Программирование на Python

лекция 5

Владимир Юрьевич Полищук

Инженерная школа
информационных технологий и
робототехники,
Томский политехнический университет

pvy3@tpu.ru

Содержание

- Объектно-ориентированное программирование (ООП)

ООП: общая картина

- Классы – это основные инструменты объектно-ориентированного программирования (ООП) в языке Python.
- Классы в языке Python создаются с помощью инструкции: инструкции **class**.

Зачем нужны классы?

- Классы – это способ определить новое «что-то», они являются отражением реальных объектов в мире программ.

Два аспекта ООП:

- Наследование
- Композиция

Зачем нужны классы?

- Классы – это программные компоненты на языке Python, точно такие же, как функции и модули.

Классы имеют три важных отличия:

- Множество экземпляров
- Адаптация через наследование
- Перегрузка операторов

ООП

Достоинства и недостатки механизма ООП

Достоинства:

1) Возможность повторного использования кода.
2) Повышение читаемости и гибкости кода.
3) Ускорение поиска ошибок и их исправления.
4) Повышение безопасности проекта.

Достоинства и недостатки механизма ООП

Недостатки:

1) Для реализации взаимосвязи классов необходимо хорошо разбираться в особенностях предметной области, а также четко представлять структуру создаваемого приложения.

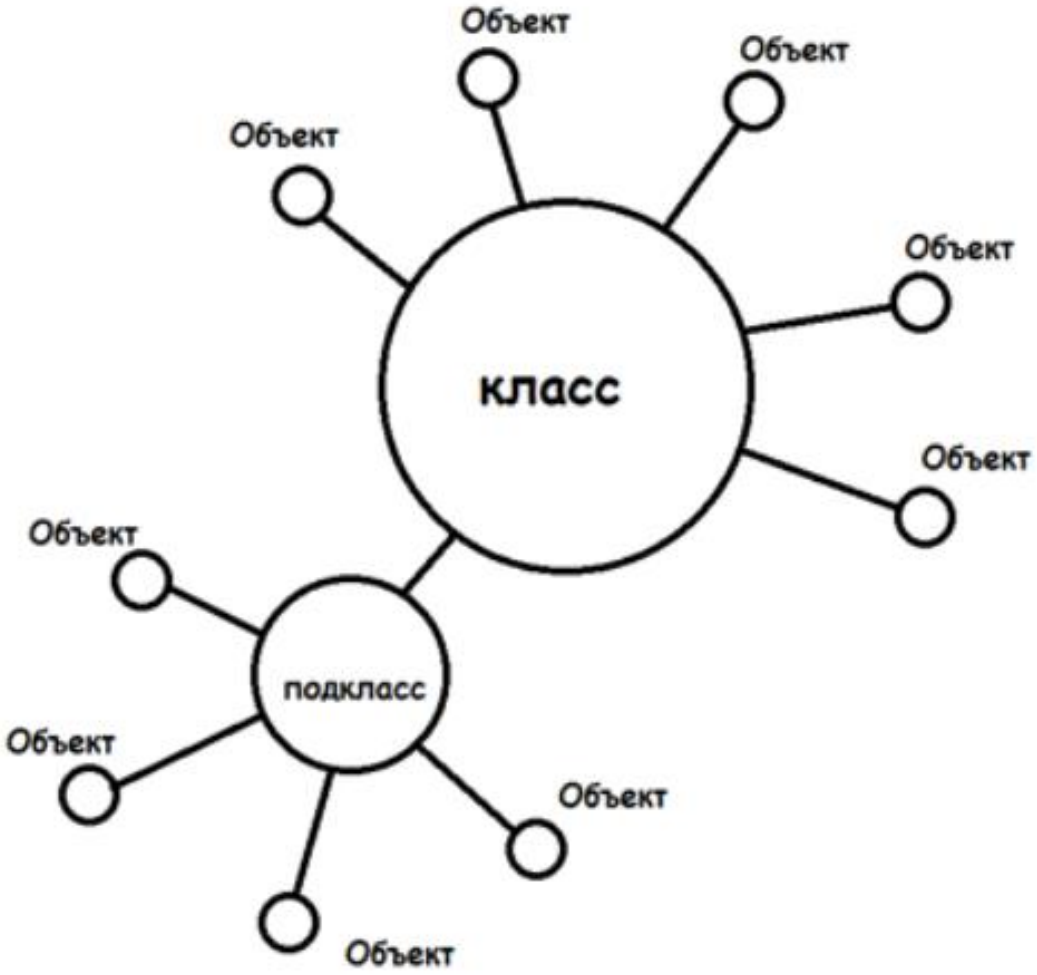
2) Сложность в разбиение проекта на классы. Новичкам может быть тяжело определить для проекта классы-шаблоны.

