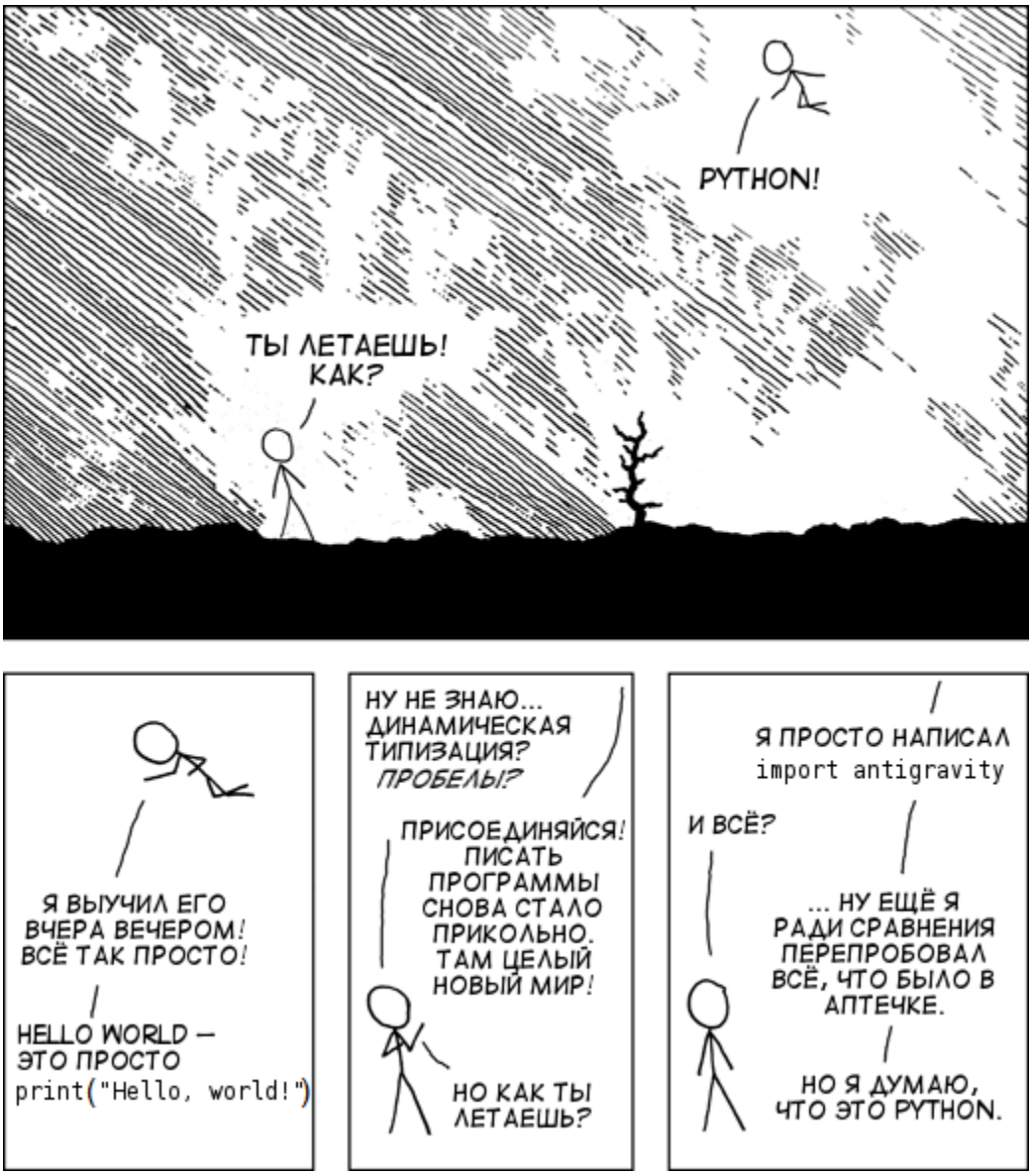
python_cheat_sheat

July 28, 2016

1 Python



2 1. Переменные

В питоне переменная объявляется в тот момент, когда ей присваивается значение. Присваивание происходит с помощью оператора =

```
In [13]: a = 5
         b = 7 + 12
```

Питон — язык с **динамической типизацией** (то есть типы переменных выводятся во время выполнения программы).

В частности, в питоне одна и та же переменная может менять свой тип.

```
In [14]: a = 12
         print(a)
         a = "hello world"
         print(a)
```

```
12
hello world
```

Название переменных в питоне может состоять из строчных или заглавных букв (не обязательно английских), цифр, знака подчеркивания (_)

```
In [24]: my_first_variable = 12
         НоваяПеременная = 34

         High_FIVE = 567
```

Название переменной не может начинаться с цифры.

```
In [25]: 123variable = 123
```

```
  File "<ipython-input-25-3aac93b48df0>", line 1
    123variable = 123
               ^
SyntaxError: invalid syntax
```

Название переменной так же не может быть ключевым словом:

```
In [75]: import keyword
         print('\n'.join(keyword.kwlist))
```

```
False
None
True
and
as
assert
break
class
continue
def
del
elif
```

```
else
except
finally
for
from
global
if
import
in
is
lambda
nonlocal
not
or
pass
raise
return
try
while
with
yield
```

(О том, что обозначает import и join — позже). На самом деле wing ide подсвечивает ключевые слова, так что вы не ошибетесь :)

```
In [76]: n
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-76-fe13119fb084> in <module>()
----> 1 n


NameError: name 'n' is not defined
```

Возникла ошибка, потому что переменной n не было присвоено значение, а значит, она не была объявлена

3 2. Типы данных

3.1 2.1 Тип int

Тип данных int в питоне служит для хранения целых чисел. Размер чисел ограничен только оперативной памятью.

```
In [17]: a = 12345
         b = -172634571256437612543712543765124735612435641273472165437126437l2
         c = 0xAF                  #шестнадцатеричная запись
         d = 0o123                 #восьмеричная запись
         e = 0b1001                #двоичная запись
```

```
print("a = ", a)
print("b = ", b)
print("c = ", c)
print("d = ", d)
print("e = ", e)
```

```
a =  12345
b =  -17263457125643761254371254376512473561243564127347216543712643712
c =  175
d =  83
e =  9
```

