OOPs :

OOP stands for Object oriented programming. s in OOPs stands for System.

Real-time entities are called objects.

We use OOPs concept for large & complex projects.

In OOPs concept we have class and object.

Class : It acts as blue print / plan/model/design for objects

Object : A physical existence of a class.

We can create any number of objects after creating a class.

In Object oriented programming class name should start with upper case.

Every object has properties and behaviour. Where properties(data) are specified by variables and behaviours can be specified by methods.

Variables:

Variables are of 3 types:

- Instance variable
- Static variable
- Local variable

1. Instance Variable

If the value of the variable is varied from object to object, such type of variables are called instance variabl/ object level variable. For every object a separate copy of variable will be created. In general we define instance variables inside constructor by using self. And self is not a key-word, instead of self we can take delf.

```
# eg:

class Student:

  def __init__(delf, name):

      delf.name = name

      delf.course = 'FSDS'

s = Student('Sai')

print(s.name)

print(s.course)
```

```
Sai
FSDS
```

2. Static variable

- If the value of the variable is not varied from object to object then it is recommended to declare that variable as static variable or class lavel variable.
- Inside class if we are declaring a variable directly then it is called static variable.
- In case of static variable a single copy will be created and shared to every object of the class.

```
#eg:

class Student:

  x= 10

  def profile(self):

      self.name = 'Sai'

s = Student()

print(s.x)
```

```
10
```

3. Local variable

- To meet temporary requirement of the programme, we can declare variables directly inside a method.
- These variables are temporary variables. And pvm destroys them once method execution completes

```
#eg:

class Student:

  def profile(self):

      y = 20

      print(y)

s = Student()

s.profile()
```

```
20
```

Referance variable

- Referance variable can be used to to refer an object, by using that variable we can invoke the functionality of that object.
- When we crate an object constructor will get created, and the variables related to object get initialized.

self

- Inside class to access the instance variables we use self. Outside the class we use referance variable.
- self referance variable is always pointing to current object within the python class.
- The first argument of constructor, instance method is always self.
- And we never provide the value of self, pvm is responsible to provide value.

OOPs ---->> **Methods**

There are 3 types of methods

1. Instance method
2. Class Method
3. Static Method

1. Instance method

- Inside a method if we are trying to access instance variables(wheather static and local variable are present or not) then this type of method is called instance method. We call these methods using object referance to access the object related properties, as these methods are related object.
- The first argument inside instance method is self.

```
# eg:
class Student:

  def __init__(delf, name):

      delf.name = name

      delf.course = 'FSDS'

  def profile(self):
      print(self.name)

s = Student('Sai')

s.profile()
```

```
Sai
```

2. Class method

- Inside method implementation if we are using only static variable and we are not using atleast one instance variable, then we go for class method.
- We define a class method using @classmethod decorator, and cls variable.
- By using class name or object referance we cal call these methods

```
# eg:

class Student:

  institute = 'iNeuron'

  @classmethod

  def get_student_info(cls):

      print(cls.institute)

Student.get_student_info()

     iNeuron
```

3. Static method

- If we are not using any instance variable or static variable inside a method body, such type of method is called static method.
- Static method is no way related to class or object. It is a general utility method.
- We use @staticmethod decorator to define these methods, and it is optional.
- We call a static method using class name or object referance. But class name is recommended.

```
class Student:

  @staticmethod
  def get_student_info():

      institute = 'iNeuron'

      print(institute)

  def details():

      y = 10

      print(y)

Student.get_student_info()

Student.details()

     iNeuron
     10
```

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Decorator :

Decorator is a function which takes a function as input argument and extend it's functionality and returns modified function with extended functionality.

