



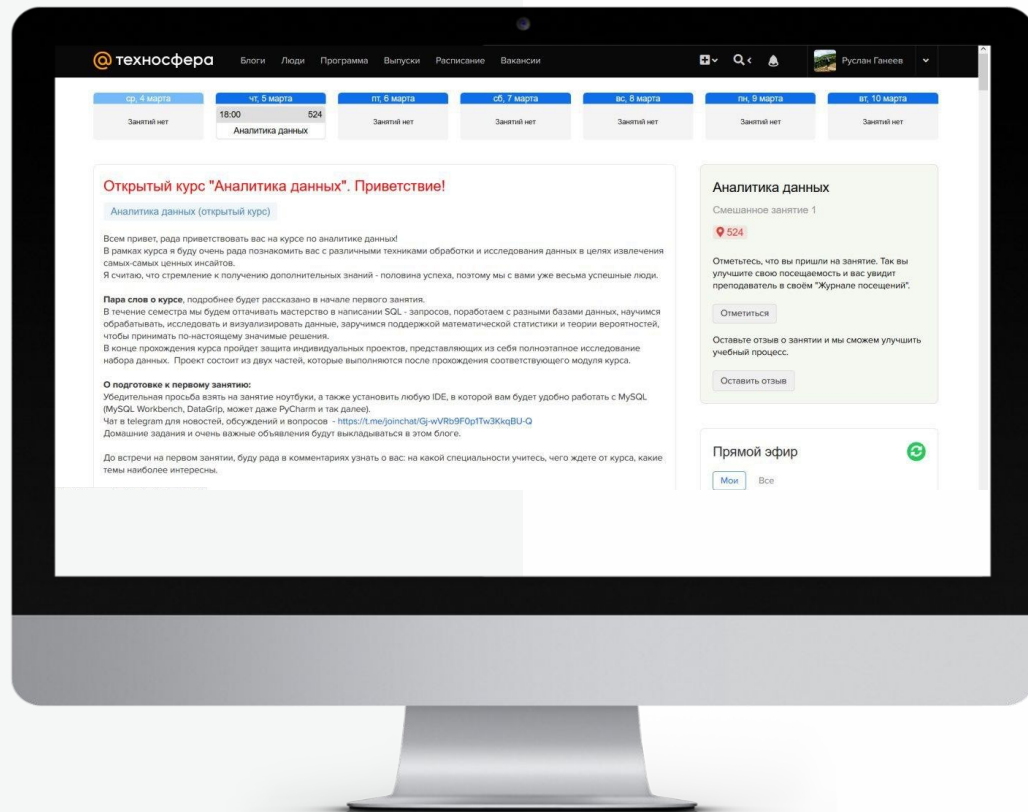
**Backend разработка на Python**

**Лекция 2**

**Объектная модель,  
метапрограммирование, память**

Кандауров Геннадий





# Напоминание отметиться на портале

+ отзывы после лекции


---

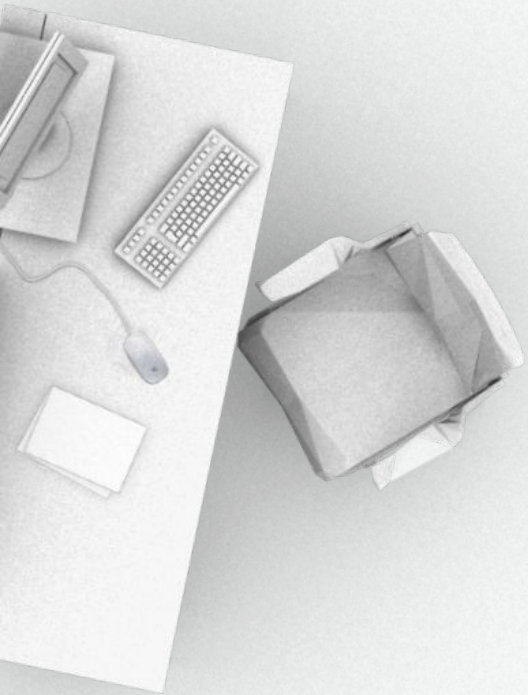
## Квиз по прошлой лекции





## Содержание занятия

- Объектная модель
  - Классы
  - Collections
  - Дескрипторы
  - Метаклассы
  - Организация и выделение памяти
  - Счетчик ссылок и сборщик мусора
  - Профилирование
- 




# Всё есть объект

*“Objects are Python’s abstraction for data.  
All data in a Python program is represented  
by objects or by relations between objects.”*

[docs.python.org](https://docs.python.org)




# Объект

- 
1. Каждый объект имеет идентичность, тип, значение
  2. **id** никогда не меняется после создания объекта (**is** сравнивает **id** объектов)
  3. Тип объекта определяет какие операции с ним можно выполнять
  4. Значение объекта может меняться



## PyObject



```
typedef struct _object {  
    _PyObject_HEAD_EXTRA  
    Py_ssize_t ob_refcnt;  
    PyTypeObject *ob_type;  
} PyObject;
```



# Стандартные типы

Типы с одним значением

1. None
2. NotImplemented
3. Ellipsis (...)



# Типы с одним значением

```
>>> None
>>> type(None)
<class 'NoneType'>
>>> NotImplemented
NotImplemented
>>> type(NotImplemented)
<class 'NotImplementedType'>
```

```
>>> ...
Ellipsis
>>> type(...)
<class 'ellipsis'>
>>> type(None)()
>>> type(None]() is None
True
>>> type(NotImplemented)() is
NotImplemented
True
```



# Стандартные типы

## **numbers.Number**

(всегда неизменяемые)

- numbers.Integral (int, bool)
- numbers.Real (float)
- numbers.Complex (complex)

```
>>> import numbers
>>> issubclass(int, numbers.Number)
True
>>> issubclass(bool, int)
True
>>> issubclass(float, numbers.Real)
True
```



# Стандартные типы

## Sequences

Представляют собой конечные упорядоченные множества, которые проиндексированы неотрицательными числами.

`len(sequence)` возвращает длину последовательности.

Делятся на:

- `immutable` — Strings, Tuples, Bytes
- `mutable` — Lists, Byte Arrays

# Стандартные типы

## Set

Множество уникальных неизменяемых объектов.

Множество не индексируется, но по нему можно итерироваться.

Существует 2 типа множеств:

- Set
- Frozenset

```
>>> s = set()
```

```
>>> s.add(1)
```

```
>>> 1 in s
```

```
True
```

```
>>> s.remove(1)
```

```
>>> 1 in s
```

```
False
```

# Стандартные типы

## Mappings

Есть только 1 маппинг тип – `Dictionaries`. Ключами могут быть только неизменяемые типы, также стоит отметить, что `hash` от ключа должен выполняться за константное время, чтобы структура данных была эффективной.

```
>>> d = {i: i * i for i in range(10)}
>>> d[10] = 100
>>> d[5]
25
>>> d.pop(5, None)
25
>>> 5 in d
False
>>> d.get(5, 99)
99
```

## Стандартные типы (вопросы)

```
>>> a = {1.0}
```

```
>>> 1.0 in a
```

```
???
```

```
>>> 1 in a
```

```
???
```

```
>>> True in a
```

```
???
```

```
>>> x = 5
```

```
>>> y = 5
```

```
>>> id(x) == id(y)
```

```
???
```

```
>>> x = 257
```

```
>>> y = 257
```

```
>>> id(x) == id(y)
```

```
???
```

```
>>> x = "str1"
```

```
>>> y = "str2"
```

```
>>> x is y
```

```
???
```

```
>>> x = "str1+"
```

```
>>> y = "str2+"
```

```
>>> x is y
```

```
???
```



# Стандартные типы

## Модули

Модули являются основным компонентом организации кода в питоне  
(и это тоже объекты).

