Лекция 8

Объектно-ориентированное программирование

Создание классов и объектов

Создание классов

In [21]:
```
# Объявление класса Car, наследника класса object (общего родителя для всех классов в Python)
class Car(object):
    """Информаци о классе"""
    pass # В блоке класса объявляются функции
```

In [22]:
```
Car?
```

In [23]:
```
# Старый формат объявления класса (лучше его не использовать):
class OldCar:
    pass
```

В Python для именования классов принято использовать CamalCase. Методы класса, атрибуты и объекты именуются с маленькой буквы через подчеркивание.

In [24]:
```
class Car(object):
    """Базовый класс для автомобилей"""
    def __init__(self, x): # конструктор класса, используется для инициализации нового объекта
        self.x = x # создаем аттрибут класса

    # метод класса; все методы класса должны в качестве первого атрибута иметь переменную self,
    # в которую автоматически передается ссылка на текущий объект
    def is_near(self, x2):
        return abs(self.x - x2) < 2.0 # self.x - обращение к атрибуту класса
```

Создание объектов

In [25]:
```
# создаем объект, при создании передаем параметр конструктора:
c_1 = Car(3.1)
```

In [26]:
```
type(c_1)
```

Out[26]:
```
__main__.Car
```

In [27]:
```
c_1.x # получаем значение аттрибута
```

Out[27]:
```
3.1
```

In [28]:
```
c_1.x = 11.2 # изменяем значение аттрибута
c_1.x
```

Out[28]:
```
11.2
```

In [29]:
```
c_2 = Car(7.1)
```

In [30]:
```
c_1.is_near(c_2.x) # вызываем метод объекта
```

Out[30]:
```
False
```

In [31]:
```
c_2.x = 10.2
```

In [32]:
```
c_1.is_near(c_2.x)
```

Out[32]:
```
True
```

Фактически в Python атрибуты хранятся в словаре принадлежащем конкретному объекту:

In [33]:
```
c_1.__dict__
```

Out[33]:
```
{'x': 11.2}
```

In [34]:
```
c_2.__dict__
```

Out[34]:
```
{'x': 10.2}
```

In [35]:
```
# При желании к атрибуту объекта можно обратится так:
c_2.__dict__['x']
```

Out[35]:
```
10.2
```

In [36]:
```
# Из-за того, что в Python атрибуты по сути это значения в словаре
# в каждом конкретном экземпляре класса можно динамически добавлять новые атрибуты, например:
c_2.speed = 88.5
```

In [37]:
```
c_2.speed
```

Out[37]:
```
88.5
```

In [38]:
```
c_2.__dict__
```

Out[38]:
```
{'speed': 88.5, 'x': 10.2}
```

In [39]:
```
# При этом набор атрибутов объекта c_1 не изменился:
c_1.__dict__
```

Out[39]:
```
{'x': 11.2}
```

Наследование и полиморфизм

In [40]:
```
# Класс, наследующий у Car
class CargoCar(Car):
    def __init__(self, x, max_load, load):
        self.x = x
        self.max_load = max_load
        self.load = load

    def is_overloaded(self):
        return self.load > self.max_load
```

In [41]:
```
cc_1 = CargoCar(6.0, 10, 2)
```

In [42]:
```
cc_1.is_near(7.9) # метод унаследован у Car
```

Out[42]:
```
True
```

In [43]:
```
cc_1.is_overloaded()
```

Out[43]:
```
False
```

In [44]:
```
cars1 = [c_1, cc_1, c_2]
```

In [45]:
```
for c in cars1:
    print(c.is_near(8.2))
```

```
False
False
False
```

In [46]:
```
# Класс, наследующий у CargoCar
class CargoCarWithTrailer(CargoCar):
    def __init__(self, x, max_load, load, trailer_length):
        super().__init__(x, max_load, load) # способ использовать реализацию конструктора базового класса
        self.trailer_length = trailer_length

    def is_near(self, x2): # перегруженный метод
        return self.x-self.trailer_length-2.0 < x2 < self.x+2.0

    def is_near_old(self, x2):
        return super().is_near(x2) # способ обратиться к реализации функции в родительском классе
```

