

ООП в Python

Часть I

O

Object

O

Oriented

P

Programming

Абстракция данных

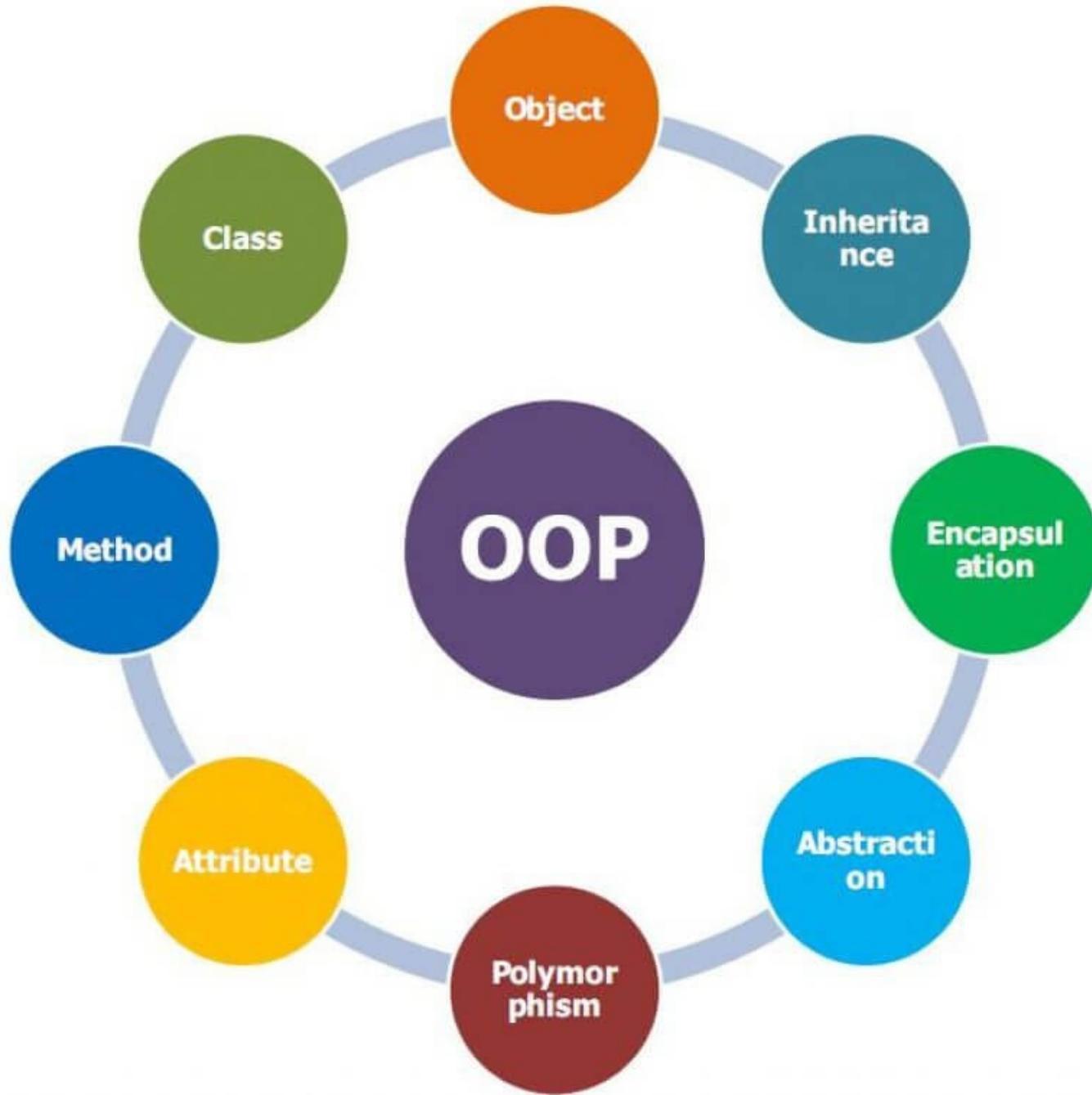
Абстрагирование означает выделение значимых характеристик и исключение из рассмотрения частных и незначимых. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор наиболее общих характеристик объекта, доступных остальной программе.

Пример:

Представьте, что водитель едет в автомобиле по оживлённому участку движения. Понятно, что в этот момент он не будет задумываться о химическом составе краски автомобиля, особенностях взаимодействия шестерён в коробке передач или влияния формы кузова на скорость . Однако, руль, педали, указатель поворота он будет использовать регулярно.

Определение ООП

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.





