Лекция Объектно-ориентированное программирование (ООП)

Павлова А.И. Дисциплина

16 марта 2023 г.

ВОПРОСЫ:

- основные понятия объектно-ориентированного программирования
- Конструктор класса
- Статические и классовые методы
- разработка базы данных с помощью SQLite

1 Введение в объектно-ориентированное программирование на Python

Циклы, ветвления, функции — все это элементы так называемого структурного программирования (директивная парадигма программирования). Для написания небольших программ возможностей структурного программирования обычно достаточно.

Дла разработки сложных программ необходимо использовать парадигму объектно-ориентированного программирования (ООП).

ООП -объектно-ориентированного программирования. Истоки ООП начинаются с 60-х годов XX века. Окончательное оформление и популяризацию можно отнести к 80-м годам XX века.

Алан Кей сформулировал для разработанного им языка программирования Smalltalk несколько принципов.

Объектно-ориентированная программа состоит из объектов.

Переменные называют атрибутами или свойствами, а функции — методами. Класс объединяет объекты, т. е. позволяет создать неограниченное количество экземпляров, основанных на этом классе.

Основными понятиями, используемыми в ООП, являются класс, объект, наследование, инкапсуляция и полиморфизм. В языке Python класс равносилен понятию тип данных.

Каждый объект может состоять из других объектов (а может и не состоять). Каждый объект принадлежит определенному классу (типу), который задает поведение объектов, созданных на его основе.

В языке программирования Python объекты класса принято называть также экземплярами. Это связано с тем, что в нем все классы сами являются объектами класса **type**. Точно также как все модули являются объектами класса module

Класс — это описание объектов определенного типа, абстракция без материального воплощения, которая позволяет систематизировать объекты системы.

На основе классов создаются объекты.

Может быть множество объектов, принадлежащих одному классу.

С другой стороны, может быть класс без объектов, реализованных на его основе.

Современные языки поддерживают несколько парадигм программирования (например, директивное, функциональное, объектноориентированное).

Такие языки являются смешанными. К ним относится и Python

Программа, написанная с использованием парадигмы объектно-ориентированного программирования, должна состоять из объектов классов (описания объектов)

взаимодействий объектов между собой, в результате которых меняются их свойства.

Существует разница между программой, разработанной на структурном «стиле» и программой в «стиле» ООП.

В первом случае, на первый план выходит логика, понимание последовательности выполнения выражений (действий) для достижения целей.

Во втором — важно системное мышление, умение видеть систему в целом, с одной стороны, и понимание роли ее частей (объектов), с другой.

Создание классов

Для создания классов предусмотрена инструкция class. Это составная инструкция, которая состоит из строки заголовка и тела. Заголовок состоит из ключевого слова class, имени класса и, возможно, названий суперклассов в скобках. Суперклассов может и не быть, в таком случае скобки не требуются. Тело класса состоит из блока различных инструкций. Тело должно иметь отступ (как и любые вложенные конструкции в языке Python).

Схематично класс можно представить следующим образом:

```
1. class ИМЯКЛАССА:
2.         ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
3.         ....
4.         def ИМЯМЕТОДА(self, ...):
5.         self.ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
```

Данная схема не является полной.

Например, в заголовке после имени класса могут быть указаны суперклассы (вскобках), а методы могут быть более сложными. Следует помнить, что методы в классах — это те же функции, за одним небольшим исключением. Они принимают один обязательный параметр — **self (с англ. "собственная личность")** для связи с объектом.

Атрибуты класса — это имена переменных вне функций и имена функций. Эти атрибуты наследуются всеми объектами, созданными на основе данного класса. Атрибуты обеспечивают свойства и поведение объекта. Объекты могут иметь атрибуты, которые создаются в теле метода, если данный метод будет вызван для конкретного объекта.

Концепция ООП заключается в создании многократно используемого кода. Существует три ключевых принципа использования ООП:

Данные структурируются в виде объектов, каждый из которых имеет определенный тип, то есть принадлежит к какому-либо классу.