In [47]:
```
ccwt_1 = CargoCarWithTrailer(6.0, 10, 2, trailer_length=3)
```

In [48]:
```
cc_1.x, ccwt_1.x
```

Out[48]:
```
(6.0, 6.0)
```

In [49]:
```
# демонстрация полиморфизма:
cc_1.is_near(3.0), ccwt_1.is_near(3.0)
```

Out[49]:
```
(False, True)
```

Утиная типизация

In [50]:
```
# класс, не входящий в иерархию классов Car
class Man(object):
    def __init__(self, name, position):
        self.name = name
        self.position = position

    def is_near(self, pos2): # метод класса
        return abs(self.position - pos2) < 1.0
```

In [51]:
```
m_1 = Man('Ivan', 9.5)
```

In [52]:
```
different_objects = [c_1, cc_1, c_2, ccwt_1, m_1]
```

In [53]:
```
# Благодаря поддержке утиной типизации в Python объекты, представляющие неродственные классы,
# но реализующие необходимый функционал, могут обрабатываться единообразно:
for ob in different_objects:
    print(ob.is_near(6.2))
```

```
False
True
False
True
False
```

Функция super()

Метод super() в методе возвращает объект который делегирует вызовы методов методам родительского класса. Наиболее наглядно его работу можно увидеть при использовании в конструкторе.

In [54]:
```
class Point2D():
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Point3D(Point2D):
    def __init__(self, x, y, z):
        super().__init__(x, y) # вызываем реализацию __init__ родительского класса
        self.z = z
```

In [55]:
```
p2d_1 = Point3D(0, -1, 1)
p2d_1.__dict__
```

Out[55]:
```
{'x': 0, 'y': -1, 'z': 1}
```

Проверка принадлежности к классу

In [56]:
```
# Функция type() позволяет определить класс объека:
for ob in different_objects:
    print(type(ob), ob.is_near(8.2))
```

```
<class '__main__.Car'> False
<class '__main__.CargoCar'> False
<class '__main__.Car'> False
<class '__main__.CargoCarWithTrailer'> False
<class '__main__.Man'> False
```

In [57]:
```
# проверка принадлежности объека определенному классу:
type(c_1) == Car
```

Out[57]:
```
True
```

In [58]:
```
# другой сопособ получить класс объекта:
c_1.__class__ == Car
```

Out[58]:
```
True
```

In [59]:
```
cc_1.__class__ == Car
```

Out[59]:
```
False
```

In [60]:
```
# чаще всего нужно знать не точную принадлежность к классу,
# а принадлежность к классу или его наследникам
#(так как именно это нужно для корректной работы полиморфизма):
for ob in different_objects:
    print('Класс: {}, является подклассом Car:\
    {}, проверка близости: {}'.format(type(ob),
                                      isinstance(ob, Car), ob.is_near(8.2)))
```

```
Класс: <class '__main__.Car'>, является подклассом Car:     True, проверка близости: False
Класс: <class '__main__.CargoCar'>, является подклассом Car:     True, проверка близости: False
Класс: <class '__main__.Car'>, является подклассом Car:     True, проверка близости: False
Класс: <class '__main__.CargoCarWithTrailer'>, является подклассом Car:     True, проверка близости: False
Класс: <class '__main__.Man'>, является подклассом Car:     False, проверка близости: False
```

In [61]:
```
# Можно проверить, является ли один класс потомком другого:
issubclass(CargoCar, Car), issubclass(Car, CargoCar), issubclass(Car, Man),\
issubclass(Man, Car)
```

Out[61]:
```
(True, False, False, False)
```

Базовые типы Python

Базовые типы Python являются объектами.

