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

Object-Oriented Programming (OOP) and procedural programming are two distinct paradigms used in software development.

Object-Oriented Programming (OOP):

OOP revolves around the concept of objects. Everything in OOP is modelled as an object, which combines data (attributes) and behaviours (methods/functions) that operate on that data.

OOP is based on four main principles: encapsulation, inheritance, polymorphism, and abstraction.

Encapsulation: Bundling data and methods that work on that data within a single unit (object), preventing direct access and manipulation of data from outside the object.

Inheritance: The ability of a class (an object's blueprint) to inherit properties and behaviours from another class, allowing for code reuse and hierarchical classification.

Polymorphism: The ability of different objects to be treated as instances of the same class but can respond to method calls differently based on their specific implementation.

Abstraction: Hiding complex implementation details and showing only the necessary functionalities or characteristics of an object.

Example Languages: Java, C++, Python, Ruby are some languages that support OOP extensively.

Procedural Programming:

Procedural programming revolves around procedures or routines, where a program is divided into functions or procedures that perform specific tasks.It uses a linear top-down approach, focusing on the sequence of steps to solve a problem.

Data Handling: In procedural programming, data is often separate from the functions operating on it, and the flow of the program is guided by function calls.

Example Languages: C, Fortran, and Pascal are historically procedural languages.

2.**Explain the principles of OOP and how they are implemented in Python.**
Describe the concepts of encapsulation, inheritance, and polymorphism in Python.

In Python, Object-Oriented Programming (OOP) principles are implemented through its support for classes and objects. Here's how the key OOP principles—encapsulation, inheritance, and polymorphism—are realised in Python:

1. Encapsulation: Encapsulation involves bundling data (attributes) and methods that operate on that data within a single unit (object). It prevents direct access to data from outside the object and ensures that data can only be accessed through controlled interfaces (methods). Python uses naming conventions and access specifiers to achieve encapsulation, although it doesn't enforce strict access control like some other languages.

By convention, attributes that should be treated as private are prefixed with an underscore (_). For example, _variable_name.Python doesn't have explicit access specifiers like public or private, relying more on developer conventions and a philosophy of "we are all consenting adults here," meaning it trusts developers to use encapsulation correctly.

2. Inheritance:Inheritance allows a new class (subclass or derived class) to inherit properties and behaviors (methods) from an existing class (base class or parent class), promoting code reuse and establishing a hierarchical relationship between classes.Python supports single and multiple inheritance, meaning a class can inherit from one or more classes.In Python, inheritance is achieved by including the parent class in parentheses after the class name. For example:

```
class ParentClass:
        # Parent class methods and attributes

class ChildClass(ParentClass):
        # Child class inherits from ParentClass
        # Child class may have additional methods or attributes
```

3. Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling them to be manipulated using the same interface. This facilitates code flexibility and reusability.Python supports polymorphism through method overriding and duck typing.

Method Overriding: Subclasses can provide a specific implementation of a method that is already defined in its superclass. When the method is called on an object of the subclass, the subclass's method is executed instead of the superclass's method.

Duck Typing: Python focuses on an object's behaviour rather than its type. If an object implements a method, it can be used wherever that method is expected, irrespective of its actual class.

Python's flexibility and dynamic nature enable developers to implement these OOP principles effectively, providing a powerful yet straightforward way to structure code using classes and objects.

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

In Python, the self keyword refers to the instance of the class itself. It's a convention used as the first parameter in instance methods within a class. When you define a method in a class and then create an object of that class to call that method, Python automatically passes the instance as the first argument to the method.

The purpose of self is to enable access to the instance's attributes and methods within the class. By using self, you can manipulate the object's state (attributes) and behaviour (methods) within the class.

Consider this example:

```
class MyClass:
        def __init__(self, x):
                self.x = x # Assigning the value of x to the instance variable self.x

         def print_x(self):
                print(self.x) # Accessing the instance variable self.x within the method

# Creating an object of MyClass
obj = MyClass(5)

# Calling the print_x method of obj
obj.print_x() # This internally translates to MyClass.print_x(obj)
```

In the print_x method, self allows access to the x attribute of the instance (self.x). When you call obj.print_x(), Python implicitly passes obj as the self argument to the print_x method. This way, you can work with the specific attributes and methods of the instance obj within the class methods.

Using self is a Python convention, but you can actually name this first parameter anything you like. However, it's strongly recommended to stick with self for better code readability and to follow the convention followed by most Python developers.