3) Возможная сложность в модификации проекта. С добавлением в проект новой функциональности придется вносить изменения в структуру классов.

Что же такое ООП

Истоки ООП берут начало с 60-х годов XX века. Однако окончательное формирование основополагающих принципов и популяризацию идеи следует отнести к 80-м годам.

Что же такое ООП



В чем разница

Разницу между программой, написанной с структурном стиле, и объектно-ориентированной программой можно выразить так:

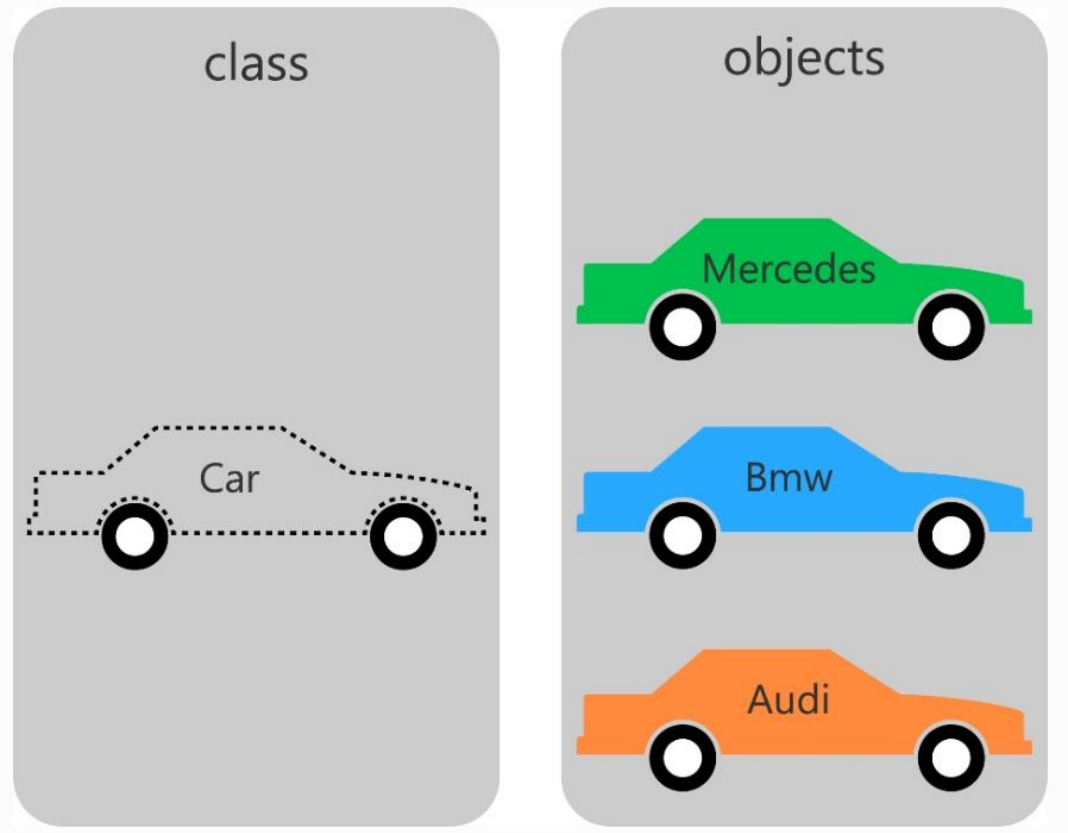
- В первом случае, на первый план выходит логика, понимание последовательности выполнения действий для достижения поставленной цели.
- Во-втором – важнее представить программу, как систему взаимодействующих объектов.

Понятия ООП

Основными понятиями, используемыми в ООП, являются:

- класс;
- объект;
- наследование;
- инкапсуляция;
- полиморфизм.