Классы – результат формализации решаемой задачи, выделения главных ее аспектов.

Внутри объекта инкапсулируется логика работы с относящейся к нему информацией.

Объекты в программе взаимодействуют друг с другом, обмениваются запросами и ответами. При этом объекты одного типа сходным образом отвечают на одни и те же запросы. Объекты могут организовываться в более сложные структуры, например, включать другие объекты или наследовать от одного или нескольких объектов.

Наследование (Inheritance) - способ создания новых классов на основе существующего класса без его модификации.

Инкапсуляция (Encapsulation) - способ скрыть некоторые частные детали класса от других объектов.

Полиморфизм (Polymorphism) - способ использования общей операции разными способами для разных вводимых данных.

Благодаря вышеперечисленным принципам, существует множество преимуществ использования ООП: Оно обеспечивает четкую модульную структуру программ, что повышает удобство повторного использования кода. Это обеспечивает простой способ решения сложных проблем. Это помогает определить более абстрактные типы данных для моделирования реальных сценариев. Он скрывает детали реализации, оставляя четко определенный интерфейс. Он объединяет данные и операции.

Создание классов и объектов:

Объекты создаются так:

```
class <Название класса>[(<Класс1>, ..., <КлассN>)]:
""" Строка документирования """
<Описание атрибутов и методов>
```

Инструкция создает новый объект и присваивает ссылку на него идентификатору, указанному после ключевого слова class. Это означает, что название класса должно полностью соответствовать правилам именования

переменных. После названия класса в круглых скобках можно указать один или несколько базовых классов через запятую. Классы-родители перечисляются в скобках через запятую

```
class SomeClass(ParentClass1, ParentClass2, ...):
"""  поля и методы класса SomeClass """
```

После такой инструкции в программе появляется объект, доступ к которому можно получить по имени переменной, связанной с ним.

При создании объект получает атрибуты его класса, т. е. объекты обладают характеристиками, определенными в их классах.

Количество объектов, которые можно создать на основе того или иного класса, не ограничено.

Объекты одного класса имеют схожий набор атрибутов, а вот значения атрибутов могут быть разными, объекты одного класса похожи, но индивидуально различимы.

Например. "Все млекопитающие принадлежат одному классу, но кажде животное отличается цветом глаз".

Свойства классов устанавливаются с помощью простого присваивания:

```
class SomeClass(object):
    attr1 = 42
    attr2 = "Hello, World"
```

Методы объявляются в виде функций:

```
class SomeClass(object):
    def method1(self, x):
         код метода
```

Self:

Методы класса — это небольшие программы, предназначенные для работы с объектами. Методы могут создавать новые свойства (данные) объекта, изменять существующие, выполнять другие действия над объектами.

Методу необходимо "знать данные какого объекта ему предстоит обрабатывать. Для этого ему в качестве первого (а иногда и единственного) аргумента передается имя переменной, связанной с объектом.

Для того чтобы в описании класса указать передаваемый в дальнейшем объект, используется параметр **self**.

self – общепринятое имя для ссылки на объект, в контексте которого вызывается метод. Этот параметр обязателен и отличает метод класса от обычной функции.

Все пользовательские атрибуты сохраняются в атрибуте **dict**, который является словарем.

Экземпляры классов

Инстанцировать класс в Python

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

    def method1(self, x):
        return 2*x

obj = SomeClass()
obj.method1(6) # 12
obj.attr1 # 42
```

Вызов метода для конкретного объекта в основном блоке программы производится следующим образом:

```
ОБЪЕКТ.ИМЯМЕТОДА(...)
```

Здесь под словом ОБЪЕКТ имеется в виду переменная, связанная с ним.

Это выражение преобразуется в классе, к которому относится объект.