In [62]:
```
s1 = 'abc'
```

In [63]:
```
isinstance(s1, str)
```

Out[63]: True

In [64]:
```
issubclass(str, object)
```

Out[64]: True

In [65]:
```
s1.index('b') # вызов метода объекта.
```

Out[65]: 1

Лекция 9

Объектно-ориентированное программирование (продолжение)

Методы классов и статические переменные и методы

In [1]:
```
class Ship(object):
    next_index = 0  # переменная класса (статическая переменная)

    @classmethod
    def generate_next_index(cls): # в classmethod первый обязательный параметр: cls - переменная, ссылающаяся
        index = cls.next_index
        cls.next_index += 1
        return index

    def __init__(self):
        self.index = Ship.generate_next_index()

    @staticmethod
    def is_from_same_epoch(sh1, sh2): # не имеет доступа ни к объекту ни к классу
        return abs(sh1.index - sh2.index) < 10
```

In [2]:
```
s1 = Ship()
s1.index
```

Out[2]: 0

In [3]:
```
fleet = [Ship() for _ in range(15)]
```

In [4]:
```
for sh in fleet:
    print(sh.index)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

In [5]:
```
Ship.next_index # доступ к переменной класса через имя класса
```

Out[5]: 16

In [6]:
```
s1.next_index # доступ к переменной класса через объект
```

Out[6]: 16

In [7]:
```
print([s.next_index for s in fleet])
```

```
[16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
```

In [8]:
```
fleet[0].next_index = 100 # приводит не к изменению в переменной класса, а к появлнию нового атрибута у данного объ
print([s.next_index for s in fleet], Ship.next_index)
```

```
[100, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16] 16
```

In [9]:
```
Ship.next_index = 50 # изменяем значение переменной класса
print([s.next_index for s in fleet], Ship.next_index)
```

```
[100, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50] 50
```

In [10]:
```
Ship.generate_next_index() # доступ к методу класса через имя класса
```

Out[10]:
```
50
```

In [11]:
```
Ship.next_index
```

Out[11]:
```
51
```

Статический метод

In [12]:
```
Ship.is_from_same_epoch(fleet[0], fleet[-1])
```

Out[12]:
```
False
```

In [13]:
```
s1.is_from_same_epoch(s1, fleet[0])
```

Out[13]:
```
True
```

In [14]:
```
for s in fleet:
    print(s, )
```

```
<__main__.Ship object at 0x000001BA994EEA20>
<__main__.Ship object at 0x000001BA994EE748>
<__main__.Ship object at 0x000001BA994EEC18>
<__main__.Ship object at 0x000001BA994EEAC8>
<__main__.Ship object at 0x000001BA994EEBA8>
<__main__.Ship object at 0x000001BA994EEB00>
<__main__.Ship object at 0x000001BA994EEB38>
<__main__.Ship object at 0x000001BA994EEA90>
<__main__.Ship object at 0x000001BA994EE940>
<__main__.Ship object at 0x000001BA994EE978>
<__main__.Ship object at 0x000001BA994EE9B0>
<__main__.Ship object at 0x000001BA994EE080>
<__main__.Ship object at 0x000001BA994EED68>
<__main__.Ship object at 0x000001BA994EECC0>
<__main__.Ship object at 0x000001BA994EEC88>
```

Управление доступом к атрибутам класса

In [66]:
```
cc_1.load, cc_1.max_load, cc_1.is_overloaded()
```

Out[66]:
```
(2, 10, False)
```

In [67]:
```
# нарушаем правила загрузки грузового автомобиля:
cc_1.load = 12
cc_1.is_overloaded()
```

Out[67]:
```
True
```

In [68]:
```
cc_1.max_load = 20
```

In [69]:
```
cc_1.is_overloaded()
```

Out[69]:
```
False
```

In [ ]:
```

```

In [70]:
```
# Класс CargoCar с контролем доступа к значениям загрузки и максимального предела загрузки
class CargoCar2(Car):
    def __init__(self, x, max_load, load):
        self.x = x
        self.__max_load = max_load
        self.__load = load
        assert not(self.is_overloaded()), 'При создании автомобиля превышено ограничение загрузки!'

    def is_overloaded(self):
        return self.__load > self.__max_load

    def get_load(self):
        return self.__load

    def set_load(self, load): # проверка при изменении значения
        assert load < self.__max_load, "Превышен предел загрузки!"
        self.__load = load

    def get_max_load(self): # для max_load есть только возможность получения значения
        return self.__max_load
```

In [71]:
```
cc2_1 = CargoCar2(5.0, 10, 11)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-71-dfc933f47500> in <module>()
----> 1 cc2_1 = CargoCar2(5.0, 10, 11)

