ООП:

Основные понятие

Класс- шаблон на основе которого будет создаваться объект. По своим свойствам напоминает тип данных. Пример создания класса человек.

```
class Person:
        pass
```

Объект - переменная (конкретно в языке Python любая переменная - объект), созданная на основе шаблона. Пример создания объекта.

```
person=Person()
# person объект, а Person() -класс
```

Пока выглядит бесполезно . Нужно добавить функциональности.

Метод- функция, определенная лишь в конкретном типе данных. Что это значит и чем это отличается от функции ? Разберем на конкретном примере.

```
simple_list=list([1,2,3]) # создаем переменную с типом данных list он же класс list
 # simple_list объект, а list() -класс
```

Допустим, мы хотим добавить в наш список еще один элемент. Что мы сделаем ? Воспользуемся методом .append.

```
simple_list.append(4)
print(simple_list)
# output : [1,2,3,4]
```

Попробуем воспользоваться этим же методом с другим типом данных, например, int.

```
a=int(10)
a.append(20)
# output : AttributeError: 'int' object has no attribute 'append'
```

Какой из этого можно сделать вывод ? **Метод может быть вызван только с конкретным типом данных/ классом, тогда как функция может быть вызвана для любого типа.**

Вернемся к нашему классу Person.

```
class Person:
        def move(self):
                print('move')
```

move - метод движения , методы создаются также как и функции, но должны иметь обязательный первый аргумент self. Его мы использовать не будем, но его **обязательно** нужно указать в скобках, такой синтаксис.

```
person.move() # вызываем метод move
```

Для чего же нужен Аргумент self ? Давайте это проверим, модифицировав метод move.

```
class Person:
        def move(self):
                return self
person=Person()
print(person.move())
# out : <__main__.Person object at 0x7f83bbcc00d0>
```

Таким образом, мы выяснили, что self является ссылкой на объект класса.

Поле/Атрибут- переменная внутри класса. В чем ее преимущество ? Разберем на примере.

Допустим, мы хотим описать человека. Какие данные нам нужны ? Пусть это будут имя и возраст , а также функционал движения . Реализуем это уже знакомыми нам средствами.

```
name='Misha'
age=22
def move():
  print('move')
```

Мы создали 2 переменные. Кажется, что с помощью этого мы можем описать человека, но нет. Что если нам нужно описать 10, 1000, 100000 человек ? Для этого нам надо структурировать данные с помощью ООП.

```
class Person:
    name='Misha'
    age=22 # name и age поля класса
    def move(self):
        print('move')
person1=Person()# person - это объект, а Person -класс
person2=Person()
person2.name='Vasya'
person2.age=31
print(person1.name,person1.age)
print(person2.name,person2.age)
# output : Misha 22 \n Vasya 31
```

Магические методы

Конструктор (метод __init__)

Самый важный из всех магических методов. Отвечает за инициализацию полей класса .

```
class Person:
    def __init__(self,name,age): # __init__ - конуструктор.Его имя обязательно должно быть таким  . self.name -поле класса, name -аргумент
        self.name=name
        self.age=age
person=Person('Misha',22)# person - это объект, а Person -класс
print(person.name,person.age) # результат - Misha 22
```

Метод __str__

```
class Person:
    def __init__(self,name,age): # __init__ - конуструктор.Его имя обязательно должно быть таким  . self.name -поле класса, name -аргумент
        self.name=name
        self.age=age
    def __str__(self):
        return f'Имя {self.name}. Возраст {self.age}'
person=Person('Misha',22)# person - это объект, а Person -класс
print(person) # результат - Имя Misha. Возраст 22
```

Деструктор (Метод __del__)

```
class Person:
    def __init__(self,name,age): # __init__ - конуструктор.Его имя обязательно должно быть таким  . self.name -поле класса, name -аргумент
        self.name=name
        self.age=age
    def __str__(self):
        return f'Имя {self.name}. Возраст {self.age}'
    def __del__(self):
        print('Объект удален')
person=Person('Misha',22)# person - это объект, а Person -класс
del person# результат - Объект удален
```

Также существуют другие методы. Ознакомьтесь с ними самостоятельно.

Отношения между классами

Наследование

```
class Monkey:
    def making_a_tool(self):
        print('Making a tool')

class Human(Monkey):
    def making_a_tool(self):
        super().making_a_tool()
    def developing_on_python(self):
        print('Developing on python language')


human=Human()
human.making_a_tool()
human.developing_on_python()
monkey=Monkey()
monkey.making_a_tool()
monkey.developing_on_python()
# out Making a tool
# Developing on python language
# Making a tool
# Traceback (most recent call last):
# File "/home/main.py", line 27, in <module>
#   monkey.developing_on_python()
# AttributeError: 'Monkey' object has no attribute 'developing_on_python'
```

Таким образом, можно сделать вывод, что класс наследник имеет функционал класса родителя, но не наоборот.

Агрегация

```
class Car:
    def __init__(self,engine):
        self.engine=engine

class Engine:
    def __init__(self,power)
        self.power=power

engine=Engine(power=250)
car=Car(engine)
```

Композиция

```
class Engine:
    def __init__(self,power)
```

```
        self.power=power

class Car:
    def __init__(self,power):
        self.engine=engine(power)

engine=Engine(power=250)
car=Car(engine)
```

Ассоциация

```
class Engine:
    def __init__(self,power=None)
        self.power=power

class Car:
    def __init__(self):
        self.engine=Engine()

engine=Engine(power=250)
car=Car()
```