In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Python 3

Классы

MIPT 2020

Атрибуты и методы

In [2]:
```
class MyLittleClass:
    color = "blue"

    def set_color(self, color_):
        color = color_
        print('set color to {}'.format(color))
```

In [3]:
```
obj = MyLittleClass()
obj.color

obj.set_color('red')
obj.color
```

Out[3]: 'blue'

```
set color to red
```

Out[3]: 'blue'

AaAAAaaa!!! Почему так произошло?!!!

Потому что обращение к атрибутам класса должно иметь форму `self.attribute_name` , а `color` в методе `set_color` -- просто локальная переменная :)

In [4]:
```
MyLittleClass.color = "red"
obj.color
obj.__class__.color
```

Out[4]: 'red'

Out[4]: 'red'

Видим, что `color` на самом деле атрибут класса, а не отдельного объекта

In [5]:
```
class MyLittleClass2:
    color = "blue"

    def set_color(self, color_):
        self.color = color_  # найдите десять отличий :)
        print(f'set color to {self.color}')
```

In [6]:
```
obj = MyLittleClass2()
obj.color

obj.set_color('red') # --> MyLittleClass2.set_color(obj, 'red')
obj.color
```

Out[6]:
```
'blue'

set color to red

'red'
```

In [7]:
```
# вообще-то, так тоже можно было, но хорошие программисты пишут т.н. методы-гет
obj.color = 'green'
obj.color
```

Out[7]:
```
'green'
```

Пример на геттеры-сеттеры

In [8]:
```
class MyLittleClass3:
    my_super_internal_color = 'blue'

    @property
    def color(self):
        return self.my_super_internal_color

    @color.setter
    def color(self, value):
        print("No-No-No, don't touch it!")
```

In [9]:
```
obj = MyLittleClass3()
obj.color
obj.color = 'Haha cheating'
obj.color
```

Out[9]:
```
'blue'

No-No-No, don't touch it!

'blue'
```

https://google.github.io/styleguide/pyguide.html#213-properties (https://google.github.io/styleguide/pyguide.html#213-properties)

Кодстайл гугла советует использовать такие геттеры-сеттеры только если они легковесные и простые, для более сложных стоит делать обычные функции

Q: Динамически определить атрибуты, которых вообще не было в определении класса?

A: Легко!

In [10]:
```
def get_color(self):
    return self.color
```

In [11]:
```
obj.get_color = get_color
```

In [12]:
```
obj.get_color()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-78f1df7c55df> in <module>
----> 1 obj.get_color()

TypeError: get_color() missing 1 required positional argument: 'self'
```

In [13]:
```
MyLittleClass2.get_color(obj)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-13-668a56f1ad34> in <module>
----> 1 MyLittleClass2.get_color(obj)

AttributeError: type object 'MyLittleClass2' has no attribute 'get_color'
```

In [14]:
```
MyLittleClass2.get_color = get_color
```

In [15]:
```
MyLittleClass2.get_color(obj)
```

Out[15]:
```
'blue'
```

In [16]:
```
obj.some_attribute = 42
print(obj.some_attribute)

obj_2 = MyLittleClass2()
print(obj_2.some_attribute)
```

```
42

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-16-7536f5af566e> in <module>
      3
      4 obj_2 = MyLittleClass2()
----> 5 print(obj_2.some_attribute)

AttributeError: 'MyLittleClass2' object has no attribute 'some_attribute'
```

Q: А а что значит self в определении метода?

A: Когда мы вызываем метод как obj.methodname(), первым аргументом передается ссылка на obj (в качестве self)

In [17]:
```
class MyLittleClass4:
    @staticmethod
    def method_without_self(arg):
        print(arg)

    def method_with_self(self, arg):
        print(arg)
```

In [18]:
```
obj = MyLittleClass4()
obj.method_with_self('i am an argument')
obj.method_without_self('i am another argument') # здесь мы на самом деле переда
```

```
i am an argument
i am another argument
```

Q: А как же тогда их вызывать?!

