:
    f.write(index + '\n')
```

```
4
3
3
3
3
```



Для тех, кто не понял, что это за цифры, поясню: метод write возвращает число записанных символов.

После окончания работы с файлом его **обязательно нужно закрыть** с помощью метода close:

```
>>> f.close()
```

Теперь попробуем воссоздать этот список из получившегося файла. Откроем файл на чтение (надеюсь, вы поняли, как это сделать?), и прочитаем строки.

```
>>> f = open('text.txt', 'r')
>>> l = [line.strip() for line in f]
>>> l
['0-1', '10', '21', '32', '43', '54', '65', '76', '87',
'98', '109', '1110', '1211', '1312', '1413', '1514',
'1615', '1716', '1817', '1918']
>>> f.close()
```

Мы получили тот же список, что и был. В более сложных случаях (словарях, вложенных кортежах и т. д.) алгоритм записи придумать сложнее. Но это и не нужно. В python уже давно придумали средства, такие как pickle или json, позволяющие сохранять в файле сложные структуры.

## With ... as - менеджеры контекста

Конструкция with ... as используется для оборачивания выполнения блока инструкций менеджером контекста. Иногда это более удобная конструкция, чем try...except...finally.

Синтаксис конструкции with ... as:

```
"with" expression ["as" target] ("," expression ["as"
target])* ":"
    suite
```

Теперь по порядку о том, что происходит при выполнении данного блока:

1. Выполняется выражение в конструкции with ... as.
2. Загружается специальный метод `__exit__` для дальнейшего использования.

3. Выполняется метод `__enter__`. Если конструкция `with` включает в себя слово `as`, то возвращаемое методом `__enter__` значение записывается в переменную.
4. Выполняется `suite`.
5. Вызывается метод `__exit__`, причём неважно, выполнилось ли `suite` или произошло исключение. В этот метод передаются параметры исключения, если оно произошло, или во всех аргументах значение `None`, если исключения не было.

Если в конструкции `with - as` было несколько выражений, то это эквивалентно нескольким вложенным конструкциям:

```
with A() as a, B() as b:  
    suite
```

эквивалентно

```
with A() as a:  
    with B() as b:  
        suite
```

Для чего применяется конструкция `with ... as`? Для гарантии того, что критические функции выполнятся в любом случае. Самый распространённый пример использования этой конструкции - открытие файлов. Я уже рассказывал об открытии файлов с помощью функции `open`, однако конструкция `with ... as`, как правило, является более удобной и гарантирует закрытие файла в любом случае.

Например:

```
with open('newfile.txt', 'w', encoding='utf-8') as g:  
    d = int(input())  
    print('1 / {} = {}'.format(d, 1 / d), file=g)
```

И вы можете быть уверены, что файл будет закрыт вне зависимости от того, что введёт пользователь.

# PEP 8 - руководство по написанию кода на Python

Этот документ описывает соглашение о том, как писать код для языка python, включая стандартную библиотеку, входящую в состав python.

PEP 8 создан на основе рекомендаций Гуидо ван Россума с добавлениями от Барри. Если где-то возникал конфликт, мы выбирали стиль Гуидо. И, конечно, этот PEP может быть неполным (фактически, он, наверное, никогда не будет закончен).

Ключевая идея Гуидо такова: код читается намного больше раз, чем пишется. Собственно, рекомендации о стиле написания кода направлены на то, чтобы улучшить читаемость кода и сделать его согласованным между большим числом проектов. В идеале, весь код будет написан в едином стиле, и любой сможет легко его прочесть.

Это руководство о согласованности и единстве. Согласованность с этим руководством очень важна. Согласованность внутри одного проекта еще важнее. А согласованность внутри модуля или функции — самое важное. Но важно помнить, что иногда это руководство неприменимо, и понимать, когда можно отойти от рекомендаций. Когда вы сомневаетесь, просто посмотрите на другие примеры и решите, какой выглядит лучше.

Две причины для того, чтобы нарушить данные правила:

1. Когда применение правила сделает код менее читаемым даже для того, кто привык читать код, который следует правилам.
2. Чтобы писать в едином стиле с кодом, который уже есть в проекте и который нарушает правила (возможно, в силу исторических причин) — впрочем, это возможность переписать чужой код.

## Внешний вид кода

### Отступы

Используйте 4 пробела на каждый уровень отступа.

Продолжительные строки должны выравнивать обернутые элементы либо вертикально, используя неявную линию в скобках (круглых, квадратных или

фигурных), либо с использованием висячего отступа. При использовании висячего отступа следует применять следующие соображения: на первой линии не должно быть аргументов, а остальные строки должны четко восприниматься как продолжение линии.

Правильно:

```
# Выровнено по открывающему разделителю
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

# Больше отступов включено для отличия его от остальных
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Неправильно:

```
# Аргументы на первой линии запрещены, если не
используется вертикальное выравнивание
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

# Больше отступов требуется, для отличия его от
остальных
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Опционально:

```
# Нет необходимости в большем количестве отступов.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

Закрывающие круглые/квадратные/фигурные скобки в многострочных конструкциях может находиться под первым непробельным символом последней строки списка, например:

```
my_list = [  
    1, 2, 3,  
    4, 5, 6,  
]  
result = some_function_that_takes_arguments(  
    'a', 'b', 'c',  
    'd', 'e', 'f',  
)
```

либо быть под первым символом строки, начинающей многострочную конструкцию:

```
my_list = [  
    1, 2, 3,  
    4, 5, 6,  
]  
result = some_function_that_takes_arguments(  
    'a', 'b', 'c',  
    'd', 'e', 'f',  
)
```

## Табуляция или пробелы?

Пробелы - самый предпочтительный метод отступов.

Табуляция должна использоваться только для поддержки кода, написанного с отступами с помощью табуляции.

Python 3 запрещает смешивание табуляции и пробелов в отступах.

Python 2 пытается преобразовать табуляцию в пробелы.

Когда вы вызываете интерпретатор Python 2 в командной строке с параметром `-t`, он выдает предупреждения (warnings) при использовании смешанного стиля в отступах, а запустив интерпретатор с параметром `-tt`, вы получите в этих местах ошибки (errors). Эти параметры очень рекомендуются!

## Максимальная длина строки

Ограничьте длину строки максимум 79 символами.

Для более длинных блоков текста с меньшими структурными ограничениями (строки документации или комментарии), длину строки следует ограничить 72 символами.

Ограничение необходимой ширины окна редактора позволяет иметь несколько открытых файлов бок о бок, и хорошо работает при использовании инструментов анализа кода, которые предоставляют две версии в соседних столбцах.

Некоторые команды предпочитают большую длину строки. Для кода, поддерживающегося исключительно или преимущественно этой группой, в которой могут прийти к согласию по этому вопросу, нормально увеличение длины строки с 80 до 100 символов (эффективно увеличивая максимальную длину до 99 символов), при условии, что комментарии и строки документации все еще будут 72 символа.

Стандартная библиотека Python консервативна и требует ограничения длины строки в 79 символов (а строк документации/комментариев в 72).

Предпочтительный способ переноса длинных строк является использование подразумеваемых продолжений строк Python внутри круглых, квадратных и фигурных скобок. Длинные строки могут быть разбиты на несколько строк, обернутые в скобки. Это предпочтительнее использования обратной косой черты для продолжения строки.

Обратная косая черта все еще может быть использована время от времени. Например, длинная конструкция `with` не может использовать неявные продолжения, так что обратная косая черта является приемлемой:

```
with open('/path/to/some/file/you/want/to/read') as
file_1, \
    open('/path/to/some/file/being/written', 'w') as
file_2:
    file_2.write(file_1.read())
```

Ещё один случай - `assert`.

Сделайте правильные отступы для перенесённой строки. Предпочтительнее вставить перенос строки после логического оператора, но не перед ним. Например:

```

class Rectangle(Blob):

    def __init__(self, width, height,
                  color='black', emphasis=None,
highlight=0):
        if (width == 0 and height == 0 and
            color == 'red' and emphasis == 'strong'
or
            highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red'
or
                                           emphasis is
None):
            raise ValueError("I don't think so -- values
are %s, %s" %
                               (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)

```

## Пустые строки

Отделяйте функции верхнего уровня и определения классов двумя пустыми строками.

Определения методов внутри класса разделяются одной пустой строкой.

Дополнительные пустые строки возможно использовать для разделения различных групп похожих функций. Пустые строки могут быть опущены между несколькими связанными однострочниками (например, набор фиктивных реализаций).

Используйте пустые строки в функциях, чтобы указать логические разделы.

Python расценивает символ control+L как незначащий (whitespace), и вы можете использовать его, потому что многие редакторы обрабатывают его как разрыв страницы — таким образом логические части в файле будут на разных страницах. Однако, не все редакторы распознают control+L и могут на его месте отображать другой символ.

## Кодировка исходного файла

Кодировка Python должна быть UTF-8 (ASCII в Python 2).

Файлы в ASCII (Python 2) или UTF-8 (Python 3) не должны иметь объявления кодировки.

В стандартной библиотеке, нестандартные кодировки должны использоваться только для целей тестирования, либо когда комментарий или строка документации требует упомянуть имя автора, содержащего не ASCII символы; в остальных случаях использование `\x`, `\u`, `\U` или `\N` - наиболее предпочтительный способ включить не ASCII символы в строковых литералах.

Начиная с версии python 3.0 в стандартной библиотеке действует следующее соглашение: все идентификаторы обязаны содержать только ASCII символы, и означать английские слова везде, где это возможно (во многих случаях используются сокращения или неанглийские технические термины). Кроме того, строки и комментарии тоже должны содержать лишь ASCII символы. Исключения составляют: (а) test case, тестирующий не-ASCII особенности программы, и (б) имена авторов. Авторы, чьи имена основаны не на латинском алфавите, должны транслитерировать свои имена в латиницу.

Проектам с открытым кодом для широкой аудитории также рекомендуется использовать это соглашение.

## Импорты

- Каждый импорт, как правило, должен быть на отдельной строке.

Правильно:

```
import os
import sys
```

Неправильно:

```
import sys, os
```

В то же время, можно писать так:

```
from subprocess import Popen, PIPE
```

- Импорты всегда помещаются в начале файла, сразу после комментариев к модулю и строк документации, и перед объявлением констант.



Импорты должны быть сгруппированы в следующем порядке:

1. импорты из стандартной библиотеки
2. импорты сторонних библиотек
3. импорты модулей текущего проекта

Вставляйте пустую строку между каждой группой импортов.

Указывайте спецификации `__all__` после импортов.

- Рекомендуется абсолютное импортирование, так как оно обычно более читаемо и ведет себя лучше (или, по крайней мере, даёт понятные сообщения об ошибках) если импортируемая система настроена неправильно (например, когда каталог внутри пакета заканчивается на `sys.path`):

- `import mypkg.sibling`
- `from mypkg import sibling`
- `from mypkg.sibling import example`

Тем не менее, явный относительный импорт является приемлемой альтернативой абсолютному импорту, особенно при работе со сложными пакетами, где использование абсолютного импорта было бы излишне подробным:

```
from . import sibling
from .sibling import example
```

В стандартной библиотеке следует избегать сложной структуры пакетов и всегда использовать абсолютные импорты.

Неявные относительно импорты никогда не должны быть использованы, и были удалены в Python 3.

- Когда вы импортируете класс из модуля, вполне можно писать вот так:

- `from myclass import MyClass`
- `from foo.bar.yourclass import YourClass`

Если такое написание вызывает конфликт имен, тогда пишете:

```
import myclass
import foo.bar.yourclass
```

И используйте "myclass.MyClass" и "foo.bar.yourclass.YourClass".

- Шаблоны импортов (from import \*) следует избегать, так как они делают неясным то, какие имена присутствуют в глобальном пространстве имён, что вводит в заблуждение как читателей, так и многие автоматизированные средства. Существует один оправданный пример использования шаблона импорта, который заключается в опубликовании внутреннего интерфейса как часть общественного API (например, переписав реализацию на чистом Python в модуле акселератора (и не будет заранее известно, какие именно функции будут перезаписаны)).

## Пробелы в выражениях и инструкциях

**Избегайте использования пробелов в следующих ситуациях:**

- Непосредственно внутри круглых, квадратных или фигурных скобок.

Правильно:

```
spam(ham[1], {eggs: 2})
```

Неправильно:

```
spam( ham[ 1 ], { eggs: 2 } )
```

- Непосредственно перед запятой, точки с запятой или двоеточием:

Правильно:

```
if x == 4: print(x, y); x, y = y, x
```

Неправильно:

```
if x == 4 : print(x , y) ; x , y = y , x
```

- Сразу перед открывающей скобкой, после которой начинается список аргументов при вызове функции:

Правильно:

```
spam(1)
```

Неправильно:

```
spam (1)
```

- Сразу перед открывающей скобкой, после которой следует индекс или срез:

Правильно:

```
dict['key'] = list[index]
```

Неправильно:

```
dict ['key'] = list [index]
```

- Использование более одного пробела вокруг оператора присваивания (или любого другого) для того, чтобы выровнять его с другим:

Правильно:

```
x = 1
y = 2
long_variable = 3
```

Неправильно:

```
x          = 1
y          = 2
long_variable = 3
```

## Другие рекомендации

- Всегда окружайте эти бинарные операторы одним пробелом с каждой стороны: присваивания (=, +=, -= и другие), сравнения (==, <, >, !=, <>, <=, >=, in, not in, is, is not), логические (and, or, not).
- Если используются операторы с разными приоритетами, попробуйте добавить пробелы вокруг операторов с самым низким приоритетом. Используйте свои собственные суждения, однако, никогда не используйте более одного пробела, и всегда используйте одинаковое количество пробелов по обе стороны бинарного оператора.

Правильно:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Неправильно:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- Не используйте пробелы вокруг знака =, если он используется для обозначения именованного аргумента или значения параметров по умолчанию.

Правильно:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

Неправильно:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

- Не используйте составные инструкции (несколько команд в одной строке).

Правильно:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Неправильно:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- Иногда можно писать тело циклов `while`, `for` или ветку `if` в той же строке, если команда короткая, но если команд несколько, никогда так не пишите. А также избегайте длинных строк!

Точно неправильно:

```
if foo == 'blah': do_blah_thing()  
for x in lst: total += x  
while t < 10: t = delay()
```

Вероятно, неправильно:

```
if foo == 'blah': do_blah_thing()  
else: do_non_blah_thing()  
  
try: something()  
finally: cleanup()  
  
do_one(); do_two(); do_three(long, argument,  
                               list, like, this)  
  
if foo == 'blah': one(); two(); three()
```

## Комментарии

Комментарии, противоречащие коду, хуже, чем отсутствие комментариев. Всегда исправляйте комментарии, если меняете код!

Комментарии должны являться законченными предложениями. Если комментарий — фраза или предложение, первое слово должно быть написано с большой буквы, если только это не имя переменной, которая начинается с маленькой буквы (никогда не изменяйте регистр переменной!).

Если комментарий короткий, можно опустить точку в конце предложения. Блок комментариев обычно состоит из одного или более абзацев, составленных из полноценных предложений, поэтому каждое предложение должно оканчиваться точкой.

Ставьте два пробела после точки в конце предложения.

Программисты, которые не говорят на английском языке, пожалуйста, пишите комментарии на английском, если только вы не уверены на 120%, что ваш код никогда не будут читать люди, не знающие вашего родного языка.

## Блоки комментариев

Блок комментариев обычно объясняет код (весь, или только некоторую часть), идущий после блока, и должен иметь тот же отступ, что и сам код. Каждая строчка такого блока должна начинаться с символа # и одного пробела после него (если только сам текст комментария не имеет отступа).

Абзацы внутри блока комментариев разделяются строкой, состоящей из одного символа #.

## "Встрочные" комментарии

Старайтесь реже использовать подобные комментарии.

Такой комментарий находится в той же строке, что и инструкция. "Встрочные" комментарии должны отделяться по крайней мере двумя пробелами от инструкции. Они должны начинаться с символа # и одного пробела.

Комментарии в строке с кодом не нужны и только отвлекают от чтения, если они объясняют очевидное. Не пишите вот так:

```
x = x + 1 # Increment x
```

Впрочем, такие комментарии иногда полезны:

```
x = x + 1 # Компенсация границы
```

## Строки документации

- Пишите документацию для всех публичных модулей, функций, классов, методов. Строки документации необязательны для приватных методов, но лучше написать, что делает метод. Комментарий нужно писать после строки с `def`.
- PEP 257 объясняет, как правильно и хорошо документировать. Заметьте, очень важно, чтобы закрывающие кавычки стояли на отдельной строке. А еще лучше, если перед ними будет ещё и пустая строка, например:

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

- 

"""

- Для однострочной документации можно оставить закрывающие кавычки на той же строке.

## Контроль версий

Если вам нужно использовать Subversion, CVS или RCS в ваших исходных кодах, делайте вот так:

```
__version__ = "$Revision: 1a40d4eaa00b $"  
# $Source$
```

Вставляйте эти строки после документации модуля перед любым другим кодом и отделяйте их пустыми строками по одной до и после.

## Соглашения по именованию

Соглашения по именованию переменных в python немного туманны, поэтому их список никогда не будет полным — тем не менее, ниже мы приводим список рекомендаций, действующих на данный момент. Новые модули и пакеты должны быть написаны согласно этим стандартам, но если в какой-либо уже существующей библиотеке эти правила нарушаются, предпочтительнее писать в едином с ней стиле.

### Главный принцип

Имена, которые видны пользователю как часть общественного API должны следовать конвенциям, которые отражают использование, а не реализацию.

### Описание: Стили имен

Существует много разных стилей. Поможем вам распознать, какой стиль именования используется, независимо от того, для чего он используется.

Обычно различают следующие стили:

- b (одионочная маленькая буква)
- B (одионочная заглавная буква)
- lowercase (слово в нижнем регистре)

- `lower_case_with_underscores` (слова из маленьких букв с подчеркиваниями)
- `UPPERCASE` (заглавные буквы)
- `UPPERCASE_WITH_UNDERSCORES` (слова из заглавных букв с подчеркиваниями)
- `CapitalizedWords` (слова с заглавными буквами, или `CapWords`, или `CamelCase`). Замечание: когда вы используете аббревиатуры в таком стиле, пишите все буквы аббревиатуры заглавными — `HTTPServerError` лучше, чем `HttpServerError`.
- `mixedCase` (отличается от `CapitalizedWords` тем, что первое слово начинается с маленькой буквы)
- `Capitalized_Words_With_Underscores` (слова с заглавными буквами и подчеркиваниями — уродливо!)

Ещё существует стиль, в котором имена, принадлежащие одной логической группе, имеют один короткий префикс. Этот стиль редко используется в python, но мы упоминаем его для полноты. Например, функция `os.stat()` возвращает кортеж, имена в котором традиционно имеют вид `st_mode`, `st_size`, `st_mtime` и так далее. (Так сделано, чтобы подчеркнуть соответствие этих полей структуре системных вызовов POSIX, что помогает знакомым с ней программистам).

В библиотеке X11 используется префикс X для всех public-функций. В python этот стиль считается излишним, потому что перед полями и именами методов стоит имя объекта, а перед именами функций стоит имя модуля.

В дополнение к этому, используются следующие специальные формы записи имен с добавлением символа подчеркивания в начало или конец имени:

- `_single_leading_underscore`: слабый индикатор того, что имя используется для внутренних нужд. Например, `from M import *` не будет импортировать объекты, чьи имена начинаются с символа подчеркивания.
  - `single_trailing_underscore_`: используется по соглашению для избежания конфликтов с ключевыми словами языка python, например:
- ```
Tkinter.Toplevel(master, class_='ClassName')
```
- `__double_leading_underscore`: изменяет имя атрибута класса, то есть в классе `FooBar` поле `__boo` становится `_FooBar__boo`.



- \_\_double\_leading\_and\_trailing\_underscore\_\_ (двойное подчеркивание в начале и в конце имени): магические методы или атрибуты, которые находятся в пространствах имен, управляемых пользователем. Например, \_\_init\_\_, \_\_import\_\_ или \_\_file\_\_. Не изобретайте такие имена, используйте их только так, как написано в документации.

## **Предписания: соглашения по именованию**

### Имена, которых следует избегать

Никогда не используйте символы `l` (маленькая латинская буква «эль»), `O` (заглавная латинская буква «о») или `I` (заглавная латинская буква «ай») как однобуквенные идентификаторы.

В некоторых шрифтах эти символы неотличимы от цифры один и нуля. Если очень нужно `l`, пишите вместо неё заглавную `L`.

### Имена модулей и пакетов

Модули должны иметь короткие имена, состоящие из маленьких букв. Можно использовать символы подчеркивания, если это улучшает читабельность. То же самое относится и к именам пакетов, однако в именах пакетов не рекомендуется использовать символ подчеркивания.

Так как имена модулей отображаются в имена файлов, а некоторые файловые системы являются нечувствительными к регистру символов и обрезают длинные имена, очень важно использовать достаточно короткие имена модулей — это не проблема в Unix, но, возможно, код окажется непереносимым в старые версии Windows, Mac, или DOS.

Когда модуль расширения, написанный на C или C++, имеет сопутствующий python-модуль (содержащий интерфейс высокого уровня), C/C++ модуль начинается с символа подчеркивания, например, `_socket`.

### Имена классов

Имена классов должны обычно следовать соглашению CapWords.

Вместо этого могут использоваться соглашения для именования функций, если интерфейс документирован и используется в основном как функции.

Обратите внимание, что существуют отдельные соглашения о встроенных именах: большинство встроенных имен - одно слово (либо два слитно

написанных слова), а соглашение `CapWords` используется только для именования исключений и встроенных констант.

### Имена исключений

Так как исключения являются классами, к исключениям применяется стиль именования классов. Однако вы можете добавить `Егг` в конце имени (если, конечно, исключение действительно является ошибкой).

### Имена глобальных переменных

Будем надеяться, что глобальные переменные используются только внутри одного модуля. Руководствуйтесь теми же соглашениями, что и для имен функций.

Добавляйте в модули, которые написаны так, чтобы их использовали с помощью `from M import *`, механизм `__all__`, чтобы предотвратить экспортирование глобальных переменных. Или же, используйте старое соглашение, добавляя перед именами таких глобальных переменных один символ подчеркивания (которым вы можете обозначить те глобальные переменные, которые используются только внутри модуля).

### Имена функций

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания — это необходимо, чтобы увеличить читабельность.

Стиль `mixedCase` допускается в тех местах, где уже преобладает такой стиль, для сохранения обратной совместимости.

### Аргументы функций и методов

Всегда используйте `self` в качестве первого аргумента метода экземпляра объекта.

Всегда используйте `cls` в качестве первого аргумента метода класса.

Если имя аргумента конфликтует с зарезервированным ключевым словом `python`, обычно лучше добавить в конец имени символ подчеркивания, чем исказить написание слова или использовать аббревиатуру. Таким образом, `class_` лучше, чем `clss`. (Возможно, хорошим вариантом будет подобрать синоним).

## Имена методов и переменных экземпляров классов

Используйте тот же стиль, что и для имен функций: имена должны состоять из маленьких букв, а слова разделяться символами подчеркивания.

Используйте один символ подчёркивания перед именем для непубличных методов и атрибутов.

Чтобы избежать конфликтов имен с подклассами, используйте два ведущих подчеркивания.

Python искажает эти имена: если класс Foo имеет атрибут с именем \_\_a, он не может быть доступен как Foo.\_\_a. (Настойчивый пользователь все еще может получить доступ, вызвав Foo.\_Foo\_\_a.) Вообще, два ведущих подчеркивания должны использоваться только для того, чтобы избежать конфликтов имен с атрибутами классов, предназначенных для наследования.

Примечание: есть некоторые разногласия по поводу использования \_\_ имена (см. ниже).

## Константы

Константы обычно объявляются на уровне модуля и записываются только заглавными буквами, а слова разделяются символами подчеркивания. Например: MAX\_OVERFLOW, TOTAL.

## Проектирование наследования

Обязательно решите, каким должен быть метод класса или экземпляра класса (далее - атрибут) — публичный или непубличный. Если вы сомневаетесь, выберите непубличный атрибут. Потом будет проще сделать его публичным, чем наоборот.

Публичные атрибуты — это те, которые будут использовать другие программисты, и вы должны быть уверены в отсутствии обратной несовместимости. Непубличные атрибуты, в свою очередь, не предназначены для использования третьими лицами, поэтому вы можете не гарантировать, что не измените или не удалите их.

Мы не используем термин "приватный атрибут", потому что на самом деле в python таких не бывает.

Другой тип атрибутов классов принадлежит так называемому API подклассов (в других языках они часто называются `protected`). Некоторые классы проектируются так, чтобы от них наследовали другие классы, которые расширяют или модифицируют поведение базового класса. Когда вы проектируете такой класс, решите и явно укажите, какие атрибуты являются публичными, какие принадлежат API подклассов, а какие используются только базовым классом.

Теперь сформулируем рекомендации:

- Открытые атрибуты не должны иметь в начале имени символа подчеркивания.
- Если имя открытого атрибута конфликтует с ключевым словом языка, добавьте в конец имени один символ подчеркивания. Это более предпочтительно, чем аббревиатура или искажение написания (однако, у этого правила есть исключение — аргумента, который означает класс, и особенно первый аргумент метода класса (`class method`) должен иметь имя `cls`).
- Назовите простые публичные атрибуты понятными именами и не пишите сложные методы доступа и изменения (`accessor/mutator`, `get/set`, — прим. перев.) Помните, что в `python` очень легко добавить их потом, если потребуется. В этом случае используйте свойства (`properties`), чтобы скрыть функциональную реализацию за синтаксисом доступа к атрибутам.

Примечание 1: Свойства (`properties`) работают только в классах нового стиля (в `Python 3` все классы являются таковыми).

Примечание 2: Постарайтесь избавиться от побочных эффектов, связанным с функциональным поведением; впрочем, такие вещи, как кэширование, вполне допустимы.

Примечание 3: Избегайте использования вычислительно затратных операций, потому что из-за записи с помощью атрибутов создается впечатление, что доступ происходит (относительно) быстро.

- Если вы планируете класс таким образом, чтобы от него наследовались другие классы, но не хотите, чтобы подклассы унаследовали некоторые атрибуты, добавьте в имена два символа подчеркивания в начало, и ни одного — в конец. Механизм изменения имен в `python` работает так, что имя класса добавится к имени такого атрибута, что позволит избежать конфликта имен с атрибутами подклассов.

Примечание 1: Будьте внимательны: если подкласс будет иметь то же имя класса и имя атрибута, то вновь возникнет конфликт имен.

Примечание 2: Механизм изменения имен может затруднить отладку или работу с `__getattr__()`, однако он хорошо документирован и легко реализуется вручную.

Примечание 3: Не всем нравится этот механизм, поэтому старайтесь достичь компромисса между необходимостью избежать конфликта имен и возможностью доступа к этим атрибутам.

## Общие рекомендации

- Код должен быть написан так, чтобы не зависеть от разных реализаций языка (PyPy, Jython, IronPython, Pyrex, Psycο и пр.).

Например, не полагайтесь на эффективную реализацию в CPython конкатенации строк в выражениях типа `a+=b` или `a=a+b`. Такие инструкции выполняются значительно медленнее в Jython. В критичных к времени выполнения частях программы используйте `".join()"` — таким образом склеивание строк будет выполнено за линейное время независимо от реализации python.

- Сравнения с `None` должны обязательно выполняться с использованием операторов `is` или `is not`, а не с помощью операторов сравнения. Кроме того, не пишите `if x`, если имеете в виду `if x is not None` — если, к примеру, при тестировании такая переменная может принять значение другого типа, отличного от `None`, но при приведении типов может получиться `False`!
- При реализации методов сравнения, лучше всего реализовать все 6 операций сравнения (`__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`), чем полагаться на то, что другие программисты будут использовать только конкретный вид сравнения.

Для минимизации усилий можно воспользоваться декоратором `functools.total_ordering()` для реализации недостающих методов.

PEP 207 указывает, что интерпретатор может поменять `y > x` на `x < y`, `y >= x` на `x <= y`, и может поменять местами аргументы `x == y` и `x != y`. Гарантируется, что операции `sort()` и `min()` используют оператор `<`, а `max()` использует оператор `>`. Однако, лучше всего осуществить все шесть операций, чтобы не возникало путаницы в других местах.

- Всегда используйте выражение `def`, а не присваивание лямбда-выражения к имени.

Правильно:

```
def f(x): return 2*x
```

Неправильно:

```
f = lambda x: 2*x
```

- Наследуйте свой класс исключения от `Exception`, а не от `BaseException`. Прямое наследование от `BaseException` зарезервировано для исключений, которые не следует перехватывать.
- Используйте цепочки исключений соответствующим образом. В Python 3, "raise X from Y" следует использовать для указания явной замены без потери отладочной информации.

Когда намеренно заменяется исключение (использование "raise X" в Python 2 или "raise X from None" в Python 3.3+), проследите, чтобы соответствующая информация передалась в новое исключение (такие, как сохранение имени атрибута при преобразовании `KeyError` в `AttributeError` или вложение текста исходного исключения в новом).

- Когда вы генерируете исключение, пишите `raise ValueError('message')` вместо старого синтаксиса `raise ValueError, message`.

Старая форма записи запрещена в python 3.

Такое использование предпочтительнее, потому что из-за скобок не нужно использовать символы для продолжения перенесенных строк, если эти строки длинные или если используется форматирование.

- Когда код перехватывает исключения, перехватывайте конкретные ошибки вместо простого выражения `except`.

К примеру, пишите вот так:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

Простое написание "except:" также перехватит и SystemExit, и KeyboardInterrupt, что породит проблемы, например, сложнее будет завершить программу нажатием control+C. Если вы действительно собираетесь перехватить все исключения, пишите "except Exception:".

Хорошим правилом является ограничение использования "except:", кроме двух случаев:

1. Если обработчик выводит пользователю всё о случившейся ошибке; по крайней мере, пользователь будет знать, что произошла ошибка.
  2. Если нужно выполнить некоторый код после перехвата исключения, а потом вновь "бросить" его для обработки где-то в другом месте. Обычно же лучше пользоваться конструкцией "try...finally".
- При связывании перехваченных исключений с именем, предпочитайте явный синтаксис привязки, добавленный в Python 2.6:

```
• try:
•     process_data()
• except Exception as exc:
•     raise DataProcessingFailedError(str(exc))
```

Это единственный синтаксис, поддерживающийся в Python 3, который позволяет избежать проблем неоднозначности, связанных с более старым синтаксисом на основе запятой.

- При перехвате ошибок операционной системы, предпочитайте использовать явную иерархию исключений, введенную в Python 3.3, вместо анализа значений errno.
- Постарайтесь заключать в каждую конструкцию try...except минимум кода, чтобы легче отлавливать ошибки. Опять же, это позволяет избежать замаскированных ошибок.

