

# Python. 😊😊 П.

Емельянов А. А.  
login-const@mail.ru

# Что такое класс

- Класс это
  - Способ объединить данные и методы работы с ними за единой абстракцией
  - Объекты, которые порождают другие объекты при вызове
  - Модель объекта
  - Способ самостоятельно создавать типы

# Принципы ООП

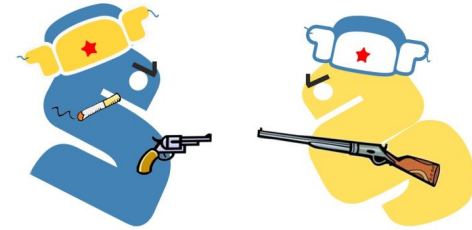
- Объектно-ориентированным может называться язык, построенный с учетом следующих принципов<sup>1</sup>:
  - Все данные представляются объектами.
  - Программа является набором взаимодействующих объектов, посылающих друг другу сообщения.
  - Каждый объект имеет собственную часть памяти и может иметь в составе другие объекты.
  - Каждый объект имеет тип.
  - Объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия)

# Объект vs. Класс

- Класс
  - Описание общих свойств набора объектов.
  - Модель (концепт).
  - Класс – это часть программы.
  - Пример 1: Персона
  - Пример 2: Машина
- Объект
  - Представление свойств отдельного экземпляра.
  - Реализация модели (феномен).
  - Объект – это часть данных и выполнения программы.
  - Навальный, Хованский, Овчинкин
  - Москвич, Вольво, Камаз, Форд Фокус

# Самый простой класс в python

- Класс object: Python 2 vs. Python 3?



```
In [42]: class SomeClass(object):  
  
    pass  
  
instance = SomeClass()  
# Динамическое добавление поля объекту  
instance.name_class = "My first class, o0!"  
  
print(instance.name_class)  
  
# Проверка принадлежности объекта классу  
print(isinstance(instance, SomeClass))  
  
My first class, o0!  
True
```

# Методы экземпляра. self

```
In [45]: class Blogger(object):  
  
    def tweet(self):  
        print("Hate all.")  
  
    pass  
  
b = Blogger()  
# Вызов метода экземпляра  
b.tweet()  
# Вызов метода класса  
Blogger.tweet(b)
```

Hate all.

Hate all.

# Плохой класс Blogger 😞

```
In [46]: class Blogger(object):  
  
    def print_last_blog(self):  
        print(self.last_blog)  
  
    def like(self):  
        self.likes += 1  
  
    def dislike(self):  
        self.dislikes += 1  
  
    pass
```

# Хороший класс Blogger 😊

- `__init__` – magic function

```
In [ ]: class Blogger(object):  
  
    def __init__(self, name, last_blog):  
        self.name = name  
        self.last_blog = last_blog  
        self.likes = 0  
        self.dislikes = 0  
  
    def print_last_blog(self):  
        print(self.last_blog)  
  
    def like(self):  
        self.likes += 1  
  
    def dislike(self):  
        self.dislikes += 1  
  
    pass
```



# Хороший класс Blogger 😊 2

```
In [58]: import numpy as np

class Blogger(object):

    def __init__(self, name, last_blog):
        self.name = name
        self.last_blog = last_blog
        # Соглашение хорошего тона:
        # поля с одним нижним подчеркиванием лучше не менять вне класса
        self._likes = 0
        self._dislikes = 0
        self.__hidden_rate = np.random.random()

    def print_last_blog(self):
        print(self.last_blog)

    def like(self):
        self.likes += 1

    def dislike(self):
        self.dislikes += 1

    def print_hidden_rate(self):
        print(self.__hidden_rate)

    pass
```

```
In [59]: b = Blogger("Blogger", "Hate all!")
```

```
In [61]: b.print_hidden_rate()
```