Рассмотрим некоторые способы создания переменных типа int

```
In [53]: a = 17
         b = int(17)
         c = int('17')
         d = int(17.1)
         e = round(17.9)


         print("a = ", a)
         print("b = ", b)
         print("c = ", c)
         print("d = ", d)
         print("e = ", e)
```

```
a =  17
b =  17
c =  17
d =  17
e =  18
```

Список основных арифметических действий, которые вам понадобятся:

- Сложение: +
- Вычитание: -
- Умножение: *
- Вещественное деление: /
- Целочисленное деление (округление производится вниз): //
- Взятие остатка: %
- Возведение в степень: **

```
In [71]: a = 123
         b = 56
         print(a + b)
         print(a - b)
         print(-b)
         print(a * b)
         print(a / b)
         print(a // b)
         print(a % b)
         print(a ** b)
         print(-b)
```

```
179
67
-56
6888
2.1964285714285716
2
11
10831441054800221700495871957152692828292650624493026036000669706814848574207410626414535094229213812948
-56
```

```
In [57]: print(-10 // 3)
         print(-10 % 3)
```

```
-4
2
```

Немного о том, как работают операторы // и %:

Из курса школьной математики, вам скорее всего известно, что для любых двух чисел x, y, где $y \neq 0$, существуют единственные числа $q, r : x = q * y + r$ и $0 \leqslant r < |y|$.

Так вот, q = x // y, r = x % y

Приоритеты операций (с самой приоритетной до самых малоприоритетных):

1. возведение в степень(**)
2. Унарные + и -
3. Умножение, деление, целочисленное деление, взятие остатка
4. Сложение и вычитание

Также можно использовать скобки

```
In [119]: print(10 * (3 + 2 ** 3) + 4 // 3)
          print(-2 ** 4)
          print((-2) ** 4)
```

```
111
-16
16
```

Также есть двойственные операторы: +=, -=, *= и прочие, которые изменяют значение переменной:

```
In [228]: a = 12
          a *= 10
          print(a)
```

```
120
```

** ++ и -- не вызывают ошибки (как два применения унарных оператора), но не дают тот эффект, что в C++ **

Для взятия модуля числа используется функция abs(x):

```
In [58]: y = -10
         x = abs(y)
         print(x)
```

```
10
```

3.2 2.2 Тип float

Служит для хранения дробных чисел.

```
In [43]: a = 1.2
         b = 4 / 3
         c = 0.1 + 0.1 + 0.1
         d = float("1.5")
         e = float(2)
         f = -23e3
         g = -23e-3

         print("a = ", a)
         print("b = ", b)
         print("c = ", c)
         print("d = ", d)
         print("e = ", e)
         print("f = ", f)
         print("g = ", g)
```

```
a =  1.2
b =  1.3333333333333333
c =  0.30000000000000004
d =  1.5
e =  2.0
f =  -23000.0
g =  -0.023
```

Обратите внимание, что, поскольку числа в компьютере хранятся в двоичной системе счисления, при работе с float может терятся точность вычислений!

Арифметические операции совпадают с арифметическими операциями для целых чисел

```
In [72]: a = 123.3
         b = 5.123
         print(a + b)
         print(a - b)
         print(-b)
         print(a * b)
         print(a / b)
         print(a // b)
         print(a % b)
         print(a ** b)
         print(-b)
```

```
128.423
118.17699999999999
-5.123
631.6659
24.0679289478821
24.0
0.34799999999999187
51523320187.66721
-5.123
```

В частности, извлечь корень из числа — это возвести его в степень 0.5:

```
In [66]: print(16 ** 0.5)
         print(17 ** 0.5)
```

```
4.0
4.123105625617661
```

3.3 2.3 Тип str

Тип str в питоне служит для работы со строками.

Одинарные и двойные кавычки не различаются.

Строки в тройных кавычках могут содержать переводы строк (длинные строки)

```
In [49]: s1 = 'string'
         s2 = "That's a string \n too!"
         s3 = str(25)
         s4 = ""               # пустая строка
         s5 = '''длинная
         строка'''

         print("s1 = ", s1)
         print("s2 = ", s2)
         print("s3 = ", s3)
         print("s4 = ", s4)
         print("s5 = ", s5)
```

```
s1 =  string
s2 =  That's a string
 too!
s3 =  25
s4 =
s5 =  длинная
строка
```

Символ выполняет особую роль: он позволяет вставить специальные символы

- \n – перевод строки
- " - двойная кавычка
- ' - одинарная кавычка
- \\ - обратный слеш

Чтобы узнать длину строки, используется функция len:

```
In [234]: s1 = "Привет, мир!"
          length1 = len(s1)
          length2 = len("Привет!")

          print(length1)
          print(length2)
```

```
12
7
```

Можно использовать ключевое слово **in** для проверки принадлежности подстроки строке

```
In [235]: print("aba" in "abacaba")
          print("d" in "abacaba")
```

```
True
False
```

Строки можно складывать (конкатенировать) и умножать на число:

```
In [79]: s1 = "Hello"
         s2 = ":)"
         print(s1 + s2)
         print(s1 + " " + s2)
         print(s2 * 10)
         print(10 * s2)
```

```
Hello:)
Hello :)
:):):):):):):):):):)
:):):):):):):):):):)
```

Также есть операторы += и *=

```
In [123]: a = "aaa"
          b = "bbb"
          c = 4
          b *= c
          print(b)
          c *= a
          print(c)
          a += b
          print(a)
```

```
bbbbbbbbbbbb
aaaaaaaaaaaa
aaabbbbbbbbbbbb
```

В питоне нет отдельного типа для символов, символ – это просто строки длины 1.