Что такое **класс** или тип

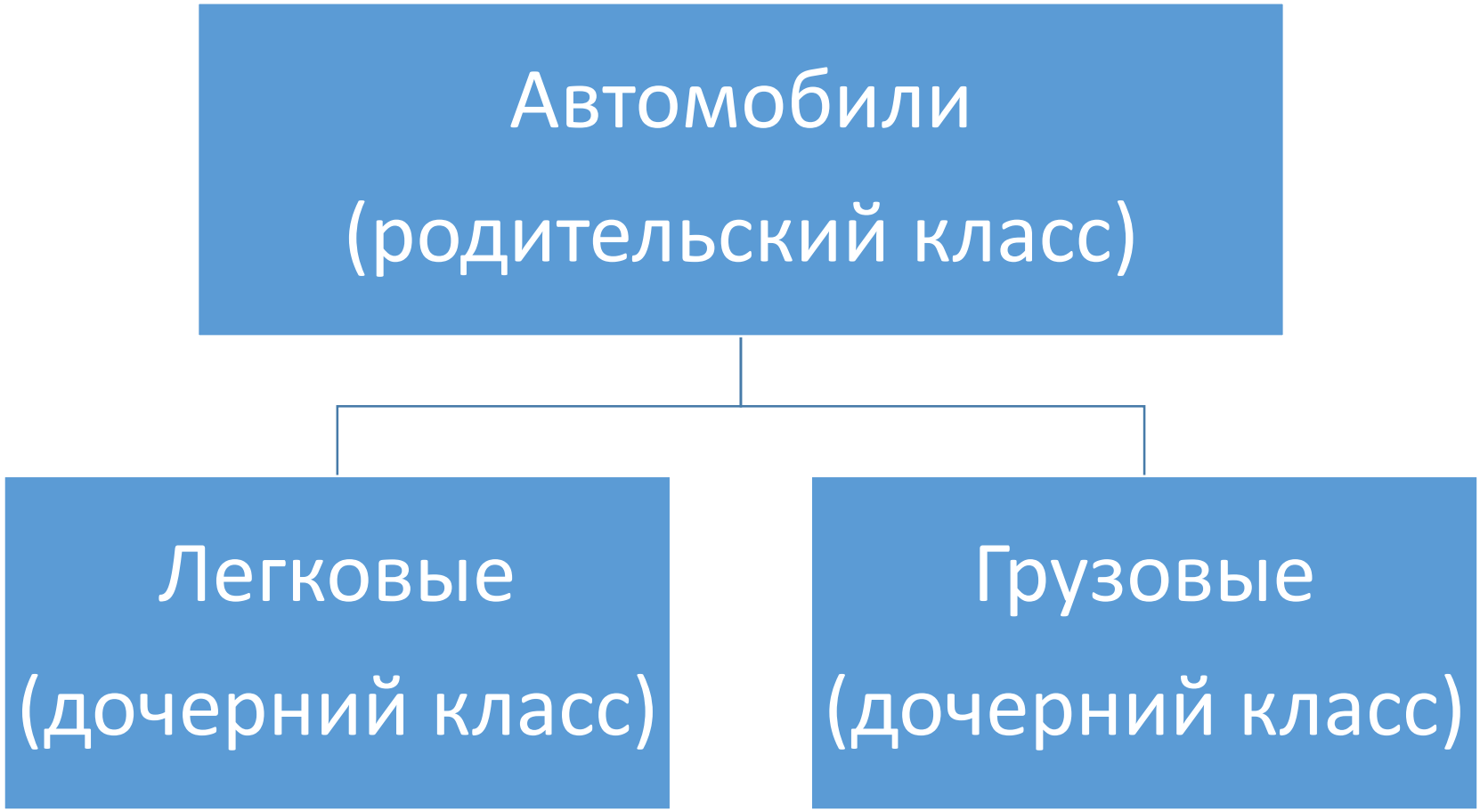


Что такое **класс** или тип

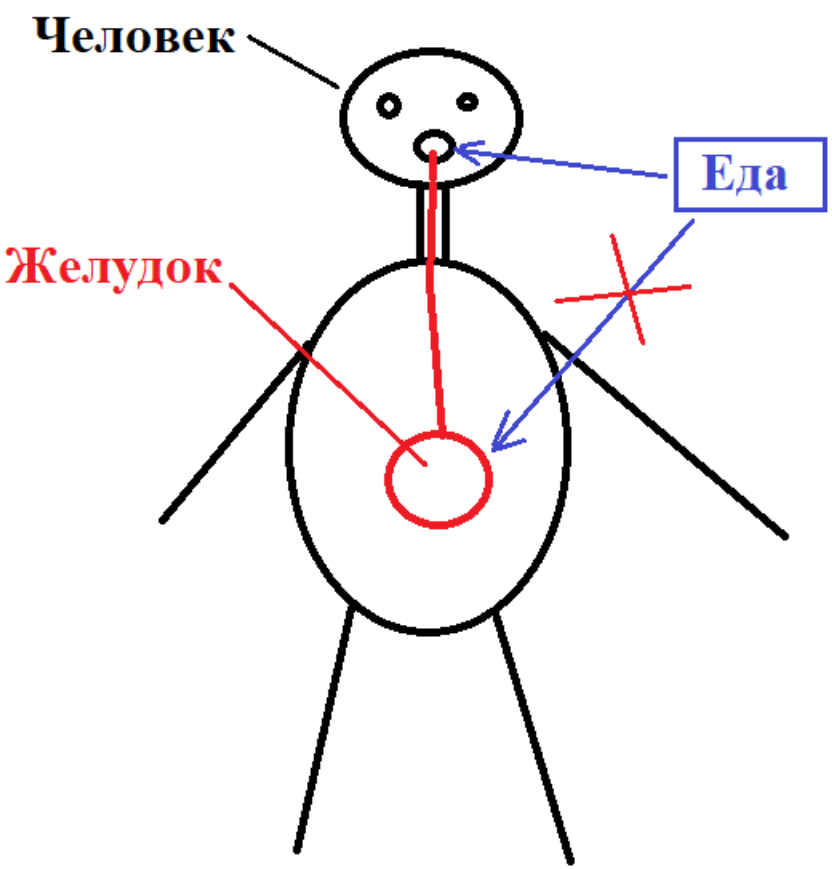
Тип int – это класс целых чисел. Числа 5, 100, -10 и т. д. – это конкретные объекты этого класса. В языке программирования Python объекты принято называть также экземплярами.

```
>>> type(list), type(int)
(<class 'type'>, <class 'type'>)
>>> import math
 >>> type(math)
<class 'module'>
```

Наследование



Инкапсуляция



Инкапсуляция в Python

В Python инкапсуляция реализуется только на уровне соглашения, которое определяет, какие характеристики являются общедоступными, а какие — внутренними.

В Python можно получить доступ к любому атрибуту объекта и изменить его.

Однако в Python есть механизм, позволяющий имитировать сокрытие данных, если это так уж необходимо.

Полиморфизм

Полиморфизм – это множество форм.

Однако в понятиях ООП имеется в виду скорее обратное. Объекты разных классов, с разной внутренней реализацией, то есть программным кодом, могут иметь одинаковые интерфейсы.

```
>>> 1 + 1
2
>>> ‘1’ + ‘1’
‘11’
```

Создание классов и объектов

```
class ИмяКласса:
    код_тела_класса

ИмяКласса()

имя_переменной = ИмяКласса()

>>> class A:
        pass
>>> a = A()
>>> b = A()
```

Класс как модуль

```
>>> class B:
        n = 5
        def adder(v):
            return v + B.n
 >>> B.n
5
>>> B.adder(4)
9
```

Класс как создатель объектов

adder()

```
>>> l = B()