- Decorator is a function. It always takes a function as argument(input function).
- The biggest advantage of this function is, i) Code reusability, ii) DRY principle (Don't repeat yourself)
- In python every function is treated as object internally.
- Decorator creates a new function with extended functionality. In this new function we may use the original function(input function). And it will return the extended function as output.
- Without affecting the original function we can extend its functionality using decorator.

```
# eg:

def decor(func):
    def inner():
        print('I am Sai Subhasish')
        print('I am a DataScientist')
    return inner

@decor
def display():
    print('I am Sai Subhasish')

display()
```

```
I am Sai Subhasish
I am a DataScientist
```

```
def decor(func):
    def inner():
        print('I am Sai Subhasish')
        print('I am a DataScientist')
    return inner

@decor
def display():
    pass

display()
```

```
I am Sai Subhasish
I am a DataScientist
```

```
def display():
    print('I am Sai Subhasish')

display()
```

```
I am Sai Subhasish
```

- In the above program when we call display() function, pvm is going to check if there is any decorator is present or not. If it is not there then original display() function will get executed.
- If decorator is present then the decorator() function will get executed. As here decorator() function is returning inner() function so it is executing inner() function.
- Here @decorator and inner() function name can be anything.

```
def fecorator(funcion):
    def fsds():
        print('I am Sai Subhasish')
        print('I am a DataScientist')
    return fsds

@fecorator
def profile():
    print('I am Sai Subhasish')

profile()
```

```
I am Sai Subhasish
I am a DataScientist
```

- Decorator() function must have return statement

```
def decorator(func):
    def fsds():
        print('I am Sai Subhasish')
        print('I am a DataScientist')
    return fsds

@decorator
def profile():
    pass

profile()
```

```
I am Sai Subhasish
I am a DataScientist
```

```
def decorator(func):
    pass

@decorator
def profile():
    pass

profile()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-b0246f1b2fb9> in <cell line: 8>()
      6     pass
      7
----> 8 profile()

TypeError: 'NoneType' object is not callable
```

SEARCH STACK OVERFLOW

```
def decorator(func):
    def fsds():
        pass

@decorator
def profile():
    pass

profile()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-66a6cb7782b1> in <cell line: 9>()
      7     pass
      8
----> 9 profile()

TypeError: 'NoneType' object is not callable
```

SEARCH STACK OVERFLOW

```
def decorator(func):
    def fsds():
        pass
    return fsds

@decorator
def profile():
    pass

profile()
```

- Practical example using decorator

```
def decor_for_add(func):
    def inner(x, y):
        print('#'*40)
        print('The sum of {} and {}'.format(x,y), end='')
        func(x,y)    # Calling original function i.e add()
        print('#'*40)
    return inner

@decor_for_add
def add(a,b):
    print(a+b)

add(10,20)
```

```
########################################
The sum of 10 and 2030
########################################
```

Who is calling decorator ?

- When we are writing @decorator, pvm is going to pass only one input function as argument.
- *If you want to call same function with and without decorator then you need to call the decorator function.**

Decorator chaining

We can configure multiple decorators for same function and this concept is known as decorator chaining.

```
def decor1(func):
    def inner1():
        print('Decor1 execution')  ## 2
        func()    ## Here original_func() will get executed
    return inner1

def decor2(func):
    def inner2():
        print('decor2 execution')  ## 1
        func()  ## Here inner1() will get executed
    return inner2  ## Here inner2() will get executed

@decor2
@decor1
def original_func():
    print(f'Original function')  ## 3

original_func()
```

```
decor2 execution
Decor1 execution
Original function
```

```
def decor1(func):
    def inner1(a,b):
        print('Output of decor1 : {}'.format(a+b))
    return inner1

def decor2(func):
    def inner2(x,y):
        print('Output of decor2 : {}'.format(x*y))
    return inner2

@decor2
@decor1
def original_func(p,q):
    print(f'Original value {p}, {q}')

original_func(10,20)
```

```
Output of decor2 : 200
```

Above program exection flow :

original_func()---->>[decor1]---->>inner1(), inner1()---->>[decor2]---->>inner2()

```
#  Let's change the order

def decor1(func):
    def inner1(a,b):
        print('Output of decor1 : {}'.format(a+b))
    return inner1

def decor2(func):
    def inner2(x,y):
        print('Output of decor2 : {}'.format(x*y))
    return inner2

@decor1
@decor2
def original_func(p,q):
    print(f'Original value {p}, {q}')

original_func(10,20)
```

```
Output of decor1 : 30
```