Основные принципы ООП

- Инкапсуляция - сокрытие реализации.
- Наследование - создание новой сущности на базе уже существующей.
- Полиморфизм - возможность иметь разные формы для одной и той же сущности.
- Абстракция - набор общих характеристик.
- Посылка сообщений - форма связи, взаимодействия между сущностями.
- Переиспользование - повторное использование кода.

Инкапсуляция

- Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя, открыв только то, что необходимо при последующем использовании.
- Цель инкапсуляции – уйти от зависимости внешнего интерфейса класса (то, что могут использовать другие классы) от реализации. Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса.

Наследование

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется предком, базовым или родительским. Новый класс – потомком, наследником или производным классом.

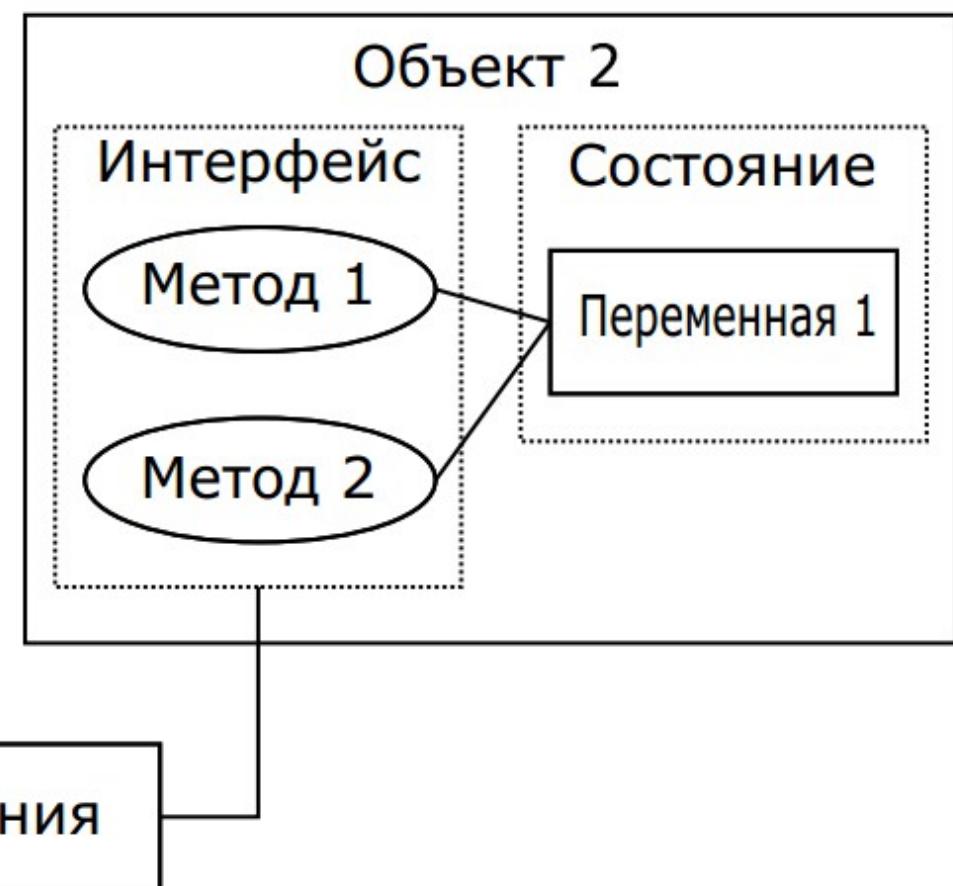
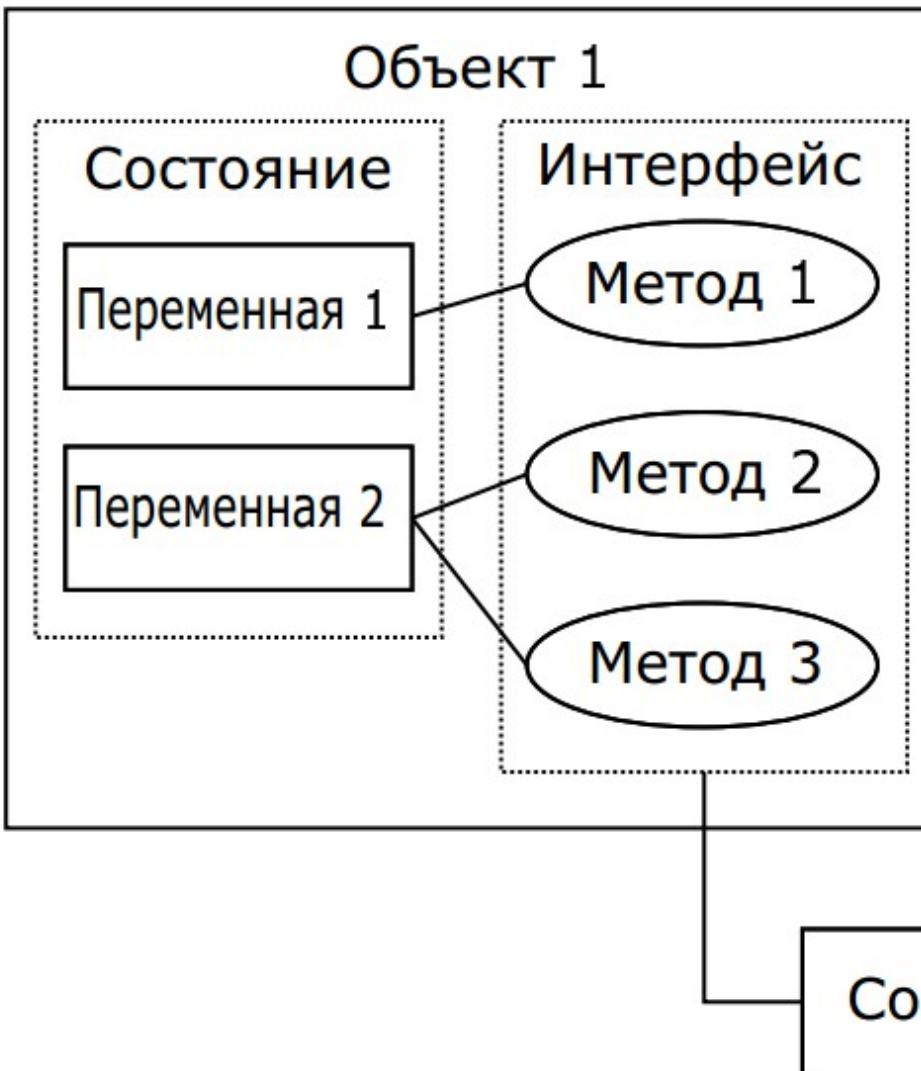
Полиморфизм