Правильно:

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
```

```
else:
    return handle_value(value)
```

Неправильно:

```
try:
    # Здесь много действий!
    return handle_value(collection[key])
except KeyError:
    # Здесь также перехватится KeyError, который может
    # быть сгенерирован handle_value()
    return key_not_found(key)
```

- Когда ресурс является локальным на участке кода, используйте выражение `with` для того, чтобы после выполнения он был очищен оперативно и надёжно.
- Менеджеры контекста следует вызывать с помощью отдельной функции или метода, всякий раз, когда они делают что-то другое, чем получение и освобождение ресурсов. Например:

Правильно:

```
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
```

Неправильно:

```
with conn:
    do_stuff_in_transaction(conn)
```

Последний пример не дает никакой информации, указывающей на то, что `__enter__` и `__exit__` делают что-то кроме закрытия соединения после транзакции. Быть явным важно в данном случае.

- Используйте строковые методы вместо модуля `string` — они всегда быстрее и имеют тот же API для `unicode`-строк. Можно отказаться от этого правила, если необходима совместимость с версиями `python` младше 2.0.

В `Python 3` остались только строковые методы.

- Пользуйтесь `".startswith()"` и `".endswith()"` вместо обработки срезов строк для проверки суффиксов или префиксов.



startswith() и endswith() выглядят чище и порождают меньше ошибок. Например:

Правильно:

```
if foo.startswith('bar'):
```

Неправильно:

```
if foo[:3] == 'bar':
```

- Сравнение типов объектов нужно делать с помощью isinstance(), а не прямым сравнением типов:

Правильно:

```
if isinstance(obj, int):
```

Неправильно:

```
if type(obj) is type(1):
```

Когда вы проверяете, является ли объект строкой, обратите внимание на то, что строка может быть unicode-строкой. В python 2 у str и unicode есть общий базовый класс, поэтому вы можете написать:

```
if isinstance(obj, basestring):
```

Отметим, что в Python 3, unicode и basestring больше не существуют (есть только str) и bytes больше не является своего рода строкой (это последовательность целых чисел).

- Для последовательностей (строк, списков, кортежей) используйте тот факт, что пустая последовательность есть false:

Правильно:

```
if not seq:  
if seq:
```

Неправильно:

```
if len(seq)  
if not len(seq)
```

- Не пользуйтесь строковыми константами, которые имеют важные пробелы в конце — они невидимы, а многие редакторы (а теперь и reindent.py) обрезают их.
- Не сравнивайте логические типы с True и False с помощью ==:

Правильно:

```
if greeting:
```

Неправильно:

```
if greeting == True:
```

Совсем неправильно:

```
if greeting is True:
```

## Документирование кода в Python

**Документирование кода в python** - достаточно важный аспект, ведь от нее порой зависит читаемость и быстрота понимания вашего кода, как другими людьми, так и вами через полгода.

PEP 257 описывает соглашения, связанные со строками документации python, рассказывает о том, как нужно документировать python код.

Цель этого PEP - стандартизировать структуру строк документации: что они должны в себя включать, и как это написать (не касаясь вопроса синтаксиса строк документации). Этот PEP описывает соглашения, а не правила или синтаксис.

При нарушении этих соглашений, самое худшее, чего можно ожидать - некоторых неодобрительных взглядов. Но некоторые программы (например, docutils), знают о соглашениях, поэтому следование им даст вам самые лучшие результаты.

## Что такое строки документации?

Строки документации - строковые литералы, которые являются первым оператором в модуле, функции, классе или определении метода. Такая строка документации становится специальным атрибутом `__doc__` этого объекта.

Все модули должны, как правило, иметь строки документации, и все функции и классы, экспортируемые модулем также должны иметь строки документации. Публичные методы (в том числе `__init__`) также должны иметь строки документации. Пакет модулей может быть документирован в `__init__.py`.

Для согласованности, всегда используйте `"""triple double quotes"""` для строк документации. Используйте `r"""raw triple double quotes"""`, если вы будете использовать обратную косую черту в строке документации.

Существует две формы строк документации: однострочная и многострочная.

### Однострочные строки документации

Однострочники предназначены для действительно очевидных случаев. Они должны уместиться на одной строке. Например:

```
def kos_root():  
    """Return the pathname of the KOS root directory."""  
    global _kos_root  
    if _kos_root: return _kos_root
```

Используйте тройные кавычки, даже если документация уместается на одной строке. Потом будет проще её дополнить.

Закрывающие кавычки на той же строке. Это смотрится лучше.

Нет пустых строк перед или после документации.

Однострочная строка документации не должна быть "подписью" параметров функции / метода (которые могут быть получены с помощью интроспекции). Не делайте:

```
def function(a, b):  
    """function(a, b) -> list"""
```

Этот тип строк документации подходит только для C функций (таких, как встроенные модули), где интроспекция не представляется возможной. Тем не менее, возвращаемое значение не может быть определено путем интроспекции. Предпочтительный вариант для такой строки документации будет что-то вроде:

```
def function(a, b):  
    """Do X and return a list."""
```

(Конечно, "Do X" следует заменить полезным описанием!)

## **Многострочные строки документации**

Многострочные строки документации состоят из однострочной строки документации с последующей пустой строкой, а затем более подробным описанием. Первая строка может быть использована автоматическими средствами индексации, поэтому важно, чтобы она находилась на одной строке и была отделена от остальной документации пустой строкой. Первая строка может быть на той же строке, где и открывающие кавычки, или на следующей строке. Вся документация должна иметь такой же отступ, как кавычки на первой строке (см. пример ниже).

Вставляйте пустую строку до и после всех строк документации (однострочных или многострочных), которые документируют класс - вообще говоря, методы класса разделены друг от друга одной пустой строкой, а строка документации должна быть смещена от первого метода пустой строкой; для симметрии, поставьте пустую строку между заголовком класса и строкой документации. Строки документации функций и методов, как правило, не имеют этого требования.

**Строки документации скрипта** (самостоятельной программы) должны быть доступны в качестве "сообщения по использованию", напечатанной, когда программа вызывается с некорректными или отсутствующими аргументами (или, возможно, с опцией "-h", для помощи). Такая строка документации должна документировать функции программы и синтаксис командной строки, переменные окружения и файлы. Сообщение по использованию может быть довольно сложным (несколько экранов) и должно быть достаточным для нового пользователя для использования программы должным образом, а также полный справочник со всеми вариантами и аргументами для искушенного пользователя.

Строки документации модуля должны, как правило, перечислять классы, исключения, функции (и любые другие объекты), которые экспортируются модулем, с краткими пояснениями (в одну строчку) каждого из них. (Эти строки,

как правило, дают меньше деталей, чем первая строка документации к объекту). Строки документации пакета модулей (т.е. строка документации в `__init__.py`) также должны включать модули и подпакеты.

Строки документации функции или метода должны обобщить его поведение и документировать свои аргументы, возвращаемые значения, побочные эффекты, исключения, дополнительные аргументы, именованные аргументы, и ограничения на вызов функции.

Строки документации класса обобщают его поведение и перечисляют открытые методы и переменные экземпляра. Если класс предназначен для подклассов, и имеет дополнительный интерфейс для подклассов, этот интерфейс должен быть указан отдельно (в строке документации). Конструктор класса должен быть задокументирован в документации метода `__init__`. Отдельные методы должны иметь свои строки документации.

Если класс - подкласс другого класса, и его поведение в основном унаследовано от этого класса, строки документации должны отмечать это и обобщить различия. Используйте глагол "override", чтобы указать, что метод подкласса заменяет метод суперкласса и не вызывает его; используйте глагол "extend", чтобы указать, что метод подкласса вызывает метод суперкласса (в дополнение к собственному поведению).

Пример:

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
    ...
```

А ещё больше примеров можно посмотреть в стандартной библиотеке python (например, в папке Lib вашего интерпретатора python).

## Создание и подключение модулей

Модулем в Python называется любой файл с программой (да-да, все те программы, которые вы писали, можно назвать модулями). В этом материале мы поговорим о том, как создать модуль, и как подключить модуль, из стандартной библиотеки или написанный вами.

Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Нужно заметить, что модуль может быть написан не только на Python, а например, на C или C++.

## Подключение модуля из стандартной библиотеки

Подключить модуль можно с помощью инструкции `import`. К примеру, подключим модуль `os` для получения текущей директории:

```
>>> import os
>>> os.getcwd()
'C:\\Python33'
```

После ключевого слова **import** указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает читаемость кода. Импортируем модули `time` и `random`.

```
>>> import time, random
>>> time.time()
1376047104.056417
>>> random.random()
0.9874550833306869
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе `e`, расположенной в модуле `math`:

```
>>> import math
>>> math.e
2.718281828459045
```

Стоит отметить, что если указанный атрибут модуля не будет найден, возбуждается исключение `AttributeError`. А если не удастся найти модуль для импортирования, то `ImportError`.

```
>>> import notexist
Traceback (most recent call last):
  File "", line 1, in
    import notexist
ImportError: No module named 'notexist'
>>> import math
>>> math.Ë
Traceback (most recent call last):
  File "", line 1, in
    math.Ë
AttributeError: 'module' object has no attribute 'Ë'
```

## Использование псевдонимов

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним, с помощью ключевого слова `as`.

```
>>> import math as m
>>> m.e
2.718281828459045
```

Теперь доступ ко всем атрибутам модуля `math` осуществляется только с помощью переменной `m`, а переменной `math` в этой программе уже не будет (если, конечно, вы после этого не напишите `import math`, тогда модуль будет доступен как под именем `m`, так и под именем `math`).

## Инструкция `from`

Подключить определенные атрибуты модуля можно с помощью инструкции `from`. Она имеет несколько форматов:

```
from import [ as ], [ [ as ] ...]
from import *
```

Первый формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова `as`.

```
>>> from math import e, ceil as c
>>> e
2.718281828459045
>>> c(4.6)
```

Импортируемые атрибуты можно разместить на нескольких строках, если их много, для лучшей читаемости кода:

```
>>> from math import (sin, cos,
                       tan, atan)
```

Второй формат инструкции from позволяет подключить все (точнее, почти все) переменные из модуля. Для примера импортируем все атрибуты из модуля sys:

```
>>> from sys import *
>>> version
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC
v.1600 32 bit (Intel)]'
>>> version_info
sys.version_info(major=3, minor=3, micro=2,
releaselevel='final', serial=0)
```

Следует заметить, что не все атрибуты будут импортированы. Если в модуле определена переменная `__all__` (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. Если переменная `__all__` не определена, то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания. Кроме того, необходимо учитывать, что импортирование всех атрибутов из модуля может нарушить пространство имен главной программы, так как переменные, имеющие одинаковые имена, будут перезаписаны.

## Создание своего модуля на Python

Теперь пришло время создать свой модуль. Создадим файл `mymodule.py`, в которой определим какие-нибудь функции:

```
def hello():
    print('Hello, world!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b
```

Теперь в этой же папке создадим другой файл, например, `main.py`:



```
import mymodule

mymodule.hello()
print(mymodule.fib(10))
```

Выведет:

```
Hello, world!
55
```

Поздравляю! Вы **сделали свой модуль**! Отвечу ещё на пару вопросов, связанных с созданием модулей:

## Как назвать модуль?

Помните, что вы (или другие люди) будут его импортировать и использовать в качестве переменной. Модуль нельзя именовать также, как и ключевое слово. Также имена модулей нельзя начинать с цифры. И не стоит называть модуль также, как какую-либо из встроенных функций. То есть, конечно, можно, но это создаст большие неудобства при его последующем использовании.

## Куда поместить модуль?

Туда, где его потом можно будет найти. Пути поиска модулей указаны в переменной `sys.path`. В него включены текущая директория (то есть модуль можно оставить в папке с основной программой), а также директории, в которых установлен python. Кроме того, переменную `sys.path` можно изменять вручную, что позволяет положить модуль в любое удобное для вас место (главное, не забыть в главной программе модифицировать `sys.path`).

## Можно ли использовать модуль, как самостоятельную программу?

Можно. Однако надо помнить, что при импортировании модуля его код выполняется полностью, то есть, если программа что-то печатает, то при её импортировании это будет напечатано. Этого можно избежать, если проверять, запущен ли скрипт как программа, или импортирован. Это можно сделать с помощью переменной `__name__`, которая определена в любой программе, и равна `"__main__"`, если скрипт запущен в качестве главной программы, и имя, если он импортирован. Например, `mymodule.py` может выглядеть вот так:

```
def hello():
```

```

    print('Hello, world!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b

if __name__ == "__main__":
    hello()
    for i in range(10):
        print(fib(i))

```

## Объектно-ориентированное программирование. Общее представление

**Объектно-ориентированное программирование (ООП)** — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

**Класс** — тип, описывающий устройство объектов. **Объект** — это экземпляр класса. Класс можно сравнить с чертежом, по которому создаются объекты.

Python соответствует принципам объектно-ориентированного программирования. В python всё является объектами - и строки, и списки, и словари, и всё остальное.

Но возможности ООП в python этим не ограничены. Программист может написать свой тип данных (класс), определить в нём свои методы.

Это не является обязательным - мы можем пользоваться только встроенными объектами. Однако ООП полезно при долгосрочной разработке программы несколькими людьми, так как упрощает понимание кода.

Приступим теперь собственно к написанию своих классов на python. Попробуем определить собственный класс:

```

# Пример самого простейшего класса
class A:

```

```
pass
```

Теперь мы можем создать несколько экземпляров этого класса:

```
>>> a = A()
>>> b = A()
>>> a.arg = 1 # у экземпляра a появился атрибут arg,
равный 1
>>> b.arg = 2 # а у экземпляра b - атрибут arg, равный 2
>>> print(a.arg)
1
```

Классу возможно задать собственные методы:

```
class A:
    def g(self): # self - обязательный аргумент,
# содержащий в себе экземпляр класса, передающийся при
вызове метода,
# поэтому этот аргумент должен
присутствовать во всех методах класса.
        return 'hello world'

>>> a = A()
>>> a.g()
'hello world'
```

И еще один пример:

```
class B:
    arg = 'Python' # Все экземпляры этого класса будут
иметь атрибут arg, равный "Python"
# Но впоследствии мы его можем
изменить
    def g(self):
        return self.arg

>>> b = B()
>>> b.g()
'Python'
>>> B.g(b)
'Python'

>>> b.arg = 'spam'
>>> b.g()
```

```
'spam'
```

# Инкапсуляция, наследование, полиморфизм

Недавно мы говорили об основах объектно-ориентированного программирования в python, теперь продолжим эту тему и поговорим о таких понятиях ООП, как **инкапсуляция, наследование и полиморфизм**.

## Инкапсуляция

Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса.

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.

Одинокое подчеркивание в начале имени атрибута говорит о том, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени.

```
class A:
    def _private(self):
        print("Это приватный метод!")

>>> a = A()
>>> a._private()
Это приватный метод!
```

Двойное подчеркивание в начале имени атрибута даёт большую защиту: атрибут становится недоступным по этому имени.

```
class B:
    def __private(self):
        print("Это приватный метод!")

>>> b = B()
```

```
>>> b.__private()
Traceback (most recent call last):
  File "", line 1, in
    b.__private()
AttributeError: 'B' object has no attribute '__private'
```

Однако полностью это не защищает, так как атрибут всё равно остаётся доступным под именем `_ИмяКласса_ИмяАтрибута`:

```
>>> b._B__private()
Это приватный метод!
```

## Наследование

Наследование подразумевает то, что дочерний класс содержит все атрибуты родительского класса, при этом некоторые из них могут быть переопределены или добавлены в дочернем. Например, мы можем создать свой класс, похожий на [словарь](#):

```
class Mydict(dict):
    def get(self, key, default = 0):
        return dict.get(self, key, default)

a = dict(a=1, b=2)
b = Mydict(a=1, b=2)
```

Класс `Mydict` ведёт себя точно так же, как и словарь, за исключением того, что метод `get` по умолчанию возвращает не `None`, а `0`.

```
>>> b['c'] = 4
>>> print(b)
{'a': 1, 'c': 4, 'b': 2}
>>> print(a.get('v'))
None
>>> print(b.get('v'))
0
```

## Полиморфизм

Полиморфизм - разное поведение одного и того же метода в разных классах. Например, мы можем сложить два числа, и можем сложить две строки. При этом получим разный результат, так как числа и строки являются разными классами.

```
>>> 1 + 1
2
>>> "1" + "1"
'11'
```

## Перегрузка операторов

Перегрузка операторов — один из способов реализации полиморфизма, когда мы можем задать свою реализацию какого-либо метода в своём классе.

Например, у нас есть два класса:

```
class A:
    def go(self):
        print('Go, A!')

class B(A):
    def go(self, name):
        print('Go, {}'.format(name))
```

В данном примере класс B наследует класс A, но переопределяет метод go, поэтому он имеет мало общего с аналогичным методом класса A.

Однако в python имеются методы, которые, как правило, не вызываются напрямую, а вызываются встроенными функциями или операторами.

Например, метод `__init__` перегружает конструктор класса. Конструктор - создание экземпляра класса.

```
class A:
    def __init__(self, name):
        self.name = name

>>> a = A('Vasya')
>>> print(a.name)
Vasya
```

Собственно, далее пойдёт список таких "магических" методов.

`__new__(cls[, ...])` — управляет созданием экземпляра. В качестве обязательного аргумента принимает класс (не путать с экземпляром). Должен возвращать экземпляр класса для его последующей его передачи методу `__init__`.

**\_\_init\_\_**(self[, ...]) - как уже было сказано выше, конструктор.

**\_\_del\_\_**(self) - вызывается при удалении объекта сборщиком мусора.

**\_\_repr\_\_**(self) - вызывается встроенной функцией `repr`; возвращает "сырые" данные, используемые для внутреннего представления в python.

**\_\_str\_\_**(self) - вызывается функциями `str`, `print` и `format`. Возвращает строковое представление объекта.

**\_\_bytes\_\_**(self) - вызывается функцией `bytes` при преобразовании к байтам.

**\_\_format\_\_**(self, format\_spec) - используется функцией `format` (а также методом `format` у строк).

**\_\_lt\_\_**(self, other) -  $x < y$  вызывает `x.__lt__(y)`.

**\_\_le\_\_**(self, other) -  $x \leq y$  вызывает `x.__le__(y)`.

**\_\_eq\_\_**(self, other) -  $x == y$  вызывает `x.__eq__(y)`.

**\_\_ne\_\_**(self, other) -  $x != y$  вызывает `x.__ne__(y)`

**\_\_gt\_\_**(self, other) -  $x > y$  вызывает `x.__gt__(y)`.

**\_\_ge\_\_**(self, other) -  $x \geq y$  вызывает `x.__ge__(y)`.

**\_\_hash\_\_**(self) - получение хэш-суммы объекта, например, для добавления в словарь.

**\_\_bool\_\_**(self) - вызывается при проверке истинности. Если этот метод не определён, вызывается метод `__len__` (объекты, имеющие ненулевую длину, считаются истинными).

**\_\_getattr\_\_**(self, name) - вызывается, когда атрибут экземпляра класса не найден в обычных местах (например, у экземпляра нет метода с таким названием).

**\_\_setattr\_\_**(self, name, value) - назначение атрибута.

**\_\_delattr\_\_**(self, name) - удаление атрибута (`del obj.name`).

**\_\_call\_\_**(self[, args...]) - вызов экземпляра класса как функции.

**\_\_len\_\_**(self) - длина объекта.

**\_\_getitem\_\_**(self, key) - доступ по индексу (или ключу).

**\_\_setitem\_\_**(self, key, value) - назначение элемента по индексу.

**\_\_delitem\_\_**(self, key) - удаление элемента по индексу.

**\_\_iter\_\_**(self) - возвращает итератор для контейнера.

**\_\_reversed\_\_**(self) - итератор из элементов, следующих в обратном порядке.

**\_\_contains\_\_**(self, item) - проверка на принадлежность элемента контейнеру (item in self).

## Перегрузка арифметических операторов

**\_\_add\_\_**(self, other) - сложение.  $x + y$  вызывает  $x.__add__(y)$ .

**\_\_sub\_\_**(self, other) - вычитание ( $x - y$ ).

**\_\_mul\_\_**(self, other) - умножение ( $x * y$ ).

**\_\_truediv\_\_**(self, other) - деление ( $x / y$ ).

**\_\_floordiv\_\_**(self, other) - целочисленное деление ( $x // y$ ).

**\_\_mod\_\_**(self, other) - остаток от деления ( $x \% y$ ).

**\_\_divmod\_\_**(self, other) - частное и остаток ( $\text{divmod}(x, y)$ ).

**\_\_pow\_\_**(self, other[, modulo]) - возведение в степень ( $x ** y$ ,  $\text{pow}(x, y[, \text{modulo}])$ ).

**\_\_lshift\_\_**(self, other) - битовый сдвиг влево ( $x << y$ ).

**\_\_rshift\_\_**(self, other) - битовый сдвиг вправо ( $x >> y$ ).

**\_\_and\_\_**(self, other) - битовое И ( $x \& y$ ).

**\_\_xor\_\_**(self, other) - битовое ИСКЛЮЧАЮЩЕЕ ИЛИ ( $x \wedge y$ ).

**\_\_or\_\_**(self, other) - битовое ИЛИ ( $x | y$ ).

Пойдём дальше.



`__radd__(self, other),`  
`__rsub__(self, other),`  
`__rmul__(self, other),`  
`__rtruediv__(self, other),`  
`__rfloordiv__(self, other),`  
`__rmod__(self, other),`  
`__rdivmod__(self, other),`  
`__rpow__(self, other),`  
`__rlshift__(self, other),`  
`__rrshift__(self, other),`  
`__rand__(self, other),`  
`__rxor__(self, other),`

`__ror__(self, other)` - делают то же самое, что и арифметические операторы, перечисленные выше, но для аргументов, находящихся справа, и только в случае, если для левого операнда не определён соответствующий метод.

Например, операция  $x + y$  будет сначала пытаться вызвать `x.__add__(y)`, и только в том случае, если это не получилось, будет пытаться вызвать `y.__radd__(x)`. Аналогично для остальных методов.

Идём дальше.

`__iadd__(self, other)` -  $+=$ .  
`__isub__(self, other)` -  $-=$ .  
`__imul__(self, other)` -  $*=$ .  
`__itruediv__(self, other)` -  $/=$ .  
`__ifloordiv__(self, other)` -  $//=$ .

`__imod__(self, other)` - %=.

`__ipow__(self, other[, modulo])` - \*\*=.

`__ilshift__(self, other)` - <<=.

`__irshift__(self, other)` - >>=.

`__iand__(self, other)` - &=.

`__ixor__(self, other)` - ^=.

`__ior__(self, other)` - |=.

`__neg__(self)` - унарный -.

`__pos__(self)` - унарный +.

`__abs__(self)` - модуль (abs()).

`__invert__(self)` - инверсия (~).

`__complex__(self)` - приведение к complex.

`__int__(self)` - приведение к int.

`__float__(self)` - приведение к float.

`__round__(self[, n])` - округление.

`__enter__(self)`, `__exit__(self, exc_type, exc_value, traceback)` - реализация менеджеров контекста.

Рассмотрим некоторые из этих методов на примере двумерного вектора, для которого переопределим некоторые методы:

```
import math

class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
```

```

        return 'Vector2D({}, {})'.format(self.x, self.y)

    def __str__(self):
        return '({}, {})'.format(self.x, self.y)

    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y +
other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __sub__(self, other):
        return Vector2D(self.x - other.x, self.y -
other.y)

    def __isub__(self, other):
        self.x -= other.x
        self.y -= other.y
        return self

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return self.x != 0 and self.y != 0

    def __neg__(self):
        return Vector2D(-self.x, -self.y)

>>> x = Vector2D(3, 4)
>>> x
Vector2D(3, 4)
>>> print(x)
(3, 4)
>>> abs(x)
5.0
>>> y = Vector2D(5, 6)
>>> y
Vector2D(5, 6)

```

```

>>> x + y
Vector2D(8, 10)
>>> x - y
Vector2D(-2, -2)
>>> -x
Vector2D(-3, -4)
>>> x += y
>>> x
Vector2D(8, 10)
>>> bool(x)
True
>>> z = Vector2D(0, 0)
>>> bool(z)
False
>>> -z
Vector2D(0, 0)

```

В заключение хочу сказать, что перегрузка специальных методов - вещь хорошая, но не стоит ей слишком злоупотреблять. Перегружайте их только тогда, когда вы уверены в том, что это поможет пониманию программного кода.

## Декораторы

Итак, что же это такое? Для того, чтобы понять, как работают декораторы, в первую очередь следует вспомнить, что функции в python являются объектами, соответственно, их можно возвращать из другой функции или передавать в качестве аргумента. Также следует помнить, что функция в python может быть определена и внутри другой функции.

Вспомнив это, можно смело переходить к декораторам. **Декораторы** — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код.

Создадим свой декоратор "вручную":

```

# Декоратор - это функция, ожидающая ДРУГУЮ функцию в
# качестве параметра
def my_shiny_new_decorator(function_to_decorate):

```

```

    # Внутри себя декоратор определяет функцию-"обёртку".
    Она будет обернута вокруг декорируемой, получая
    возможность исполнять произвольный код до и после неё.
    def the_wrapper_around_the_original_function():
        print("Я - код, который отработает до вызова
функции")
        function_to_decorate() # Сама функция
        print("А я - код, срабатывающий после")
    # Теперь, вернём функцию-обёртку, которая содержит в
    себе декорируемую функцию, и код, который необходимо
    выполнить до и после.
    return the_wrapper_around_the_original_function

# Представим теперь, что у нас есть функция, которую мы
не планируем больше трогать.
def stand_alone_function():
    print("Я простая одинокая функция, ты ведь не
посмеешь меня изменять?")
stand_alone_function()
# выведет: Я простая одинокая функция, ты ведь не
посмеешь меня изменять?

# Однако, чтобы изменить её поведение, мы можем
декорировать её, то есть просто передать декоратору,
который обернет исходную функцию в любой код, который нам
потребуется, и вернёт новую, готовую к использованию
функцию:
stand_alone_function_decorated =
my_shiny_new_decorator(stand_alone_function)
stand_alone_function_decorated()
#выведет:
# Я - код, который отработает до вызова функции
# Я простая одинокая функция, ты ведь не посмеешь меня
изменять?
# А я - код, срабатывающий после

```

Наверное, теперь мы бы хотели, чтобы каждый раз, во время вызова `stand_alone_function`, вместо неё вызывалась `stand_alone_function_decorated`. Для этого просто перезапишем `stand_alone_function`:

```

stand_alone_function =
my_shiny_new_decorator(stand_alone_function)

```

```
stand_alone_function()  
#выведет:  
# Я - код, который отработает до вызова функции  
# Я простая одинокая функция, ты ведь не посмеешь меня  
изменить?..  
# А я - код, срабатывающий после
```

Собственно, это и есть декораторы. Вот так можно было записать предыдущий пример, используя синтаксис декораторов:

```
@my_shiny_new_decorator  
def another_stand_alone_function():  
    print("Оставь меня в покое")  
  
another_stand_alone_function()  
#выведет:  
# Я - код, который отработает до вызова функции  
# Оставь меня в покое  
# А я - код, срабатывающий после
```

То есть, декораторы в python — это просто синтаксический сахар для конструкций вида:

```
another_stand_alone_function =  
my_shiny_new_decorator(another_stand_alone_function)
```

При этом, естественно, можно использовать несколько декораторов для одной функции, на пример так:

```
def bread(func):  
    def wrapper():  
        print("")  
        func()  
        print("<\_____/>")  
    return wrapper  
  
def ingredients(func):  
    def wrapper():  
        print("#помидоры#")  
        func()  
        print("~салат~")  
    return wrapper
```

```

def sandwich(food="--ветчина--"):
    print(food)

sandwich()
#Выведет: --ветчина--
sandwich = bread(ingredients(sandwich))
sandwich()
#Выведет:
#
# #помидоры#
# --ветчина--
# ~салат~
# <\_____/>

```

И используя синтаксис декораторов:

```

@bread
@ingredients
def sandwich(food="--ветчина--"):
    print(food)

sandwich()
#Выведет:
#
# #помидоры#
# --ветчина--
# ~салат~
# <\_____/>

```

Также нужно помнить о том, что важен порядок декорирования. Сравните с предыдущим примером:

```

@ingredients
@bread
def sandwich(food="--ветчина--"):
    print(food)

sandwich()
#Выведет:
# #помидоры#
#
# --ветчина--

```

```
# <\_____/>
# ~салат~
```

## Передача декоратором аргументов в функцию

Однако, все декораторы, которые мы рассматривали, не имели одного очень важного функционала — передачи аргументов декорируемой функции. Собственно, это тоже несложно сделать.

```
def a_decorator_passing_arguments(function_to_decorate):
    def a_wrapper_accepting_arguments(arg1, arg2):
        print("Смотри, что я получил:", arg1, arg2)
        function_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments
```

# Теперь, когда мы вызываем функцию, которую возвращает декоратор, мы вызываем её "обёртку", передаём ей аргументы и уже в свою очередь она передаёт их декорируемой функции

```
@a_decorator_passing_arguments
def print_full_name(first_name, last_name):
    print("Меня зовут", first_name, last_name)
```

```
print_full_name("Vasya", "Pupkin")
# выведет:
# Смотри, что я получил: Vasya Pupkin
# Меня зовут Vasya Pupkin
```

## Декорирование методов

Один из важных фактов, которые следует понимать, заключается в том, что функции и методы в Python — это практически одно и то же, за исключением того, что методы всегда ожидают первым параметром ссылку на сам объект (self). Это значит, что мы можем создавать декораторы для методов точно так же, как и для функций, просто не забывая про self.

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie -= 3
        return method_to_decorate(self, lie)
    return wrapper
```



```

class Lucy:
    def __init__(self):
        self.age = 32

    @method_friendly_decorator
    def sayYourAge(self, lie):
        print("Мне {} лет, а ты бы сколько
дал?".format(self.age + lie))

l = Lucy()
l.sayYourAge(-3)
# выведет: Мне 26 лет, а ты бы сколько дал?

```

Конечно, если мы создаём максимально общий декоратор и хотим, чтобы его можно было применить к любой функции или методу, то можно воспользоваться распаковкой аргументов:

```

def
a_decorator_passing_arbitrary_arguments(function_to_decor
ate):
    # Данная "обёртка" принимает любые аргументы
    def a_wrapper_accepting_arbitrary_arguments(*args,
**kwargs):
        print("Передали ли мне что-нибудь?:")
        print(args)
        print(kwargs)
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments

@a_decorator_passing_arbitrary_arguments
def function_with_no_argument():
    print("Python is cool, no argument here.")

function_with_no_argument()
# выведет:
# Передали ли мне что-нибудь?:
# ()
# {}
# Python is cool, no argument here.

@a_decorator_passing_arbitrary_arguments
def function_with_arguments(a, b, c):

```

```

    print(a, b, c)

function_with_arguments(1, 2, 3)
# выведет:
# Передали ли мне что-нибудь?:
# (1, 2, 3)
# {}
# 1 2 3

@a_decorator_passing_arbitrary_arguments
def function_with_named_arguments(a, b, c,
platypus="Почему нет?"):
    print("Любят ли {}, {} и {} утконосов? {}".format(a,
b, c, platypus))

function_with_named_arguments("Билл", "Линус", "Стив",
platypus="Определенно!")
# выведет:
# Передали ли мне что-нибудь?:
# ('Билл', 'Линус', 'Стив')
# {'platypus': 'Определенно!'}
# Любят ли Билл, Линус и Стив утконосов? Определенно!

class Mary(object):
    def __init__(self):
        self.age = 31

    @a_decorator_passing_arbitrary_arguments
    def sayYourAge(self, lie=-3): # Теперь мы можем
указать значение по умолчанию
        print("Мне {} лет, а ты бы сколько
дал?".format(self.age + lie))

m = Mary()
m.sayYourAge()
# выведет:
# Передали ли мне что-нибудь?:
# (<__main__ .Mary object at 0xb7d303ac>,)
# {}
# Мне 28 лет, а ты бы сколько дал?