A: Они не привязаны к инстансу (потому что не имеют доступа к его локальным данным), зато привязаны к классу

In [19]:
```
MyLittleClass4.method_without_self('i am another argument') # а здесь мы передае
```

```
i am another argument
```

Q: Можно ли "оторвать" метод от инстанса?

A: Ну, попробуем

In [20]:
```
func = MyLittleClass4.method_without_self
func("hello")
```

```
hello
```

In [21]:
```
func2 = MyLittleClass4.method_with_self
func2("hello") # передаем один аргумент
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-f24f4c94cf25> in <module>
      1 func2 = MyLittleClass4.method_with_self
----> 2 func2("hello") # передаем один аргумент

TypeError: method_with_self() missing 1 required positional argument: 'arg'
```

In [22]:
```
obj = MyLittleClass4()
func2(obj, "hello") # ой, нам же ещё нужен объект для self!
```

```
hello
```

Q: А наоборот?

A: Да это же питон. Конечно, можно!

In [23]:
```
obj.get_color()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-23-78f1df7c55df> in <module>
----> 1 obj.get_color()

AttributeError: 'MyLittleClass4' object has no attribute 'get_color'
```

In [24]:
```
def get_color_function(self):
    return self.color

MyLittleClass4.get_color = get_color_function
obj = MyLittleClass4()
obj.get_color()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-24-6bff4d0895db> in <module>
      4 MyLittleClass4.get_color = get_color_function
      5 obj = MyLittleClass4()
----> 6 obj.get_color()

<ipython-input-24-6bff4d0895db> in get_color_function(self)
      1 def get_color_function(self):
----> 2     return self.color
      3
      4 MyLittleClass4.get_color = get_color_function
      5 obj = MyLittleClass4()

AttributeError: 'MyLittleClass4' object has no attribute 'color'
```

Ах да, цвета-то у нас нет. Но не беда, это же питон!

In [25]:
```
obj.color = 'pink'
obj.get_color()
```

Out[25]:
```
'pink'
```

In [26]:
```
del obj.color
```

Q: А как же узнать, что мы уже определили, а что нет?

A: Легко!

In [27]:
```
print(dir(obj))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__for
mat__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init
_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce_
_', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__sub
classhook__', '__weakref__', 'get_color', 'method_with_self', 'method_without_s
elf']
```

In [28]:
```
print(getattr(obj, 'method_with_self'))
print(getattr(obj, '__doc__'))
```

```
<bound method MyLittleClass4.method_with_self of <__main__.MyLittleClass4 objec
t at 0x7f122408a9d0>>
None
```

In [29]:
```
# оставим только методы
print([name for name in dir(obj) if callable(getattr(obj, name))])
```

```
['__class__', '__delattr__', '__dir__', '__eq__', '__format__', '__ge__', '__ge
tattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__se
tattr__', '__sizeof__', '__str__', '__subclasshook__', 'get_color', 'method_wit
h_self', 'method_without_self']
```

In [30]:
```
class ClassWithNothing:
    pass

nobject = ClassWithNothing()

def print_custom_attrs(obj=None):
    if obj is None:
        # в локальной области видимости!
        attrs = dir()
    else:
        attrs = dir(obj)
    print([name for name in attrs if not name.startswith('__')])

print_custom_attrs(nobject)
print_custom_attrs(ClassWithNothing)
print_custom_attrs()
print(dir())
```

```
[]
[]
['obj']
['ClassWithNothing', 'In', 'InteractiveShell', 'MyLittleClass', 'MyLittleClass2
', 'MyLittleClass3', 'MyLittleClass4', 'Out', '_', '_15', '_25', '_3', '_4', '_
6', '_7', '_9', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loade
r__', '__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_i10', '_i11
', '_i12', '_i13', '_i14', '_i15', '_i16', '_i17', '_i18', '_i19', '_i2', '_i20
', '_i21', '_i22', '_i23', '_i24', '_i25', '_i26', '_i27', '_i28', '_i29', '_i3
', '_i30', '_i4', '_i5', '_i6', '_i7', '_i8', '_i9', '_ih', '_ii', '_iii', '_oh
', 'exit', 'func', 'func2', 'get_color', 'get_color_function', 'get_ipython', '
nobject', 'obj', 'obj_2', 'print_custom_attrs', 'quit']
```