0.909795709791629

# Атрибуты класса

- mutable vs. immutable

```
In [73]: class Blogger(object):  
  
    # Хранится в классе, а не в объекте  
    comments = []  
  
    def __init__(self, name):  
        self.name = name  
  
    pass
```

```
In [75]: b1, b2 = Blogger("Вася"), Blogger("Кузьма")  
b1.comments.append("Ваш блог безвкусный. Отсутствие стиля, говорит об отсутствии мозгов.")  
b2.comments
```

```
Out[75]: ['Ваш блог безвкусный. Отсутствие стиля, говорит об отсутствии мозгов.']
```

```
In [76]: Blogger.comments.append("Замечательное сообщение!")
```

```
In [77]: b1.comments == b2.comments, b1.comments
```

```
Out[77]: (True,  
          ['Ваш блог безвкусный. Отсутствие стиля, говорит об отсутствии мозгов.',  
           'Замечательное сообщение!'])
```

# Связанные методы

- Связанный метод – метод, вызываемый из объекта.

```
class MyDict(object):  
    def __init__(self, d):  
        self.dict = d  
  
    def max(self):  
        return max(self.dict, key=self.dict.get)  
  
    pass
```

```
MyDict.max
```

```
<function __main__.MyDict.max>
```

```
d = MyDict({"a": 1, "b": 2, "c": -1})
```

```
d.max
```

```
<bound method MyDict.max of <__main__.MyDict object at 0x00000D54DAF8588>>
```

```
d.max()
```

```
'b'
```

# Внутренние переменные класса

```
class Empty(object):  
    """Empty doc"""  
    pass
```

```
print(Empty.__name__)  
print(Empty.__doc__)  
print(Empty.__module__)
```

```
Empty  
None  
__main__
```

---

```
Empty.__dict__
```

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Empty' objects>,  
              '__doc__': None,  
              '__module__': '__main__',  
              '__weakref__': <attribute '__weakref__' of 'Empty' objects>})
```

# Работа с объектом класса – работа с его dict

```
class Blogger(object):  
    # Хранится в классе, а не в объекте  
    comments = []  
    |  
    def __init__(self, name):  
        self.name = name  
  
    def tweet(self, msg):  
        self.comments.append(msg)  
  
    pass
```

```
b = Blogger("Бася")
```


```
{'name': 'Бася'}
```

```
print(Blogger.__dict__)  
print(Blogger.__dict__["comments"])  
print(b.__dict__["name"])  
print(b.__dict__)
```

```
{'__module__': '__main__', 'comments': [], '__init__': <function Blogger.__init__ at 0x000000D54DAEA048>, 'tweet': <function Blogger.tweet at 0x000000D54DAEA2F0>, '__dict__': <attribute '__dict__' of 'Blogger' objects>, '__weakref__': <attribute '__weakref__' of 'Blogger' objects>, '__doc__': None}  
[]  
Бася  
{'name': 'Бася'}
```

# Наследование 1

```
class Human(object):  
  
    def __init__(self, name, age, count_foos):  
        self.name = name  
        self.age = age  
        self.count_foos = count_foos  
        self.count_hands = 2  
  
    pass
```

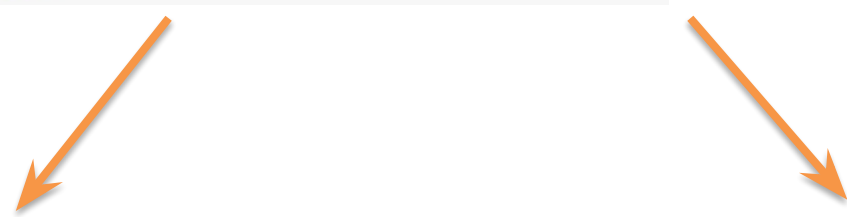
Two orange arrows originate from the 'Human' class box. One arrow points diagonally down and to the left towards the 'Student' class box. The other arrow points diagonally down and to the right towards the 'Teacher' class box.

```
class Student(Human):  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
        self.count_foos = 2  
        self.count_hands = 2  
  
    pass
```

```
class Teacher(Human):  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
        self.count_foos = 4  
        self.count_hands = 4  
  
    pass
```

# Наследование 2

```
class Human(object):  
  
    def __init__(self, name, age, count_foots):  
        self.name = name  
        self.age = age  
        self.count_foots = count_foots  
        self.count_hands = 2  
  
    pass
```




Two orange arrows originate from the bottom of the Human class box. One arrow points diagonally down and to the left towards the Student class box. The other arrow points diagonally down and to the right towards the Teacher class box.

