1. **Explain the procedural and object-oriented paradigm along with advantages and disadvantages.**

Procedural Programming:

Procedural programming is a programming paradigm based on the concept of procedure calls, in which blocks of code are organized into procedures or functions. Here are some key points:

- **Procedure or Function:** In procedural programming, the focus is on procedures or functions. These are blocks of code that perform a specific task. Data is passed between procedures explicitly through parameters.
- **Data and Procedures are Separate:** In procedural programming, data and procedures are kept separate. Data is usually stored in data structures like arrays or structs, and procedures act on this data.

Advantages of Procedural Programming:

1. **Simplicity:** Procedural programming tends to be simpler and easier to understand, especially for beginners or for solving smaller-scale problems.
2. **Efficiency:** Procedural languages often perform better in terms of execution speed and memory usage compared to some other paradigms.

Disadvantages of Procedural Programming:

1. **Limited Modularity:** As the codebase grows, it can become difficult to manage and maintain due to limited modularity. Changes to one part of the code may require modifications in multiple places.
2. **Limited Reusability:** Reusability is limited because functions are tied closely to the data they operate on. Functions cannot easily be reused in different contexts without modification.

Object-Oriented Programming (OOP):

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields, and code, in the form of procedures, known as methods. Here are some key points:

- **Objects:** Objects are instances of classes, which encapsulate data for the object and the methods that operate on that data.

- **Abstraction:** OOP emphasizes abstraction, allowing programmers to create models that mimic real-world entities. This makes the code more intuitive and easier to understand.
- **Inheritance:** Inheritance allows classes to inherit attributes and methods from other classes, enabling code reuse and promoting modularity.
- **Polymorphism:** Polymorphism allows objects to be treated as instances of their parent class, enabling flexibility and extensibility in the code.

Advantages of Object-Oriented Programming:

1. **Modularity:** OOP promotes modularity by encapsulating data within objects and providing clear interfaces for interacting with those objects. This makes code easier to manage and maintain.
2. **Reusability:** Objects and classes can be reused in different parts of the code or in different projects, leading to increased productivity and reduced development time.
3. **Flexibility and Extensibility:** OOP allows for easier adaptation and extension of code through features like inheritance and polymorphism. This makes it easier to accommodate changes and new requirements.

Disadvantages of Object-Oriented Programming:

1. **Complexity:** OOP can be more complex and harder to grasp, especially for beginners or for solving simpler problems.
2. **Performance Overhead:** OOP can sometimes incur a performance overhead due to features like dynamic dispatch and inheritance, although modern programming languages and compilers have mitigated much of this overhead.

What are pillars of OOP paradigm. Explain each one of them with example.

The four pillars of Object-Oriented Programming (OOP) are:

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**

Let's delve into each of them with examples:

1. Encapsulation: Encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit called a class. It restricts access to some of the object's components, typically hiding the internal state of an object and only allowing access through well-defined interfaces.

Example:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.__speed = 0  # Private attribute

    def accelerate(self, increment):
        self.__speed += increment

    def get_speed(self):
        return self.__speed

# Creating an instance of Car
my_car = Car("Toyota", "Corolla")
my_car.accelerate(20)
print(my_car.get_speed())  # Output: 20
```

In this example, `make` and `model` are public attributes, but `__speed` is a private attribute. This means that `__speed` cannot be accessed directly from outside the class; instead, it's accessed through the `accelerate()` and `get_speed()` methods.

2. Inheritance: Inheritance is a mechanism where a new class inherits properties and behaviors (methods) from an existing class. The existing class is called the base class or superclass, and the new class is called the derived class or subclass. It promotes code reusability and establishes a relationship between classes.

Example:

```
class Animal:
    def sound(self):
        pass

class Dog(Animal):  # Dog inherits from Animal
    def sound(self):
        return "Woof!"

class Cat(Animal):  # Cat inherits from Animal
    def sound(self):
        return "Meow!"

