

# **Практикум по программированию на языке Python**

## **Занятие 2: Модель памяти, встроенные типы данных**

**Мурат Апишев (mel-lain@yandex.ru)**

**Москва, 2019**

## Объекты и переменные в Python

- Переменные  $\neq$  объекты
- Объект - это сущность, созданная и используемая в коде
- Объектами являются числа, строки, контейнеры, классы, функции и т.п.
- Переменная - это именованная ссылка на объект
- У одного объекта может быть много ссылок-переменных
- Объекты являются строго типизированными, ссылки - нет

## Стадии жизни объекта

- Создание объекта приводит к выделению фрагмента памяти
- У каждого объекта есть счётчик ссылок
- Каждое новая ссылающаяся переменная увеличивает этот счётчик на единицу
- Ссылку можно удалить с помощью оператора `del`
- Когда счётчик ссылок на объект становится равным 0, объект удаляется
- Удаление и создание объектов может не приводит к изменениям памяти в некоторых случаях из-за кэширования (например 0 или None)
- Но в общем случае память выделяется и освобождается

## Сборщик мусора

- Освобождение памяти производит сборщик мусора (Garbage Collector)
- Памяти освобождается не моментально, а в некоторый недетерминированный момент
- Одной из сложностей поиска объектов для удаления являются т.н. циклические ссылки

```
In [3]: a = [None]
a[0] = a
del a
```

- Сборщик мусора умеет работать с такими случаями

## Полезные функции

```
In [5]: import gc
gc.collect()  # call collection immediately
```

```
In [23]: import sys
a = 1
print(sys.getrefcount(a))  # cached popular value
print(sys.getrefcount('some str'))  # 1 after creation, 1 temporary, 1 - ???
```

```
2331
3
```

## Изменяемые переменные

- Изменяемые переменные модифицируются на месте
- Создание новых ссылок на изменяемые переменные не создаёт новых объектов

```
In [24]: a = [2]
         b = a
         a += [1]
         print(b)
```

```
[2, 1]
```

- можно создавать объекты, имеющие ссылки на другие объекты

```
In [25]: a.append([1, 2, 3])
         a
```

```
Out[25]: [2, 1, [1, 2, 3]]
```

## Копирование изменяемых объектов

```
In [26]: a = [1, 2, 3]
         b = a[:]
         b.append(4)
         print(a)
```

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

```
In [27]: a = [1, 2, 3]
         b = a.copy()
         b.append(4)
         print(a)
```

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

## Копирование вложенных изменяемых объектов

```
In [32]: a = [['a', 'b', 'c'], 2, 3]
b = a.copy()
b[0].append('d')
print(a)
print(b)
```

```
 [['a', 'b', 'c', 'd'], 2, 3]
 [['a', 'b', 'c', 'd'], 2, 3]
```

```
In [31]: import copy

a = [['a', 'b', 'c'], 2, 3]
b = copy.deepcopy(a)
b[0].append('d')
print(a)
print(b)
```

```
 [['a', 'b', 'c'], 2, 3]
 [['a', 'b', 'c', 'd'], 2, 3]
```



## Слабые ссылки

- Иногда возникает необходимость вести учёт объектов только когда они используются где-нибудь ещё
- Но само слежение уже создаёт ссылку, из-за которой объект будет оставаться в памяти
- Слабые ссылки дают возможность вести учёт объекта без создания ссылок на них
- Для этого используется модуль `weakref`
- Классическое приложение - кэширование объектов, пересоздание которых является затратным
- Слабые ссылки нельзя создавать на объекты базовых типов

## Слабые ссылки

```
In [60]: # http://old.pynsk.ru/posts/2015/Sep/19/tainstvo-standartnoi-biblioteki-slabye-ssylki-weakref/#.XkhkS9kueV4

import weakref, gc

class A:
    def __init__(self, value):
        self.value = value
    def __repr__(self): # magic method: information for print
        return str(self.value)

a = A(10) # create object and strong reference
d = weakref.WeakValueDictionary() # dict with weak references

d['k'] = a
d['k']
```