```
class Student(Human):  
  
    def __init__(self, name, age):  
        Human.__init__(self, name, age, 2)  
    pass
```

```
class Teacher(Human):  
  
    def __init__(self, name, age):  
        Human.__init__(self, name, age, 4)  
        self.count_hands = 4  
  
    pass
```

# Наследование 3

```
class Human(object):  
  
    def __init__(self, name, age, count_foots):  
        self.name = name  
        self.age = age  
        self.count_foots = count_foots  
        self.count_hands = 2  
  
    pass
```



Two orange arrows originate from the 'Human' class box. One arrow points diagonally down and to the left towards the 'Student' class box. The other arrow points diagonally down and to the right towards the 'Teacher' class box. This visualizes that both 'Student' and 'Teacher' classes inherit from the 'Human' class.

```
class Student(Human):  
  
    def __init__(self, name, age):  
        super().__init__(name, age, 2)  
    pass
```

```
class Teacher(Human):  
  
    def __init__(self, name, age):  
        super().__init__(name, age, 4)  
        self.count_hands = 4  
  
    pass
```



# Особенности наследования

- Множественное наследование.
- Поиск методов и атрибутов происходит сначала в объекте, потом в его классе а далее в его предках в порядке задаваемом MRO (Method Resolution Order, задаётся алгоритмом C3<sup>1</sup>).
- Порядок поиска можно посмотреть с помощью метода `.mro()` у класса.
- Не любая иерархия является корректной (линеаризуемой).

# Проблемы наследования

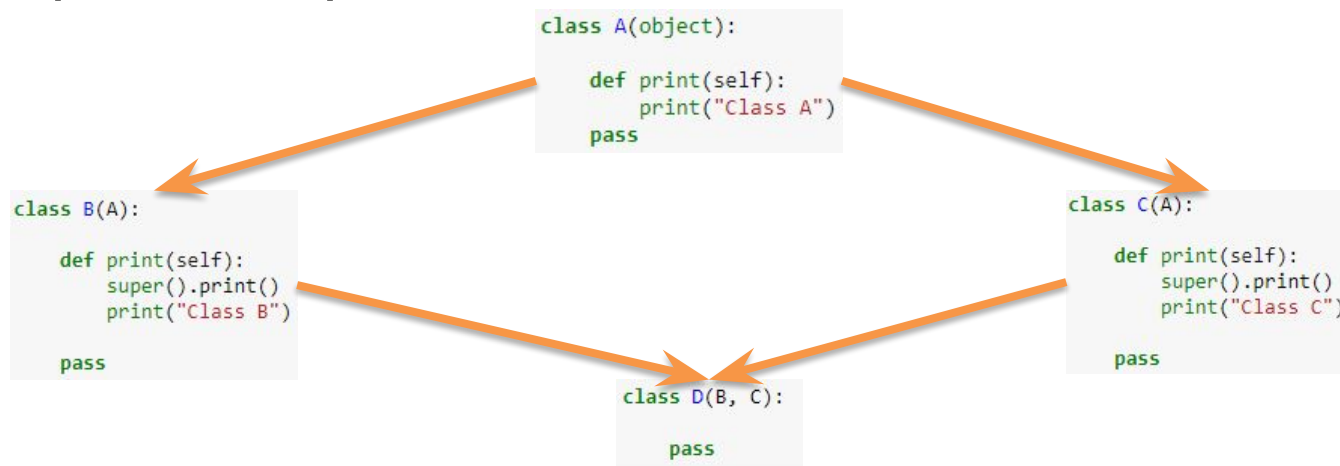
```
class A(object):  
    def print(self):  
        print("Class A")  
    pass  
  
class B(A):  
    def print(self):  
        A.print(self)  
        C.print(self)  
        print("Class B")  
    pass  
  
class C(A):  
    def print(self):  
        A.print(self)  
        print("Class C")  
    pass  
  
class D(B, C):  
    pass
```

```
d = D()  
d.print()
```

```
Class A  
Class A  
Class C  
Class B
```

# Линеаризуемая конструкция

- Проблема ромба<sup>1</sup>.