Для работы с символами есть функции ord, chr (узнать ascii-код символа и наоборот, получить сивол из ascii кода)

```
In [83]: print(ord('a'))
         print(chr(97))
```

```
97
a
```

Еще полезные методы, для работы с символами: проверка, является ли символ буквой/числом

```
In [90]: print("a".isalpha())
         print("A".isalpha())
         print("1".isdigit())
         print("_".isdigit())
         print("5".isalpha())
```

```
True
True
True
False
False
```

Эти методы возвращают значение типа bool, о котором мы поговорим немного позднее
Еще пара методов:

```
In [99]: print("aBBa".upper())
         print("ВеРнИтЕ 2007".lower())
```

```
ABBA
верните 2007
```

3.3.1 Индексация

В питоне можно обратиться к i-му символу строки с помощью оператора []:

```
In [101]: s = "My string"
          print(s[0])
          print(s[3])
          print(s[-1])
          print(s[-9])
          print(s[-0])
```

```
M
s
g
M
M
```

Индексация — с нуля.
Также, в питоне есть отрицательная индексация: -1 элемент - это первый с конца, -2 - второй с конца и так далее

```
+---+---+---+---+---+---+---+---+---+
| m | y |   | s | t | r | i | n | g |
+---+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7   8
 -9  -8  -7  -6  -5  -4  -3  -2  -1
```

Если обратиться к слишком большому (или слишком маленькому (отрицательному)) индексу, возникнет ошибка

```
In [105]: s = "abacaba"
          s[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-105-fcd8af702b70> in <module>()
  1 s = "abacaba"
----> 2 s[100]


IndexError: string index out of range
```

```
In [106]: s = "abacaba"
          s[-100]
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-106-36279e7bc44e> in <module>()
  1 s = "abacaba"
----> 2 s[-100]


IndexError: string index out of range
```

3.3.2 Срезы

Чтобы брать подстроки (или некоторые специальные подпоследовательности) в питоне есть срезы:

- s[a:b] – взять подстроку с индекса a включительно по индекс b невключительно (b может быть больше длины строки)
- s[a:] – взять подстроку с индекса a включительно и до конца строки
- s[:b] – взять подстроку до индекса b невключительно
- s[a:b:c] – взять подпоследовательность, начиная с a, с шагом c не доходя до b (c может быть отрицательным)

```
In [109]: s = "my string"
          print("s[3:8] =", s[3:8])
          print("s[5:] =", s[5:])
          print("s[:5] =", s[:5])
          print("s[0:5] =", s[0:5])
          print("s[-8:-3] =", s[-8:-3])
          print("s[1:-3] =", s[1:-3])
          print("s[:-1] =", s[:-1])
          print("s[:] =", s[:])
          print("s[2:1] =", s[2:1])
```

```
s[3:8] = strin
s[5:] = ring
s[:5] = my st
s[0:5] = my st
s[-8:-3] = y str
s[1:-3] = y str
s[:-1] = my strin
s[:] = my string
s[2:1] =
```

```
In [112]: s = "my string"
          print("s[3:8:2] =", s[3:8:2])
          print("s[3:100:2] =", s[3:100:2])
          print("s[3:8:1] =", s[3:8:1])
          print("s[8:3:-1] =", s[8:3:-1])
          print("s[8:3:-2] =", s[8:3:-2])
          print("s[::2] =", s[::2])
          print("s[::-1] =", s[::-1])
```

```
s[3:8:2] = srn
s[3:100:2] = srn
```

```
s[3:8:1] = strin
s[8:3:-1] = gnirt
s[8:3:-2] = git
s[::2] = m tig
s[::-1] = gnirts ym
```

```
In [111]: print("s[3:8:0] = ", s[3:8:0])
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-111-9160819a1c4d> in <module>()
    ----> 1 print("s[3:8:0] = ", s[3:8:0])


        ValueError: slice step cannot be zero
```

3.3.3 Некоторые методы обработки строк

- s.find(sample, [start, [end]]) — возвращает индекс начала первого вхождения sample или -1 (rfind – последнее вхождение)

```
In [116]: s = "mabacabadabac"
          print(s.find("aba"))
          print(s.find("aba", 5)) #начиная с 5 символа
          print(s.find("aba", 2, 6))
          print(s.rfind("aba"))
```

```
1
5
-1
9
```

- s.replace(from, to) — возвращает строку, в которой все непересекающиеся вхожения from заменены на to (сама строка не меняется)

```
In [124]: s = "abababacaba"
          print(s.replace("aba", "_"))
          print(s)
```

```
_bac_
abababacaba
```

- s.count(sample) — возвращает количество непересекающихся вхождений sample

```
In [118]: s = "abababacaba"
          print(s.count("aba"))
```

```
2
```

- s.strip([chars]) — возвращает строку, у которой удалены символы chars из начала и конца строки. Если chars не задан, то будут удалятся пробельные символы (пробелы, табуляция, переносы строк). Сама строка s не изменяется.

lstrip, rstrip – делают то же самое, но с либо только слева, либо только справа

```
In [133]: s = "   abacaba\t\n"
          print(s.strip())
          print(s)

          t = "abacabadaba"
          print(t.rstrip("abd"))
```

```
abacaba
   abacaba

abac
```

3.3.4 Строки — неизменяемые объекты!