```

## Декораторы с аргументами

А теперь попробуем написать декоратор, принимающий аргументы:

```
def decorator_maker():
    print("Я создаю декораторы! Я буду вызван только раз:
    когда ты попросишь меня создать декоратор.")

    def my_decorator(func):
        print("Я - декоратор! Я буду вызван только раз: в
        момент декорирования функции.")

        def wrapped():
            print ("Я - обёртка вокруг декорируемой
            функции. "
                    "Я буду вызвана каждый раз когда ты
            вызываешь декорируемую функцию. "
                    "Я возвращаю результат работы
            декорируемой функции.")
            return func()

        print("Я возвращаю обёрнутую функцию.")
        return wrapped

    print("Я возвращаю декоратор.")
    return my_decorator

# Давайте теперь создадим декоратор. Это всего лишь ещё
# один вызов функции
new_decorator = decorator_maker()
# выведет:
# Я создаю декораторы! Я буду вызван только раз: когда ты
# попросишь меня создать тебе декоратор.
# Я возвращаю декоратор.

# Теперь декорируем функцию
def decorated_function():
    print("Я - декорируемая функция.")

decorated_function = new_decorator(decorated_function)
# выведет:
```

```
# Я - декоратор! Я буду вызван только раз: в момент
декорирования функции.
# Я возвращаю обёрнутую функцию.

# Теперь наконец вызовем функцию:
decorated_function()
# выведет:
# Я - обёртка вокруг декорируемой функции. Я буду вызвана
каждый раз когда ты вызываешь декорируемую функцию.
# Я возвращаю результат работы декорируемой функции.
# Я - декорируемая функция.
```

Теперь перепишем данный код с помощью декораторов:

```
@decorator_maker()
def decorated_function():
    print("Я - декорируемая функция.")
# выведет:
# Я создаю декораторы! Я буду вызван только раз: когда ты
попросишь меня создать тебе декоратор.
# Я возвращаю декоратор.
# Я - декоратор! Я буду вызван только раз: в момент
декорирования функции.
# Я возвращаю обёрнутую функцию.

decorated_function()
# выведет:
# Я - обёртка вокруг декорируемой функции. Я буду вызвана
каждый раз когда ты вызываешь декорируемую функцию.
# Я возвращаю результат работы декорируемой функции.
# Я - декорируемая функция.
```

Вернёмся к аргументам декораторов, ведь, если мы используем функцию, чтобы создавать декораторы "на лету", мы можем передавать ей любые аргументы, верно?

```
def decorator_maker_with_arguments(decorator_arg1,
decorator_arg2):

    print("Я создаю декораторы! И я получил следующие
аргументы:", decorator_arg1, decorator_arg2)
```

```

def my_decorator(func):
    print("Я - декоратор. И ты всё же смог передать
мне эти аргументы:", decorator_arg1, decorator_arg2)

    # Не перепутайте аргументы декораторов с
аргументами функций!
    def wrapped(function_arg1, function_arg2) :
        print ("Я - обёртка вокруг декорируемой
функции.\n"
               "И я имею доступ ко всем аргументам:
\n"
               "\t- и декоратора: {0} {1}\n"
               "\t- и функции: {2} {3}\n"
               "Теперь я могу передать нужные
аргументы дальше"
               .format(decorator_arg1, decorator_arg2,
                       function_arg1, function_arg2))
        return func(function_arg1, function_arg2)

    return wrapped

return my_decorator

@decorator_maker_with_arguments("Леонард", "Шелдон")
def decorated_function_with_arguments(function_arg1,
function_arg2):
    print ("Я - декорируемая функция и я знаю только о
своих аргументах: {0}"
           " {1}".format(function_arg1, function_arg2))

decorated_function_with_arguments("Раджеш", "Говард")
# выведет:
# Я создаю декораторы! И я получил следующие аргументы:
Леонард Шелдон
# Я - декоратор. И ты всё же смог передать мне эти
аргументы: Леонард Шелдон
# Я - обёртка вокруг декорируемой функции.
# И я имею доступ ко всем аргументам:
#   - и декоратора: Леонард Шелдон
#   - и функции: Раджеш Говард
# Теперь я могу передать нужные аргументы дальше

```

**# Я - декорируемая функция и я знаю только о своих аргументах: Раджеш Говард**

Таким образом, мы можем передавать декоратору любые аргументы, как обычной функции. Мы можем использовать и распаковку через `*args` и `**kwargs` в случае необходимости.

### **Некоторые особенности работы с декораторами**

- Декораторы несколько замедляют вызов функции, не забывайте об этом.
- Вы не можете "раздекорировать" функцию. Безусловно, существуют трюки, позволяющие создать декоратор, который можно отсоединить от функции, но это плохая практика. Правильнее будет запомнить, что если функция декорирована — это не отменить.
- Декораторы оборачивают функции, что может затруднить отладку.

Последняя проблема частично решена добавлением в модуле `functools` функции `functools.wraps`, копирующей всю информацию об оборачиваемой функции (её имя, из какого она модуля, её документацию и т.п.) в функцию-обёртку.

Забавным фактом является то, что `functools.wraps` тоже является декоратором.

```
def foo():
    print("foo")

print(foo.__name__)
# выведет: foo

# Однако, декораторы мешают нормальному ходу дел:
def bar(func):
    def wrapper():
        print("bar")
        return func()
    return wrapper

@bar
def foo():
    print("foo")

print(foo.__name__)
# выведет: wrapper
```

```

import functools # "functools" может нам с этим помочь

def bar(func):
    # Объявляем "wrapper" оборачивающим "func"
    # и запускаем магию:
    @functools.wraps(func)
    def wrapper():
        print("bar")
        return func()
    return wrapper

@bar
def foo():
    print("foo")

print(foo.__name__)
# выведет: foo

```

## Примеры использования декораторов

Декораторы могут быть использованы для расширения возможностей функций из сторонних библиотек (код которых мы не можем изменять), или для упрощения отладки (мы не хотим изменять код, который ещё не устоялся).

Также полезно использовать декораторы для расширения различных функций одним и тем же кодом, без повторного его переписывания каждый раз, например:

```

def benchmark(func):
    """
    Декоратор, выводящий время, которое заняло
    выполнение декорируемой функции.
    """
    import time
    def wrapper(*args, **kwargs):
        t = time.clock()
        res = func(*args, **kwargs)
        print(func.__name__, time.clock() - t)
        return res
    return wrapper

```

```

def logging(func):
    """
    Декоратор, логирующий работу кода.
    (хорошо, он просто выводит вызовы, но тут могло быть
    и логирование!)
    """
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        print(func.__name__, args, kwargs)
        return res
    return wrapper

def counter(func):
    """
    Декоратор, считающий и выводящий количество вызовов
    декорируемой функции.
    """
    def wrapper(*args, **kwargs):
        wrapper.count += 1
        res = func(*args, **kwargs)
        print("{0} была вызвана:
{1}x".format(func.__name__, wrapper.count))
        return res
    wrapper.count = 0
    return wrapper

@benchmark
@logging
@counter
def reverse_string(string):
    return ''.join(reversed(string))

print(reverse_string("А поза упала на лапу Азора"))
print(reverse_string("A man, a plan, a canoe, pasta,
heros, rajahs, a coloratura, maps, snipe, percale,
macaroni, a gag, a banana bag, a tan, a tag, a banana bag
again (or a camel), a crepe, pins, Spam, a rut, a Rolo,
cash, a jar, sore hats, a peon, a canal: Panama!"))

# выведет:
# reverse_string ('А роза упала на лапу Азора',) {}
# wrapper 0.0

```



```
# reverse_string была вызвана: 1x
# арозА упал ан алапу азор А
# reverse_string ('A man, a plan, a canoe, pasta, heros,
rajahs, a coloratura, maps, snipe, percale, macaroni, a
gag, a banana bag, a tan, a tag, a banana bag again (or a
camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar,
sore hats, a peon, a canal: Panama!',) {}
# wrapper 0.0
# reverse_string была вызвана: 2x
# !amanaP :lanac a ,noep a ,stah eros ,raj a ,hsac ,oloR
a ,tur a ,mapS ,snip ,eperc a ,)lemac a ro( niaga gab
ananab a ,gat a ,nat a ,gab ananab a ,gag a ,inoracam
,elacrep ,epins ,spam ,arutaroloc a ,shajar ,soreh ,atsap
,eonac a ,nalp a ,nam A
```

## Python2 vs Python3: различия синтаксиса

### Print - функция

Оператор print был заменён функцией print(), с именованными аргументами для замены большей части синтаксиса старого оператора print. Примеры:

```
Python2: print "The answer is", 2*2
Python3: print("The answer is", 2*2)
```

```
Python2: print x,          # Запятая в конце подавляет
перевод строки
Python3: print(x, end=" ") # Добавляет пробел вместо
перевода строки
```

```
Python2: print             # Печатает перевод строки
Python3: print()           # Нужно вызвать функцию!
```

```
Python2: print >>sys.stderr, "fatal error"
Python3: print("fatal error", file=sys.stderr)
```

```
Python2: print (x, y)      # Печатает repr((x, y))
Python3: print((x, y))    # Не путать с print(x, y)!
```

Также вы можете настроить разделитель между элементами, например:

```
>>> print("There are <", 2**32, "> possibilities!",  
sep="")  
There are <4294967296> possibilities!
```

Функция `print()` не поддерживает особенность "программный пробел" старого оператора `print`. Например, в Python 2, `print "A\n", "B"` напечатает "A\nB\n"; но в Python 3, `print("A\n", "B")` напечатает "A\n B\n".

## Отображения и итераторы вместо списков

Некоторые хорошо известные методы не возвращают списки в Python 3:

- Методы словарей `dict.keys()`, `dict.items()` и `dict.values()` возвращают "отображения" вместо списков. Например, больше не работает: `k = d.keys(); k.sort()`. Используйте `k = sorted(d)`.
- Соответственно, методы `dict.iterkeys()`, `dict.iteritems()` и `dict.itervalues()` более не поддерживаются.
- `map()` и `filter()` возвращают итераторы. Если вам действительно нужен список, быстрым исправлением станет `list(map(...))`, но часто лучшим исправлением будет использование генераторов списков (особенно когда оригинальный код использует лямбда-выражения), либо переписать код так, чтобы ему не нужен был список как таковой. Особенно сложно, что `map()` вызывает побочные эффекты функции; правильное преобразование заключается в использовании цикла (создание списка просто расточительно).
- `range()` теперь ведёт себя как `xrange()`, но работает со значениями любого размера. `xrange()` больше не существует.
- `zip()` возвращает итератор.

## Операторы сравнения

Python 3 упростил правила для операторов сравнения:

Операторы сравнения (`<`, `<=`, `>=`, `>`) поднимают исключение `TypeError`, когда операнды не упорядочиваемы. Таким образом, выражения типа `1 < "`, `0 > None` или `len <= len` более не разрешены, и, например, `None < None` поднимает `TypeError`, а не возвращает `False`. Следствием является то, что сортировка списка с разными типами данных больше не имеет смысла - все элементы должны быть сравнимы друг с другом. Обратите внимание, что это не относится к операторам `==` и `!=`: объекты различных несравнимых типов всегда неравны друг другу.

`builtin.sorted()` и `list.sort()` больше не принимают аргумент `cmp`, обеспечивающий функцию сравнения. Вместо этого используйте аргумент `key`. Аргументы `key` и `reverse` теперь "keyword-only".

Функция `cmp()` должна рассматриваться как устаревшая, и специальный метод `__cmp__()` в Python 3 не поддерживается. Используйте `__lt__()` для сортировки, `__eq__()` с `__hash__()` для сравнения. (Если вам действительно нужна функциональность `cmp()`, вы можете использовать выражение `(a > b) - (a < b)` в качестве эквивалента для `cmp(a, b)`.)

## Целые числа

- PEP 0237: `long` переименован в `int`.
- PEP 0238: Выражение вида `1/2` возвращает `float`. Используйте `1//2` для отсечения дробной части. (Этот синтаксис существует с Python 2.2)
- Константа `sys.maxint` была удалена, с тех пор, как более не существует предела значения целых чисел. Однако, `sys.maxsize` может быть использован как число, большее любого практического индекса списка или строки. `sys.maxsize` соответствует "естественному" размеру целого и, как правило, имеет такое же значение, как `sys.maxint` на той же платформе (при условии одних и те же параметров сборки).
- `repr()` от длинного целого числа не включает более завершающий символ `L`, так что код, который безусловно отрезает этот символ, будет отрезать вместо этого последнюю цифру. (Используйте `str()` вместо этого.)
- Восьмеричные литералы более не имеют формы вида `0720`; используйте `0o720`.

## Текст, Unicode и 8-битные строки

Все, что вы знали о бинарных данных и Unicode, изменилось.

Python 3 использует понятия текста и (бинарных) данных вместо строк Unicode и 8-битных строк. Весь текст - Unicode; Однако кодированные Unicode строки представлены в виде двоичных данных. Тип, используемый для хранения текста является `str`, тип, используемый для хранения данных - `bytes`. Самое большое различие с python 2.x является то, что любая попытка комбинировать текст и данные в Python 3.0 поднимает `TypeError`, в то время как если бы вы смешивали Unicode и 8-битные строки в Python 2.x, это будет работать, если 8-битная строка содержала только 7-битные (ASCII) символы, но вы получите

UnicodeDecodeError, если она содержит не-ASCII символы. Такое поведение вызывало многочисленные скорбные лица на протяжении многих лет.

Как следствие этого изменения в философии, значительная часть кода, который использует Unicode, кодировки или бинарные данные, скорее всего, должна измениться. Это изменения к лучшему, так как в python 2.x были многочисленные ошибки, имеющие отношение к смешиванию закодированного и декодированного текста. Чтобы быть подготовленным к этому, в Python 2.x следует начать использовать Unicode для всего незакодированного текста, и str только для бинарных или закодированных данных. Затем инструмент 2to3 будет делать большую часть работы за вас.

Вы можете больше не использовать литерал u"..." для текста Unicode. Тем не менее, вы должны использовать литерал b"..." для бинарных данных.

Так как str и bytes не могут быть смешаны, вы всегда должны их явно преобразовывать. Используйте str.encode(), чтобы перейти от str к bytes и bytes.decode(), чтобы перейти от bytes к str. Вы также можете использовать bytes(s, encoding=...) и str(b, encoding=...), соответственно.

Как str, тип bytes неизменен. Существует отдельный изменяемый тип для двоичных данных, bytearray. Почти все функции, которые принимают bytes также принимают bytearray.

Все обратные косые черты в "сырых" строковых литералах интерпретируются буквально. Это означает, что "\U" и "\u" в сырых строках не рассматриваются особо. Например, r"\u20ac" это строка из 6 символов в Python 3.0, в то время как в 2.6, ur"\u20ac" был один символ "евро". (Конечно, это изменение влияет только на сырые строковые литералы).

Встроенный абстрактный тип basestring был удален. Используйте str вместо него. str и bytes не имеют достаточно общей функциональности, чтобы оправдать общий базовый класс. Инструмент 2to3 (см. ниже) заменяет каждое вхождение basestring на str.

PEP 3138: repr() для строки больше не экранирует символы, не входящие в набор ASCII. Однако, он по-прежнему экранирует управляющие символы

PEP 3120: Кодировка исходного кода по умолчанию теперь UTF-8.

PEP 3131: не-ASCII символы разрешены в идентификаторах. (Тем не менее, стандартная библиотека остается ASCII, за исключением имен авторов в комментариях.)

Модули StringIO и cStringIO удалены. Вместо этого, импортируйте модуль io и используйте io.StringIO или io.BytesIO для текста и данных соответственно.

## Обзор изменений синтаксиса

Этот раздел дает краткий обзор каждого синтаксического изменения Python 3.0.

### Новый синтаксис

PEP 3107: аннотации для аргументов функции и возвращаемых значений.

```
>>> def foo(a: 'x', b: 5 + 6, c: list) -> max(2, 9):  
...     pass  
>>> foo.__annotations__  
{'a': 'x', 'b': 11, 'c': <class 'list'>, 'return': 9}
```

PEP 3102: Keyword-only аргументы.

PEP 3104: nonlocal. Переменная во внешней (но не глобальной) области видимости.

```
>>> def outer():  
...     x = 1  
...     def inner():  
...         x = 2  
...         print("inner:", x)  
...     inner()  
...     print("outer:", x)
```

```
>>> outer()  
inner: 2  
outer: 1
```

```
>>> def outer():  
...     x = 1  
...     def inner():  
...         nonlocal x  
...         x = 2  
...         print("inner:", x)  
...     inner()  
...     print("outer:", x)
```

```
>>> outer()
inner: 2
outer: 2
```

PEP 3132: Extended Iterable Unpacking

```
>>> (a, *rest, b) = range(5)
>>> a
0
>>> rest
[1, 2, 3]
>>> b
4
```

Генераторы словарей: {k: v for k, v in stuff} (то же самое, что и dict(stuff))

Литералы множеств (например, {1, 2}). Заметьте, что {} - это пустой словарь. Используйте set() для пустых множеств. Генераторы множеств: {x for x in stuff}

Новые восьмеричные литералы, например 0o720, вместо старых (0720).

Новые двоичные литералы, например 0b1010. Новая встроенная функция, bin().

## Изменённый синтаксис

PEP 3109 and PEP 3134: новый синтаксис выражения raise: raise [expr [from expr]].

"as" и "with" зарезервированные слова.

"True" и "False" и "None" - зарезервированные слова.

Изменено "except exc, var" на "except exc as var".

PEP 3115: Новый синтаксис для метаклассов. Вместо:

```
class C:
    __metaclass__ = M
    ...
```

Вы должны использовать:

```
class C(metaclass=M):
```

```
...
```

Переменная `__metaclass__` более не поддерживается.

Генераторы списков больше не поддерживают синтаксическую форму `[... for var in item1, item2, ...]`. Используйте `[... for var in (item1, item2, ...)]`.

## Удаленный синтаксис

PEP 3113: распаковка кортежей в параметрах удалена. Вы больше не можете писать

```
def foo(a, (b, c)):  
    ...
```

Пишите

```
def foo(a, b_c):  
    b, c = b_c  
    ...
```

Удалены обратные кавычки (backtick). Используйте `repr()`.

Удалено `<>`. Используйте `!=`.

`exec` - функция. Перестала быть зарезервированным словом.

`from module import *` запрещено использовать внутри функций.

## Встроенные функции

PEP 3135: Новый `super()`. Теперь вы можете вызывать `super()` без аргументов и (при условии, что это метод экземпляра, определенный внутри определении класса) класс и экземпляр будут автоматически выбраны. С аргументами, поведение `super()` остается без изменений.

PEP 3111: `raw_input()` переименован в `input()`. Вместо `input()` в Python 2, вы можете использовать `eval(input())`.

Добавлена функция `next()`, вызывающая метод `__next__()` объекта.

Перемещен `intern()` в `sys.intern()`.

Удалено: `apply()`. Вместо `apply(f, args)` используйте `f(*args)`.

Удалено: `callable()`. Вместо `callable(f)` используйте `hasattr(f, "__call__")`. Функция `operator.isCallable()` также удалена.

Удалено: `coerce()`.

Удалено: `execfile()`. Вместо `execfile(fn)` используйте `exec(open(fn).read())`.

Удалено: `file`. Используйте `open()`.

Перемещено: `reduce()` в `functools.reduce()`

Перемещено: `reload()` в `imp.reload()`.

Удалено: `dict.has_key()`. Используйте оператор `in`.

## **Введение в Python с PyCharm Educational Edition**

Компания JetBrains сделала еще один шаг навстречу образованию, выпустив PyCharm Educational Edition, образовательную версию среды разработки для Python, которая включает в себя возможность создавать интерактивные курсы.

Также PyCharm Educational Edition содержит предустановленный курс "Introduction to Python", который хорошо подходит для тех, кто начинает изучать Python.

Разумеется, я не мог его обойти стороной, и сегодня мы будем вместе его проходить, а заодно и познакомимся с этим замечательным редактором.

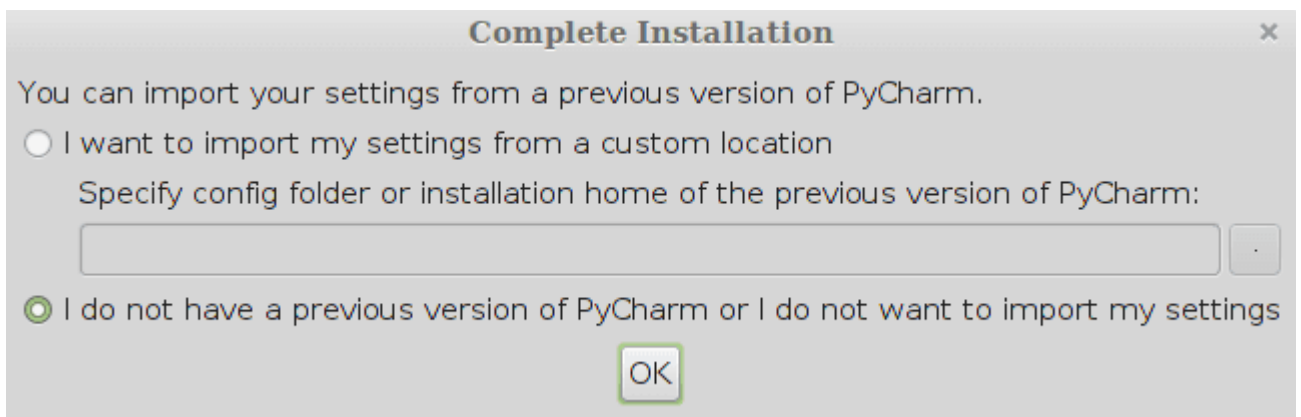




## Установка PyCharm Educational Edition

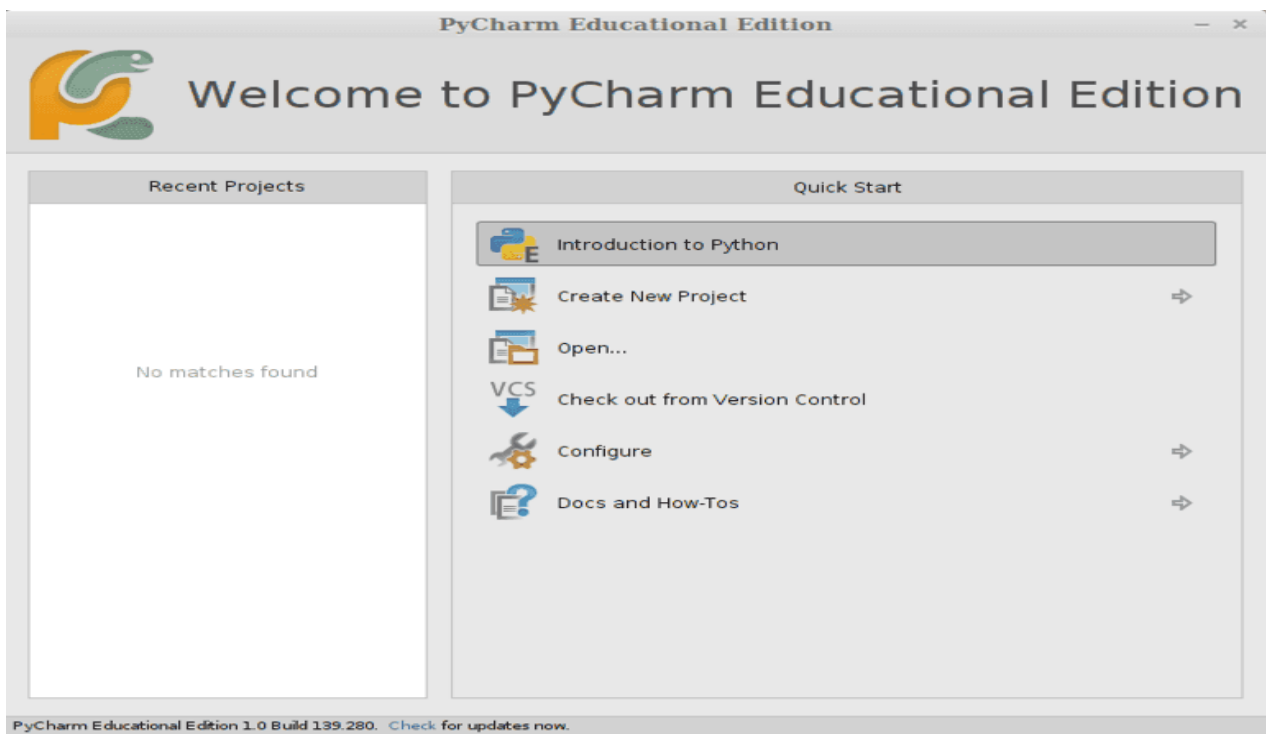
Устанавливаем [отсюда](#).

После установки вас спросят, не хотите ли вы импортировать настройки из других версий PyCharm.



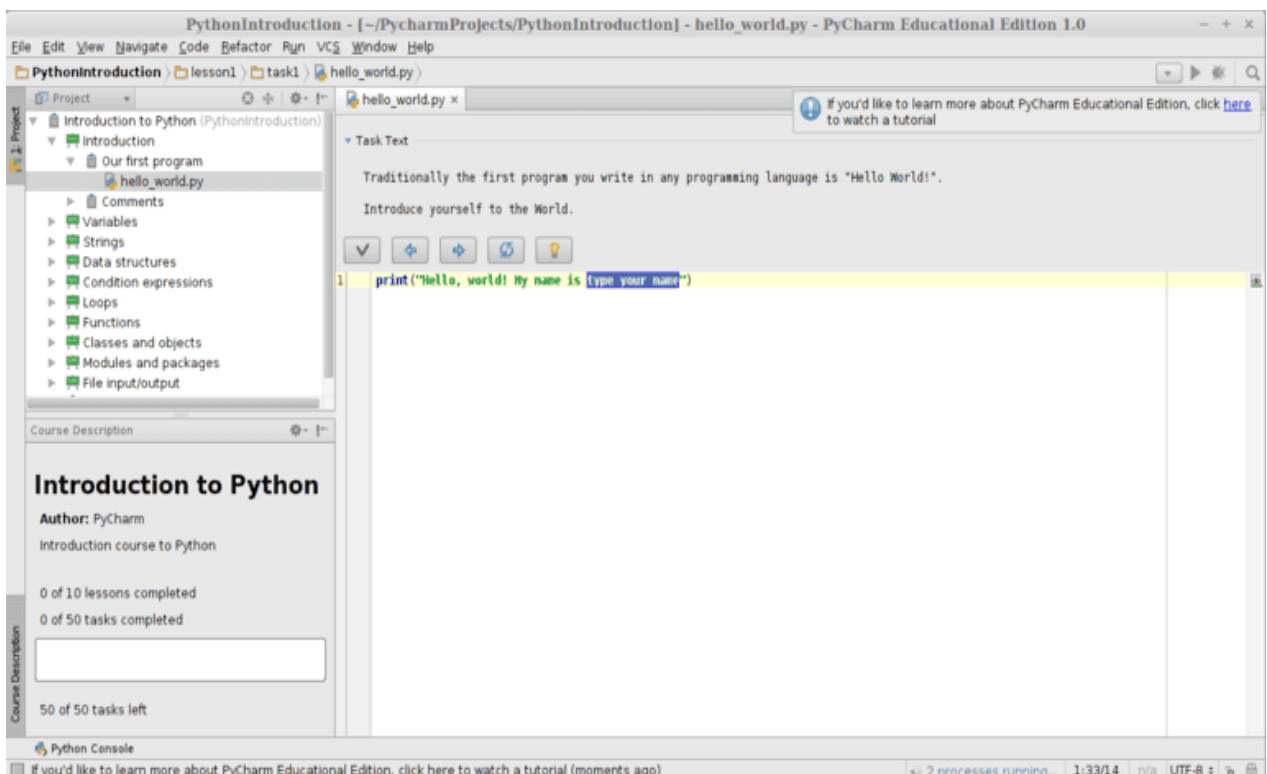
Так как у вас (скорее всего) не было других версий PyCharm, оставляете как есть.

После запуска PyCharm вас встретит таким вот окном:

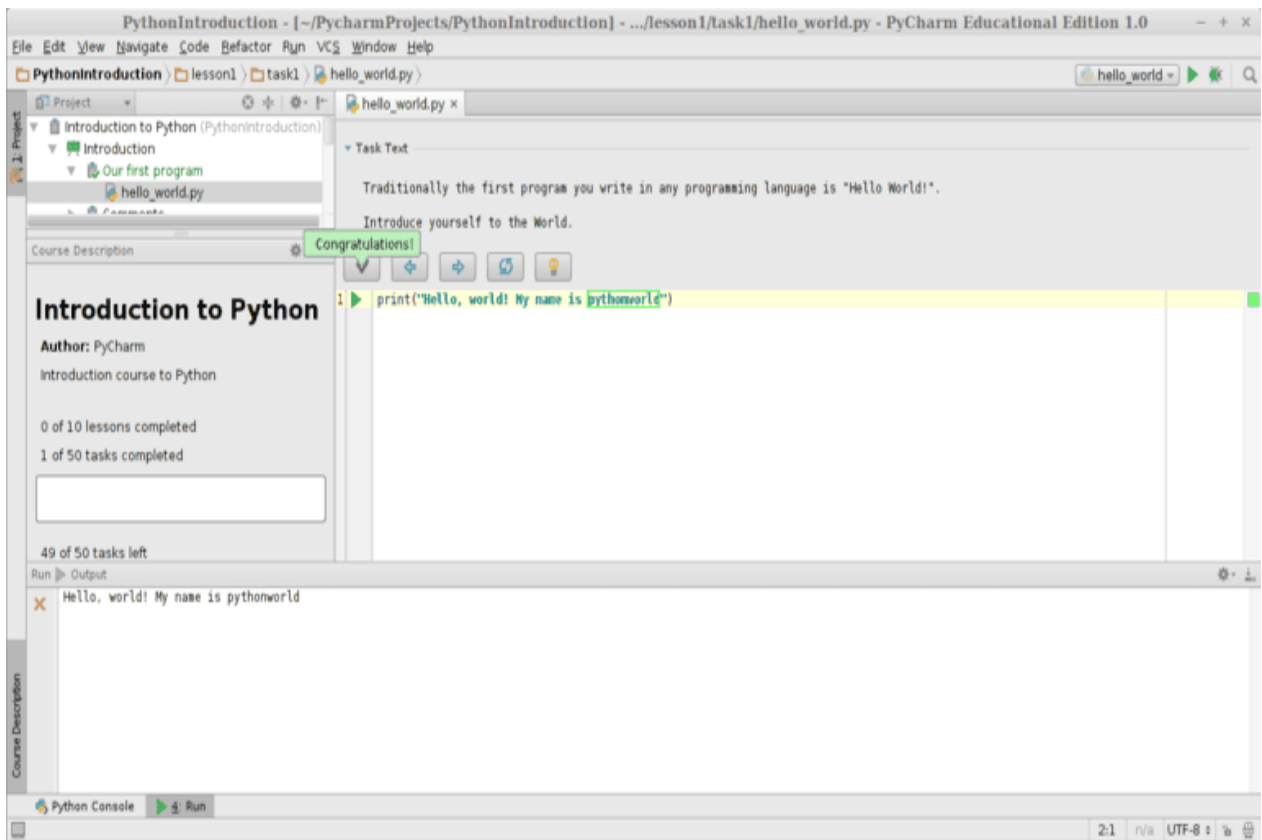


Выбираем "Introduction to Python". И начинаем!