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного и того же интерфейса для задания единого набора действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор языка программирования. Отсюда следует ключевая особенность полиморфизма - использование объекта производного класса, вместо объекта базового (потомки могут изменять родительское поведение, даже если обращение к ним будет производиться по ссылке родительского типа).

Посылка сообщений (вызов метода)

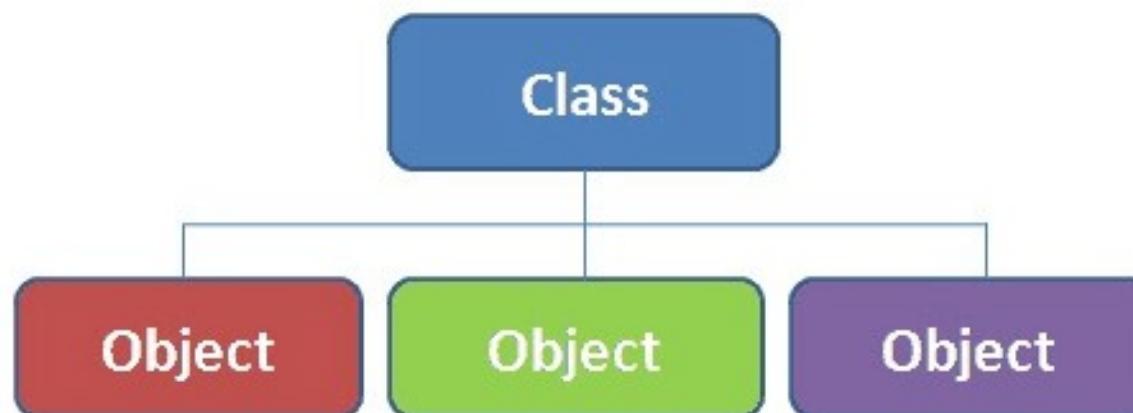
Объекты взаимодействуют, посылая и получая сообщения. Сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия. В ООП посылка сообщения (вызов метода) — это единственный путь передать управление объекту. Если объект должен «отвечать» на это сообщение, то у него должна иметься соответствующий данному сообщению метод. Так же объекты, используя свои методы, могут и сами посыпать сообщения другим объектам.





Понятия ООП

- Класс
- Объект
- Интерфейс



Класс

Класс — универсальный, комплексный **тип данных**, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является **моделью информационной сущности** с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей). В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определённым полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю.

Объект (экземпляр)

Объект - это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом. Каждый объект имеет конкретные значения атрибутов и методы, работающие с этими значениями на основе правил, заданных в классе.

