Day 26 (06/09/23): (OOPS Concept:)

OOPS (Object Oriented Programming
It is a real-world entity programming.
It is the structure of the programming.
Reusable, Maintenance.

Concepts:
1) Class & objects
2) Inheritance
3) Polymorphism
4) Encapsulation
5) Data abstraction

Class & Objects:
A Class is a collection of objects.
An Object is a behaviour / Blueprint of the Class.

Syntax:
Class class_name:
    Print statement
Obj=class_name()

Code:

```
class besant:
    def __init__(self,firstname,lastname,age):
        #Instance attribute
        self.firstname=firstname
        self.lastname=lastname
        self.age=age
    def python_students(self):
        print(f"my firstname is {self.firstname}")
        print(f"My lastname is {self.lastname}")
        print(f"My age is {self.age}")
    def age_checking(self):
        if self.age>=18:
            print("You are allowed to learn")
        else:
            print("You are not allowed")
firstname=input("Enter your firstname: ")
lastname=input("Enter your Lastname: ")
age=int(input("Enter your name: "))
obj1=besant(firstname,lastname,age)
obj1.python_students()
obj1.age_checking()
```

Output:
Enter your firstname: Manoj
Enter your Lastname: Kumar
Enter your name: 19
my firstname is Manoj
My lastname is Kumar
My age is 19
You are allowed to learn

What is constructor?
a constructor (abbreviation: ctor) is a special type of function called to create an object.

__init__ is a constructor to initialize the objects.
self is a keyword to identify the class objects.

Attribute:
Assigning a value to a variable.

Types:
1) Class Attribute
2) Instance Attribute.

Functions are called as methods in OOPS & JAVA.

Inheritance:
One class is derived from another class. Separate data access by unique (User & admin).

Types:
Single Inheritance
Multi-level Inheritance
Hierarchical inheritance
Multiple Inheritance

Code:

```
class student:
    def
__init__(self,student_name,student_department,student_attendance_percent):
        student_name=input("Enter the student name: ")
        student_department=input("Enter student department: ")
        student_attendance_percent=int(input("Enter the Student Attendance
percentage: "))
        self.student_name=student_name
        self.student_department=student_department
        self.student_attendance_percent=student_attendance_percent
    def student_attendance(self):
        print(f"{self.student_name} of {self.student_department} has an
attendance percentage of {self.student_attendance_percent}")
class teacher(student):
    def
__init__(self,student_name,student_department,student_attendance_percent,teacher_
name,teacher_department,teacher_attendance_percent):
        teacher_name=input("Enter the teacher name: ")
        teacher_department=input("Enter teacher department: ")
        teacher_attendance_percent=int(input("Enter the teacher Attendance
percentage: "))
        self.teacher_name=teacher_name
        self.teacher_department=teacher_department
        self.teacher_attendance_percent=teacher_attendance_percent
        #invoking the class (to access the contents of another class)
        student.__init__(self,student_name,student_department,student_attendance_
percent)
    def teacher_attendance(self):
        print(f"{self.teacher_name}-{self.teacher_department}-
>{self.teacher_attendance_percent}")
teacher=teacher("raj kumar","ECE","75","malar","MCA","98")
student=student("arun","ECE","95")
teacher.teacher_attendance()
teacher.student_attendance()
student.student_attendance()
#student.teacher_attendance() #error
```

Encapsulation: Wrapping the data (private Attribute). Banking concepts. Access only by corresponding user. Private attributes can be created by 'self.__xxxxx' (using double underscore)

Code:

```
print("Encapsulation:") #wrapping the data (private attribute)
print(student.student_name) #add __ in self.xxxx to make it capsulated and use as
a secured python code
print("")
class student:
    def
__init__(self,student_name,student_department,student_attendance_percent):
        student_name=input("Enter the student name: ")
        student_department=input("Enter student department: ")
        student_attendance_percent=int(input("Enter the Student Attendance
percentage: "))
        self.__student_name=student_name
        self.__student_department=student_department
        self.__student_attendance_percent=student_attendance_percent
    def student_attendance(self):
        print(f"{self.__student_name} of {self.__student_department} has an
attendance percentage of {self.__student_attendance_percent}")
print(student.student_name) #error
```

Polymorphism: A simple concept used for multiple purpose. Same Method Name using different classes.

Code:

```
class india:
    def fav_game(self):
        print('indian game')

class USA:
    def fav_game(self):
        print("USA's game")

class brazil:
    def fav_game(self):
        print("brazil same")

obj1=india()
obj2=USA()
obj3=brazil()

obj1.fav_game()
obj2.fav_game()
obj3.fav_game()
```

Data Abstraction:
To set a rule of methods.
Rules applied and followed, otherwise error.

What is (abc) abstract class?
Abstract class is common application programming.

What is decorator? Ex: @abstractmethod

Code:

```
from abc import ABC,abstractmethod #abc--> abstract class (common application )
class IOB_MAIN_BRANCH(ABC):
    @abstractmethod
    def rule1(self):
        pass #nothing inside

class iob_subbranch(IOB_MAIN_BRANCH):
    def rule1(self):
        print ("1 is rule 1 is executed in sub branch.")

class iob_subbranch2(IOB_MAIN_BRANCH):
    def rule2(self):
        print("2 is rule 1 is executed in sub branch")

obj=iob_subbranch2()
obj.rule2()
```

Output:

```
Traceback (most recent call last):
  File "c:\Vaishnav\Courses\Python\Codes\Beasent\Exercises\Day#26.py", line 155, in <module>
    obj=iob_subbranch2()
TypeError: Can't instantiate abstract class iob_subbranch2 with abstract method rule1
```