Модули объединяются в пакеты



# Стандартные типы

## Callable types

- Пользовательские функции
- Методы класса
- Функции генераторы
- Корутины
- Асинхронные генераторы
- Built-in functions
- Классы
- Экземпляры класса





# Магические поля

## Пользовательские функции

`__doc__` докстринг, изменяемое

`__name__` имя функции, изменяемое

`__qualname__` fully qualified имя, изменяемое

`__module__` имя модуля, в котором определена функция, изменяемое

# Магические поля

## Пользовательские функции

```
>>> def foo():  
...     """aaaaaaa"""  
...     pass  
...  
>>> foo.__doc__  
'aaaaaaa'  
>>> foo.__name__  
'foo'
```

```
>>> def wrapper():  
...     a= 1  
...     def foo():  
...         print(a)  
...     return foo  
...  
>>> wrapper().__qualname__  
'wrapper.<locals>.foo'  
>>> wrapper.__module__  
'__main__'
```



# Магические поля

## Пользовательские функции

`__defaults__` — tuple дефолтных значений, изменяемое

`__kwdefaults__` — словарь дефолтных значений кваргов, изменяемое

`__globals__` — словарь глобальных значений модуля, где функция объявлена,  
неизменяемое

`__dict__` — namespace функции, изменяемое

# Магические поля

```
>>> def foo(a=1, b=2):  
...     pass  
...  
>>> foo.__defaults__  
(1, 2)  
>>> foo.__kwdefaults__  
{'a': 1, 'b': 2}
```

```
>>> foo.__globals__  
{... '__name__': '__main__',  
 'numbers': <module 'numbers' from  
 '/usr/local/lib/python3.7/numbers  
 .py'>...}  
>>> foo.a = 1  
>>> foo.__dict__  
{'a': 1}
```

# Магические поля

## Пользовательские функции

`__annotations__` — словарь аннотаций, изменяемое

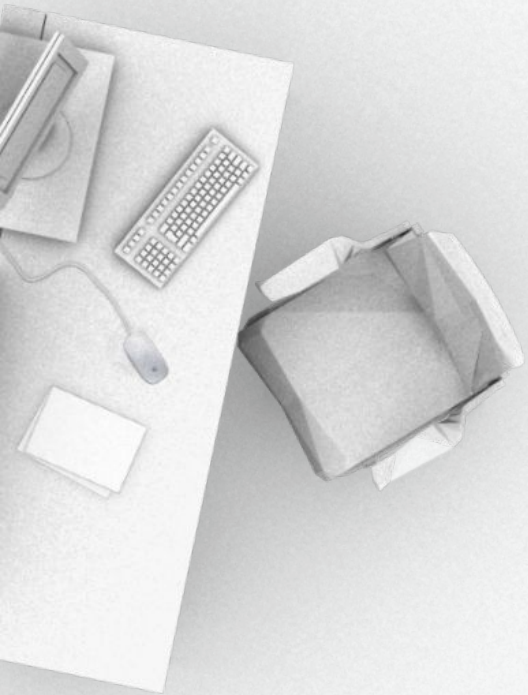
`__code__` — объект типа `code`, изменяемое

`__closure__` — tuple ячеек, которые содержат биндинг к переменным замыкания

```
>>> def foo(a: int, b: float):  
...     pass  
...
```

```
>>> foo.__annotations__  
{'a': <class 'int'>, 'b': <class 'float'>}
```

```
>>> foo.__code__  
<code object foo at 0x7f98fe73d660, file "<stdin>", line 1>
```



**Классы**

# Пользовательские классы

```
class A:
    def __init__(self, param):
        self.param = para
        self.__priv = 99
    def print_param(self):
        print(self.param, self.__priv)
```

```
class B(A): pass
```

```
>>> b = B(10)
>>> b.print_param()
10, 99
>>> b.__priv ???
>>> isinstance(b, B) ???
>>> isinstance(b, A) ???
>>> isinstance(B, A) ???
```

# Классы

```
>>> class A:
...     def foo(self):
...         pass
...
>>> A.foo
<function A.foo at 0x1025929d8>
>>> A().foo
<bound method A.foo of <__main__.A object at 0x102595048>>
>>> A().foo.__func__
<function A.foo at 0x1025929d8>
>>> A().foo.__self__
<__main__.A object at 0x102595048>
```





# Магические поля

## Классы

`__name__` — имя класса

`__module__` — модуль, в котором объявлен класс

`__qualname__` — fully qualified имя

`__doc__` — докстринг

`__annotations__` — аннотации статических полей класса

`__dict__` — namespace класса

## Методы

`__self__` — объект класса

`__func__` — сама функция, которую мы в классе объявили

# Магические поля

## Методы

```
>>> class A:
...     def foo(self):
...         pass
...
>>> A.foo
<function A.foo at 0x1025929d8>
>>> A().foo
<bound method A.foo of <__main__.A object at
0x102595048>>
>>> A().foo.__func__
<function A.foo at 0x1025929d8>
>>> A().foo.__self__
<__main__.A object at 0x102595048>
```



# Магические поля

Классы (поля, относящиеся к наследованию)

`__bases__` — базовые классы

`__base__` — базовый класс, который указан первым по порядку

`__mro__` — список классов, упорядоченный по вызову `super` функции



# Магические поля

## `__slots__`

Поле позволяет явно указать поля, которые будут в классе. В случае указания `__slots__` пропадают поля `__dict__` и `__weakref__`.

Используя `__slots__` можно сильно экономить на памяти и времени доступа к атрибутам объекта.



# Магические поля

## To string

`__repr__` — представление объекта. Если возможно, должно быть валидное python выражение для создание такого же объекта

`__str__` — вызывается функциями `str`, `format`, `print`

`__format__` — вызывается при форматировании строки



# Магические поля

## Rich comparison

`object.__lt__(self, other)`

`object.__le__(self, other)`

`object.__eq__(self, other)`

`object.__ne__(self, other)`

`object.__gt__(self, other)`

`object.__ge__(self, other)`

`x < y == x.__lt__(y), <=, ==, !=, >, >=`



# Магические поля

`__hash__`

Вызывается функцией `hash()` и коллекциями, которые построены на основе hash-таблиц. Нужно, чтобы у равных объектов был одинаковый hash.

Если определен метод `__eq__` и не определен `__hash__`, то объект не может быть ключом в hashable коллекции.