Out[60]: 10

```
In [61]: del a
```

```
In [62]: gc.collect()
d['k']
```

Out[62]: 10

- при запуске в интерпретаторе через командную строку получается вывод "KeyError: 'k'"
- по всей видимости, дело в том, что в ноутбуке создаётся какая-то лишняя ссылка

## Идентификатор объекта

- у каждого объекта в Python есть целочисленный идентификатор
- на время жизни объекта он является уникальным
- получить идентификатор объекта можно с помощью функции `id`

In [31]:

```
a = 10  
b = 12  
print(id(a), id(b))
```

93923529852224 93923529852288

- идентификатор зависит от реализации, это может быть адрес в памяти
- у одинаковых константных объектов этот идентификатор всегда одинаковый

In [32]:

```
a, b = 10, 10  
c, d = 12, 12  
print(id(a), id(b), '\n')  
print(id(c), id(d))
```

93923529852224 93923529852224

93923529852288 93923529852288

## Идентификатор константного объекта

- таким образом, одинаковые константы хранятся в единственном экземпляре
- Python делает так для экономии памяти
- типы в Python тоже являются объектами

```
In [33]: print(id(type(12)), id(type(5)), id(int), '\n')  
         print(id(None), id(type(None)))
```

```
93923529721376 93923529721376 93923529721376
```

```
93923529715952 93923529715552
```

- проверка равенства идентификаторов делается с помощью оператора `is`

```
In [34]: a = None  
         print(a is None)  
         print(a == None)
```

```
True
```

```
True
```

## Идентификатор изменяемого объекта

- два одинаковых списка не идентичны:

```
In [35]: a = [1, 2, 3]
         b = [1, 2, 3]
         a is b
```

```
Out[35]: False
```

- две ссылки на один список идентичны:

```
In [39]: c = a
         a is c
```

```
Out[39]: True
```

- и все между собой, очевидно, равны:

```
In [40]: a == b == c
```

```
Out[40]: True
```

## Почему на None надо проверять с помощью is

In [41]: *# <https://stackoverflow.com/questions/3257919/what-is-the-difference-between-is-none-and-none>*

```
class Foo:
    def __eq__(self, other): # magic method: == operator
        return True
```

```
foo=Foo()
```

```
print(foo==None)
```

```
print(foo is None)
```

True  
False

- кроме того, проверка на идентичность в общем случае быстрее, чем на эквивалентность

## Атрибуты объекта

- любой объект в Python имеет атрибуты: набор полей и методов, определяющих свойства объекта и способы работы с ним
- атрибуты можно читать, устанавливать и менять
- список всех атрибутов получается с помощью встроенной функции `dir`

```
In [44]: print(len(dir(5)))  
         print(dir(5)[: 4])
```

```
70  
['__abs__', '__add__', '__and__', '__bool__']
```

```
In [57]: a = 5  
         a.__and__(0)  # exactly the same as 'a and 0'
```

```
Out[57]: 0
```

## Литераты числовых типов языка

- Целые  
(бесконечные):

```
In [5]: 100, -20, 0 # decimal
```

```
Out[5]: (100, -20, 0)
```

```
In [25]: 0b11, 0B10 # binary
```

```
Out[25]: (3, 2)
```

```
In [17]: 0o11, 0011 # octal
```

```
Out[17]: (9, 9)
```

```
In [20]: 0x90A, 0X9F # hexadecimal
```

```
Out[20]: (2314, 159)
```

```
In [26]: bin(1000), oct(1000), hex(1000)
```

```
Out[26]: ('0b1111101000', '0o1750', '0x3e8')
```



## Литераты числовых типов языка

- Вещественные (в CPython - double из C):

```
In [7]: 1.3, 4., 1e+5, 1.0E+54
```

```
Out[7]: (1.3, 4.0, 100000.0, 1e+54)
```

- Комплексные:

```
In [22]: 3+4j, 2.0+1j, 5j
```

```
Out[22]: ((3+4j), (2+1j), 5j)
```

- Расширение для дробных чисел

```
In [45]: import fractions
a = fractions.Fraction(1, 3 ** 1000000)
len(str(a.denominator))
```

```
Out[45]: 477122
```