Нельзя	Можно
s[len(s)] = 'a'	s = s + 'a'
s[0] = 'a'	s = 'a' + s[1:]
s[3] = 'a'	s = s[3:] + 'a' + s[4:]
удалить символ	s = s[3:] + s[4:]
вставить символ	s = s[3:] + 'a' + s[3:]
s.replace('a', 'b')	s = s.replace('a', 'b')
перевернуть строку	s = s[::-1]

3.4 2.4 Тип tuple

Тип tuple служит для представления неизменяемого (immutable) массива

- Операции — как со строками (индексы, срезы)
- Хранит любые объекты (на самом деле ссылки на объекты), в том числе и изменяемые

Создание:

```
In [232]: a = (40, ) # Если элемент 1, запятая в конце обязательна!
          print(a)
          a = tuple("abc")
          print(a)
          a = tuple()
          print(a)
          a = (1, 2, 'q', (3, 5))
          print(a)
          print(a[3][1])
          print(len(a))
          print('q' in a)
```

```
(40,)
('a', 'b', 'c')
()
(1, 2, 'q', (3, 5))
5
4
True
```

```
In [155]: a = (1, 2, 'q', (3, 5))
          a[0] = 2


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-155-ea50b69c9bfc> in <module>()
          1 a = (1, 2, 'q', (3, 5))
    ----> 2 a[0] = 2


        TypeError: 'tuple' object does not support item assignment


In [159]: a = (1, 2, 'q', (3, 5))
          print(a[1:3])
          print(a * 10)
          print(a + a)

(2, 'q')
(1, 2, 'q', (3, 5), 1, 2, 'q', (3, 5), 1, 2, 'q', (3, 5), 1, 2, 'q', (3, 5), 1, 2, 'q', (3, 5), 1, 2, '
(1, 2, 'q', (3, 5), 1, 2, 'q', (3, 5))
```

Есть некоторые методы, как у строк:

```
In [141]: a.count('q')

Out[141]: 1
```

3.5 2.5 Тип list

Тип list используется для представления изменяемых (mutable) массивов

Операции — как с tuple и str:

```
In [233]: s = [1, 2, 3, 'abc']
          print(s)
          s = list("abc")
          print(s)
          s = []
          print(s)
          s = list()
          print(s)
          s1 = [1, 2, 3]
          s2 = ["a", 'b', "c"]
          print(s1 + s2)
          print(s1 * 3)
          print(len(s1))

[1, 2, 3, 'abc']
['a', 'b', 'c']
[]
[]
[1, 2, 3, 'a', 'b', 'c']
[1, 2, 3, 1, 2, 3, 1, 2, 3]
3
```

```
In [163]: s = [1, 2, 3, 'abc', (2, 3), [4, 5, 6]]
          print(s[0])
          print(s[-4])
          print(s[:5])
          print(s[-1][0])

1
3
[1, 2, 3, 'abc', (2, 3)]
4
```

Многомерные массивы — списки списков
Есть методы для изменения списка:

```
In [203]: s = [1, 2, 3, 4, 5, 6, 7, 8]
          print(s)

[1, 2, 3, 4, 5, 6, 7, 8]

In [204]: s[0] = 0
          print(s)

[0, 2, 3, 4, 5, 6, 7, 8]

In [205]: s.append(5)
          print(s)

[0, 2, 3, 4, 5, 6, 7, 8, 5]

In [206]: s.insert(1, "aba")
          print(s)

[0, 'aba', 2, 3, 4, 5, 6, 7, 8, 5]

In [207]: del s[5] # удаление по индексу
          print(s)

[0, 'aba', 2, 3, 4, 6, 7, 8, 5]

In [208]: s.remove(5) # удаление по значению
          print(s)

[0, 'aba', 2, 3, 4, 6, 7, 8]

In [209]: del s[2:7]
          print(s)

[0, 'aba', 8]

In [210]: s.extend([1, 2, 3])
          print(s)

[0, 'aba', 8, 1, 2, 3]

In [211]: print(s.pop())
          print(s)

3
[0, 'aba', 8, 1, 2]
```

```
In [212]: s = [1, 2, 3, 4, 5]
          s[1:4] = [5]
          print(s)
```

```
[1, 5, 5]
```

```
In [213]: s[::2] = [1]
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-213-dec715ed2611> in <module>()
    ----> 1 s[::2] = [1]


        ValueError: attempt to assign sequence of size 1 to extended slice of size 2
```

```
In [214]: s[::2] = [1, 2]
```

```
In [217]: print(s)
```

```
[1, 5, 2]
```

```
In [218]: s.reverse()
          print(s)
```

```
[2, 5, 1]
```

Сортировка: s.sort() – сортирует список по возрастанию s.sort(reverse = True) — по убыванию sorted(s) — возвращает новый список, не изменяя старый

```
In [216]: s1 = [7, 1, 5, 4, 2]
          print(sorted(s1, reverse=True))
          print(s1)
          s1.sort()
          print(s1)
```

```
[7, 5, 4, 2, 1]
[7, 1, 5, 4, 2]
[1, 2, 4, 5, 7]
```

Функции, связывающие списки и строки

- sep.join(list_of_strings) — Склеивает все строки из списка в одну, разделяя их sep

```
In [222]: print(" ".join(["first", "second", "third"]))
          print("a".join(("bbb", "ccc"))) # не обязательно список, можно и tuple
          print("x".join("abc")) # или даже строку
```

```
first second third
bbbaccc
axbxc
```

- s.split([sep]) — Разрезает строку на части по строке sep (без параметров – по пробельным символам, считая несколько пробельных символов подряд за один разделитель

```
In [224]: s = "a    v    kasjdh   kasjd, 987\n2398"
          print(s.split())
          print(s.split(" "))
          print(s.split("9"))
```

```
['a', 'v', 'kasjdh', 'kasjd,', '987', '2398']
['a', '', '', '', 'v', '', '', '', 'kasjdh', '', '', 'kasjd,', '987\n2398']
['a    v    kasjdh   kasjd, ', '87\n23', '8']
```

map map(func, list) — вернуть специальный объект, который можно превратить в список или tuple, к каждому элементу которого применена функция func

(сам список не меняется, на месте списка может быть tuple, str)

```
In [230]: s = ["1", "123", "3"]
          print(map(int, s))
          print(list(map(int, s)))
          print(s)
          t = ("  aba  ", " b\n", "c ab")
          print(list(map(str.strip, t)))
```

```
<map object at 0x0000000004A10198>
[1, 123, 3]
['1', '123', '3']
['aba', 'b', 'c ab']
```

Как видно из примера выше, вместо методов можно вызывать функции
имя_типа.имя_метода(к_чему_применяем, остальные аргументы)

```
In [240]: s = "abacaba"
          print(str.count(s, "a"))
```

```
4
```

3.5.1 Еще несколько полезных функций для list и tuple:

- min
- max
- sum

```
In [241]: s = [1, 2, 3]
          print(sum(s))
          print(min(s))
          print(max(s))
          print(min(7, 2, 3, 4, 5))
```

```
6
1
3
2
```

3.5.2 Ссылки и =

Давайте немного подробнее поговорим о присваивании списков, с этим в Python есть несколько сложностей. Python интерпретируемый язык. Во время выполнения он должен хранить все данные которые у вас есть в программе, поэтому у интерпретатора во время выполнения хранится множество существующих объектов, и переменная это всего лишь ссылка на объект, который хранит интепретатор.