# Хэшируемость

**set, dict**

```
>>> key1 = (1, 2, 3)
```

```
>>> key2 = (1, 2, 3, [4, 5])
```

```
>>> s = set()
```

```
>>> s.add(key1)
```

```
>>> s.add(key2)
```





# Магические поля

## Эмуляция контейнеров

`object.__len__(self)`

`object.__length_hint__(self)`

`object.__getitem__(self, key)`

`object.__setitem__(self, key, value)`

`object.__delitem__(self, key)`

`object.__missing__(self, key)`

`object.__iter__(self)`


`object.__reversed__(self)`

`object.__contains__(self, item)`



# Магические поля

## Эмуляция чисел



```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other) (@)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

# Магические поля

## Эмуляция чисел

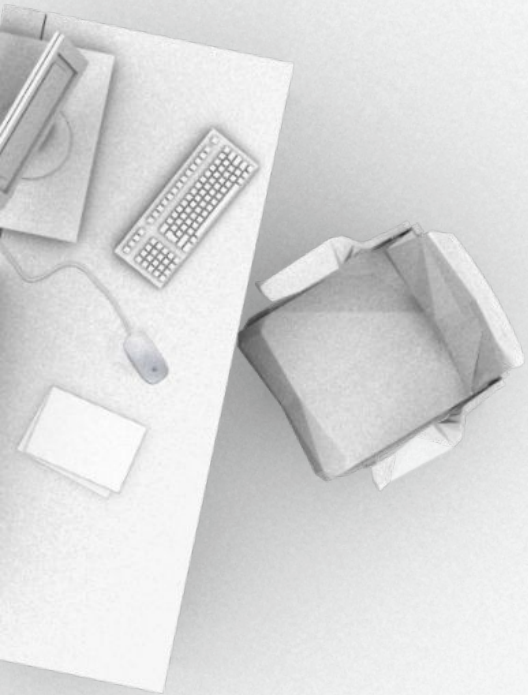
Методы вызываются, когда выполняются операции (+, -, \*, @, /, //, %, divmod(), pow(), \*\*, <<, >>, &, ^, |) над объектами

`x + y == x.__add__(y)`

Есть все такие же с префиксом `r` и `i`.

`__radd__` - вызывается, если левый операнд не поддерживает `__add__`


`__iadd__` - вызывается, когда `x += y`



**collections**



# Что такое модуль `collections`?



Данный модуль реализует специализированные типы данных контейнера, предоставляя альтернативы для встроенных контейнеров таких как `dict`, `list`, `set` и `tuple`.



# Что такое модуль collections?

Рассмотрим:

- `defaultdict`
- `OrderedDict`
- `Counter`
- `namedtuple`
- `deque`



## defaultdict

`collections.defaultdict([default_factory[, ...]])`

Ничем не отличается от обычного словаря за исключением того, что по умолчанию всегда вызывается функция, возвращающая значение.

## defaultdict | Пример

```
collections.defaultdict([default_factory[, ...]])

>>> import collections
>>> defdict = collections.defaultdict(list)
>>> print(defdict)
defaultdict(<class 'list'>, {})
>>> for i in range(5):
...     defdict[i].append(i)
...
>>> print(defdict)
>>> defaultdict(<class 'list'>, {0: [0], 1: [1], 2: [2], 3:
[3], 4: [4]})
```





# OrderedDict

`collections.OrderedDict([items])`

Похожий на словарь объект, но он помнит порядок, в котором ему были даны ключи.

```
>>> import collections
>>> d = collections.OrderedDict(
...     [('a', 'A'), ('b', 'B'), ('c', 'C')])
... )
>>> for k, v in d.items():
...     print(k, v)
>>> d.move_to_end('b')
```



# Counter


`collections.Counter([iterable-or-mapping])`

Это подкласс dict для подсчёта хешируемых объектов.

- `elements()`
- `most_common([n])`
- `subtract([iterable-or-mapping])`
- `update([iterable-or-mapping])`



## Counter | Пример



```
>>> import re
>>> words = re.findall(
    r'\w+',
    open('hamlet.txt').read().lower()
)
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669),
('i', 631), ('you', 554), ('a', 546), ('my', 514),
('hamlet', 471), ('in', 451)]
```

# namedtuple

```
collections.namedtuple(typename, field_names, *, rename=False,  
defaults=None, module=None)
```

Именованные кортежи являются неизменяемыми подобно обычным кортежам. Вы не можете изменять их после того, как вы что-то поместили в них.

```
>>> import collections  
>>> Point = collections.namedtuple('Point', ['x', 'y'])  
>>> p = Point(11, y=22)  
>>> p[0] + p[1] # p = (11, 22)  
33  
>>> x, y = p  
>>> x, y  
(11, 22)  
>>> p.x + p.y  
33
```



## namedtuple | Выводы

- `collections.namedtuple` — краткая форма для создания вручную эффективно работающего с памятью неизменяемого класса;
- Именованные кортежи могут помочь сделать ваш код чище, обеспечивая вас более простыми в понимании структурами данных;
- Именованные кортежи предоставляют несколько полезных вспомогательных методов которые начинаются с символа подчёркивания (`_`), но являются частью открытого интерфейса. Использовать их — это нормальная практика.




## deque

```
collections.deque([iterable[, maxlen]])
```

Очередь из итерируемого объекта с максимальной длиной `maxlen`. Очереди очень похожи на списки, за исключением того, что добавлять и удалять элементы можно либо справа, либо слева.




## deque | Методы

- 
- `append(x)/appendleft(x)`
  - `clear()`
  - `copy()`
  - `count(x)`
  - `extend(iterable)/extendleft(iterable)`
  - `index(x[, start[, stop]])`
  - `insert(i, x)`
  - `pop()/popleft()`
  - `remove(value)`
  - `reverse()`

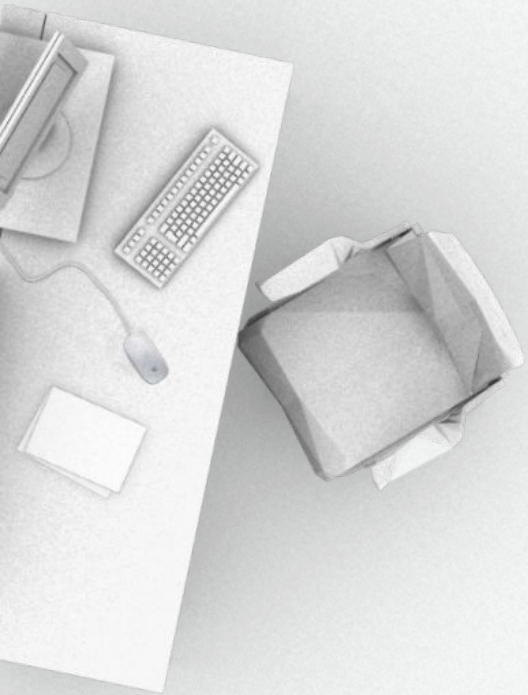


## deque | Пример



```
>>> from collections import deque
>>> d = deque('ghi') # make a new deque with three items
>>> d.append('j') # add a new entry to the right side
>>> d.appendleft('f') # add a new entry to the left side
>>> d # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])
>>> d.pop() # return and remove the rightmost item
'j'
>>> d.popleft() # return and remove the leftmost item
'f'
```





# Дескрипторы

доступ к атрибутам и дескрипторы



# Доступ к атрибутам

Рассмотрим подробнее атрибут `__dict__`

Чтобы найти атрибут объекта *obj*, python обыскивает:

1. Сам объект (`o.__dict__` и его системные атрибуты)
2. Класс объекта (`o.__class__.__dict__`).
3. Классы, от которых унаследован класс объекта (`o.__class__.__mro__`)

# Дескрипторы

```
>>> class A:
    def foo(self):
        Pass
a = A()
```

```
>>> a.foo.__class__.__get__
<slot wrapper '__get__' of 'method' objects>
```

```
>>> A.__dict__['foo'] # Внутренне хранится как функция
<function foo at 0x00C45070>
```

```
>>> A.foo # Доступ через класс возвращает несвязанный метод
<unbound method A.foo>
```