# Creating instances of Dog and Cat
dog = Dog()
cat = Cat()

print(dog.sound())  # Output: Woof!
print(cat.sound())  # Output: Meow!
```

In this example, both `Dog` and `Cat` classes inherit the `sound()` method from the `Animal` class. They can then override this method to provide their own implementation of the sound.

3. Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables flexibility and extensibility in the code by providing a way to perform a single action in different ways. Polymorphism can be achieved through method overriding or method overloading.

Example (Method Overriding):

```
class Shape:
    def area(self):
        pass

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

```
        self.length = length
        self.width = width

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

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

    def area(self):
        return 3.14 * self.radius * self.radius

# Creating instances of Rectangle and Circle
rectangle = Rectangle(5, 4)
circle = Circle(3)

print(rectangle.area())  # Output: 20
print(circle.area())  # Output: 28.26
```

In this example, both `Rectangle` and `Circle` classes have an `area()` method, which overrides the `area()` method of the superclass `Shape`. Despite being called on objects of different types, the `area()` method behaves polymorphically, providing the area calculation specific to each shape.

4. Abstraction: Abstraction refers to the process of hiding the complex implementation details and showing only the essential features of the object. It helps in reducing programming complexity and avoiding repetition by providing a clear separation between what an object does and how it achieves it.

Example:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

```
    def area(self):
        return self.length * self.width

# Creating an instance of Rectangle
rectangle = Rectangle(5, 4)
print(rectangle.area())  # Output: 20
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

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

# Creating an instance of Rectangle
rectangle = Rectangle(5, 4)
print(rectangle.area())  # Output: 20
```

In this example, `Shape` is an abstract class that defines the method `area()` as an abstract method using the `@abstractmethod` decorator. The `Rectangle` class inherits from `Shape` and provides an implementation of the `area()` method. Abstraction allows us to define a common interface for different shapes without worrying about their specific implementations.

Differentiate between data overloading and data overriding in Python.

In Python, there isn't a concept of "data overloading" and "data overriding" as there is in other programming languages like C++. However, I can explain the concepts of method overloading and method overriding in Python, which might be what you're referring to.

Method Overloading:

Method overloading refers to defining multiple methods in a class with the same name but with different signatures (i.e., different number or types of parameters). In Python, method overloading is not supported in the same way as in languages like Java or C++, where you can have multiple methods with the same name but different parameters.

However, you can achieve a similar effect in Python by using default parameter values or variable-length argument lists (`*args` and `**kwargs`). Python allows a single method name to be used for different behavior based on the parameters passed to it.

Example of method overloading in Python using default parameter values:

```
class MyClass:
    def method(self, x=None):
        if x is None:
            print("No argument provided")
        else:
            print("Argument provided:", x)

obj = MyClass()
obj.method()        # Output: No argument provided
obj.method(5)       # Output: Argument provided: 5
```

Method Overriding:

Method overriding refers to defining a method in a subclass that has the same name as a method in its superclass. The method in the subclass overrides the behavior of the method with the same name in the superclass. This allows for different behavior to be implemented in the subclass while maintaining a common interface with the superclass.

Example of method overriding in Python:

```
class Animal:
    def sound(self):
        print("Generic sound")

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

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

dog = Dog()
cat = Cat()

dog.sound()  # Output: Woof!
cat.sound()  # Output: Meow!
```

In this example, both `Dog` and `Cat` classes have a `sound()` method, which overrides the `sound()` method of the `Animal` superclass. When `sound()` is called on instances of `Dog` and `Cat`, the overridden method in each subclass is invoked, producing different sounds for each animal.

In summary, method overloading in Python is achieved through default parameter values or variable-length argument lists, while method overriding involves defining a method in a subclass with the same name as a method in its superclass to provide a different implementation.

---

2. Imagine you are designing a software system for a zoo. One of the key features is managing different types of animals. You decide to use abstract classes and methods to represent these animals.