Давайте чуть-чуть подробнее изучим, как устроен лист с учетом ссылочного устройства.

```
In [238]: a = [1, 2, 3, 4] # a - ссылка на лист, каждый элемент листа это ссылка на элементы 1, 2, 3, 4
          b = a  # b - ссылка теперь на тот же лист
          a[0] = -1
          print("a =", a)
          print("b =", b)

a = [-1, 2, 3, 4]
b = [-1, 2, 3, 4]
```

a и b по прежнему указывают на один объект!

```
In [239]: a = [1, 2, 3]
          b = [1, 2, 3]
          a[0] = 0
          print(b)

[1, 2, 3]
```

В данном примере, это были два разных списка, с одинаковыми значениями
Более нетривиальный пример:

```
In [242]: a = (1, 2, [3, 4])
          a[-1][0] = 1
          print(a)

(1, 2, [1, 4])
```

Как упоминалось выше, list и tuple хранят ссылки на объекты, а tuple может хранить ссылки на изменяемые объекты. Таким образом, видимое "значение" кортежа изменилось (хотя значение не изменилось)
[КАРТИНОЧКА!]

```
In [244]: a = (1, 2, [3, 4])
          b = a
          a[-1][0] = 1
          print(b)

(1, 2, [1, 4])
```

Как же скопировать список, а не просто ссылку на него?
Самая короткая запись:

```
In [245]: a = [1, 2, 3]
          b = a[:]
          a[0] = 2
          print(a)
          print(b)

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

Здесь мы взяли срез a. Каждый раз, когда мы берем срез, создается новый объект, поэтому b указывает на другое место.
Однако проблема со вложенными списками остается:

```
In [246]: a = [[1, 2], [3]]
          b = a[:]
          a[0] = [5]
          a[1][0] = 1
          print(a)
          print(b)
```

```
[[5], [1]]
[[1, 2], [1]]
```

[КАРТИНОЧКА]
В следуещем разделе мы решим эту проблему.

3.5.3 2.6 Тип bool

Тип bool нужен для представления логического (булева) типа. Переменная типа bool может принимать лишь два значения: True или False (оба являются ключевыми словами, пишется с большой буквы)

```
In [253]: a = True
          print(a)
          b = False
          print(b)
```

```
True
False
```

Несколько слов о преобразовании других типов к bool:

- Пустой список (строка, кортеж) – False, все остальные списки, кортежи, строки — True

```
In [252]: a = bool([])
          b = bool([False])
          print(a)
          print(b)
```

```
False
True
```

- 0 и 0.0 — False, все остальные int и float — True

```
In [256]: a = bool(0.0)
          b = bool(100)
          print(a)
          print(b)
```

```
False
True
```

3.5.4 Операции над типом bool

- Логическое или: or
- Логическое и: and
- Логическое отрицание: not

3.5.5 Логические операции над числами:

- Равенство: ==
- Неравенство: !=
- Меньше: <
- Меньше либо равно: <=
- Больше: >
- Больше либо равно: >=

Про and и or: and и or не вычисляют второй аргумент, если все и так понятно (если первый аргумент and ложь или первый аргумент or — истина)

and и or возвращают свой значимый (последний вычисленный) аргумент, **не приводя его к bool**.

```
In [1]: print(5 > 2)
        print(not 5 > 2)
        a = 5
        b = 2
        print(a > b and a >= b)
        print(True and 5)
        print(True or 1 // 0)  # нет ошибки
```

```
True
False
True
5
True
```

Также можно использовать сложные сравнения:

```
In [4]: print(3 < 4 <= 5)
```

```
True
```

4 3. Модули

Иногда нам хочется использовать какие-то дополнительные функции, которые по умолчанию не встроены в интерпретатор. Для этого люди пишут программы, называемые библиотеками (модулями) , которые содержат часто используемые функции, не встроенные в интерпретатор.

В питоне есть некоторое количество встроеных модулей.

4.1 3.1 Модуль copy

Рассмотрим первый из них, который поможет нам решить проблему с копированием вложеных списков. Этот модуль — copy

Чтобы подключить модуль используется ключевое слово import

```
In [247]: import copy
```

Теперь к функциям из этого модуля можно обращаться как имя_модуля.имя_функции

Например, в модуле copy есть функция copy, которая эквивалентна взятию среза [:]

```
In [248]: import copy

          a = [[1, 2], [3]]
          b = copy.copy(a)
```

```
          a[0] = [5]
          a[1][0] = 1
          print(a)
          print(b)
```

```
[[5], [1]]
[[1, 2], [1]]
```

Также в модуле есть функция deepcopy, которая копирует список (кортеж) вместе со всеми вложенными списками, кортежами:

```
In [261]: import copy
          a = [[[1, 2, 3]]]
          b = copy.deepcopy(a)
          a[0][0][0] = 0
          print(a)
          print(b)
```

```
[[[0, 2, 3]]]
[[[1, 2, 3]]]
```

Также можно импортировать не модуль целиком, а отдельные функции из него с помощью конструкции

from имя_модуля import имя_функции [as новое_имя_фунции]

Если новое имя функции не задано, то считается, что новое имя функции совпадает со старым.

После такой инструкции к функции можно обращаться как просто новое_имя_фукнции:

```
In [264]: from copy import deepcopy as super_copy
          a = [[[1, 2, 3]]]
          b = super_copy(a)
          a[0] = 0
          print(a)
          print(b)
```