```
>>> a.foo # Доступ через экземпляр объекта возвращает связанный метод
<bound method A.foo of <__main__.A object at 0x00B18C90>>
```



# Дескрипторы

*“Дескриптор это атрибут объекта со “связанным поведением”, то есть такой атрибут, при доступе к которому его поведение переопределяется методом протокола дескриптора. Эти методы `__get__`, `__set__` и `__delete__`. Если хотя бы один из этих методов определен в объекте , то можно сказать что этот объект дескриптор.”*

Раймонд Хеттингер



# Дескрипторы

Если определен один из методов `__get__`, `__set__` и `__delete__`, объект считает дескриптором.

Если объект дескриптора определяет `__get__`, `__set__` - он считает data дескриптором.

Если объект дескриптора определяет `__get__` - он считает non-data дескриптор.

# Дескрипторы

```
class MyDescriptor:
    def __get__(self, obj, objtype):
        print(f"access from {obj} class {cls}")

    def __set__(self, obj, val):
        print(f"set {val} for {obj}")


    def __delete__(self, obj):
        print(f"delete from {obj}")

class MyClass:
    field = MyDescriptor()
```

```
>>> inst = MyClass()
>>> MyClass.field # вот тут будет вызван __get__ с obj None
>>> inst.field
>>> inst.field = 1
>>> del inst.field
```



# Дескрипторы




```
from sqlalchemy import Column, Integer, String

class User(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String)
```



# Дескрипторы



```
class Order:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    def total(self):
        return self.price * self.quantity

apple_order = Order('apple', 1, 10)
apple_order.total()
```





# Дескрипторы

```
class Order:
    price = NonNegative('price')
    quantity = NonNegative('quantity')

    def __init__(self, name, price, quantity):
        self._name = name
        self.price = price
        self.quantity = quantity
    def total(self):
        return self.price * self.quantity
```

```
apple_order = Order('apple', 1, 10)
apple_order.total()
# 10
apple_order.price = -10
# ValueError: Cannot be negative
apple_order.quantity = -10
```



# Дескрипторы


## Методы доступа к атрибутам (yet another магия)

Методы `__getattr__()`, `__setattr__()`, `__delattr__()` и `__getattribute__()`.

В отличие от дескрипторов их следует определять для объекта, содержащего атрибуты и вызываются они при доступе к любому атрибуту этого объекта.



# Магические методы. Кастомизация объектов



`object.__new__(cls[, ...])` – создает новый объект класса, статический метод по преданию.

После создание объекта вызывается (уже у объекта) метод `__init__`. Он ничего не должен возвращать, иначе будет `TypeError`

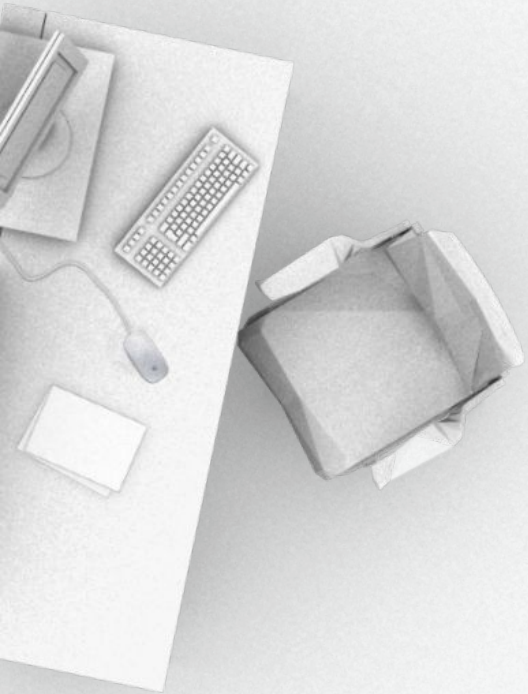
# Кастомизация объектов. Подумать

```
>>> class Foo:
...     pass
...
>>> x = Foo()
>>> type(x)
<class '__main__.Foo'>
>>> type(Foo)
???
>>> type(type)
???
>>> isinstance(type, object)
???
```



# Кастомизация

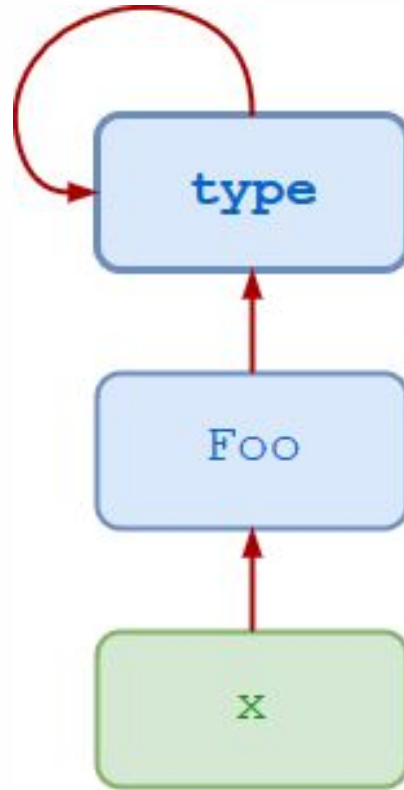
```
class Singleton(object):  
    _instance = None  
  
    def __new__(cls, *args, **kwargs):  
        if cls._instance is None:  
            cls._instance = super().__new__(cls, *args, **kwargs)  
        return cls._instance
```



# Метаклассы

классы, экземпляры которых являются классами

**type**





# Метаклассы



Новые классы создаются с помощью вызова  
`type(<name>, <bases>, <classdict>)`

`name` – имя класса (`__name__`)

`bases` – базовые классы (`__bases__`)

`classdict` – namespace класса (`__dict__`)

`MyClass = type('MyClass', (), {})`



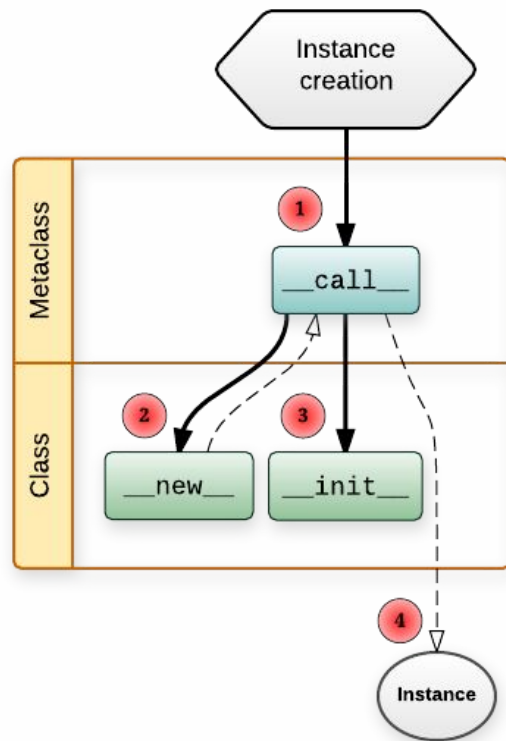
# Метаклассы

```
>>> Bar = type('Bar', (Foo,), dict(attr=100))
>>> x = Bar()
>>> x.attr
100
>>> x.__class__
<class '__main__.Bar'>
>>> x.__class__.__bases__
(<class '__main__.Foo'>,)