>>> l.n
5
>>> l.adder(100)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: adder() takes 1 positional argument but 2 were given
```

Класс как создатель объектов

```
>>> l.n = 10
>>> l.n
10
>>> B.n
5
```

Класс как создатель объектов

```
>>> l = B()
>>> l.n
5
>>> l.adder(100)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: adder() takes 1 positional argument but 2 were given
```

Класс как создатель объектов

Выражение l.adder(100) выполняется следующим образом:

1. Ищу атрибут adder() у объекта l
2. Иду искать в класс B
3. Здесь нахожу искомый метод

Другими словами, выражение **l.adder(100)** преобразуется в выражение **B.adder(l, 100)**

В Python для ссылки на объект используется имя **self**

Обновим метод adder()

```
>>> class B:
        n = 5
        def adder(self, v):
            return v + self.n
```

Как будет работать обновленный метод

```
>>> l = B()
>>> m = B()
>>> l.n = 10
>>> l.adder(3)
13
>>> m.adder(4)
9
```

```
>>> m.n is B.n
True
>>> l.n is B.n
False
```

!!

В методе **adder()** выражение **self.n** – это обращение к свойству n, переданного объекта, и не важно, на каком уровне наследования оно будет найдено.

В Python для имитации статических методов используется специальный декоратор, после чего метод можно вызывать не только через класс, но и через объект, не передавая сам объект.

Их можно воспринимать как методы, которые “не знают, к какому классу относятся”.