Интерфейс

Интерфейс - это набор методов класса, доступных для использования. Интерфейсом класса будет являться набор всех его публичных методов в совокупности с набором публичных атрибутов. По сути, **интерфейс специфицирует класс**, чётко определяя все возможные действия над ним.

Класс, объект

Когда речь идёт об объектно-ориентированном программировании в Python, первое, что нужно сказать – это то что любые переменные любых типов данных являются объектами, а типы являются классами.

Каждый объект имеет набор атрибутов, к которым можно получить доступ с помощью оператора “.”. Мы уже имеем опыта работы с атрибутами, пример списков:

```
lst = [1, 2, 3]
```

```
lst.append(4)
```

Атрибуты, которые являются функциями, называются методами.

Создание класса CLASS

Создать класс можно с помощью конструкции **class**

```
# Создание пустого класса Animal
class Animal:
    pass
```

Создание объекта класса

Для того чтобы создать объект на основе класса Animal нужно вызвать его как функцию.

```
animal = Animal()
```

Определение отношения объекта

Для определения, к какому классу относится объект, можно вызвать внутренний атрибут объекта `__class__`. Также можно воспользоваться функцией `isinstance` (рекомендуется этот вариант).

```
# Определить класс объекта
print(isinstance(animal, Animal))

print(animal.__class__)
```

```
# Поля объекта
print(dir(animal))
```

Класс с атрибутами

```
class Animal:  
    color = "red"  
    weight = 200  
  
animal = Animal()  
  
  
  
# Поля объекта  
print(dir(animal))  
  
print(animal • color)  
  
print(animal • weight)
```

Изменение атрибутов

```
class Animal:  
    color = "red"  
    weight = 200  
  
a1 = Animal()  
a2 = Animal()  
  
a1.weight = 400  
print(a1.color, a1.weight)  
a2.color = "blue"  
print(a2.color, a2.weight)
```

Создание новых атрибутов внутри объектов

```
# Добавим новый атрибут speed к объектам a1, a2
```

```
class Animal:
```

```
    color = "red"
```

```
    weight = 200
```

```
a1 = Animal()
```

```
a2 = Animal()
```

```
a1.speed = 300
```

```
a2.speed = 400
```

```
print(a1.color, a1.weight, a1.speed) -> red 200 300
```

```
print(a2.color, a2.weight, a2.speed) -> red 200 400
```

Доступ к атрибутам класса

Класс Animal так же является объектом и внутри него так же есть атрибуты, как и внутри объектов, созданных с помощью класса Animal.

```
print(Animal.color, Animal.weight)  
Animal.color = "green"  
Animal.weight = 300  
print(a1.color, a1.weight, a1.speed) green 300 300  
print(a2.color, a2.weight, a2.speed) green 300 400
```

Как вы могли заметить, в объектах изменились значения только тех атрибутов, которые мы не меняли до этого в этих объектах. Это поведение будет объяснено позже.

Методы класса

Для объявления методов внутри класса нужно просто создать функцию внутри класса. Эта функция должна принимать как минимум один аргумент. По общепринятым соглашениям этот аргумент называется **self** и в него передаётся сам объект класса, из которого этот метод и вызывается.

```
class Animal:  
    def set(self, value):  
        self.value = value  
  
    def print(self):  
        print("VALUE:", self.value)  
  
animal = Animal()  
animal.set(20)  
animal.print()  
animal.set("May...")  
animal.print()
```

Конструктор `__init__`

В классах можно описывать так называемые "волшебные" методы, то есть методы, которые имеют заранее прописанный функционал в языке Python. Одним из таких методов является конструктор.

Конструктор – специальный метод, который вызывается сразу же после создания объекта и производит первичные действия. Чтобы создать конструктор в Python – нужно создать функции в классе с именем `__init__`.

```
class Animal:  
    def __init__(self, color, weight):  
        self.color = color  
        self.weight = weight  
  
a1 = Animal("red", 300)    # при создании объекта нужно  
a2 = Animal("green", 200)  # передавать все аргументы, кроме  
                         # self, которые принимаются в  
                         # конструкторе.  
  
a1.color = "blue"  
a2.weight = 500  
print(a1.color, a1.weight)  
print(a2.color, a2.weight)
```

Деструктор `__del__`

Также существует такой "волшебный" метод который называется деструктор. Он вызывается в момент удаления объекта из памяти.