```
ИМЯМЕТОДА(ОБЪЕКТ, ...)
```

Пример. В приведенном примере кода мы сначала определили класс - People, с именем и возрастом в качестве данных, а также метод greet.

Затем мы инициализировали объект - person1 с определенными именем и возрастом. класс определяет всю структуру, в то время как объект является просто экземпляром класса.

Позже мы инициализировали еще один объект - person2.

person1 и person2 независимы друг от друга, хотя все они инициализирвоаны из одного класса.

2 Разработка программ с классами

ПРИМЕР 1.

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

    def greet(self):
        print("Сотрудник, " + self.name)

person1 = People(name='Иван', age=54)
person1.greet()
print(person1.name)
print(person1.age)
```

```
person2 = People(name = 'Сидоров', age = 43)
person2.greet()
print(person2.name)
print(person2.age)

Результата вывода на экран:

Сотрудник, Иван
Иван
54
Сотрудник, Сидоров
Сидоров
43
```

С точки зрения пространства имен класс можно представить подобным модулю. Также как в модуле в классе могут быть свои переменные со значениями и функции. Также как в модуле у класса есть собственное пространство имен, доступ к которому возможен через имя класса:

ПРИМЕР . Пример использования метода:

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

В случае классов используется особая терминология. Имена, определенные в классе, называются атрибутами этого класса. В примере имена n и adder – это атрибуты класса B. Атрибуты-переменные называют полями или свойствами (в других языках понятия "поле"и "свойство"не совсем одно и то же). Полем является n. Атрибуты-функции называются методами. Методом в классе B является adder. Количество свойств и методов в классе может быть любым.

Создание атрибута класса аналогично созданию обычной переменной. Метод внутри класса создается так же, как и обычная функция, – с помощью инструкции def. Среди атрибутов можно выделить две группы.

Атрибуты первой группы принадлежат самому классу, а не конкретному экземпляру.

Доступ к таким атрибутам вне тела класса осуществляется через точечную нотацию через объект объявления класса.

Создание атрибута класса

Создание атрибута класса аналогично созданию обычной переменной. Метод внутри класса создается так же, как и обычная функция, – с помощью инструкции def. Среди атрибутов можно выделить две группы. Атрибуты первой группы принадлежат самому классу, а не конкретному экземпляру. Доступ к таким атрибутам вне тела класса осуществляется через точечную нотацию через объект объявления класса.

```
<Имя класса>.<Имя атрибута>
        Если этот атрибут - метод, то по аналогии можно его вызвать.
<Имя класса>.<Имя метода>([<Параметры>])
```

Во второй группе находятся атрибуты, принадлежащие конкретному экземпляру. В отличии от многих других языков программирования, эти атрибуты создаются уже во время существования объекта. Чтобы создать экземпляр класса используется синтаксис:

```
<экземпляр класса> = <Название класса>([<Параметры>])
```

Когда экземпляр класса существует, можно задавать атрибуты уникальные для этого самого экземпляра:

```
<экземпляр класса>.<Имя атрибута> = <Значение>
```

Если не удается найти атрибут с таким именем у экземпляра, то поиск продолжается в атрибутах, принадлежащих классу. В связи с этим можно считать, что атрибуты самого класса разделяются между всеми его экземплярами

```
class MyClass:
     x = 50 # Создаем атрибут объекта класса MyClass

c1, c2 = MyClass(), MyClass() # Создаем два экземпляра класса

c1.y = 10 # Создаем атрибут экземпляра класса c1

c2.y = 20 # Создаем атрибут экземпляра класса c2

print(c1.x, ' ', c1.y) # Вывод: 50 10

print(c2.x, ' ', c2.y) # Вывод: 50 20
```

В этом примере мы определяем класс MyClass и создаем атрибут объекта класса: x. Этот атрибут будет доступен всем создаваемым экземплярам класса. Затем создаем два экземпляра класса и добавляем одноименные атрибуты: y. Значения этих атрибутов будут разными в каждом экземпляре класса. Но если создать новый экземпляр (например, c3), то атрибут y в