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

Method overriding in Python occurs when a subclass defines a method that is already defined in its superclass. When an object of the subclass calls the overridden method, the subclass's version of the method is executed instead of the superclass's method.

Here's an example:

```
class Animal:
    def sound(self):
        print("Some generic sound")

class Dog(Animal):
    def sound(self):
        print("Bark!")

class Cat(Animal):
    def sound(self):
        print("Meow!")
```

In this example:Animal is the superclass with a method sound.Dog and Cat are subclasses of Animal that override the sound method with their specific implementations.

When you create objects of Dog or Cat and call the sound method, Python invokes the overridden method from the respective subclass:

```
dog = Dog()
dog.sound() # Output: "Bark!"

cat = Cat()
cat.sound() # Output: "Meow!"
```

Why method overriding is useful:

Polymorphism: Method overriding enables polymorphism, allowing different objects to be treated uniformly based on a common superclass. This promotes code flexibility and reusability.

Customization: Subclasses can provide their own specialised implementation of a method inherited from the superclass, allowing customization based on specific requirements.

Refinement: It allows refining or extending the behaviour of a superclass method without modifying the superclass itself, maintaining the original behaviour for other subclasses or instances.

Method overriding is a fundamental concept in OOP that facilitates code organisation, extensibility, and the implementation of different behaviours based on class hierarchies, promoting a more adaptable and maintainable codebase.

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

In Python, class variables and instance variables are two types of variables associated with classes and instances (objects), respectively.

1. Class Variables:Class variables are variables that are shared among all instances of a class. They are defined within the class but outside of any class method.Class variables are accessible to all instances of the class and the class itself.They are used to store data that is common to all instances of the class.Defined directly within the class using the class name.

Example:

```
class MyClass:
        class_var = 10 # This is a class variable

        def __init__(self, inst_var):
                self.inst_var = inst_var # This is an instance variable

                # Accessing class variable
                print(MyClass.class_var) # Output: 10

# Class variable can be accessed via instances as well
obj1 = MyClass(20)
print(obj1.class_var) # Output: 10
```

2. Instance Variables: Instance variables are unique to each instance of a class. They are defined within the class methods, typically within the constructor (__init__ method) using self.Instance variables are accessible only to the specific instance/object they belong to. They store data that is specific to each instance/object of the class.Defined within the instance methods using self.

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

Abstract classes in Python are classes that cannot be instantiated on their own. They are designed to serve as blueprints for other classes and typically contain abstract methods, which are methods without any implementation. These abstract methods must be implemented by the subclasses that inherit from the abstract class.
Python provides a module called abc (Abstract Base Classes) that allows you to create abstract classes and abstract methods.

```
from abc import ABC, abstractmethod

class Shape(ABC):  # Define an abstract class using ABC as a base class
    @abstractmethod
    def calculate_area(self):
        pass  # Abstract method without implementation

    @abstractmethod
    def calculate_perimeter(self):
        pass  # Another abstract method

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        return self.length * self.width

    def calculate_perimeter(self):
        return 2 * (self.length + self.width)

# Trying to instantiate the abstract class (Shape) will result in an error
# shape = Shape()  # This will raise TypeError: Can't instantiate abstract class Shape with abstract methods calculate_area, calculate_perimeter

# Creating an object of the concrete subclass (Rectangle)
rectangle = Rectangle(5, 4)

# Calling methods implemented in the subclass
print("Area:", rectangle.calculate_area())  # Output: 20
print("Perimeter:", rectangle.calculate_perimeter())  # Output: 18
```

Abstract methods: Defined using the @abstractmethod decorator in the abstract class. Subclasses must provide implementations for these methods.

Instantiation: You cannot create instances of abstract classes directly. They serve as templates for subclasses to inherit and implement.

Inheritance: Subclasses must implement all abstract methods from the abstract class; otherwise, they will also be considered abstract and cannot be instantiated.

Enforcement: Abstract classes help enforce a structure and contract that subclasses need to fulfill. They define a common interface that subclasses must adhere to.

Abstract classes in Python provide a way to create a blueprint with certain requirements that subclasses must fulfill. They promote code consistency, help in defining a common interface, and ensure that specific methods are implemented in subclasses, enhancing the overall structure and maintainability of the codebase.