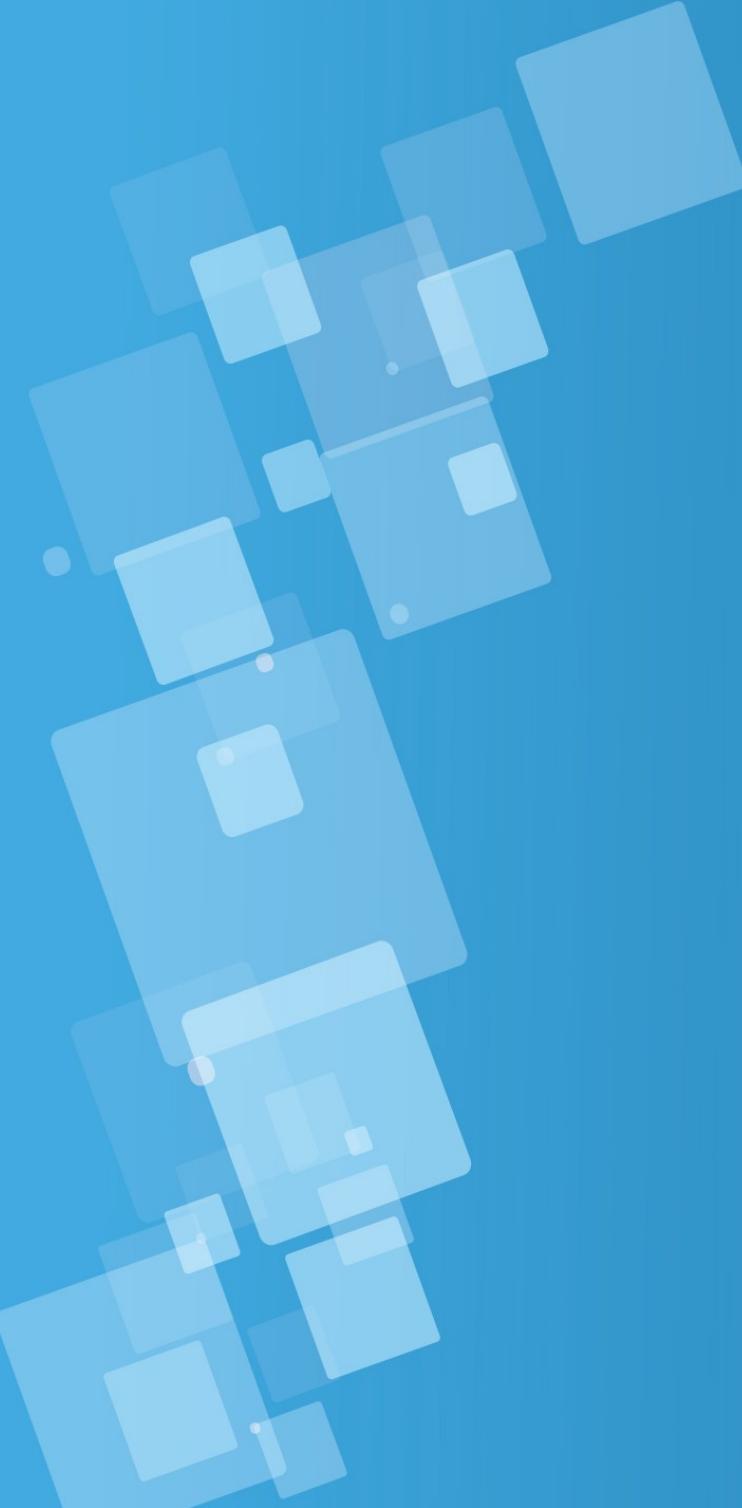
```
class Animal:

    def __init__(self, name):
        self.name = name

    def __del__(self):
        print(self.name, "will deleted.")

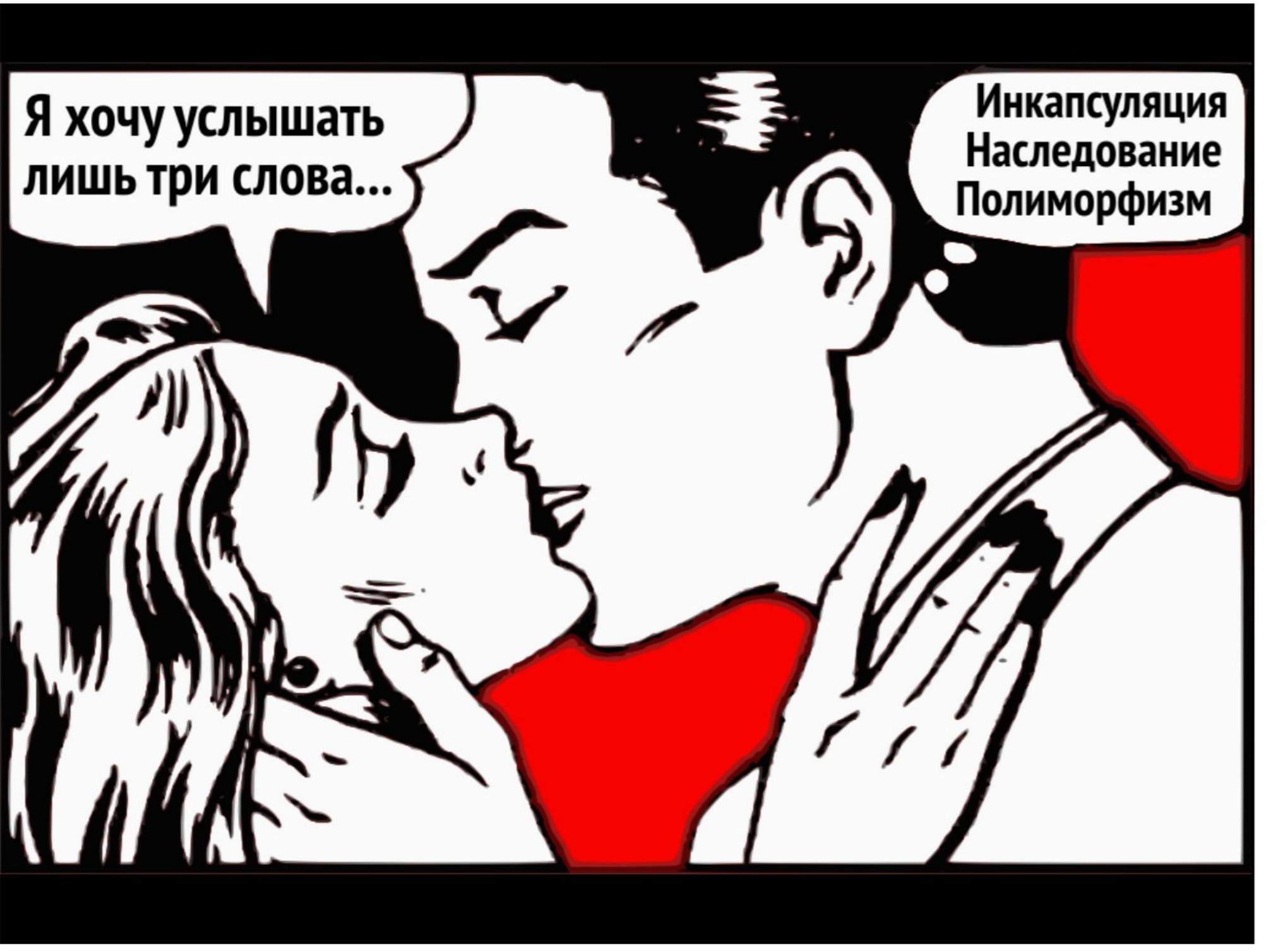
a1 = Animal("Kent")
a2 = Animal("Afon")
a3 = Animal("Zuk")
del a2 # Удаление переменной
```