In [31]:
```
ClassWithNothing.my_attribute = 'my value'
nobject.my_instance_attribute = "my value 2"

print_custom_attrs(nobject)
print_custom_attrs(ClassWithNothing)
```

```
['my_attribute', 'my_instance_attribute']
['my_attribute']
```

Наследование

In [32]:
```
from abc import abstractmethod


class Animal:
    some_value = "animal"
    def __init__(self):
        print("i am an animal")

    @abstractmethod
    def speak(self):
        raise NotImplementedError('i don\'t know how to speak')


class Cat(Animal):
    some_value = "cat"
    def __init__(self):
        super().__init__()
        print("i am a cat")

    def speak(self):
        print('meoooow')


class Hedgehog(Animal):
    def __init__(self):
        super().__init__()
        print("i am a hedgehog")


class Dog(Animal):
    some_value = "dog"
    def __init__(self):
        super().__init__()
        print("i am a dog")


class CatDog(Cat, Dog):  # ромбовидное наследование возможно, но не делайте так,
    def __init__(self):
        super().__init__()
        print("i am a CatDog!")
```

In [33]:
```
animal = Animal()
animal.some_value
```

```
i am an animal
```

Out[33]:
```
'animal'
```

In [34]:
```
cat = Cat()
cat.some_value # переопределено
```

```
i am an animal
i am a cat
```

Out[34]:
```
'cat'
```

In [35]:
```
hedgehog = Hedgehog()
hedgehog.some_value # не переопределено
```

```
i am an animal
i am a hedgehog
```

Out[35]:
```
'animal'
```

In [36]:
```
dog = Dog()
dog.some_value # переопределено
```

```
i am an animal
i am a dog
```

Out[36]: 'dog'

In [37]:
```
catdog = CatDog()
catdog.some_value
```

```
i am an animal
i am a dog
i am a cat
i am a CatDog!
```

Out[37]: 'cat'

Q: А как определяется порядок?

A: Порядок перечисления родителей важен!

In [38]:
```
class CatDog(Dog, Cat):  # теперь наоборот, найдите два отличия!
    def __init__(self):
        super().__init__()
        print("i am a CatDog!")

catdog = CatDog()
catdog.some_value
```

```
i am an animal
i am a cat
i am a dog
i am a CatDog!
```

Out[38]: 'dog'

Q: А что с методами?

A: Всё то же, что и с атрибутами!

In [39]:
```
cat.speak() # переопределено
dog.speak() # не переопределено
```

```
meoooow

---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
<ipython-input-39-17e611034cbb> in <module>
      1 cat.speak() # переопределено
----> 2 dog.speak() # не переопределено

<ipython-input-32-c4cade189536> in speak(self)
      9     @abstractmethod
     10     def speak(self):
---> 11         raise NotImplementedError('i don\'t know how to speak')
     12
     13

NotImplementedError: i don't know how to speak
```

In [40]:
```
catdog.speak()
```

```
meoooow
```

Приватность?