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

The super() function in Python is essential in inheritance as it allows you to access methods and properties from a parent or superclass within a subclass. It facilitates calling and accessing the superclass's methods, enabling a more organised and controlled way of working with inheritance hierarchies. Here's why super() is important:

Accessing Superclass Methods: In a subclass, super() provides a way to access and invoke methods defined in the superclass. This is particularly useful when the subclass overrides a method from the superclass but still wants to call the superclass's method.

```
class Parent:
    def method(self):
        print("Parent method")

class Child(Parent):
    def method(self):
        super().method()  # Calls the method from the parent class
        print("Child method")

# Creating an instance of Child and invoking its method
child = Child()
child.method()
# Output:
# Parent method
# Child method
```

Maintaining Code Flexibility: Using super() allows for more flexibility in the inheritance chain. Even if the hierarchy changes or additional classes are inserted between parent and child classes, super() adapts to refer to the immediate superclass.

Support for Multiple Inheritance: When dealing with multiple inheritance, super() ensures the methods are invoked in the Method Resolution Order (MRO), a defined sequence for method lookup. This helps in resolving diamond inheritance issues by following a consistent resolution path.

Dynamic Method Resolution: super() adapts to the instance's actual class dynamically, allowing changes in method resolution based on the instance type.

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 feature enables a subclass to inherit from multiple superclasses. However, multiple inheritance can introduce complexities and challenges:

1. Method Resolution Order (MRO):

Python uses the C3 Linearization algorithm to determine the Method Resolution Order, which defines the order in which methods are resolved in multiple inheritance.The MRO ensures a consistent and predictable order for method lookup in complex inheritance hierarchies.

2. Diamond Inheritance (The Diamond Problem):

When a subclass inherits from two classes that have a common ancestor, ambiguity can arise if both parent classes have the same method or attribute.This scenario creates confusion about which method or attribute to prioritize in the inheritance chain.

3. Complexity and Readability:

Multiple inheritance can make the codebase more intricate and challenging to understand, especially when dealing with a large number of parent classes and complex hierarchies.Code readability may suffer if the relationships among multiple parent classes and their interactions are not well-documented or clear.

4. Accidental Name Clashes:

Inadvertent name clashes can occur if different parent classes have methods or attributes with the same names. This can lead to unintended method overriding or difficulties in accessing the desired method or attribute.

5. Maintenance and Debugging:

Debugging and maintaining code with multiple inheritance can be challenging due to the intricate relationships among different classes and potential conflicts arising from method resolution.

To mitigate the challenges of multiple inheritance in Python, it's essential to:

- Use it judiciously: Prefer simpler and more straightforward design patterns when possible.
- Understand and document the Method Resolution Order to anticipate how methods will be resolved.
- Resolve ambiguity or conflicts by explicitly specifying method overrides or using aliases to disambiguate conflicting names.
- Favor composition (using objects of other classes within a class) over complex multiple inheritance structures when appropriate.

While multiple inheritance can be a powerful tool in Python, it requires careful consideration and design to avoid complications and maintain a clean, understandable, and manageable codebase.

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

In Python, a decorator is a function that modifies or enhances another function or method. Decorators allow you to add functionality to existing functions or methods without modifying their actual code. They are denoted by the @decorator_name syntax and are placed above the function or method definition.

In the context of Object-Oriented Programming (OOP), decorators can be used in various ways:

1. Method Decorators:

Decorators can modify the behavior of methods within a class. They can add functionality before or after method execution or perform checks before allowing the method to run.

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before method execution")
        result = func(*args, **kwargs)
        print("After method execution")
        return result
    return wrapper


class MyClass:
    @my_decorator
    def my_method(self):
        print("Inside method")


obj = MyClass()
```

```
obj.my_method()

# Output:

# Before method execution

# Inside method

# After method execution
```

2. Class Decorators:

Decorators can also modify the behavior of an entire class. They can wrap the class definition and modify its attributes or methods.

```
def add_custom_attribute(cls):

    cls.custom_attribute = 100

    return cls


@add_custom_attribute

class MyClass:

    pass


print(MyClass.custom_attribute)  # Output: 100
```

3. Instance Methods and Class Methods Decorators:

Decorators can be used with instance methods (self) and class methods (@classmethod) in a similar way to regular methods.