## Сравнение чисел

In [46]: `x, y, z = 1, 2, 3`

In [47]: `x < y < z`

Out[47]: `True`

In [48]: `x < y >= z`

Out[48]: `False`

In [49]: `x < y != z`

Out[49]: `True`

## Округление вещественных чисел

In [51]: `11.0 // 3.0 # remove remainder`

Out[51]: 3.0

In [58]: `import math`  
  
`print(math.trunc(4.7)) # move to zero`  
`print(math.trunc(-4.7))`

4  
-4

In [57]: `print(math.floor(4.7)) # move to lowest integer`  
`print(math.floor(-4.7))`

4  
-5

In [62]: `print(round(4.3)) # standart round`  
`print(round(-4.7))`

4  
-5

## Полезные встроенные функции для чисел

In [69]: `pow(2, 4) == 2 ** 4`

Out[69]: True

In [71]: `abs(-2)`

Out[71]: 2

In [72]: `sum((1, 2, 3))`

Out[72]: 6

In [73]: `min(1, 2, -7, 44)`

Out[73]: -7

In [76]: `import math`  
`print(math.sqrt(4))`  
`print(math.sin(8))`

2.0

0.9893582466233818

# Побитовые операции

Работаем с целыми числами как с битовыми массивами

In [79]: `1 << 2 # 001b -> 100b == 4d`

Out[79]: 4

In [91]: `7 >> 1 # 111b -> 011b == 3d`

Out[91]: 3

In [82]: `1 & 2 # 01b || 10b == 00b`

Out[82]: 0

In [85]: `1 | 2 # 01b || 10b == 11b == 3d`

Out[85]: 3

## Операции над множествами

```
In [120]: A, B = set('abc'), {'a', 'c', 'd'}
```

```
In [100]: A - B  # minus: in A and not in B (== A.difference(B))
```

```
Out[100]: {'b'}
```

```
In [105]: A | B  # union: in A or in B (== A.union(B))
```

```
Out[105]: {'a', 'b', 'c', 'd'}
```

```
In [106]: A & B  # intersection: in A and in B (== A.intersection(B))
```

```
Out[106]: {'a', 'c'}
```

```
In [108]: A ^ B  # sym diff: (in A and not in B) and via versa (== A.symmetric_difference(B))
```

```
Out[108]: {'b', 'd'}
```

```
In [113]: {'a', 'b'} < A  # is subset (== {'a', 'b'}.issubset(A))
```

```
Out[113]: True
```

## Операции над множествами

```
In [121]: A.add('e')  
A
```

```
Out[121]: {'a', 'b', 'c', 'e'}
```

```
In [122]: A.update(B)  
A
```

```
Out[122]: {'a', 'b', 'c', 'd', 'e'}
```

```
In [123]: A.remove('a')  
A
```

```
Out[123]: {'b', 'c', 'd', 'e'}
```

```
In [124]: 'a' in A
```

```
Out[124]: False
```

## Подробнее об индексировании и срезах (на примере строк)

```
In [125]: s = 'qwerty'  
s[0]
```

```
Out[125]: 'q'
```

```
In [126]: s[0: 10]
```

```
Out[126]: 'qwerty'
```

```
In [128]: s[slice(1, 3)]
```

```
Out[128]: 'we'
```

```
In [127]: s[:10:2]  # QwErTy____
```

```
Out[127]: 'qet'
```

```
In [136]: s[::-1]
```

```
Out[136]: 'ytrewq'
```



# Форматирование строк

В Python есть похожих способа форматирования - выражение и метод

```
In [139]: 'x = %d, y = %f' % (10.5, 11)
```

```
Out[139]: 'x = 10, y = 11.000000'
```

```
In [156]: 'x = %o, y = %E' % (8, 1.0 / 3)  # octal format, exponential + upper case
```

```
Out[156]: 'x = 10, y = 3.333333E-01'
```

```
In [158]: 'x = %10.2f' % (1.0 / 3)  # min width and precision
```

```
Out[158]: 'x =          0.33'
```

```
In [186]: # add leading zeros if len < min width
s = '''
x = %(value_1)010.2f
x = %(value_2)010.2f
''' % ({'value_1': 10000, 'value_2': 0.12345})

print(s)  # __str__
s         # __repr__
```

```
x = 0010000.00
```

```
x = 0000000.12
```

```
Out[186]: '\nx = 0010000.00\nx = 0000000.12\n'
```

