

Введение в машинное обучение

Введение в язык программирования



Питон (Python)

Александр Дьяконов

Московский государственный университет имени М.В. Ломоносова

Python

«Питон» или «пайтон»

в честь комедийных серий BBC «Летающий цирк Монти-Пайтона»

Создатель голландец Гвидо ван Россум (Guido van Rossum), 1991 г.

Особенности

- интерпретируемый
- объектно-ориентированный
- высокоуровневый язык
- встроенные высокоуровневые структуры данных
- динамическая типизация
- синтаксис прост в изучении
- поддержка модулей и пакетов (МНОГО бесплатных библиотек)
- универсальный
- интеграция с другими языками (C, C++, Java)

Важно для нас

- **Прост в освоении**

рекомендуют и используют как первый язык программирования

- **Короткие конструкции**

программы пишутся быстро и легко читаются

- **Есть хорошие библиотеки для ML**

numpy, scipy, scikit-learn, pandas, matplotlib + обёртки для библиотек DL

Дальше

Быстрый и более-менее полный обзор

Задача: не сильно напрягаясь понять, что есть

ДЗ Проработать все синтаксически конструкции, написать мини-программы (по выбору), пройти тест (на сайте)

Ветки (несовместимые) языка:

- Python 2.x
- Python 3.x

Здесь – 3.x (с указанием особенностей в 2.x, слушайте лектора!)

Поддерживаемые парадигмы:

- императивное (процедурный, структурный, модульный подходы) программирование
- объектно-ориентированное программирование
- функциональное программирование

PEP8 <https://www.python.org/dev/peps/pep-0008/>

стилистические рекомендации по оформлению кода

- отступ – 4 пробела
- длина строки < 80 символов
- переменные: `var_recommended`
- константы: `CONST_RECOMMENDED`
- ...

```
import this
```

The Zen of Python, by Tim Peters

- **Beautiful is better than ugly.**
- **Explicit is better than implicit.**
- **Simple is better than complex.**
- **Complex is better than complicated.**
- **Flat is better than nested.**
- **Sparse is better than dense.**
- **Readability counts.**
- **Special cases aren't special enough to break the rules.**
- **Although practicality beats purity.**
- **Errors should never pass silently.**
- **Unless explicitly silenced.**
- **In the face of ambiguity, refuse the temptation to guess.**
- **There should be one-- and preferably only one --obvious way to do it.**
- **Although that way may not be obvious at first unless you're Dutch.**
- **Now is better than never.**
- **Although never is often better than *right* now.**
- **If the implementation is hard to explain, it's a bad idea.**
- **If the implementation is easy to explain, it may be a good idea.**
- **Namespaces are one honking great idea -- let's do more of those!**

Основы Python: условный оператор, функция

```
# функция
def sgn(x):
    """
    функция 'знак числа'
    +1 - для положительного аргумента
    -1 - для отрицательного аргумента
    0 - для нуля
    Пример: sgn(-2.1) = -1
    """
    # if - условный оператор
    if x > 0:
        a = +1
    elif x < 0:
        a = -1
    else:
        a = 0
    return a
```

```
sgn(2.1), sgn(0), sgn(-2)
(1, 0, -1)
```

```
"nonzero" if x != 0 else "zero" # другой вариант условного оператора
```

**многострочных комментариев
нет – часто используются строки**

**но важны отступы (4 пробела)
нет операторных скобок и end**

обратите внимание на двоеточие

**после return скобки не
обязательны**

**для помощи – help (sgn)
выведется оранжевый текст**

Основы Python: цикл for, вывод

```
# for - цикл
for i in range(1, 4):
    s = ""
    for j in range(1, 4):
        s += ("%i " % (i * j))
        #print ("%i" % (i*j),)
    print (s)
```

```
1 2 3
2 4 6
3 6 9
```

```
# можно много по чему итерироваться
for i in [10, 20]:
    for j in 'ab':
        print (i, j)
```

```
(10, 'a')
(10, 'b')
(20, 'a')
(20, 'b')
```

range – это итератор (см. дальше)

после «:» должно быть 4 пробела

«+» – конкатенация строк (см. дальше)

нет явного счётчика (см. дальше)

как и ожидается, есть
continue
break

Интересно: есть и такие сокращения операций (работают с числами)

«+=»

«-=»

«*=»

«/=»

Основы Python: цикл while, ввод

while – цикл

```
s = input("Введите строку:")
```

```
while s: # s != "":  
    print (s)  
    s = s[1:-1]
```

```
Введите строку:12345  
12345  
234  
3
```

`input` – **ввод именно строки**
(в Python3 !)

`[1:-1]` – «слайсинг»:
без первой и последней букв
(см. дальше)

Нет цикла с постусловием!

Пример решения задачи на Python

```
import math
```

```
def primes(N):  
    """Возвращает все простые от 2 до N"""  
    sieve = set(range(2, N))  
    for i in range(2, round(math.sqrt(N))):  
        if i in sieve:  
            sieve -= set(range(2 * i, N, i))  
    return sieve
```

```
primes(20)
```

```
{2, 3, 5, 7, 11, 13, 17, 19}
```

Вывести простые числа

**Здесь задействован тип
«множество» (см. дальше)**

Можно переносить строки с помощью «\», иногда просто переносить

```
x = 1 + 2 + 3 + 4 + \  
5 + 6 + 7 + 8
```

```
x = (1 + 2 + 3 + 4 +  
5 + 6 + 7 + 8)
```

Где один пробел – можно много

Запуск Python-кода

- 1) интерпретатор
- 2) python test.py
- 3) Jupyter notebook

Filter output as discrete convolution

The discrete convolution of infinite sequences x_n and h_n is defined as

$$y_n = \sum_{k \in \mathbb{Z}} x_k^* h_{n-k}$$

where the asterisk superscript denotes complex conjugation. If we have a finite filter length of M ($h_n = 0, \forall n \notin \{0, 1, \dots, M-1\}$), then the filter output reduces to

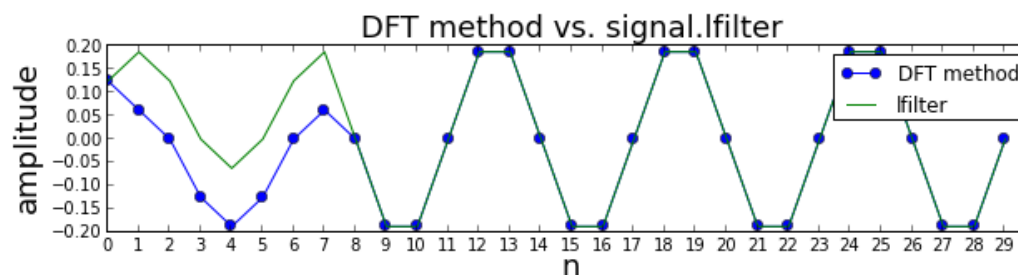
$$y_n = \sum_{k=0}^{M-1} h_k x_{n-k}$$

Note this is closely related to but not the same as the circular convolution we have already discussed because there is no wrap-around. However, because it is very efficient to compute this using a DFT, we need to relate these two versions of convolution.

```
In [4]: h=ones(ma_length)/ma_length # filter sequence
yc=fft.ifft(fft.fft(h,len(x)+len(h)-1)*np.conj(fft.fft(x,len(x)+len(h)-1))).real

fig,ax=subplots()
fig.set_size_inches((10,2))
ax.plot(n,yc[ma_length-1:], 'o-', label='DFT method')
ax.plot(n,y, label='lfilter')
ax.set_title('DFT method vs. signal.lfilter', fontsize=18)
ax.set_xlabel('n', fontsize=18)
ax.set_ylabel('amplitude', fontsize=18)
ax.legend(loc=0)
ax.set_xticks(n);

# fig.savefig('figure_00@.png', bbox_inches='tight', dpi=300)
```



Jupyter notebook

```
In [31]: [1, 2, 3]
```

```
Out[31]: [1, 2, 3]
```

```
In [32]: _[0] # предыдущая ячейка
```

```
Out[32]: 1
```

```
In [33]: sum(__) # пред-предыдущая ячейка
```

```
Out[33]: 6
```

```
In [35]: Out[31] # конкретная ячейка
```

```
Out[35]: [1, 2, 3]
```

```
In [38]: pwd # unix-dos-команды
```

```
Out[38]: u'C:\\tmp\\notebooks'
```

возможность программировать (и проводить эксперименты) в браузере

Булева логика

```
x, y = True, False # так можно
```

```
print (x and y)
print (x or y)
print (not y)
print (x and y)
print ((1 == 2) | (2 == 2))
print ((1 < 2) | (2 != 2))
```

```
False
True
True
False
True
False
```

```
# приведение типов
```

```
print (bool('True') == bool(1))
```

```
True
```

```
print (1 < 2 < 3 < 4)
```

```
print (1 < 3 < 3 < 4)
```

```
True
```

```
False
```

**Можно использовать такие
«сложные условия»**

```
x = 4
```

```
if 3 < x < 5: # можно без скобок
```

```
    print ('четыре')
```

```
четыре
```

```
if 3 < x and x < 5:
```

```
    print ('четыре')
```

```
четыре
```

```
# проверка списка на пустоту
```

```
if not lst:
```

```
    ...
```

объект False, если он пуст

Числа

```
print (2 ** 1000) # Python3
print (2L ** 1000) # Python2
```

```
10715086071862673209484250490600018105614048117055336074437503
88370351051124936122493198378815695858127594672917553146825187
14528569231404359845775746985748039345677748242309854210746050
62371141877954182153046474983581941267398767559165543946077062
914571196477686542167660429831652624386837205668069376
```

```
print (type(1))
print (type(1.0))
print (type(int(1.0))) # преобразование
                        типов
print (type(-1.2+3.7j + 5))
print (type(None))
```

```
<class 'int'>
<class 'float'>
<class 'int'>
<class 'complex'>
NoneType
```

- int – **целые**
- long – **произвольная точность (нет в Python 3)**
- float
- complex

**В Python2 операция деление
целых целочисленное**

**В Python 3 результат уже
более предсказуемый**

```
print(10 // 3) # Python 3 - деление нацело
print(10 % 3) # остаток
from __future__ import division
print(10 / 3) # в Python 2 - нацело
```

```
3
1
3.3333333333333335
```

Строка

в Python3 – неизменяемы массив символов Юникода

**«рекурсивная структура данных» –
каждый символ объект типа `str` длины 1
нет понятия символ (это одноэлементная строка)**

задание одной и той же строки

```
s1 = "string"
```

```
s2 = 'string'
```

```
s3 = """string"""
```

```
s4 = 'st' 'rin' 'g' # будет склейка (аналогично +)
```

```
s5 = 'st' + 'rin' + 'g'
```

```
s = u'\u043f\u0440\u0438\u0432\u0435\u0442' # можно убрать u
```

```
print(s)
```