```
In [159]: d = D()
```

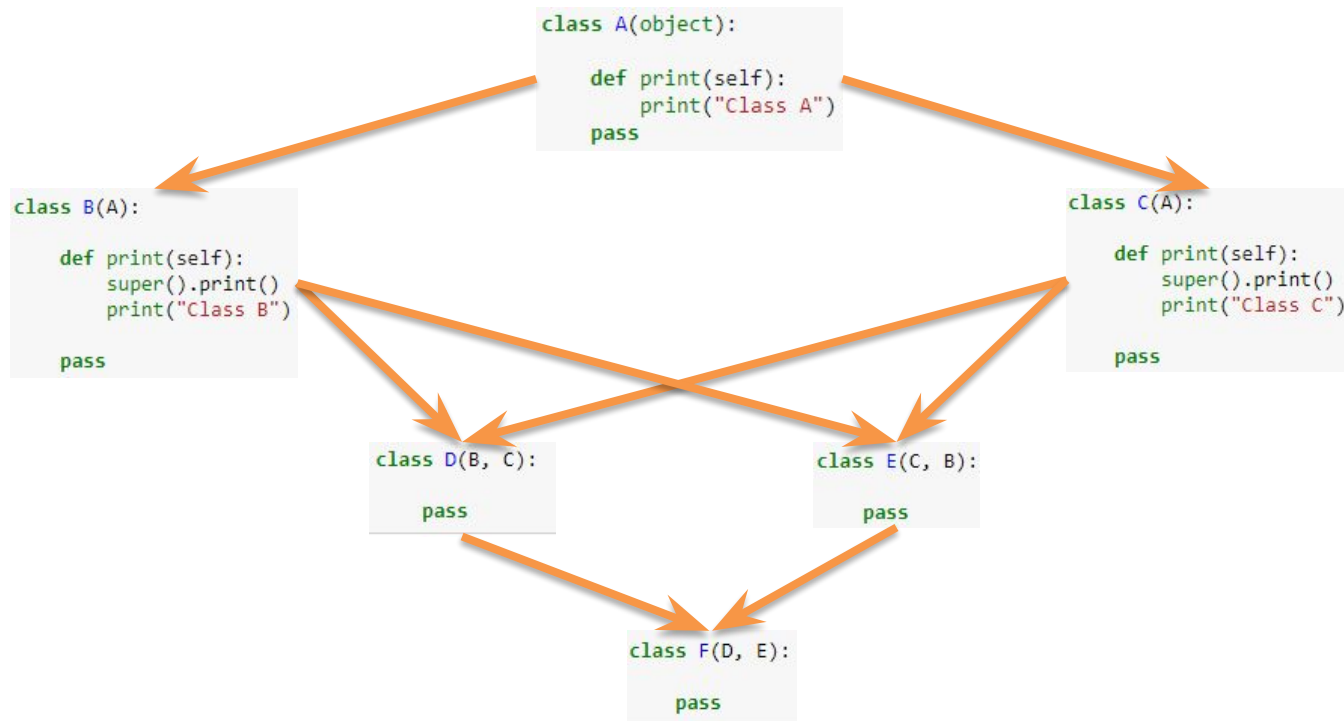
```
In [160]: d.print()
```

```
Class A  
Class C  
Class B
```

```
In [161]: D.mro()
```

```
Out[161]: [__main__.D, __main__.B, __main__.C, __main__.A, object]
```

# Не линеаризуемая конструкция



# Полезные функции