## Задание 1: Hello World



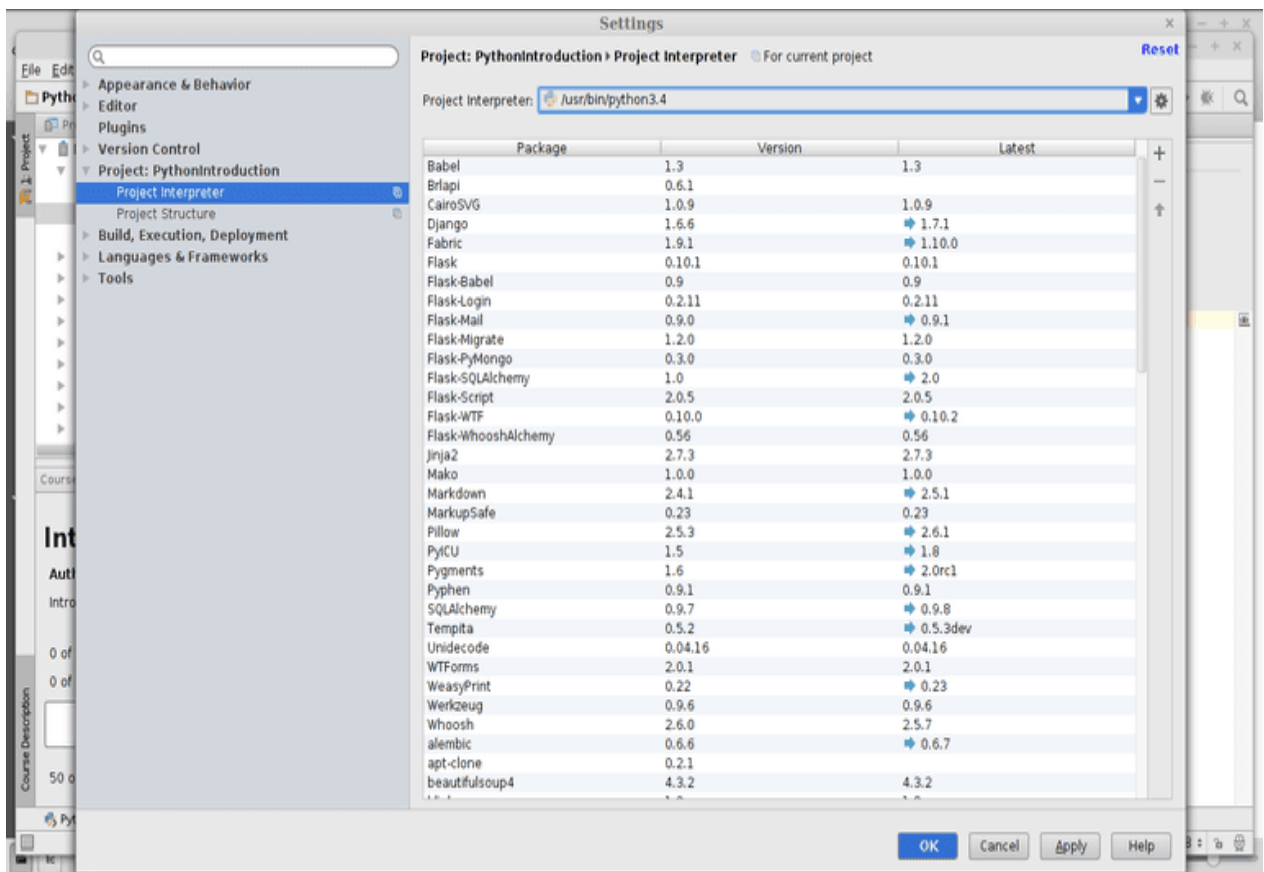
Вводное задание. Печатаем своё имя (или не своё), нажимаете галочку, получаете поздравления, и переходим к следующему заданию с помощью стрелки вправо.



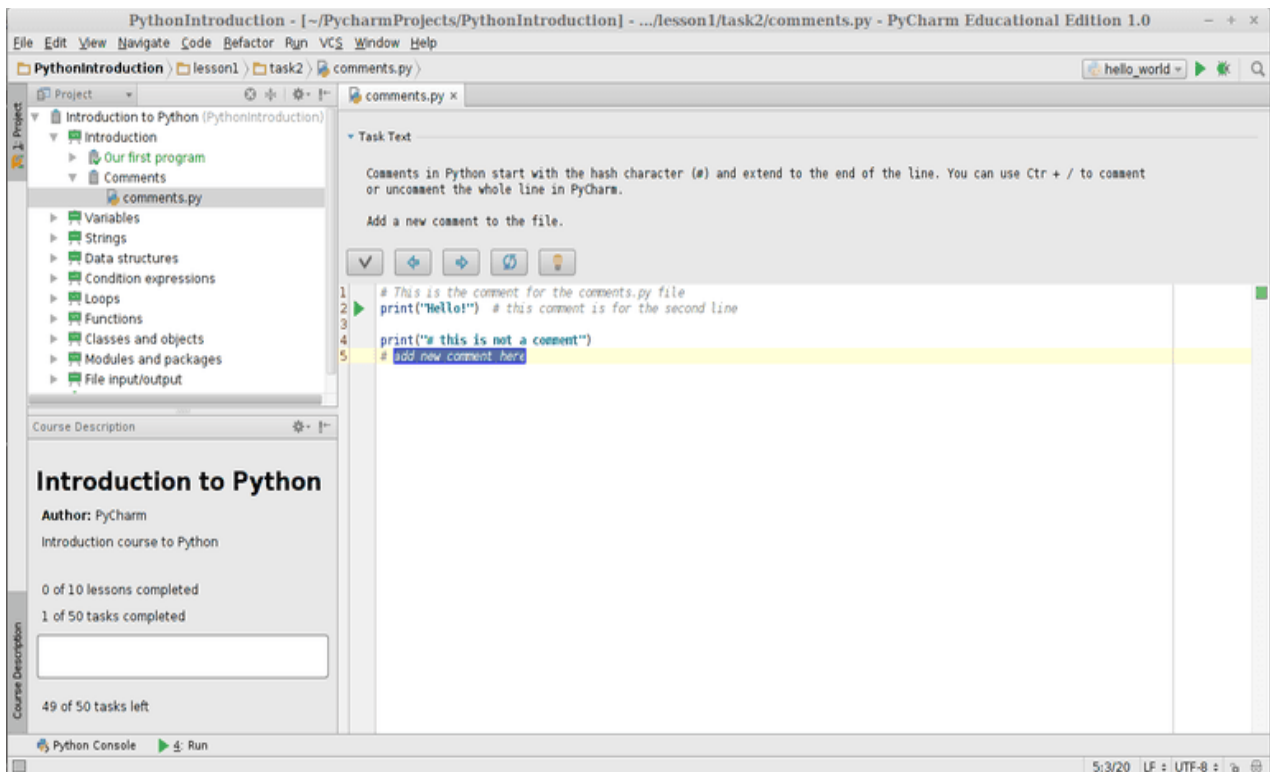
## Смена интерпретатора

Может так случиться (особенно если вы сидите на linux), что PyCharm по умолчанию поставит интерпретатор Python 2.

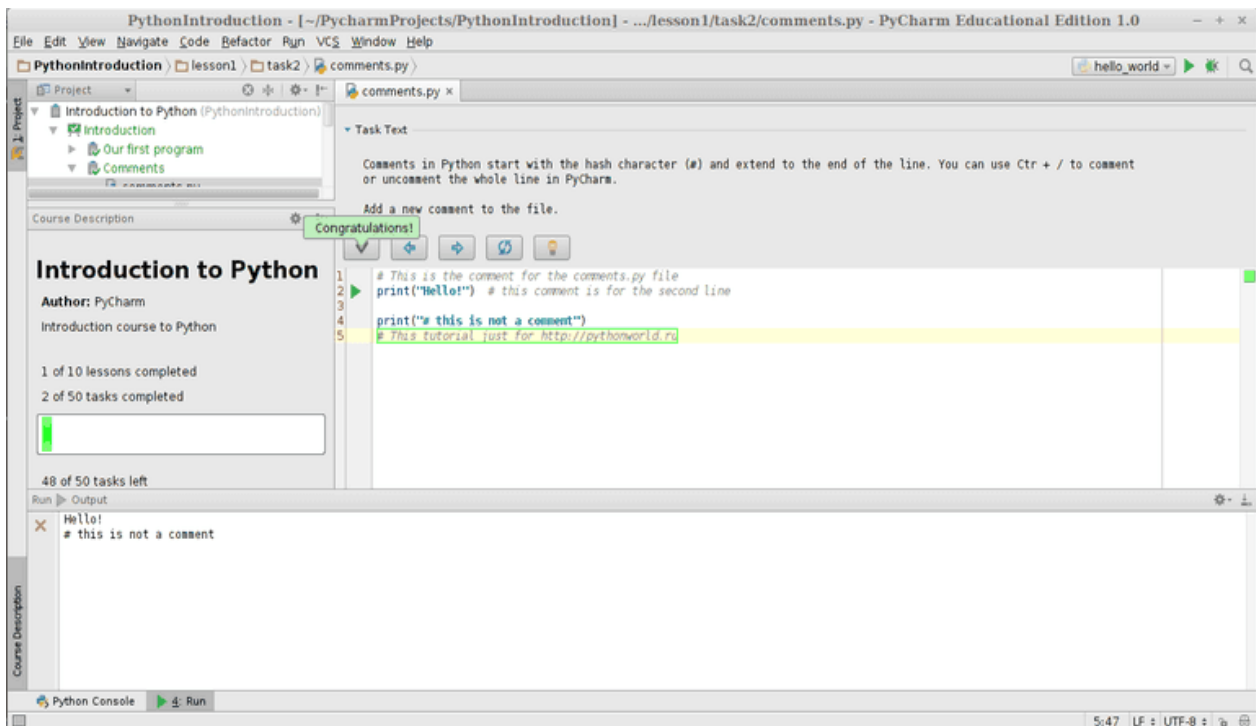
Поэтому проверьте, и при необходимости поменяйте интерпретатор (File → Settings → Project Interpreter)



## Задание 2: Комментарии

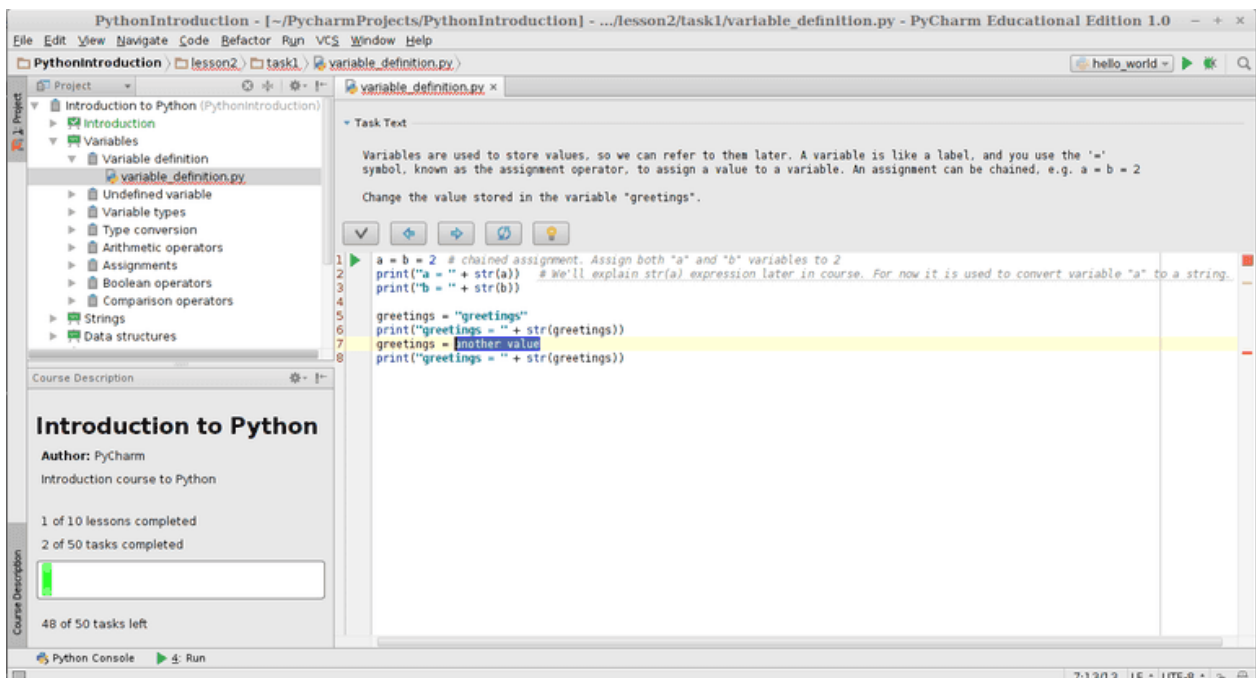


Комментарии. Достаточно написать любой комментарий. Кстати, в PyCharm нажатием "Ctrl и /" можно закомментировать / раскомментировать любую строку.

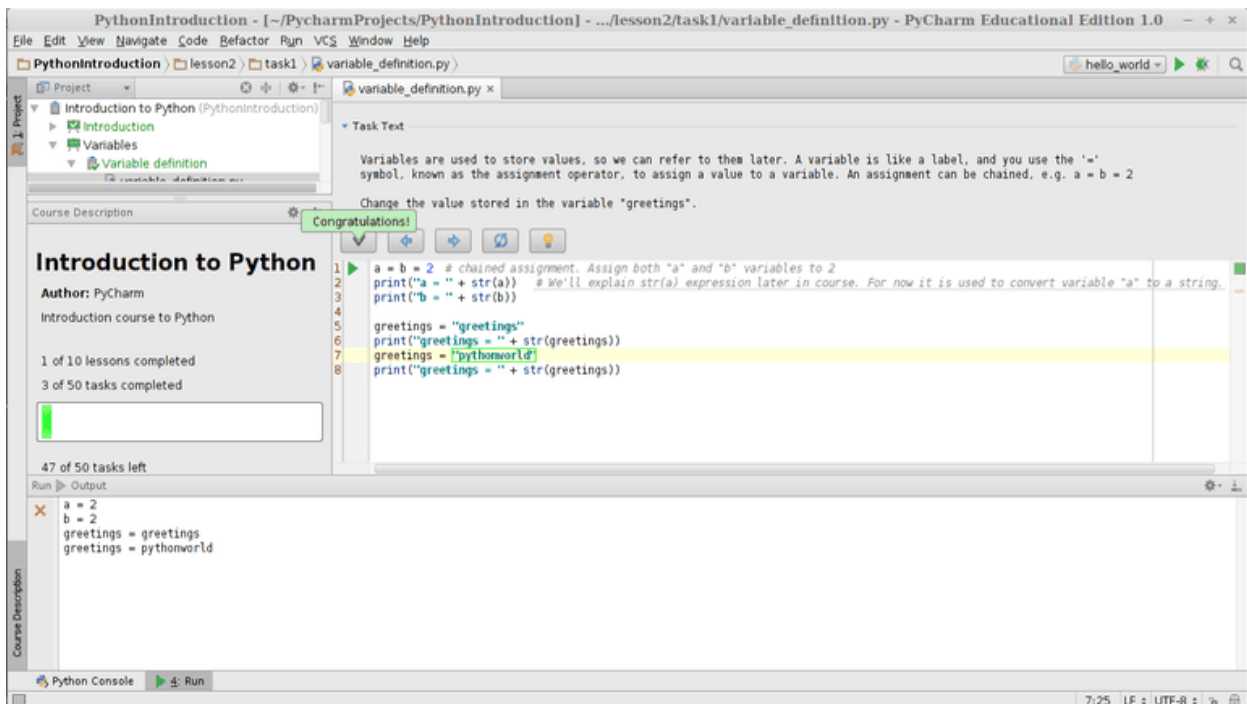


## Задание 3: Переменные

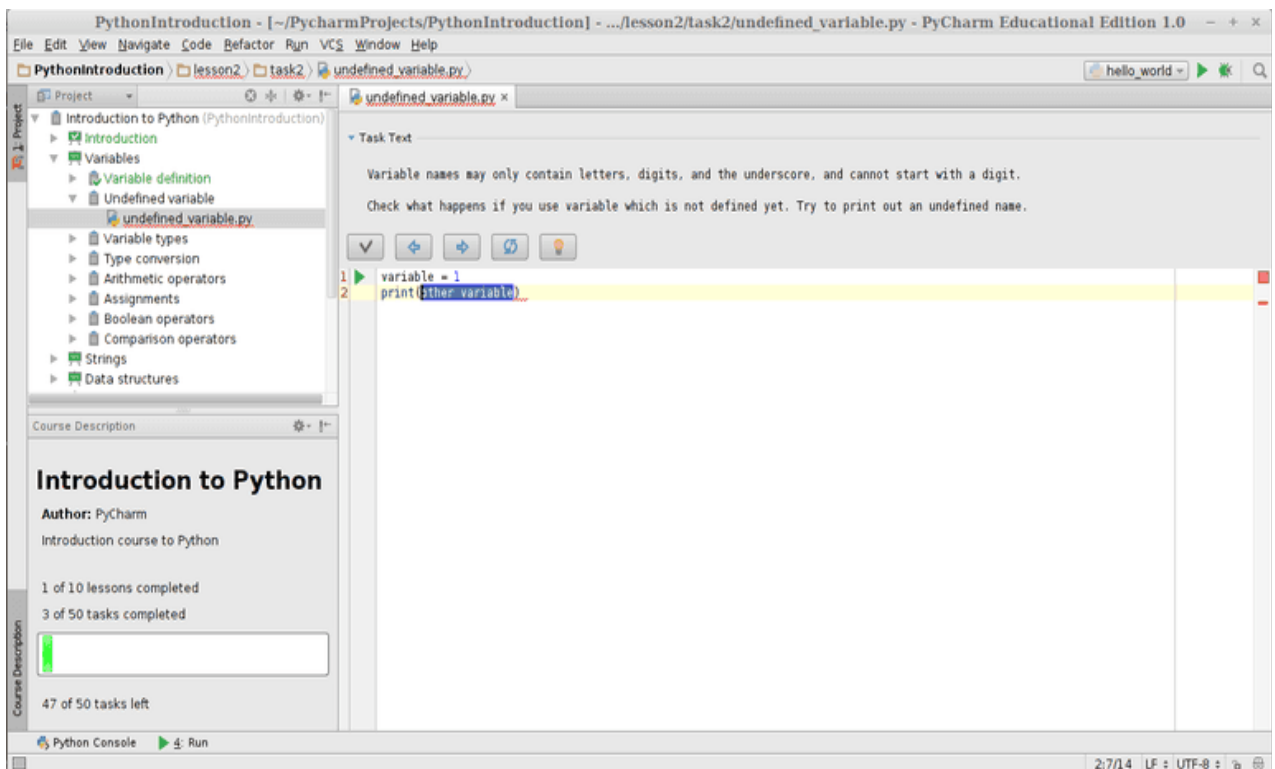
Переменные в Python являются ссылками на объект.



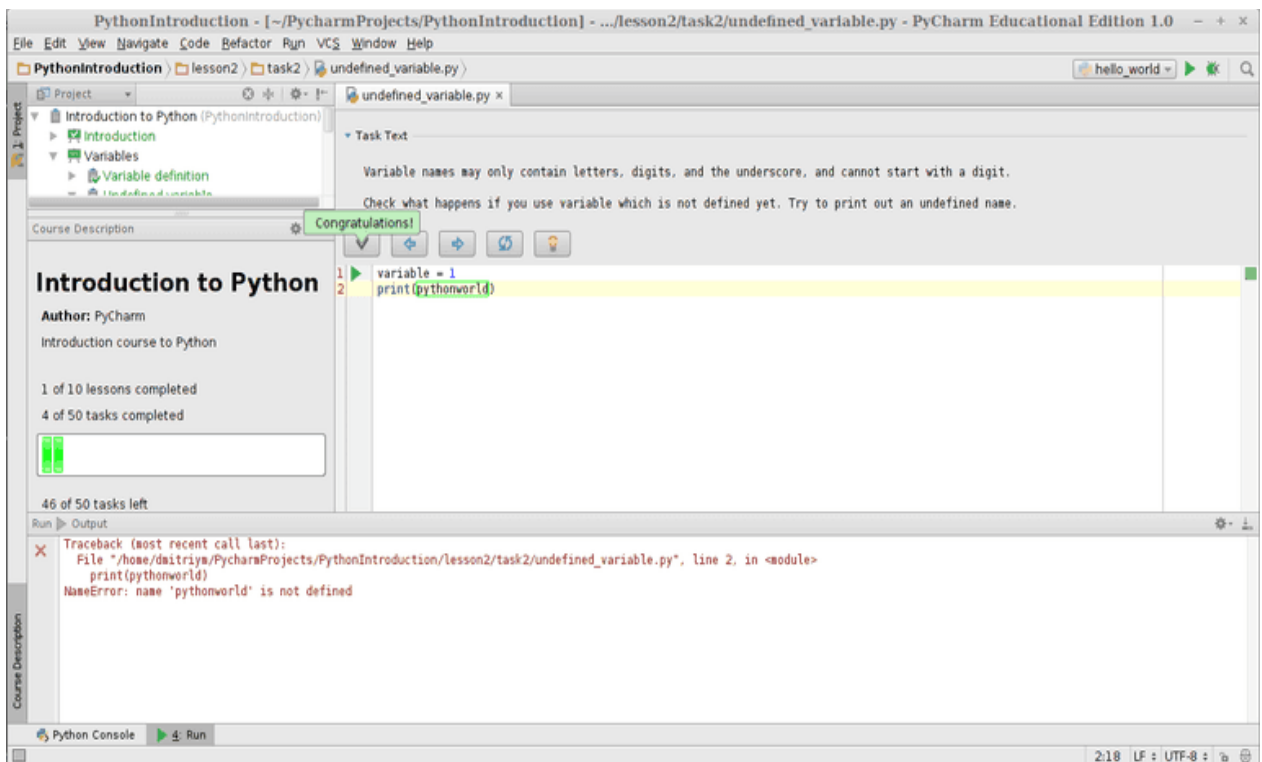
Нужно изменить значение переменной greetings.



## Задание 4: Несуществующие переменные



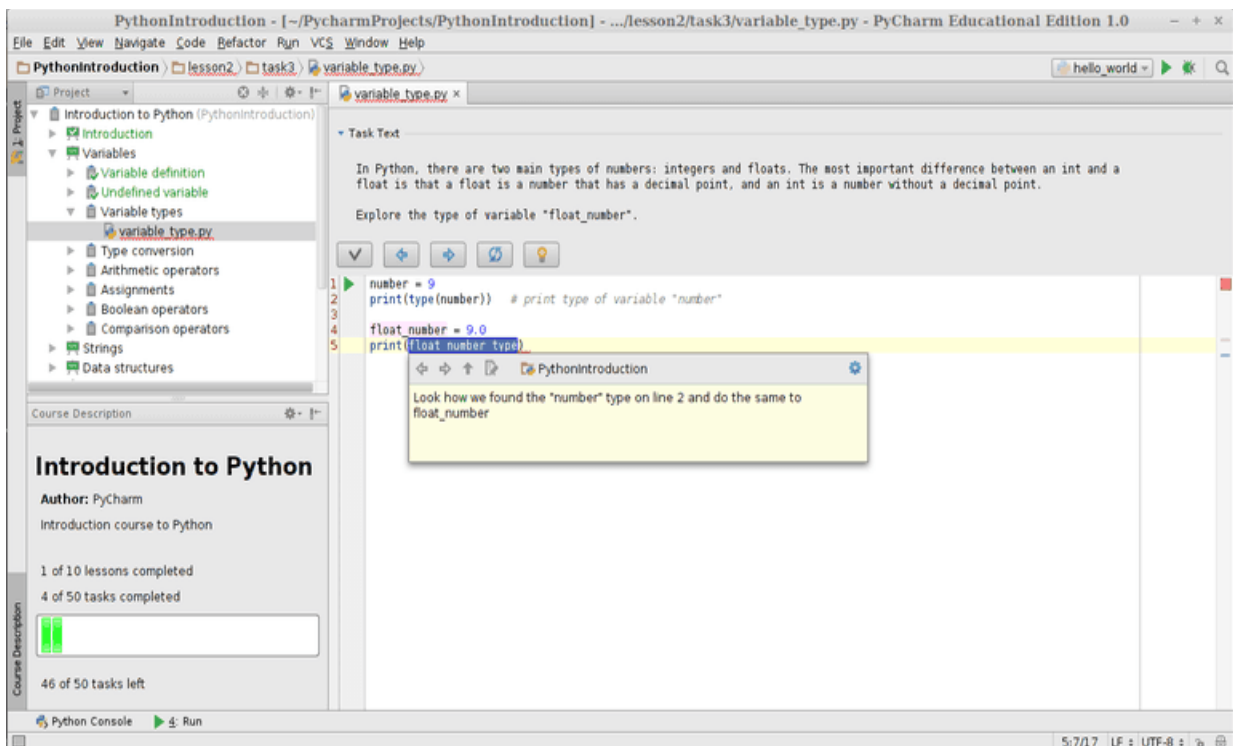
Нужно попытаться вывести несуществующую переменную.



Как видите, вызвалось исключение `NameError`.

## Задание 5: тип переменной

У каждого объекта есть тип (например, `int` или `float`).

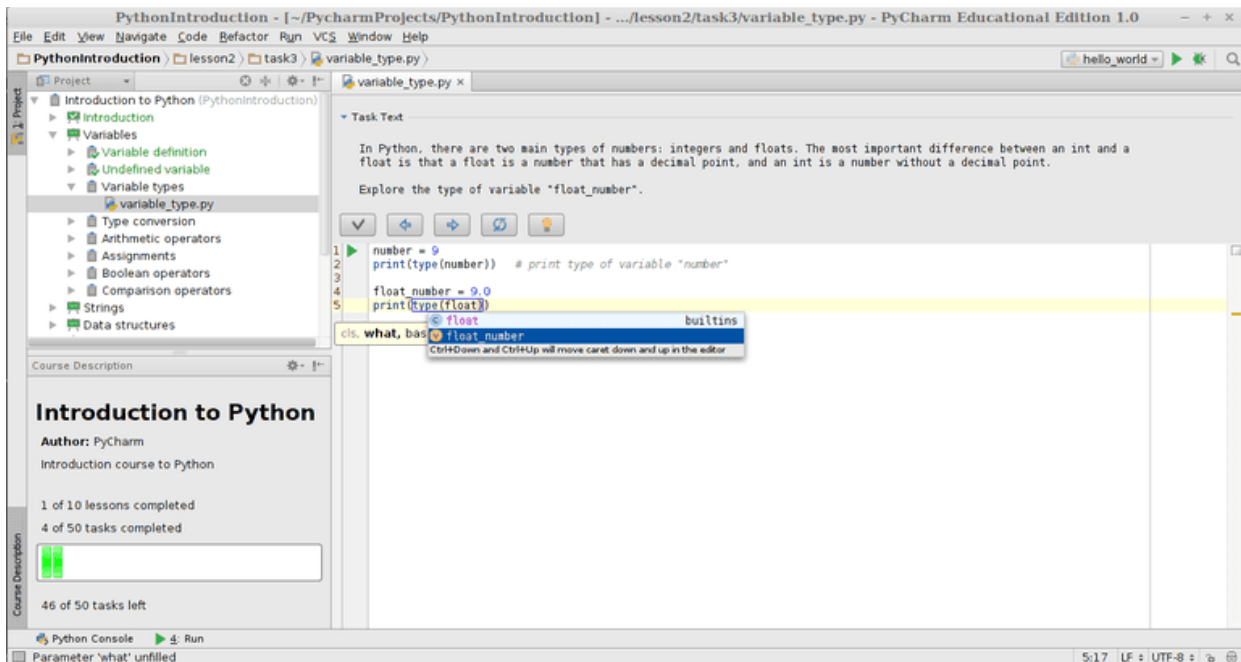




Посмотреть тип переменной в данный момент времени можно с помощью встроенной функции `type`.

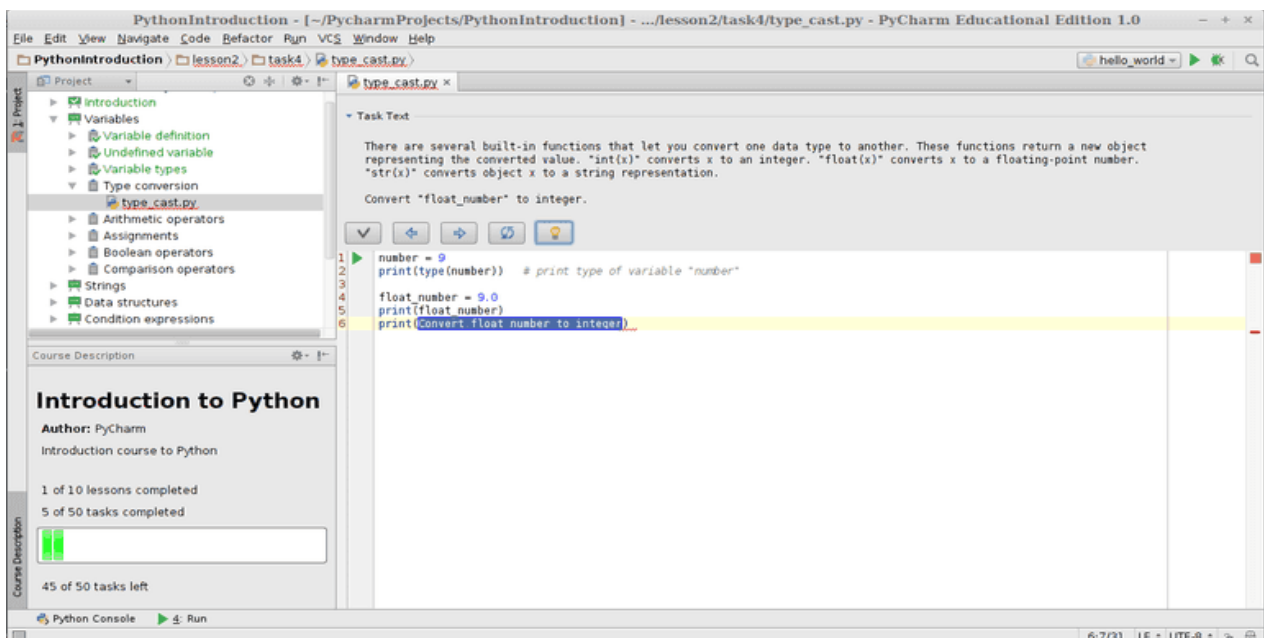
Кстати, в курсе есть подсказки, которые появляются при нажатии на лампочку.