привет

Строки

```
s5 = """раз  
два\nтри"""  
print(s5)
```

раз
два
три

```
s4 = '\1\2'  
s5 = r'\1\2'  
print(s4, '--', s5)  
-- \1\2
```

**Потом в списках типичная ошибка –
пропуск запятой**

```
lst = ['Петя',  
       'Маша',  
       'Вася',  
       ]  
print(lst)
```

```
['Петя', 'МашаВася']
```

Операции над строками

```
print('A' + 'B') # конкатенация
```

AB

```
print('A' * 3) # повтор
```

AAA

```
# например
```

```
s = 'one'  
s += 'two'  
s *= 3  
print(s)
```

onetwoonetwoonetwo

Форматирование строк – четыре способа

```
print ('Привет, %s' % name)
```

```
print ('Привет, {}'.format(name))
```

форматированные строковые литералы (Formatted String Literals)

```
print (f'Привет, {name}')
```

шаблонные строки

```
from string import Template
```

```
print (Template('Привет, $name').substitute(name=name))
```

Пример второго форматирования

разные форматы чисел

```
print ("int - {0:d}, hex - {0:x}, bin - {0:b}".format(12))
```

```
int - 12, hex - c, bin - 1100
```

можно именовать аргументы и использовать индексы

```
print ("x={0[0]}, y={0[1]}, z={z}".format([1,2], 3, z=4))
```

```
x=1, y=2, z=4
```


Строки: форматирование **todo**

форматирование

```
print('%010.3g' % (1 / 3))  
print('%+10.4g' % (1 / 3))  
print('%-10.5g' % (1 / 3))  
print('%-10.5e' % (1 / 3))
```

```
000000.333  
    +0.3333  
0.33333  
3.33333e-01
```

вывод в блоке одной длины

```
print("{:~^10}".format("a"))  
print("{:~^10}".format("aaa"))  
print("{:~^10}".format("aaaaa"))  
~~~~a~~~~~  
~~~aaa~~~~  
~~aaaaa~~~
```

другой способ форматирования (более гибкий)

```
print("a={:0.4e}, b={:+2.3f}, c={}".format(1 / 3, 1 / 7, 1 / 11))  
print("a={:>5s}, b={:%}, c={:06.2f}".format('one' , 1 / 7, 1 /  
11))
```

```
a=3.3333e-01, b=+0.143, c=0.0909090909090909091  
a=  one, b=14.285714%, c=000.09
```

Строки: операции

```
s = 'one,one'
```

```
s.count('on') # подсчёт вхождения подстроки
```

```
2
```

```
s.find('on') # поиск подстроки (есть ещё index - с исключениями)
```

```
0
```

```
s.rfind('on') # поиск последней подстроки (последнее вхождение)
```

```
4
```

```
s.isalpha() # только буквы
```

```
False
```

```
s.islower() # только строчные / isupper / istitle / isspace
```

```
True
```

```
s.isdigit() # число
```

```
False
```

```
s.isalnum() # только буквы и цифры
```

```
False
```

```
s.replace('on','off') # замена подстрок
```

```
offe,offe
```

```
s.translate({ord('o'): 'a', ord('n'): 'b'}) #множеств.замена симв.
```

```
abe,abe
```

Строки: операции

```
' 12 '.strip() # del 1-ых и lst-их пробелов,ещё: lstrip, rstrip
12
s.upper() # в верхний регистр + lower
ONE,ONE
s.capitalize() # первую букву в верхний регистр, ост. - в нижний
One,One
'file.txt'.endswith('.exe') # startswith
False
s.rpartition(',') # расщепление по разделителю
('one', ',', 'one')

# вхождение подстроки (проверка, а не поиск)
s = "one,two,three"

'on' in s
True

'ab' not in s
True
```

Строки: операции

```
s = 'one,two,three'
s2 = s.split(',') # расщепление в список
print (s, s2)
one,two,three ['one', 'two', 'three']
```

```
print (";".join(s2)) # объединение через разделитель
one;two;three
```

```
# выравнивание в блоке фиксированной длины
s = 'my string'
print (s.ljust(13, ' '))
print (s.center(13, '-'))
print (s.rjust(13)) # пробел можно не указывать
```

```
my string
--my string--
    my string
```

Строки: индексация (дальше: как в списках)

```
s = 'string'
print (s[0], s[:3], s[-2:])
s str ng
```

Байты

```
x = b"\00\01\10"
```

```
x
```

```
b'\x00\x01\x08'
```

```
s = "строка"
```

```
s.encode("utf-8")
```

```
b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
```

```
x = b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
```

```
x.decode("utf-8")
```

```
'строка'
```

Кодировки

Стандарт Unicode:

для каждого символа есть

1) идентификатор символа – его кодовая позиция

U+0041 для **A**

2) его байтовое представление – зависит от кодировки

\x41 в **UTF8** и **\x41\x00** в **UTF-16LE**

В Win часто файлы создаются в кодировке cp1252

В начале файла – # coding: cp1252

В Unix – UTF8 (она и в Python 3 по умолчанию)

BOM = byte-order mark (маркер порядка байтов = порядок принятый в процессоре Intel)

Всегда явно указывать кодировку!
(при открытии-закрытии файлов)

Список (list)

изменяемый динамический массив

**Простейший и удобный контейнер – для хранения перечня объектов
(а в питоне всё – объект)**

**контейнер для разнородных
элементов**

```
s = [1, 'string', [1,2,3], True]
```

```
s[1]  
'string'
```

```
s[2][0]  
1
```

**списки могут быть
вложенные**

```
a = [1, 2, 3]  
b = [4, 5, 6]  
lst = [1, [a, b]]
```

```
lst[1][0][2]  
3
```

Список (list): задание

Для начала, простой вариант – перечень чисел

```
[1, 2, 3] # список
```

```
[x for x in range(3)] # потом узнаем о ...  
[0, 1, 2]
```

```
# преобразование типов  
list(range(3)) # из генератора
```

```
[0, 1, 2]
```

```
list('строка') # из строки  
['с', 'т', 'р', 'о', 'к', 'а']
```

```
list({1, 3, 1, 2}) # из множества  
[1, 2, 3]
```

```
list((1, 1, 2)) # из кортежа  
# меньше скобок нельзя  
[1, 1, 2]
```

**Задаётся с помощью
квадратных скобок**

[] – пустой список

**Можно преобразовывать из
других объектов**

Индексация, нарезка (slicing)

для списков, строк и т.д.

```
s = [0, 1, 2, 3, 4, 5]
```

```
s[2] # третий! элемент  
2
```

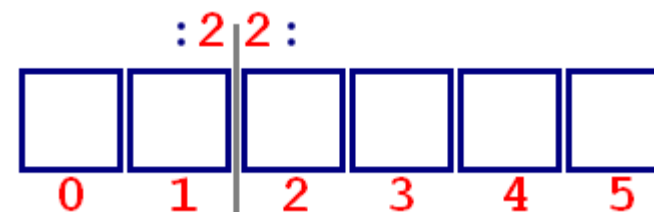
```
s[:2] # первые два элемента  
[0, 1]
```

```
s[2:] # после второго  
[2, 3, 4, 5]
```

```
s[:-2] # без двух элементов  
[0, 1, 2, 3]
```

```
s[-2:] # последние два  
[4, 5]
```

```
s[0:4:2] # от : до : шаг  
[0, 2]
```



Нумерация с нуля

Индексация слева и справа

Схема (от : до : шаг)

```
s[::3] # все через шаг  
[0, 3]
```

```
s[::-1] # в обратном порядке  
[5, 4, 3, 2, 1, 0]
```

Индексация: нетривиальности

```
s = [0, 1, 2, 3, 4, 5]
```

```
s[0:0] = [-2, -1] # добавление слева  
[-2, -1, 0, 1, 2, 3, 4, 5]
```

```
s[-1:0] = [0.1, 0.2] # добавление справа  
[-2, -1, 0, 1, 2, 3, 4, 0.1, 0.2, 5]
```

«Именованные срезы»

```
person = ["Иван", "Иванов", 22, "мая", 2001]  
NAME, BIRTHDAY = slice(2), slice(2, None)  
print(person[NAME], person[BIRTHDAY]) # вместо :2 и 2:
```

```
['Иван', 'Иванов'] [22, 'мая', 2001]
```

Очистка списка

```
# можно так  
lst = [1, 2, 3]  
del lst[:]  
print (lst)
```

```
[]
```

```
# Python 3  
lst = [1, 2, 3]  
lst.clear()  
print (lst)
```

```
[]
```

Список (list): операции

```
s = [1, 2, 3] # это список
```

```
len(s) # длина списка!
```

```
3
```

```
max(s) # максимальный элемент
```

```
3
```

```
2 in s # принадлежность списку
```

```
True
```

```
s + s # конкатенация
```

```
# создаётся новый список!
```

```
[1, 2, 3, 1, 2, 3]
```

```
s * 2 # "удвоение"
```

```
[1, 2, 3, 1, 2, 3]
```

```
del s[1] # удаление элемента
```

```
s
```

```
[2, 3]
```

```
s[0] = 100 # присваивание значения
```

```
s
```

```
[100, 3]
```

```
# сравнение (лексикографический  
порядок)
```

```
print([1, 2] < [1, 3])
```

```
print([1, 2] < [1, 2, 1])
```

```
print([2] < [1, 3])
```

```
True
```

```
True
```

```
False
```

```
l = [1, 2, 3, 4, 5]
```

```
l[2:5] = 100 # не работает
```

```
l[2:5] = [100] # Работает!
```

**Есть естественные функции:
максимум, минимум, сумма**

Список (list): операции

<code>s = [4] * 3</code>	# [4, 4, 4]	# вырезает элемент (по индексу)
# удаление первого вхождения элемента		<code>pop()</code> - последний
<code>s.remove(4)</code>		<code>s.pop(1)</code>
	[4, 4]	3, [2, 3, 4, 4]
# добавление элемента		# вставка элемента
<code>s.append(2)</code>		<code>s.insert(0, 1)</code>
	[4, 4, 2]	[1, 2, 3, 4, 4]
# добавление последовательности		# вставка элемента
<code>s.extend([3, 3])</code>		<code>s.insert(-1, 5)</code>
	[4, 4, 2, 3, 3]	[1, 2, 3, 4, 5, 4]
# сколько элементов		# вставка элементов
<code>s.count(4)</code>	2	<code>s[-4:] = [0]*4</code>
		[1, 2, 0, 0, 0, 0]
# индекс элемента (первое вхождение), если не входит - исключение <code>ValueError</code>		# удаление элементов
<code>s.index(2)</code>	2	<code>del s[-3:]</code>
		[1, 2, 0]
# инвертирование		
<code>s.reverse()</code> # или <code>s = s[::-1]</code>		
	[3, 3, 2, 4, 4]	
<code>s.sort()</code> # сортировка		
	[2, 3, 3, 4, 4]	

**Здесь не всегда показаны
возвращаемые функциями значения, а
просто текущий список!**

Тонкости питона: копирование

```
x = [[0]]*2 # делаем список
```

```
[[0], [0]]
```

```
x[0][0] = 1 # меняем один элемент  
# ... а поменялись оба
```

```
[[1], [1]]
```

```
id(x[0]), id(x[1])
```