In [41]:
```
class VeryPrivateDataHolder:
    _secret = 1
    __very_secret = 2

    def get_very_secret(self):
        return __very_secret
```

In [42]:
```
obj = VeryPrivateDataHolder()
print(obj._secret)
print(obj.get_very_secret())
print(obj.__very_secret)
```

```
1

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-42-248090a0d84b> in <module>
      1 obj = VeryPrivateDataHolder()
      2 print(obj._secret)
----> 3 print(obj.get_very_secret())
      4 print(obj.__very_secret)

<ipython-input-41-069447e9554a> in get_very_secret(self)
      4
      5     def get_very_secret(self):
----> 6         return __very_secret

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

Q: То есть, в питоне всё-таки есть приватность?

A: Ну...

In [43]:
```
obj._VeryPrivateDataHolder__very_secret  # а так вообще никогда не делайте, осо
```

Out[43]:
```
2
```

In [44]:
```
obj._VeryPrivateDataHolder__very_secret = 'new secret'
obj._VeryPrivateDataHolder__very_secret
```

Out[44]:
```
'new secret'
```

Еще полезные декораторы

In [45]:
```
from abc import abstractmethod


class Student:
    _expected_sleep_hours = 7

    @abstractmethod
    def say_hello():
        pass


class MiptStudent(Student):
    _expected_sleep_hours = 3

    @classmethod
    def add_innovative_subject(cls):
        cls._expected_sleep_hours -= 1

    @staticmethod
    def say_hello():
        print("hello")

    @property
    def sleep_hours(self):
        return self._expected_sleep_hours

    @sleep_hours.setter
    def sleep_hours(self, value):
        self._expected_sleep_hours = value
```

In [46]:
```
vasya = MiptStudent()

vasya.sleep_hours
vasya.sleep_hours = 5
vasya.sleep_hours
vasya.add_innovative_subject()
vasya.sleep_hours

MiptStudent().sleep_hours

MiptStudent().say_hello()
```

Out[46]: 3

Out[46]: 5

Out[46]: 5

Out[46]: 2

hello

Генераторы и итераторы: повторение с новой точки зрения

В теории всё выглядит как-то так:

1. Итератор -- это объект, у которого есть методы `iter` и `next` .
2. Генератор -- это объект, возвращаемый из функции. Например, с помощью `yield` . Это упрощает создание итераторов. Ну а еще у него есть некоторый дополнительный функционал
3. Каждый генератор является итератором (неявно реализует интерфейс итератора). Обратное, конечно, неверно.

На практике всё, к счастью, выглядит несколько понятнее. Ниже -- типичный итератор, вид "из-под капота":

In [47]:
```
class my_range_iterator:
    def __init__(self, n_max):
        self.i = 0
        self.n_max = n_max

    def __iter__(self):
        # да, он почти всегда выглядит именно так
        # потому что у генераторов тоже есть такой метод, который возвращает со
        return self

    def __next__(self):
        if self.i < self.n_max:
            i = self.i
            self.i += 1
            return i
        else:
            # специальное исключение, которое означает "элементы кончились!"
            # впрочем, может никогда и не бросаться
            raise StopIteration()
```

In [48]:
```
iterator_obj = my_range_iterator(3)
print(iterator_obj)
print(next(iterator_obj))
print(iterator_obj.__next__())
print(iterator_obj.__next__())
print(iterator_obj.__next__())
print(iterator_obj.__next__())
```

```
<__main__.my_range_iterator object at 0x7f120ff8f910>
0
1
2

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-48-7b023504f49a> in <module>
      4 print(iterator_obj.__next__())
      5 print(iterator_obj.__next__())
----> 6 print(iterator_obj.__next__())
      7 print(iterator_obj.__next__())

<ipython-input-47-bd6c3cb6df19> in __next__(self)
     17             # специальное исключение, которое означает "элементы кончил
ись!"
     18             # впрочем, может никогда и не бросаться
---> 19             raise StopIteration()

StopIteration:
```

Q: И что, чтобы им пользоваться, надо ловить исключения?

A: Конечно, нет! Это non-pythonic way

In [49]:
```
iterator_obj = my_range_iterator(3)
print(type(iterator_obj))
for x in iterator_obj:
    print(x)
```