## Форматирование строк

```
In [187]: 'x = {}, y = {}'.format(10, 20)
```

```
Out[187]: 'x = 10, y = 20'
```

```
In [190]: 'x = {1}, y = {0}'.format(10, 20)
```

```
Out[190]: 'x = 20, y = 10'
```

```
In [191]: f'x = {10}, y = {20}'
```

```
Out[191]: 'x = 10, y = 20'
```

```
In [200]: 'x = {val_1:f}, y = {val_2:010.2f}'.format(val_1=10, val_2=10)
```

```
Out[200]: 'x = 10.000000, y = 0000010.00'
```

```
In [205]: import sys  
'platform: {sys.platform}'.format(sys=sys)
```

```
Out[205]: 'platform: linux'
```

## Регулярные выражения

- Стандартный основной модуль: re
- Расширенный дополнительный: regex

In [229]: `import re`

- `[ ]` - множество допустимых СИМВОЛОВ
- `-` - обозначает диапазон
- `*` - любое число повторений (в т.ч. 0)

In [220]: `eng_letters = re.compile('[a-zA-Z]*')`

```
print(eng_letters.match(''))  
print(eng_letters.match('qwerty'))  
print(eng_letters.match('йцукен'))  
print(eng_letters.match('QWERTY'))  
print(eng_letters.match('qweкен'))
```

```
<re.Match object; span=(0, 0), match=''>  
<re.Match object; span=(0, 6), match='qwerty'>  
<re.Match object; span=(0, 0), match=''>  
<re.Match object; span=(0, 6), match='QWERTY'>  
<re.Match object; span=(0, 3), match='qwe'>
```

## Регулярные выражения

- Букву Ё прописываем отдельно, она вне диапазона
- Как в обычных строках, спецсимволы вносим как обычные с помощью \
- + - любое число повторений, но не менее одного

```
In [231]: eng_letters = re.compile('[a-zA-Za-яА-ЯёЁ\\- .@]+')
```

```
print(eng_letters.match(''))  
print(eng_letters.match('йцукен'))  
print(eng_letters.match('Ёжик'))  
print(eng_letters.match('Фрекен-Бок'))  
print(eng_letters.match('mail@ya.ru'))
```

None

```
<re.Match object; span=(0, 6), match='йцукен'>  
<re.Match object; span=(0, 4), match='Ёжик'>  
<re.Match object; span=(0, 10), match='Фрекен-Бок'>  
<re.Match object; span=(0, 10), match='mail@ya.ru'>
```

## Регулярные выражения

- ^ - обозначает необходимость начала строки при совпадении
- \$ - обозначает необходимость конца строки при совпадении
- {} - после квадратных скобок позволяет ограничить количество вхождений СИМВОЛОВ

```
In [236]: phone_pattern = re.compile('^(\+[0-9]{1} ?\((?[0-9]{3}\)\)? ?[0-9]{7})+$')

print(phone_pattern.match('89167698275'))
print(phone_pattern.match('+7 (916) 7698275'))
print(phone_pattern.match('+7(916)7698275'))
```

None

<re.Match object; span=(0, 16), match='+7 (916) 7698275'>

<re.Match object; span=(0, 14), match='+7(916)7698275'>

```
In [239]: email_pattern = re.compile('^([A-Za-z0-9\.\_\-]+@[A-Za-z0-9\.\_\-]+\.[a-zA-Z]*$)')

print(email_pattern.match('mel-lain@yandex.ru'))
print(email_pattern.match('mel&lain@yandex.ru'))
print(email_pattern.match('mel-lain@yandex'))
```

