TEST QUESTIONS – 8 JAN 2023

1. What is Object-Oriented Programming, and how does it differ from Procedural programming?

- **Procedural Programming** can be defined as a programming model which is derived from structured programming, based upon the **concept of calling procedure**. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program's execution, any given procedure might be called at any point, including by other procedures or itself.

Eg: **C, FORTRAN, COBOL**

- **Object-oriented programming** can be defined as a programming model which is based upon the **concept of objects**. Objects contain data in the form of attributes and code in the form of methods. In object-oriented programming, computer programs are designed using the concept of objects that interact with the real world. Object-oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.
- Class is a blueprint that defines some properties and behaviours. An object is an instance of a class that has those properties and behaviours attached. A class is not allocated memory when it is defined. An object is allocated memory when it is created. Class is a logical entity whereas objects are physical entities.

Eg: **Java, C++, C#, Python**

A procedural language is focused on the functions and the data. An object-oriented language is focused on capturing the functions and the data into objects, and then performing essentially the same code.

2.Explain the principles of OOP and how they are implemented in Python.

- **Abstraction** is the concept of hiding all the implementation of your class away from anything outside of the class.

```
# Abstraction
class Dog:

    def __init__(self, name):
        self.name = name
        print(self.name + " was adopted.")

    def bark(self):
        print("woof!")

spot = Dog("Blacky") # Blacky was adopted.
spot.bark() # woof!
```

- **Inheritance** is the mechanism for creating a child class that can inherit behaviour and properties from a parent(derived) class.

```python
# Inheritance
class Animal:

    def __init__(self, name):
        self.name = name
        print(self.name + " was adopted.")

    def run(self):
        print("Running!")

class Dog(Animal):

    def bark(self):
        print("woof!")

spot = Dog("Blacky") # Blacky was adopted.
spot.run() # running!
```

- **Encapsulation** is the method of keeping all the state, variables, and methods private unless declared to be public.

```python
# Encapsulation
class Employee:

    def __init__(self, name, id, salary, project):
        # data members
        self.name = name
        self.id = id
        self.salary = salary
        self.project = project

    def salary_details(self):
        print("Name: ", self.name)
        print('Salary:', self.salary,'LPA')

    def projects(self):
        print(self.name, 'is working on', self.project,'Project')

# creating object of a class
emp = Employee('James', 102, 100000, 'Customer Segmentation')

# calling public method of the class
emp.salary_details()
                    # Name:  James
                    #Salary: 100000 LPA
emp.projects()  # James is working on Customer Segmentation Project
```

- **Polymorphism** is a way of interfacing with objects and receiving different forms or results.

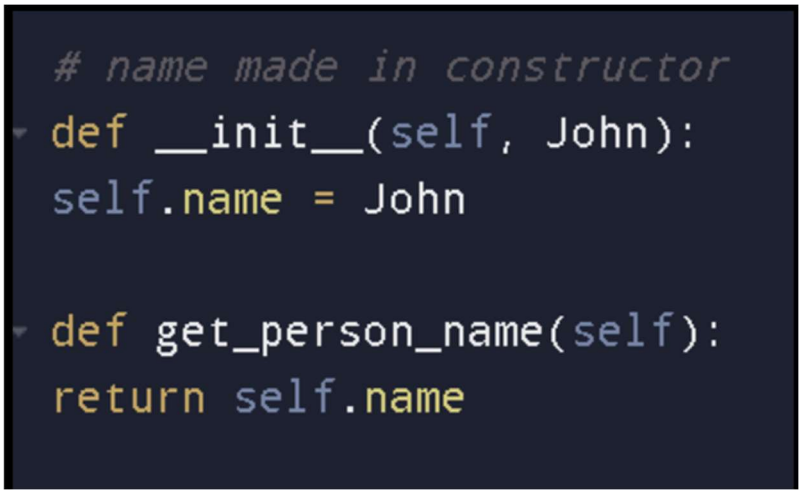
```python
# Polymorphism
from math import pi
class Shape:
    def __init__(self, name):
        self.name = name
    def area(self):
        pass
    def fact(self):
        return "Two-dimensional shape with no edges and no sides"
    def __str__(self):
        return self.name

class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius
    def area(self):

        return round(pi*self.radius**2,3)

shape_circle = Circle(7)
print(shape_circle) # Circle
print('Area: ',shape_circle.area(),'sq.unit')
# Area:  153.938 sq.unit
print('Feature :',shape_circle.fact())
# Feature : Two-dimensional shape with no edges and no sides
```

3. Describe the concepts of encapsulation, inheritance, and polymorphism in Python.

- **Encapsulation** is the process of bundling data members and methods inside a single class. Bundling similar data elements inside a class also helps in data hiding. Encapsulation also ensures that objects of a class can function independently.
- **Polymorphism** refers to a function having the same name but being used in different ways and different scenarios. It basically creates a structure that can use many forms of objects. It can be implemented in different ways, including operator overloading, built-in functions, classes, and inheritance.
- **Inheritance** is a mechanism through which we create a class or object based on another class or object. In other words, the new objects will have all the features or attributes of the class or object on which they are based. We refer to the created class as the derived or child class, and the class from which it inherits is the base or parent class. Inheritance is one of the most important concept of OOP. It provides code reusability, readability and transition of properties which helps in optimized and efficient code building.

4.What is the purpose of the self keyword in Python class methods?

The self is used to represent the instance of the class. With this keyword, you can access the attributes and methods of the class in python. It binds the attributes with the given arguments. The reason why we use self is that Python does not use the '@' syntax to refer to instance attributes.



self is also used to refer to a variable field within the class. In the above example, self refers to the name variable of the entire Person class. Here, if we have a variable within a method, self will not work. That variable is simply existent only while that method is running and hence, is local to that method. For defining global fields or the variables of the complete class, we need to define them outside the class methods.