<ipython-input-70-665119e3baad> in __init__(self, x, max_load, load)
      5         self.__max_load = max_load
      6         self.__load = load
----> 7         assert not(self.is_overloaded()), 'При создании автомобиля превышено ограничение загрузки!'
      8
      9     def is_overloaded(self):

AssertionError: При создании автомобиля превышено ограничение загрузки!
```

In [72]:
```
cc2_2 = CargoCar2(5.0, 10, 9)
```

In [73]:
```
cc2_2.__load #приватная переменная защищена от доступа извне класса
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-73-3c7b11e844c8> in <module>()
----> 1 cc2_2.__load #приватная переменная защищена от доступа извне класса

AttributeError: 'CargoCar2' object has no attribute '__load'
```

In [74]:
```
cc2_2.get_load()
```

Out[74]:
```
9
```

In [75]:
```
cc2_2.set_load(8)
cc2_2.get_load()
```

Out[75]:
```
8
```

In [76]:
```
cc2_2.set_load(11)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-76-b19a384cbf01> in <module>()
----> 1 cc2_2.set_load(11)

<ipython-input-70-665119e3baad> in set_load(self, load)
     14
     15     def set_load(self, load): # проверка при изменении значения
---> 16         assert load < self.__max_load, "Превышен предел загрузки!"
     17         self.__load = load
     18

AssertionError: Превышен предел загрузки!
```

In [77]:
```
# Класс CargoCar с контролем доступа к значениям, выполненным в стиле Python
class CargoCar3(Car):
    def __init__(self, x, max_load, load):
        self.x = x
        self.__max_load = max_load
        self.__load = load
        assert not(self.is_overloaded()), 'При создании автомобиля превышено ограничение загрузки!'

    def is_overloaded(self):
        return self.__load > self.__max_load

    @property # Декоратор функции, оформляющий функцию как функцию доступа
    def load(self):
        return self.__load

    @load.setter # Декоратор функции, оформляющий функцию как функцию-сеттер
    def load(self, val): # проверка при изменении значения
        assert val < self.__max_load, "Превышен предел загрузки!"
        self.__load = val

    # при необходимости, есть декоратор вида: @load.deletter

    @property
    def max_load(self): # для max_load есть только возможность получения значения
        return self.__max_load
```

In [78]:
```
cc3_1 = CargoCar3(5.0, 10, 9)
```

In [79]:
```
cc3_1.__load
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-79-de4f37b54fa5> in <module>()
----> 1 cc3_1.__load

AttributeError: 'CargoCar3' object has no attribute '__load'
```

In [80]:
```
cc3_1.load
```

Out[80]: 9

In [81]:
```
cc3_1.load = 8
cc3_1.load
```

Out[81]: 8

In [82]:
```
cc3_1.load = 11
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-82-dfdfbf268fb6> in <module>()
----> 1 cc3_1.load = 11

<ipython-input-77-1b55e3ef66e4> in load(self, val)
     16     @load.setter # Декоратор функции, оформляющий функцию как функцию-сеттер
     17     def load(self, val): # проверка при изменении значения
---> 18         assert val < self.__max_load, "Превышен предел загрузки!"
     19         self.__load = val
     20

AssertionError: Превышен предел загрузки!
```

In [83]:
```
cc3_1.max_load
```

Out[83]: 10

In [84]:
```
cc3_1.max_load = 7
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-84-c80a9f34fe6d> in <module>()
----> 1 cc3_1.max_load = 7

AttributeError: can't set attribute
```

Динамические операции с атрибутами и интроспекция

In [85]:
```
# создаем объект, при создании передаем параметр конструктора:
ob_1 = Car(3.1)
ob_2 = Car(4.1)
ob_3 = CargoCar3(5.0, 10, 9)
my_objects = [ob_1, ob_2, ob_3]
```

In [86]:
```
ob_1.x
```

Out[86]: 3.1

In [87]:
```
ob_1.length
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-87-df02834c5008> in <module>()
----> 1 ob_1.length