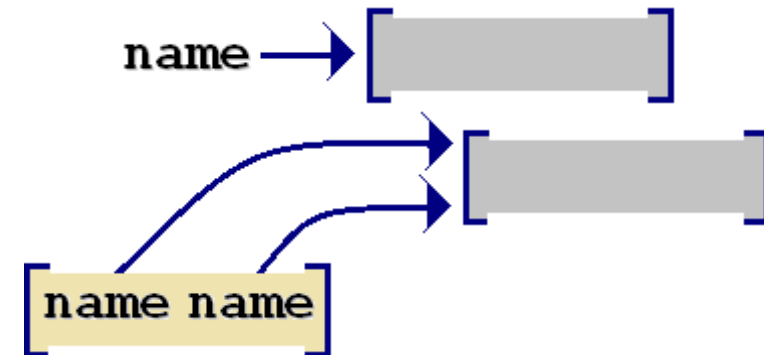
```
(72882632, 72882632)
```

```
x = x + x  
x[0][0] = 2 # такой же эффект
```

```
[[2], [2], [2], [2]]
```

При операции * не происходит копирования списка!

id – идентификатор объекта (уникален)



```
[[0] for x in range(2)] # можно так!
```

```
# но так снова плохо...  
b = [0]  
a = [b for x in range(2)]
```

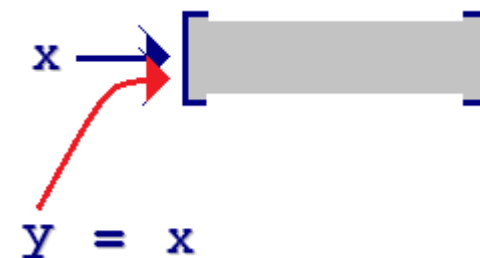
Тонкости питона: копирование

Как быть – копирование (поверхностное)

```
from copy import copy  
x = [copy([0]) for i in range(2)]  
x[0][0] = 1
```

```
[[1], [0]]
```

При присваивании просто
создаётся ссылка на объект



Способы копирования списка:

```
new_list = old_list[:]
```

```
new_list = list(old_list) # аналогично dict, set
```

```
import copy  
new_list = copy.copy(old_list)
```

Тонкости питона: копирование

**Копирование не всегда помогает,
выручает глубокое копирование (частично ;)**

```
from copy import copy
```

```
a = 1
b = [a, a]
c = [b, b]
c2 = copy(c)
c2[0][0] = 0
```

```
print (c)
print (c2)
```

```
[[0, 1], [0, 1]]
[[0, 1], [0, 1]]
```

```
from copy import deepcopy
```

```
a = 1
b = [a, a]
c = [b, b]
c2 = deepcopy(c)
c2[0][0] = 0
```

```
print (c)
print (c2)
```

```
[[1, 1], [1, 1]]
[[0, 1], [0, 1]]
```


Кортеж (tuple): присваивание через кортеж

Кортеж – «неизменяемый список», вместо [] будут () или ничего

```
a, b, c = 1, 2, 3 # a это кортеж!  
print (a, b, c)  
(1, 2, 3)
```

При инициализации переменных часто бывают такие конструкции

```
(x, y), (z, t) = [1, 2], [3, 4]  
print (x, y, z, t)  
(1, 2, 3, 4)
```

```
# сработает последнее присваивание  
x, (x, x) = 1, (2, 3)  
print (x)  
3
```

Переменные могут обмениваться значениями без использования ещё одной переменной

```
a, b = 1, 2  
a, b = b, a # присваивание кортежей
```

Кстати, о присваивании... допустимы такие конструкции

```
i = j = k = 1
```

Кортеж (tuple): задание

разные способы задания кортежа

```
a = 1, 2, 3
```

```
a = (1, 2, 3)
```

```
a = tuple((1, 2, 3))
```

пустой кортеж

```
a = () # раньше (,)
```

```
a
```

```
()
```

x = (2,) # одноэлементный кортеж

```
print (x)
```

[y] = x # элемент этого кортежа

```
y
```

```
(2,)
```

```
2
```

Неизменяемый тип

Контейнер

**Может содержать объекты
разных типов**

итерация по кортежу и его вывод

```
for x in a:
```

```
    print (x)
```

```
1
```

```
2
```

```
3
```

Кортеж (tuple): операции всё аналогично спискам

```
x = (1, 2)
y = (3, 4)
x + y # конкатенация
(1, 2, 3, 4)
```

Кортеж неизменяем, но...

```
a = (1, 2, [1, 2])
a[-1].append(3)
a
```

```
(1, 2, [1, 2, 3])
```

**это три неизменяемые ссылки,
третья – на объект, который имеет метод `append`**

Словарь (dict)

ассоциативный массив (associative array), ассоциативными хеш-таблицами (hashmap), поисковая таблица (lookup table)

**контейнер для хранения данных вида (key, value), порядок не важен
(в Python ≥ 3.6 он всё-таки автоматически выдерживается)**

**ключ – любой хешируемый тип
(есть hash-значение, которое не меняется)**

Может содержать разные объекты (разных типов)

Словарь (dict)

```
dct = {'a': 1, 'b': 2} # словарь  
{'b': 2, 'a': 1}
```

```
dct = dict(a=1, b=2) # другой способ  
{'b': 2, 'a': 1}
```

```
# добавление к словарю  
dct = dict(dct, a=3, d=2)  
{'d': 2, 'a': 3, 'b': 2}
```

```
# преобразование из списка  
dct = dict([('a', 1), ('b', 2)])  
{'b': 2, 'a': 1}
```

```
dct['a'] # обращение по ключу (если  
нет - исключение KeyError)  
1
```

```
# обращение по ключу со значением по  
умолчанию (когда ключ не найден)  
dct.get('c', 0)  
0
```

```
# проверка не-вхождения  
'a' not in dct  
False
```

```
del dct['a'] # удаление по ключу  
dct  
{'b': 2}
```

```
dct.keys() # ключи  
dict_keys(['b', 'a'])
```

```
dct.values() # значения  
dict_values([2, 1])
```

```
dct.items() # пары (ключ, значение)  
dict_items([('b', 2), ('a', 1)])
```

**обратите внимание на
использование dict**

Словарь (dict)

```
d = dict(a=True, b="02", c=[1, 2, 3])
{'c': [1, 2, 3], 'b': '02', 'a': True}
```

попытка добавить значение, если нет

```
d.setdefault('c', 100.0)
```

```
d.setdefault('d', 100.0)
```

```
{'c': [1, 2, 3], 'd': 100.0, 'b':
'02', 'a': True}
```

добавить ещё значений

```
d.update(e=1, f=2)
```

```
{'c': [1, 2, 3], 'd': 100.0, 'b':
'02', 'f': 2, 'a': True, 'e': 1}
```

Стандартное присваивание значений

элементам такое:

```
d['a'] = 100.0
```

**setdefault – не трогает уже
существующие значения**

```
d.update([('e', 3), ('f', 4)])
# добавить ещё значений – старые
значения заменятся на новые
print('c = ', d.pop('c')) #
возвращаем значение и удаляем его
из словаря
```

```
c = [1, 2, 3]
```

```
print(d)
```

```
{'d': 100.0, 'b': '02', 'f': 4,
'a': True, 'e': 3}
```

```
class MyDict(dict):
```

```
    # значение по умолчанию
```

```
    def __missing__(self, key):
```

```
        self[key] = rv = []
```

```
        return rv
```

```
x = MyDict()
```

```
x['b'] = 1
```

```
x['a']
```

```
print(x)
```

```
{'b': 1, 'a': []}
```

Итерации по словарю

```
dct = {'a': 1, 'b': 2}
dct[0] = 5
```

```
# цикл по парам
```

```
for key, val in dct.items():
    print (key, val)
```

```
b 2
0 5
a 1
```

```
# цикл по ключам словаря
```

```
for key in dct: # dct.keys()
    print (key, dct[key])
```

```
b 2
0 5
a 1
```

```
# цикл по значениям словаря
```

```
for val in dct.values():
    print (val)
```

```
2
5
1
```

**Ключ не обязательно строка,
м.б. число
(главное, что должен
хэшироваться)**

```
print ('длина словаря = %i' %
len(dct)) # количество пар в словаре
```

длина словаря = 3

```
d.clear() # удалить все значения
```

Ещё способы задания словаря

словарь со значением по умолчанию

```
dict.fromkeys("abc", True)
```

```
{'a': True, 'b': True, 'c': True}
```

преобразование типов

```
a = ['a', 'b', 'c']
```

```
b = [1, 2, 3]
```

```
dict(zip(a, b)) # см. потом про zip
```

```
{'a': 1, 'b': 2, 'c': 3}
```

Задача: объединить два словаря, не портя их

```
dct = {'a': 1, 'b': 2}
```

```
dct2 = {'b': 3, 'c': 4}
```

```
union = {**dct, **dct2} # Python3-  
способ
```

```
print(union, dct, dct2)
```

```
union = dct.copy() # Python2-способ
```

```
union.update(dct2)
```

```
print(union, dct, dct2)
```

```
# а можно так...
```

```
union = dict(dct, **dct2)
```

```
print(union, dct, dct2)
```

```
{'a': 1, 'b': 3, 'c': 4} {'a': 1, 'b': 2} {'c': 4, 'b': 3}
```


Одно из применений словарей – имитация switch

```
def first():  
    print ('one')  
def second():  
    print ('two')  
def third():  
    print ('three')
```

```
x = 2
```

```
# плохой способ
```

```
if (x == 1):  
    first()  
elif (x == 2):  
    second()  
elif (x == 3):  
    third()
```

```
# Python-style способ
```

```
dct = {1: first, 2: second, 3: third}  
dct[x]()
```

Множество (set)

```
s = {'key1', 'key1', 'key2'}
print (s)
{'key1', 'key2'}

'key2' in s
True

# тут "=", а дальше - нет
s = s.union({1, 2})
{1, 2, 'key1', 'key2'}

s.difference({1, 3, 4})
{2, 'key1', 'key2'}

s.add(121) # добавить 1 элемент
{1, 2, 'key1', 'key2', 121}

# добавляем несколько элементов
s.update([122, 123, 121])
{1, 2, 'key2', 'key1', 121, 122, 123}
```

```
s.remove('key1')
# если нет - исключение
# есть ещё discard (без исключений)
{1, 2, 'key2', 121, 122, 123}

# преобразование типов
x = [1, 2, 2]
set(x)
{1, 2}
```

**опять: только хэшируемые
объекты могут быть элементами
множества (числа, строки)**

**есть ещё frozenset –
неизменяемое множество**

Множество (set): операции

```
a = {1, 2, 3}
```

```
b = {2, 3, 4}
```

```
# пересечение
```

```
a & b
```

```
a.intersection(b) # 2-й способ
```

```
{2, 3}
```

```
# объединение
```

```
a | b
```

```
a.union(b) # 2-й способ
```

```
{1, 2, 3, 4}
```

```
# разность
```

```
a - b
```

```
a.difference(b) # 2-й способ
```

```
{1}
```

```
# вложения
```

```
a <= b
```

```
False
```

```
a < b
```

```
False
```

```
a > b
```

```
False
```