```
isinstance(d, A)
```

True

```
isinstance(d, (A, B))
```

True

```
issubclass(D, A)
```

True

# Магические методы

- Специальные методы, которые вызывает сам интерпретатор при обработке вызовов или других использований класса.



- `__new__(cls, [...])`
  - Это первый метод, который будет вызван при инициализации объекта.
- `__init__(self, [...])`
  - Инициализатор класса.
- `__del__(self)`
  - Деструктор класса. Всегда вызывается при завершении работы интерпретатора.

# Магические методы: сравнение

- `__cmp__(self, other)`
  - Базовый метод сравнения.
- `__eq__(self, other)`
  - Определяет знак равенства `==`.
- `__ne__(self, other)`
  - Определяет поведение оператора неравенства, `!=`.
- `__lt__(self, other)`
  - Определяет поведение оператора меньше, `<`.
- `__gt__(self, other)`
  - Определяет поведение оператора больше, `>`.
- `__le__(self, other)`
  - Определяет поведение оператора меньше или равно, `<=`.
- `__ge__(self, other)`
  - Определяет поведение оператора больше или равно, `>=`.



# Магические методы: унарные операторы и функции

- `__pos__(self)`
  - Определяет поведение для унарного плюса (+some\_object).
- `__neg__(self)`
  - Определяет поведение для отрицания(-some\_object).
- `__abs__(self)`
  - Определяет поведение для встроенной функции `abs()`.
- `__invert__(self)`
  - Определяет поведение для инвертирования оператором `~`.
- `__round__(self, n)`
  - Определяет поведение для встроенной функции `round()`. `n` это число знаков после запятой, до которого округлить.
- `__floor__(self)`
  - Определяет поведение для `math.floor()`, то есть, округления до ближайшего меньшего целого.
- `__ceil__(self)`
  - Определяет поведение для `math.ceil()`, то есть, округления до ближайшего большего целого.
- `__trunc__(self)`
  - Определяет поведение для `math.trunc()`, то есть, обрезания до целого.

# Магические методы: обычные арифметические операторы

- `__add__(self, other)` – Сложение.
- `__sub__(self, other)` - Вычитание.
- `__mul__(self, other)` - Умножение.
- `__floordiv__(self, other)` - Целочисленное деление, оператор `//`.
- `__div__(self, other)` - Деление, оператор `/`.
- `__truediv__(self, other)` - *Правильное* деление.
- `__mod__(self, other)` - Остаток от деления, оператор `%`.
- `__divmod__(self, other)` - Определяет поведение для встроенной функции `divmod()`.
- `__xor__(self, other)` - Двоичный xor, оператор `^`.
- `__pow__` - Возведение в степень, оператор `**`.
- `__lshift__(self, other)` - Двоичный сдвиг влево, оператор `<<`.
- `__rshift__(self, other)` - Двоичный сдвиг вправо, оператор `>>`.
- `__and__(self, other)` - Двоичное И, оператор `&`.
- `__or__(self, other)` - Двоичное ИЛИ, оператор `|`.

# Магические методы: отражённые арифметические операторы

- `__radd__(self, other)` - Отражённое сложение.
- `__rsub__(self, other)` - Отражённое вычитание.
- `__rmul__(self, other)` - Отражённое умножение.
- `__rfloordiv__(self, other)` - Отражённое целочисленное деление, оператор `//`.
- `__rdiv__(self, other)` - Отражённое деление, оператор `/`.
- `__rtruediv__(self, other)` - Отражённое *правильное* деление. `__rmod__(self, other)` - Отражённый остаток от деления, оператор `%`.
- `__rdivmod__(self, other)` - Определяет поведение для встроенной функции `divmod()`, когда вызывается `divmod(other, self)`.
- `__rpow__(self, other)` - Отражённое возведение в степень, оператор `**`.
- `__rlshift__(self, other)` - Отражённый двоичный сдвиг влево, оператор `<<`.
- `__rrshift__(self, other)` - Отражённый двоичный сдвиг вправо, оператор `>>`.
- `__rand__(self, other)` - Отражённое двоичное И, оператор `&`.
- `__ror__(self, other)` - Отражённое двоичное ИЛИ, оператор `|`.
- `__rxor__(self, other)` - Отражённый двоичный xor, оператор `^`.

# Магические методы: составное присваивание

- `__iadd__(self, other)` - Сложение с присваиванием.
- `__isub__(self, other)` - Вычитание с присваиванием.
- `__imul__(self, other)` - Умножение с присваиванием.
