Программирование

Лекция № 4.

Объектно-ориентированное программирование

Нижний
Новгород
2019 г.

Основные понятия ООП

ОО программирование - это методология написания кода.

Класс

Представьте, что вы проектируете автомобиль. Вы знаете, что автомобиль должен содержать двигатель, подвеску, две передних фары, 4 колеса, и т.д. Ещё вы знаете, что ваш автомобиль должен иметь возможность набирать и сбавлять скорость, совершать поворот и двигаться задним ходом. И, что самое главное, вы точно знаете, как взаимодействует двигатель и колёса, согласно каким законам движется распредвал и коленвал, а также как устроены дифференциалы. Вы уверены в своих знаниях и начинаете проектирование.

Основные понятия ООП

Вы описываете все запчасти, из которых состоит ваш автомобиль, а также то, каким образом эти запчасти взаимодействуют между собой. Кроме того, вы описываете, что должен сделать пользователь, чтобы машина затормозила, или включился дальний свет фар. Результатом вашей работы будет некоторый **эскиз**.

Вы только что разработали то, что в ООП называется ***класс***.

Класс - способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью (контракт).

С точки зрения программирования класс можно рассматривать как **набор данных** (полей, атрибутов, членов класса) и **функций** для работы с ними (методов).

С точки зрения структуры программы, класс является **сложным типом данных**.

В нашем случае, класс будет отображать сущность – автомобиль. **Атрибутами** класса будут являться **двигатель**, **подвеска**, **кузов**, **четыре колеса** и т.д. **Методами** класса будет «**открыть дверь**», «**нажать на педаль газа**», а также «**закачать порцию бензина из бензобака в двигатель**». Первые два метода доступны для выполнения другим классам (в частности, классу «Водитель»). Последний описывает

Объект

Вы отлично потрудились и машины, разработанные по вашим чертежам, сходят с конвейера. Вот они, стоят ровными рядами на заводском дворе. Каждая из них точно повторяет ваши чертежи. Все системы взаимодействуют именно так, как вы спроектировали. Но каждая машина уникальна. Они все имеют номер кузова и двигателя, но все эти номера разные, автомобили различаются цветом, а некоторые даже имеют литьё вместо штампованных дисков. Эти автомобили, по сути, являются **объектами** вашего класса.

Объект (экземпляр) – это отдельный **представитель** класса, имеющий **конкретное** состояние и поведение, полностью определяемое классом.

Говоря простым языком, объект имеет **конкретные значения атрибутов** и **методы**, работающие с этими значениями на основе правил, заданных в классе. В данном примере, если **класс** – это **некоторый абстрактный автомобиль** из «мира идей», то **объект** – это **конкретный автомобиль**, стоящий у вас под окнами.

Интерфейс

Когда мы подходим к автомату с кофе или садимся за руль, мы начинаем взаимодействие с ними. Обычно, **взаимодействие** происходит с помощью некоторого набора элементов: щель для приёмки монеток, кнопка выбора напитка и отсек выдачи стакана в кофейном автомате; руль, педали, рычаг коробки переключения передач в автомобиле. Всегда существует некоторый ограниченный набор **элементов управления**, с которыми мы можем взаимодействовать.

Интерфейс

Интерфейс – это набор методов класса, доступных для использования другими классами.

Очевидно, что **интерфейсом** класса будет являться набор всех его **публичных методов (функций)** в совокупности с набором **публичных атрибутов (переменных)**. По сути, интерфейс специфицирует класс, чётко определяя **все возможные действия над ним**. Хорошим **примером** интерфейса может служить **приборная панель** автомобиля, которая **позволяет** вызвать такие методы, как увеличение скорости, торможение, поворот, переключение передач, включение фар, и т.п. То есть все действия, которые может осуществить другой класс (в нашем случае – водитель) при взаимодействии с автомобилем.

Python проектировался как объектно-ориентированный *язык программирования*. Это означает, что он построен с учетом следующих принципов:

- Все данные в нем представляются объектами.
- Программу можно составить как набор взаимодействующих объектов, посылающих друг другу сообщения.
- Каждый объект имеет собственную часть памяти и может состоять из других объектов.
- Каждый объект имеет тип.
- Все объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия).

Классы

Простейший класс:

```
class Person:
    pass # Пустой блок


p = Person()
print(p)
```

Вывод: $ python3 simplestclass.py

<__main__.Person object at 0x019F85F0>

self

Методы класса имеют одно отличие от обычных функций: они должны иметь дополнительно имя, добавляемое к **началу** списка параметров. Однако, при вызове метода никакого значения этому параметру присваивать не нужно – его укажет Python. Эта переменная указывает на **сам объект** экземпляра класса, и по традиции она называется **self**.

myobject.method(arg1, arg2)

MyClass.method(myobject, arg1, arg2)

Методы объектов

```
class Person:
    def sayHi(self):
        print('Привет! Как дела?')


p = Person()
p.sayHi()
```

Вывод: $ python3 method.py
Привет! Как дела?