Сначала на экран вывелась строка об удалении Afon, а потом об удалении остальных объектов. Несмотря на то, что мы явно их не удаляем, они всё равно удаляются при достижении программой конца.



ООП

Часть II



Я хочу услышать
лишь три слова...

Инкапсуляция
Наследование
Полиморфизм

Инакапсуляция

По умолчанию атрибуты в классах являются общедоступными (public), а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его.

```
class Person:
    def __init__(self, name):
        self.name = name      # устанавливаем имя
        self.age = 1           # устанавливаем возраст

    def display_info(self):
        print(f"Имя: {self.name}\tВозраст:
{self.age}")

obj = Person("Pupkin")
tom.age = 50                  # изменяем атрибут age
tom.display_info()            # Имя:Pupkin Возраст: 50
```

Инкапсуляция

Все объекты в Python инкапсулируют внутри себя данные и методы работы с ними, предоставляя публичные интерфейсы для взаимодействия.

Атрибут может быть объявлен **приватным** (private) с помощью **нижнего подчеркивания** перед именем, но настоящего скрытия на самом деле не происходит – все на уровне соглашений.

```
class SomeClass:  
    def __private(self):  
        print("Это внутренний метод объекта")  
  
obj = SomeClass()  
obj.__private() # Это внутренний метод объекта
```

