ООП

Якупов Павел

Что такое ООП

ООП

ООП — одна из ведущий парадигм программирования в современном мире. Она чаще востребована в больших компаниях, которые стремятся съекономить огромное количество человеко-часов за счет постоянного реиспользования классов. Однако почему ООП получило такое широкое распространение?

Официальное определение

Объектно-ориентированная методология(ООП) — методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы могут образовывать иерархию наследования.

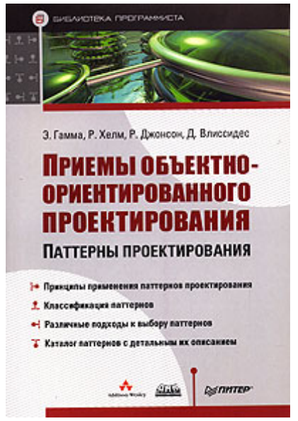
Это если в общем. А на самом деле каждый язык программирования предоставляет свои возможности к реализации ООП парадигмы. В JS и прочих скриптовых языках их возможности весьма ограничены, в языках C и Go класса как понятия нет в принципе. С другой стороны, возможности ООП в Python достаточно велики — может быть, не настолько, как в C++, C# или Java, в которых вы без классов не можете создать практически ничего.

А теперь попробуем по-простому

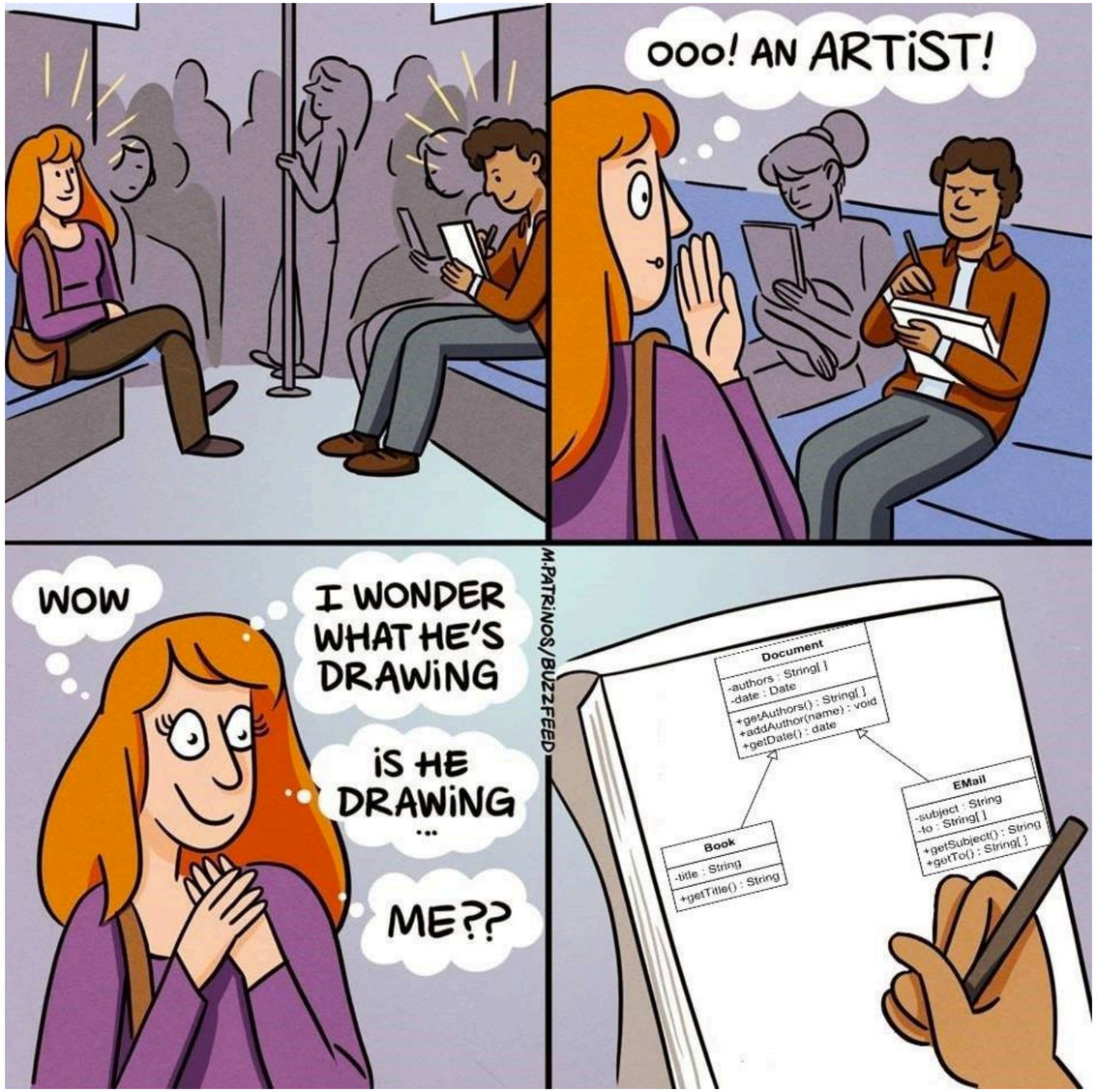
Объектно-ориентированное программирование берет концепции из реальной жизни. Вы легко можете перенести в ООП любую реальную вещь — например, автомобиль. Есть класс Автомобиль, который может создавать конкретные экземпляры автомобилей с конкретными характеристиками и методами. Несмотря на то, что существует множество экземпляров автомобилей, у всех у них есть какой-то базовые возможности - в них можно перемещаться и возить грузы, у них 4 колеса (или тогда это уже не класс автомобилей), и они потребляют, как и любые механизмы, энергия для работы.