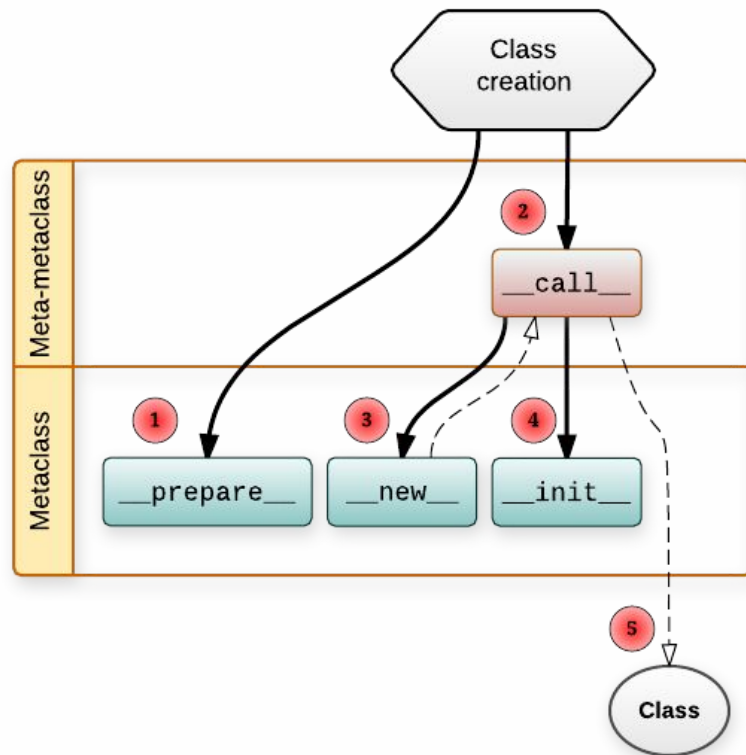
>>> class Bar(Foo):
...     attr = 100
...
>>> x = Bar()
>>> x.attr
100
>>> x.__class__.__bases__
(<class '__main__.Foo'>,)

```

# Создание объекта



# Создание класса





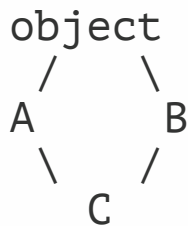
# Как создается класс

- определяются базовые классы
- определяется метакласс
- подготавливается namespace класса (`__prepare__`)
- выполняется тело класса
- создается класс (`__new__`, `__init__`)

# MRO

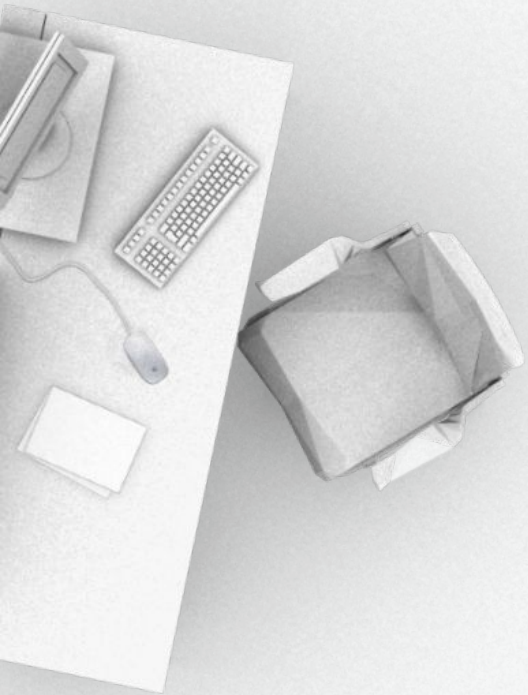
Порядок разрешения методов (method resolution order) позволяет python выяснить, из какого класса-предка нужно вызывать метод, если он не обнаружен непосредственно в классе-потомке.

```
.__mro__  
.mro()
```



# Локальный порядок старшинства

```
>>> class A:
...     pass
...
>>> class B:
...     pass
...
>>> class C(A, B):
...     pass
...
>>> C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
>>>
>>> class C(B, A):
...     pass
...
>>> C.mro()
[<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```




# ABC

добавляем абстракции



# ABC



Модуль, который позволяет определять абстрактные базовые классы (abstract base classes).



## ABC example. Hashable

```
class Hashable(metaclass=ABCMeta):
    __slots__ = ()
    @abstractmethod
    def __hash__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Hashable:
            return _check_methods(C, "__hash__")

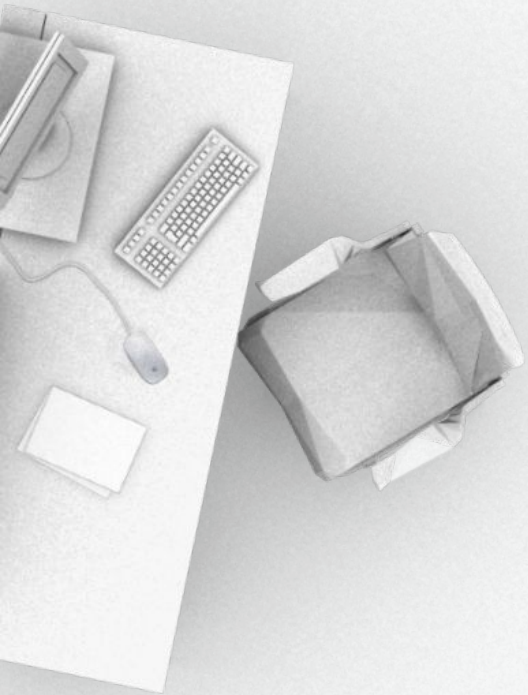
        return NotImplemented
```

```
>>> from collections.abc import Hashable
>>> isinstance("123", Hashable)
>>> isinstance({}, Hashable)
```

# ABC example

```
>>> from abc import ABCMeta
>>> class C(metaclass=ABCMeta):
...     @abstractmethod
...     def abs_method(self):
...         pass
>>> c = C()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class C with abstract methods absMethod

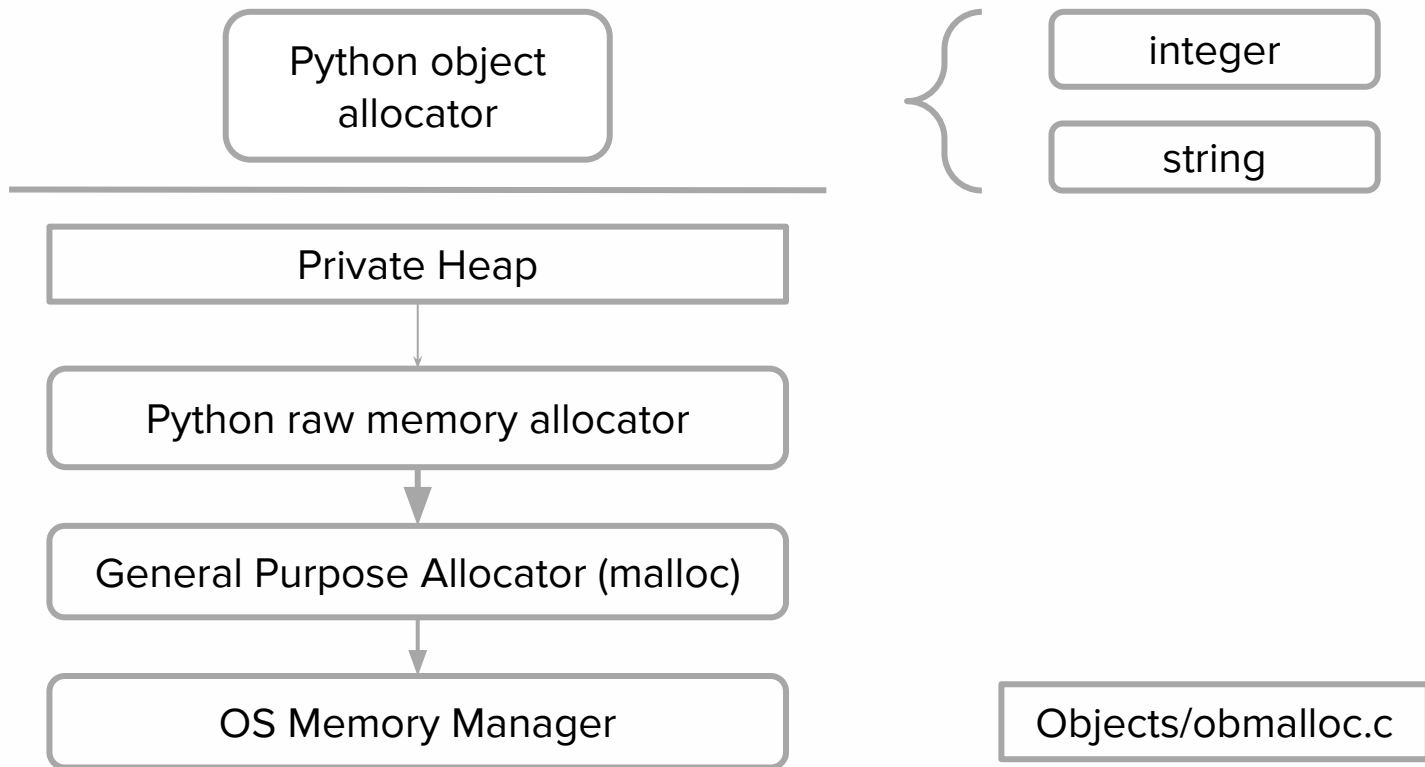
>>> class B(C):
...     def abs_method(self):
...         print("Now a concrete method")
>>> b = B()
>>> b.abs_method()
Now a concrete method
```