```
<class '__main__.my_range_iterator'>
0
1
2
```

In [50]:
```
for x in iterator_obj:
    print(x)
```

Q: Повторно использовать нельзя?!

A: Объект итератора, как можно понять из кода, хранит своё состояние. Он уже выдал нам всё, что должен был

In [51]:
```
def my_range_generator(n_max):
    i = 0
    while i < n_max:
        yield i
        i += 1
```

In [52]:
```
generator_obj = my_range_generator(3)
type(generator_obj)
# мы не определяли магических функций итератора, но...
generator_obj.__iter__
generator_obj.__iter__()
generator_obj.__next__
```

Out[52]:
```
generator
```

Out[52]:
```
<method-wrapper '__iter__' of generator object at 0x7f122409f270>
```

Out[52]:
```
<generator object my_range_generator at 0x7f122409f270>
```

Out[52]:
```
<method-wrapper '__next__' of generator object at 0x7f122409f270>
```

In [53]:
```
for x in generator_obj:
    print(x)
```

```
0
1
2
```

In [54]:
```
for x in generator_obj:
    print(x)
```

Q: А чем отличается практическое использование?

A: Как правило, почти ничем

In [55]:
```
print(sum(my_range_generator(5)))
print(sum(my_range_iterator(5)))
```

```
10
10
```

A: Но вообще говоря различия есть...

In [56]:
```
def cumulative_mean_generator(count):
    mean = 0.0
    for i in range(count + 1):
        new_value = yield mean
        mean = (mean * i + new_value) / (i + 1)

numbers = [1, 2, 3, 4, 5]
meaner = cumulative_mean_generator(len(numbers))

next(meaner)
for num in numbers:
    meaner.send(num)  # returns last yield value

meaner.send(2)
```

Out[56]: 0.0

Out[56]: 1.0

Out[56]: 1.5

Out[56]: 2.0

Out[56]: 2.5

Out[56]: 3.0

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-56-a0940987667a> in <module>
     12     meaner.send(num)  # returns last yield value
     13
---> 14 meaner.send(2)

StopIteration:
```

А еще `next == send(None)` у генераторов

In [57]:
```
def printer_generator(count):
    for i in range(count):
        value = yield i
        print(value)
r = printer_generator(3)
_ = r.send(None)
_ = r.send(None)
_ = r.send(2)
_ = r.send(3)
```

```
None
2
3

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-57-ba4892393026> in <module>
      7 _ = r.send(None)
      8 _ = r.send(2)
----> 9 _ = r.send(3)

StopIteration:
```

In [58]:
```
r = printer_generator(3)
r.send(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-047eaaeb6770> in <module>
      1 r = printer_generator(3)
----> 2 r.send(10)

TypeError: can't send non-None value to a just-started generator
```

In [59]:
```
print([x for x in dir(printer_generator(1)) if not x.startswith('_')])
```

```
['close', 'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom', 'send', 'throw']
```

In [60]:
```
r = printer_generator(4)
next(r)
r.close()
r.send(2)
```

Out[60]:
```
0

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-60-fa9a5d5d1755> in <module>
      2 next(r)
      3 r.close()
----> 4 r.send(2)

StopIteration:
```

В следующей серии: magic methods. Будет много магии!

In [61]:
```
class MyClass:
    __slots__ = ["a", "b"]
```

In [62]:
```
obj = MyClass()
obj.a = 5
obj.b = 7
obj.c = 4
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-62-e97573f9a967> in <module>
      2 obj.a = 5
      3 obj.b = 7
----> 4 obj.c = 4

AttributeError: 'MyClass' object has no attribute 'c'
```

In [63]:
```
print(MyClass.__dict__)
print(MyLittleClass.__dict__)
```

```
{'__module__': '__main__', '__slots__': ['a', 'b'], 'a': <member 'a' of 'MyClas
s' objects>, 'b': <member 'b' of 'MyClass' objects>, '__doc__': None}
{'__module__': '__main__', 'color': 'red', 'set_color': <function MyLittleClas
s.set_color at 0x7f12248df820>, '__dict__': <attribute '__dict__' of 'MyLittleC
lass' objects>, '__weakref__': <attribute '__weakref__' of 'MyLittleClass' obje
cts>, '__doc__': None}
```