Аргументов может быть много

```
x, y, z = {1, 2}, {3}, {1, 3, 4}
```

```
set.union(x, y, z)
```

```
{1, 2, 3, 4}
```

```
set.difference(x, y, z) # x - y - z
```

```
{2}
```

Задача: только ли из уникальных элементов состоит список

```
if len(x) == len(set(x)):  
    print('List is unique!')
```

Файл (file)

```
f1 = open("file1.txt", "r")
f2 = open("file2.txt", "w",
          encoding="cp1251")
for line in f1.readlines():
    f2.write(line)
f2.close()
f1.close()
```

```
# чтобы не забывать закрывать файлы
with open('tmp.txt') as fin:
    for line in fin:
        # ...
```

**Это называется
менеджер контекста
(не надо явно закрывать файл)**

Файлоподобные объекты

```
import urllib
f1 = urllib.urlopen("http://python.onego.ru")
```

Функциональное программирование

**в ФП вычисление – вычисление значений математических функций,
а не последовательность процедур**

императивный стиль

```
target = []  
# для каждого элемента  
# исходного списка  
for item in source_list:  
    # применить функцию G()  
    trans1 = G(item)  
    # применить функцию F()  
    trans2 = F(trans1)  
    # добавить ответ в список  
    target.append(trans2)
```

функциональный стиль

```
# языки ФП часто имеют  
# встроенную функцию compose()  
compose2 = lambda A, B: lambda x: A(B(x))  
target = map(compose2(F, G),  
             source_list)  
  
# list(...) в Python-3
```

«что нужно вычислить, а не как»

Функциональное программирование

- **Есть функции первого класса / высшего порядка**
(принимают другие функции в качестве аргументов или возвращают другие функции, их можно присваивать и хранить)
- **Рекурсия – основная управляющая структура в программе**
(нет цикла – он реализован через рекурсию)
- **Обработка списков** (например, `print(len([1+1, 1/0]))`)
- **Запрещение побочных эффектов у функций**
(чистые функции – зависят только от своих параметров и возвращают только свой результат)
- **Описываем не шаги к цели, а математическую зависимость данные–цель**
(в идеале, программа - одно выражение с сопутствующими определениями)

Питон – язык с элементами функционального стиля!

Функции первого класса

```
# функции первого класса
def create_adder(x):
    def adder(y): # определяем функцию внутри
        return x + y
    return adder # её же возвращаем
```

```
add_10 = create_adder(10)
print (add_10(3))
f = add_10 # та же функция
del add_10 # не удаляет саму функцию
print(f(0))
```

13

10

```
print(f.__name__)
```

adder

Интересно здесь также, что эта ф-ия – лексическое замыкание (lexical closure) – помнит значение из лексического контекста: что надо прибавлять именно 3!

Функции – полноправные объекты

могут быть

- **созданы во время выполнения**
- **присвоены переменной**
- **переданы функции в качестве аргументов**
- **возвращены функцией в качестве результата**

**Всегда что-то возвращают
по умолчанию `return None`**

Функции

операции это тоже функции

```
from operator import add, mul
```

```
print(add(2, mul(3, 4)))
```

```
14
```

определение функции

```
# delta -необязательный аргумент
```

```
# со значением по умолчанию
```

```
def inc(n, delta=1):
```

```
    return n + delta
```

```
# эта же функция (просто другое  
имя)
```

```
myadd = inc
```

```
print(inc(20))
```

```
print(myadd(30))
```

```
21
```

```
31
```

```
def f():
```

```
    pass
```

```
def g():
```

```
    """
```

```
    помощь
```

```
    """
```

```
def h():
```

```
    # можно объявлять функцию
```

```
    # внутри функции
```

```
    print('h')
```

```
10
```

```
print(f(), g())
```

```
print(g.__doc__)
```

```
help(g)
```

```
(None, None)
```

```
    помощь
```

```
Help on function g in module
```

```
__main__:
```

```
g()
```

```
    помощь
```

Аргументы функций

именованные аргументы

```
def f(x=1, y=2):  
    print ('x=%g, y=%g' % (x, y))
```

```
f(3, 4)  
f(3)  
f(y=10, x=20)  
f(y=0)
```

```
x=3, y=4  
x=3, y=2  
x=20, y=10  
x=1, y=0
```

сколько угодно аргументов – с помощью «упаковки»

```
def max_min(*args):  
    # args - список аргументов  
    # в порядке их указания при  
    # вызове  
    return max(args), min(args)
```

```
print (max_min(1, 2, 3, 4, 5))  
print (max_min(*[4, 0, 3]))  
print (max_min(*(1, 7, 3)))  
print (max_min(*{6, 2, 4}))
```

```
(5, 1)  
(4, 0)  
(7, 1)  
(6, 2)
```

**возвратить можно только одно
значение, но это м.б. кортеж!**

Что такое распаковка (в Python 3)

```
first, *other = range(3)
print(first, other)
0 [1, 2]
```

```
*a, b, c = range(4)
a, b, c
([0, 1], 2, 3)
```

```
for a, *b in [range(3),
range(2)]:
    print(a, b)
0 [1, 2]
0 [1]
```

```
[*range(5), 6]
[0, 1, 2, 3, 4, 6]
```

```
# инициализация контейнера
d = {'a':1, 'b':2}
d = {**d, 'a':3}
d
{'a': 3, 'b': 2}
```

```
def print_vec(x, y, z):
    print('<%s, %s, %s>' % (x, y, z))
```

```
lst = [1, 2, 3]
tpl = (4, 5, 6)
gen = (i*i for i in range(3))
```

```
print_vec(*lst)
print_vec(*tpl)
print_vec(*gen)
```

```
dct = {'y': 10, 'z': 20, 'x': 30}
print_vec(*dct) # нет гарантии порядка
print_vec(**dct)
```

```
<1, 2, 3>
<4, 5, 6>
<0, 1, 4>
<y, z, x>
<30, 10, 20>
```

Аргументы функций (с неизвестным числом аргументов)

```
# arg1 - фиксированный (здесь - 1 мы  
#          обязательно должны передать)  
# args - произвольные  
# kwargs - любые
```

```
def swiss_knife(arg1, *args, **kwargs):  
    print (arg1)  
    print (args)  
    print (kwargs)  
    return None
```

```
swiss_knife(1, 2, [3, 4], b=-1, a=0)
```

```
1  
(2, [3, 4])  
{ 'b': -1, 'a': 0 }
```

фактический синтаксис здесь – «*» и «»**

***args собирает аргументы в кортеж**

****kwargs – в словарь**

Аргументы функций (с неизвестным числом аргументов)

```
# arg1 - фиксированный
# (здесь - 1 обязательно передать)
# args - произвольные
# kwargs - любые
```

```
def swiss_knife(arg1, *args,
                **kwargs):
    print (arg1)
    print (args)
    print (kwargs)
    return None
```

```
swiss_knife(1)
```

```
1
()
{}
```

```
d = {'a':1, 'b':2}
```

```
swiss_knife(d)
{'b': 2, 'a': 1}
()
{}
```

```
swiss_knife(*d) # распаковка
b
('a',)
{}
```

```
swiss_knife(**d) # будет ошибка
at least 1 argument (0 given)
```

```
s = [1, 2, 3]
swiss_knife(s)
[1, 2, 3]
()
{}
```

```
swiss_knife(*s)
1
(2, 3)
{}
```

```
swiss_knife(1, 2, b=-1, a=0, [3,
4])
# так нельзя!
```

Лямбда-функции (анонимные)

```
# лямбда-функции (анонимные)
func = lambda x, y: x + y

print (func(1, 2))
```

3

**неявная инструкция return –
что вычисляется, то сразу возвращается**

Пример использования

```
print (sorted(tuples, key=lambda x: x[0])) # так по умолчанию!
print (sorted(tuples, key=lambda x: x[0]*x[0]))
```

```
[(-2, 'MINUS TWO'), (-1, 'MINUS ONE'), (1, 'ONE'), (4, 'FOUR')]
[(1, 'ONE'), (-1, 'MINUS ONE'), (-2, 'MINUS TWO'), (4, 'FOUR')]
```

КСТАТИ, МОЖНО ТАК:

```
import operator
sorted(tuples, key=operator.itemgetter(0))
```

Является ли объект вызываемым (функцией)

```
f = lambda x: x + 1
x = [1, 2, 3]
callable(f), callable(len), callable(x)
```

```
(True, True, False)
```

**Любой объект может вести себя как функция, если определить
__call__**

Сохранение значений аргументов

```
# lst - хранится...
def mylist(val, lst=[]):
    lst.append(val)
    return lst
```

```
print(mylist(1))
print(mylist(2))
print(mylist(3))
```

```
[1]
[1, 2]
[1, 2, 3]
```

Значения по умолчанию

вычисляются один раз – в момент определения функции.

Python просто присваивает это значение (**ссылку на него!**) нужной переменной при каждом вызове функции.

```
# lst не сохраняется!
def mylist(val, lst=None):
    lst = lst or []
    # if lst is None:
    #     lst = []
    lst.append(val)
    return lst
```

```
print(mylist(1))
print(mylist(2))
print(mylist(3))
```

```
[1]
[2]
[3]
```

Часто очень полезно!

Тут не используем изменяемое значение как значение по умолчанию.

Глобальные и локальные переменные

```
# глобальные переменные
globvar = 0

def set_globvar_to_one():
    global globvar # глобальная
    # без этого нельзя сделать,
    # например, globvar+=1
    globvar = 1 # если не объявить
    # глобальной - тут будет локал.

def print_globvar():
    # не надо объявлять
    print (globvar)

set_globvar_to_one()
print_globvar()
```

1

Чтобы изменить глобальную переменную – сообщите интерпретатору

```
def f():
    min = 1 # локальная переменная
    max = 10
    def g():
        min = 2 # другая локал. п.
        print('locals = ' +
              str(locals()))

    g()
    print('locals = ' +
          str(locals()))

    g()

max = 0 # глобальная переменная
f()
print(min, max) # встроенная функция

locals = {'min': 2}
locals = {'max': 10, 'g': <function g
at 0x0000000003C507B8>, 'min': 1}
locals = {'min': 2}
<built-in function min> 0
```

Глобальные и локальные переменные

```

b = 10
def f(a):
    # global b - если вставить - работает!
    print (a)
    print (b)
    b = 0 # ИЛИ если убрать - работает!