# Устройство памяти


Выделение, очищение, управление

# Выделение памяти





## PyObject



```
typedef struct _object {  
    _PyObject_HEAD_EXTRA  
    Py_ssize_t ob_refcnt;  
    PyTypeObject *ob_type;  
} PyObject;
```

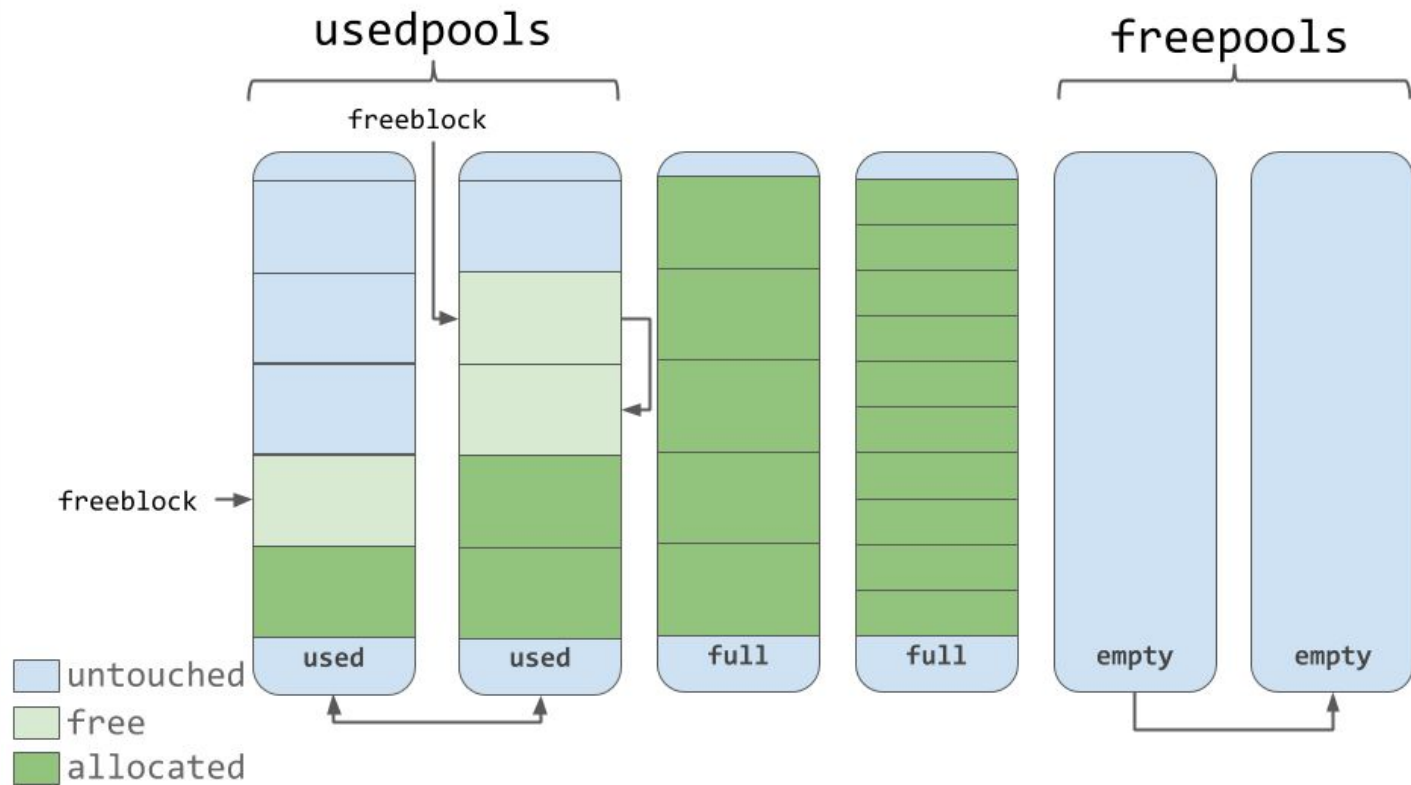


## Выделение памяти

- Большие объекты ( $> 512$  байт): C allocator;
- Меньшие объекты ( $\leq 512$  байт): арены, пулы, блоки;
  - Блок хранит один объект от 1 до 512 байт;
  - Пул хранит блоки, занимает одну страницу памяти (4Кб);
  - Арена хранит пулы, занимает 256Кб;