Above program exection flow :

original_func()---->>[decor2]---->>inner2(), inner2()---->>[decor1]---->>inner1()

```
## Decorator call will execute decorator function first then normal function, here we are passing the *args and **kwargs to accept any ar
```

```
#eg 2:

from datetime import datetime
```

```
def log_date_time(func):
  def inner(*args, **kwargs):
    print(f'Date and Time: {datetime.now()}')
    return func(*args, **kwargs)
  return inner

@log_date_time
def greet(name):
  print(f'Hello {name}')

greet('Sai')
```

```
Date and Time: 2023-06-24 08:24:37.453576
Hello Sai
```

Here are some basic questions which will help you to better understand the topic

Q1. What is the purpose of Python's OOP?

Ans.

In python Object Oriented Programming deals with the concept of class and Object. It is used to structure the python program into simple and reusable form(classes) of which we can create instance/object. It aims to implement inheritance, abstraction, polymorphism, encapsulation.

Q2. Where does an inheritance search look for an attribute?

Ans.

In inheritance what ever methods, variables, constructors parent class has by-default available to child class, there is not required to redefine them in child class. In case of code execution inheritance search look for attributes of parent class.

We use inheritance for code reusability and extendability.

Q3. How do you distinguish between a class object and an instance object?

Ans.

Class object is a real object which is the blue print and built from a class.

Instance object is a virtual copy of the object

You can distinguish them by the address of the object. For instance object the address will be always same. But in case of class object the address of the object is different always.

Q4. What makesthe first argument in a class’s method function special?

Ans.

The first argument is a class method should be object itself. Self is the variable which indicates current class object.

It is not a keyword, only a naming convention.

Q5. What is the purpose of the **init** method?

Ans.

init is the constructor. It is a special type of method which gets executed when an object is created. The main purpose of constructor is to initialize the instance variables.

It takes the first argument as self. In python name of the constructor is always **init**.

In python constructor is optional, if we are not creating any constructor then default constructor will be given by PVM.

```
# Eg :

class A:
    def __init__(self):
        self.B = 10
a = A()
print(a.__dict__) #displays the properties of the class
```

```
{'B': 10}
```

```
# eg2

class Employee:
    def __init__(self, eno, esal):
      self.eno = eno
      self.esal = esal

e = Employee(101, 180000)

print(e.eno)

print(e.esal)
```

```
101
180000
```

Q6. What is the process for creating a class instance?

Ans.

In python to create an object we have only one process. First we have to create a class by giving the keyword and class name. Then we need to create and object by giving paranthesis after the name of the class. To access it we need to assign it to a variable.

```
# Eg :

class A:
  def function(self, ch):
    return ch

a = A()
print(a.function('S'))
```

```
S
```

Q7. What is the process for creating a class?

Ans.

In python to create a class we need to write class name followed by class keyword and the we need to provide colon. As per OOP concept the class name should start with upper case then following lower case.

```
# Eg:

class A:
  def function(self, ch):
    return ch
```

Q8. How would you define the superclasses of a class?

Ans.

To define a super class of a class we need to write the super class name in parantheses after child class name while creating the child class. By this process the child class will extend the parent class attributes.

To access the attributes of parent class from child class, you need to create an object of child class.

```
# Eg:

class Parent:
    def p(self):
        print('Parent method')


class Child(Parent):
    def c(self):
        print('Child Method')


c = Child()
print(c.c())
print(c.p())
```

```
Child Method
None
Parent method
None
```

Q9. What is the relationship between classes and modules?

Ans.

To define a class we use class keyword.

Module is the combination of classes, functions and other attributes

We can modify the attributes of the class by using inheritance. But we can simply use the codes written in a module by importing it.

Q10. How do you make instances and classes?

Ans.

To make classes we use class keyword.

To make instance of a class you need to define **init**() method inside class. Then create object by passing the arguments it's constructor takes then refer it to an instance.

```
# Eg :

class Instance:
  def __init__(self, name):
    self.name = name

i = Instance('Sai')
print(i.name)
```

```
Sai
```

Q11. Where and how should be class attributes created?