```

```
f(1)
```

```

1
UnboundLocalError:
local variable 'b'
referenced before assignment

```

3	0 LOAD_GLOBAL	0 (print)
	3 LOAD_FAST	0 (a)
	6 CALL_FUNCTION	1 (1)
positional, 0 keyword pair)	9 POP_TOP	
4	10 LOAD_GLOBAL	0 (print)
	13 LOAD_FAST	1 (b)
	16 CALL_FUNCTION	1 (1)
positional, 0 keyword pair)	19 POP_TOP	
5	20 LOAD_CONST	1 (0)
	23 STORE_FAST	1 (b)
	26 LOAD_CONST	0 (None)
	29 RETURN_VALUE	

Сначала выводится 1!

Интерпретатор думает, что надо распечатать локальную переменную, а она ещё не объявлена!

Глобальные и локальные переменные

Замыкание – ф-ия, которая обращается к неглобальным переменным, которые определены вне её тела

```
def mean_dinamic_times():
    count = 0
    total = 0

    def averager(new_val):
        nonlocal count, total # иначе ошибка
        count += 1
        total += new_val
        return total / count

    return averager

t = mean_dinamic_times()

t(1), t(2), t(3)

(1.0, 1.5, 2.0)
```

**Считаем среднее
динамического ряда –
после добавления
очередного значения**

**Без nonlocal
интерпретатор
расценивает count += 1
как объявление
переменной (локальной)!**

Передача аргументов по ссылке

по ссылке

```
a = {'a' : 1, 'b' : 2}
def f(a):
    a['b'] = 20
    a.update({'c' : 3})
    print('in', a)
print ('before', a)
f(a)
print('out', a)
('before', {'a': 1, 'b': 2})
('in', {'a': 1, 'c': 3, 'b': 20})
('out', {'a': 1, 'c': 3, 'b': 20})
```

```
a = [1, 2]
def f(a):
    a[1] = 3
    print('in', a)
print ('before', a)
f(a)
print('out', a)
('before', [1, 2])
('in', [1, 3])
('out', [1, 3])
```

по значению

(на самом деле поведение как
«по значению»)

```
a = 1
def f(a):
    a = 2
    print('in', a)
print ('before', a)
f(a)
print('out', a)

('before', 1)
('in', 2)
('out', 1)
```

Зависит от изменяемости типа

Но не всё так просто...
Попробуйте a = [3, 4]

Описание функций

```
class MyClass:
    """
    __comment__
    """
    def __init__(self, var):
        self.var = var
    def __repr__(self):
        # для разработчиков
        return '__repr__ %g' % self.var
    def __str__(self):
        # удобочитаемо (если нет - repr)
        return '__str__ %g' % self.var

mc = MyClass(2)
print (mc)
print (MyClass)
help(mc)
str(mc)
mc
```

```
__str__ 2

<class '__main__.MyClass'>

Help on MyClass in module __main__ object:

class MyClass(builtins.object)
|   __comment__
|
|   Methods defined here:
|
|   __init__(self, var)
|       Initialize self.
|   __repr__(self)
|       Return repr(self).
|
|   __str__(self)
|       Return str(self).
|
|   ...
__str__ 2

__repr__ 2
```

Обобщённые / специальные функции

Их реализация м.б. специализирована для конкретного типа

```
print (len([1, 2, 3]))  
print (len({1, 2, 3}))  
print (len(range(4)))
```

3

3

[0, 1, 2, 3]

```
print (str([1, 2, 3]))  
print (str({1, 2, 3}))  
print (str(range(4)))
```

[1, 2, 3]

set([1, 2, 3])

[0, 1, 2, 3]

```
print (sum([1, 2, 3]))  
print (sum({1, 2, 3}))  
print (sum(range(4)))
```

6

6

6

Списковые включения (List Comprehensions)

= генераторы списков != генераторы

```
[(i, j) for i in range(3) for j in range(5) if i > j]  
[(1, 0), (2, 0), (2, 1)]
```

```
[x**2 for x in range(5)]  
[0, 1, 4, 9, 16]
```

+ zip

```
['%s=%s' % (x, y) for y, x in zip(range(5), 'abcde')]  
['a=0', 'b=1', 'c=2', 'd=3', 'e=4']
```

Set Comprehensions

```
lst = [10, 5, 100, 3, 20, 10, 3, 20]  
{x for x in lst if 10 * round(x / 10) == x}  
{10, 20, 100}
```

Dictionary Comprehensions

```
{x: y for y, x in zip(range(5), 'abcde') if y < 3}  
{ 'a': 0, 'b': 1, 'c': 2 }
```

Reduce (это итератор)

```
from functools import reduce
```

```
reduce(lambda x, y: x * y, [1, 2, 3, 4])
```

24

```
reduce(lambda x, y: '(' + str(x) + '*' + str(y) + ')',  
        [1, 2, 3, 4])
```

(((1*2) *3) *4)

В Python 2.x была обычная функция reduce...

Map (это итератор)

```
l1 = [1, 2, 3, 4]
```

```
l2 = [0, -1, +1, +2] # если разной длины - то по длине наименьшей
```

```
list(map(lambda x, y: x + y, l1, l2))
```

```
[1, 1, 4, 6]
```

```
list(map(max, l1, l2))
```

```
[1, 2, 3, 4]
```

Если списки разной длины, то по длине наименьшего.

Map с одним списком

```
l = [1, 2, 3]
```

```
list(map(lambda x: x * x, l))
```

```
[1, 4, 9]
```

```
list(map(lambda x: x * x, range(3))) # по генератору
```

```
[0, 1, 4]
```

Но эффективнее и понятнее с помощью генератора списков!!!

```
[x * x for x in l]
```

Filter (это итератор)

```
list(filter(lambda x: x.isalpha(), 'Привет, мир!'))  
['П', 'р', 'и', 'в', 'е', 'т', 'м', 'и', 'р']
```

```
list(filter(lambda x: x % 2, range(10)))  
[1, 3, 5, 7, 9]
```

list нужен в Python3, т.к. там filter возвращает итератор

```
lst = [12, 1.2, '12', 1, 2]  
list(filter(lambda x: type(x) is int, lst))  
[12, 1, 2]
```

Но лучше с помощью генератора списка:

```
[x for x in lst if type(x) is int]
```

Zip (это итератор)

```
x = range(5)
y = 'abcde'
z = [0, 1, 0, 1, 0]
```

```
list(zip(x, y, z)) # list - python3
```

```
[(0, 'a', 0), (1, 'b', 1), (2, 'c', 0), (3, 'd', 1), (4, 'e', 0)]
```

Задача: по строке сформировать перечень пар соседних букв

```
x = 'Привет!'
list(zip(x, x[1:])) # можно подавать разные по длине аргументы!
```

```
[('П', 'р'), ('р', 'и'), ('и', 'в'), ('в', 'е'), ('е', 'т'),  
('т', '!')]
```

Итераторы

**Цикл for работает с любой последовательностью
(есть next до исключения StopIteration)**

что такое итератор

```
it = iter([1, 2, 3, 4, 5])
```

```
print (next(it))
```

```
print (next(it))
```

```
print ([x for x in it])
```

```
1
```

```
2
```

```
[3, 4, 5]
```

```
print (next(it))
```

```
# будет исключение
```

```
# итератор генерирует последовательность:
```

```
# элемент = сумме предыдущих
```

```
def forit(mystate=[]):
```

```
    if len(mystate) < 5:
```

```
        new_element = max(sum(mystate), 1)
```

```
        mystate.append(new_element)
```

```
        # print (mystate)
```

```
    return new_element
```

```
it2 = iter(forit, None) # если не возвращает  
значения явно, то None
```

```
[x for x in it2]
```

```
[1, 1, 2, 4, 8]
```

Итераторы: enumerate

```
[x for x in enumerate("abcd")]
```

```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

```
dict(enumerate(['test', 'train', 'val']))  
{0: 'test', 1: 'train', 2: 'val'}
```

```
for i, j in enumerate("abcd"):  
    print (i, j)
```

```
(0, 'a')  
(1, 'b')  
(2, 'c')  
(3, 'd')
```

**Это способ иметь
индекс итерирования в
цикле**

Модуль itertools

соединение 2х итераторов

```
from itertools import chain
it1 = iter([1, 2, 3])
it2 = iter([4, 5])

a = []
for i in chain(it1, it2):
    a.append(i)
print (a)
[1, 2, 3, 4, 5]
```

повторение итератора

```
from itertools import repeat
b = []
for i in repeat(1, 4):
    b.append(i)
print (b)
[1, 1, 1, 1]
```

бесконечный итератор

```
from itertools import count
c = []
for i in count(1):
    c.append(i)
    if i > 10:
        break
print(c)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

циклический итератор

```
from itertools import cycle
d = []
for i, j in enumerate(cycle([1, 2, 3])):
    d.append(j)
    if i > 10:
        break
print (d)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Модуль itertools

Срезы

```
from itertools import islice
e1 = islice(range(10), 3, 6)
e2 = islice(range(10), 3, 9, 2)

print ("e[3:6] " + str(list(e1)))
e[3:6] [3, 4, 5]

print ("e[3:9:2] " + str(list(e2)))
e[3:9:2] [3, 5, 7]
```

«части» итератора

```
from itertools import dropwhile
# ещё есть takewhile
f = dropwhile(lambda x: x<5, range(10))
print (list(f))
[5, 6, 7, 8, 9]
```

независимые копии итераторов

```
from itertools import tee
it = range(3)
# три н. копии
a, b, c = tee(it, 3)
# этот итератор "уничтожится"
tmp = list(c)
print (list(a), list(b), list(c))
[0, 1, 2] [0, 1, 2] []
```

декартово произведение

```
from itertools import product
it = product("AB", repeat=2)
print (list(it))
[('A', 'A'), ('A', 'B'), ('B', 'A'), ('B', 'B')]
```

Модуль itertools

перестановки

```
from itertools import permutations
```

```
it = permutations("YN")
print (list(it))
[('Y', 'N'), ('N', 'Y')]
```

сочетания (без повторений)

```
from itertools import combinations
```

```
it = combinations("ABC", 2)
print (list(it))
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

сочетания с повторениями

```
from itertools import combinations_with_replacement
```

```
it = combinations_with_replacement("AB", 2)
print (list(it))
[('A', 'A'), ('A', 'B'), ('B', 'B')]
```

группировка

```
from itertools import groupby
```

```
it = groupby("ABBAACBCC")
for i, j in it:
    print (i, list(j))
```

```
('A', ['A'])
('B', ['B', 'B'])
('A', ['A', 'A', 'A'])
('C', ['C'])
('B', ['B'])
('C', ['C', 'C'])
```


Пишем свой итератор

```
class Fibonacci:
    """
    Итератор последовательности
    Фибоначчи до N
    """

    def __init__(self, N):
        self.n, self.a, self.b, self.max = 0, 0, 1, N

    def __iter__(self):
        return self

    # должна быть такая функция
    def __next__(self): # Python 2: def next(self)
        if self.n < self.max:
            a, self.n, self.a, self.b = self.a, self.n+1, self.b,
self.a+self.b
            return a
        else:
            raise StopIteration

list(Fibonacci(10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Пишем свой итератор

простой вариант

```
class Identity:
    def __getitem__(self, idx):
        if idx > 5:
            raise IndexError(idx)
        return idx
```

```
list(Identity())
[0, 1, 2, 3, 4, 5]
```

Генератор

– упрощённый итератор

сравните...