AttributeError: 'Car' object has no attribute 'length'
```

In [88]:
```
# присваиваем объекту значение для нового атрибута
ob_1.length = 11
```

In [89]:
```
ob_1.length
```

Out[89]: 11

In [90]:
```
# у других объектов этого же типа данный атрибут отсутствует:
ob_2.length
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-90-c37cfae760eb> in <module>()
      1 # у других объектов этого же типа данный атрибут отсутствует:
----> 2 ob_2.length

AttributeError: 'Car' object has no attribute 'length'
```

In [91]:
```
# атрибут у объекта можно не только создать, но и удалить:
del ob_1.length
```

In [92]:
```
ob_1.length
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-92-df02834c5008> in <module>()
----> 1 ob_1.length

AttributeError: 'Car' object has no attribute 'length'
```

Получать значения атрибутов, задвать значения атрибутов и удалять их можно по их имени, хранящемуся в виде строки при помощи встроенных функций:

getattr()

setattr()

delattr()

In [93]:
```
new_attr = 'number'
for i, o in enumerate(my_objects):
    setattr(o, new_attr, i)
```

In [94]:
```
ob_2.number
```

Out[94]:
```
1
```

In [95]:
```
getattr(ob_2, new_attr)
```

Out[95]:
```
1
```

In [1]:
```
getattr?
```

Данные функции позволяют обращаться к атрибутам, имена которых заранее неизвестны. Это особенно важно для реализации интроспекции. Интроспекция (type introspection) в программировании — возможность в объектно-ориентированных языках определить тип и структуру объекта во время выполнения программы. Эта возможность присуща языкам, позволяющих манипулировать типами объектов как объектами первого класса (first class citizens).

In [96]:
```
dir(ob_3)
```

Out[96]:
```
['_CargoCar3__load',
 '_CargoCar3__max_load',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'is_near',
 'is_overloaded',
 'load',
 'max_load',
 'number',
 'x']
```

In [1]:
```
?dir
```

In [97]:
```
# получаем значения и тип всех незащищенных переменных объекта:
for v_name in dir(ob_3):
    if v_name[0] == '_':
        continue
    attr = getattr(ob_3, v_name)
    print(v_name, attr, type(attr))
```

```
is_near <bound method Car.is_near of <__main__.CargoCar3 object at 0x00000053BFA58240>> <class 'metho
d'>
is_overloaded <bound method CargoCar3.is_overloaded of <__main__.CargoCar3 object at 0x00000053BFA5824
0>> <class 'method'>
load 9 <class 'int'>
max_load 10 <class 'int'>
number 2 <class 'int'>
x 5.0 <class 'float'>
```

In [98]:
```
# получаем значения и тип всех незащищенных переменных объекта:
import types

for v_name in dir(ob_3):
    if v_name[0] == '_':
        continue
    attr = getattr(ob_3, v_name)
    attr_t = type(attr)
    if attr_t is types.MethodType:
        print(v_name, '(method)', attr_t)
    else:
        print(v_name, attr, attr_t)
```

```
is_near (method) <class 'method'>
is_overloaded (method) <class 'method'>
load 9 <class 'int'>
max_load 10 <class 'int'>
number 2 <class 'int'>
x 5.0 <class 'float'>
```

In [99]:
```
# Возвращает только аттрибуты объекта для которых можно и получить и задать значения:
vars(ob_3)
```

Out[99]:
```
{'_CargoCar3__load': 9, '_CargoCar3__max_load': 10, 'number': 2, 'x': 5.0}
```

In [100]:
```
# наличие атрибута можно проверить с помощью функции hasattr():
for o in my_objects:
    if hasattr(o, 'load'):
        print(o.number, o.load)
```