Еще немного теории

Работа классов и объектов часто зависит и от языка, и от системы, которую надо построить. Однако есть разделы Computer Science, которые изучают ООП в отрыве от конкретной реализации. Существует даже специальный язык UML — в котором можно описывать взаимоотношения между классами и объектами. Ест легендарный труд «банды четверых», который рекомендуют для изучения ООП — программирования. Но читать его я вам рекомендую его только тогда, когда вы освоите ООП программирования на Python в полной мере.



Немного про архитектуру



ООП непосредственно в Python

ООП непосредственно в Python

В языке Python класс — это тип, который описывает набор свойств, которые характеризуют объект. Каждый **класс** имеет структуру данных, которая может содержать как функции, так и переменные, которые характеризуют объект. Членами класса могут быть функции, которые называются методами, а так же переменные (объявленные внутри структуры класса), которые именно в Python называют **атрибутами.**

ООП непосредственно в Python

Объявление класса начинается с ключевого слова class, после которого следует указанное программистом имя класса (при выборе имени придерживаются обычных правил наименований переменных и функций, но имя класса традиционно начинается с Заглавной буквы):

```
class ClassName:
    ''' строка документации '''
    объявление переменных класса
    объявление методов  класса
```

ООП непосредственно в Python

Объявление класса, в котором определяются его атрибуты и методы, является образцом, из которого могут быть произведены рабочие копии(экземпляры) класса.

Все переменные, которые объявлены внутри определения методов, известны как переменные экземпляра и доступны только локально в том методе, в котором они были объявлены — к ним нельзя обратиться извне структуры класса.

Как правило, переменные экземпляра содержат данные, которые передаются вызывающим оператором во время создания экземпляра класса. Поскольку эти данные доступны только локально (для внутреннего пользования) они фактически скрыты от остальной части программы. Такой прием называется инкапсуляцией данных, и он гарантирует, что данные надежно спрятаны только внутри класса (первый столп ООП)

ООП непосредственно в Python

Ко всем свойствам класса можно обратиться локально, используя точечную запись с префиксом self, например, атрибут с именем name можно записать следующим образом self.name. Кроме того, все определения методов должны содержать self в качестве первого аргумента.

Так же при создании экземпляра класса автоматически вызывается специальный метод __init__(self). Если необходимо передать еще значения для инициализации его атрибутов, то в скобки могут быть добавлены необходимые аргументы.