- `__ifloordiv__(self, other)` - Целочисленное деление с присваиванием, оператор `//`.
- `__idiv__(self, other)` - Деление с присваиванием, оператор `/`.
- `__itruediv__(self, other)` - *Правильное* деление с присваиванием.
- `__imod__(self, other)` - Остаток от деления с присваиванием, оператор `%`.
- `__ipow__` - Возведение в степень с присваиванием, оператор `**`.
- `__ilshift__(self, other)` - Двоичный сдвиг влево с присваиванием, оператор `<<`.
- `__irshift__(self, other)` - Двоичный сдвиг вправо с присваиванием, оператор `>>`.
- `__iand__(self, other)` - Двоичное И с присваиванием, оператор `&`.
- `__ior__(self, other)` - Двоичное ИЛИ с присваиванием, оператор `|`.
- `__ixor__(self, other)` - Двоичный xor с присваиванием, оператор `^`.

# Магические методы: преобразования типов

- `__int__(self)` - Преобразование типа в `int`.
- `__long__(self)` - Преобразование типа в `long`.
- `__float__(self)` - Преобразование типа в `float`.
- `__complex__(self)` - Преобразование типа в комплексное число.
- `__oct__(self)` - Преобразование типа в восьмеричное число.
- `__hex__(self)` - Преобразование типа в шестнадцатичное число.
- `__index__(self)` - Преобразование типа к `int`, когда объект используется в срезах (выражения вида `[start:stop:step]`).
- `__trunc__(self)` - Вызывается при `math.trunc(self)`.
- `__coerce__(self, other)` - Метод для реализации арифметики с операндами разных типов.

# Магические методы: представление своих классов

- `__str__(self)` - Определяет поведение функции `str()`, вызванной для экземпляра вашего класса.
- `__repr__(self)` - Определяет поведение функции `repr()`, вызываемой для экземпляра вашего класса.
- `__unicode__(self)` - Определяет поведение функции `unicode()`, вызываемой для экземпляра вашего класса. `unicode()` похож на `str()`, но возвращает строку в юникоде.
- `__format__(self, formatstr)` - Определяет поведение, когда экземпляр вашего класса используется в форматировании строк нового стиля.
- `__hash__(self)` - Определяет поведение функции `hash()`, вызываемой для экземпляра вашего класса.
- `__nonzero__(self)` - Определяет поведение функции `bool()`, вызванной для экземпляра вашего класса.
- `__dir__(self)` - Определяет поведение функции `dir()`, вызванной на экземпляре вашего класса.
- `__sizeof__(self)` - Определяет поведение функции `sys.getsizeof()`, вызываемой на экземпляре вашего класса.

# Магические методы: контроль доступа к атрибутам

- `__getattr__(self, name)` - Вы можете определить поведение для случая, когда пользователь пытается обратиться к атрибуту, который не существует (совсем или пока ещё).
- `__setattr__(self, name, value)` - В отличие от `__getattr__`, `__setattr__` решение для инкапсуляции.
- `__delattr__` - Это то же, что и `__setattr__`, но для удаления атрибутов, вместо установки значений.
- `__getattribute__(self, name)` - может использоваться только с классами нового типа (в новых версиях Питона все классы нового типа, а в старых версиях вы можете получить такой класс унаследовавшись от `object`).

# Магические методы: вызываемые объекты

- `__call__(self, [args...])`
  - Позволяет любому экземпляру вашего класса быть вызванным как-будто он функция. Главным образом это означает, что `x()` означает то же, что и `x.__call__()`.



# Магические методы: ограничения атрибутов

- *`__slots__` - создает ограничения на создание новых атрибутов класса.*

# Домашнее задание 2

- Целью этого задания является знакомство с классами в python.
- **Deadline (получение полных баллов): 5.03.2020**
- **Адрес:** login-const@mail.ru

- Задание состоит из трех частей:

## Часть 1

- Реализовать алгоритм передачи данных

## Часть 2

- Написать класс CounterGetter.
- Написать класс Vector

- **Текс условия доступен посмотреть на [git](#)**



Физкек

16 сен 2017 в 18:27

Изи изи, рил ток, синк абаут ит  
~~#любая\_училка\_по\_англу~~



Цитаты преподавателей МФТИ

14 сен 2017 в 23:34

Постарайтесь вникнуть в эти уравнения...  
Ну, а если не получилось, то забейте.  
#Овчинкин