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

Object-Oriented Programming (OOP) and procedural programming are two different programming paradigms, each with its own approach to organizing and structuring code.

- Data Organization:
  - OOP organizes data and behavior together in the form of objects and classes.
  - Procedural programming separates data and procedures, focusing on functions or procedures that operate on the data.
- Code Reusability:
  - OOP promotes code reuse through concepts like inheritance.
  - Procedural programming may reuse functions, but the focus is on procedures rather than objects.
- Abstraction:
  - OOP provides a higher level of abstraction through encapsulation, inheritance, and polymorphism.
  - Procedural programming uses functions for abstraction but may not achieve the same level of abstraction as OOP.

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

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

Python is an object-oriented programming (OOP) language, and it incorporates the fundamental principles of OOP: encapsulation, inheritance, and polymorphism.

1. **Encapsulation** : Encapsulation is the bundling of data and the methods that operate on that data into a single unit, known as a class. It restricts direct access to some of an object's components and can prevent unintended interference.

   Implementation in Python.

```
class Car:
    def __init__(self, make, model):
        self.__make = make  # Private attribute
        self.__model = model  # Private attribute
```

```
    def get_make(self):
        return self.__make

    def get_model(self):
        return self.__model

    def display_info(self):
        print(f"Car: {self.__make} {self.__model}")

# Creating an instance of the Car class
my_car = Car("Toyota", "Camry")

# Accessing data through methods
print("Make:", my_car.get_make())
print("Model:", my_car.get_model())
```

2.**Inheritance** : Inheritance is a mechanism that allows a class (subclass or derived class) to inherit attributes and methods from another class (superclass or base class). It promotes code reuse and the creation of a hierarchy of classes.

Implementation in python:

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

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

```
# Creating instances of subclasses
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Accessing inherited attributes and methods
print(dog.name)  # Accessing the 'name' attribute from the Animal class
print(dog.speak())  # Accessing the 'speak' method from the Dog class

print(cat.name)
print(cat.speak())
```

3.**Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different types.

Implementation in python:

```
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        return "Sparrow flying"

class Penguin(Bird):
    def fly(self):
        return "Penguin can't fly"

# Using polymorphism with a common interface
def let_bird_fly(bird):
    return bird.fly()

# Creating instances of different subclasses
sparrow = Sparrow()
penguin = Penguin()

# Calling the same method on different objects
```

```
print(let_bird_fly(sparrow))  # Output: Sparrow flying
print(let_bird_fly(penguin))  # Output: Penguin can't fly
```

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

The 'self' keyword is used as a conventional name for the first parameter of instance methods in a class. It serves as a reference to the instance of the class itself. When we define a method within a class, and want that method to operate on the instance variables of that class, we must include 'self' as the first parameter in the method definition.

4. **How does method overriding work in Python, and why is it useful**

Method overriding in Python occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This allows the subclass to customize or extend the behavior of the inherited method. Method overriding is a crucial aspect of polymorphism, which is one of the key principles of object-oriented programming.

Inheritance:

- The subclass inherits a method from its superclass.

Method Redefinition:

- The subclass provides a new implementation for the inherited method with the same signature (name and parameters).

Dynamic Dispatch:

- When an object of the subclass calls the overridden method, Python dynamically determines which version of the method to execute based on the actual type of the object at runtime.

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

- Class variables are shared among all instances of a class, while instance variables are specific to each instance.
- Class variables are defined outside of methods using the class name, and instance variables are defined within methods using the self keyword.

- Changes to a class variable affect all instances, while changes to an instance variable only affect the specific instance.
- Class variables are often used for constants or shared data, while instance variables represent the state of an individual object.

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

An abstract class in Python is a class that cannot be instantiated on its own and typically serves as a blueprint for other classes. It may contain abstract methods, which are methods declared but not implemented in the abstract class. Subclasses are then required to provide concrete implementations for these abstract methods. Abstract classes provide a way to define a common interface for a group of related classes while allowing individual classes to provide their own specific implementations

Implementation in Python

Python provides a module called abc(Abstract Base Classes) that allows you to define abstract classes and abstract methods. The ABC (Abstract Base Class) meta-class is used in conjunction with the @abstractmethod decorator to declare abstract methods.