Метод __init__

Существует много методов, играющих специальную роль в классах Python.

Метод **__init__** запускается, как только объект класса реализуется. Этот метод полезен для осуществления разного рода инициализации, необходимой для данного объекта. Обратите внимание на двойные подчёркивания в начале и в конце имени.

Метод __init__

```
class Person:
    def __init__(self, name):
        self.name = name
    def sayHi(self):
        print('Привет! Меня зовут', self.name)
p = Person('Swaroop')
p.sayHi()
```

Вывод: $ python3 class_init.py

Привет! Меня зовут Swaroop

Переменные класса и объекта

Данные, т.е. поля, являются обычными переменными, заключёнными в **пространствах имён** классов и объектов. Это означает, что их имена действительны только в контексте этих классов или объектов. Отсюда и название «пространство имён». Существует два типа полей: **переменные класса** и **переменные объекта**, которые различаются в зависимости от того, принадлежит ли переменная классу или объекту соответственно.

Переменные класса и объекта

Переменные класса разделяемы – доступ к ним могут получать **все** экземпляры этого класса. Переменная класса существует только **одна**, поэтому когда любой из объектов изменяет переменную класса, это изменение отразится и во всех остальных экземплярах того же класса.

Переменные объекта принадлежат **каждому** отдельному экземпляру класса. В этом случае у каждого объекта есть своя собственная **копия** поля, т.е. **не разделяемая** и никоим образом **не связанная** с другими такими же полями в других экземплярах.

```
class Robot:
    '''Представляет робота с именем.'''
    # Переменная класса, содержащая количество роботов
    population = 0

    def __init__(self, name):
        '''Инициализация данных.'''
        self.name = name
        print('(Инициализация {0})'.format(self.name))

        # При создании этой личности, робот добавляется
        # к переменной 'population'
        Robot.population += 1

    def __del__(self):
        '''Я умираю.'''
        print('{0} уничтожается!'.format(self.name))

        Robot.population -= 1

        if Robot.population == 0:
            print('{0} был последним.'.format(self.name))
        else:
            print('Осталось {0:d} работающих роботов.'.format(Robot.population))

    def sayHi(self):
        '''Приветствие робота.

        Да, они это могут.'''
        print('Приветствую! Мои хозяева называют меня {0}.'.format(self.name))
```

```
    def howMany():
        '''Выводит численность роботов.'''
        print('У нас {0:d} роботов.'.format(Robot.population))

    howMany = staticmethod(howMany)

droid1 = Robot('R2-D2')
droid1.sayHi()
Robot.howMany()
droid2 = Robot('C-3PO')
droid2.sayHi()
Robot.howMany()

print("\nЗдесь роботы могут проделать какую-то работу.\n")

print("Роботы закончили свою работу. Давайте уничтожим их.")
del droid1
del droid2

Robot.howMany()
```

Вывод:

```
$ python3 objvar.py
(Инициализация R2-D2)
Приветствую! Мои хозяева называют меня R2-D2.
У нас 1 роботов.
(Инициализация C-3PO)
Приветствую! Мои хозяева называют меня C-3PO.
У нас 2 роботов.

Здесь роботы могут проделать какую-то работу.

Роботы закончили свою работу. Давайте уничтожим их.
R2-D2 уничтожается!
Осталось 1 работающих роботов.
C-3PO уничтожается!
C-3PO был последним.
У нас 0 роботов.
```

Декораторы

Декораторы можно считать неким упрощённым способом вызова явного оператора

```
@staticmethod
def howMany():
    '''Выводит численность роботов.'''
    print('У нас {0:d} роботов.'.format(Robot.population))
```

Наследование

Одно из главных достоинств объектно-ориентированного программирования заключается в **многократном** использовании **одного и того же кода**, и один из способов этого достичь – при помощи механизма **наследования**. Легче всего представить себе наследование в виде отношения между классами как **тип** и **подтип**.

Наследование

Представим, что нам нужно написать программу, которая отслеживает информацию о преподавателях и студентах в колледже. У них есть некоторые **общие характеристики**: имя, возраст и адрес. Есть также и **специфические характеристики**, такие как зарплата, курсы и отпуск для преподавателей, а также оценки и оплата за обучение для студентов.

Можно создать для них независимые классы и работать с ними, **но** тогда **добавление** какой-либо новой общей характеристики потребует добавления её **к каждому** из этих независимых классов в отдельности, что делает программу неповоротливой.

Наследование

Лучше создать **общий** класс с именем SchoolMember, и сделать так, чтобы классы преподавателя и студента наследовали этот класс. Они станут подтипами этого типа (класса) -> можно добавить любые специфические характеристики к этим подтипам.