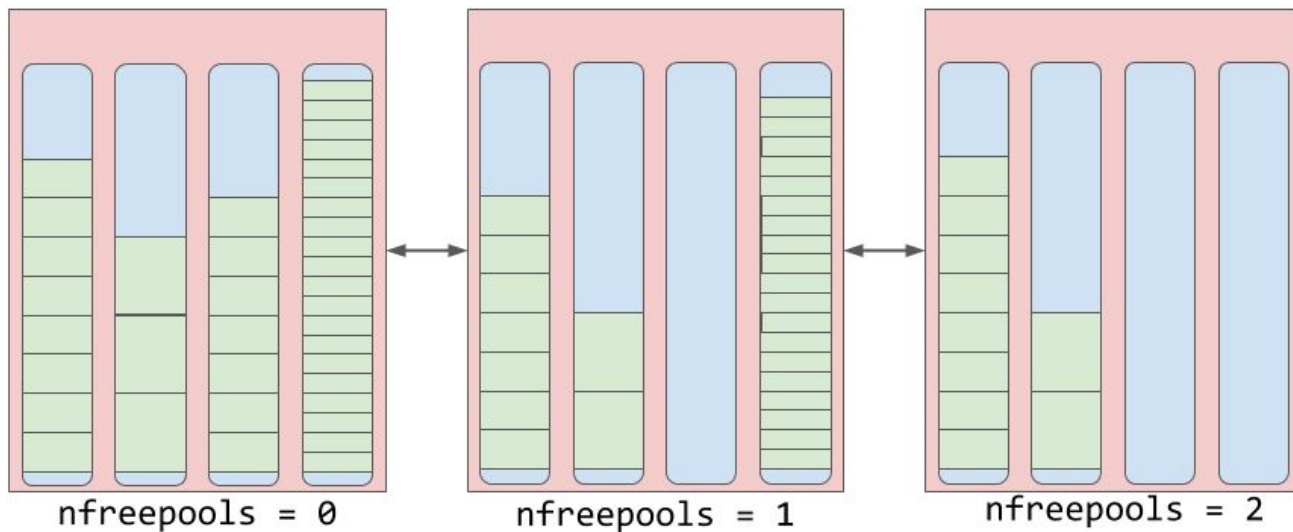
Только арена может освобождать память

## Выделение памяти: пул



# Выделение памяти: арена

usable\_arenas







## Освобождение памяти




*“The only reliable way to free memory is to terminate the process”*



## Освобождение памяти

- счетчик ссылок, refcounter
- generational garbage collector, модуль gc (опционален)



```
>>> import sys
>>> foo = []
>>> print(sys.getrefcount(foo))
>>> def bar(a):
>>>     print(sys.getrefcount(a))
>>> bar(foo)
>>> print(sys.getrefcount(foo))
```

## Счетчик ссылок (refcount)



Память сразу можно  
очистить



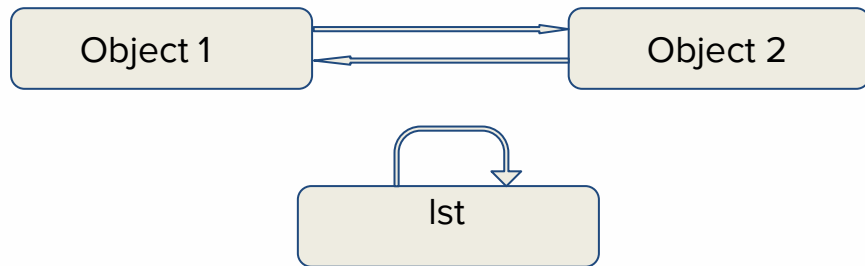
Циклические ссылки



Блокирование потоков




Доп расход CPU и RAM





## Сборка мусора

- 
- + Не нужно думать об очистке памяти
  - + Никаких double free ошибок
  - + Решение проблем утечек памяти

- Дополнительное использование CPU и RAM
- Момент сборки мусора непредсказуем

\_\_\_\_\_





# Сборщик мусора

GC следит только за объектами контейнерами, если они содержат тоже объекты-контейнеры

1. list
2. dict
3. tuple
4. class
5. Etc

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

## Сборщик мусора: отключение

1. Отключение gc

```
gc.disable()
```

```
gc.collect()
```

2. weakref (<https://docs.python.org/3/library/weakref.html>)

```
weakref.ref
```

```
WeakKeyDictionary, WeakValueDictionary, WeakSet,
```

```
WeakMethod;
```

```
finalize;
```

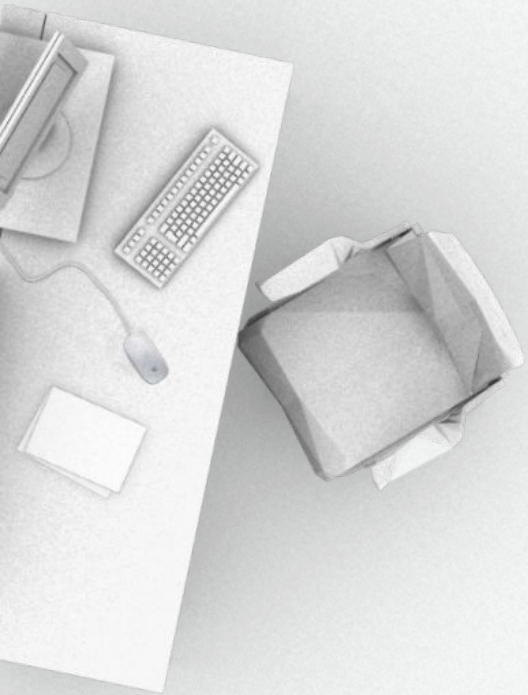
```
list, dict: только для подклассов;
```

```
tuple, int: не поддерживаются.
```

## weakref

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
>>> del o, o2
>>> print(r())
None
```





# Мониторинг потребления ресурсов



## top/atop

**top** - консольная команда, которая выводит список работающих в системе процессов и информацию о них.

**PID** - идентификатор процесса

**USER** - пользователь, под которым запущен процесс

**VIRT** - объем виртуальной памяти, занимаемой процессом

**RES** - текущее использование RAM

**%CPU** - процент доступного времени процессора

**atop** - продвинутый интерактивный полноэкранный монитор производительности, написанный для Linux.

```
atop -r /var/log/atop/atop_<date> [-b hh:mm]
```



## iotop/iostat

**iotop** - утилита, выводящая данные по использованию жесткого диска.

- `iotop -o` (активные процессы)
- `iotop -o -a` (собрать статистику за время)

**iostat** - утилита, предназначенная для мониторинга использования дисковых разделов.

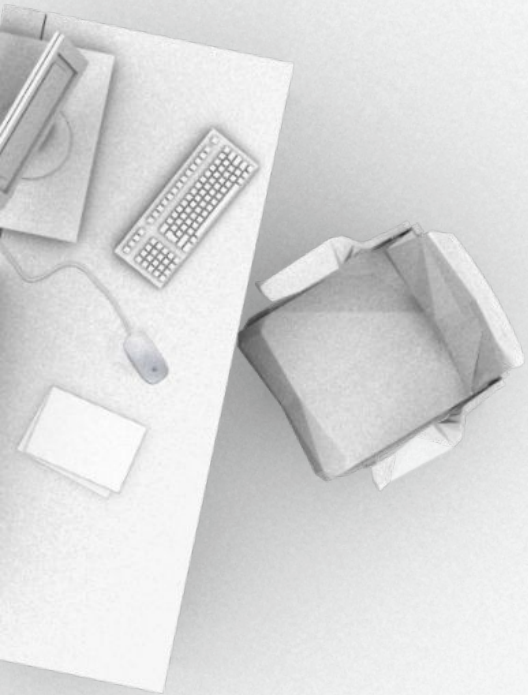
```
iostat -d -t -p sda -x
```

`-c` вывести отчет по CPU

`-d` вывести отчет по использованию диска

`-t` интервал, за который усредняются значения

`-x` вывести расширенную статистику



# Профилирование

сбор характеристик работы программы, таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно предсказанных условных переходов, число кэш-промахов и т. д.



# Профилирование

Цель:

- найти узкие места в коде

Основные способы:

- CPU
- Память
- Частота/продолжительность вызовов функций

Методы:

- Статистический (сэмплирование)
- Детерминированный (инструментирование)

## Профилирование python

- ***cProfile*** - написанная на C, быстрая реализация профилировщика
- ***profile*** - нативная реализация профилировщика на чистом python, значительно медленнее