```
[0]
[[[1, 2, 3]]]
```

Можно импортировать сразу несколько функций:

```
In [265]: from copy import deepcopy as super_copy, copy as lame_copy
          from copy import deepcopy, copy
```

4.2 3.2 Модули __hello__, this

После вызова import могут не только импортироваться функции, но и происходить какие-то действия (хотя вы, скорее всего, с этим не столкнетесь)

```
In [266]: import __hello__
```

```
Hello world!
```

```
In [267]: 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!
```

(Можете почитать на досуге)

4.3 3.3 Модуль math

В модуле math есть следующие функции и константы:

- floor(x) – округление вниз
- ceil(x) – округление вверх
- trunc(x) – отбрасывает дробную часть (эквивалентно floor для положиельных, ceil для отрицательных)
- sqrt(x) – квадратный корень
- pi – чило pi (3.1415926...)
- e – число e (2.718281828...)
- log(x, base=math.e) – возврает логарифм x по основанию base (e по умелчанию)
- log10(x) – то же самое, что и log(x, 10)
- тригонометрия — sin, cos, asin, acos, tan, atan
- exp(x) — то же самое, что и math.e ** x
- factorial(n) — факториал

```
In [269]: from math import * # это импортирует все из модуля, так делать не стоит
          print(floor(-10.3))
          print(trunc(-10.3))
          print(sin(-10.3))
          print(factorial(10))
```

```
-11
-10
0.7676858097635825
3628800
```

5 4. Ввод и вывод

5.1 4.1 Вывод

Для вывода можно использовать функцию print (которая уже не раз использовалась выше)
print(arg1, arg2, arg3, ..., argn, sep='', end='')

Выводит последовательнос arg1, arg2, . . . , argn разделяя их строкой sep (пробел по умолчанию) и в конце печатая строку end (перевод строки по умолчанию)

```
In [270]: print(1, "string", [1, 2, 'string in list'], sep='SEP_STRING', end='END_STRING')
```

```
1SEP_STRINGstringSEP_STRING[1, 2, 'string in list']END_STRING
```

5.2 4.2 Ввод

Для ввода данных с клавиатуры можно использовать функцию input([message]) (если задано message, то сначала выведется оно без перевода строки и пробела)

Функция возвращает строку (str) введенную с клавиатуры

```
In [273]: a = input()
          print(a)
```

```
3 2
3 2
```

Дальше с этой строчкой можно работать уже известными нам функциями и методами.

Например, если нам нужно считать два числа на разных строках:

```
In [ ]: a = int(input())
        b = int(input())
        print(a + b)
```

Если же нам нужно список строк, разделенных пробельными символами:

```
In [ ]: a = input().split()
        print(a)
```

Если нам нужно считать список чисел, можно использовать изветсную конструнцию list(map(. . .)):

```
In [ ]: a = list(map(int, input().split()))
        print(sum(a))
```

6 5 Управляющие конструкции:

6.1 5.1 Операторы if..elif..else

Суть, пожалуй, ясна, нескольно слов по синтаксису:

```
if условие1:
    действие в случе выполнения условия
elif условие2:
    действие в случае невыполнения условия1 и выполнения условия2
elif условие3:
    ....
else:
    действие, если ни одно из условий не выполнено
```

Блоки elif и else можно опустить

После условия в if и elif или после else ставится двоеточие :. Это часть синтаксиса python. Все, что будет выполняться после if должно идти с отступом 4 пробела (на самом деле тут есть вольности, но мы их опустим). относительно предыдущей строки. Аналогично с else и elif.

[НАСТРОИТЬ ПЕРЕВОД ТАБОВ В ПРОБЕЛЫ]

Условия в if и elif можно читать, как будто сверху навешено bool

```
In [5]: if True:
            print("True")
        else:
            print("False")
```

```
True
```

```
In [6]: if (True):
            if (1 == 0):
                print("1 is equal to 0")
            elif (1 == 2):
                print("1 is equal to 2")
            else:
                print("something went wrong")
```

```
something went wrong
```

```
In [7]: if (1 == 1 and (not "aba" in "dab")):
            print("OK")
```

```
OK
```

По отступам очень удобно отслеживать, какой else к какому if относится. Если не ставить двоеточие или отступы, то программа будет выдавать ошибку и не запустится.

```
In [8]: if 6 > 5
            print("Пропущено двоеточие!")
```

```
  File "<ipython-input-8-615a4bd4ed19>", line 1
    if 6 > 5
           ^
SyntaxError: invalid syntax
```

```
In [9]: if 6 > 5:
        print("Пропущен отступ!")
```

```
  File "<ipython-input-9-641a99981a4e>", line 2
    print("Пропущен отступ!")
        ^
IndentationError: expected an indented block
```

Некоторые среды программирования, такие как, Wing Ide, в котором мы будем работать, умеют автоматически расставлять отступы там, где это необходимо.

Заметим, что поскольку питон – язык интерпретируемый, то некоторые ошибки могут быть не вызваны

```
In [15]: x = 1
         if (x == 1):
             a = 0
             print(a)
         else:
             print(a + 1) # ошибка -- a не инициализировано, но мы этого не заметим
```

```
0
```

это может быть опасно, например, в следующем коде:

```
In [16]: #x = input()
         #if (len(x) != 0):
         #    a = 5
         #    print(a)
         #else:
         #    print(a + 1)
```

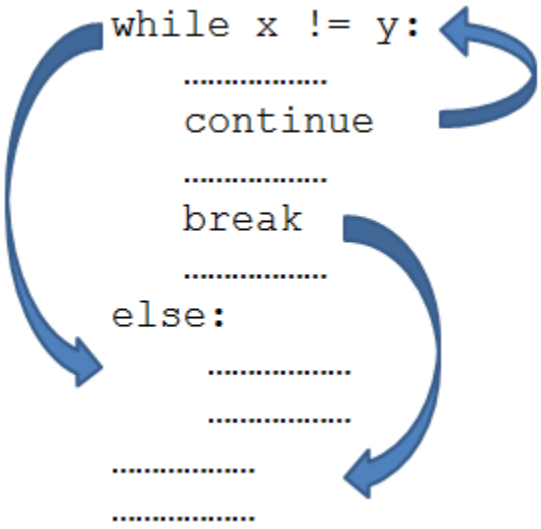
6.2 5.2 while

Очень часто случается, что нам надо повторить некоторые действия до выполнения некоторого условия. Самый просто устроенный цикл - while. Он работает точно так же как и в остальных языках программирования.