А также в PyCharm работает автодополнение.



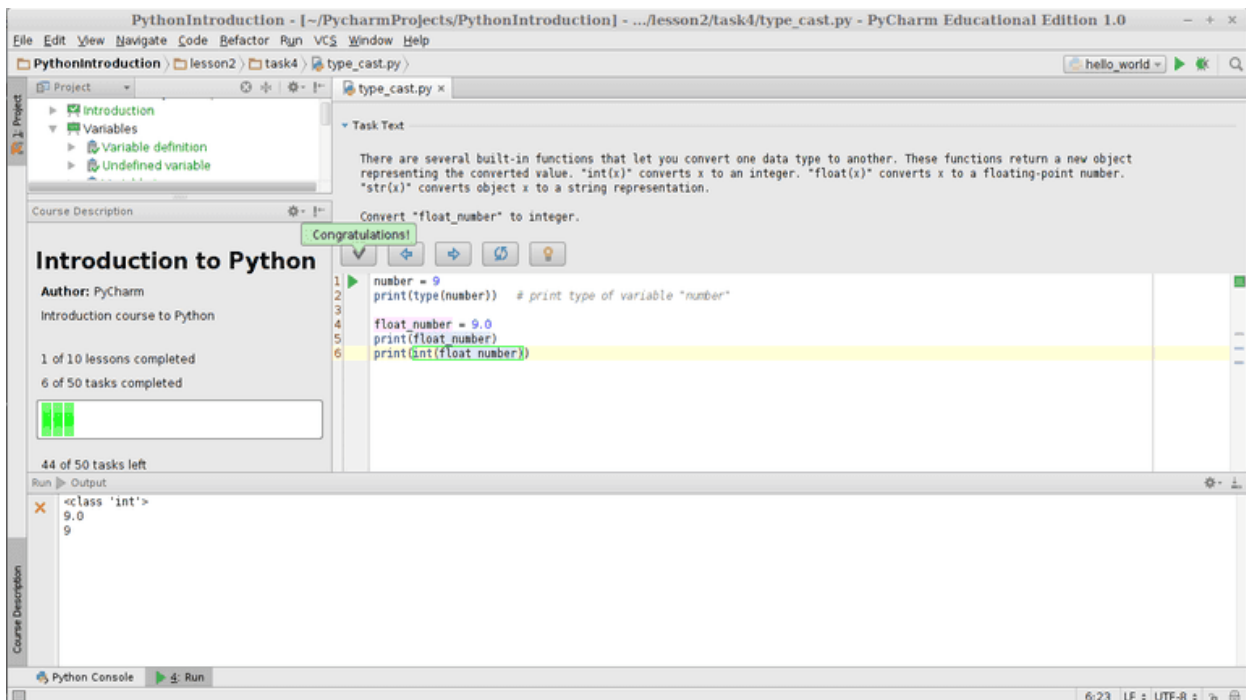
## Задание 6: преобразование типов

Типы можно преобразовывать с помощью соответствующих функций.





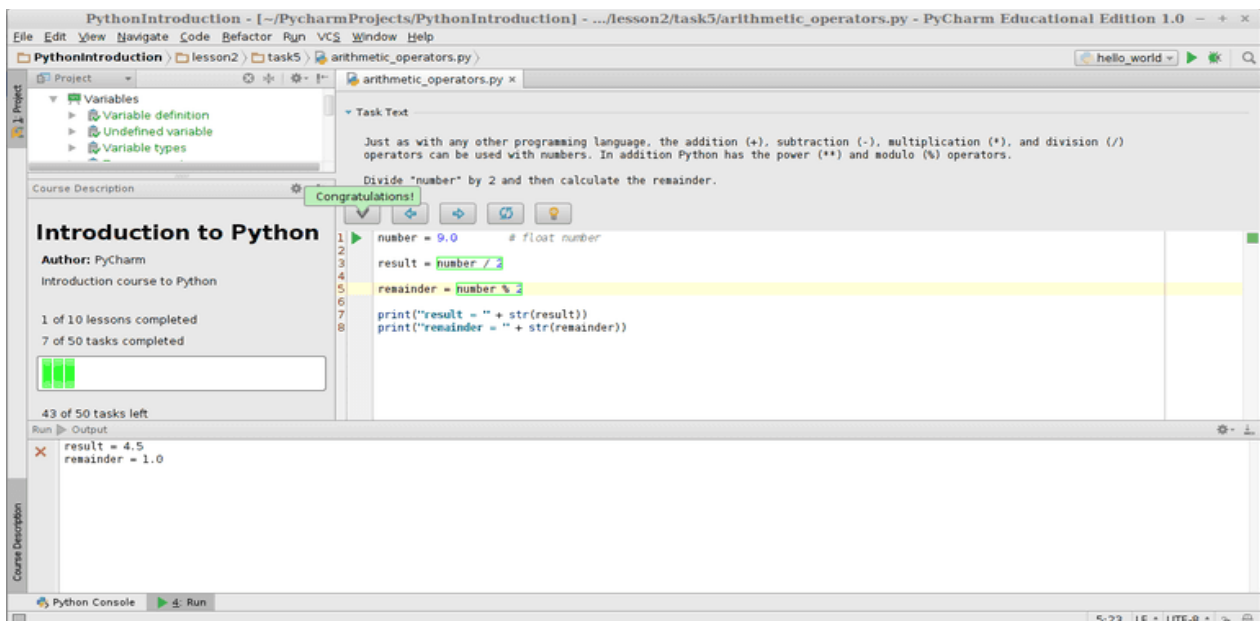
Преобразование к целому числу можно выполнить с помощью функции `int`.



## Задание 7: арифметические операции

Python поддерживает арифметические операции: сложение (+), вычитание (-), умножение (\*), деление (/), целочисленное деление (//), взятие остатка от деления (%), возведение в степень.

Сразу ответ к заданию:



Как видите, курс несложный, прекрасно подходит для начинающих, а также позволяет легко научиться работе с PyCharm. Советую пройти его весь, тем более, что на каждое задание есть подсказки.

Если останутся вопросы, Python-сообщество всегда радо будет помочь.

## **Компиляция программы на python 3 в exe с помощью программы cx\_Freeze**

Наверняка, у каждого Python-программиста возникало непреодолимое желание, а изредка и реальная потребность, скомпилировать свою программу на python в exe файл. Сегодня я расскажу, как это сделать с помощью программы cx\_Freeze.

### **Вопрос 1: а оно надо?**

Прежде, чем начинать компилировать программу, нужно убедиться, что это действительно необходимо.

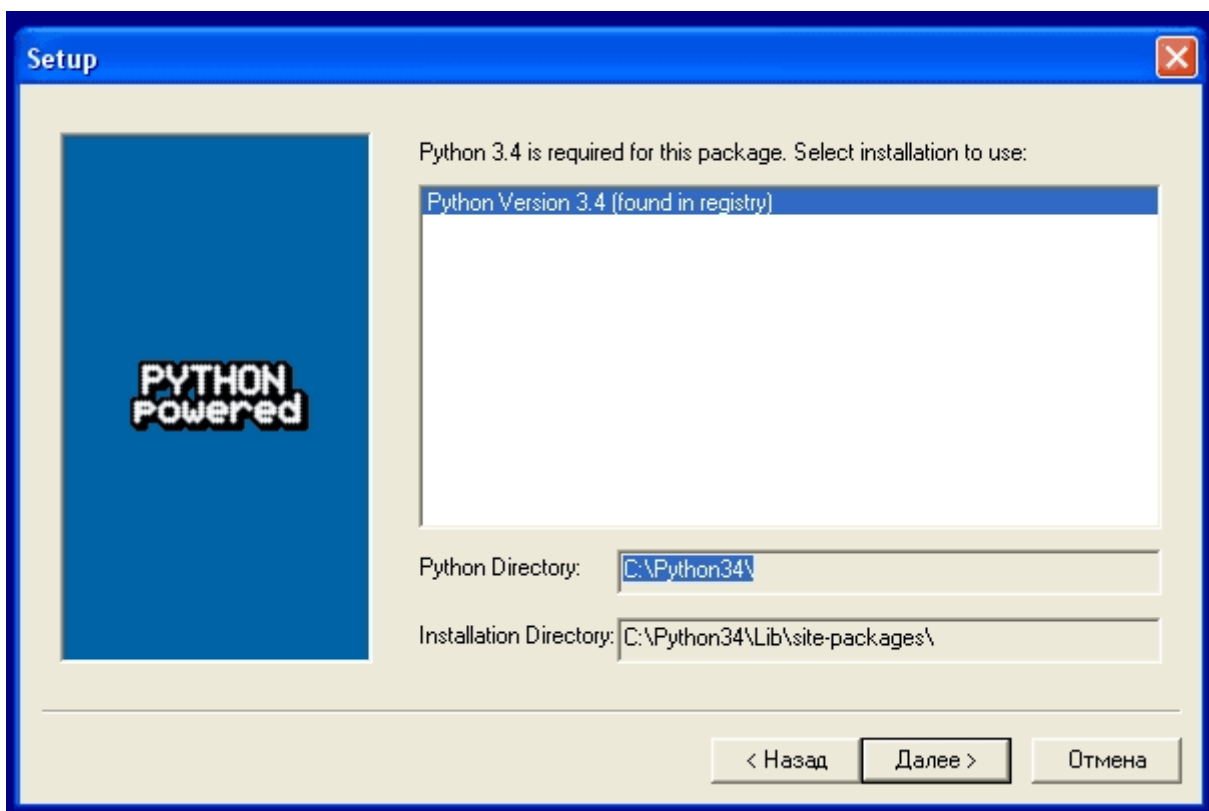
"Компиляция" программы на python - процесс, который может сопровождаться множеством проблем. Более того, это не компиляция в машинный код, как, например, программы на C, а лишь "сборка" в исполняемый файл вашей программы вместе с частью интерпретатора python.

Поэтому, если вы будете использовать программу сами или на компьютерах тех, кто её будет использовать, будет стоять интерпретатор python, то вам **не нужно** её компилировать.

### **Установка cx\_Freeze**

Если компилировать программу на python всё-таки нужно, то скачиваем cx\_Freeze с [http://www.lfd.uci.edu/~gohlke/pythonlibs/#cx\\_freeze](http://www.lfd.uci.edu/~gohlke/pythonlibs/#cx_freeze) (так как в ней нет одного неприятного бага, который есть в [официальной версии](#)).

Затем устанавливаем, не забывая правильно выбрать директорию, где расположен python.



## Компиляция

Компилировать будем программу "блэкджек" (файл 21.py) (кому лень смотреть, привожу полный исходный код).

```
import random
random.shuffle(koloda)

print('Поиграем в очко?')
count = 0

while True:
    choice = input('Будете брать карту? y/n\n')
    if choice == 'y':
        current = koloda.pop()
        print('Вам попала карта достоинством %d'
%current)
        count += current
        if count > 21:
            print('Извините, но вы проиграли')
            break
        elif count == 21:
```

```

        print('Поздравляю, вы набрали 21!')
        break
    else:
        print('У вас %d очков.' %count)
elif choice == 'n':
    print('У вас %d очков и вы закончили игру.'
%count)
    break

print('До новых встреч!')

```

Создаём в папке с программой файл setup.py с содержанием:

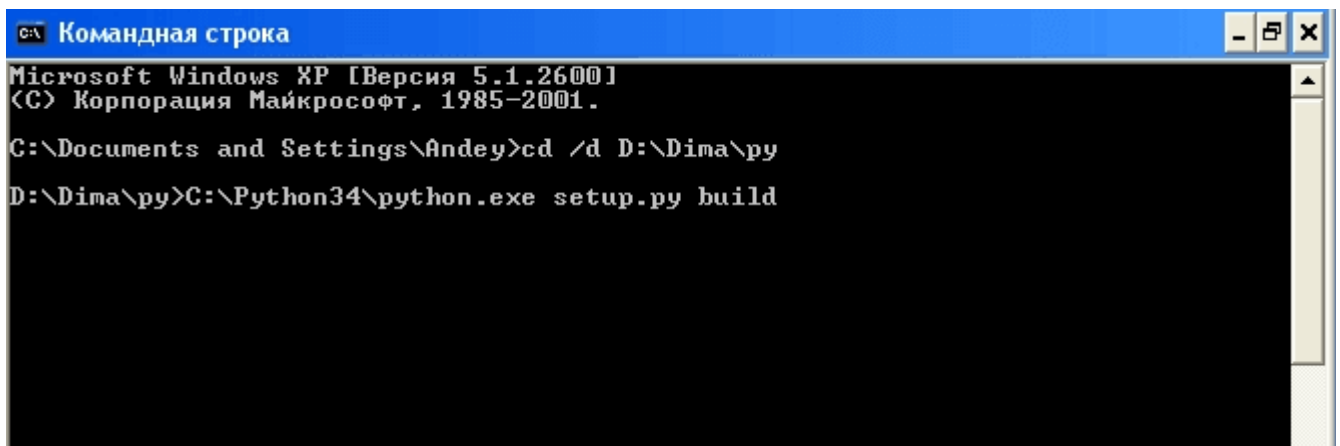
```

from cx_Freeze import setup, Executable

setup(
    name = "21",
    version = "0.1",
    description = "Blackjack",
    executables = [Executable("21.py")]
)

```

Переходим в командную строку (обычно Пуск → Стандартные → Командная строка). Переходим в папку с программой (в моём случае D:\Dima\py). Запускаем команду **C:\Python34\python.exe setup.py build** (вместо C:\Python34 нужно написать папку, куда установлен python).



```

C:\> Командная строка
Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\Andey>cd /d D:\Dima\py
D:\Dima\py>C:\Python34\python.exe setup.py build

```

Начнется процесс сборки, в котором cx\_Freeze может выдавать предупреждения, но, скорее всего, они не повлияют на работу программы.

```
Командная строка - C:\Python34\python.exe setup.py build
m gettext
m hashlib C:\Python34\lib\hashlib.py
m heapq
P importlib C:\Python34\lib\importlib\__init__.py
m importlib._bootstrap
m io
m itertools
m keyword
m linecache
m locale
P logging C:\Python34\lib\logging\__init__.py
m math
m nt
m ntpath
m operator
m optparse
m os
m posixpath
m quopri
m random C:\Python34\lib\random.py
m re
m reprlib
m sre_compile
m sre_constants
m sre_parse
m stat
m string
m stringprep
m struct
m sys
m textwrap
m threading
m time
m token
m tokenize
m traceback
m types
m unicodedata C:\Python34\DLLs\unicodedata.pyd
m warnings
m weakref
m zipimport
m zlib

Missing modules:
? _dummy_threading imported from dummy_threading
? ce imported from os
? doctest imported from heapq
? getopt imported from base64, quopri
? org.python.core imported from copy
? os.path imported from os
? posix imported from os
? pwd imported from posixpath
? subprocess imported from os
This is not necessarily a problem - the modules may not be needed on this platform.

copying C:\Python34\DLLs\_bz2.pyd -> build\exe.win32-3.4\_bz2.pyd
copying C:\Python34\DLLs\_hashlib.pyd -> build\exe.win32-3.4\_hashlib.pyd
```

Поздравляю! В папке build теперь есть ваша папка с исполняемым файлом!

```
D:\Dima\py>cd build
D:\Dima\py\build>cd exe.win32-3.4
D:\Dima\py\build\exe.win32-3.4>21.exe
Поиграем в очко?
Будете брать карту? y/n
```

Заметьте, не файл, а папка! Все остальные файлы также нужны для работы программы (да, это недостаток `sx_Freeze`). К тому же, при завершении программы окно сразу закрывается (а не ждет нажатия клавиши), поэтому в конец программы нужно дописать что-то вроде:

```
input("Press Enter")
```

И скомпилировать ещё раз.

Также можно создать msi-архив командой `C:\Python34\python.exe setup.py bdist_msi`.

Разумеется, при компиляции более сложных программ вы можете столкнуться с более сложными проблемами.

Для их решения следует воспользоваться официальной документацией.

## NumPy: начало работы

NumPy — это библиотека языка Python, добавляющая поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.

### Установка NumPy

На linux - пакет `python3-numpy` (или аналогичный для вашей системы), или через `pip`. Ну или же собирать из исходников <https://sourceforge.net/projects/numpy/files/NumPy/>.

На Windows на том же сайте есть exe установщики. Или, если возникают проблемы, рекомендую ещё хороший сборник библиотек <http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy>.

## Начинаем работу

Основным объектом NumPy является однородный многомерный массив (в numpy называется `numpy.ndarray`). Это многомерный массив элементов (обычно чисел), одного типа.

Наиболее важные атрибуты объектов `ndarray`:

**`ndarray.ndim`** - число измерений (чаще их называют "оси") массива.

**`ndarray.shape`** - размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси. Для матрицы из  $n$  строк и  $m$  столбцов, `shape` будет  $(n,m)$ . Число элементов кортежа `shape` равно `ndim`.

**`ndarray.size`** - количество элементов массива. Очевидно, равно произведению всех элементов атрибута `shape`.

**`ndarray.dtype`** - объект, описывающий тип элементов массива. Можно определить `dtype`, используя стандартные типы данных Python. NumPy здесь предоставляет целый букет возможностей, как встроенных, например: `bool_`, `character`, `int8`, `int16`, `int32`, `int64`, `float8`, `float16`, `float32`, `float64`, `complex64`, `object_`, так и возможность определить собственные типы данных, в том числе и составные.

**`ndarray.itemsize`** - размер каждого элемента массива в байтах.

**`ndarray.data`** - буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

## Создание массивов

В NumPy существует много способов создать массив. Один из наиболее простых - создать массив из обычных списков или кортежей Python, используя функцию `numpy.array()` (запомните: `array` - функция, создающая объект типа `ndarray`):

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>
```

Функция `array()` трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности (но можно и переопределить его в момент создания).

```
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Можно также переопределить тип в момент создания:

```
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]],
dtype=np.complex)
>>> b
array([[ 1.5+0.j,  2.0+0.j,  3.0+0.j],
       [ 4.0+0.j,  5.0+0.j,  6.0+0.j]])
```

Функция `array()` не единственная функция для создания массивов. Обычно элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип создаваемого массива — `float64`).

Функция `zeros()` создает массив из нулей, а функция `ones()` — массив из единиц. Обе функции принимают кортеж с размерами, и аргумент `dtype`:

```
>>> np.zeros((3, 5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> np.ones((2, 2, 2))
array([[[ 1.,  1.],
        [ 1.,  1.]],
       [[ 1.,  1.],
        [ 1.,  1.]])
```



Функция `eye()` создаёт единичную матрицу (двумерный массив)

```
>>> np.eye(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

Функция `empty()` создает массив без его заполнения. Исходное содержимое случайно и зависит от состояния памяти на момент создания массива (то есть от того мусора, что в ней хранится):

```
>>> np.empty((3, 3))
array([[ 6.93920488e-310,  6.93920488e-310,
 6.93920149e-310],
       [ 6.93920058e-310,  6.93920058e-310,
 6.93920058e-310],
       [ 6.93920359e-310,  0.00000000e+000,
 6.93920501e-310]])
>>> np.empty((3, 3))
array([[ 6.93920488e-310,  6.93920488e-310,
 6.93920147e-310],
       [ 6.93920149e-310,  6.93920146e-310,
 6.93920359e-310],
       [ 6.93920359e-310,  0.00000000e+000,
 3.95252517e-322]])
```

Для создания последовательностей чисел, в NumPy имеется функция `arange()`, аналогичная встроенной в Python `range()`, только вместо списков она возвращает массивы, и принимает не только целые значения:

```
>> arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 1, 0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,
 0.8,  0.9])
```

Вообще, при использовании `arange()` с аргументами типа `float`, сложно быть уверенным в том, сколько элементов будет получено (из-за ограничения точности чисел с плавающей запятой). Поэтому, в таких случаях обычно лучше использовать функцию `linspace()`, которая вместо шага в качестве одного из аргументов принимает число, равное количеству нужных элементов:

```
>>> np.linspace(0, 2, 9) # 9 чисел от 0 до 2
ВКЛЮЧИТЕЛЬНО
array([ 0. ,  0.25,  0.5 ,  0.75,  1. ,  1.25,  1.5 ,  1.75,
        2. ])
```

fromfunction(): применяет функцию ко всем комбинациям индексов

```
>>> def f1(i, j):
...     return 3 * i + j
...
>>> np.fromfunction(f1, (3, 4))
array([[ 0.,  1.,  2.,  3.],
       [ 3.,  4.,  5.,  6.],
       [ 6.,  7.,  8.,  9.]])
>>> np.fromfunction(f1, (3, 3))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

## Печать массивов

Если массив слишком большой, чтобы его печатать, NumPy автоматически скрывает центральную часть массива и выводит только его уголки.

```
>>> print(np.arange(0, 3000, 1))
[  0    1    2 ..., 2997 2998 2999]
```

Если вам действительно нужно увидеть весь массив, используйте функцию `numpy.set_printoptions`:

```
np.set_printoptions(threshold=np.nan)
```

И вообще, с помощью этой функции можно настроить печать массивов "под себя". Функция `numpy.set_printoptions` принимает несколько аргументов:

*precision* : количество отображаемых цифр после запятой (по умолчанию 8).

*threshold* : количество элементов в массиве, вызывающее обрезание элементов (по умолчанию 1000).

*edgeitems* : количество элементов в начале и в конце каждой размерности массива (по умолчанию 3).

*linewidth* : количество символов в строке, после которых осуществляется перенос (по умолчанию 75).

*suppress* : если True, не печатает маленькие значения в scientific notation (по умолчанию False).

*nanstr* : строковое представление NaN (по умолчанию 'nan').

*infstr* : строковое представление inf (по умолчанию 'inf').

*formatter* : позволяет более тонко управлять печатью массивов. Здесь я его рассматривать не буду, можете почитать [здесь](#) (на английском).

И вообще, пользуйтесь [официальной документацией по numpy](#), а в этом пособии я постараюсь описать всё необходимое. В следующей части мы рассмотрим базовые операции над массивами.

## NumPy: базовые операции над массивами

### Базовые операции

Математические операции над массивами выполняются поэлементно. Создается новый массив, который заполняется результатами действия оператора.

```
>>> import numpy as np
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> a + b
array([20, 31, 42, 53])
>>> a - b
array([20, 29, 38, 47])
>>> a * b
array([ 0, 30, 80, 150])
>>> a / b # При делении на 0 возвращается inf
(бесконечность)
array([          inf, 30.          , 20.          ,
16.66666667])
<string>:1: RuntimeWarning: divide by zero encountered in
true_divide
>>> a ** b
array([ 1, 30, 1600, 125000])
>>> a % b # При взятии остатка от деления на 0
возвращается 0
```

```
<string>:1: RuntimeWarning: divide by zero encountered in  
remainder  
array([0, 0, 0, 2])
```

Для этого, естественно, массивы должны быть одинаковых размеров.

```
>>> c = np.array([[1, 2, 3], [4, 5, 6]])  
>>> d = np.array([[1, 2], [3, 4], [5, 6]])  
>>> c + d  
Traceback (most recent call last):  
  File "<input>", line 1, in <module>  
ValueError: operands could not be broadcast together with  
shapes (2,3) (3,2)
```

Также можно производить математические операции между массивом и числом. В этом случае к каждому элементу прибавляется (или что вы там делаете) это число.

```
>>> a + 1  
array([21, 31, 41, 51])  
>>> a ** 3  
array([ 8000, 27000, 64000, 125000])  
>>> a < 35 # И фильтрацию можно проводить  
array([ True,  True, False, False], dtype=bool)
```

NumPy также предоставляет множество математических операций для обработки массивов:

```
>>> np.cos(a)  
array([ 0.40808206,  0.15425145, -0.66693806,  
 0.96496603])  
>>> np.arctan(a)  
array([ 1.52083793,  1.53747533,  1.54580153,  
 1.55079899])  
>>> np.sinh(a)  
array([ 2.42582598e+08,  5.34323729e+12,  
 1.17692633e+17,  
 2.59235276e+21])
```

Полный список можно посмотреть [здесь](#) (кстати, нужно ли их переводить?)

Многие унарные операции, такие как, например, вычисление суммы всех элементов массива, представлены также и в виде методов класса ndarray.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.sum(a)
21
>>> a.sum()
21
>>> a.min()
1
>>> a.max()
6
```

По умолчанию, эти операции применяются к массиву, как если бы он был списком чисел, независимо от его формы. Однако, указав параметр `axis`, можно применить операцию для указанной оси массива:

```
>>> a.min(axis=0) # Наименьшее число в каждом столбце
array([1, 2, 3])
>>> a.min(axis=1) # Наименьшее число в каждой строке
array([1, 4])
```

## Индексы, срезы, итерации

Одномерные массивы осуществляют операции индексирования, срезов и итераций очень схожим образом с обычными списками и другими последовательностями Python (разве что удалять с помощью срезов нельзя).

```
>>> a = np.arange(10) ** 3
>>> a
array([ 0,  1,  8, 27, 64, 125, 216, 343, 512, 729])
>>> a[1]
1
>>> a[3:7]
array([ 27,  64, 125, 216])
>>> a[3:7] = 8
>>> a
array([ 0,  1,  8,  8,  8,  8,  8, 343, 512, 729])
>>> a[::-1]
array([729, 512, 343,  8,  8,  8,  8,  8,  1,  0])
>>> del a[4:6]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: cannot delete array elements

>>> for i in a:
```

```
...     print(i ** (1/3))
...
0.0
1.0
2.0
2.0
2.0
2.0
2.0
7.0
8.0
9.0
```

У многомерных массивов на каждую ось приходится один индекс. Индексы передаются в виде последовательности чисел, разделенных запятыми (то бишь, кортежами):

```
>>> b = np.array([[ 0, 1, 2, 3],
                  [10, 11, 12, 13],
                  [20, 21, 22, 23],
                  [30, 31, 32, 33],
                  [40, 41, 42, 43]])
>>> b[2,3] # Вторая строка, третий столбец
23
>>> b[(2,3)]
23
>>> b[2][3] # Можно и так
23
>>> b[:,2] # Третий столбец
array([ 2, 12, 22, 32, 42])
>>> b[:2] # Первые две строки
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
>>> b[1:3, :] # Вторая и третья строки
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Когда индексов меньше, чем осей, отсутствующие индексы предполагаются дополненными с помощью срезов:

```
>>> b[-1] # Последняя строка. Эквивалентно b[-1,:]
array([40, 41, 42, 43])
```

`b[i]` можно читать как `b[i, <столько символов ':', сколько нужно>]`. В NumPy это также может быть записано с помощью точек, как `b[i, ...]`.

Например, если `x` имеет ранг 5 (то есть у него 5 осей), тогда

- `x[1, 2, ...]` эквивалентно `x[1, 2, :, :, :]`,
- `x[... , 3]` то же самое, что `x[:, :, :, :, 3]` и
- `x[4, ... , 5, :]` это `x[4, :, :, 5, :]`.

```
>>> a = np.array([[0, 1, 2], [10, 12, 13]], [[100, 101, 102], [110, 112, 113]])
>>> a.shape
(2, 2, 3)
>>> a[1, ...] # то же, что a[1, :, :] или a[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[... , 2] # то же, что a[:, :, 2]
array([[ 2, 13],
       [102, 113]])
```

Итерирование многомерных массивов начинается с первой оси:

```
>>> for row in a:
...     print(row)
...
[[ 0  1  2]
 [10 12 13]]
[[100 101 102]
 [110 112 113]]
```

Однако, если нужно перебрать поэлементно весь массив, как если бы он был одномерным, для этого можно использовать атрибут `flat`:

```
>>> for el in a.flat:
...     print(el)
...
0
1
2
10
12
13
100
```

```
101
102
110
112
113
```

## Манипуляции с формой

Как уже говорилось, у массива есть форма (shape), определяемая числом элементов вдоль каждой оси:

```
>>> a
array([[ [ 0, 1, 2],
         [10, 12, 13]],

       [[100, 101, 102],
         [110, 112, 113]]])
>>> a.shape
(2, 2, 3)
```

Форма массива может быть изменена с помощью различных команд:

```
>>> a.ravel() # Делает массив плоским
array([ 0, 1, 2, 10, 12, 13, 100, 101, 102, 110,
       112, 113])
>>> a.shape = (6, 2) # Изменение формы
>>> a
array([[ 0, 1],
       [ 2, 10],
       [12, 13],
       [100, 101],
       [102, 110],
       [112, 113]])
>>> a.transpose() # Транспонирование
array([[ 0, 2, 12, 100, 102, 112],
       [ 1, 10, 13, 101, 110, 113]])
>>> a.reshape((3, 4)) # Изменение формы
array([[ 0, 1, 2, 10],
       [12, 13, 100, 101],
       [102, 110, 112, 113]])
```

Порядок элементов в массиве в результате функции `ravel()` соответствует обычному "С-стилю", то есть, чем правее индекс, тем он "быстрее изменяется":



за элементом `a[0,0]` следует `a[0,1]`. Если одна форма массива была изменена на другую, массив переформируется также в "С-стиле". Функции `ravel()` и `reshape()` также могут работать (при использовании дополнительного аргумента) в FORTRAN-стиле, в котором быстрее изменяется более левый индекс.

```
>>> a
array([[ 0,  1],
       [ 2, 10],
       [12, 13],
       [100, 101],
       [102, 110],
       [112, 113]])
>>> a.reshape((3, 4), order='F')
array([[ 0, 100,  1, 101],
       [ 2, 102, 10, 110],
       [12, 112, 13, 113]])
```

Метод `reshape()` возвращает ее аргумент с измененной формой, в то время как метод `resize()` изменяет сам массив:

```
>>> a.resize((2, 6))
>>> a
array([[ 0,  1,  2, 10, 12, 13],
       [100, 101, 102, 110, 112, 113]])
```

Если при операции такой перестройки один из аргументов задается как -1, то он автоматически рассчитывается в соответствии с остальными заданными:

```
>>> a.reshape((3, -1))
array([[ 0,  1,  2, 10],
       [12, 13, 100, 101],
       [102, 110, 112, 113]])
```

## Объединение массивов

Несколько массивов могут быть объединены вместе вдоль разных осей с помощью функций `hstack` и `vstack`.

`hstack()` объединяет массивы по первым осям, `vstack()` — по последним:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
>>> np.vstack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.hstack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Функция `column_stack()` объединяет одномерные массивы в качестве столбцов двумерного массива:

```
>>> np.column_stack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Аналогично для строк имеется функция `row_stack()`.

```
>>> np.row_stack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

## Разбиение массива

Используя `hsplit()` вы можете разбить массив вдоль горизонтальной оси, указав либо число возвращаемых массивов одинаковой формы, либо номера столбцов, после которых массив разрезается "ножницами":

```
>>> a = np.arange(12).reshape((2, 6))
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> np.hsplit(a, 3) # Разбить на 3 части
[array([[0, 1], [6, 7]]),
 array([[2, 3], [8, 9]]),
 array([[4, 5], [10, 11]])]
```

```
>>> np.hsplit(a, (3, 4)) # Разрезать a после третьего и
четвёртого столбца
[array([[0, 1, 2], [6, 7, 8]]),
 array([[3], [9]]),
 array([[ 4,  5], [10, 11]])]
```

Функция `vsplit()` разбивает массив вдоль вертикальной оси, а `array_split()` позволяет указать оси, вдоль которых произойдет разбиение.