Давайте попробуем создать класс, к примеру, автомобиля:

Пример просто класса в Python

```
class Car:
    """Базовый класс автомобиля"""
    def __init__(self, marka, speed):
        """ инициализирует атрибуты марки и скорости
автомобиля"""
        self.marka = marka
        self.speed = speed
    def car_ride(self):
        return "машинка " + self.marka+" едет со скоростью "+
str(self.speed)


bmw = Car("bmw", 90)

print(bmw.car_ride())
```

Обращение к методам и атрибутам

После того, как вы создали свой экземпляр, вы можете обращаться к нему через «точечную» нотацию. К примеру, можно узнать свойства нашего экземпляра:

print(bmw.marka)

print(bmw.speed)

чтобы удалить свойство, можно воспользоваться следующим оператором del:

del bmw.speed - удаление свойства у экземпляра

Альтернативный синтаксис

Можно воспользоваться альтернативным синтаксисом Python для добавления, изменения или удаления переменной экземпляра — для этого нам могут пригодится следующие методы:

getattr(bmw, ‘speed’) — возвращает значение атрибута экземпляра класса.

hasattr(bmw, ‘speed’) — возвращает логическое True, если у объекта есть это свойство.

setattr(bmw, ‘speed’, 180) — модифицирует текущее значение, либо создает новый атрибут для экземпляра.

delattr(bmw, ‘speed’) — удаляет атрибут из экземпляра

Встроенные атрибуты

В языке Python каждый класс автоматически создается с определенным набором встроенных «частных» атрибутов. Доступ к их значениям можно получить, использую точечную запись. Например, чтобы получить значение атрибута строки документации определенного класса, вам нужно записать имя_класса.__doc__

Встроенный атрибут __dict__ является словарем, который содержит пары ключей и связанных с ними значений. Ключами здесь являются имена атрибутов, а значениями — соответствующие значения атрибута.

Встроенные атрибуты

Давайте попробуем написать программу, которая выводит нам встроенные в класс свойства и его словарь, который связан с его атрибутами, которые создали мы, и выведем их в формате ключ : значение:

```
for attr in dir(bmw):
    if attr[0] == '_':
        print(attr)
for item in bmw.__dict__:
    print(item, bmw.__dict__[item])
```

Инициализация нескольких объектов и импорт

Достаточно популярна стратегия размещения отдельного класса в отдельном файле. Если он вам нужен - вы его импортируете. Если не нужен — можете не импортировать. Зато если сложность программы возрастет, и экземпляров и классов станет очень много, вы легко сможете с ними разобраться, потому каждый класс будет в отдельном файле (и название класса желательно должно совпадать с названием файла). Пока мы не будем переименовать название файла, и сможем сделать импорт точно также, как мы это делали с функциями:

Инициализация нескольких объектов и импорт

```
from simple_class import Car


jugl = Car("jigul", 110)
audi = Car("audi", 150)
nissan = Car("nissan", 90)
print(jugl.car_ride())
print(audi.car_ride())
print(nissan.car_ride())
```

Назначение атрибута значения по умолчанию

Каждый атрибут класса должен иметь исходное значение, даже если оно равно 0 или пустой строке. В некоторых случаях (например, при задании значений по умолчанию) это исходное значение есть смысл задавать в теле метода __init__(): в таком случае передавать параметр для этого атрибута при создании объекта не обязательно.

Добавим атрибут со значением odometr_reading, исходное значение которое всегда будет равно 0. Также в класс будет включен метод read_odometer() для чтения текущих показаний одометра:

Назначение атрибута значения по умолчанию

```
class Car:
    """Базовый класс автомобиля"""
    def __init__(self, marka, speed):
        """ инициализирует атрибуты марки и скорости автомобиля"""
        self.marka = marka
        self.speed = speed
        self.odometr_reading = 0
    def car_ride(self):
        return "машинка " + self.marka+" едет со скоростью "+ str(self.speed)
    def read_odometr(self):
        """Выведет нам пробег автомобиля """
        print("У этого автомобиля пробег " + self(self.odometr_reading) + "на счетчике")
#однако 0 на счетчике держится недолго, поэтому нам потребуется метод для этого изменения
```

Изменение значения атрибута

Значение атрибута можно изменить одним из трех способов: изменить его в экземпляре, задать значение при помощи метода или изменить его с приращением (то есть прибавлением определенной величины) при помощи метода.

