ООП в Python

Что такое ООП?

Объектно-ориентиированное программирование (ООП) —методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Основные принципы ООП

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

Абстракция

Абстракция – выделение в моделируемом реальном объекте важных для решения конкретной задачи свойств и методов для создания класса.

Например, рассмотрим класс СТОЛ:

1. Разрабатываем приложение для перевозки мебели, какие параметры необходимо добавить классу СТОЛ ?

2. Разрабатываем приложение для бронирования мест в ресторане, какие параметры необходимо добавить классу СТОЛ ?

Что такое объекты?

Объект — это набор данных (переменных) и методов (функций).

Можно думать об объектах как о существительных, а об их методах —как о глаголах.

На объектах и классах строится всё ООП. А чем они отличаются друг от друга?

- **Класс** — это тип данных. Он содержит разные свойства и метод.
- **Объект** — это экземпляр класса, или его копия, которая находится в памяти компьютера.

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

    def info(self):
        print(f"Меня зовут {self.name}, мне {self.age} лет.")

p1 = Person("Алиса", 25)
p2 = Person("Боб", 30)

p1.info()  # Меня зовут Алиса, мне 25 лет.
p2.info()  # Меня зовут Боб, мне 30 лет.
```

конструктор

Метод **__init__** — **конструктор** класса. Он вызывается сразу после создания объекта, чтобы присваивать значения динамическим атрибутам.

self — ссылка на текущий объект, она даёт доступ к атрибутам и методам класса.

Деструктор вызывается при удалении объекта. Деструктор представляет собой метод __del__(self), в который, как и в конструктор, передается ссылка на текущий объект.

```
1  class Person:
2
3      def __init__(self, name):
4          self.name = name
5          print("Создан человек с именем", self.name)
6
7      def __del__(self):
8          print("Удален человек с именем", self.name)
9
10 tom = Person("Tom")
```

```
Создан человек с именем Tom
Удален человек с именем Tom
```

```python
class FileHandler:  1 usage
    def __init__(self, filename):
        self.file = open(filename, 'w')
        print(f"Файл {filename} открыт")

    def write(self, text):  1 usage
        self.file.write(text)

    def __del__(self):
        self.file.close()
        print("Файл закрыт и объект удалён")

handler = FileHandler("test.txt")    # Создаём объект
handler.write("Hello, world!")       # Работаем с объектом

del handler  # Удаляем объект
```

В деструкторе определяются действия, которые надо выполнить при удалении объекта, например, освобождение или удаление каких-то ресурсов, которые использовал объект.

Выполнение →

```
Файл test.txt открыт
Файл закрыт и объект удалён
```

```python
class Person:

    def __init__(self, name):
        self.name = name
        print("Создан человек с именем", self.name)

    def __del__(self):
        print("Удален человек с именем", self.name)


def create_person():
    tom = Person("Tom")

create_person()
print("Конец программы")
```

```
Создан человек с именем Tom
Удален человек с именем Tom
Конец программы
```

Здесь объект Person создается и используется внутри функции create_person, поэтому жизнь создаваемого объекта Person ограничена областью этой функции.
Соответственно, когда функция завершит свое выполнение, у объекта Person будет автоматически вызываться деструктор.

Инкапсуляция

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

    @property  2 usages
    def age(self):
        return self._age

    @age.setter  1 usage
    def age(self, new_age):
        if new_age > self._age:
            self._age = new_age
        return self._age
```

аннотации свойств

геттер

сеттер

Символ _ делает атрибуты закрытыми.

Но как обратиться к закрытым атрибутам?

Что бы изменить возраст, достаточно просто написать

cat.age = 5

Пример кода с функциями геттера и сеттера, без применения аннотаций свойств

```python
class Person:
    def __init__(self, name, age):
        self.__name = name    # устанавливаем имя
        self.__age = age       # устанавливаем возраст

    # сеттер для установки возраста
    def set_age(self, age):
        if 0 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

    # геттер для получения возраста
    def get_age(self):
        return self.__age

    # геттер для получения имени
    def get_name(self):
        return self.__name

    def print_person(self):
        print(f"Имя: {self.__name}\tВозраст: {self.__age}")
```