Достоинства:

- Если мы **добавим/изменим** какую-либо функциональность в SchoolMember, это **автоматически** отобразится и во всех подтипах. Например, новое поле удостоверения для преподавателей и студентов, можно просто добавить к классу SchoolMember.
- **Изменения в подтипах никак не влияют на другие подтипы**.
- **Обращаться к объекту преподавателя или студента можно как к объекту SchoolMember**, что может быть полезно в ряде случаев, например, для подсчёта количества человек в школе.
- **Подтип может быть подставлен в любом месте, где ожидается родительский тип**, т.е. объект считается экземпляром родительского класса, это называется **полиморфизмом**.

Наследование

Код родительского класса используется многократно, и нет необходимости копировать его во всех классы, как пришлось бы в случае использования независимых классов.

Класс SchoolMember в этой ситуации называют **базовым классом** или надклассом. Классы Teacher и Student называют **производными классами** или **подклассами.**

```
class SchoolMember:
    '''Представляет любого человека в школе.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Создан SchoolMember: {0})'.format(self.name))
    def tell(self):
        '''Вывести информацию.'''
        print('Имя:"{0}" Возраст:"{1}"'.format(self.name, self.age), end=" ")

class Teacher(SchoolMember):
    '''Представляет преподавателя.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Создан Teacher: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Зарплата: "{0:d}"'.format(self.salary))
```

Наследование

```
class Student(SchoolMember):
    '''Представляет студента.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Создан Student: {0})'.format(self.name))
    def tell(self):
        SchoolMember.tell(self)
        print('Оценки: "{0:d}"'.format(self.marks))

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)
print() # печатает пустую строку
members = [t, s]
for member in members:
    member.tell() # работает как для преподавателя, так и для студента
```

Вывод:

```
$ python3 inherit.py
(Создан SchoolMember: Mrs. Shrividya)
(Создан Teacher: Mrs. Shrividya)
(Создан SchoolMember: Swaroop)
(Создан Student: Swaroop)

Имя:"Mrs. Shrividya" Возраст:"40" Зарплата: "30000"
Имя:"Swaroop" Возраст:"25" Оценки: "75"
```

Метаклассы

Точно так же, как классы используются для создания объектов, можно использовать **метаклассы для создания классов**. Метаклассы существуют **для изменения** или **добавления нового поведения** в классы.
Допустим, мы хотим быть уверены, что мы всегда создаём исключительно **экземпляры подклассов** класса SchoolMember, и **не создаём** экземпляры **самого** класса SchoolMember.
Для этого можно использовать концепцию под названием «**абстрактные базовые классы**».
Такой класс абстрактен, т.е. является лишь некой **концепцией**, **не предназначенной** для использования в **качестве реального класса**.

Мы можем объявить наш класс как **абстрактный базовый класс** при помощи встроенного метакласса по имени **ABCMeta**

Метаклассы

Точно так же, как классы используются для создания объектов, можно использовать **метаклассы для создания классов**. Метаклассы существуют **для изменения** или **добавления нового поведения** в классы.
Допустим, мы хотим быть уверены, что мы всегда создаём исключительно **экземпляры подклассов** класса SchoolMember, и **не создаём** экземпляры **самого** класса SchoolMember.
Для этого можно использовать концепцию под названием «**абстрактные базовые классы**».
Такой класс абстрактен, т.е. является лишь некой **концепцией**, **не предназначенной** для использования в **качестве реального класса**.

Мы можем объявить наш класс как **абстрактный базовый класс** при помощи встроенного метакласса по имени **ABCMeta**

```
from abc import *
class SchoolMember(metaclass=ABCMeta):
    '''Представляет любого человека в школе.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Создан SchoolMember: {0})'.format(self.name))
    @abstractmethod
    def tell(self):
        '''Вывести информацию.'''
        print('Имя:"{0}" Возраст:"{1}"'.format(self.name, self.age), end=" ")

class Teacher(SchoolMember):
    '''Представляет преподавателя.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Создан Teacher: {0})'.format(self.name))
    def tell(self):
        SchoolMember.tell(self)
        print('Зарплата: "{0:d}"'.format(self.salary))

class Student(SchoolMember):
    '''Представляет студента.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Создан Student: {0})'.format(self.name))
    def tell(self):
        SchoolMember.tell(self)
        print('Оценки: "{0:d}"'.format(self.marks))
```

```
t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

#m = SchoolMember('abc', 10)
# Это приведёт к ошибке: "TypeError: Can't instantiate abstract class
# SchoolMember with abstract methods tell"
print() # печатает пустую строку
members = [t, s]
for member in members:
    member.tell() # работает как для преподавателя, так и для студента
```

Вывод:

```
$ python3 inherit.py
(Создан SchoolMember: Mrs. Shrividya)
(Создан Teacher: Mrs. Shrividya)
(Создан SchoolMember: Swaroop)
(Создан Student: Swaroop)

Имя:"Mrs. Shrividya" Возраст:"40" Зарплата: "30000"
Имя:"Swaroop" Возраст:"25" Оценки: "75"
```