Рассмотрим прямое изменения значения атрибута. Для этого мы можем обратиться к нему через экземпляр:

```
 audi.odometr_reading = 100
audi.read_odometr()
#мы же не собираемся скручивать счетчик?
```

Назначение свойства через метод

Теперь давайте сделаем обновим наш код, чтобы мы могли назначать количество километров в нашем одометре через метод:

```
class Car:
…
def read_odometr(self):
        """Выведет нам пробег автомобиля """
        print("У этого автомобиля пробег " + str(self.odometr_reading) + " на счетчике")
    def update_odometr(self, mileage):
        self.odometr_reading = mileage

#теперь мы сможем обновлять значения одометра с помощью вызова метода
bmw = Car('bmw', 90)
bmw.update_odometr(290)
bmw.read_odometr()
```

Изменение атрибута с приращением

Иногда значение атрибута требуется изменить с заданным приращением (вместо того, чтобы присваивать атрибуту произвольное новое значение).

Допустим мы программируем класс, который сможет обслужить любую машину, которая приедет к нам в пункт перепродажи поддержанных автомобилей. Для этого нам для того, чтобы каждый раз не перезаписывать значения экземпляра, нужно изначально задать пробег, а потом его увеличивать, принимая на вход продолжительность и скорость во время тест-драйва.

Изменение атрибута с приращением

```
def update_odometr(self, kilo_age):
        self.odometr_reading = kilo_age
    def increment_odometr(self, kilometrs, hours):
        """ Увеличивает значения одометра с заданным приращением"""
        self.odometr_reading += kilometrs * hours

#теперь мы сможем обновлять значения одометра с помощью вызова метода
bmw = Car('bmw', 90)
bmw.update_odometr(290)
bmw.read_odometr()
bmw.increment_odometr(100, 3)
bmw.read_odometr()
```

Наследование

Работа на новый классом не обязана начинаться с нуля. Если класс, который вы пишите, представляет собой специализированную версию ранее написанного класса, вы можете воспользоваться наследованием. Один класс, наследующий от другого, автоматически получает все атрибуты и методы первого класса. Исходный класс тогда называется родителем, а новый класс — потомком. Класс-потомок наследует атрибуты и методы родителя, но при этом может определять свои собственные атрибуты и методы.

Метод __init__()

Первое, что делает Python при создании экземпляра класса- потомка — присваивает значения всем атрибутам класса-родителя. Для этого методу __init__ класса потомка необходима помощь со стороны родителя.

К нашему базовому класса обычного автомобиля мы попробуем создать класс-потомок электроавтомобиль. Для того, чтобы один класс смог унаследовать другой, класс родитель нужно передать в качестве аргумента класс-предку.

Создание класса потомка

```
class ElectricCar(Car):
    """ Представляет класс автомобиля с элетродвигателем"""
    def __init__(self, marka, speed):
        """ Инициализирует атрибуты класса -родителя"""
        super().__init__(marka, speed)


myTesla = ElectricCar('tesla', 300)
print(myTesla.car_ride())
```


Я НА ГЛУБИНЕ 300 МЕТРОВ.
ЗДЕСЬ НА МЕНЯ ОКАЗЫВАЕТСЯ
ОГРОМНОЕ ДАВЛЕНИЕ...
ПОЧЕМУ В PYTHON НЕТ
ПОДДЕРЖКИ МНОЖЕСТВЕННОГО
НАСЛЕДОВАНИЯ?
risovach.ru

Функция super()

Функция super() — специальная функция, которая помогает Python связать потомка с родителем. Эта строка приказывает Python вызвать метод __init__. класса, который является родителем ElectricCar, в результате чего экземпляр ElectricCar получает все атрибуты класса родителя. Имя super происходит из распространённой терминологии: класс родитель называется суперклассом, а класс-потомок — субклассом.