## Копии и представления

При работе с массивами, их данные иногда необходимо копировать в другой массив, а иногда нет. Это часто является источником путаницы. Возможно 3 случая:

### Вообще никаких копий

Простое присваивание не создает ни копии массива, ни копии его данных:

```
>>> a = np.arange(12)
>>> b = a # Нового объекта создано не было
>>> b is a # a и b это два имени для одного и того же
объекта ndarray
True
>>> b.shape = (3, 4) # изменит форму a
>>> a.shape
(3, 4)
```

Python передает изменяемые объекты как ссылки, поэтому вызовы функций также не создают копий.

### Представление или поверхностная копия

Разные объекты массивов могут использовать одни и те же данные. Метод `view()` создает новый объект массива, являющийся представлением тех же данных.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a # c это представление данных,
принадлежащих a
```

```

True
>>> c.flags.owndata
False
>>>
>>> c.shape = (2,6)  # форма a не поменяется
>>> a.shape
(3, 4)
>>> c[0,4] = 1234  # данные a изменятся
>>> a
array([[ 0,  1,  2,  3],
       [1234,  5,  6,  7],
       [ 8,  9, 10, 11]])

```

Срез массива это представление:

```

>>> s = a[:,1:3]
>>> s[:] = 10
>>> a
array([[ 0, 10, 10,  3],
       [1234, 10, 10,  7],
       [ 8, 10, 10, 11]])

```

## Глубокая копия

Метод `copy()` создаст настоящую копию массива и его данных:

```

>>> d = a.copy()  # создается новый объект массива с
# новыми данными
>>> d is a
False
>>> d.base is a  # d не имеет ничего общего с a
False
>>> d[0, 0] = 9999
>>> a
array([[ 0, 10, 10,  3],
       [1234, 10, 10,  7],
       [ 8, 10, 10, 11]])

```

## NumPy: random

Сейчас мы узнаем, как создавать массивы из случайных элементов и как работать со случайными элементами в NumPy.

### Путь первый

Создавать списки, используя встроенный модуль random, а затем преобразовывать их в numpy.array:

```
>>> import numpy as np
>>> import random
>>> np.array([random.random() for i in range(10)])
array([ 0.99538667,  0.16860511,  0.78952804,
        0.09676316,  0.86110208,
         0.89674666,  0.56401347,  0.63431468,
        0.51110935,  0.64944844])
```

Но есть способ лучше.

### numpy.random

Для создания массивов со случайными элементами служит модуль **numpy.random**.

```
>>> import numpy as np # Импортировать numpy и писать
np.random
>>> np.random
<module 'numpy.random' from
'/usr/local/lib/python3.4/dist-
packages/numpy/random/__init__.py'>
>>> import numpy.random as rand # Можно и присвоить
отдельное имя. Вопрос вкуса
>>> rand
<module 'numpy.random' from
'/usr/local/lib/python3.4/dist-
packages/numpy/random/__init__.py'>
```

### Создание массивов

Самый простой способ задать массив со случайными элементами - использовать функцию sample (или random, или random\_sample, или randf - это всё одна и та же функция).

```
>>> np.random.sample()
0.6336371838734877
>>> np.random.sample(3)
array([ 0.53478558,  0.1441317 ,  0.15711313])
>>> np.random.sample((2, 3))
array([[ 0.12915769,  0.09448946,  0.58778985],
       [ 0.45488207,  0.19335243,  0.22129977]])
```

Без аргументов возвращает просто число в промежутке  $[0, 1)$ , с одним целым числом - одномерный массив, с кортежем - массив с размерами, указанными в кортеже (все числа - из промежутка  $[0, 1)$ ).

С помощью функции `randint` или `random_integers` можно создать массив из целых чисел. Аргументы: `low`, `high`, `size`: от какого, до какого числа (`randint` не включает в себя это число, а `random_integers` включает), и `size` - размеры массива.

```
>>> np.random.randint(0, 3, 10)
array([0, 2, 0, 1, 1, 0, 2, 2, 2, 0])
>>> np.random.random_integers(0, 3, 10)
array([2, 2, 3, 3, 1, 1, 0, 2, 3, 2])
>>> np.random.randint(0, 3, (2, 10))
array([[0, 1, 2, 0, 0, 0, 1, 1, 1, 2],
       [0, 0, 2, 2, 2, 0, 1, 2, 2, 1]])
```

Также можно генерировать числа согласно различным [распределениям](#) (Гаусса, Парето и другие). Чаще всего нужно равномерное распределение, которое можно получить с помощью функции `uniform`.

```
>>> np.random.uniform(2, 8, (2, 10))
array([[ 3.1517914 ,  3.10313483,  2.84007134,
         3.21556436,  4.64531786,
          2.99232714,  7.03064897,  4.38691765,
         5.27488548,  2.63472454],
       [ 6.39470358,  5.63084131,  4.69996748,
         7.07260546,  7.44340813,
         4.10722203,  7.52956646,  4.8596943 ,
         3.97923973,  5.64505363]])
```

## Выбор и перемешивание

Перемешать NumPy массив можно с помощью функции `shuffle`:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.random.shuffle(a)
>>> a
array([2, 8, 7, 3, 5, 0, 4, 9, 1, 6])
```

Также можно перемешать массив с помощью функции `permutation` (она, в отличие от `shuffle`, возвращает перемешанный массив). Также она, вызванная с одним аргументом (целым числом), возвращает перемешанную последовательность от 0 до N.

```
>>> np.random.permutation(10)
array([1, 2, 3, 8, 7, 9, 4, 6, 5, 0])
```

Сделать случайную выборку из массива можно с помощью функции `choice`. Про неё стоит рассказать подробнее.

**`numpy.random.choice(a, size=None, replace=True, p=None)`**

- `a` : одномерный массив или число. Если массив, будет производиться выборка из него. Если число, то выборка будет производиться из `np.arange(a)`.
- `size` : размерности массива. Если `None`, возвращается одно значение.
- `replace` : если `True`, то одно значение может выбираться более одного раза.
- `p` : вероятности. Это означает, что элементы можно выбирать с неравными вероятностями. Если не заданы, используется равномерное распределение.

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.random.choice(a, 10, p=[0.5, 0.25, 0.25, 0, 0, 0, 0, 0, 0, 0])
array([0, 0, 0, 0, 1, 2, 0, 0, 1, 1])
```

## Инициализация генератора случайных чисел

seed(число) - инициализация генератора.

```
>>> np.random.seed(1000)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,  0.4821914
,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,  0.2331322
,  0.84174072])
>>> np.random.seed(1000)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,  0.4821914
,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,  0.2331322
,  0.84174072])
```

get\_state и set\_state - возвращают и устанавливают состояние генератора.

```
>>> np.random.seed(1000)
>>> state = np.random.get_state()
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,  0.4821914
,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,  0.2331322
,  0.84174072])
>>> np.random.set_state(state)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,  0.4821914
,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,  0.2331322
,  0.84174072])
```



## NumPy: linalg

### Возведение в степень

**linalg.matrix\_power**(M, n) - возводит матрицу в степень n.

### Разложения

**linalg.cholesky**(a) - разложение Холецкого.

**linalg.qr**(a[, mode]) - QR разложение.

**linalg.svd**(a[, full\_matrices, compute\_uv]) - сингулярное разложение.

### Некоторые характеристики матриц

**linalg.eig**(a) - собственные значения и собственные векторы.

**linalg.norm**(x[, ord, axis]) - норма вектора или оператора.

**linalg.cond**(x[, p]) - число обусловленности.

**linalg.det**(a) - определитель.

**linalg.slogdet**(a) - знак и логарифм определителя (для избежания переполнения, если сам определитель очень маленький).

### Системы уравнений

**linalg.solve**(a, b) - решает систему линейных уравнений  $Ax = b$ .

**linalg.tensorsolve**(a, b[, axes]) - решает тензорную систему линейных уравнений  $Ax = b$ .

**linalg.lstsq**(a, b[, rcond]) - метод наименьших квадратов.

**linalg.inv**(a) - обратная матрица.

Замечания:

- **linalg.LinAlgError** - исключение, вызываемое данными функциями в случае неудачи (например, при попытке взять обратную матрицу от вырожденной).

- Подробная документация, как всегда, на английском: <https://docs.scipy.org/doc/numpy/reference/routines.linalg.html>
- Массивы большей размерности в большинстве функций `linalg` интерпретируются как набор из нескольких массивов нужной размерности. Таким образом, можно одним вызовом функции проделывать операции над несколькими объектами.

```
>>> a = np.arange(18).reshape((2, 3, 3))
>>> a
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],
       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]]])
>>> np.linalg.det(a)
array([ 0.,  0.] )
```

## Что нового в Python 3,3?

### Основная информация

Новые возможности синтаксиса:

- Новое выражение **yield from** для реализации генераторов.
- Синтаксис **u"unicode"** снова разрешен для объектов типа `str`.

Новые встроенные модули:

- `faulthandler` (отладка низкоуровневых ошибок)
- `ipaddress` (высокоуровневая работа с IP-адресами и масками)
- `lzma` (компрессия с использованием алгоритма XZ / LZMA)
- `venv` (виртуальная среда Python)

Новые встроенные улучшения:

- Переработана иерархия I/O исключений

Реализация улучшений:

- Переписана машина импорта на основе `importlib`
- Более компактные строки `unicode`
- Более компактные словари

Улучшения безопасности:

- Рандомизация хеш-суммы включена по умолчанию

## **PEP 405: Виртуальная среда**

Виртуальные среды помогут создать отдельные установки Python, разделяя общесистемные базовые установки для удобства обслуживания. Виртуальные среды имеют свой собственный набор частных пакетов (т.е. локально установленные библиотеки), и необязательно отделены от общесистемных пакетов.

Этот PEP добавляет модуль `venv` для программного доступа, и скрипт `pyenv` для доступа к командной строке и администрирования. Интерпретатору Python становится известно о `pyenv.cfg` файле, существование которого сигнализирует базу деревьев каталогов виртуальной среды.

## **PEP 420: Пространство имен пакетов**

Встроенная поддержка пакетов, которые не требуют файлы `__init__.py`

## **PEP 393: гибкое представление строк**

Строки изменены для поддержки нескольких внутренних представлений, в зависимости от символа с наибольшим порядковым `Unicode` (1, 2 или 4 байта). Это позволяет эффективно представлять строки в большинстве случаев, но дает доступ к полному `UCS-4` на всех системах. Для совместимости с существующим API, несколько представлений могут существовать параллельно, с течением времени, эта совместимость должна быть прекращена.

На стороне C API, PEP 393 является полностью обратно совместимым. API должны оставаться доступными по крайней мере пять лет. Приложения, использующие API не будут в полной мере пользоваться сокращением памяти, или - еще хуже - может использовать немного больше памяти, так как Python,

возможно, придется поддерживать две версии каждой строки (в устаревшем формате и в новом эффективного хранения).

## **Функциональность**

Изменения, внесенные PEP 393:

- Python теперь всегда поддерживает полный набор кодов Unicode, в том числе не-BMP символы (т.е. от U+0000 до U+10FFFF). Различие между узкой и широкой строкой больше не существует, и Python теперь ведет себя как при широкой сборке, даже под Windows.
- Со смертью узкой версии, проблемы, характерные для узкой версии также были исправлены, например:
  - o `len()` теперь всегда возвращает 1 для не-BMP символов, поэтому `len('\U0010FFFF') == 1` ;
  - o суррогатные пары не рекомбинируют в строковые литералы, так `'\uDBFF\uDFFF' != '\U0010FFFF'` ;
  - o индекс или срез не-BMP символов возвращает ожидаемое значение, поэтому `'\U0010FFFF'[0]` теперь возвращает `'\U0010FFFF'` , а не `'\uDBFF'` ;
  - o Все другие функции в стандартной библиотеке теперь корректно обрабатывают не-BMP коды.
- Значение `sys.maxunicode` теперь всегда 1114111 ( 0x10FFFF в шестнадцатеричной). `PyUnicode_GetMax()` по-прежнему возвращает либо 0xFFFF или 0x10FFFF для обратной совместимости, и его не следует использовать с новым API Unicode.
- `./configure` флаг `--with-wide-unicode` был удален.

## **Производительность и использование ресурсов**

Хранение Unicode строк теперь зависит от максимального кода в кодовой строке:

- ASCII и Latin1 строки ( U +0000- U +00 FF ) используют 1 байт;
- BMP строки ( U +0000- U + FFFF ) используют 2 байта;
- не-BMP строки ( U +10000- U +10 FFFF ) используют 4 байта.

Результатом является то, что для большинства приложений, использование памяти должно значительно уменьшиться - особенно по сравнению с бывшим

Unicode, так как во многих случаях, строки будут чистым ASCII даже в международном контексте (потому что много строк хранят данные не-человеческим языком, такие как XML-фрагменты, HTTP заголовки, JSON-кодированные данных и т.д.). Мы также надеемся, что она, по тем же причинам, повысит эффективность кэш-памяти процессора для нетривиальных приложений. Использование памяти Python 3.3 в два-три раза меньше, чем Python 3.2, и немного меньше, чем Python 2.7, на тесте Django.

## **PEP 3151: Переработка ОС и IO иерархии исключений**

Иерархия исключений упрощена и более детальна.

Вам не придется больше беспокоиться о выборе подходящего типа исключения между OSError, IOError, EnvironmentError, WindowsError, mmap.error, socket.error или select.error. Все эти типы исключений теперь только один: OSError. Другие имена хранятся в качестве псевдонимов для обеспечения совместимости.

Кроме того, теперь легче поймать определенное условие ошибки. Вместо проверки атрибут ERRNO для конкретной постоянной из ERRNO модуля, вы можете обрабатывать конкретный подкласс OSError. Доступные подклассы:

- BlockingIOError
- ChildProcessError
- ConnectionError
- FileExistsError
- FileNotFoundError
- InterruptedError
- IsADirectoryError
- NotADirectoryError
- PermissionError
- ProcessLookupError
- TimeoutError

И ConnectionError также имеет подклассы:

- BrokenPipeError
- ConnectionAbortedError
- ConnectionRefusedError

- ConnectionResetError

Например, следующий код, написанный для Python 3.2:

```
from errno import ENOENT, EACCES, EPERM

try:
    with open("document.txt") as f:
        content = f.read()
except IOError as err:
    if err.errno == ENOENT:
        print("document.txt file is missing")
    elif err.errno in (EACCES, EPERM):
        print("You are not allowed to read document.txt")
    else:
        raise
```

Теперь может быть написан без импорта ERRNO и без ручной проверки атрибутов исключения:

```
try:
    with open("document.txt") as f:
        content = f.read()
except FileNotFoundError:
    print("document.txt file is missing")
except PermissionError:
    print("You are not allowed to read document.txt")
```

### PEP 380: Синтаксис для делегирования Subgenerator

PEP 380 добавляет выражение **yield from**, что позволяет генератору делегировать часть своих операций на другой генератор. Это позволяет части кода, содержащего 'yield', чтобы быть вынесена и помещена в другой генератор. Кроме того, subgenerator разрешили вернуться со значением, а значение становится доступным для делегирования генератора.

Хотя предназначен в первую очередь для использования в делегировании subgenerator, yield from фактически позволяет делегировать произвольные subiterators.

Для простых итераторов **yield from iterable** это просто сокращенная форма для **for item in iterable: yield item**

```
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

Однако, в отличие от обычного цикла, **yield from** позволяет и вернуть окончательное значение во внешний генератор:

```
>>> def accumulate(start=0):
...     tally = start
...     while 1:
...         next = yield
...         if next is None:
...             return tally
...         tally += next
...
>>> def gather_tallies(tallies, start=0):
...     while 1:
...         tally = yield from accumulate()
...         tallies.append(tally)
...
>>> tallies = []
>>> acc = gather_tallies(tallies)
>>> next(acc) # Ensure the accumulator is ready to accept
values
>>> for i in range(10):
...     acc.send(i)
...
>>> acc.send(None) # Finish the first tally
>>> for i in range(5):
...     acc.send(i)
...
>>> acc.send(None) # Finish the second tally
>>> tallies
```

[45, 10]

Основной принцип изменения - позволить разделить генератор на несколько субгенераторов так же легко, как одна большая функция может быть разбита на несколько подфункций.

## PEP 409: Подавление контекста исключения

PEP 409 вводит новый синтаксис, который позволяет отключить контекст исключений. Это позволяет получать чистое сообщение об ошибках в приложениях, конвертировать в разные типы исключений:

```
>>> class D:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr) from None
...
>>> D({}).x
Traceback (most recent call last):
  File "", line 1, in
  File "", line 8, in __getattr__
AttributeError: x
```

Без выражения **from None** будет возбуждено стандартное исключение:

```
>>> class C:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr)
...
>>> C({}).x
Traceback (most recent call last):
  File "", line 6, in __getattr__
KeyError: 'x'
```



During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "", line 1, in
  File "", line 8, in __getattr__
AttributeError: x
```

Однако оригинальное исключение все равно остается доступным, для упрощения отладки:

```
>>> try:
...     D({}).x
... except AttributeError as exc:
...     print(repr(exc.__context__))
...
KeyError('x', )
```

## Полные имена для классов и функций

Функции и объекты класса в python 3.3 имеют новый атрибут `__qualname__` (представляет «путь» от модуля верхнего уровня). Для глобальных функций и классов это то же, что и `__name__`. Для других функций и классов, он обеспечивает лучшую информацию о том, где они были фактически определены, и как они могут быть доступны из глобальной области.

Пример с (несвязанным) методом:

```
>>> class C:
...     def meth(self):
...         pass
>>> C.meth.__name__
'meth'
>>> C.meth.__qualname__
'C.meth'
```

Пример с вложенными классами:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.D.__name__
'D'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__name__
'meth'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Пример с вложенной функцией:

```
>>> def outer():
...     def inner():
...         pass
...     return inner
...
>>> outer().__name__
'inner'
>>> outer().__qualname__
'outer..inner'
```

Строковое представление этих объектов также изменено, для отображения большей информации:

```
>>> str(C.D)
'''
>>> str(C.D.meth)
''
```

## РЕР 412: Ключ-шеринг словари

Словари в python 3.3 теперь имеют возможность совместно использовать часть памяти, хранящую ключи и их хеши, что повышает производительность большинства не встроенных типов.

# Вышел Python 3.4.0

## Новые особенности:

- В поставку добавлен инсталлятор для пакетного менеджера pip.
- Новые файловые дескрипторы теперь по умолчанию не наследуются дочерними подпроцессами.
- Опция командной строки для изолированного режима (-I).
- Улучшения в обработке нетекстовых кодеков.
- Стандартизирован тип "ModuleSpec" для предоставления метаданных системы импорта модулей на стадии до непосредственной загрузки модуля.
- marshal стал более компактным и эффективным.

## Новые модули:

- asyncio: новый предварительный API для асинхронного ввода-вывода.
- ensurepip: загрузка инсталлятора pip.
- enum: реализация классов IntEnum и Enum для работы со списками перечислимых констант.
- pathlib: объектно-ориентированный интерфейс для доступа к файловой системе.
- selectors: высокоуровневое и эффективное мультиплексирование ввода-вывода, надстройка над модулем select.
- statistics: модуль для математической статистики.
- tracemalloc: трассировка распределения памяти.

## Значительно улучшенные модули:

- В functools добавлены generic-функции одиночной диспетчеризации (Single-dispatch generic functions).
- Новый протокол pickle 4.
- multiprocessing имеет опцию для избежания использования os.fork в Unix.
- email имеет новый подмодуль contentmanager, и новый подкласс класса Message (EmailMessage), что упрощает обработку MIME.

- inspect и pydoc способны к корректной интроспекции более широкого круга вызываемых объектов, что улучшает вывод функции help().
- ipaddress объявлен стабильным.

### **Улучшения безопасности:**

- Более безопасный алгоритм хэширования.
- Новая функция hashlib.pbkdf2\_hmac() (хэширование по алгоритму PBKDF2).
- В модуль ssl добавлена поддержка SNI (Server Name Indication, позволяет обеспечить доступ через шифрованное соединение к виртуальным хостам на одном IP) на стороне сервера, а также поддержка TLSv1.1 и TLSv1.2.

### **Улучшения CPython:**

- Безопасная финализация объектов.
- Новый C API для создания собственных методов распределения памяти.

Подробнее: <https://docs.python.org/3/whatsnew/3.4.html>.

## **pythondigest.ru - самые свежие новости из мира Python**

Многие уже видели этот ресурс, а кто не видел - рассказываю, так как ресурс действительно стоящий.

[pythondigest.ru](http://pythondigest.ru) - агрегатор новостей о Python, где каждый может [добавить](#) свою (или увиденную на просторе интернета) новость из мира Python.

Новости можно фильтровать по языку (русский или английский), а также по разделам:

- Интересные проекты, инструменты, библиотеки
- Книги и документация
- Конференции, события, встречи разработчиков
- Новости
- Релизы

- Статьи и интервью
- Видео

Например, [все русские новости](#).

Также каждую неделю составляется выпуск новостей. Например, [самый первый выпуск](#).

Также можно подписаться на новости от pythondigest [по RSS](#) или через [twitter](#).

## Модуль `fractions`

Модуль `fractions` предоставляет поддержку рациональных чисел.

```
class fractions.Fraction(numerator=0, denominator=1)
```

```
class fractions.Fraction(other_fraction)
```

```
class fractions.Fraction(float)
```

```
class fractions.Fraction(decimal)
```

```
class fractions.Fraction(string)
```

Класс, представляющий собой рациональные числа. Экземпляр класса можно создать из пары чисел (числитель, знаменатель), из другого рационального числа, числа с плавающей точкой, числа типа `decimal.Decimal`, и из строки, представляющей собой число.

```
>>> from fractions import Fraction
>>> Fraction(1, 3)
Fraction(1, 3)
>>> Fraction(2, 6)
Fraction(1, 3)
>>> Fraction(100)
Fraction(100, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' 3/7 ')
Fraction(3, 7)
>>> Fraction('3.1415')
Fraction(6283, 2000)
```

```
>>> Fraction(3.1415)
Fraction(7074029114692207, 2251799813685248)
```

Необходимо заметить, что, поскольку числа с плавающей точкой не совсем точны, получающееся рациональное число может отличаться от того, что мы хотим получить. Можете поделить столбиком 7074029114692207 на 2251799813685248 и убедиться :)

Рациональные числа можно, как int и float, складывать, умножать, делить...

```
>>> from fractions import Fraction
>>> a = Fraction(1, 7)
>>> b = Fraction(1, 3)
>>> a + b
Fraction(10, 21)
>>> a - b
Fraction(-4, 21)
>>> a * b
Fraction(1, 21)
>>> a / b
Fraction(3, 7)
>>> a % b
Fraction(1, 7)
>>> b % a
Fraction(1, 21)
>>> a ** b
0.5227579585747102
>>> abs(a - b)
Fraction(4, 21)
```

**Fraction.limit\_denominator**(max\_denominator=1000000) - ближайшее рациональное число со знаменателем не больше данного.

```
>>> from fractions import Fraction
>>> a = Fraction(3.1415)
>>> a
Fraction(7074029114692207, 2251799813685248)
>>> a.limit_denominator()
Fraction(6283, 2000)
```

Также, помимо класса рациональных чисел, модуль fractions предоставляет функцию для нахождения наибольшего общего делителя.

**fractions.gcd(a, b)** - наибольший общий делитель чисел a и b.

```
>>> from fractions import gcd
>>> gcd(1, 5)
1
>>> gcd(1000, 3)
1
>>> gcd(4, 6)
2
>>> gcd(0, 2)
2
>>> gcd(0, 0)
0
```

## Модуль cmath

Модуль cmath – предоставляет функции для работы с комплексными числами.

**cmath.phase(x)** - возвращает фазу комплексного числа (её ещё называют аргументом). Эквивалентно `math.atan2(x.imag, x.real)`. Результат лежит в промежутке  $[-\pi, \pi]$ .

Получить модуль комплексного числа можно с помощью встроенной функции `abs()`.

**cmath.polar(x)** - преобразование к полярным координатам. Возвращает пару (r, phi).

**cmath.rect(r, phi)** - преобразование из полярных координат.

**cmath.exp(x)** -  $e^x$ .

**cmath.log(x[, base])** - логарифм x по основанию base. Если base не указан, возвращается натуральный логарифм.

**cmath.log10(x)** - десятичный логарифм.

**cmath.sqrt(x)** - квадратный корень из x.

**cmath.acos(x)** - арккосинус x.

**cmath.asin(x)** - арксинус x.

**cmath.atan(x)** - арктангенс x.

**cmath.cos(x)** - косинус x.

**cmath.sin(x)** - синус x.

**cmath.tan(x)** - тангенс x.

**cmath.acosh(x)** - гиперболический арккосинус x.

**cmath.asinh(x)** - гиперболический арксинус x.

**cmath.atanh(x)** - гиперболический арктангенс x.

**cmath.cosh(x)** - гиперболический косинус x.

**cmath.sinh(x)** - гиперболический синус x.

**cmath.tanh(x)** - гиперболический тангенс x.

**cmath.isfinite(x)** - True, если действительная и мнимая части конечны.

**cmath.isinf(x)** - True, если либо действительная, либо мнимая часть бесконечна.

**cmath.isnan(x)** - True, если либо действительная, либо мнимая часть NaN.

**cmath.pi** -  $\pi$ .

**cmath.e** - e.

## Модуль glob

Модуль glob находит все пути, совпадающие с заданным шаблоном в соответствии с правилами, используемыми оболочкой Unix. Обработываются



символы "\*" (произвольное количество символов), "?" (один символ), и диапазоны символов с помощью []. Для использования тильды "~" и переменных окружения необходимо использовать `os.path.expanduser()` и `os.path.expandvars()`.

Для поиска спецсимволов, заключайте их в квадратные скобки. Например, `[?]` соответствует символу "?".

**glob.glob(pathname)** возвращение список (возможно, пустой) путей, соответствующих шаблону `pathname`. Путь может быть как абсолютным (например, `/usr/src/Python-1.5/Makefile`) или относительный (как `../Tools/*/*.gif`).

**glob.iglob(pathname)** - возвращает итератор, дающий те же значения, что и `glob.glob`.

**glob.escape(pathname)** - экранирует все специальные символы для `glob` ("?", "\*", и "["). Специальные символы в имени диска не экранируются (так как они там не учитываются), то есть в Windows `escape("///?/c:/Quo vadis?.txt")` возвращает `///?/c:/Quo vadis[?].txt`. (новое в python 3.4).

Рассмотрим, например, каталог, содержащий только следующие файлы: `1.gif`, `2.txt` и `card.gif`. `glob.glob()` вернёт следующие результаты. Обратите внимание, что любые ведущие компоненты пути сохраняются.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

Если каталог содержит файлы, начинающиеся с ".", они не будут включаться по умолчанию. Рассмотрим, например, каталог, содержащий `card.gif` и `.card.gif`:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

# Модуль `copy` - поверхностное и глубокое копирование объектов

Операция присваивания не копирует объект, он лишь создаёт ссылку на объект. Для изменяемых коллекций, или для коллекций, содержащих изменяемые элементы, часто необходима такая копия, чтобы её можно было изменить, не изменяя оригинал. Данный модуль предоставляет общие (поверхностная и глубокая) операции копирования.

`copy.copy(x)` - возвращает поверхностную копию `x`.

`copy.deepcopy(x)` - возвращает полную копию `x`.

Исключение `copy.error` - возникает, если объект невозможно скопировать.

Разница между поверхностным и глубоким копированием существенна только для составных объектов, содержащих изменяемые объекты (например, список списков, или словарь, в качестве значений которого - списки или словари):

- **Поверхностная копия** создает новый составной объект, и затем (по мере возможности) вставляет в него ссылки на объекты, находящиеся в оригинале.
- **Глубокая копия** создает новый составной объект, и затем рекурсивно вставляет в него копии объектов, находящихся в оригинале.

```
>>> import copy
>>> test_1 = [1, 2, 3, [1, 2, 3]]
>>> test_copy = copy.copy(test_1)
>>> print(test_1, test_copy)
[1, 2, 3, [1, 2, 3]] [1, 2, 3, [1, 2, 3]]
>>> test_copy[3].append(4)
>>> print(test_1, test_copy)
[1, 2, 3, [1, 2, 3, 4]] [1, 2, 3, [1, 2, 3, 4]]
>>> test_1 = [1, 2, 3, [1, 2, 3]]
>>> test_deepcopy = copy.deepcopy(test_1)
>>> test_deepcopy[3].append(4)
>>> print(test_1, test_deepcopy)
[1, 2, 3, [1, 2, 3]] [1, 2, 3, [1, 2, 3, 4]]
```

Для операции глубокого копирования часто возникают две проблемы, которых нет у операции поверхностного копирования:

- Рекурсивные объекты (составные объекты, которые явно или неявно содержат ссылки на себя) могут стать причиной рекурсивного цикла;
- Поскольку глубокая копия копирует всё, она может скопировать слишком много, например, административные структуры данных, которые должны быть разделяемы даже между копиями.

Функция `deepcopy` решает эти проблемы путем:

- Хранения "мето" словаря объектов, скопированных во время текущего прохода копирования;
- Позволения классам, определенным пользователем, переопределять операцию копирования или набор копируемых компонентов.

```
>>> r = [1, 2, 3]
>>> r.append(r)
>>> print(r)
[1, 2, 3, [...]]
>>> p = copy.deepcopy(r)
>>> print(p)
[1, 2, 3, [...]]
```

Этот модуль не копирует типы вроде модулей, классов, функций, методов, следа в стеке, стековых кадров, файлов, сокетов, окон, и подобных типов.

Поверхностная копия изменяемых объектов также может быть создана методом `.copy()` у списков(начиная с Python 3.3), присваиванием среза (`copied_list = original_list[:]`), методом `.copy()` словарей и множеств. Создавать копию неизменяемых объектов (таких, как, например, строк) необязательно (они же неизменяемые).

Для того, чтобы определить собственную реализацию копирования, класс может определить специальные методы `__copy__()` и `__deepcopy__()`. Первый вызывается для реализации операции поверхностного копирования; дополнительных аргументов не передается. Второй вызывается для реализации операции глубокого копирования; ему передается один аргумент, словарь `memo`. Если реализация `__deepcopy__()` нуждается в создании глубокой копии компонента, то он должен вызвать функцию `deepcopy()` с компонентом в качестве первого аргумента и словарем `memo` в качестве второго аргумента.

# Модуль **functools**

Модуль `functools` - сборник функций высокого уровня: взаимодействующих с другими функциями или возвращающие другие функции.

Модуль `functools` определяет следующие функции:

**`functools.cmp_to_key(func)`** - превращает функцию сравнения в `key`-функцию. Используется с инструментами, принимающие `key`-функции (`sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). Эта функция в основном используется в качестве переходного инструмента для программ, преобразованных из Python 2, которые поддерживали использование функций сравнения.

Функция сравнения - функция, принимающая два аргумента, сравнивающая их и возвращающая отрицательное число, если первый аргумент меньше, ноль, если равен и положительное число, если больше. `Key`-функция - функция, принимающая один аргумент и возвращающая другое значение, определяющее положение аргумента при сортировке.

**`@functools.lru_cache(maxsize=128, typed=False)`** - декоратор, который сохраняет результаты `maxsize` последних вызовов. Это может сэкономить время при дорогих вычислениях, если функция периодически вызывается с теми же аргументами.

Поскольку в качестве кэша используется словарь, все аргументы должны быть хешируемыми.

Если `maxsize` установлен в `None`, кэш может возрасти бесконечно. Также функция наиболее эффективна, если `maxsize` это степень двойки.

Если `typed` - `True`, аргументы функции с разными типами будут кэшироваться отдельно. Например, `f(3)` и `f(3.0)` будут считаться разными вызовами, возвращающие, возможно, различный результат.

Чтобы помочь измерить эффективность кэширования и отрегулировать размер кэша, обернутая функция дополняется функцией `cache_info()`, возвращающая `namedtuple`, показывающий попадания в кэш, промахи, максимальный размер и текущий размер. В многопоточном окружении, количество попаданий и промахов приблизительно.

Также имеется функция `cache_clear()` для очистки кэша.

Оригинальная функция доступна через атрибут `__wrapped__`.