```
In [12]: i = 0
         while i < 5:
             i += 1
             print(i)
```

```
1
2
3
4
5
```

Естественно, после while может стоять любое условие. Также есть операторы break и continue, break заканчивает цикл, continue переходит в начало цикла:



Также после блока while можно написать блок else: он будет выполнен, если не был вызван break

```
In [23]: x = 1
         y = 2
         while (x != y):
```

```
            y -= 1
            break
        else:
            print("ELSE")
```

```
In [25]: x = 1
         y = 2
         while (x != y):
             x += 1
             if (y == 1):
                 break
         else:
             print("ELSE")
```

```
ELSE
```

Для примера давайте напишем программу, которая выводит наименьшую степень двойки, которая больше заданного числа n.

```
In [27]: #n = int(input())

         #current_power = 0
         #current_number
         #while n >= current_number:
         #    current_number *= 2
         #    current_power += 1
         #print(current_power)
```

6.3 5.3 for

В языке питон можно итерироваться (проходится) по объектам (не по всем. Те, по которым можно, называются итерирабельными (iterable)), в частности, по спискам, строкам, кортежам. Итерирование происходит следующей конструкцией:

```
for item in items:

    do something
```

```
In [29]: a = [1, 2, "string", (12, 13)]

         for i in a:
             print(i)
```

```
1
2
string
(12, 13)
```

Заметим, что если внутри цикла

```
for item in items:
    ....
```

Написать item = ... то в items ничего не изменится:

```
In [33]: a = [1, 2, 3, 4]
         for i in a:
             i = 0
         print(a)
```

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

Однако:

```
In [36]: a = [[1], [2], [3]]
         for i in a:
             i[0] = 0
         print(a)
```

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

Почему так происходит? (КАРТИНОЧКА)

6.3.1 5.3.1 range

Как же пройтись по числам от 1 до n?

Можно, например, использовать цикл while:

```
In [38]: i = 1
         n = 10
         while (i <= n):
             print(i)
             i += 1
```

```
1
2
3
4
5
6
7
8
9
10
```

Но это занимает много места, а про i += 1 в конце можно забыть

Для таких функций в питоне есть объект range:

range(a, b, c) — 'список' чисел от a включительно, до b не включительно с шагом n

(первый и последний параметры можно опускать: первый по умолчанию 0, последний – 1)

"список" взято в кавычки, потому что этот объект не является списком и не занимает много места в памяти, но однако по нему можно итерироваться:

```
In [39]: n = 10
         for i in range(1, n + 1):
             print(i, end=' ')
         print()
```

```
1 2 3 4 5 6 7 8 9 10
```

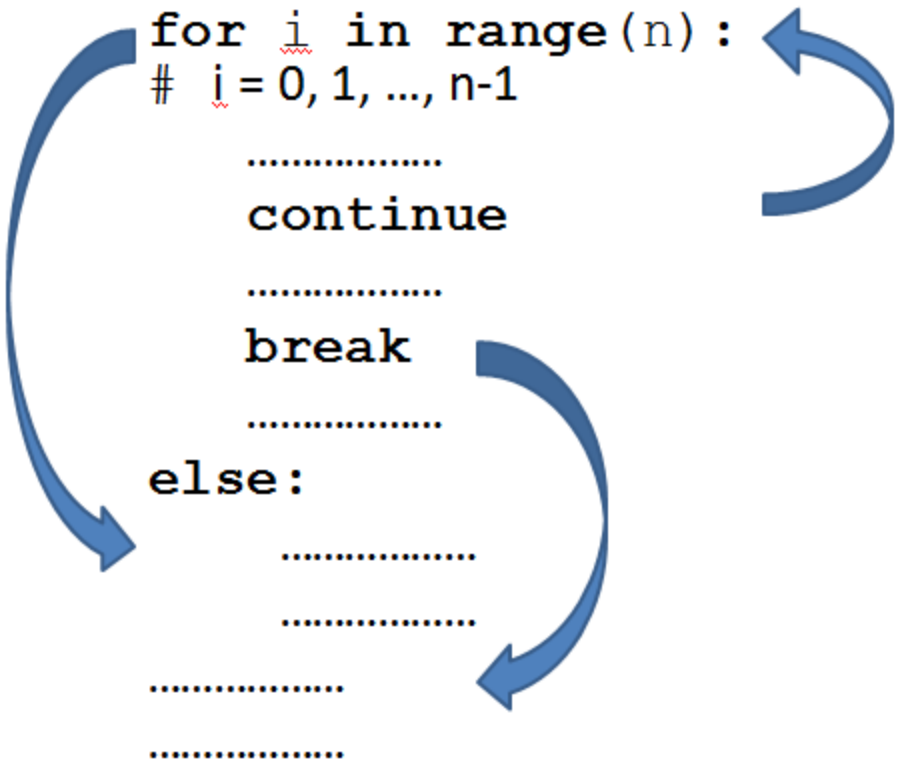
Еще пример:

```
In [43]: a = [0] * 3

         for i in range(len(a)):
             a[i] = i + 1
         print(a)
```

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

У for также есть break, continue и else:


for i in range(n):
# i = 0, 1, ..., n-1
.....
continue
.....
break
.....
else:
.....
.....
.....
.....

7 6. Функции

В питоне, как и в любом другом нормальном языке, есть функции.

Чтобы объявить функцию используется ключевое слово def:

```
def function_name(arg1, arg2, ..., argn):
    ...#something
    return blahblah
```

После def идет название функции, открывающая скобка, список аргументов, закрывающая скобка, двоеточие.

Чтобы вернуть что-то из функции используется return. return необязателен.

Все, что будет выполняться в функции должно идти с отступом 4 пробела относительно предыдущей строки.