@staticmethod

Изменение полей объекта

```
>>> l.test = "hi"
>>> B.test
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: type object 'B' has no attribute 'test'
>>> l.test
'hi'
```

Изменение полей объекта

```
class User:
    def setName(self, n):
            self.name = n
    def getName(self):
        try:
            return self.name
        except:
            print("No name")
```

```
>>> first = User()
>>> second = User()
>>> first.setName("Bob")
>>> first.getName()
'Bob'
>>> second.getName()
No name
```

Конструктор класса – метод __init__()

В объектно-ориентированном программировании конструктором класса называют метод, который автоматически вызывается при создании объектов. Его также можно назвать конструктором объектов класса.

В Python роль конструктора играет метод **__init__()**

Конструктор класса – метод __init__()

```
class Person:
   def setName(self, n, s):
       self.name = n
       self.surname = s

>>>from test import Person
>>>p1 = Person()
>>>p1.setName('Bill','Ross')
>>>p1.name, p1.surname
('Bill', 'Ross')
```

Конструктор класса – метод __init__()

```
class Person:
        def __init__(self, n, s):
            self.name = n
            self.surname = s

p1 = Person("Sam", "Baker")
print(p1.name, p1.surname)
```

Конструктор класса – метод __init__()

```
>>> p1 = Person()

Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'n' and 's'
```

Конструктор класса – метод __init__()

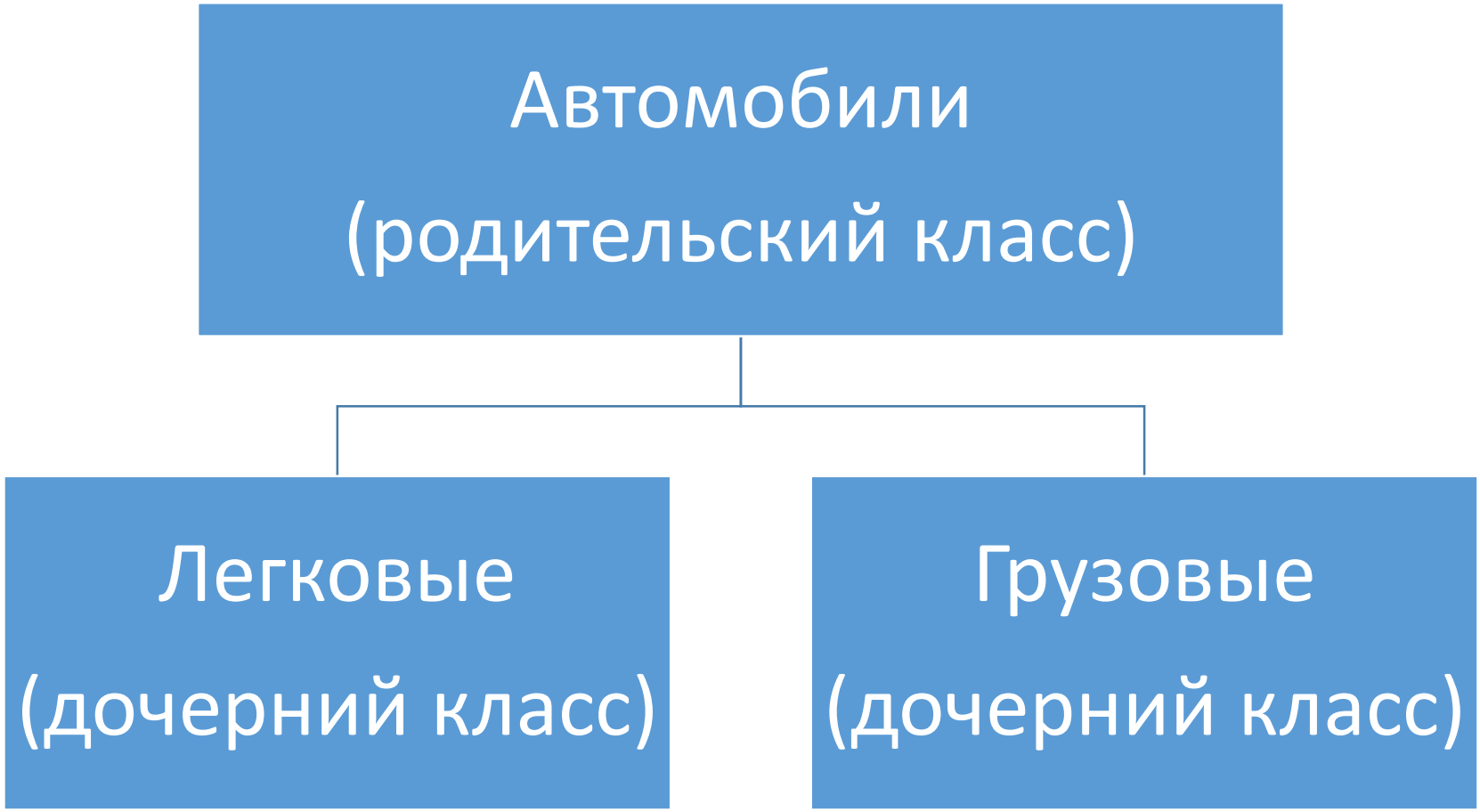
```
class Rectangle:
    def __init__(self, w = 0.5, h = 1):
        self.width = w
        self.height = h
    def square(self):
        return self.width * self.height
```