```
class Repeater:
    def __init__(self, value):
        self.value = value
    def __iter__(self):
        return self
    def __next__(self):
        return self.value

def repeater(value):
    while True:
        yield value
```

Итератор – объект, который имеет метод `__next__` (`next` в Python2)

Генератор – функция, в которой есть `yield`-выражение

генератор \Rightarrow итератор

Простые генераторы

помогают делать ленивые вычисления (lazy computations)

```
def Fib(N):  
    a, b = 0, 1  
    for i in range(N):  
        yield a # вместо return  
        # для выдачи след. значения  
        a, b = b, a + b
```

```
for i in Fib(10):  
    print (i)
```

0
1
1
2
3
5
8
13
21
34

yield - похожа на return,

но работа функции приостанавливается и
выдаётся значение

```
def f():  
    yield 1  
    yield 2  
    yield 3  
    # return 10 - нет эффекта
```

```
def g():  
    # взять выход у f!  
    x = yield from f()  
    yield 4  
    yield 5  
    yield 6  
    # return 100 - нет эффекта
```

```
list(g())  
[1, 2, 3, 4, 5, 6]
```

Простые генераторы

```
def double_numbers(iterable):  
    for i in iterable:  
        yield i + i
```

```
list(double_numbers(range(5)))  
[0, 2, 4, 6, 8]
```

генераторное выражение (Generator Expressions)

```
# это список  
print ( [x * x for x in range(5)] )  
[0, 1, 4, 9, 16]
```

```
# а это - генераторное выражение  
print ( (x * x for x in range(5)) )  
<generator object <genexpr> at  
0x00000000046F6BA0>
```

```
# тоже генераторное выражение  
print ( sum(x * x for x in range(5)) )  
30
```

Генератор нельзя переиспользовать

```
gen = (x*x for x in range(5))  
print ('использование  
генераторного выражения:')  
for y in gen:  
    print (y)
```

```
# ничего не будет!  
print ('переиспользование:')  
for y in gen:  
    print (y)
```

использование генераторного
выражения:

0
1
4
9
16

переиспользование:

Сопрограммы (coroutines)

больше одной точки входа

остановка исполнения, сохранение состояния и продолжение

```
def grep(pattern):  
    print("Ищем {!r}".format(pattern))  
    while True:  
        line = yield # точка входа в line засылает метод send  
        if pattern in line:  
            print('нашли в: ' + line)
```

```
gen = grep("Мир")
```

next(gen) # обязательно нужно - это инициализация (инициализацию можно спрятать в декораторе)

Ищем 'Мир'

```
gen.send("Предложение")
```

```
gen.send("Предложение с Миром")
```

нашли в: Предложение с Миром

```
gen.send("Предложение с миром")
```

```
gen.send("Миру мир!")
```

нашли в: Миру мир!

Декораторы

**Декоратор – вызываемый объект:
вход – функция (декорируемая),
выход – функция**

**позволяет расширять и изменять поведение вызываемых объектов
(функций, методов и классов) без их необратимой модификации**

```
def decorate(f):  
    print('decorate')  
    return f
```

```
@decorate  
def func():  
    print('func')
```

decorate

```
func()
```

func

**декоратор ничего не меняет,
но «действует» до функции
выполняются сразу после загрузки
модуля!**

```
def decorate(f):  
    print('decorate')  
    return len
```

```
@decorate  
def func(x):  
    print('func', x)
```

decorate

```
func([1, 2])
```

2

декоратор делает подмену функции!

Сохраняет читабельность кода

Цепочки декораторов: декораторов может быть много!

```
def square(f): # на вход - функция
    # выход - функция,
    # которая будет реально
    выполняться
    return lambda x: f(x * x)
```

```
def add1(f): # на вход - функция
    # выход - функция,
    # которая будет реально
    выполняться
    return lambda x: f(x + 1)
```

```
# два декоратора у функции
@square
@add1
def time2(x):
    return (x * 2)
```

```
time2(3) # (3*3 + 1)*2
```

```
def makebold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped
```

```
def makeitalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped
```

```
@makebold
@makeitalic
def hello():
    return "hello world"
```

```
print(hello())
<b><i>hello world</i></b>
```

Пройдитесь по декораторам «сверху вниз»!

Здесь – они готовят данные для функции

Декораторы с аргументами

```
def dec_upper(f):  
    def g(*args, **kwargs):  
        return f(*args, **kwargs).upper()  
    return g
```

```
def dec_lower(f):  
    def g(*args, **kwargs):  
        return f(*args, **kwargs).lower()  
    return g
```

```
@dec_upper  
@dec_lower  
def say(x):  
    return('Hello' + x + '!') # ' + x + '
```

```
say('Mike')
```

```
'HELLOMIKE!'
```

Декораторы

```
class my_decorator(object):  
    def __init__(self, f):  
        print("внутри my_decorator.__init__()")  
        f() # вызов - можно убрать  
    def __call__(self):  
        print("внутри my_decorator.__call__()")
```

```
@my_decorator  
def aFunction():  
    print("внутри aFunction()")
```

```
print("окончание декорирования aFunction()")
```

```
aFunction() # вызов функции
```

внутри my_decorator.__init__()

внутри aFunction()

окончание декорирования aFunction()

внутри my_decorator.__call__()

**на самом деле
вызывается
my_decorator.__call__()**

Декораторы

```
class entry_exit(object):  
    def __init__(self, f):  
        self.f = f  
    def __call__(self):  
        print("ВЫЗОВ - ", self.f.__name__)  
        self.f()  
        print("ВЫХОД - ", self.f.__name__)
```

```
@entry_exit  
def func1():  
    print("работа func1()")
```

```
@entry_exit  
def func2():  
    print("работа func2()")
```

**Что-то делаем до и
после вызова
функции...**

**Внешне это не
заметно – просто
вызываем функцию**

func1()

ВЫЗОВ - func1
работа func1()
ВЫХОД - func1

func2()

ВЫЗОВ - func2
работа func2()
ВЫХОД - func2

Декораторы

```
def entry_exit(f):  
    def new_f():  
        print("Вызов: ", f.__name__)  
        f()  
        print("Выход: ", f.__name__)  
    new_f.__name__ = f.__name__  
    # меняем даже имя функции  
    # (попробуйте убрать)  
    return new_f
```

```
@entry_exit  
def func1():  
    print("работа func1()")
```

```
@entry_exit  
def func2():  
    print("работа func2()")
```

**здесь декоратор –
функция**

**главное, чтобы декоратор
можно было вызвать**

func1()

Вызов: func1
работа func1()
Выход: func1

func2()

Вызов: func2
работа func2()
Выход: func2

```
print(func1.__name__)  
func1
```

Декоратор без аргументов (можно пока пропустить)

```
class decorator_without_arguments(object):
    def __init__(self, f):
        """
        передаём в конструкторе функцию
        """
        print("Работа __init__()")
        self.f = f
    def __call__(self, *args):
        """
        В __call__ передаём аргументы.
        """
        print("Работа __call__()")
        self.f(*args)
        print("После self.f(*args)")

@decorator_without_arguments
def sayHello(a1, a2, a3, a4):
    print('Аргументы:', a1, a2, a3, a4)
```

Работа __init__()

**В методе __call__ решаем
проблему с передачей функции
аргументов**

```
sayHello("say", "hello",
"argument", "list")
```

Работа __call__()
Аргументы: say hello argument
list
После self.f(*args)

```
sayHello("a", "different", "set
of", "arguments")
Работа __call__()
Аргументы: a different set of
arguments
После self.f(*args)
```

Декоратор с аргументами (можно пока пропустить)

```
class decorator_with_arguments(object):
    def __init__(self, arg1, arg2, arg3):
        """
        Если пишем с аргументами,
        то их передаём в конструктор,
        а функция не передаётся!
        """
        print("работа __init__()")
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3

    def __call__(self, f):
        """
        Если пишем с аргументами, то __call__()
        вызывается лишь раз,
        как часть процесса декорации,
        ей можно передать только функцию!
        """
        print("работа __call__()")

        def wrapped_f(*args):
            print("работа wrapped_f()")
            print("Аргументы:", self.arg1,
self.arg2, self.arg3)
            f(*args)
            print("выход из f(*args)")

        return wrapped_f
```

```
@decorator_with_arguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print('Аргументы sayHello:', a1, a2, a3,
a4)

print("После декорации...")
sayHello("say", "hello", "argument", "list")
print("Ещё раз...")
sayHello("a", "different", "set of",
"arguments")
```

работа __init__()
работа __call__()
После декорации...
работа wrapped_f()
Аргументы: hello world 42
Аргументы sayHello: say hello argument list
выход из f(*args)
Ещё раз...
работа wrapped_f()
Аргументы: hello world 42
Аргументы sayHello: a different set of
arguments
выход из f(*args)

Декораторная функция с аргументами (можно пока пропустить)

```
def decorator_function_with_arguments(arg1, arg2, arg3):  
    def wrap(f):  
        print("Внутри wrap()")  
        def wrapped_f(*args):  
            print("Внутри wrapped_f()")  
            print("Аргументы ДЕКОРАТОРА:", arg1, arg2,  
arg3)  
            f(*args)  
            print("After f(*args)")  
        return wrapped_f  
    return wrap  
  
@decorator_function_with_arguments("hello", "world", 42)  
def sayHello(a1, a2, a3, a4):  
    print('аргументы функции:', a1, a2, a3, a4)  
  
print("После декорации")  
sayHello("say", "hello", "argument", "list")  
print("Ещё разок...")  
sayHello("a", "different", "set of", "arguments")
```

```
Внутри wrap()  
После декорации  
Внутри wrapped_f()  
Аргументы ДЕКОРАТОРА: hello world  
42  
аргументы функции: say hello  
argument list  
After f(*args)  
Ещё разок...  
Внутри wrapped_f()  
Аргументы ДЕКОРАТОРА: hello world  
42  
аргументы функции: a different set  
of arguments  
After f(*args)
```

Именованные переменные – «дандеры» (double underscores – с двойным подчёркиванием)

для внутреннего пользования

`_var = 1` #, не импортируются `from my_module import *`

чтобы избежать конфликта имён

`var_ = 1`

нельзя использовать (механизм защиты)

`__var = 1` # искажением имени (name mangling)

зарезервированные (`__init__`, `__call__`, ...)

`__var__ = 1` # не будет искажения! Зарезервированы: и т.п.

неважные переменные

`_ = 1` # при распаковке,
часто: результат последнего выражения

Модули

модуль – один файл с расширением *.py (сейчас уже и zip-архив), задаёт своё пространство имён

пакет – директория, в которой есть файл `__init__.py` (просто для организации кода). Может содержать поддиректории. Пользователю не так важно, с чем работать