Два нижних подчеркивани (double underscore) - дандер

```
# Если поставить перед именем атрибута два
# подчеркивания, к нему нельзя будет обратиться
# напрямую.

class SomeClass():

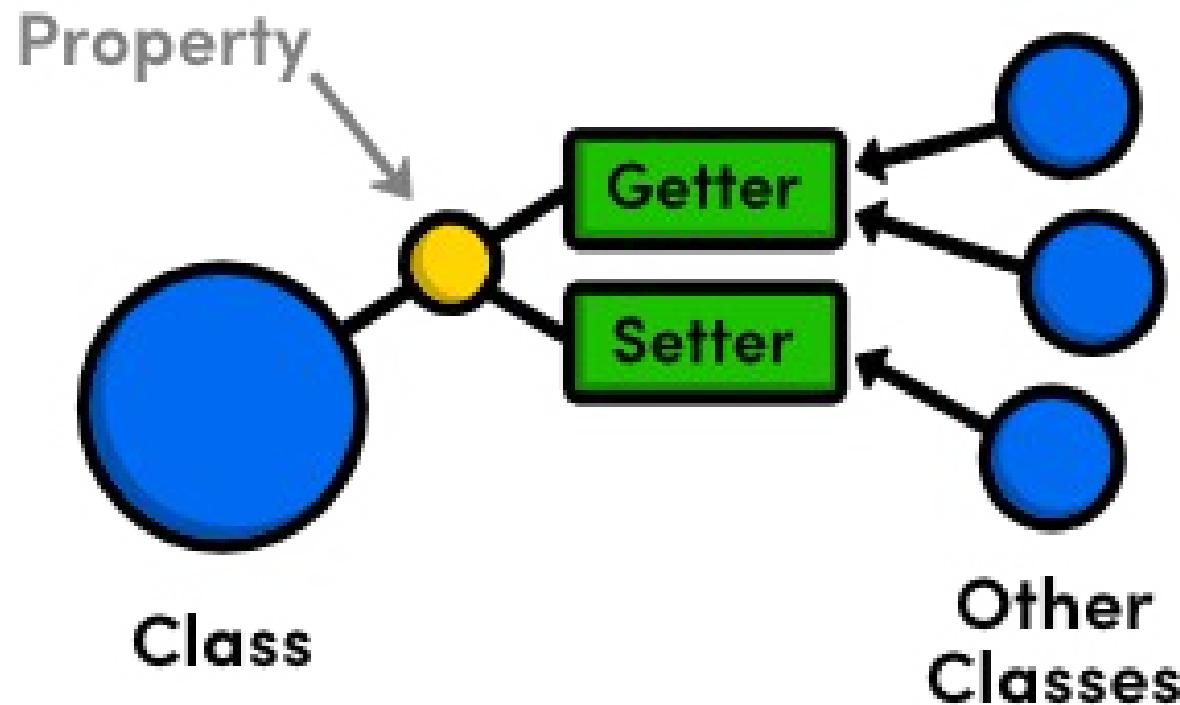
    def __init__(self):
        self.__param = 42 # приватный атрибут

obj = SomeClass()

obj.__param # AttributeError: 'SomeClass' object has
no attribute '__param'

obj._SomeClass__param # - обходной способ
```

Методы доступа к свойствам



```
class Person:
    def __init__(self, name):
        self.__name = name # устанавливаем имя
        self.__age = 1 # устанавливаем возраст

    def set_age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

    def get_age(self):
        return self.__age

    def get_name(self):
        return self.__name

    def display_info(self):
        print(f"Имя: {self.__name}\tВозраст: {self.__age}")

tom = Person("Tom")
tom.display_info() # Имя: Том Возраст: 1
tom.set_age(25)
tom.display_info() # Имя: Том Возраст: 25
```

Встроенные методы

Вместо того чтобы вручную создавать геттеры и сеттеры для каждого атрибута, можно перегрузить встроенные методы

- `__getattr__`
- `__setattr__`
- `__delattr__`

Например, так можно перехватить обращение к свойствам и методам, которых в объекте не существует:

Встроенные методы `__getattr__`

```
#    автоматически вызывается при получении  
несуществующего свойства класса
```

```
class SomeClass():  
    attr1 = 42
```

```
    def __getattr__(self, attr):  
        return attr.upper()
```

```
obj = SomeClass()  
obj.attr1 # 42  
obj.attr2 # ATTR2
```

__getattribute__

Перехватывает все обращения (в том числе и к существующим атрибутам) :

```
class SomeClass():

    attr1 = 42

    def __getattribute__(self, attr):
        return attr.upper()

obj = SomeClass()

obj.attr1 # ATTR1

obj.attr2 # ATTR2
```

Встроенные методы `__setattr__`

#автоматически вызывается при изменении свойства класса;

```
class SomeClass():
    age = 42

    def __setattr__(self, attr, value):
        if attr == 'age':
            print( 'Age, {} !'.format( value ) )
            self.age = value
```

```
obj = SomeClass()
obj.age # 45
obj.age = 100 # Вызовет метод __setattr__
```

```
obj.name = 'Pupkin'    ← Что произойдет при вызове ?
```