Конструктор класса – метод __init__()

```
rec1 = Rectangle(5, 2)
rec2 = Rectangle()
rec3 = Rectangle(3)
rec4 = Rectangle(h = 4)

>>>print(rec1.square())
10
>>> print(rec2.square())
0.5
>>> print(rec3.square())
3
>>> print(rec4.square())
2.0
```

Наследование



Наследование

```
class Table:
  def __init__(self, l, w, h):
      self.lenght = l
      self.width = w
      self.height = h
```

```
class KitchenTable(Table):
      def setPlaces(self, p):
          self.places = p

class DeskTable(Table):
          def square(self):
              return self.width * \
self.length
```

Наследование

```
>>> from test import *

>>> t1 = KitchenTable()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: __init__() missing 3 required positional arguments: 'l', 'w', and 'h'

 >>> t1 = KitchenTable(2, 2, 0.7)
>>> t2 = DeskTable(1.5, 0.8, 0.75)
>>> t3 = KitchenTable(1, 1.2, 0.8)
```

Наследование

```
>>> t4 = Table(1, 1, 0.5)
 >>> t2.square()
1.2000000000000002

>>> t4.square()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Table' object has no attribute 'square‘

 >>> t3.square()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'KitchenTable' object has no attribute 'square'
```

Полное переопределение метода надкласса

```
class ComputerTable(DeskTable):
    def square(self, e):
        return self.width * self.length – e

>>> from test import ComputerTable
>>> ct = ComputerTable(2, 1, 1)
>>> ct.square(0.3)
1.7
```

Дополнение, оно же расширение, метода

```
class ComputerTable(DeskTable):
    def square(self, e):
        return DeskTable.square(self) – e
```

```
class ComputerTable(DeskTable):
    def square(self, e):
        return self.width * self.length – e
```

Еще пример

```
class KitchenTable(Table):
        def __init__(self, l, w, h, p):
            self.length = l
            self.width = w
            self.height = h
            self.places = p

class KitchenTable(Table):
        def __init__(self, l, w, h, p):
            Table.__init__(self, l, w, h)
            self.places = p
```

Еще пример

```
>>> tk = KitchenTable(2, 1.5, 0.7, 10)
>>> tk.places 10
>>> tk.width 1.5
```

Полиморфизм

Полиморфизм в ООП – это возможность обработки разных типов данных, принадлежащих к разным классам, с помощью "одно и той же" функции, или метода.

Полиморфизм

```
class T1:
        n=10
         def total(self, N):
                self.total = int(self.n) + int(N)
class T2:
        def total(self,s):
                self.total = len(str(s))

>>>  t1 = T1()
>>>  t2 = T2()
>>>  t1.total(45)
>>>  t2.total(45)
>>>print(t1.total)
55
>>>print(t2.total)
2
```

Пример полиморфизма

```
>>> class A:
        def __init__(self, v1, v2):
            self.field1 = v1
            self.field2 = v2
>>> a = A(3, 4)
>>> print(a)
<__main__.A object at 0x7f840c8acfd0>
```

Пример полиморфизма

```
class A:
        def __init__(self, v1, v2):
            self.field1 = v1
            self.field2 = v2
        def __str__(self):
            return str(self.field1) + " " + str(self.field2)

>>> a = A(3, 4)
>>> print(a)
3 4
```

Пример полиморфизма

```
class Rectangle:
    def __init__(self, width, height, sign):
        self.w = int(width)
        self.h = int(height)
        self.s = str(sign)
    def __str__(self):
        rect = []
        for i in range(self.h): # количество строк
            rect.append(self.s * self.w) # знак повторяется w раз
        rect = '\n'.join(rect) # превращаем список в строку
        return rect

>>> b = Rectangle(10, 3, '*')
>>> print(b)
**********
**********
**********
```

Инкапсуляция

Инкапсуляция - ограничение доступа к методам и переменным, в Python работает лишь на уровне соглашения о том, какие атрибуты являются общедоступными, а какие — внутренними.