```
import datetime # импортируем модуль
```

```
# появляется объект с этим названием
```

```
datetime.date(2004, 11, 20)
```

```
2004-11-20
```

```
print (datetime.__name__) # имя
```

```
datetime
```

```
print (datetime.__doc__) # описание
```

```
Fast implementation of the datetime type.
```

```
print (datetime.__file__) # файл
```

```
C:\Anaconda3\lib\datetime.py
```

```
pak1
```

```
|-- __init__.py
```

```
|-- pak12
```

```
| |-- __init__.py
```

```
| |-- f.py
```

```
|-- h.py
```

```
from pak1.pak12 import f
```

Модули

можно назначать синонимы модулей и функций

```
import datetime as dt # сокращение имени модуля
print (dt.date(2004, 11, 20))
2004-11-20
```

```
# импортирование конкретной функции
from datetime import date as dt
print (dt(2004, 11, 20))
2004-11-20
```

не рекомендуется такой импорт

```
from datetime import *
```

здесь питон ищет модули

```
import sys
sys.path
['',
 'C:\\\\Users\\Александр Дьяконов\\Anaconda3\\python35.zip',
 'C:\\\\Users\\Александр Дьяконов\\Anaconda3\\DLLs',
 'C:\\\\Users\\Александр Дьяконов\\Anaconda3\\lib',
 'C:\\\\Users\\Александр Дьяконов\\Anaconda3',
```

Модули: перезагрузка

```
reload(module) # Python 2.x
```

```
importlib.reload # >=Python 3.4
```

```
imp.reload # Python 3.0 - 3.3
```

**Несколько раз использовать
`import` бесполезно!**

Исключения

```
class MyError(Exception): # от Exception
    pass

try:
    # что-то делаем...

    raise (MyError) # исключение
# else: Python3 - не работает!
#     print ("Не было исключения")

# принимаем исключение
except ZeroDivisionError:
    print ("Деление на 0")
except KeyboardInterrupt:
    print ("Прерывание с клавиатуры")
except (MyError):
    print ("Моя ошибка")
except:
    print ("Ошибка")
finally:
    print ("Последние действия")
    # тут можно, например, закрыть файл
```

Exception – класс, от которого должны наследоваться все исключения

AssertionError – ошибка assert

ImportError – ошибка import

NameError – не найдена переменная

AttributeError – читаем/пишем несуществующий атрибут

KeyError / IndexError – в контейнере нет элемента по указанному ключу / индексу

TypeError – несоответствие типов см.

<https://docs.python.org/3/library/exceptions.html>

Исключения

```
# обработка любого исключения
import traceback

try:
    1 / 0
except Exception as e:
    print (e.args) # переданные аргументы
    # информация о стеке вызовов
    # на момент исключения
    traceback.print_tb(e.__traceback__)
    # pass

print ('выполнение программы продолжается')

('division by zero',)

File "<ipython-input-9-640c85b08d09>", line
5, in <module>
    1/0
```

выполнение программы продолжается

**Уже встречали менеджер
контекста
(также основан на
исключениях)**

```
with resource_1() as r1, \
    resource_2() as r2:
    do_smth(r1, r2)

# в случае с файлами -
# не надо явно закрывать
```

Менеджер контекста

эти фрагменты эквивалентны

```
with open('hello.txt', 'w') as f:  
    f.write('привет, мир!')
```

```
f = open('hello.txt', 'w')  
try:  
    f.write('привет, мир!')  
finally:  
    f.close()
```

Пример – печатаем с отступами

```
class Indenter:  
    def __init__(self):  
        self.level = 0  
  
    def __enter__(self):  
        self.level += 1  
        return self  
  
    def __exit__(self, exc_type, exc_val,  
exc_tb):  
        self.level -= 1  
  
    def print(self, text):  
        print(' ' * self.level + text)
```

```
with Indenter() as indent:  
    indent.print('раз')  
    with indent:  
        indent.print('два')  
        with indent:  
            indent.print('три')  
            indent.print('четыре')
```

раз
 два
 три
 четыре

Пишем менеджер контекста

```
class m_c:
    def __init__(self):
        print ('init')
        # возвращает None

    def __enter__(self):
        # тут можно, например, открыть файл
        print ('enter')
        # возвращаемое значение доступно по имени with ... as NAME:
        return ('m_c-name')

    def __exit__(self, *exc_info):
        # вызывается в конце, аргументы:
        # - тип исключения
        # - исключение
        # - объект типа traceback
        print ('exit')
```



```
with m_c() as name:
    print ('working... with ' + name)
```

init
enter
working... with m_c-name
exit

Особенности хранения объектов

интерпретатор кэширует числа от -5 до 256 Иногда это происходит и
Каждое число – лишь один объект так

```
print (int("-5") is -5)
True
print (int("-6") is -6)
False
```

```
a = 257
b = 257
print (a is b)
False
```

Разные задания **одного** объекта

```
s1 = "abcde"
s2 = "abcde"
s3 = "ab" + "cd" + "e"

print ("hash:", hash(s1), hash(s2), hash(s3))
('hash:', -1332677140, -1332677140, -1332677140)

print ("id:", id(s1), id(s2), id(s3))
# а раньше id(s1) = id(s2) != id(s3)
('id:', 66624808L, 66624808L, 66624808L)
```

**умный интерпретатор
сделает одним объектом**

```
def f():
    a = 257 #
    b = 257 #
    print (a is b)
```

```
f()
True
print (1000+0 is 1000+0)
False
print (1000 is 1000)
True
```


Байткод

□ интерпретатор CPython исполняет программы так:
трансляция в промежуточный байткод
исполнение байткода на VM со стековой архитектурой

```
# байткод
import dis
```

```
def myf(x):
    return(x + 1, x - 1)
```

```
dis.dis(myf)
```

5	0	LOAD_FAST	0	(x)
	3	LOAD_CONST	1	(1)
	6	BINARY_ADD		
	7	LOAD_FAST	0	(x)
	10	LOAD_CONST	1	(1)
	13	BINARY_SUBTRACT		
	14	BUILD_TUPLE	2	
	17	RETURN_VALUE		

Немного о скорости

**В цикле наращиваем список (не совсем правильное решение)
Как быстрее?**

```
def f():  
    l = []  
    for i in range(10000):  
        l.append(i)  
    return l
```

```
%timeit f()
```

1000 loops, best of 3: 675 µs
per loop

**Здесь дольше, т.к. в цикле будем
искать метод в хэш-таблице**

**На время timeit отключается
сборщик мусора**

```
def g():  
    l = []  
    li = l.append  
    # сразу сообщим функцию –  
    # чтобы не искать в цикле  
    for i in range(10000):  
        li(i)  
    return l
```

```
%timeit g()
```

1000 loops, best of 3: 440 µs
per loop

**Здесь знаем метод, который
вызывать**

Немного о скорости

второй вариант быстрее:

```
%timeit x, y = 1, 2  
17 ns ± 1.68 ns per loop
```

```
%timeit x = 1; y = 2  
12.9 ns ± 0.0339
```

Когда нет приведения типов

```
x = 10  
%timeit "a = x > 1.0"  
6.94 ns ± 0.104
```

```
x = 10.  
%timeit "a = x > 1.0"  
6.37 ns ± 0.015
```

Последний вариант самый быстрый

```
x = "one"  
y = "two"  
  
print (x + '+' + y)  
print ("".join((x, '+', y)))  
print ("%s+%s" % (x, y))  
print ("{0}+{1}".format(x, y))  
print (f"{x}+{y}")
```

иногда зависит от версии... что быстрее:

x + x или **x * 2**
x * x или **x ** 2**

Неожиданное поведение

Не увлекайтесь функциональным программированием...

```
l = [lambda: x for x in "abcdefg"]  
for r in l:  
    print (r())
```

g
g
g
g
g
g
g
g

```
# объяснение - особенности лямбда-функций  
print (id(lambda: 1) == id(lambda: 2))
```

True

Для неименованных лямбда-функций только одна ссылка!

Проверка all, any

```
lst = [2, 5, 7, 5]
```

```
# какой-нибудь элемент
```

```
if (any(x > 1 for x in lst)):  
    print('any - yes')
```

any - yes

```
# все элементы
```

```
if (all(x > 1 for x in lst)):  
    print('all - yes')
```

all - yes

Вычисления eval

```
a = 1
```

```
b = 2
```

```
c = eval('a + b') # вычисление выражений
```

```
# eval('c = a + b') # нельзя так
```

Интересности: приоритет

```
i, j = 1, 2
print("%i" % (i * j))
2
print("%i" % i * j)
11
```

% и * имеют один приоритет!

Поэтому ответ '1'*2

Интересности: хеширование

```
{True: 'да', 1: 'нет', 1.0: 'возможно'}
{True: 'возможно'}
```

```
True == 1 == 1.0
True
```

Объектно-ориентированное программирование (ООП)

— методология программирования, основанная на

- представлении программы в виде совокупности объектов,
- каждый объект является экземпляром определенного класса,
- классы образуют иерархию наследования.

**в Python всё – объекты
имеют id и значение**

```
a = 1
b = [1, 2]
print (id(a), id(b))
a = a + 1
b.append(3) # id не изменится
print (id(a), id(b))
del a # удаление объекта
# a # будет ошибка
```

```
1578102544 73531656
1578102576 73531656
```

**есть понятие «класс»,
есть – «экземпляры класса»
(объекты)**

```
class MyClass:
    val = 0
```

```
m = MyClass()
```

```
Myclass, m
```

```
(__main__.Myclass,
<__main__.Myclass at 0x7fd1391e1b00>)
```

Определение класса

первый аргумент всех методов - экземпляр класса

```
class MyGraph:
```

```
    def __init__(self, V, E): # конструктор (деструктор - __del__)
```

```
        self.vertices = set(V)
```

```
        self.edges = set(E)
```

```
    def add_vertex(self, v): # метод - функция, объявленная в теле класса
```

```
        self.vertices.add(v)
```

```
    def add_edge(self, e):
```

```
        self.vertices.add(e[0])
```

```
        self.vertices.add(e[1])
```

```
        self.edges.add(e)
```

```
    def __str__(self): # представление в виде строки
```

```
        return ("%s; %s" % (self.vertices, self.edges))
```

```
g = MyGraph([1, 2, 3], [(1, 2), (1, 3)])
```

```
g.add_edge((3, 4))
```

```
print (g)
```

```
print (g.vertices)
```

```
print (g.__getattr__('vertices'))
```

```
g.__setattr__('vertices',  
              set([1, 2, 3, 4, 5]))
```

```
print (g)
```

```
{1, 2, 3, 4}; {(1, 2), (1, 3), (3, 4)}
```

```
{1, 2, 3, 4}
```

```
{1, 2, 3, 4}
```

```
{1, 2, 3, 4, 5}; {(1, 2), (1, 3), (3, 4)}
```


Атрибуты (были особенности именования переменных)

```
class X:
    a = 0 # обычный атрибут
    _b = 1 # не желательно пользоваться, но доступен
    __c = 2 # доступен под другим именем ('_X__c')
print (dir(X))
print (X.__dict__) # все атрибуты в виде словаря
```