```
def instance_method_decorator(func):

    def wrapper(self, *args, **kwargs):
```

```
        print("Before instance method execution")
        result = func(self, *args, **kwargs)
        print("After instance method execution")
        return result
    return wrapper

def class_method_decorator(cls, method_name):
    method = getattr(cls, method_name)
    def wrapper(*args, **kwargs):
        print("Before class method execution")
        result = method(*args, **kwargs)
        print("After class method execution")
        return result
    setattr(cls, method_name, wrapper)
    return cls

class MyClass:
    @instance_method_decorator
    def my_instance_method(self):
        print("Inside instance method")

    @classmethod
    @class_method_decorator
    def my_class_method(cls):
```

```
        print("Inside class method")


obj = MyClass()

obj.my_instance_method()

MyClass.my_class_method()

# Output:

# Before instance method execution

# Inside instance method

# After instance method execution

# Before class method execution

# Inside class method

# After class method execution
```

Decorators in Python are powerful tools that enable modular and reusable code by allowing the addition of functionalities or modifications without altering the original code. They promote cleaner code structures and facilitate the implementation of cross-cutting concerns like logging, authentication, caching, etc., in an OOP paradigm.

10. What do you understand about Descriptive Statistics? Explain by Example.

Descriptive statistics refers to a set of summary statistics that describe and summarize the main features or characteristics of a dataset. These statistics help in understanding the basic properties of the data, providing insights into its central tendency, dispersion, shape, and more.

Examples of Descriptive Statistics:

Measures of Central Tendency:

- Mean: It's the average value of the dataset.
- Median: The middle value when the dataset is sorted. It's less sensitive to outliers than the mean.
- Mode: The most frequent value(s) in the dataset.

Measures of Dispersion:

- Range: The difference between the maximum and minimum values in the dataset.
- Variance: Measures how far individual data points are from the mean.
- Standard Deviation: Square root of the variance, indicating the average deviation from the mean.

Measures of Distribution Shape:

- Skewness: Measures the asymmetry of the dataset distribution.
- Kurtosis: Measures the "peakedness" or "flatness" of the distribution

```
# Example dataset
data = [10, 15, 20, 20, 25, 30]

# Calculating mean, median, and mode
mean_value = sum(data) / len(data)
median_value = sorted(data)[len(data) // 2]
from statistics import mode
mode_value = mode(data)

print("Mean:", mean_value)     # Output: Mean: 20.0
print("Median:", median_value) # Output: Median: 20
print("Mode:", mode_value)     # Output: Mode: 20
```

```
# Example dataset
data = [10, 15, 20, 20, 25, 30]

# Calculating range, variance, and standard deviation
data_range = max(data) - min(data)
from statistics import variance, stdev
variance_value = variance(data)
std_deviation = stdev(data)

print("Range:", data_range)      # Output: Range: 20
print("Variance:", variance_value)  # Output: Variance: 25
print("Standard Deviation:", std_deviation) # Output: Standard Deviation: 5.0
```

```
# Example dataset
data = [10, 15, 20, 20, 25, 30]

# Calculating skewness and kurtosis
from scipy.stats import skew, kurtosis
skewness = skew(data)
kurtosis_value = kurtosis(data)

print("Skewness:", skewness)       # Output: Skewness: 0.0
print("Kurtosis:", kurtosis_value) # Output: Kurtosis: -1.2
```

These descriptive statistics offer a concise summary of the dataset, aiding in understanding its characteristics, distribution, and properties without delving into complex data analysis techniques. They form the foundation for initial data exploration and help in making informed decisions in various fields such as economics, finance, science, and more.

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

Inferential statistics is a branch of statistics that involves using sample data to make inferences or predictions about a larger population. It allows us to draw conclusions or make generalisations about a population based on a sample from that population. Inferential statistics involves hypothesis testing, estimation, and prediction.

Example of Inferential Statistics:

Consider a scenario where you want to estimate the average height of all students in a school, but it's impractical to measure every student's height due to time or resource constraints. Instead, you collect the heights of a sample of students and use inferential statistics to estimate the average height of the entire student population.

Hypothesis Testing:

You might have a hypothesis that the average height of students in the school is 65 inches.You collect a sample of 100 students and find that the sample mean height is 63 inches.Through hypothesis testing (such as a t-test), you assess whether the difference between the sample mean and the hypothesised population mean of 65 inches is statistically significant. This helps determine if your sample mean is likely to represent the true population mean.

Estimation:
Using the sample data, you calculate a confidence interval for the population mean height. For instance, you might determine that with 95% confidence, the true population mean height lies within a certain range.

Inferential statistics allows us to make conclusions or predictions about a larger population based on sample data. These techniques provide valuable insights and help in decision-making processes across various fields, including research, business, healthcare, and social sciences.