Object Oriented Programming in Python

- object -oriented Programming lets you structure your code using classes and objects
-

class Server:

```
def __init__(self,name,ip):
self.name=name
self.ip=ip
def status(self):
print(f"{self.name} at {self.ip} is running")

web_server=Server("WebServer","198.162.1.10")
web_server.status()
```

1. class Server=> blueprint of server
2. _ _ init _ _ =>Runs when your create a new Server
3. self.name , self.ip => These are the server's details (Attributes)
4. status() => a function (Method) will print the name and ip address
5. we_server=>Server(...) Creating an Actual Server (Object)
6. web_server.status=>calling method of the class

TASK:1 Write a program to call different methods of a class in Python-using Class and Object (5 Min)
Hint: create a class that stores information about a server and has a method to check if it need an update or not

What is inheritance?

- inheritance means one class can use the features of another class like a child and parent relationship, child can inherits traits from a parent

```
class Animal:
def __init__(self,name):
```

```
self.name=name
def speak(self):
print(f"{self.name} makes a sound.")
class Dog(Animal):
def speak(self):
print(f"{self.name} says Woof!")
class Cat(Animal)
def speak(self):
print(f"{self.name} says Meow!")
dog1= Dog("Buddy")
cat1=Cat("Mimi")


dog1.speak()
cat1.speak()



class Animal:
def __init__(self,name):
self.name=name
def speak(self):
print(f"{self.name} makes a sound.")
class Dog(Animal):
def speak(self):
print(f"{self.name} says Woof!")
class Cat(Animal):
def speak(self):
print(f"{self.name} says Meow!")
dog1= Dog("Buddy")
cat1=Cat("Mimi")


dog1.speak()
cat1.speak()
```

What is Method Overriding?

when child overrides the method of a parent class then it is known as Method overridding

What is abstraction ?

Abstraction means showing only the essential features , hiding the complex details.
Like while pressing a start button of a car . you don't know how the mechanish perform , but you only know about how to start

```
from abc import ABC,abstractmethod

class Car(ABC):
@abstractmethod
def start_engine(self):
pass
class Tesla(Car):
def start_engine(self):
print("Starting Tesla engine Silently....")
class Tata(Car):
def start_engine(self):
print("Starting TATA engine Silently....")
my_car1=Tesla()
my_car1.start_engine()


my_car2=Tata()
my_car2.start_engine()
```

- car(ABC)=> Abstract class not used directly
- @abstractionmethod=> this method must be written in the child class only
- Tesla(Car)=> child class that implements the real start_engine() logic
- Tata(Car)=> child class that implements the real start_engine() logic

TASK:2 write an Program to initiate Transaction using Abstraction method using different Payment Option (5 minutes)

```
from abc import ABC, abstractmethod

##### Step 1: Abstract class
class PaymentMethod(ABC):
@abstractmethod
def pay(self, amount):
pass

##### Step 2: Child classes with specific implementations
```

```
class UPI(PaymentMethod):
def pay(self, amount):
print(f"Paying ₹{amount} using UPI.")

class CreditCard(PaymentMethod):
def pay(self, amount):
print(f"Paying ₹{amount} using Credit Card.")

class PayPal(PaymentMethod):
def pay(self, amount):
print(f"Paying ₹{amount} using PayPal.")

##### Step 3: Using the classes
payment1 = UPI()
payment2 = CreditCard()
payment3 = PayPal()

payment1.pay(200)
payment2.pay(1500)
payment3.pay(900)
```

What is Encapsulation ?

- It Means Hiding Internal Data
- Protecting Data from direct access
- only allowing access via methods

```
class BankAccount:
def __init__(self,owner,balance):
self.owner=owner
self.__balance=balance # private Variable
def deposit(self,amount):
self.__balance +=amount
def withdraw(self, amount):
if amount <=self.__balance:
self.__balance -=amount
else:
print("Insufficient Balance!")
def get_balance(self):
```

```
return self.__balance

#using the class
acc=BankAccount("Nikunj Soni",1000)
acc.deposit(200)
acc.withdraw(100)

print("Balanace",acc.get_balance())
```

Different Libraries in Python

●OS module: This module allows interaction with the operating system, enabling tasks such as file and directory manipulation.

●subprocess: Along with the os module, subprocess is used for executing shell commands and scripts.

●boto3: This is the AWS SDK for Python, which allows you to manage AWS resources. It enables DevOps engineers to automate the management of AWS services, such as EC2, S3, and Lambda, directly from Python scripts.

●Paramiko: Paramiko is an SSH library that facilitates secure communication with remote servers. It enables you to execute commands, transfer files, and manage remote servers securely via SSH, making it crucial for automating tasks across a network of machines.
●Ansible: Ansible is a powerful automation tool that is particularly useful for system management duties. It is coded purely in Python.

1.

```
import os
import shutil

path="D:\\" if os.name =="nt" else "/mnt/d"


total,used,free= shutil.disk_usage(path)
def to_gb(bytes_value):
return bytes_value // (1024**3)


# printing the value
```

```
print(f"OS Name:${os.name}")
print(f"Checking disk usage for:{path}")
print("Total:",to_gb(total),"GB")
print("Used:",to_gb(used),"GB")
print("Free:",to_gb(free),"GB")
```