нем определен не будет. Таким образом, с помощью классов можно имитировать типы данных, поддерживаемые другими языками программирования (например, тип struct, доступный в языке C). Обычно все методы класса объявляются на уровне класса, а не экземпляра. Вызов такого метода через экземпляр класса с помощью синтаксиса

```
<экземпляр класса>.<Имя метода>([<Параметры>])
```

Неявно преобразуется к

```
<Имя класса>.<Имя метода>(<экземпляр класса>, [<параметры>])
```

Методам класса в первом параметре, который необходимо указывать явно, автоматически передается ссылка на экземпляр класса. Общепринято этот параметр называть именем self, хотя это и не обязательно. Доступ к атрибутам и методам класса внутри определяемого метода производится через переменную self с помощью точечной нотации. Обратите внимание на то, что при вызове метода не нужно передавать ссылку на экземпляр класса в качестве параметра, как это делается в определении метода внутри класса. Ссылку на экземпляр класса интерпретатор передает автоматически. Определим класс MyClass с атрибутом x и методом

```
print_x()
```

, выводящим значение этого атрибута, а затем создадим экземпляр класса и вызовем метод:

ПРИМЕР 2. Пример использования метода:

```
  class MyClass:

def __init__(self): # Конструктор

self.x = 1 # Атрибут экземпляра класса

def print_x(self): # self - это ссылка на экземпляр класса

print(self.x) # Выводим значение атрибута

c = MyClass() # Создание экземпляра класса

# Вызываем метод print_x()

c.print_x()# self не указывается при вызове метода

print(c.x) # К атрибуту можно обратиться непосредственно
```

Все атрибуты класса в языке Python являются открытыми (public), т. е. доступными для непосредственного изменения как из самого класса, так и из других классов и из основного кода программы.

3 Констурктор класса

Метод

`__init__`

При создании экземпляра класса интерпретатор автоматически вызывает метод инициализации

`__init__`

. Такой метод принято называть конструктором класса.

Метод

`__init__`

предполагает передачу аргументов при создании объектов.

С помощью метода

`__init__`

атрибутам класса можно присвоить начальные значения. При создании экземпляра класса параметры этого метода указываются после имени класса в круглых скобках

Например, если в примере выше создать объект так:

```
obj1 = YesInit(),
```

т.е. не передать классу аргументы, то произойдет ошибка. Чтобы избежать подобных ситуаций, можно в методе

```
__init__
```

присваивать параметрам значения по умолчанию. Если при вызове класса были заданы аргументы для данных параметров, то они будут использованы

ПРИМЕР 3. Пример использования метода:

```
<Экземпляр класса> = <Имя класса> ([<Значение1> [, ...,<ЗначениеN>]])
```

```
class MyClass:
def __init__(self, valuel, value2): # Конструктор
self.x = valuel
```

```
self.y = value2
c = MyClass(100, 300) # Создаем экземпляр класса
print(c.x, c.y) # Вывод: 100 300
```

Большинство классов имеют специальный метод, который автоматически при создании объекта создает ему атрибуты. Т.е. вызывать данный метод не нужно, т.к. он сам запускается при вызове класса. (Вызов класса происходит, когда создается объект.) Такой метод называется конструктором класса и в языке программирования Python носит имя

`__init__.` `(В начале и конце по два знака подчеркивания.)}`

Первым параметром, как и у любого другого метода,

`у __init__`

является self, на место которого подставляется объект в момент его создания. Второй и последующие (если есть) параметры заменяются аргументами, переданными в конструктор при вызове класса. Рассмотрим два класса: в одном будет использоваться конструктор, а в другом нет. Требуется создать два атрибута объекта.

ПРИМЕР 4.

```
class YesInit:
    def __init__(self,one="студент",two="не студент"):
        self.fname = one
        self.sname = two

obj1 = YesInit("Сережа","Петров")
obj2 = YesInit()
obj3 = YesInit("Петя")
obj4 = YesInit(two="Сидоров")

print (obj1.fname, obj1.sname)
print (obj2.fname, obj2.sname)
print (obj3.fname, obj3.sname)
print (obj4.fname, obj4.sname)
```

```
Результат вывода:
Сережа Петров
студент не студент
Петя не студент
студент Сидоров
```

В данном случае, второй объект создается без передачи аргументов, поэтому в методе

```
__init__
```

используются значения по умолчанию ("студент"и "не студент").

Программа с использованием конструктора.

Зададим класс, значение начальных атрибутов

```
(из метода __init__)
```

которого зависит от переданных аргументов при создании объектов. Свойства объектов, созданных на основе данного класса, можно менять с помощью методов

ПРИМЕР 5.

```
class Student():

    def __init__(self, sid, name, gender):
        self.sid = sid
        self.name = name
        self.gender = gender
        self.type = 'learning'

    def say_name(self):
        print("My name is " + self.name)

    def report(self, score):
        self.say_name()
        print("My id is: " + self.sid)
        print("My score is: " + str(score))

student1 = Student("001", "Susan", "F")
student2 = Student("002", "Mike", "M")

student1.say_name()
student2.say_name()
print(student1.type)
print(student1.gender)

Вывод результата:

My name is Susan
My name is Mike
learning
F
**************************
My name is Susan
```

```
My id is: 001
My score is: 95
My name is Mike
My id is: 002
My score is: 90
```

Атрибуты класса и экземпляра Атрибуты называют атрибутами экземпляра

Атрибуты принадлежат только определенному экземпляру, необходимо использовать атрибут self.attribute внутри класса.

Существует другой тип атрибутов, называемых атрибутами класса - общие для всех экземпляров, созданных на основе этого класса.

ПРИМЕР 6.

Измените класс Student, добавив атрибут класса n, который будет записывать, сколько объектов мы создаем. Также добавьте метод

```
num_instances
```

, чтобы вывести это число.

```
class Student():
    n_instances = 0

    def __init__(self, sid, name, gender):
        self.sid = sid
        self.name = name
        self.gender = gender
        self.type = 'learning'
        Student.n_instances += 1

    def say_name(self):
        print("My name is " + self.name)

    def report(self, score):
        self.say_name()
        print("My id is: " + self.sid)
        print("My score is: " + str(score))

    def num_instances(self):
        print(f'We have {Student.n_instances}-instance in total')

student1 = Student("001", "Susan", "F")
student1.num_instances()
student2 = Student("002", "Mike", "M")
student1.num_instances()
student2.num_instances()
```

4 Статические и классовые методы

Для создания статических методов в Python предназначен декоратор @staticmethod

У статических методов нет обязательных параметров-ссылок self. Доступ к методам можно получить от экземпляра класса и от класса:

ПРИМЕР 7.

```
class SomeClass(object):

    @staticmethod

    def hello():

        print("Hello, world")


SomeClass.hello() # Hello, world

obj = SomeClass()

obj.hello() # Hello, world
```

методы классов аналогичны методам экземпляров, но выполняются не в контексте объекта, а в контексте самого класса (классы – это тоже объекты). Методы создаются с помощью декоратора

```
@classmethod
```

и требуют обязательную ссылку на класс **(cls)**.

ПРИМЕР 8.

```
lass SomeClass(object):

    @classmethod

    def hello(cls):

        print('Hello, класс {}'.format(cls.__name__))


SomeClass.hello() # Hello, класс SomeClass



\textbf{ПРИМЕР 9.}  Класс и значения атрибутов

\begin{verbatim}
```

```
class Cat():
  def __init__(self, breed, color, age):
     self._breed = breed
     self._color = color
     self._age = age

  @property
  def breed(self):
    return self._breed

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

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

  @age.setter
  def age(self, new_age):
    if new_age > self._age:
      self._age = new_age
    return self._age
```

Создание экземпляра класса

```
cat = Cat('Абиссинская', 'Рыжая', 4)
```

Выводим значения атрибутов:

```
print(cat.breed) # Абиссинская
print(cat.color) # Рыжая
print(cat.age) # 4
```