__delattr__ удаление атрибута

```
class Car:

    def __init__(self):
        self.speed = 100

    def __delattr__(self, attr):
        self.speed = 42

# Создаем объект
porsche = Car()
print(porsche.speed)
# 100
delattr(porsche, 'speed') ← Удаление атрибута у объекта
print(porsche.speed) → 42
```

Перехват обращение к свойствам

Например, так можно перехватить обращение к свойствам и методам, которых в объекте не существует:

```
class AccessControl:  
    def __setattr__(self, attr, value):  
        if attr == 'age':  
            self.__dict__[attr] = value  
        else:  
            raise AttributeError, attr + ' not allowed'
```

```
X = AccessControl()  
X.age = 40  
X.name = 'pythonlearn'  
AttributeError: name not allowed
```

Если мы используем метод `__setattr__`, все присваивания в нем придется выполнять посредством словаря атрибутов. Используйте `self.__dict__['age'] = x`, а не `self.name = x`:

Декораторы свойств

Для создания свойства-геттера над свойством ставится аннотация **@property**.

Для создания свойства-сеттера над свойством устанавливается аннотация **имя_свойства_геттера.setter**.

```
class Person:
    def __init__(self, name):
        self.__age = 0 # устанавливаем возраст

    @property
    def age(self):
        return self.__age

    #Свойство-сеттер определяется после свойства-геттера.

    @age.setter
    def age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

tom = Person("Том")
tom.age = -100 # Недопустимый возраст
```



Наследование

Одиночное наследование

#Родительский класс помещается в скобки после имени класса. Объект производного класса наследует все свойства родительского.

```
class Tree(object):  
    def __init__(self, kind, height):  
        self.kind = kind  
        self.age = 0  
        self.height = height  
    def grow(self):  
        """ Метод роста """  
        self.age += 1  
  
class FruitTree(Tree):  
    def __init__(self, kind, height):  
        # Необходимо вызвать метод инициализации родителя.  
        super().__init__(kind, height)  
  
    def give_fruits(self):  
        print ("Collected 20kg of {}s".format(self.kind))  
  
f_tree = FruitTree("apple", 0.7)  
f_tree.give_fruits()  
f_tree.grow()
```

Множественное наследование

При множественном наследовании дочерний класс наследует все свойства родительских классов. Синтаксис множественного наследования очень похож на синтаксис обычного наследования.

```
class Horse():
    isHorse = True
class Donkey():
    isDonkey = True
class Mule(Horse, Donkey):
    pass
mule = Mule()
mule.isHorse # True
mule.isDonkey # True
```

Многоуровневое наследование

Мы также можем наследовать класс от уже наследуемого. Это называется многоуровневым наследованием. Оно может иметь сколько угодно уровней.

В многоуровневом наследовании свойства родительского класса и наследуемого от него класса передаются новому наследуемому классу.

```
class Horse():
    isHorse = True
class Donkey(Horse):
    isDonkey = True
class Mule(Donkey):
    pass
mule = Mule()
mule.isHorse # True
mule.isDonkey # True
```

Композиция

Где один класс является полем другого.

```
class Salary:
    def __init__(self, pay):
        self.pay = pay

    def getTotal(self):
        return (self.pay*12)

class Employee:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus
        self.salary = Salary(self.pay)

    def annualSalary(self):
        return "Total: " + str(self.salary.getTotal()) +
self.bonus

employee = Employee(100, 10)
print(employee.annualSalary())
```

Полиморфизм

Все методы в языке изначально **виртуальные**. Это значит, что дочерние классы могут их переопределять и решать одну и ту же задачу разными путями, а конкретная реализация будет выбрана только во время исполнения программы. Такие классы называют полиморфными.

```
class Mammal:  
    def move(self):  
        print('Двигается')
```

```
class Hare(Mammal):  
    def move(self):  
        print('Прыгает')
```

```
animal = Mammal()  
animal.move() # Двигается  
hare = Hare()  
hare.move() # Прыгает
```

```
class Animal:
    def __init__(self, name):
        self.name = name

class Horse(Animal):
    def run(self, distance):
        print(self.name, "run", distance, "meters")

    def sound(self):
        print(self.name, "says: igogo")

class Bird(Animal):
    def fly(self, distance):
        print(self.name, "fly", distance, "meters")

    def sound(self):
        print(self.name, "says:chik-chirik")

class Unicorn( Bird, Horse):
    pass

pegas = Unicorn("Пегас")    ← Вызов констуктора баз. класса
pegas.run(20)
pegas.fly(800)
pegas.sound()               ← Что будет издавать Пегас ?
```

```
class Lection:

    def __init__(self, name):
        self.name = name

    def __del__(self):
        print(self.name, "Продолжение следует.")

obj = lection("lection_17")
del obj
```



Продолжение следует ...