Object Oriented Programming in Python

Advantages of OOP over function :

1. **Code Reuse** : Let suppose a computer game. There may be many `trees`, `pedestrians`, `cars` and much more. Here `pedestrians` is a object which code will be written once and used multiple times.

   OOP is a smart code reuse.

2. **Encapsulation** : Programmer should not get full access of the code. We can make some data private, some protected with the help of OOP. We can make sure that programmer cant change value of some variables what we want not to be changed.
3. **Polymorphism** : If there is two similar functions, we will not write their code separately.

Classes, Objects and Constructors :

- **Analogy** : Class is like a template of PowerPoint. Which we use and edit some changes as per our requirements for different projects.
- Without using class :

```
class Employee:
    pass

# All Employees are similar so we made them from Employee template.
harry = Employee()
rohan = Employee()

# Methods & Attributes are used to set name, email, salary etc.
# Functions are used to increase salary to Specific employee

# Without Using class
harry.fname = 'harry'
harry.lname = 'jackson'
harry.salary = 44000

rohan.fname = 'rohan'
rohan.lname = 'das'
rohan.salary = 4000

print(harry.salary)  #Outputs 44000
```

- Using Class :

```
# We will increase code reusability by class
class Employee:
    # initialise constructor
    def __init__(self,fname,lname,salary):
        self.fname = fname
        self.lname = lname
        self.salary = salary

# Creating different employees from Employee template
harry = Employee('harry','jackson','44000')
rohan = Employee('rohan','das','64000')

print(harry.salary)  #Outputs 44000
```

`self` should be given in any normal class methods or constructors because, in Python, `self` is automatically passed when object is created.

Instance & Class Variables :

Instance Variables : Those which in `__init__` function

Class Variables : There will be some variables which will common for all employees (like employee total number, common increment). Those variables we save into Class Variables.

```
class Employee:
    increment = 1.5
    def __init__(self,fname,lname,salary):
        # instance variables
        self.fname = fname
        self.lname = lname
        self.salary = salary
        self.increment = 1.2

    def increase(self):
        self.salary = int(float(self.salary) * self.increment)

# Creating different employees from Employee template
harry = Employee('harry','jackson','44000')
rohan = Employee('rohan','das','64000')

print(harry.salary)  #Outputs 44000
harry.increase()
print(harry.salary)
```

- Case 1 -> `self.salary = self.salary * self.increment` : Here it search for `increment` in Instance variable first. If not found, then it search in class variables.
- Case 2 -> `self.salary = self.salary * Employee.increment` : Here it search for `increment` directly in class variable.
- Case 3 -> `self.salary = self.salary * increment` : Error (No variable found)

To see all the Instance variables : (`.__dict__`)

```
print(harry.__dict__)
# output: {'fname': 'harry', 'lname': 'jackson', 'salary': 66000}
```

To add a instance variable for a particular Employee :

```
harry.email = 'harry@gmail.com'
```

To see all Class Variables :

```
print(Employee.__dict__)
# Output: {'__module__': '__main__', 'increment': 1.5, '__init__': <function
Employee.__init__ at 0x009F7898>, 'increase': <function Employee.increase at
0x00B12190>, '__dict__': <attribute '__dict__' of 'Employee' objects>,
'__weakref__': <attribute '__weakref__' of 'Employee' objects>, '__doc__': None}
```

To count number of objects created:

```
class Employee:
    # class variable
    increment = 1.5
    no_of_employees = 0
    # initialise constructor
    def __init__(self,fname,lname,salary):
        self.fname = fname
        self.lname = lname
        self.salary = salary
        self.increment = 1.2
        Employee.no_of_employees += 1

print(Employee.no_of_employees) # Output: 0
# Lets create 2 objects of Employee class
harry = Employee('harry','jackson','44000')
rohan = Employee('rohan','das','64000')
print(Employee.no_of_employees) # Output: 2
```

Class Methods :

- We can define Class Methods by `@classmethod` syntax on the top of method.
- Class Methods take whole class as a argument. But the normal methods takes object as an argument(not class).

```
# We will increase code reusability by class
class Employee:
    # class variable
    increment = 1.5
    no_of_employees = 0
    # initialise constructor
    def __init__(self,fname,lname,salary):
        self.fname = fname
        self.lname = lname
        self.salary = salary
        self.increment = 1.2
```

```
        Employee.no_of_employees += 1
# normal method
    def increase(self):
        self.salary = int(float(self.salary) * Employee.increment)

# class methods takes whole class as a argument.
    @classmethod
    def change_increment(cls,amount):
        cls.increment = amount

# Creating different employees from Employee template
harry = Employee('harry','jackson','44000')
rohan = Employee('rohan','das','64000')

print(harry.salary)  #Outputs 44000
# harry.increase()
Employee.change_increment(2)
harry.increase() #automatically takes self as a argument
print(harry.salary) #Outputs 88000
```

- **Class Methods as alternative Constructor :**

- **Static Methods** :

Inheritance :