5.What is the difference between class and instance variables in Python?

- Python class variables are declared within a class and their values are the same across all instances of a class. Python instance variables can have different values across multiple instances of a class.
- Class variables share the same value among all instances of the class. The value of instance variables can differ across each instance of a class.
- Class variables can only be assigned when a class has been defined. Instance variables, on the other hand, can be assigned or changed at any time.
- Both class variables and instance variables store a value in a program, just like any other Python variable.

6.Discuss the concept of abstract classes and how they are implemented in Python.

Abstraction refers to the process of hiding the details and showcasing essential information to the user. In Python, abstraction is achieved by using abstract classes and methods. An **abstract class** is a blueprint for creating other classes. Any class is referred to as an abstract class if it contains one or

more abstract methods. An **abstract method** is a function that has a declaration but does not have any implementation.

These abstract classes are used when we want to provide a common interface for different implementations of a component.

Consider an example where we create an abstract class Animal. We derive four classes Snake, Dog and Cat from the Animal class that implement the methods defined in this class. Here the Animal class is the parent abstract class and the child classes are Cat, Dog and Snake. We won't be able to access the methods of the Animal class by simply creating an object, we will have to create the objects of the derived classes: Cat and Snake to access the methods.

```python
# Code implementation for Abstraction
from abc import ABC,abstractmethod

class Animal(ABC):
    #concrete method
    def sleep(self):
        print("sleeping")

    def eat(ABC):
        print('eating')

    @abstractmethod
    def sound(self):
        pass

class Snake(Animal):
    def sound(self):
        print("Snake says, I can hiss")

class Dog(Animal):
    def sound(self):
        print("Dog says, I can bark")

class Cat(Animal):
    def sound(self):
        print("Cat says, I can meow")

c = Cat()
c.sleep() # sleeping
c.sound() # Cat says, I can meow
s = Snake()
s.sound() # Snake says, I can hiss
d=Dog()
d.sound() # Dog says, I can bark
d.eat()  # eating
```

7.What is a decorator in Python, and how can it be used in the context of OOP?

Decorators are functions (or classes) that provide enhanced functionality to the original function (or class) without the programmer having to modify their structure.

Suppose we want to add a method to our **Student** class that takes a student's score and total marks and then returns a percentage.

```python
# Code Implementation
def grade_decorator(f):
    def wrapper(score, total):
        percent = f(score, total)

        grades = {5: 'A',4: 'A',3: 'B', 2: 'C', 1: 'D'}

        return percent, grades[percent // 20]
    return wrapper


class Student:
    def __init__(self, name, score, total):
        self.name = name
        self.score = score
        self.total = total

    @grade_decorator
    def get_percent(score, total):
        return score / total * 100

    @classmethod
    def from_str(cls, str_arg):
        name, score, total = str_arg.split(',')
        return cls(name, int(score), int(total))

    def get_record(self):
        percent, grade = Student.get_percent(self.score, self.total)
        return f"Name: {self.name} | Percentage scored: {percent} % | Grade: {grade}"

    def __str__(self):
        return ("Name: " + str(self.name) + "; Score: " + str(self.score) +  "; Total : " + str(self.total))

student = Student("John", 20, 100)
print(student)  # Name: John; Score: 20; Total : 100
```

8.How does method overriding work in Python, and why is it useful?

Method overriding is a feature of object-oriented programming languages where the subclass or child class can provide the program with specific characteristics or a specific implementation process of data provided that are already defined in the parent class or superclass.

When the same returns, parameters, or name is input in the subclass as in the parent class, then the method of implementation in the subclass overrides the method as mentioned in the parent class. This is known as method overriding.

Its execution depends on the data that is used to invoke the method and not the reference data already provided in the parent class. If an object of the parent class is used to invoke the method of implementation that is specific to a program, then the version of the method as written in the parent class is invoked. On the other hand, if an object of the subclass is used to invoke the method, the execution will be according to the features mentioned in the subclass.

9.Explain the importance of the super () function in Python inheritance.

The super () function is useful in multiple inheritance scenarios where a class inherits from more than one parent class. The super () function we use in the child class returns a temporary created object of the superclass, that allow us to access all of its method present in the child class. In this case, the super () function helps ensure that each parent class's methods are called only once and in the correct order.

10.How does Python support multiple inheritance, and what challenges can arise from it?

In multiple inheritance a class inherits the features of its base classes.

Multiple inheritance isn't a way to get some functionality from some class as a part of the subclass as the entire public interface of the base class is always inherited. Multiple inheritance also makes the structure of the program more complex and it should therefore be used with caution.

In multiple inheritance, it's quite possible that inheriting classes could have methods of the same name (Diamond problem). The MRO dictates which of them gets called. These name clashes can cause problems and can make the code a little confusing.

11.What do you understand by Descriptive Statistics? Explain by Example.

Descriptive statistics are methods used to summarize and describe the main features of a dataset. It describes the important characteristics/ properties of the data using the measures the central tendency like mean/ median/mode and the measures of dispersion like range, standard deviation, variance etc. subsequently, data can be summarized and represented in an accurate way using charts, tables and graphs.

For example: We have marks of 1000 students and we may be interested in the overall performance of those students and the distribution as well as the spread of marks.

12. What do you understand by Inferential Statistics? Explain by Example

Inferential statistics makes inferences and predictions about a population based on a sample of data taken from the population in question using regression analysis, confidence ranges, and hypothesis testing.

For instance, you could use inferential statistics to assess whether there is a significant difference in the outcomes of patients who receive the drug compared to those who receive a placebo if you want to know if a new drug is effective.