Мы можем посмотреть, какое состояние счетчика у нашей новой Теслы:

```
myTesla.read_odometr()
```

Определение атрибутов и методов класса-потомка

После создания класса — потомка, наследующего от класса-родителя, можно переходить к добавлению новых атрибутов и методов, необходимых для того, чтобы потомок отличался от родителя.

Давайте добавим атрибут, который специфичен именно для электроавтомобилей (к примеру, мощность аккумулятора) и метод, который у нас будет выводить значение этого атрибута:

Экземпляры как атрибуты

При моделировании явлений реального мира в программах классы нередко дополняются все большим количеством подробностей. Списки атрибутов и методов растут, и через какое-то время файлы становятся слишком длинными и громоздкими. В такой ситуации часть одного класса нередко стоит записать в виде отдельного класса. Один большой класс разбивается на несколько небольших, которые куда удобнее тестировать.

Например, при дальнейшей доработке класса ElecticalCar может оказаться, что в нем появилось слишком много новых атрибутов и методов, которые относятся только к аккумулятору. В таком случае куда логичнее создать отдельный класс Battery:

Импортирование классов

Мы с вами уже успели посмотреть на импортирование функций из модуля, и в общем импортирование классов работает по такой же логике. Давайте быстренько повторим:

если в вашем модуле(файле) заключено несколько классов, вы можете импортировать из модуля те классы, которые вам нужны — from car import Car, ElectricCar

Так же импортировать весь модуль целиком — import car, и потом обращаться к каждому классу как к car.Electric_Car()

можно вывалить все в глобальную область видимости — from car import *

Реализация полиморфизма

Полиморфизм является одним из крауегоульных камней ООП; Полиморфизм(гр. «много форм») описывает способность назначать элементу различные смысловые значения в зависимости от контекста, в котором он используется. В языке Python например, такой элемент, как + , может быть классифицирован как полиморфный, потому что он представляет из себя либо оператор арифметической операции, либо оператор конкатенации строк. Поэтому в данном контексте под полиморфизмом понимается множество форм одного и того же слова — имени метода.

Статичные методы

Иногда вы не хотите создавать экземпляр класса, а хотите воспользоваться каким-то методом класса как шаблоном. Для этого вам может понадобиться статичные методы @staticmethod:

```
@staticmethod
def get_class_details():
    print("Это класс Car")

Car.get_class_details()
```

Стандартная библиотека Python

Стандартная библиотека Python представляет собой набор модулей, включаемых в каждую установленную копию интерпретатора Python. Чтобы использовать любую команду или класс из стандартной библиотеки, достаточно включить простую команду import в начало файла. Для примера мы можем рассмотреть класс *OrderDict* из модуля *collections.*

Как вы уже знаете, словари позволяют связывать информационные фрагменты, но они не отслеживают порядок добавления пар «ключ» — «значение». Если вы хотите создать словарь , но при это сохранит порядок добавления ваших ключей и значений, тогда вам потребуется импорт.

Работа с неупорядоченным словарем

```
favorite_languages = {}
favorite_languages['jen'] = 'python'
favorite_languages['sarah'] ='c'
favorite_languages['phil'] =  'python'
favorite_languages['edward'] = 'ruby'


# friends = ['phil', 'sarah']
print(favorite_languages)
```

Использьвование **OrderedDict**

```
from _collections import OrderedDict

favorite_languages = OrderedDict()

favorite_languages['jen'] = 'python'
favorite_languages['sarah'] = 'c'
favorite_languages['edward'] = 'ruby'
favorite_languages['phil'] = 'python'

print(favorite_languages)
```

Оформление классов

Имена классов должны записываться в «верблюжной» схеме.

Имена экземпляров и модулей записываются в нижнем регистре с разделением слов символами подчеркивания.

Каждый класс должен иметь строку документации, которая следует сразу за определением класса. Строка документации должна содержать краткое описание того, что делает класс, и в ней должны соблюдаться те соглашения по форматированию, которые вы использовали для написания строк документации в функциях.