Зачем вообще что-то скрывать

- Могут существовать поля и методы классов, которые не должны использоваться за его пределами.
- Проверять значение на корректность в основном коде программы неправильно.
- Проверочный код должен быть помещен в метод, который получает данные, для присвоения полю. А само поле должно быть закрыто для доступа из вне класса.

Инкапсуляция

```
class B:
        count = 0
        def __init__(self):
            B.count += 1
        def __del__(self):
            B.count -= 1

>>>a = B()
>>>b = B()
>>>print(B.count)
2

>>> del a
>>> print(B.count)
1

>>> B.count -= 1
>>> print(B.count) # будет выведен 0, хотя остался объект b
0
```

Инкапсуляция

```
class B:
        __count = 0 # приватный атрибут
        def __init__(self):
                B.__count += 1
        def __del__(self):
                B.__count -= 1

>>>a = B()
>>>print(B.__count)
File "test.py", line 9, in <module>
        print(B.__count)
AttributeError: type object 'B' has no attribute '__count'

>>>print(B._B__count) # _ИмяКласса__ИмяАтрибута
1
```

Инкапсуляция

```
class A:
    def _private(self):
        print("Это приватный метод!")

>>> a = A()
>>> a._private()
Это приватный метод!
```

Инкапсуляция

```
class B:
        __count = 0
        def __init__(self):
                B.__count += 1
        def __del__(self):
                B.__count -= 1
        def qtyObject():
                return B.__count

>>>a = B()
>>>b = B()

>>> print(B.qtyObject())
2
```

Инкапсуляция

```
class DoubleList:
        def __init__(self, l):
            self.double = DoubleList.__makeDouble(l)
        def __makeDouble(old):
            new = []
            for i in old:
                new.append(i)
                new.append(i)
            return new
```

```
>>>nums = DoubleList([1, 3, 4, 6, 12])
>>>print(nums.double)
[1, 1, 3, 3, 4, 4, 6, 6, 12, 12]

>>>print(DoubleList.__makeDouble([1,2]))
 Traceback (most recent call last):
    File "test.py", line 13, in <module>
        print(DoubleList.__makeDouble([1,2]))
AttributeError: type object 'DoubleList' has no attribute '__makeDouble'
```

Метод __setattr__()

```
class A:
        def __init__(self, v):
                self.field1 = v


>>> a = A(10)
>>> a.field2 = 20
>>> a.field1, a.field2
(10, 20)
```

Метод __setattr__()

```
class A:
        def __init__(self, v):
                self.field1 = v
        def __setattr__(self, attr, value):
                if attr == 'field1':
                        self.__dict__[attr] = value
                else:
                         raise AttributeError
```

Метод __setattr__()

```
>>> a = A(15)
>>> a.field1
15


>>> a.field2 = 30
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 8, in __setattr__
AttributeError


>>> a.field2
Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'field2'


>>> a.__dict__
{'field1': 15}
```

Композиция

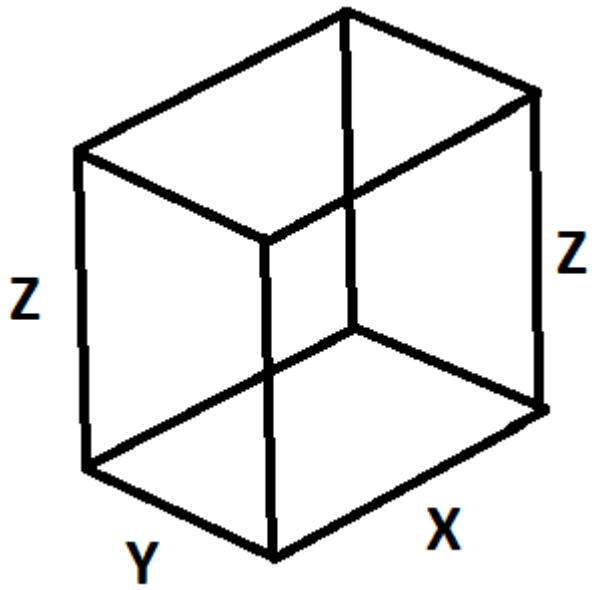
Композиционный подход заключается в том, что есть класс-контейнер, он же агрегатор, который включает в себя вызовы других классов.