```
2 9
```

Встроенные функции для выполнения задач объектно-ориентированного программирования:

http://python-reference.readthedocs.io/en/latest/docs/functions/#object-oriented-functions (http://python-reference.readthedocs.io/en/latest/docs/functions/#object-oriented-functions)

https://docs.python.org/3/library/functions.html (https://docs.python.org/3/library/functions.html)

Специальные методы

*_repr_*(self) и *_str_*(self) - служат для преобразования объекта в строку. Метод *_repr_*() вызывается при выводе в интерактивной оболочке, а также при использовании функции repr(). Метод *_str_*() вызывается при выводе с помощью функции print(), а также при использовании функции str(). Если метод *_str_*() отсутствует, то будет вызван метод *_repr_*(). В качестве значения методы *_repr_*() и *_str_*() должны возвращать строку. Причем, значение возвращаемое *_repr_*() по возможности должно возврващать строку имеющую вид конструктора аналогичного объекта. Т.е. должно быть истинно выражание: eval(repr(obj)) == obj.

In [101]:
```
eval('[11, 22]+[33]')
```

Out[101]:
```
[11, 22, 33]
```

In [102]:
```
for s in fleet:
    print(s)
```

```
<__main__.Ship object at 0x00000053BF96F6D8>
<__main__.Ship object at 0x00000053BF96F4E0>
<__main__.Ship object at 0x00000053BF96F710>
<__main__.Ship object at 0x00000053BF96F780>
<__main__.Ship object at 0x00000053BF96F7F0>
<__main__.Ship object at 0x00000053BF96F828>
<__main__.Ship object at 0x00000053BF96F860>
<__main__.Ship object at 0x00000053BF96F898>
<__main__.Ship object at 0x00000053BF96F8D0>
<__main__.Ship object at 0x00000053BF96F908>
<__main__.Ship object at 0x00000053BF96F940>
<__main__.Ship object at 0x00000053BF96F978>
<__main__.Ship object at 0x00000053BF96F9B0>
<__main__.Ship object at 0x00000053BF96F9E8>
<__main__.Ship object at 0x00000053BF96FA20>
```

In [103]:
```
class ShipS(Ship):
    def __str__(self):
        return f'Ship with index {self.index}'
```

In [104]:
```
fleet2 = [ShipS() for _ in range(5)]
for s in fleet2:
    print(s)
```

```
Ship with index 27
Ship with index 28
Ship with index 29
Ship with index 30
Ship with index 31
```

Когда объект используется в строке формата, вызывается метод *_format__(self, format*spec) объекта с самим объектом и спецификацией формата в виде аргументов. Метод возвращает строку с экземпляром, отформатированным соответствующим образом.

Специальные методы для поддержки преобразования типов:

- *_bool_*(self) - вызывается при использовании функции bool()
- *_int_*(self) - вызывается при преобразовании объекта в целое число с помощью функции int ()
- *_float_*(self) - вызывается при преобразовании объекта в вещественное число с помощью функции float ();
- *_complex_*(self) -вызывается при использовании функции complex ()

Специальные мтоды для поддержки операций сравнения:

- x == y - равно - x.*_eq_*(y)
- x != y - неравно - x.*_ne_*(y)
- x < y - меньше - x.*_lt_*(y)
- x > y - больwe - x.*_gt_*(y)
- x <= y - меньшеилиравно - x.*_le_*(y)
- x >= y - больше или равно - x.*_ge_*(y)
- y in x - проверка на вхождение - x.*_contains_*(y)

Интерпретатор Python будет автоматически подставлять метод *_ne_*() (not equal - не равно), реализующий действие оператора неравенства (!=), если в классе присутствует реализация метода *_eq_*(), но отсутствует реализация метода *_ne_*() .

По умолчанию экземпляры наших собственных классов поддерживают оператор == (который всегда возвращает False) и являются хешируемыми (поэтому они могут использоваться в качестве ключей словаря или добавляться в множества). Но если реализовать специальный метод *_eq_*(), выполняющий корректную проверку на равенство, экземпляры перестанут быть хешируемыми. Это можно исправить, реализовав специальный метод *_hash_*(). Язык Python предоставляет функцию хеширования строк, чисел, фиксированных множеств и других классов.

Специальный метод _*del*_(self) вызывается при уничтожении объекта - по крайней мере в теории. На практике метод _*del*_() может не вызываться никогда, даже при завершении программы.

Материалы для подготовки к следующей лекции:

Прохоренок: Глава 11 "Пользовательские функции"

Саммерфильд: Глава 8 "Улучшенные приемы программирования" (разделы: Улучшенные приемы процедурного программирования; Функциональное программирование

In [ ]: