Классы и объекты

Python имеет множество встроенных типов, например, int, str и так далее, которые мы можем использовать в программе. Но также Python позволяет определять собственные типы с помощью **классов**. Класс представляет некоторую сущность. Конкретным воплощением класса является объект.

Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики Человек может выполнять некоторые действия - ходить, бегать, думать и т.д. То есть это представление, которое включает набор характеристик и действий, можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек будет представлять объект этого класса.

Класс определяется с помощью ключевого слова **class**:

```
class название_класса:
    атрибуты_класса
    методы_класса
```

Внутри класса определяются его атрибуты, которые хранят различные характеристики класса, и методы - функции класса.

Создадим простейший класс:

```
class Person:
    pass
```

В данном случае определен класс Person, который условно представляет человека. В данном случае в классе не определяется никаких методов или атрибутов. Однако поскольку в нем должно быть что-то определено, то в качестве заменителя функционала класса применяется оператор **pass**. Этот оператор применяется, когда синтаксически необходимо определить

некоторый код, однако мы не хотим его, и вместо конкретного кода вставляем оператор pass.

После создания класса можно определить объекты этого класса. Например:

```
class Person:
    pass

tom = Person()      # определение объекта tom
bob = Person()      # определение объекта bob
```

После определения класса Person создаются два объекта класса Person - tom и bob. Для создания объекта применяется специальная функция - конструктор, которая называется по имени класса и которая возвращает объект класса. То есть в данном случае вызов Person() представляет вызов конструктора. Каждый класс по умолчанию имеет конструктор без параметров:

```
tom = Person()      # Person() - вызов конструктора, который возвращает объект класса Person
```

Методы классов

Методы класса фактически представляют функции, которые определенны внутри класса и которые определяют его поведение. Например, определим класс Person с одним методом:

```
class Person:       # определение класса Person
    def say_hello(self):
        print("Hello")

tom = Person()
tom.say_hello()    # Hello
```

Здесь определен метод say_hello(), который условно выполняет приветствие - выводит строку на консоль. При определении методов любого класса следует учитывать, что все они должны принимать в качестве первого параметра ссылку на текущий объект, который согласно условностям называется **self**. Через эту ссылку внутри класса мы можем

обратиться к функциональности текущего объекта. Но при самом вызове метода этот параметр не учитывается.

Используя имя объекта, мы можем обратиться к его методам. Для обращения к методам применяется нотация точки - после имени объекта ставится точка и после нее идет вызов метода:

```
объект.метод([параметры метода])
```

Например, обращение к методу say_hello() для вывода приветствия на консоль:

```
tom.say_hello()    # Hello
```

В итоге данная программа выведет на консоль строку "Hello".
Если метод должен принимать другие параметры, то они определяются после параметра self, и при вызове подобного метода для них необходимо передать значения:

```
class Person:       # определение класса Person
    def say(self, message):     # метод
        print(message)


tom = Person()
tom.say("Hello SYNERGY")    # Hello SYNERGY
```

Здесь определен метод say(). Он принимает два параметра: self и message. И для второго параметра - message при вызове метода необходимо передать значение.

self

Через ключевое слово self можно обращаться внутри класса к функциональности текущего объекта:

```
self.атрибут    # обращение к атрибуту
self.метод      # обращение к методу
```

Например, определим два метода в классе Person:

```
class Person:

    def say(self, message):
        print(message)

    def say_hello(self):
        self.say("Hello study")  # обращаемся к выше определенному методу say


tom = Person()
tom.say_hello()     # Hello study
```

Здесь в одном методе - say_hello() вызывается другой метод - say():

```
self.say("Hello study")
```

Поскольку метод say() принимает кроме self еще параметры (параметр message), то при вызове метода для этого параметра передается значение.

Причем при вызове метода объекта нам обязательно необходимо использовать слово **self**, если мы его не используем:

```
def say_hello(self):
    say("Hello study")  # ! Ошибка
```

То мы столкнемся с ошибкой

Конструкторы

Для создания объекта класса используется конструктор. Так, выше когда мы создавали объекты класса Person, мы использовали конструктор по умолчанию, который не принимает параметров и который неявно имеют все классы:

```
tom = Person()
```

Однако мы можем явным образом определить в классах конструктор с помощью специального метода, который называется __init__() (по два прочерка с каждой стороны). К примеру, изменим класс Person, добавив в него конструктор:

```
class Person:
    # конструктор
    def __init__(self):
        print("Создание объекта Person")

    def say_hello(self):
        print("Hello")


tom = Person()      # Создание объекта Person
tom.say_hello()     # Hello
```

Итак, здесь в коде класса Person определен конструктор и метод say_hello(). В качестве первого параметра конструктор, как и методы, также принимает ссылку на текущий объект - self. Обычно конструкторы применяются для определения действий, которые должны производиться при создании объекта.

Теперь при создании объекта:

```
tom = Person()
```

будет производится вызов конструктора __init__() из класса Person, который выведет на консоль строку "Создание объекта Person".

Атрибуты объекта

Атрибуты хранят состояние объекта. Для определения и установки атрибутов внутри класса можно применять слово **self**. Например, определим следующий класс Person:

```
class Person:

    def __init__(self, name):
        self.name = name    # имя человека
        self.age = 1        # возраст человека


tom = Person("Tom")

# обращение к атрибутам
```

```
# получение значений
print(tom.name)     # Tom
print(tom.age)      # 1
# изменение значения
tom.age = 37
print(tom.age)      # 37
```

Теперь конструктор класса Person принимает еще один параметр - name. Через этот параметр в конструктор будет передаваться имя создаваемого человека.

Внутри конструктора устанавливаются два атрибута - name и age (условно имя и возраст человека):

```
def __init__(self, name):
    self.name = name
    self.age = 1
```

Атрибуту self.name присваивается значение переменной name. Атрибут age получает значение 1.

Если мы определили в классе конструктор __init__, мы уже не сможем вызвать конструктор по умолчанию. Теперь нам надо вызывать наш явным образом определенный конструктор __init__, в который необходимо передать значение для параметра name:

```
tom = Person("Tom")
```

Далее по имени объекта мы можем обращаться к атрибутам объекта - получать и изменять их значения:

```
print(tom.name)     # получение значения атрибута name
tom.age = 37        # изменение значения атрибута age
```

В принципе нам не обязательно определять атрибуты внутри класса - Python позволяет сделать это динамически вне кода:

```
class Person:

    def __init__(self, name):
        self.name = name    # имя человека
```

```
        self.age = 1        # возраст человека


tom = Person("Tom")

tom.company = "Microsoft"
print(tom.company)  # Microsoft
```

Здесь динамически устанавливается атрибут company, который хранит место работы человека. И после установки мы также можем получить его значение. В то же время подобное определение чревато ошибками. Например, если мы попытаемся обратиться к атрибуту до его определения, то программа генерирует ошибку:

```
tom = Person("Tom")
print(tom.company)  # ! Ошибка - AttributeError: Person object has no attribute company
```

Для обращения к атрибутам объекта внутри класса в его методах также применяется слово self:

```
class Person:

    def __init__(self, name):
        self.name = name    # имя человека
        self.age = 1        # возраст человека

    def display_info(self):
        print(f"Name: {self.name}  Age: {self.age}")


tom = Person("Tom")
tom.display_info()      # Name: Tom  Age: 1
```

Здесь определяется метод display_info(), который выводит информацию на консоль. И для обращения в методе к атрибутам объекта применяется слово self: self.name и self.age

Создание объектов

Выше создавался один объект. Но подобным образом можно создавать и другие объекты класса:

```
class Person:

    def __init__(self, name):
        self.name = name    # имя человека
        self.age = 1        # возраст человека

    def display_info(self):
        print(f"Name: {self.name}  Age: {self.age}")


tom = Person("Tom")
tom.age = 37
tom.display_info()      # Name: Tom  Age: 37

bob = Person("Bob")
bob.age = 41
bob.display_info()      # Name: Bob  Age: 41
```

Здесь создаются два объекта класса Person: tom и bob. Они соответствуют определению класса Person, имеют одинаковый набор атрибутов и методов, однако их состояние будет отличаться.