```
In [45]: def average(number_list):
             if (len(number_list) != 0):
                 return sum(number_list) / len(number_list)

In [47]: average([1, 2, 3])

Out[47]: 2.0

In [48]: average((3, 3, 4))

Out[48]: 3.3333333333333335

In [49]: average(["a", "b", "c"])


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-49-a13945a7c1ad> in <module>()
     ----> 1 average(["a", "b", "c"])


         <ipython-input-45-3df0f95ac678> in average(number_list)
           1 def average(number_list):
           2     if (len(number_list) != 0):
     ----> 3         return sum(number_list) / len(number_list)


         TypeError: unsupported operand type(s) for +: 'int' and 'str'


In [51]: print(average([]))

None
```

None – специальный объект в питоне. Используется, например, как возвращаемое значение функции если мы ничего не возвращаем

```
In [54]: def print_hello(name):
             print("Hello,", name)

         print(print_hello("Ivan"))

Hello, Ivan
None
```

7.0.2 Аргументы

Я думаю, вы уже догадываетесь, что происходит в следующих примерах:

```
In [56]: def make_zero(x):
             x = 0
             print("Inside func:", x)

         def make_zeroes(x_list):
             for i in range(len(x_list)):
                 x_list[i] = 0
         x = 5
         make_zero(x)
         print(x)

         x_list = [1, 2, 3]
         make_zeroes(x_list)
         print(x_list)
```

```
Inside func: 0
5
[0, 0, 0]
```

7.0.3 Глобальные и локальные переменные

Внутри функции можно использовать переменные, объявленные вне этой функции

```
In [74]: def f():
             print(a)
         a = 1
         f()
```

```
1
```

Если инициализировать какую-то переменную внутри функции, использовать эту переменную вне функции не удастся. Например:

```
In [69]: def f():
             ac = 1
         f()
         print(ac)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-69-2c0b59e2953d> in <module>()
  2     ac = 1
  3 f()
----> 4 print(ac)


NameError: name 'ac' is not defined
```

```
In [70]: a = 0
         def f():
             a = 1
         f()
         print(a)
```

```
0
```

Интерпретатор Питон считает переменную локальной, если внутри нее есть хотя бы одна инструкция, модифицирующая значение переменной (это может быть оператор =, += и т.д., или использование этой переменной в качестве параметра цикла for, то эта переменная считается локальной и не может быть использована до инициализации. При этом даже если инструкция, модицифицирующая переменную никогда не будет выполнена: интерпретатор это проверить не может, и переменная все равно считается локальной

```
In [76]: def f():
             print (a)
             if False:
                 a = 0
         a = 1
         f()
```

```
         ---------------------------------------------------------------------------
         UnboundLocalError                         Traceback (most recent call last)

         <ipython-input-76-cc82d49feb0a> in <module>()
           4         a = 0
           5 a = 1
     ----> 6 f()


         <ipython-input-76-cc82d49feb0a> in f()
           1 def f():
     ----> 2     print (a)
           3     if False:
           4         a = 0
           5 a = 1


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

Однако:

```
In [77]: def f():
             a[0] = 1
             print(a)
         a = [0, 0, 0]
         f()
         print(a)
```

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

Чтобы функция могла изменить значение глобальной переменной, необходимо объявить эту переменную внутри функции, как глобальную, при помощи ключевого слова global:

```
In [78]: def f():
             global a
             a = 1
             print (a)
         a = 0
         f()
         print(a)
```

```
1
1
```

8 7. Параллельное присваивание

Конструкции вида

```
a1, a2, ..., an = b1, b2, ..., bn
```

Называются параллельным присваиванием
Также, можно писать
a1, a2, . . . , an = list_of_n_elements

8.0.4 swap

Чтобы поменять значение переменных a и b можно воспользоваться параллельным присваиванием:
a, b = b, a

```
In [80]: a, b = 1, 2
         print(a, b)
```

```
1 2
```

```
In [82]: a, b = 3, 5 / 2
         print(a, b)
```

```
3 2.5
```

```
In [83]: a, b = 1, 2
         a, b = b, a
         print(a, b)
```

```
2 1
```

9 8. Правила оформления кода (Codestyle)

[КАРТИНОЧКИ]

9.1 PEP-8

10 9. Работа с файлами

Для работы с файлами понадобятся следующие функции:

- some_file = open(filename, mode) — открыть файл с именем filename в режиме mode: [ДОКУМЕНТАЦИЯ]
- some_file.close() — закрыть файл (в спортивном программировании необязательно, в целом – надо)
- s = some_file.readline() — считать строку из файла (оставляет)
- list_of_strings = some_file.readlines() — считать строки из файла в список
- s = some_file.read() — считать весь файл в строку
- some_file.write(string) – записать строку (строго типа str!) в файл
- print(. . . , file=some_file) — запись в файл
- for line in some_file: — прохож по строкам файла

Для олимпиадных задач вам скорее всего будет достаточно следкющей конструкции:

11 10. Бонусы

Немного удобных функция для работы с системами счисления:

```
In [1]: bin(123) #превращает число в строку, содержащую двоичное представление

Out[1]: '0b1111011'
```

Аналогично для восьмеричной и шестнадцатеричной системы счисления:

```
In [2]: hex(8123)

Out[2]: '0x1fbb'

In [3]: oct(6523)

Out[3]: '0o14573'

In [4]: int("100010", 2) #перевод из 2-ичной системы счисления числа "100010"

Out[4]: 34

In [5]: int("12", 5) #перевод из 5-ичной системы счисления числа "12"

Out[5]: 7
```

Полезные модули (читайте документацию):

- decimal — более точные дробные числа
- fractions — дроби

```
In [ ]:
```