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

The `super()` function in Python plays a crucial role in supporting and enhancing the inheritance mechanism.The `super()` function promotes cooperative behavior in inheritance. It allows subclasses to call methods of their superclass, facilitating code reuse and extension. Subclasses can build upon the functionality provided by the superclass without duplicating code.

When defining the `__init__` method in a subclass, using `super().__init__()` ensures that the constructor of the superclass is called before the subclass's constructor. This supports consistent initialization of attributes throughout the inheritance hierarchy.

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

Python supports multiple inheritance, allowing a class to inherit attributes and methods from more than one parent class. This means that a class can have multiple base classes, and it inherits features from all of them. However, while multiple inheritance provides flexibility, it can also introduce challenges and complexities

Implementation in Python

```
class DerivedClass(BaseClass1, BaseClass2, ...):
    # Class definition
```

Challenges of Multiple inheritance

1.Diamond Inheritance (Ambiguity)
2.Complexity and Readability
3.Dependency on Class Hierarchy
4.Ordering of Base Classes

multiple inheritance can be a powerful tool when used judiciously. It allows for code reuse, flexibility, and the creation of rich class hierarchies. To mitigate challenges, developers should carefully design class hierarchies, consider alternative design patterns, and document the relationships between classes.

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

In Python, a decorator is a special type of function that can be used to modify or extend the behavior of another function or method. Decorators are often used for tasks such as logging, authorization, memoization, or other cross-cutting concerns without modifying the original function's code. They provide a clean and reusable way to wrap functions and add functionality to them.

decorators can be applied to methods of a class to enhance or modify their behavior. They are a powerful tool for extending the functionality of methods without altering the original class code.

Use of decorators in oops

```
def log_method_call(func):
```

```
    def wrapper(self, *args, **kwargs):
        print(f"Calling {func.__name__} with arguments {args} and keyword
arguments {kwargs}")
        result = func(self, *args, **kwargs)
        print(f"{func.__name__} returned: {result}")
        return result
    return wrapper

class Calculator:
    def __init__(self):
        pass

    @log_method_call
    def add(self, x, y):
        return x + y

    @log_method_call
    def subtract(self, x, y):
        return x - y

# Creating an instance of Calculator
calculator = Calculator()

# Using decorated methods
result1 = calculator.add(3, 5)
result2 = calculator.subtract(10, 4)

# Output:
# Calling add with arguments (3, 5) and keyword arguments {}
# add returned: 8
# Calling subtract with arguments (10, 4) and keyword arguments {}
# subtract returned: 6
```

10. **What do you understand by Descriptive Statistics? Explain by Example.**
    Descriptive statistics is a branch of statistics that involves summarizing and presenting data in a meaningful way. It aims to describe and summarize the main

features of a dataset, providing insights into its central tendencies, variability, and distribution. Descriptive statistics includes measures such as mean, median, mode, range, variance, and standard deviation.

Suppose you have the exam scores (out of 100) of a group of students in a class: 75, 82, 90, 88, 92, 78, 85, 95, 80, 88

- Mean (Average):
  - Add up all the scores and divide by the number of scores.
  - $\text{Mean}=(75+82+90+88+92+78+85+95+80+88)/10=85.3$
- Median:
  - Arrange the scores in ascending order and find the middle value.

  - $\text{Median}=85$ (since there are 10 scores, and 85 is the 5th score when ordered).
- Mode:
  - Identify the score that appears most frequently.
  - $\text{Mode}=88$ (it appears twice, more than any other score).
- Range: The difference between the highest and lowest scores.
- $\text{Range}=95-75=20$

11. **What do you understand by Inferential Statistics? Explain by Example**

Inferential statistics is a branch of statistics that involves making inferences or predictions about a population based on a sample of data drawn from that population. The goal is to generalize findings from a sample to the larger population and to draw conclusions about parameters, relationships, or trends within the population. Inferential statistics uses probability theory and hypothesis testing to make these inferences.