```
python -m cProfile -o output.txt ptest.py
```

```
import pstats
```

```
p = pstats.Stats('output.txt')
```

```
p.strip_dirs().sort_stats(-1).print_stats()
```

# Профилирование python

```
import cProfile, pstats, io
```

```
pr = cProfile.Profile()
```

```
pr.enable()
```

```
# ... do something ...
```

```
pr.disable()
```

```
s = io.StringIO()
```

```
sortby = 'cumulative'
```

```
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
```

```
ps.print_stats()
```

```
print(s.getvalue())
```

# Профилирование python

1567629 function calls (1166637 primitive calls) in 809.730 seconds

Ordered by: cumulative time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.164	0.164	809.738	809.738	/Users/project/src/.env3/lib/python3.7/site-packages/tornado/ioloop.py:568(start)
4961	806.444	0.163	806.444	0.163	/Users/project/src/.env3/lib/python3.7/site-packages/tornado/platform/kqueue.py:66(poll)
9982/8005	0.086	0.000	3.095	0.000	/Users/project/src/.env3/lib/python3.7/site-packages/tornado/stack_context.py:269(wrapped)
5657	0.011	0.000	2.767	0.000	/Users/project/src/.env3/lib/python3.7/site-packages/tornado/ioloop.py:471(_run_callback)
6766/2479	0.083	0.000	1.869	0.001	/Users/project/src/.env3/lib/python3.7/site-packages/tornado/gen.py:507(run)
2445	0.009	0.000	1.775	0.001	/Users/project/src/.env3/lib/python3.7/site-packages/tornado/gen.py:567(inner)
2445	0.005	0.000	1.764	0.001	/Users/project/src/.env3/lib/python3.7/site-packages/tornado/gen.py:497(set_result)
430	0.008	0.000	0.902	0.002	/Users/project/src/gekko/net/resolver.py:414(resolve)
75	0.000	0.000	0.669	0.009	/Users/project/src/gekko/handlers2/executor.py:93(callback)
75	0.000	0.000	0.669	0.009	/Users/project/src/gekko/handlers2/executor.py:72(_handler_callback)
48	0.000	0.000	0.669	0.014	/Users/project/src/gekko/handlers2/executor.py:114(_done)
72	0.000	0.000	0.612	0.009	/Users/project/src/gekko/location2.py:266(_call_location_method)
60	0.000	0.000	0.610	0.010	/Users/project/src/gekko/location2.py:91(create_gen_tasks)
63	0.000	0.000	0.609	0.010	/Users/project/src/gekkoapps/gosearch/locations/ajax_web.py:27(get)
9	0.000	0.000	0.576	0.064	/Users/project/src/gekkoapps/common/locations/base.py:104(create_response)
9	0.001	0.000	0.572	0.064	/Users/project/src/gekkoapps/common/locations/base.py:97(render_view)
9	0.000	0.000	0.242	0.027	/Users/project/src/gekkoapps/common/locations/base.py:173(get_data_from_view)
9	0.000	0.000	0.242	0.027	/Users/project/src/gekkoapps/common/views/base.py:136(get_data)
9	0.000	0.000	0.239	0.027	/Users/project/src/gekkoapps/gosearch/v1/web/view/compat.py:14(create_location_data)
9	0.000	0.000	0.238	0.026	/Users/project/src/gekkoapps/gosearch/v1/web/view/produce.py:518(get_data)
9	0.000	0.000	0.220	0.024	/Users/project/src/gekkoapps/common/locations/base.py:183(render_json)
9	0.000	0.000	0.220	0.024	/Users/project/src/gekko/template/helpers.py:148(do_json)
9	0.013	0.001	0.220	0.024	/Users/project/src/.env3/lib/python3.7/site-packages/simplejson/encoder.py:371(encode)
3626	0.030	0.000	0.214	0.000	/Users/project/src/gekko/net/resolver.py:185(resolve)
27	0.000	0.000	0.209	0.008	/Users/project/src/gekkoapps/common/views/serp/v1/creator.py:23(create)



## Профилирование памяти

```
pip install memory_profiler
```

```
# run.py  
from memory_profiler import profile
```

```
@profile  
def some_func():  
    lst1 = []  
    lst2 = "1" * 100000
```

```
python -m memory_profiler run.py
```

## Достойны упоминания: pdb, dis, inspect, psutil

```
python3 -m pdb script.py
```

или

```
# script.py
```

```
def some_func():
```

```
    lst1 = []
```

```
    import pdb; pdb.set_trace()
```

```
    lst2 = "1" * 100000
```

```
python script.py
```



---

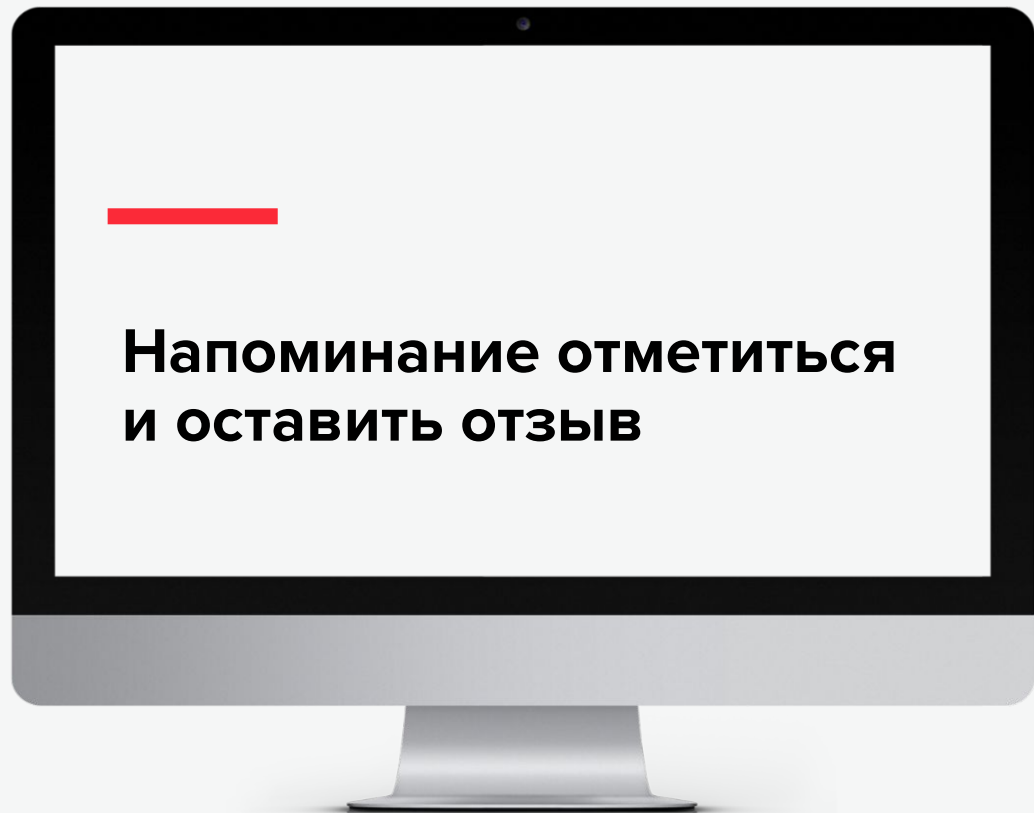
## Домашнее задание по лекции 2

ДЗ #2

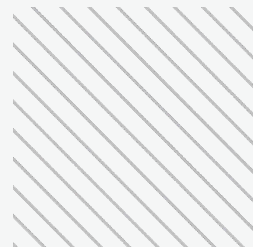
15.06.2020

срок сдачи

- Реализовать класс, отнаследованный от списка, такой, что можно выполнять сложение/вычитание одного списка от другого (+тесты).
- Написать метакласс, который в начале названий всех атрибутов и методов (кроме магических) добавляет префикс "custom\_" (+тесты).



**Напоминание отметить  
и оставить отзыв**



**СПАСИБО  
ЗА ВНИМАНИЕ**