При выполнении программы Python динамически будет определять **self** - он представляет объект, у которого вызывается метод. Например, в строке:

```
tom.display_info()      # Name: Tom  Age: 37
```

Это будет объект tom

А при вызове

```
bob.display_info()
```

Это будет объект bob

В итоге мы получим следующий консольный вывод:

```
Name: Tom  Age: 37
Name: Bob  Age: 41
```

Наследование позволяет создавать новый класс на основе уже существующего класса. Наряду с инкапсуляцией наследование является

одним из краеугольных камней объектно-ориентированного программирования.

Ключевыми понятиями наследования являются **подкласс** и **суперкласс**. **Подкласс** наследует от суперкласса все публичные атрибуты и методы. Суперкласс еще называется базовым (base class) или родительским (parent class), а подкласс - производным (derived class) или дочерним (child class).

Синтаксис для наследования классов выглядит следующим образом:

```
class подкласс (суперкласс):
    методы_подкласса
```

Например, у нас есть класс Person, который представляет человека:

```
class Person:

    def __init__(self, name):
        self.__name = name   # имя человека

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name} ")
```

Предположим, нам необходим класс работника, который работает на некотором предприятии. Мы могли бы создать с нуля новый класс, например, класс Employee:

```
class Employee:

    def __init__(self, name):
        self.__name = name  # имя работника

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name} ")
```

```
    def work(self):
        print(f"{self.name} works")
```

Однако класс Employee может иметь те же атрибуты и методы, что и класс Person, так как работник - это человек. Так, в выше в классе Employee только добавляется метод works, весь остальной код повторяет функционал класса Person. Но чтобы не дублировать функционал одного класса в другом, в данном случае лучше применить наследование.

Итак, наследуем класс Employee от класса Person:

```
class Person:

    def __init__(self, name):
        self.__name = name   # имя человека

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name} ")


class Employee(Person):

    def work(self):
        print(f"{self.name} works")


tom = Employee("Tom")
print(tom.name)     # Tom
tom.display_info()  # Name: Tom
tom.work()          # Tom works
```

Класс Employee полностью перенимает функционал класса Person, лишь добавляя метод work(). Соответственно при создании объекта Employee мы можем использовать унаследованный от Person конструктор:

```
tom = Employee("Tom")
```

И также можно обращаться к унаследованным атрибутам/свойствам и методам:

```
print(tom.name)     # Tom
tom.display_info()  # Name: Tom
```

Однако, стоит обратить внимание, что для Employee НЕ доступны закрытые атрибуты типа __name. Например, мы НЕ можем в методе work обратиться к приватному атрибуту self.__name:

```
def work(self):
    print(f"{self.__name} works")   # ! Ошибка
```

Множественное наследование

Одной из отличительных особенностей языка Python является поддержка множественного наследования, то есть один класс можно унаследовать от нескольких классов:

```
#  класс работника
class Employee:
    def work(self):
        print("Employee works")


#  класс студента
class Student:
    def study(self):
        print("Student studies")


class WorkingStudent(Employee, Student):        # Наследование от классов Employee и Student
    pass


# класс работающего студента
tom = WorkingStudent()
tom.work()      # Employee works
tom.study()     # Student studies
```

Здесь определен класс Employee, который представляет сотрудника фирмы, и класс Student, который представляет учащегося студента. Класс WorkingStudent, который представляет работающего студента, не определяет никакого функционала, поэтому в нем определен оператор **pass**. Класс WorkingStudent просто наследует функционал от двух классов

Employee и Student. Соответственно у объекта этого класса мы можем вызвать методы обоих классов.

При этом наследуемые классы могут более сложными по функциональности, например:

```
class Employee:

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

    @property
    def name(self):
        return self.__name

    def work(self):
        print(f"{self.name} works")


class Student:

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

    @property
    def name(self):
        return self.__name

    def study(self):
        print(f"{self.name} studies")


class WorkingStudent(Employee, Student):
    pass


tom = WorkingStudent("Tom")
tom.work()      # Tom works
tom.study()     # Tom studies
```