Ans.

Class attributes belong to class itself, they will be shared by all the instances. They are defined at the class body parts.

Q12. Where and how are instance attributes created?

Ans.

Instance attributes are defined inside constructor which is defined directly inside a class using parameter self.

Q13. What does the term "self" in a Python class mean?

Ans.

Self is a reference variable which indicates current class instance/object. And it is used to access variables which belong to the particular class, inside the class.

Q14. How does a Python class handle operator overloading?

Ans.

In python + operator works as overloaded operator. It performs addition when we use it with int class. It performs concatenation when we use it with string and list class. To handle operator overloading python has inbuilt magic functions. (eg: **add**(self, other),**sub**(self, other), **mul**(self, other), **truediv**(self, other),**floordiv**(self, other), **mod**(self, other), **pow**(self, other), **rshift**(self, other), **lshift**(self, other), **and**(self, other), **or**(self, other), **xor**(self, other))

Q15. When do you consider allowing operator overloading of your classes?

Ans.

Allowing operator overloading means using operators (+,-,*,<,>,&,|,^) for different classes to perform different operations. We will allow operator overloading for the classes when we are dealing with same type classes (int-int, str-str, list-list) to perform either addition or concatenation operation.

Q16. What is the most popular form of operator overloading?

Ans.

The most popular form of operator loading is +. Which is used for both addition and concatenation.

Q17. What are the two most important concepts to grasp in order to comprehend Python OOP code?

Ans.

The most important concepts to grasp Python OOP code are

1. Inheritance
2. Polymorphism

Inheritance :

- Acquiring the properties of one class to another class is the concept of inheritance.
- It is used for code reusability and extendability.

```
# Eg:

class Employee:

    def __init__(self, eno, ename, esal):

        self.eno = eno

        self.ename = ename

        self.esal = esal

    def display(self):

        print('Employee Number : ', self.eno)

        print('Employee Name : ', self.ename)

        print('Employee Salary : ', self.esal)


e = Employee(1118, 'Subhasish', 100000)

e.display()
```

```
Employee Number :  1118
Employee Name :  Subhasish
Employee Salary :  100000
```

Polymorphism :

- One name multiple form, is the concept of polymorphism.

18.What is the concept of an abstract superclass?

Ans.

i. Abstract means partially implemented.

ii. A class is called as abstract class if it contains at-least one abstract method. An abstract method is a method which is only declared but not implemented. It's next level child class is responsible for implementation.

iii. Abstract class may not be instantiated.

```
# Eg :

class ABC:
    def m(self):
        pass

class XYZ(ABC):
    def m(self,a,b):
        return a+b
```

19. What happens when a class statement's top level contains a basic assignment statement?

Ans.

If the class statement's top level contains a basic assignment stament then it can be usable in further codes if we are using the same variable without re-assigning any other value. In case we are re-assigning any value PVM will consider the recent assigned value.

20. Why does a class need to manually call a superclass's **init** method?

Ans.

When we will create an object of the child class it will only execute the constructor present in particular class. If a class extends to a super class and we want to execute super class constructor then we need to call super class constructor manually from child class constructor with respective arguments.

```
# Eg :

class Parent:
    def __init__(self,B,C):
        print(B, C)
        self.B = B
        self.C = C

class Child(Parent):
    def __init__(self, B, C):
        super().__init__(B,C)



c = Child(10,20)
```

```
10 20
```

21. How can you augment, instead of completely replacing, an inherited method?

Ans.

We can augment a method instead of completely replacing it by performing different operations.

```
# Eg :

class ABC:

    def m(self):

        global a

        a = 10

class XYZ(ABC):

    def m(self):

        b = 20

        c = 30

        return b+c


x = XYZ()

print(x.m())
```

```
50
```

22. How is the local scope of a class different from that of a function?

Ans.

i. Local variables have the scope with in the function, method. It will get destroyed by destructor once the execution completes. If we will try to access them out side of that block we will get error.

ii. If we are creating a function then we can call it within the module, as it is provides functionality all across the module.

To get such content

Follow: https://github.com/saisubhasish