<re.Match object; span=(0, 18), match='mel-lain@yandex.ru'>

None

None

## Регулярные выражения

```
In [250]: email_pattern = re.compile('[A-Za-z0-9\.\_\- ]+@[A-Za-z0-9\.\_\- ]+\.[a-zA-Z]*')

print(email_pattern.match('mel-lain@yandex.ru mel-lain@yandex.ru'))
email_pattern.findall('mel-lain@yandex.ru mel-lain@yandex.ru')

<re.Match object; span=(0, 18), match='mel-lain@yandex.ru'>
```

```
Out[250]: ['mel-lain@yandex.ru', 'mel-lain@yandex.ru']
```

Вместо конкретных символов можно использовать общие обозначения:

- \s - один пробел
- \S - один не пробел
- \w - один alpha-numeric  
символ

```
In [259]: re.split('\w', '&a%6!7#b*')
```

```
Out[259]: ['&', '%', '!', '#', '*']
```

## Подробнее о работе со словарями

```
In [278]: d = dict.fromkeys(list('abccc'))  
d
```

```
Out[278]: {'a': None, 'b': None, 'c': None}
```

```
In [279]: for k in d.keys(): # view object  
          d[k] = ord(k)  
d
```

```
Out[279]: {'a': 97, 'b': 98, 'c': 99}
```

```
In [280]: for k, v in d.items(): # view object  
          print(k, v)
```

```
a 97  
b 98  
c 99
```

```
In [283]: D = dict(a=ord('a'), b=ord('b'), c=ord('c'))  
d == D
```

```
Out[283]: True
```

## Стандартная задача: подсчёт числа элементов

```
In [341]: s = 'GDJKFHGKJXBZFBVJHBJZBXXXXXXXXXXXXXXXXXXXXHFG'
```

```
In [343]: d = {}
for element in s:
    if not element in d:
        d[element] = 0
    d[element] += 1

result = ''
for k, v in d.items():
    result += f'({k}: {v}) '
print(result.strip())
```

(G: 3) (D: 1) (J: 4) (K: 2) (F: 3) (H: 3) (X: 20) (B: 3) (Z: 2) (V: 1)

```
In [342]: from collections import Counter

for k, v in Counter(s).items():
    print(f'({k}: {v}) ', end='') # print can end not only with '\n'
print()
```

(G: 3) (D: 1) (J: 4) (K: 2) (F: 3) (H: 3) (X: 20) (B: 3) (Z: 2) (V: 1)



## Данные в файлы можно добавлять

```
In [310]: with open('test.py', 'r') as fin: # synonim name
          for line in fin:
              print(line[:-1]) # rm final '\n' without strip to save \t
```

```
print(11)
def new_print(a):
    print(a)
```

```
In [311]: with open('test.py', 'a') as fout: # open to append data
          fout.write('new_print(5)\n')

          with(open('test.py')) as fin:
              exec(fin.read())
```

```
11
5
```

# Содержимое файлов как массив байтов

Полезно при работе с сериализаторами

```
In [335]: s = bytearray('строка', 'utf-8')  
with open('tmp.bin', 'wb') as fout:  
    fout.write(s)
```

```
In [337]: with open('tmp.bin', 'r') as fin:  
    print(list(fin.read()))
```

```
['c', 'т', 'р', 'о', 'к', 'а']
```

```
In [338]: with open('tmp.bin', 'rb') as fin:  
    print(list(fin.read()))
```

```
[209, 129, 209, 130, 209, 128, 208, 190, 208, 186, 208, 176]
```

```
In [339]: len('з'.encode('utf-8'))
```

```
Out[339]: 2
```

## Модуль os для работы с системой

```
In [351]: import os  
os.path.exists('test.py')
```

Out[351]: True

```
In [352]: os.path.isfile('test.py')
```

Out[352]: True

```
In [353]: os.path.isdir('test.py')
```

Out[353]: False

```
In [358]: len(os.listdir('.'))
```

Out[358]: 26

```
In [366]: addr = os.path.dirname(os.path.abspath('test.py'))  
print(addr)
```

```
In [369]: os.path.join(*addr.split('/')) # f(*[a, b, c]) -> f(a, b, c)
```

Out[369]: 'home/mel-lain/mipt-python/tmp'

**Спасибо за внимание!**