```
['_X__c', '__doc__', '__module__', '_b', 'a']
{'a': 0, '__module__': '__main__', '_X__c': 2, '_b': 1, '__doc__': None}
```

Определение всех возможных атрибутов

Занимает меньше памяти

```
class Myclass:
    __slots__ = ['name1', 'name2'] # ВСЕ возможные атрибуты

c = Myclass()
c.name1 = 10
c.name2 = lambda x: x * x
# c.name3 = 20 # ошибка в Python3 (в Python2 - нет)
```

Множественное наследование

```
class A:
    def make(self, x):
        print(x+1)
class B:
    def make(self, x):
        print(x-1)
class C(A, B):
    pass

c = C()
c.make(2)
print (C.__mro__) # в таком порядке ищутся методы
```

3

```
(<class '__main__.C'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'object'>)
```

Другой способ задания классов (динамический)

```
class C: pass
# эквивалентная запись:
C = type('C', (), {})
```

type – дефолтный метакласс(класс для создания других классов)

```
def myprint(self):
    print("список: ", self)
```

```
# такой способ создания класса
```

```
MyList = type('MyList', (list,), dict(x=10, myprint=myprint))
```

```
m1 = MyList()
m1.append("one")
m1.append("two")
m1.myprint()
```

```
список:  ['one', 'two']
```

Атрибуты/методы класса и экземпляра

```
class MyClass: pass
# это класс!
MyClass.field = 10
MyClass.method = lambda x: u"Привет, мир!"

x = MyClass()
x.field2 = 5
x.method2 = lambda x: u"Пока, мир!"

print(x.field, x.method(), x.field2, x.method2(None))

10 Привет, мир! 5 Пока, мир!
```

Для `x.method2` нужен аргумент!

Атрибуты / методы можно приписывать (изменять!) после инициализации

При изменении атрибута класса – изменение будет во всех экземплярах

Атрибуты/методы класса и экземпляра

```
class My:
    a = 2 # переменная класса
    def __init__(self, b):
        self.b = b # переменная экземпляра
```

```
print (My.a)
```

2

```
m = My(10)
print (m.a, m.b)
```

2 10

```
m2 = My(20)
m.a = 0
print (m.a, m.b, m2.a, m2.b, My.a)
```

0 10 2 20 2

Значения атрибутов

```
class MyClass:  
    val = 0
```

```
m = MyClass()
```

```
setattr(m, "val", 2) # setattr – безопасное добавление атрибута  
setattr(m, "val_other", 3.0) # можно m.val_other = 3.0
```

```
print (getattr(m, "val")) # getattr – безопасный вызов атрибута  
2
```

```
print (getattr(m, "val_some_other", 1.0))  
1.0
```

```
m.__dict__  
{'val': 2, 'val_other': 3.0}
```

setattr применяется, например, когда не знаем имени атрибута
(лежит в переменной)

hasattr – есть ли у объекта метод / атрибут

Разные виды методов

```
class MyClass:
    def method(self):
        return 'вызван метод экземпляра', self
    @classmethod
    def classmethod(cls):
        return 'вызван метод класса', cls
    @staticmethod
    def staticmethod():
        return 'вызван статический метод'
```

```
obj = MyClass()
obj.method() ('вызван метод экземпляра', <__main__.MyClass at 0x7f246804a828>)
MyClass.method(obj) ('вызван метод экземпляра', <__main__.MyClass at 0x7f246804a828>)
obj.classmethod() ('вызван метод класса', __main__.MyClass)
obj.staticmethod() 'вызван статический метод'
MyClass.classmethod() ('вызван метод класса', __main__.MyClass)
MyClass.staticmethod() 'вызван статический метод'
MyClass.method() TypeError: method() missing 1 required positional argument: 'self'
```

статические методы не могут получить доступ ни к состоянию экземпляра объекта, ни к состоянию класса

Пример класса

```
class Human(object):
    species = "H. sapiens" # атрибут

    # инициализатор
    def __init__(self, name):
        self.name = name
        self.age = 0

    # метод класса, 1ый арг. - self
    def say(self, msg):
        s = "{0}: {1}".format(self.name, msg)
        return s

    # общий метод для всех экземпляров
    # первый аргумент - какой КЛАСС вызвал
    @classmethod
    def get_species(cls):
        return cls.species

    # вызывается без ссылки на вызвавшего
    @staticmethod
    def grunt():
        return "статика..."
```

```
# свойство - превращает
# метод в атрибут
@property
def age(self):
    return self._age

# для присваивания
# свойству
@age.setter
def age(self, age):
    self._age = age

# для удаления свойства
@age.deleter
def age(self):
    del self._age
```


Пример класса (продолжение)

инициализация

```
i = Human(name="Иван")  
print (i.say("привет"))
```

Иван: привет

```
j = Human("Сергей")  
print (j.say("пока"))
```

Сергей: пока

```
print (i.get_species())  
H. sapiens
```

меняем атрибут общий - для всего класса

```
Human.species = "H. neanderthalensis"  
print (i.get_species(), i.species)  
print (j.get_species(), j.species)  
H. neanderthalensis H. neanderthalensis  
H. neanderthalensis H. neanderthalensis
```

```
print (Human.grunt()) # статический метод  
Статика...
```

i.age = 42 # свойство

```
print (i.age)  
42
```

```
del i.age  
# i.age # будет исключение
```

Подсчёт числа объектов соответствующего класса

```
class Counter:
    Count = 0 # счётчик

    def __init__(self, name): # внимание к отступам
        self.name = name # обращение через self
        Counter.Count += 1
        print (name, ' created, count = ', Counter.Count)
        # Counter.Count - у класса, а не объекта!

    def __del__(self):
        Counter.Count -= 1
        print (self.name, ' deleted, count = ', Counter.Count)
        if Counter.Count == 0:
            print ('That\'s all...')

x = Counter("First")
('First', ' created, count = ', 1)
y = Counter("Second")
('Second', ' created, count = ', 2)
del x
('First', ' deleted, count = ', 1)
z = Counter("Third")
('Third', ' created, count = ', 2)
```

Дескриптор

– атрибут объекта со скрытым поведением, которое задают методы в протоколе дескриптора: `get()`, `set()`, and `delete()`

```
class RevealAccess(object):  
    """ Пример дескриптора данных: устанавливает и  
        возвращает значения, а также пишет сообщения  
    """  
  
    def __init__(self, initval=None, name='var'):  
        self.val = initval  
        self.name = name  
  
    def __get__(self, obj, objtype):  
        print('Выдаём', self.name)  
        return (self.val + 1) # выдаёт следующее число  
  
    def __set__(self, obj, val):  
        print('Получаем', self.name)  
        if val < 0:  
            print('Отрицательное значение - будет обнулено')  
            self.val = 0  
        else:  
            self.val = val
```

Дескриптор (продолжение)

```
class MyClass(object):  
    # у переменной скрытое поведение  
    x = RevealAccess(10, 'var "x"')  
    y = 5
```

```
m = MyClass()  
print(m.x)  
Выдаём var "x"  
11
```

```
m.x = 20  
Получаем var "x"
```

```
print(m.x)  
Выдаём var "x"  
21
```

```
m.x = -20  
Получаем var "x"  
Отрицательное значение – будет  
обнулено
```

```
print(m.x)  
Выдаём var "x"  
1
```

```
print(m.y) # обычное поведение  
5
```

Свойство – быстрый дескриптор

«безопасный класс» (контролирует значения атрибутов)

```
class SafeClass:
    def _get_attr(self):
        return self._x

    def _set_attr(self, x):
        assert x > 0, "необходимо положительное значение"
        self._x = x

    def _del_attr(self):
        del self._x

x = property(_get_attr, _set_attr, _del_attr)

safe = SafeClass()
safe.x = 1
# safe.x = -2 # будет исключение
```

Дополнение (другие типы данных): словарь с порядком ключей

```
import collections
d = collections.OrderedDict(one=1, two=2, three=3)
```

```
OrderedDict([('one', 1), ('two', 2), ('three', 3)])
```

словарь со значение по умолчанию

```
import collections
d = collections.defaultdict(list, one=1, two=2, three=3, default='111')
d['one'], d['2'] # по умолчанию - пустой список
```

```
(1, [])
```

словари -> в один словарь

```
from collections import ChainMap
d1 = {1:1, 2:2}
d2 = {2:3, 3:4}
d3 = {3:5, 4:6}
cm = ChainMap(d1, d2, d3)
```

```
cm[1], cm[2], cm[3], cm[4]
(1, 2, 4, 6)
```

Дополнение (другие типы данных): типизированные массивы

```
import array
arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
```

ИМЕННОВАННЫЙ СПИСОК

```
from collections import namedtuple
Car = namedtuple('Авто', 'цвет пробег автомат')
print (Car('red', '1200', True))
```

Авто(цвет='red', пробег='1200', автомат=True)

ИМЕННОВАННЫЙ СПИСОК (Python 3.x)

```
from typing import NamedTuple
class Car(NamedTuple):
    цвет: str
    пробег: float
    автомат: bool
print (Car('red', '1200', True))
```

Car(цвет='red', пробег='1200', автомат=True)

Дополнение (другие типы данных):

мультимножество

```
from collections import Counter
```

```
inventory = Counter()  
loot = {'клинок': 1, 'хлеб': 3}  
inventory.update(loot)  
inventory.update({'клинок': 1, 'яблоко': 1})  
inventory, len(inventory), sum(inventory.values())
```

```
(Counter({'клинок': 2, 'хлеб': 3, 'яблоко': 1}), 3, 6)
```

очередь с двусторонним доступом (идеально для FIFO)

```
from collections import deque  
# LifoQueue - в параллельных вычислениях  
s = deque()  
s.append('1')  
s.append('2')  
s.pop(), s.pop()  
('2', '1')
```

```
s = deque()  
s.append('1')  
s.append('2')  
s.popleft(), s.popleft()  
  
('1', '2')
```


Дополнение (другие типы данных):

очереди с приоритетом

```
from queue import PriorityQueue
q = PriorityQueue()
q.put((2, 'b'))
q.put((1, 'a'))
q.put((3, 'c'))
while not q.empty():
    next_item = q.get()
    print(next_item)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
```

Спасибо за хорошие материалы

- **Bruce Eckel** Python 3 Patterns, Recipes and Idioms
- **Никита Лесников** Беглый обзор внутренностей Python // slideshare
- **Сергей Лебедев** Лекции по языку Питон // youtube, канал "Computer Science Center" очень хороший курс
- **Learn X in Y minutes** <https://learnxinyminutes.com/docs/python/>
- **Роман Сузи** Язык программирования Python // НОУ Интуит
- <http://stackoverflow.com>
- **Jake VanderPlas** A Whirlwind Tour of Python, 2016 O'Reilly Media Inc. 98 р. для новичков
- **Лучано Рамальо** «Python. К вершинам мастерства» для профи
- **Дэн Бейдер** Чистый Python. Тонкости программирования для профи тонкости программирования