```
from functools import lru_cache

# Пример получения веб-страниц
import urllib.request

@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/'
    % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320,
9991:
    pep = get_pep(n)
    print(n, len(pep))

print(get_pep.cache_info())
# CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)

# Числа Фибоначчи (попробуйте убрать lru_cache) :)

@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

```
print([fib(n) for n in range(100)])
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
377, 610...]

print(fib.cache_info())
# CacheInfo(hits=196, misses=100, maxsize=None,
currsz=100)
```

**@functools.total\_ordering** - декоратор класса, в котором задан один или более методов сравнения. Этот декоратор автоматически добавляет все остальные методы. Класс должен определять один из методов `__lt__()`, `__le__()`, `__gt__()`, или `__ge__()`. Кроме того, он должен определять метод `__eq__()`.

Например:

```
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(),
self.firstname.lower()) ==
                (other.lastname.lower(),
other.firstname.lower()))
    def __lt__(self, other):
        return ((self.lastname.lower(),
self.firstname.lower()) <
                (other.lastname.lower(),
other.firstname.lower()))
```

**functools.partial(func, \*args, \*\*keywords)** - возвращает partial-объект (по сути, функцию), который при вызове вызывается как функция `func`, но дополнительно передают туда позиционные аргументы `args`, и именованные аргументы `kwargs`. Если другие аргументы передаются при вызове функции, то позиционные добавляются в конец, а именованные расширяют и перезаписывают.

Например:

```
from functools import partial
basetwo = partial(int, base=2)
basetwo.__doc__ = 'Convert base 2 string to an int.'
print(basetwo('10010'))
# 18
```

**functools.reduce**(function, iterable[, initializer]) - берёт два первых элемента, применяет к ним функцию, берёт значение и третий элемент, и таким образом сворачивает iterable в одно значение. Например, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` эквивалентно `((((1+2)+3)+4)+5)`. Если задан `initializer`, он помещается в начале последовательности.

**functools.update\_wrapper**(wrapper, wrapped, assigned=WRAPPER\_ASSIGNMENTS, updated=WRAPPER\_UPDATES) - обновляет функцию-оболочку, чтобы она стала похожей на обёрнутую функцию. `assigned` - кортеж, указывающий, какие атрибуты исходной функции копируются в функцию-оболочку (по умолчанию это `WRAPPER_ASSIGNMENTS` (`__name__`, `__module__`, `__annotations__` и `__doc__`)). `updated` - кортеж, указывающий, какие атрибуты обновляются (по умолчанию это `WRAPPER_UPDATES` (обновляется `__dict__` функции-оболочки)).

**@functools.wraps**(wrapped, assigned=WRAPPER\_ASSIGNMENTS, updated=WRAPPER\_UPDATES) - удобная функция для вызова `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)` как [декоратора](#) при определении функции-оболочки. Например:

```
from functools import wraps
def my_decorator(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        print('Calling decorated function')
        return f(*args, **kwargs)
    return wrapper

@my_decorator
def example():
    """Docstring"""
    print('Called example function')

example()
# Calling decorated function
# Called example function
print(example.__name__)
# example
print(example.__doc__)
# Docstring
```

# Модуль **os.path**

**os.path** является вложенным модулем в модуль **os**, и реализует некоторые полезные функции на работы с путями.

**os.path.abspath(path)** - возвращает нормализованный абсолютный путь.

**os.path.basename(path)** - базовое имя пути (эквивалентно **os.path.split(path)[1]**).

**os.path.commonprefix(list)** - возвращает самый длинный префикс всех путей в списке.

**os.path.dirname(path)** - возвращает имя директории пути **path**.

**os.path.exists(path)** - возвращает **True**, если **path** указывает на существующий путь или дескриптор открытого файла.

**os.path.expanduser(path)** - заменяет **~** или **~user** на домашнюю директорию пользователя.

**os.path.expandvars(path)** - возвращает аргумент с подставленными переменными окружения (**\$name** или **\${name}** заменяются переменной окружения **name**). Несуществующие имена не заменяет. На Windows также заменяет **%name%**.

**os.path.getatime(path)** - время последнего доступа к файлу, в секундах.

**os.path.getmtime(path)** - время последнего изменения файла, в секундах.

**os.path.getctime(path)** - время создания файла (Windows), время последнего изменения файла (Unix).

**os.path.getsize(path)** - размер файла в байтах.

**os.path.isabs(path)** - является ли путь абсолютным.

**os.path.isfile(path)** - является ли путь файлом.

**os.path.isdir(path)** - является ли путь директорией.

**os.path.islink(path)** - является ли путь символической ссылкой.

**os.path.ismount(path)** - является ли путь точкой монтирования.



**os.path.join**(path1[, path2[, ...]]) - соединяет пути с учётом особенностей операционной системы.

**os.path.normcase**(path) - нормализует регистр пути (на файловых системах, не учитывающих регистр, приводит путь к нижнему регистру).

**os.path.normpath**(path) - нормализует путь, убирая избыточные разделители и ссылки на предыдущие директории. На Windows преобразует прямые слешы в обратные.

**os.path.realpath**(path) - возвращает канонический путь, убирая все символические ссылки (если они поддерживаются).

**os.path.relpath**(path, start=None) - вычисляет путь относительно директории start (по умолчанию - относительно текущей директории).

**os.path.samefile**(path1, path2) - указывают ли path1 и path2 на один и тот же файл или директорию.

**os.path.sameopenfile**(fp1, fp2) - указывают ли дескрипторы fp1 и fp2 на один и тот же открытый файл.

**os.path.split**(path) - разбивает путь на кортеж (голова, хвост), где хвост - последний компонент пути, а голова - всё остальное. Хвост никогда не начинается со слеша (если путь заканчивается слешем, то хвост пустой). Если слешей в пути нет, то пустой будет голова.

**os.path.splitdrive**(path) - разбивает путь на пару (привод, хвост).

**os.path.splitext**(path) - разбивает путь на пару (root, ext), где ext начинается с точки и содержит не более одной точки.

**os.path.supports\_unicode\_filenames** - поддерживает ли файловая система Unicode.

## Модуль json

JSON (JavaScript Object Notation) - простой формат обмена данными, основанный на подмножестве синтаксиса JavaScript. Модуль json позволяет кодировать и декодировать данные в удобном формате.

Кодирование основных объектов Python:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\foo\bar"))
"\foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\''))
"\'"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0},
sort_keys=True))
{"a": 0, "b": 0, "c": 0}
```

Компактное кодирование:

```
>>> import json
>>> json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',',
':'))
'[1,2,3,{"4":5,"6":7}]'
```

Красивый вывод:

```
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True,
indent=4))
{
    "4": 5,
    "6": 7
}
```

Декодирование (парсинг) JSON:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\"foo\\bar\"')
'foo\x08ar'
```

## ОСНОВЫ

**json.dump**(obj, fp, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw) - сериализует obj как форматированный JSON поток в fp.

Если `skipkeys = True`, то ключи словаря не базового типа (`str`, `unicode`, `int`, `long`, `float`, `bool`, `None`) будут проигнорированы, вместо того, чтобы вызывать исключение `TypeError`.

Если `ensure_ascii = True`, все не-ASCII символы в выводе будут экранированы последовательностями `\uXXXX`, и результатом будет строка, содержащая только ASCII символы. Если `ensure_ascii = False`, строки запишутся как есть.

Если `check_circular = False`, то проверка циклических ссылок будет пропущена, а такие ссылки будут вызывать `OverflowError`.

Если `allow_nan = False`, при попытке сериализовать значение с запятой, выходящее за допустимые пределы, будет вызываться `ValueError` (`nan`, `inf`, `-inf`) в строгом соответствии со спецификацией JSON, вместо того, чтобы использовать эквиваленты из JavaScript (`NaN`, `Infinity`, `-Infinity`).

Если `indent` является неотрицательным числом, то массивы и объекты в JSON будут выводиться с этим уровнем отступа. Если уровень отступа 0, отрицательный или `""`, то вместо этого будут просто использоваться новые строки. Значение по умолчанию `None` отражает наиболее компактное представление. Если `indent` - строка, то она и будет использоваться в качестве отступа.

Если `sort_keys = True`, то ключи выводимого словаря будут отсортированы.

**json.dumps**(obj, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw) - сериализует obj в строку JSON-формата.

Аргументы имеют то же значение, что и для `dump()`.

Ключи в парах ключ/значение в JSON всегда являются строками. Когда словарь конвертируется в JSON, все ключи словаря преобразовываются в строки. В результате этого, если словарь сначала преобразовать в JSON, а потом обратно в словарь, то можно не получить словарь, идентичный исходному. Другими словами, `loads(dumps(x)) != x`, если x имеет нестроковые ключи.

**json.load**(fp, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw) - десериализует JSON из fp.

`object_hook` - опциональная функция, которая применяется к результату декодирования объекта (dict). Используется будет значение, возвращаемое этой функцией, а не полученный словарь.

`object_pairs_hook` - опциональная функция, которая применяется к результату декодирования объекта с определённой последовательностью пар ключ/значение. Будет использован результат, возвращаемый функцией, вместо исходного словаря. Если задан так же `object_hook`, то приоритет отдаётся `object_pairs_hook`.

`parse_float`, если определён, будет вызван для каждого значения JSON с плавающей точкой. По умолчанию, это эквивалентно `float(num_str)`.

`parse_int`, если определён, будет вызван для строки JSON с числовым значением. По умолчанию эквивалентно `int(num_str)`.

`parse_constant`, если определён, будет вызван для следующих строк: `"-Infinity"`, `"Infinity"`, `"NaN"`. Может быть использовано для возбуждения исключений при обнаружении ошибочных чисел JSON.

Если не удастся десериализовать JSON, будет возбуждено исключение `ValueError`.

**`json.loads(s, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)`** - десериализует `s` (экземпляр `str`, содержащий документ JSON) в объект Python.

Остальные аргументы аналогичны аргументам в `load()`.

## Кодировщики и декодировщики

Класс **`json.JSONDecoder(object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None)`** - простой декодер JSON.

Выполняет следующие преобразования при декодировании:

| JSON         | Python |
|--------------|--------|
| object       | dict   |
| array        | list   |
| string       | str    |
| number (int) | int    |

| JSON          | Python |
|---------------|--------|
| number (real) | float  |
| true          | True   |
| false         | False  |
| null          | None   |

Он также понимает NaN, Infinity, и -Infinity как соответствующие значения float, которые находятся за пределами спецификации JSON.

Класс **json.JSONEncoder**(skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, sort\_keys=False, indent=None, separators=None, default=None)

Расширяемый кодировщик JSON для структур данных Python. Поддерживает следующие объекты и типы данных по умолчанию:

| Python      | JSON   |
|-------------|--------|
| dict        | object |
| list, tuple | array  |
| str         | string |
| int, float  | number |
| True        | true   |
| False       | false  |
| None        | null   |

## Модуль calendar

Модуль calendar позволяет напечатать себе календарик (а также содержит некоторые другие полезные функции для работы с календарями).

**calendar.Calendar**(firstweekday=0) - класс календаря. firstweekday - первый день недели (0 - понедельник, 6 - воскресенье).

Методы:

**iterweekdays()** - итератор дней недели, начиная с firstweekday.

**itermonthdates**(year, month) - итератор для месяца month года year. Возвращает все дни этого месяца (как объекты datetime.date), а также дни до и после этого месяца до полной недели.

**itermonthdays2**(year, month) - как itermonthdates, только дни возвращаются не как datetime.date объекты, а кортежи (номер дня, номер дня недели).

**itermonthdays**(year, month) - как itermonthdates, только дни возвращаются не как datetime.date объекты, а номера дней.

**monthdatescalendar**(year, month) - список недель в месяце. Неделя - список из 7 объектов datetime.date.

**monthdays2calendar**(year, month) - как monthdatescalendar, но объекты - кортежи (номер дня, номер дня недели).

**monthdayscalendar**(year, month) - как monthdatescalendar, но объекты - номера дней.

**calendar.TextCalendar**(firstweekday=0) - класс для генерации текстового календаря.

Методы:

**formatmonth**(theyear, themonth, w=0, l=0) - возвращает календарь на месяц в виде строки, с шириной колонки w и высотой l.

**prmonth**(theyear, themonth, w=0, l=0) - печатает календарь на месяц.

**formatyear**(theyear, w=2, l=1, c=6, m=3) - возвращает календарь на год; из m колонок, шириной даты w, высотой недели l и количеством пробелов между месяцами c.

**pryear**(theyear, w=2, l=1, c=6, m=3) - печатает календарь на год.

**calendar.HTMLCalendar**(firstweekday=0) - класс для генерации HTML календаря.

Методы:

**formatmonth**(theyear, themonth, withyear=True) - календарь на месяц в виде HTML таблицы. Если withyear True, номер года будет включен в заголовок.

**formatyear**(theyear, width=3) - календарь на год в виде HTML таблицы. width - количество месяцев в ряду.

**formatyearpage**(theyear, width=3, css="calendar.css", encoding=None) - календарь на год в виде полноценной HTML страницы, с подключением файла css (который вы можете создать сами), и в кодировке encoding.

**calendar.LocaleTextCalendar**(firstweekday=0, locale=None) - позволяет создать текстовый календарь с названиями на родном языке.

**calendar.LocaleHTMLCalendar**(firstweekday=0, locale=None) - позволяет создать HTML календарь с названиями на родном языке.

Например, вот такой календарик получился у меня:

```
import calendar
a = calendar.LocaleHTMLCalendar(locale='Russian_Russia')
with open('calendar.html', 'w') as g:
    print(a.formatyear(2014, width=4), file=g)
```

| 2014                                                                                                                              |                                                                                                                          |                                                                                                                                      |                                                                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| Январь                                                                                                                            | Февраль                                                                                                                  | Март                                                                                                                                 | Апрель                                                                                                                         |
| П В С Ч П С В<br>н т р т т б с<br>1 2 3 4 5<br>6 7 8 9 10 11 12<br>13 14 15 16 17 18 19<br>20 21 22 23 24 25 26<br>27 28 29 30 31 | П В С Ч П С В<br>н т р т т б с<br>1 2<br>3 4 5 6 7 8 9<br>10 11 12 13 14 15 16<br>17 18 19 20 21 22 23<br>24 25 26 27 28 | П В С Ч П С В<br>н т р т т б с<br>1 2<br>3 4 5 6 7 8 9<br>10 11 12 13 14 15 16<br>17 18 19 20 21 22 23<br>24 25 26 27 28 29 30<br>31 | П В С Ч П С В<br>н т р т т б с<br>1 2 3 4 5 6<br>7 8 9 10 11 12 13<br>14 15 16 17 18 19 20<br>21 22 23 24 25 26 27<br>28 29 30 |
| Май                                                                                                                               | Июнь                                                                                                                     | Июль                                                                                                                                 | Август                                                                                                                         |
| П В С Ч П С В<br>н т р т т б с<br>1 2 3 4<br>5 6 7 8 9 10 11<br>12 13 14 15 16 17 18                                              | П В С Ч П С В<br>н т р т т б с<br>1<br>2 3 4 5 6 7 8<br>9 10 11 12 13 14 15                                              | П В С Ч П С В<br>н т р т т б с<br>1 2 3 4 5 6<br>7 8 9 10 11 12 13<br>14 15 16 17 18 19 20                                           | П В С Ч П С В<br>н т р т т б с<br>1 2 3<br>4 5 6 7 8 9 10<br>11 12 13 14 15 16 17                                              |

|                      |                      |                      |                      |
|----------------------|----------------------|----------------------|----------------------|
| 19 20 21 22 23 24 25 | 16 17 18 19 20 21 22 | 21 22 23 24 25 26 27 | 18 19 20 21 22 23 24 |
| 26 27 28 29 30 31    | 23 24 25 26 27 28 29 | 28 29 30 31          | 25 26 27 28 29 30 31 |
|                      | 30                   |                      |                      |
| <b>Сентябрь</b>      | <b>Октябрь</b>       | <b>Ноябрь</b>        | <b>Декабрь</b>       |
| П В С Ч П С В        | П В С Ч П С В        | П В С Ч П С В        | П В С Ч П С В        |
| н т р т т б с        | н т р т т б с        | н т р т т б с        | н т р т т б с        |
| 1 2 3 4 5 6 7        | 1 2 3 4 5            | 1 2                  | 1 2 3 4 5 6 7        |
| 8 9 10 11 12 13 14   | 6 7 8 9 10 11 12     | 3 4 5 6 7 8 9        | 8 9 10 11 12 13 14   |
| 15 16 17 18 19 20 21 | 13 14 15 16 17 18 19 | 10 11 12 13 14 15 16 | 15 16 17 18 19 20 21 |
| 22 23 24 25 26 27 28 | 20 21 22 23 24 25 26 | 17 18 19 20 21 22 23 | 22 23 24 25 26 27 28 |
| 29 30                | 27 28 29 30 31       | 24 25 26 27 28 29 30 | 29 30 31             |

Также модуль `calendar` предоставляет несколько полезных функций:

**`calendar.setfirstweekday(weekday)`** - устанавливает первый день недели (0 - понедельник, 6 - воскресенье). Также предоставлены значения `calendar.MONDAY`, `calendar.TUESDAY`, `calendar.WEDNESDAY`, `calendar.THURSDAY`, `calendar.FRIDAY`, `calendar.SATURDAY` и `calendar.SUNDAY`.

**`calendar.firstweekday()`** - возвращает первый день недели.

**`calendar.isleap(year)`** - является ли год високосным.

**`calendar.leapdays(y1, y2)`** - количество високосных лет в последовательности от `y1` до `y2`.

**`calendar.weekday(year, month, day)`** - день недели для этой даты.

**`calendar.monthrange(year, month)`** - день недели первого дня месяца и количество дней в этом месяце.

## Модуль `os`

Модуль `os` предоставляет множество функций для работы с операционной системой, причём их поведение, как правило, не зависит от ОС, поэтому программы остаются переносимыми. Здесь будут приведены наиболее часто используемые из них.

Будьте внимательны: некоторые функции из этого модуля поддерживаются не всеми ОС.



**os.name** - имя операционной системы. Доступные варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

**os.environ** - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

**os.getlogin()** - имя пользователя, вошедшего в терминал (Unix).

**os.getpid()** - текущий id процесса.

**os.uname()** - информация об ОС. возвращает объект с атрибутами: sysname - имя операционной системы, nodename - имя машины в сети (определяется реализацией), release - релиз, version - версия, machine - идентификатор машины.

**os.access(path, mode, \*, dir\_fd=None, effective\_ids=False, follow\_symlinks=True)** - проверка доступа к объекту у текущего пользователя. Флаги: **os.F\_OK** - объект существует, **os.R\_OK** - доступен на чтение, **os.W\_OK** - доступен на запись, **os.X\_OK** - доступен на исполнение.

**os.chdir(path)** - смена текущей директории.

**os.chmod(path, mode, \*, dir\_fd=None, follow\_symlinks=True)** - смена прав доступа к объекту (mode - восьмеричное число).

**os.chown(path, uid, gid, \*, dir\_fd=None, follow\_symlinks=True)** - меняет id владельца и группы (Unix).

**os.getcwd()** - текущая рабочая директория.

**os.link(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None, follow\_symlinks=True)** - создаёт жёсткую ссылку.

**os.listdir(path=".")** - список файлов и директорий в папке.

**os.mkdir(path, mode=0o777, \*, dir\_fd=None)** - создаёт директорию. OSError, если директория существует.

**os.makedirs(path, mode=0o777, exist\_ok=False)** - создаёт директорию, создавая при этом промежуточные директории.

**os.remove(path, \*, dir\_fd=None)** - удаляет путь к файлу.

**os.rename**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None) - переименовывает файл или директорию из src в dst.

**os.rename**(old, new) - переименовывает old в new, создавая промежуточные директории.

**os.replace**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None) - переименовывает из src в dst с принудительной заменой.

**os.rmdir**(path, \*, dir\_fd=None) - удаляет пустую директорию.

**os.removedirs**(path) - удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

**os.symlink**(source, link\_name, target\_is\_directory=False, \*, dir\_fd=None) - создаёт символическую ссылку на объект.

**os.sync**() - записывает все данные на диск (Unix).

**os.truncate**(path, length) - обрезает файл до длины length.

**os.utime**(path, times=None, \*, ns=None, dir\_fd=None, follow\_symlinks=True) - модификация времени последнего доступа и изменения файла. Либо times - кортеж (время доступа в секундах, время изменения в секундах), либо ns - кортеж (время доступа в наносекундах, время изменения в наносекундах).

**os.walk**(top, topdown=True, onerror=None, followlinks=False) - генерация имён файлов в дереве каталогов, сверху вниз (если topdown равен True), либо снизу вверх (если False). Для каждого каталога функция walk возвращает кортеж (путь к каталогу, список каталогов, список файлов).

**os.system**(command) - исполняет системную команду.

**os.urandom**(n) - n случайных байт. Возможно использование этой функции в криптографических целях.

os.path - модуль, реализующий некоторые полезные функции на работы с путями.

## Модуль pickle

Модуль pickle реализует мощный алгоритм сериализации и десериализации объектов Python. "Pickling" - процесс преобразования объекта Python в поток

байтов, а "unpickling" - обратная операция, в результате которой поток байтов преобразуется обратно в Python-объект. Так как поток байтов легко можно записать в файл, модуль pickle широко применяется для сохранения и загрузки сложных объектов в Python.

Не загружайте с помощью модуля pickle файлы из ненадёжных источников. Это может привести к необратимым последствиям.

Модуль pickle предоставляет следующие функции для удобства сохранения/загрузки объектов:

**pickle.dump(obj, file, protocol=None, \*, fix\_imports=True)** - записывает сериализованный объект в файл. Дополнительный аргумент protocol указывает используемый протокол. По умолчанию равен 3 и именно он рекомендован для использования в Python 3 (несмотря на то, что в Python 3.4 добавили протокол версии 4 с некоторыми оптимизациями). В любом случае, записывать и загружать надо с одним и тем же протоколом.

**pickle.dumps(obj, protocol=None, \*, fix\_imports=True)** - возвращает сериализованный объект. Впоследствии вы его можете использовать как угодно.

**pickle.load(file, \*, fix\_imports=True, encoding="ASCII", errors="strict")** - загружает объект из файла.

**pickle.loads(bytes\_object, \*, fix\_imports=True, encoding="ASCII", errors="strict")** - загружает объект из потока байт.

Модуль pickle также определяет несколько исключений:

- **pickle.PickleError**
- o **pickle.PicklingError** - случились проблемы с сериализацией объекта.
- o **pickle.UnpicklingError** - случились проблемы с десериализацией объекта.

Этих функций вполне достаточно для сохранения и загрузки встроенных типов данных.

```
import pickle
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
```

```

        'c': set([None, True, False])
    }

    with open('data.pickle', 'wb') as f:
        pickle.dump(data, f)

    with open('data.pickle', 'rb') as f:
        data_new = pickle.load(f)

    print(data_new)

```

## Модуль datetime

Модуль `datetime` предоставляет классы для обработки времени и даты разными способами. Поддерживается и стандартный способ представления времени, однако больший упор сделан на простоту манипулирования датой, временем и их частями.

### Классы, предоставляемые модулем `datetime`:

Класс **`datetime.date`**(year, month, day) - стандартная дата. Атрибуты: year, month, day. Неизменяемый объект.

Класс **`datetime.time`**(hour=0, minute=0, second=0, microsecond=0, tzinfo=None) - стандартное время, не зависит от даты. Атрибуты: hour, minute, second, microsecond, tzinfo.

Класс **`datetime.timedelta`** - разница между двумя моментами времени, с точностью до микросекунд.

Класс **`datetime.tzinfo`** - абстрактный базовый класс для информации о временной зоне (например, для учета часового пояса и / или летнего времени).

Класс **`datetime.datetime`**(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None) - комбинация даты и времени.

Обязательные аргументы:

- `datetime.MINYEAR (1) ≤ year ≤ datetime.MAXYEAR (9999)`

- $1 \leq \text{month} \leq 12$
- $1 \leq \text{day} \leq \text{количество дней в данном месяце и году}$

Необязательные:

- $0 \leq \text{minute} < 60$
- $0 \leq \text{second} < 60$
- $0 \leq \text{microsecond} < 1000000$

Методы класса `datetime`:

**`datetime.today()`** - текущая дата, время равно 0.

**`datetime.fromtimestamp(timestamp)`** - дата из стандартного представления времени.

**`datetime.fromordinal(ordinal)`** - дата из числа, представляющего собой количество дней, прошедших с 01.01.1970.

**`datetime.now(tz=None)`** - объект `datetime` из текущей даты и времени.

**`datetime.combine(date, time)`** - объект `datetime` из комбинации объектов `date` и `time`.

**`datetime.strptime(date_string, format)`** - преобразует строку в `datetime` (так же, как и функция `strptime` из модуля `time`).

**`datetime.strptime(format)`** - см. функцию `strptime` из модуля `time`.

**`datetime.date()`** - объект даты (с отсечением времени).

**`datetime.time()`** - объект времени (с отсечением даты).

**`datetime.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]])`** - возвращает новый объект `datetime` с изменёнными атрибутами.

**`datetime.timetuple()`** - возвращает `struct_time` из `datetime`.

**`datetime.toordinal()`** - количество дней, прошедших с 01.01.1970.

**`datetime.timestamp()`** - возвращает время в секундах с начала эпохи.

**datetime.weekday()** - день недели в виде числа, понедельник - 0, воскресенье - 6.

**datetime.isoweekday()** - день недели в виде числа, понедельник - 1, воскресенье - 7.

**datetime.isocalendar()** - кортеж (ISO year, ISO week number, ISO weekday).

**datetime.isoformat(sep="T")** - красивая строка вида "YYYY-MM-DDTHH:MM:SS.mmmmmmm" или, если microsecond == 0, "YYYY-MM-DDTHH:MM:SS"

**datetime.ctime()** - см. ctime() из модуля time.

Пример работы с классом datetime:

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)    # GMT
+1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y
%H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all
attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
```

```

0          # second
1          # weekday (0 = Monday)
325        # number of days since 1st January
-1         # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006       # ISO year
47         # ISO week
2          # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'

```

## Модуль bisect

Модуль **bisect** - обеспечивает поддержку списка в отсортированном порядке с помощью алгоритма деления пополам.

Набор функций:

**bisect.insort(list, elem)**, он же **bisect.insort\_right(list, elem)** - вставка элемента в отсортированный список, при этом elem располагается как можно правее (все элементы, равные ему, остаются слева).

**bisect.insort\_left(list, elem)** - вставка элемента в отсортированный список, при этом elem располагается как можно левее (все элементы, равные ему, остаются справа).

**bisect.bisect(list, elem)**, он же **bisect.bisect\_right(list, elem)** - поиск места для вставки элемента в отсортированный список, таким образом, чтобы elem располагался как можно правее.

**bisect.bisect\_left(list, elem)** - поиск места для вставки элемента в отсортированный список, таким образом, чтобы elem располагался как можно левее.

Для полного счастья не хватает только функции для проверки наличия элемента в отсортированном списке. К счастью, это легко решается.

```

from bisect import bisect_left

def contains(l, elem):
    index = bisect_left(l, elem)
    if index < len(l):
        return l[index] == elem
    return False

```

И пример работы:

```

>>> contains(list(range(1000)), -10)
False
>>> testlist = (1, 2, 3, 6, 8, 10, 15)
>>> contains(testlist, 10)
True
>>> contains(testlist, 0)
False
>>> contains(testlist, 20)
False

```

## Модуль collections

Модуль **collections** - предоставляет специализированные типы данных, на основе словарей, кортежей, множеств, списков.

Первым рассматриваемым типом данных будет Counter.

### collections.Counter

**collections.Counter** - вид словаря, который позволяет нам считать количество неизменяемых объектов (в большинстве случаев, строк). Пример:

```

>>> import collections
>>> c = collections.Counter()
>>> for word in ['spam', 'egg', 'spam', 'counter',
'counter', 'counter']:
    c[word] += 1

>>> c
Counter({'counter': 3, 'spam': 2, 'egg': 1})

```



```
>>> c['counter']
3
>>> c['collections']
0
```

Но возможности Counter на этом не заканчиваются. У него есть несколько специальных методов:

**elements()** - возвращает список элементов в лексикографическом порядке.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

**most\_common([n])** - возвращает n наиболее часто встречающихся элементов, в порядке убывания встречаемости. Если n не указано, возвращаются все элементы.

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

**subtract([iterable-or-mapping])** - вычитание

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Наиболее часто употребляемые шаблоны для работы с Counter:

- `sum(c.values())` - общее количество.
- `c.clear()` - очистить счётчик.
- `list(c)` - список уникальных элементов.
- `set(c)` - преобразовать в множество.
- `dict(c)` - преобразовать в словарь.
- `c.most_common()[::-n:-1]` - n наименее часто встречающихся элементов.
- `c += Counter()` - удалить элементы, встречающиеся менее одного раза.

Counter также поддерживает сложение, вычитание, пересечение и объединение:

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d
Counter({'a': 4, 'b': 3})
>>> c - d
Counter({'a': 2})
>>> c & d
Counter({'a': 1, 'b': 1})
>>> c | d
Counter({'a': 3, 'b': 2})
```

Следующими на очереди у нас очереди (deque)

### **collections.deque**

**collections.deque**(iterable, [maxlen]) - создаёт очередь из итерируемого объекта с максимальной длиной maxlen. Очереди очень похожи на списки, за исключением того, что добавлять и удалять элементы можно либо справа, либо слева.