Можно обойтись и без аннотаций свойств, но тогда изменение или получение параметра будет осуществляться вызовом функций сеттеров и геттеров, что уже интуитивно менее понятно, чем просто обращение к свойству

```python
tom = Person("Tom", 39)
tom.print_person()  # Имя: Tom  Возраст: 39
tom.set_age(-3486)  # Недопустимый возраст
tom.set_age(25)
tom.print_person()  # Имя: Tom  Возраст: 25
```

Наследование

```python
class Person:  1 usage
    def __init__(self, name):
        self.__name = name  # имя человека
    @property  2 usages
    def name(self):
        return self.__name
    def display_info(self):  1 usage
        print(f"Name: {self.__name} ")

class Employee(Person):  1 usage
    def work(self):  1 usage
        print(f"{self.name} works")

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

Множественное наследование

```
10  class Employee(Person):  2 usages
11      def work(self):  1 usage
12          print(f"{self.name} works")
13
14  class Student:  1 usage
15      def study(self):
16          print("Student studies")
17
18  class WorkingStudent(Employee, Student):
19      pass
```

В Python поддерживается множественное наследование

```python
class Employee:
    def do(self):
        print("Employee works")

class Student:
    def do(self):
        print("Student studies")


# class WorkingStudent(Student,Employee):
class WorkingStudent(Employee, Student):
    pass

tom = WorkingStudent()
tom.do()     # ?
```

Проблема:

Метод do() определен в обоих базовых классах, какая реализация метода вызовется в классе потомке?

Первым в списке базовых классов идет класс Employee, поэтому реализация метода do будут браться из класса Employee.

```python
class Person:

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

```python
class Employee(Person):

    def __init__(self, name, company):
        super().__init__(name)
        self.company = company
```

Здесь в классе Employee добавляется новый атрибут - company.

Соответственно метод __init__() принимает три параметра: второй для установки имени и третий для установки компании.

Но если в базом классе определен конструктор с помощью метода __init__, и мы хотим в производном классе изменить логику конструктора, то в конструкторе производного класса мы должны вызвать конструктор базового класса. То есть в конструкторе Employee надо вызвать конструктор класса Person.

Для обращения к базовому классу используется выражение **super(),** которое вызывает конструктора класса Person

```python
class Person:

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

```python
class Employee(Person):

    def display_info(self):
        super().display_info()
        print(f"Company: {self.company}")
```

```python
tom = Employee("Tom", "Microsoft")
tom.display_info()
```

```
Name: Tom
Company: Microsoft
```

Здесь при переопределении метода **display_info()** необходимо повторить вывод с именем, но что бы не повторять код в разных классах тоже можно использовать выражение **super().**

Атрибуты класса

```
class Person:
    type = "Person"
    description = "Describes a person"


print(Person.type)          # Person
print(Person.description)   # Describes a person

Person.type = "Class Person"
print(Person.type)          # Class Person
```

```
tom = Person("Tom")
bob = Person("Bob")
print(tom.type)     # Person
print(bob.type)     # Person
```

Статические методы

Кроме обычных методов класс может определять статические методы. Такие методы предваряются аннотацией **@staticmethod** и относятся в целом к классу.

```
1  class Person:
2      __type = "Person"
3
4      @staticmethod
5      def print_type():
6          print(Person.__type)
7
8
9  Person.print_type()      # Person - обращение к статическому методу через имя класса
10
11 tom = Person()
12 tom.print_type()      # Person - обращение к статическому методу через имя объекта
```

Класс object

В языке программирования Python все классы неявно имеют один общий суперкласс - ***object*** и все классы по умолчанию наследуют его методы.

Один из наиболее используемых методов класса object - метод ***__str__()***.

```
1  class Person:
2      def __init__(self, name, age):
3          self.name = name  # устанавливаем имя
4          self.age = age  # устанавливаем возраст
5
6      def display_info(self):
7          print(f"Name: {self.name}  Age: {self.age}")
8
9
10 tom = Person("Tom", 23)
11 print(tom)
```

```
<__main__.Person object at 0x10a63dc00>
```

Пример

```python
class Person:
    def __init__(self, name, age):
        self.name = name  # устанавливаем имя
        self.age = age  # устанавливаем возраст

    def display_info(self):
        print(self)
        # print(self.__str__())     # или так

    def __str__(self):
        return f"Name: {self.name}  Age: {self.age}"


tom = Person("Tom", 23)
print(tom)          # Name: Tom  Age: 23
tom.display_info()  # Name: Tom  Age: 23
```