    1. Create an abstract class called Animal with the following abstract methods:

- make_sound: This method should be implemented by subclasses to make the specific sound of each animal.
- move: This method should be implemented by subclasses to define how each animal moves.

2. Create two subclasses of Animal called Lion and Monkey.
    - For the Lion class, implement the make_sound method to roar and the move method to walk.
    - For the Monkey class, implement the make_sound method to chatter and the move method to climb.
3. Instantiate objects of both Lion and Monkey classes and demonstrate their functionalities by calling their make_sound and move methods.

```
# Abstract class Animal
class Animal:
    def make_sound(self):
        pass

    def move(self):
        pass

# Subclass Lion
class Lion(Animal):
    def make_sound(self):
        return "roar"

    def move(self):
        return "walk"

# Subclass Monkey
class Monkey(Animal):
    def make_sound(self):
        return "chatter"

    def move(self):
        return "climb"

# Instantiate Lion and Monkey objects
lion = Lion()
monkey = Monkey()

# Demonstrate functionalities
print("Lion sounds:", lion.make_sound())
```

```
print("Lion moves:", lion.move())

print("Monkey sounds:", monkey.make_sound())
print("Monkey moves:", monkey.move())
```

This code defines an abstract class Animal with abstract methods make_sound and move. It then creates two subclasses, Lion and Monkey, which implement these methods accordingly. Finally, it instantiates objects of both subclasses and demonstrates their functionalities by calling their methods.

==========

Demonstrate method overriding in Python to achieve runtime polymorphism with a code example

Method overriding in Python allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This enables runtime polymorphism, where the method that gets executed is determined by the type of object at runtime. Here's a simple example:

```
class Animal:
    def make_sound(self):
        print("Generic animal sound")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

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

class Cow(Animal):
    pass  # Cow inherits make_sound from Animal

# Function to demonstrate polymorphism
def animal_sound(animal):
    animal.make_sound()

# Creating instances of different animals
animal1 = Dog()
animal2 = Cat()
```

```
animal3 = Cow()

# Calling the function with different types of animals
animal_sound(animal1)  # Output: Woof!
animal_sound(animal2)  # Output: Meow!
animal_sound(animal3)  # Output: Generic animal sound
```

Explain the relationship between Python's garbage collection mechanism and destructors. How does Python manage memory deallocation, and when does the destructor get called?

1. **Garbage Collection Mechanism**: Python uses automatic memory management through a mechanism called reference counting combined with a cycle detector. Reference counting tracks the number of references to an object, and when the count reaches zero, the object is deallocated. However, reference counting alone cannot handle circular references, where objects reference each other in a loop. To handle such cases, Python employs a cyclic garbage collector that periodically checks for and collects cyclically referenced objects.
2. **Destructors**: In Python, a destructor is a special method named `__del__()` that gets called when an object is about to be destroyed or deallocated. It's used to perform cleanup actions before an object is removed from memory. Destructors are not explicitly called by the programmer; instead, they are automatically invoked by Python's garbage collector when the reference count of an object drops to zero or when circular references are detected and the cyclic garbage collector determines that an object is unreachable.
3. **Memory Deallocation**: When an object's reference count drops to zero, meaning there are no more references to that object, Python's garbage collector automatically deallocates the memory occupied by the object. If the object has a destructor (`__del__()` method), Python calls it just before the object is destroyed. The destructor can perform cleanup tasks such as closing files, releasing network connections, or releasing other external resources associated with the object.
4. **When the Destructor Gets Called**: The exact timing of destructor invocation can be somewhat unpredictable due to Python's garbage collection mechanism. The destructor is called when the object is being deallocated, which occurs when its reference count drops to zero. However, it's important to note that the precise moment when the destructor is called within the lifetime of the program is not deterministic. It depends on factors such as the timing of reference count drops, garbage collection cycles, and the order in which objects are destroyed.