Методы, определённые в deque:

**append(x)** - добавляет x в конец.

**appendleft(x)** - добавляет x в начало.

**clear()** - очищает очередь.

**count(x)** - количество элементов, равных x.

**extend(iterable)** - добавляет в конец все элементы iterable.

**extendleft(iterable)** - добавляет в начало все элементы iterable (начиная с последнего элемента iterable).

**pop()** - удаляет и возвращает последний элемент очереди.

**popleft()** - удаляет и возвращает первый элемент очереди.

**remove(value)** - удаляет первое вхождение value.

**reverse()** - разворачивает очередь.

**rotate(n)** - последовательно переносит n элементов из начала в конец (если n отрицательно, то с конца в начало).

## **collections.defaultdict**

**collections.defaultdict** ничем не отличается от обычного словаря за исключением того, что по умолчанию всегда вызывается функция, возвращающая значение:

```
>>> import collections
>>> defdict = collections.defaultdict(list)
>>> defdict
defaultdict(<class 'list'>, {})
>>> for i in range(5):
    defdict[i].append(i)

>>> defdict
defaultdict(<class 'list'>, {0: [0], 1: [1], 2: [2], 3: [3], 4: [4]})
```

## **collections.OrderedDict**

**collections.OrderedDict** - ещё один похожий на словарь объект, но он помнит порядок, в котором ему были даны ключи. Методы:

**popitem(last=True)** - удаляет последний элемент если last=True, и первый, если last=False.

**move\_to\_end(key, last=True)** - добавляет ключ в конец если last=True, и в начало, если last=False.

```
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])
```

```
>>> OrderedDict(sorted(d.items(), key=lambda t:
len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2),
('banana', 3)])
```

## collections.namedtuple()

Класс **collections.namedtuple** позволяет создать тип данных, ведущий себя как кортеж, с тем дополнением, что каждому элементу присваивается имя, по которому можно в дальнейшем получать доступ:

```
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = point(x=1, y=2)
>>> p
Point(x=1, y=2)
>>> p.x
1
>>> p[0]
1
```

## Модуль array. Массивы в python

Модуль array определяет массивы в python. Массивы очень похожи на списки, но с ограничением на тип данных и размер каждого элемента.

Размер и тип элемента в массиве определяется при его создании и может принимать следующие значения:

| Код типа | Тип в C          | Тип в python | Минимальный размер в байтах |
|----------|------------------|--------------|-----------------------------|
| 'b'      | signed char      | int          | 1                           |
| 'B'      | unsigned char    | int          | 1                           |
| 'h'      | signed short     | int          | 2                           |
| 'H'      | unsigned short   | int          | 2                           |
| 'i'      | signed int       | int          | 2                           |
| 'I'      | unsigned int     | int          | 2                           |
| 'l'      | signed long      | int          | 4                           |
| 'L'      | unsigned long    | int          | 4                           |
| 'q'      | signed long long | int          | 8                           |

| Код типа | Тип в C            | Тип в python | Минимальный размер в байтах |
|----------|--------------------|--------------|-----------------------------|
| 'Q'      | unsigned long long | int          | 8                           |
| 'f'      | float              | float        | 4                           |
| 'd'      | double             | float        | 8                           |

Класс **array.array**(TypeCode [, инициализатор]) - новый массив, элементы которого ограничены TypeCode, и инициализатор, который должен быть списком, объектом, поддерживающий интерфейс буфера, или итерируемый объект.

**array.typecodes** - строка, содержащая все возможные типы в массиве.

**Массивы изменяемы.** Массивы поддерживают все списковые методы (индексация, срезы, умножения, итерации), и другие методы.

### Методы массивов (array) в python

**array.typecode** - TypeCode символ, использованный при создании массива.

**array.itemsize** - размер в байтах одного элемента в массиве.

**array.append(x)** - добавление элемента в конец массива.

**array.buffer\_info()** - кортеж (ячейка памяти, длина). Полезно для низкоуровневых операций.

**array.byteswap()** - изменить порядок следования байтов в каждом элементе массива. Полезно при чтении данных из файла, написанного на машине с другим порядком байтов.

**array.count(x)** - возвращает количество вхождений x в массив.

**array.extend(iter)** - добавление элементов из объекта в массив.

**array.frombytes(b)** - делает массив array из массива байт. Количество байт должно быть кратно размеру одного элемента в массиве.

**array.fromfile(F, N)** - читает N элементов из файла и добавляет их в конец массива. Файл должен быть открыт на бинарное чтение. Если доступно меньше

N элементов, генерируется исключение EOFError , но элементы, которые были доступны, добавляются в массив.

**array.fromlist(список)** - добавление элементов из списка.

**array.index(x)** - номер первого вхождения x в массив.

**array.insert(n, x)** - включить новый пункт со значением x в массиве перед номером n. Отрицательные значения рассматриваются относительно конца массива.

**array.pop(i)** - удаляет i-ый элемент из массива и возвращает его. По умолчанию удаляется последний элемент.

**array.remove(x)** - удалить первое вхождение x из массива.

**array.reverse()** - обратный порядок элементов в массиве.

**array.tobytes()** - преобразование к байтам.

**array.tofile(f)** - запись массива в открытый файл.

**array.tolist()** - преобразование массива в список.

Вот и всё, что можно было рассказать про массивы. Они используются редко, когда нужно достичь высокой скорости работы. В остальных случаях массивы можно заменить другими типами данных: списками, кортежами, строками.

## Модуль itertools

Модуль itertools - сборник полезных итераторов.

**itertools.count(start=0, step=1)** - бесконечная арифметическая прогрессия с первым членом start и шагом step.

**itertools.cycle(iterable)** - возвращает по одному значению из последовательности, повторенной бесконечное число раз.

**itertools.repeat(elem, n=Inf)** - повторяет elem n раз.

**itertools.accumulate(iterable)** - аккумулирует суммы.

```
accumulate([1, 2, 3, 4, 5]) --> 1 3 6 10 15
```

**itertools.chain**(\*iterables) - возвращает по одному элементу из первого итератора, потом из второго, до тех пор, пока итераторы не кончатся.

**itertools.combinations**(iterable, [r]) - комбинации длиной r из iterable без повторяющихся элементов.

```
combinations('ABCD', 2) --> AB AC AD BC BD CD
```

**itertools.combinations\_with\_replacement**(iterable, r) - комбинации длиной r из iterable с повторяющимися элементами.

```
combinations_with_replacement('ABCD', 2) --> AA AB AC AD  
BB BC BD CC CD DD
```

**itertools.compress**(data, selectors) - (d[0] if s[0]), (d[1] if s[1]), ...

```
compress('ABCDEF', [1, 0, 1, 0, 1, 1]) --> A C E F
```

**itertools.dropwhile**(func, iterable) - элементы iterable, начиная с первого, для которого func вернула ложь.

```
dropwhile(lambda x: x < 5, [1, 4, 6, 4, 1]) --> 6 4 1
```

**itertools.filterfalse**(func, iterable) - все элементы, для которых func возвращает ложь.

**itertools.groupby**(iterable, key=None) - группирует элементы по значению. Значение получается применением функции key к элементу (если аргумент key не указан, то значением является сам элемент).

```
from itertools import groupby  
things = [("animal", "bear"), ("animal", "duck"),  
          ("plant", "cactus"),  
          ("vehicle", "speed boat"), ("vehicle", "school  
bus")]
```

```

for key, group in groupby(things, lambda x: x[0]):
    for thing in group:
        print("A %s is a %s." % (thing[1], key))
    print()

# A bear is a animal.
# A duck is a animal.
#
# A cactus is a plant.
#
# A speed boat is a vehicle.
# A school bus is a vehicle.

```

**itertools.islice**(iterable[, start[, stop[, step]]) - итератор, состоящий из среза.

**itertools.permutations**(iterable, r=None) - перестановки длиной r из iterable.

**itertools.product**(\*iterables, repeat=1) - аналог вложенных циклов.

```
product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
```

**itertools.starmap**(function, iterable) - применяет функцию к каждому элементу последовательности (каждый элемент распаковывается).

```
starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
```

**itertools.takewhile**(func, iterable) - элементы до тех пор, пока func возвращает истину.

```
takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
```

**itertools.tee**(iterable, n=2) - кортеж из n итераторов.

**itertools.zip\_longest**(\*iterables, fillvalue=None) - как встроенная функция zip, но берет самый длинный итератор, а более короткие дополняет fillvalue.

```
zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
```

## Модуль time

Time - модуль для работы со временем в Python.



**time.altzone** - смещение DST часового пояса в секундах к западу от нулевого меридиана. Если часовой пояс находится восточнее, смещение отрицательно.

**time.asctime([t])** - преобразовывает кортеж или struct\_time в строку вида "Thu Sep 27 16:42:37 2012". Если аргумент не указан, используется текущее время.

**time.clock()** - в Unix, возвращает текущее время. В Windows, возвращает время, прошедшее с момента первого вызова данной функции.

**time.ctime([сек])** - преобразует время, выраженное в секундах с начала эпохи в строку вида "Thu Sep 27 16:42:37 2012".

**time.daylight** - не 0, если определено, зимнее время или летнее (DST).

**time.gmtime([сек])** - преобразует время, выраженное в секундах с начала эпохи в struct\_time, где DST флаг всегда равен нулю.

**time.localtime([сек])** - как gmtime, но с DST флагом.

**time.mktime([t])** - преобразует кортеж или struct\_time в число секунд с начала эпохи. Обратна функции time.localtime.

**time.sleep(сек)** - приостановить выполнение программы на заданное количество секунд.

**time.strftime(формат, [t])** - преобразует кортеж или struct\_time в строку по формату:

| Формат | Значение                        |
|--------|---------------------------------|
| %a     | Сокращенное название дня недели |
| %A     | Полное название дня недели      |
| %b     | Сокращенное название месяца     |
| %B     | Полное название месяца          |
| %c     | Дата и время                    |
| %d     | День месяца [01,31]             |
| %H     | Час (24-часовой формат) [00,23] |
| %I     | Час (12-часовой формат) [01,12] |
| %j     | День года [001,366]             |
| %m     | Номер месяца [01,12]            |
| %M     | Число минут [00,59]             |

| Формат | Значение                                                               |
|--------|------------------------------------------------------------------------|
| %p     | До полудня или после (при 12-часовом формате)                          |
| %S     | Число секунд [00,61] (2)                                               |
| %U     | Номер недели в году (нулевая неделя начинается с воскресенья) [00,53]  |
| %w     | Номер дня недели [0(Sunday),6]                                         |
| %W     | Номер недели в году (нулевая неделя начинается с понедельника) [00,53] |
| %x     | Дата                                                                   |
| %X     | Время                                                                  |
| %y     | Год без века [00,99]                                                   |
| %Y     | Год с веком                                                            |
| %Z     | Временная зона                                                         |
| %%     | Знак '%'                                                               |

**time.strptime**(строка [, формат]) - разбор строки, представляющей время в соответствии с форматом. Возвращаемое значение `struct_time`. Формат по умолчанию: "%a %b %d %H:%M:%S %Y".

Класс **time.struct\_time** - тип последовательности значения времени. Имеет интерфейс кортежа. Можно обращаться по индексу или по имени.

1. `tm_year`
2. `tm_mon`
3. `tm_mday`
4. `tm_hour`
5. `tm_min`
6. `tm_sec`
7. `tm_wday`
8. `tm_yday`
9. `tm_isdst`

**time.time()** - время, выраженное в секундах с начала эпохи.

**time.timezone** - смещение местного часового пояса в секундах к западу от нулевого меридиана. Если часовой пояс находится восточнее, смещение отрицательно.

**time.tzname** - кортеж из двух строк: первая - имя DST часового пояса, второй - имя местного часового пояса.

## Модуль **sys**

Модуль **sys** обеспечивает доступ к некоторым переменным и функциям, взаимодействующим с интерпретатором **python**.

**sys.argv** - список аргументов командной строки, передаваемых сценарию **Python**. **sys.argv[0]** является именем скрипта (пустой строкой в интерактивной оболочке).

**sys.byteorder** - порядок байтов. Будет иметь значение 'big' при порядке следования битов от старшего к младшему, и 'little', если наоборот (младший байт первый).

**sys.builtin\_module\_names** - кортеж строк, содержащий имена всех доступных модулей.

**sys.call\_tracing**(функция, аргументы) - вызывает функцию с аргументами и включенной трассировкой, в то время как трассировка включена.

**sys.copyright** - строка, содержащая авторские права, относящиеся к интерпретатору **Python**.

**sys.\_clear\_type\_cache()** - очищает внутренний кэш типа.

**sys.\_current\_frames()** - возвращает словарь-отображение идентификатора для каждого потока в верхнем кадре стека в настоящее время в этом потоке в момент вызова функции.

**sys.dllhandle** - целое число, определяющее дескриптор DLL **Python** (Windows).

**sys.exc\_info()** - возвращает кортеж из трех значений, которые дают информацию об исключениях, обрабатывающихся в данный момент.

**sys.exec\_prefix** - каталог установки **Python**.

**sys.executable** - путь к интерпретатору **Python**.

**sys.exit([arg])** - выход из **Python**. Возбуждает исключение **SystemExit**, которое может быть перехвачено.

**sys.flags** - флаги командной строки. Атрибуты только для чтения.

**sys.float\_info** - информация о типе данных float.

**sys.float\_repr\_style** - информация о применении встроенной функции repr() для типа float.

**sys.getdefaultencoding()** - возвращает используемую кодировку.

**sys.getdlopenflags()** - значения флагов для вызовов dlopen().

**sys.getfilesystemencoding()** - возвращает кодировку файловой системы.

**sys.getrefcount(object)** - возвращает количество ссылок на объект. Аргумент функции getrefcount - еще одна ссылка на объект.

**sys.getrecursionlimit()** - возвращает лимит рекурсии.

**sys.getsizeof(object[, default])** - возвращает размер объекта (в байтах).

**sys.getswitchinterval()** - интервал переключения потоков.

**sys.getwindowsversion()** - возвращает кортеж, описывающий версию Windows.

**sys.hash\_info** - информация о параметрах хэширования.

**sys.hexversion** - версия python как шестнадцатеричное число (для 3.2.2 final это будет 30202f0).

**sys.implementation** - объект, содержащий информацию о запущенном интерпретаторе python.

**sys.int\_info** - информация о типе int.

**sys.intern(строка)** - возвращает интернированную строку.

**sys.last\_type, sys.last\_value, sys.last\_traceback** - информация об обрабатываемых исключениях. По смыслу похоже на sys.exc\_info().

**sys.maxsize** - максимальное значение числа типа Py\_ssize\_t (2:sup:31 на 32-битных и 2<sup>63</sup> на 64-битных платформах).

**sys.maxunicode** - максимальное число бит для хранения символа Unicode.

**sys.modules** - словарь имен загруженных модулей. Изменяем, поэтому можно позабавиться :)

**sys.path** - список путей поиска модулей.

**sys.path\_importer\_cache** - словарь-кэш для поиска объектов.

**sys.platform** - информация об операционной системе.

|                     |          |
|---------------------|----------|
| Linux (2.x and 3.x) | 'linux'  |
| Windows             | 'win32'  |
| Windows/Cygwin      | 'cygwin' |
| Mac OS X            | 'darwin' |
| OS/2                | 'os2'    |
| OS/2 EMX            | 'os2emx' |

**sys.prefix** - папка установки интерпретатора python.

**sys.ps1, sys.ps2** - первичное и вторичное приглашение интерпретатора (определены только если интерпретатор находится в интерактивном режиме). По умолчанию `sys.ps1 == ">>> "`, а `sys.ps2 == "... "`.

**sys.dont\_write\_bytecode** - если true, python не будет писать .рус файлы.

**sys.setdlopenflags(flags)** - установить значения флагов для вызовов `dlopen()`.

**sys.setrecursionlimit(предел)** - установить максимальную глубину рекурсии.

**sys.setswitchinterval(интервал)** - установить интервал переключения потоков.

**sys.settrace(tracefunc)** - установить "след" функции.

**sys.stdin** - стандартный ввод.

**sys.stdout** - стандартный вывод.

**sys.stderr** - стандартный поток ошибок.

**sys.\_\_stdin\_\_, sys.\_\_stdout\_\_, sys.\_\_stderr\_\_** - исходные значения потоков ввода, вывода и ошибок.

**sys.tracebacklimit** - максимальное число уровней отслеживания.

**sys.version** - версия python.

**sys.api\_version** - версия C API.

**sys.version\_info** - Кортеж, содержащий пять компонентов номера версии.

**sys.warnoptions** - реализация предупреждений.

**sys.winver** - номер версии python, использующийся для формирования реестра Windows.

## Модуль random

Модуль random предоставляет функции для генерации случайных чисел, букв, случайного выбора элементов последовательности.

**random.seed([X], version=2)** - инициализация генератора случайных чисел. Если X не указан, используется системное время.

**random.getstate()** - внутреннее состояние генератора.

**random.setstate(state)** - восстанавливает внутреннее состояние генератора. Параметр state должен быть получен функцией getstate().

**random.getrandbits(N)** - возвращает N случайных бит.

**random.randrange(start, stop, step)** - возвращает случайно выбранное число из последовательности.

**random.randint(A, B)** - случайное целое число N,  $A \leq N \leq B$ .

**random.choice(sequence)** - случайный элемент непустой последовательности.

**random.shuffle(sequence, [rand])** - перемешивает последовательность (изменяется сама последовательность). Поэтому функция не работает для неизменяемых объектов.

**random.sample(population, k)** - список длиной k из последовательности population.

**random.random()** - случайное число от 0 до 1.

**random.uniform(A, B)** - случайное число с плавающей точкой,  $A \leq N \leq B$  (или  $B \leq N \leq A$ ).

**random.triangular(low, high, mode)** - случайное число с плавающей точкой,  $low \leq N \leq high$ . Mode - распределение.

**random.betavariate(alpha, beta)** - бета-распределение.  $alpha > 0$ ,  $beta > 0$ . Возвращает от 0 до 1.

**random.expovariate(lambd)** - экспоненциальное распределение. lambd равен  $1/\text{среднее желаемое}$ . Lambd должен быть отличным от нуля. Возвращаемые значения от 0 до плюс бесконечности, если lambd положительно, и от минус бесконечности до 0, если lambd отрицательный.

**random.gammavariate(alpha, beta)** - гамма-распределение. Условия на параметры  $alpha > 0$  и  $beta > 0$ .

**random.gauss(значение, стандартное отклонение)** - распределение Гаусса.

**random.lognormvariate(mu, sigma)** - логарифм нормального распределения. Если взять натуральный логарифм этого распределения, то вы получите нормальное распределение со средним mu и стандартным отклонением sigma. mu может иметь любое значение, и sigma должна быть больше нуля.

**random.normalvariate(mu, sigma)** - нормальное распределение. mu - среднее значение, sigma - стандартное отклонение.

**random.vonmisesvariate(mu, kappa)** - mu - средний угол, выраженный в радианах от 0 до  $2\pi$ , и kappa - параметр концентрации, который должен быть больше или равен нулю. Если kappa равна нулю, это распределение сводится к случайному углу в диапазоне от 0 до  $2\pi$ .

**random.paretovariate(alpha)** - распределение Парето.

**random.weibullvariate(alpha, beta)** - распределение Вейбулла.

## Модуль math

Модуль math – один из наиважнейших в Python. Этот модуль предоставляет обширный функционал для работы с числами.

**math.ceil(X)** – округление до ближайшего большего числа.

**math.copysign(X, Y)** - возвращает число, имеющее модуль такой же, как и у числа X, а знак - как у числа Y.

**math.fabs(X)** - модуль X.

**math.factorial(X)** - факториал числа X.

**math.floor(X)** - округление вниз.

**math.fmod(X, Y)** - остаток от деления X на Y.

**math.frexp(X)** - возвращает мантиссу и экспоненту числа.

**math.ldexp(X, I)** -  $X * 2^I$ . Функция, обратная функции **math.frexp()**.

**math.fsum(последовательность)** - сумма всех членов последовательности. Эквивалент встроенной функции **sum()**, но **math.fsum()** более точна для чисел с плавающей точкой.

**math.isfinite(X)** - является ли X числом.

**math.isinf(X)** - является ли X бесконечностью.

**math.isnan(X)** - является ли X NaN (Not a Number - не число).

**math.modf(X)** - возвращает дробную и целую часть числа X. Оба числа имеют тот же знак, что и X.

**math.trunc(X)** - усекает значение X до целого.

**math.exp(X)** -  $e^X$ .

**math.expm1(X)** -  $e^X - 1$ . При  $X \rightarrow 0$  точнее, чем **math.exp(X)-1**.

**math.log(X, [base])** - логарифм X по основанию base. Если base не указан, вычисляется натуральный логарифм.

**math.log1p(X)** - натуральный логарифм  $(1 + X)$ . При  $X \rightarrow 0$  точнее, чем **math.log(1+X)**.

**math.log10(X)** - логарифм X по основанию 10.

**math.log2(X)** - логарифм X по основанию 2. Новое в Python 3.3.



**math.pow(X, Y)** -  $X^Y$ .

**math.sqrt(X)** - квадратный корень из X.

**math.acos(X)** - арккосинус X. В радианах.

**math.asin(X)** - арксинус X. В радианах.

**math.atan(X)** - арктангенс X. В радианах.

**math.atan2(Y, X)** - арктангенс Y/X. В радианах. С учетом четверти, в которой находится точка (X, Y).

**math.cos(X)** - косинус X (X указывается в радианах).

**math.sin(X)** - синус X (X указывается в радианах).

**math.tan(X)** - тангенс X (X указывается в радианах).

**math.hypot(X, Y)** - вычисляет гипотенузу треугольника с катетами X и Y ( $\text{math.sqrt}(x * x + y * y)$ ).

**math.degrees(X)** - конвертирует радианы в градусы.

**math.radians(X)** - конвертирует градусы в радианы.

**math.cosh(X)** - вычисляет гиперболический косинус.

**math.sinh(X)** - вычисляет гиперболический синус.

**math.tanh(X)** - вычисляет гиперболический тангенс.

**math.acosh(X)** - вычисляет обратный гиперболический косинус.

**math.asinh(X)** - вычисляет обратный гиперболический синус.

**math.atanh(X)** - вычисляет обратный гиперболический тангенс.

**math.erf(X)** - функция ошибок.

**math.erfc(X)** - дополнительная функция ошибок ( $1 - \text{math.erf}(X)$ ).

**math.gamma(X)** - гамма-функция X.

`math.lgamma(X)` - натуральный логарифм гамма-функции X.

`math.pi` -  $\pi = 3,1415926\dots$

`math.e` -  $e = 2,718281\dots$

## Интерпретатор `hq9+`

Это всего лишь 4 команды:

- `H` - выводит "Hello, world!".
- `Q` - выводит текст исполняемой программы.
- `9` - выводит текст песни "99 Bottles of Beer".
- `+` - увеличивает никому не нужный счётчик.

Зная это, мы можем написать свой интерпретатор `hq9+`.

Ввод мы будем осуществлять из файла, вывод на консоль. Для простоты, любые другие символы мы будем просто игнорировать.

```
f = open(input('Enter file name: '))
s = f.read()
f.close()
```

И, собственно, сам интерпретатор:

```
template = '''{} bottles of beer on the wall.
Take one down and pass it around, {} bottles of beer on
the wall.'''
count = 0 # Никому не нужный счётчик

for i in s.upper(): # Игнорируем регистр
    if i == 'H':
        print('Hello, world!') # Выводим 'Hello, world!'
```

```

elif i == 'Q':
    print(s) # Выводим саму программу
elif i == '9':
    for i in range(99, 1, -1):
        print(template.format(i, i-1)) # Выводим
текст песни
        print('1 bottle of beer on the wall.\nTake one
down and pass it around, no more bottles of beer on the
wall.')
        print('No more bottles of beer on the wall.\nGo
to the store and buy some more, 99 bottles of beer on the
wall.')
elif i == '+':
    count += 1

```

Самое "сложное" здесь - вывод текста песни "99 Bottles of Beer". Можно было записать весь текст, но тогда файл получился бы довольно объёмный. Поэтому вместо этого я использую форматирование строк, и вывожу песню с помощью цикла.

У меня на этом всё, и в конце - онлайн-интерпретатор [hq9+](#).

**Введите программу на [hq9+](#)**



## Задача про словарь

Сегодня я разберу одну из олимпиадных задач, довольно простую.

Суть задачи в том, чтобы из англо-латинского словаря сделать латино-английский.

Примеры тестов

Входные данные

```
3
apple - malum, pomum, popula
fruit - baca, bacca, popum
punishment - malum, multa
Выходные данные
```

```
7
baca - fruit
bacca - fruit
malum - apple, punishment
multa - punishment
pomum - apple
popula - apple
popum - fruit
```

Входные данные черпаются из input.txt, вывод сбрасывается в output.txt. Довольно стандартное требование, про работу с файлами в python я недавно писал. Входные данные лексикографически отсортированы, и выходные данные тоже требуются отсортированными.

Вначале я покажу своё решение, потом начну объяснять. Итак, моё решение:

```
f = open('input.txt')
N = f.readline()
d = {}
for line in f:
    words = line.strip().split(' - ')
    en = words[0]
    lat = words[1].split(', ')
    for key in lat:
        if key in d:
            d[key].append(en)
        else:
```

```

        d[key] = [en]
f.close()

for key in d:
    d[key].sort()

g = open('output.txt', 'w')
g.write(str(len(d)) + '\n')
for lat in sorted(d):
    g.write(lat + ' - ' + ', '.join(d[lat]) + '\n')

g.close()

```

Открываем файл, читаем число N (оно нам не понадобится); создаём пустой словарь.

Отделяем слово от его переводов; из строки с переводами получаем список латинских слов.

Далее смотрим, есть ли латинское слово в нашем словаре. Если есть - дописываем ещё один английский перевод. Если нет - пишем его. После махинаций закрываем файл. Далее нужно все списки переводов отсортировать (требования задачи).

Открываем файл на запись. Первой строкой записываем число получившихся слов. Далее в отсортированном порядке записываем латинские слова, затем их переводы. Закрываем файл.

Задача решена, все тесты пройдены. Несомненно, возможны и другие решения, но это мне показалось наиболее красивым.

## Пишем блэkdжек

Для реализации нам понадобится колода карт, из которой каждый раз мы будем вынимать по карте и прибавлять к результату.

Далее, сами "карты": шестерка, семерка, восьмерка, девятка, десятка, валет (достоинством 2), дама (3), король (4), и туз (11).

```
koloda = [6, 7, 8, 9, 10, 2, 3, 4, 11] * 4
```

Случайным образом перемешаем карты, используя функцию `shuffle` из модуля `random`.

```
import random
random.shuffle(koloda)
```

И, собственно, начинаем играть:

```
print('Поиграем в очко?')
count = 0

while True:
    choice = input('Будете брать карту? y/n\n')
    if choice == 'y':
        current = koloda.pop()
        print('Вам попалась карта достоинством %d'
%current)
        count += current
        if count > 21:
            print('Извините, но вы проиграли')
            break
        elif count == 21:
            print('Поздравляю, вы набрали 21!')
            break
        else:
            print('У вас %d очков.' %count)
    elif choice == 'n':
        print('У вас %d очков и вы закончили игру.'
%count)
        break

print('До новых встреч!')
```

Изначально у пользователя 0 очков. Мы его спрашиваем, будет ли он брать карту, на что он должен ответить у или n. Если пользователь ответил n, то мы говорим ему, сколько очков он набрал, и завершаем программу. Если он изъявил желание взять карту (ух, какой нехороший пользователь :)), то мы снимаем ему карту из списка (с помощью [метода pop](#)). Мы снимаем последнюю карту, хотя вообще без разницы, какую снимать, ведь они перемешаны.

Прибавляем к числу очков достоинство снятой карты, а дальше смотрим, сколько всего очков у пользователя. Если количество очков больше 21, то

извиняйте, пользователь проиграл. Если число очков равно 21, то пользователь выиграл. Если меньше - еще раз спросим пользователя, будет ли он брать карту.

В конце игры прощаемся с пользователем.

P.S. Кто-нибудь знает, как у этой хреновины 21 набрать? А то уже двадцать раз играю, все время перебор :)

## Интерпретатор brainfuck

Для тех, кто не знает, о чем это я говорю, поясняю: язык brainfuck для хранения данных использует ячейки (по-хорошему бесконечное число ячеек) и состоит всего из восьми команд, поэтому выучить его будет легко.

Вот эти команды:

| Команда Brainfuck | Описание команды                                                                                                                    |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| >                 | перейти к следующей ячейке                                                                                                          |
| <                 | перейти к предыдущей ячейке                                                                                                         |
| +                 | увеличить значение в текущей ячейке на 1                                                                                            |
| -                 | уменьшить значение в текущей ячейке на 1                                                                                            |
| .                 | напечатать значение из текущей ячейки                                                                                               |
| ,                 | ввести извне значение и сохранить в текущей ячейке                                                                                  |
| [                 | если значение текущей ячейки 0, перейти вперед по тексту программы на ячейку, следующую за соответствующей ] (с учётом вложенности) |
| ]                 | если значение текущей ячейки не 0, перейти назад по тексту программы на символ [ (с учётом вложенности)                             |

Итак, вернемся к интерпретатору. Программный код будем считывать со стандартного ввода (если кто захочет, может переделать на считывание из файла).

Итак, будем считать, что прочитали (с помощью встроенной функции `input()`), затем обработаем строки, удалив все нежелательные символы.

```
def parse(code):  
    new = ''  
    for c in code:
```

```

        if c in '><+-. , []':
            new += c
    return new

```

Или проще:

```

def parse(code):
    return ''.join(c for c in code if c in '><+-. , []')

```

Далее сопоставим начало и конец каждого цикла (кода, заключенного в []).

```

def block(code):
    opened = []
    blocks = {}
    for i in range(len(code)):
        if code[i] == '[':
            opened.append(i)
        elif code[i] == ']':
            blocks[i] = opened[-1]
            blocks[opened.pop()] = i
    return blocks

```

Функция возвращает словарь {начало:конец и конец:начало} для быстрой навигации по программному коду.

Ну и, собственно, сам интерпретатор:

```

def run(code):
    code = parse(code)
    blocks = block(code)
    x = i = 0
    bf = {0: 0}
    while i < len(code):
        sym = code[i]
        if sym == '>':
            x += 1
            bf.setdefault(x, 0)
        elif sym == '<':
            x -= 1
        elif sym == '+':
            bf[x] += 1
        elif sym == '-':
            bf[x] -= 1
        i += 1

```



```

elif sym == '.':
    print(chr(bf[x]), end='')
elif sym == ',':
    bf[x] = int(input('Input: '))
elif sym == '[':
    if not bf[x]: i = blocks[i]
elif sym == ']':
    if bf[x]: i = blocks[i]
i += 1

```

Как это работает? Сначала обрабатывается код, составляется список циклов. Далее ячейки, которые я реализовал в качестве словаря. Далее, разбор brainfuck-программы.

Если символ '>', то увеличиваем x (номер ячейки) на единицу, и, если ячейки с таким номером в словаре нет, инициализируем нулем (методом setdefault).

Если '<', то уменьшаем номер ячейки на 1. Так как вообще отрицательные ячейки не разрешены, то и ячейка всегда найдется (если хотите, можете добавить поддержку и отрицательных номеров ячеек). Если символ '[', то проверяем текущую ячейку, и, если она 0, переходим в конец цикла.

Полный код интерпретатора brainfuck:

```

def block(code):
    opened = []
    blocks = {}
    for i in range(len(code)):
        if code[i] == '[':
            opened.append(i)
        elif code[i] == ']':
            blocks[i] = opened[-1]
            blocks[opened.pop()] = i
    return blocks

def parse(code):
    return ''.join(c for c in code if c in '><+-.,[]')

def run(code):
    code = parse(code)
    x = i = 0
    bf = {0: 0}
    blocks = block(code)
    l = len(code)

```

```
code = input()
run(code)
```

+++++ [ >+++++>+++++>+>+<<<- ] >+ . >+ . +++++  
+ . . + + . >+ . <<+ + + + + + + + + + . > . + + . - - - - . >+ . > .