В результате получается, что при создании объекта класса-контейнера, также создаются объекты включенных в него классов.

Композиция

Можно выделить три типа объектов:

- **окна**
- **двери**
- **комнаты**



$$S = 2xz + 2yz = 2z(x+y)$$

Композиция

```
class Win_Door:
        def __init__(self, x, y):
            self.square = x * y

class Room:
    def __init__(self, x, y, z):
        self.square = 2 * z * (x + y)
        self.wd = []
    def addWD(self, w, h):
        self.wd.append(Win_Door(w, h))
    def workSurface(self):
        new_square = self.square
        for i in self.wd:
            new_square -= i.square
        return new_square
```

```
>>>r1 = Room(6, 3, 2.7)
>>>print(r1.square)
48.6
>>>r1.addWD(1, 1)
>>>r1.addWD(1, 1)
>>>r1.addWD(1, 2)
>>>print(r1.workSurface())
44.6
```

Перегрузка операторов

Перегрузка операторов в Python – это возможность с помощью специальных методов в классах **переопределять** различные операторы языка.

Имена таких методов включают **двойное подчеркивание спереди и сзади**.

Перегрузка операторов

- **__init__()** – конструктор объектов класса, вызывается при создании объектов
- **__del__()** – деструктор объектов класса, вызывается при удалении объектов
- **__str__()** – преобразование объекта к строковому представлению, вызывается, когда объект передается функциям print() и str()
- **__add__()** – метод перегрузки оператора сложения, вызывается, когда объект участвует в операции сложения будучи операндом с левой стороны
- **__setattr__()** – вызывается, когда атрибуту объекта выполняется присваивание

Перегрузка операторов

```
class A:
        def __init__(self, arg):
            self.arg = arg
        def __str__(self):
            return str(self.arg)

class B:
        def __init__(self, *args):
            self.aList = []
            for i in args:
                self.aList.append(A(i))

>>>group = B(5, 10, 'abc')
>>>print(group.aList[1])
```

Перегрузка операторов

```
class B:
        def __init__(self, *args):
            self.aList = []
            for i in args:
                self.aList.append(A(i))
        def __getitem__(self, i):
            return self.aList[i]

>>>group = B(5, 10, 'abc')
>>>print(group.aList[1])
10
>>>print(group[0])
5
>>>print(group[2])
abc
```

Перегрузка операторов

a = A()

a()

a(3, 4)

Метод **__call__()**

объект = некийКласс()

объект([возможные аргументы])

Перегрузка операторов

```
class Changeable:
        def __init__(self, color):
            self.color = color
        def __call__(self, newcolor):
            self.color = newcolor
        def __str__(self):
            return "%s" % self.color

>>>canvas = Changeable("green")
>>>frame = Changeable("blue")

>>>canvas("red")
>>>frame("yellow")

>>>print (canvas, frame)
```

Перегрузка операторов

```
class A:
    def __str__(self):
        return "This is object of A"
```

```
>>> a = A()
>>> print(a)
This is object of A

>>> a
<__main__.A instance at 0x7fe964a4cdd0>

>>> str(a)
'This is object of A'

>>> repr(a)
'<__main__.A instance at 0x7fe964a4cdd0>'
```

Перегрузка операторов

```
>>> a = '3 + 2'
>>> b = repr(a)
>>> a
 '3 + 2'
>>> b
"'3 + 2'"
>>> eval(a)
5
>>> eval(b)
'3 + 2‘
```

```
>>> c = "Hello\nWorld"
>>> d = repr(c)
>>> c
'Hello\nWorld'
>>> d
"'Hello\\nWorld'"
>>> print(c)
Hello
World
>>> print(d)
'Hello\nWorld'
```

```
>>> c = "Hello\nWorld"
>>> c # аналог print(repr(c))
'Hello\nWorld'
>>> print(c) # аналог print(str(c))
Hello
World
```

Перегрузка операторов

```
class A:
    def __repr__(self):
        return "It's obj of A"
```

```
>>> a = A()
>>> a
It's obj of A
>>> repr(a)
"It's obj of A"
>>> str(a)
"It's obj of A"
>>> print(a)
It's obj of A
```