```
Name: Tom  Age: 23
```

Переопределение метода __str__() в классе Person

Магические методы в Python

Магические методы (или dunder-методы) – это специальные методы в Python, которые начинаются и заканчиваются двойным подчеркиванием (__method__). Они позволяют изменять стандартное поведение объектов, перегружать операторы и делать классы более удобными в использовании.

Магические методы как правило не вызываются напрямую, а вызываются встроенными функциями или операторами, например:

print() вызывает __str__()

some_ob=SomeObject() вызывает __new__(),
который в свою очередь вызывает __init__()

сложение (x + y) вызывает __add__()

вычитание (x - y) вызывает __sub__() и т.д.

Некоторые магические методы и соответствующие им операторы

Операция	Синтаксис	Функция
Сложение	a + b	__add__(a, b)
Объединение	seq1 + seq2	__concat__(seq1, seq2)
Деление	a / b	__truediv__(a, b)
Целочисленное деление	a // b	__floordiv__(a, b)
Поразрядное И	a & b	__and__(a, b)
Двоичное ИЛИ	a | b	__or__(a, b)
Двоичный xor	a ^ b	__xor__(a, b)
Равенство	a == b	__eq__(a, b)
Меньше или равно	a <= b	__le__(a, b)
Больше или равно	a <= b	__ge__(a, b)

Больше операций и соответствующих им методов можно посмотреть здесь
https://metanit.com/python/tutorial/7.7.php

Пример перегрузки операторов

```python
class Vector:  4 usages
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector( x: 1,  y: 2)
v2 = Vector( x: 3,  y: 4)

print(v1 + v2)  # Vector(4, 6)
print(v1 - v2)  # Vector(-2, -2)
```

Пример переопределения арифметических операторов

```python
class Product:  2 usages
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def __eq__(self, other):
        return self.price == other.price

    def __lt__(self, other):
        return self.price < other.price

p1 = Product( name: "Телефон",  price: 500)
p2 = Product( name: "Планшет",  price: 700)

print(p1 == p2)  # False
print(p1 < p2)   # True
print(p1 > p2)   # False
```

Пример перегрузки операторов сравнения

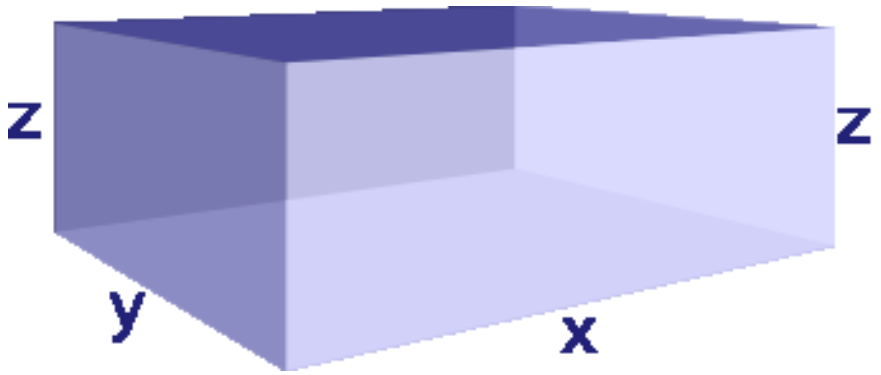
Как будут здесь сравниваться товары?

Композиция пример

Требуется написать программу, которая вычисляет площадь обоев для оклеивания помещения. При этом окна, двери, пол и потолок оклеивать не надо.

Площадь стен комнаты S = 2xz + 2yz = 2z(x+y).

Потом из этой площади надо будет вычесть общую площадь дверей и окон, поскольку они не оклеиваются.



В дверях и окнах нас интересует только их площадь, создадим для них общий класс:

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

```python
class Room:
    def __init__(self, x, y, z):
        self.square = 2 * z * (x + y)
        self.wd = []
    def addWD(self, w, h):
        self.wd.append(WinDoor(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) # выведет 48.6
r1.addWD(1, 1)
r1.addWD(1, 1)
r1.addWD(1, 2)
print(r1.workSurface()) # выведет 44.6
```

Класс "комната" – это класс-контейнер для окон и дверей. Он должен содержать вызовы класса "окно_дверь".

При создании объекта класса-контейнера или вызове его методов, также создаются объекты включенных в него классов. Это и есть композиция.