Q.1 1. Explain what inheritance is in object-oriented programming and why it is used.

Inheritance is one of the key concepts in **Object-Oriented Programming (OOP)**, which allows a class to **inherit properties and behaviors** (methods and attributes) from another class. It promotes reusability, modularity, and maintainability of code.

Key Concepts of Inheritance:

1. **Parent Class (Base Class)**: The class that is inherited from. It contains common attributes and methods that can be shared.
2. **Child Class (Derived Class)**: The class that inherits from the parent class. It can reuse or extend the functionality of the parent class by adding new methods or overriding existing ones.

Types of Inheritance:

- **Single Inheritance**: A child class inherits from one parent class.
- **Multiple Inheritance**: A child class inherits from more than one parent class.
- **Multilevel Inheritance**: A class inherits from another class, which itself is a child of a different class.
- **Hierarchical Inheritance**: Multiple child classes inherit from a single parent class.
- **Hybrid Inheritance**: A combination of two or more types of inheritance.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):  # Overriding the speak method
        print("Dog barks")

# Creating an object of Dog class
dog = Dog()
dog.speak()  # Output: Dog barks
```

In this example:

- The `Dog` class inherits from the `Animal` class.
- The `Dog` class has its own implementation of the `speak` method, overriding the one in the `Animal` class.

Why Inheritance is Used:

1. **Code Reusability**: Inheritance allows you to reuse code from the parent class in the child class, reducing redundancy. Instead of writing the same code in multiple places, you write it once in a base class and reuse it.

2. **Maintainability**: When a bug or update is needed in a common behavior, it can be fixed in the parent class, and all child classes will automatically inherit the fix.
3. **Extensibility**: Inheritance allows a child class to build upon the functionality of the parent class by adding or modifying methods. This enables developers to create more specialized versions of a class.
4. **Polymorphism**: Through inheritance, classes can implement the same method in different ways, which is known as polymorphism. This allows objects of different classes to be treated as objects of a common parent class.

Real-World Analogy:

Think of **Inheritance** like a **family tree**: children inherit traits (like eye color or height) from their parents but can also develop their own unique traits. Similarly, in OOP, a child class inherits attributes and methods from a parent class but can also define its own specialized behaviors.

In summary, **inheritance** helps to organize code, make it more efficient, and allow developers to create more complex systems based on shared behaviors.

Q2. 2. Discuss the concept of single inheritance and multiple inheritance, highlighting their differences and advantages.

Single Inheritance

Single inheritance is a type of inheritance in object-oriented programming (OOP) where a class (child class) inherits from only **one parent class**.

Example of Single Inheritance:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):  # Dog inherits from Animal (single inheritance)
    def speak(self):
        print("Dog barks")

# Create an object of the Dog class
dog = Dog()
dog.speak()  # Output: Dog barks
```

In the above example, `Dog` inherits from `Animal`, which means `Dog` gets all the properties and methods of the `Animal` class, but it can also override or extend them.

Multiple Inheritance

Multiple inheritance is a type of inheritance where a class (child class) can inherit from **more than one parent class**.

Example of Multiple Inheritance:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Mammal:
    def have_fur(self):
        print("Mammals have fur")

class Dog(Animal, Mammal):  # Dog inherits from both Animal and Mammal
(multiple inheritance)
    def speak(self):
        print("Dog barks")

# Create an object of the Dog class
dog = Dog()
dog.speak()  # Output: Dog barks
dog.have_fur()  # Output: Mammals have fur
```

In this example, `Dog` inherits both from `Animal` and `Mammal`. This allows the `Dog` class to access the `speak` method from `Animal` and the `have_fur` method from `Mammal`.

Differences Between Single and Multiple Inheritance

Aspect	Single Inheritance	Multiple Inheritance
Number of Parent Classes	A class inherits from only one parent class.	A class inherits from more than one parent class.
Complexity	Simpler to implement and understand.	More complex, as the child class can inherit from multiple sources.
Ambiguity	No ambiguity, as there is only one parent class.	Ambiguity can arise if different parent classes have methods with the same name (known as the "diamond problem").
Code Reusability	Limited reusability since it inherits from only one class.	Higher reusability because the child class can inherit features from multiple parent classes.
Inheritance Hierarchy	Straightforward, with a clear, linear hierarchy.	More complex hierarchy with possible multiple branches.

Advantages and Disadvantages

Single Inheritance:

Advantages:

- **Simplicity**: It is easier to implement and understand since the class hierarchy is linear and straightforward.
- **No Ambiguity**: There's no risk of ambiguity or conflicts between parent classes (e.g., method name clashes).
- **Better Maintainability**: Since there's only one parent, it's easier to manage changes or updates to the base class, and the effects on the child class are more predictable.

Disadvantages:

- **Limited Reusability**: Since the class can inherit from only one parent, you might need to duplicate code if similar functionality is needed from different classes.
- **Less Flexibility**: If you want to combine features from multiple sources, single inheritance is restrictive.

Multiple Inheritance:

Advantages:

- **Increased Reusability**: The child class can reuse code from multiple parent classes, which reduces redundancy.
- **Flexibility**: Multiple inheritance allows combining the behavior of multiple classes into a single class. It's useful when you want to combine features from different sources.

Disadvantages:

- **Increased Complexity**: The hierarchy becomes more complex, and understanding the flow of inheritance may become challenging.
- **Ambiguity (Diamond Problem)**: If two parent classes have methods with the same name, the child class might face ambiguity in terms of which method to inherit or call. This problem is known as the "diamond problem."

  Example of the diamond problem:

```
class A:
    def speak(self):
        print("Class A speaks")

class B(A):
    def speak(self):
        print("Class B speaks")

class C(A):
    def speak(self):
        print("Class C speaks")

class D(B, C):  # D inherits from both B and C
    pass

d = D()
d.speak()  # Which speak method is called? B or C?
```

In the above case, there might be confusion about which `speak` method is inherited by class `D`.

Python resolves this using the **Method Resolution Order (MRO)**, but not all languages handle multiple inheritance the same way.

When to Use Single or Multiple Inheritance:

- **Use Single Inheritance** when the relationship between classes is simple, clear, and straightforward. It is ideal for cases where a child class logically extends the functionality of one parent class.
- **Use Multiple Inheritance** when a class needs to inherit functionality from more than one source, such as when combining features from multiple classes or when modeling complex real-world entities that exhibit multiple behaviors. However, ensure the design remains manageable and avoid conflicts between parent classes.

In conclusion, **single inheritance** is simpler and less prone to issues, while **multiple inheritance** provides greater flexibility but requires careful management to avoid complications like ambiguity or method clashes.

Q3. Explain the terms "base class" and "derived class" in the context of inheritance.

In the context of **inheritance** in object-oriented programming (OOP), the terms **"base class"** and **"derived class"** are used to describe the relationship between classes in an inheritance hierarchy.

Base Class (Parent Class or Superclass)

- The **base class** (also called the **parent class** or **superclass**) is the class that is inherited from.
- It contains common attributes (data members) and methods (functions) that are shared by one or more other classes.
- The base class defines the general characteristics and behaviors that can be extended or customized by the **derived class**.

Example:

```
class Animal:  # Animal is the base class
    def speak(self):
        print("Animal makes a sound")
```

In this example, `Animal` is the **base class**, and it defines a method `speak()` that is common to all animals.

Derived Class (Child Class or Subclass)

- The **derived class** (also called the **child class** or **subclass**) is the class that inherits from one or more base classes.
- It can **reuse** the attributes and methods from the base class, **extend** them by adding new attributes or methods, or **override** the methods of the base class to provide specialized behavior.
- The derived class has all the features of the base class, but it can also introduce its own unique features or modify existing ones.

Example:

```
class Dog(Animal):  # Dog is the derived class, inheriting from Animal
    def speak(self):  # Overriding the speak method
        print("Dog barks")
```

In this case, `Dog` is the **derived class** that inherits from the `Animal` class. It **overrides** the `speak()` method to make it more specific for dogs.

Key Points:

- The **base class** provides common functionality, acting as a template or foundation for other classes.
- The **derived class** inherits the properties and behaviors of the base class and can also **extend** or **override** them to suit its specific needs.

Real-World Analogy:

Think of the **base class** as a **general blueprint** for something, like a **vehicle**. The **derived class** would be a more specific vehicle, like a **car** or **bike**, which inherits the general properties (like having wheels and moving) from the base class (vehicle) but also adds specific behaviors (like having a horn for the car).

Inheritance Example (Base Class and Derived Class):

```
# Base Class
class Animal:
    def eat(self):
        print("Eating food")

# Derived Class
class Dog(Animal):
    def bark(self):
        print("Barking")

# Create an object of the Dog class
dog = Dog()
dog.eat()  # Inherited from the Animal class
dog.bark()  # Defined in the Dog class
```

- In this example, `Animal` is the **base class** and `Dog` is the **derived class**.

- The `Dog` class **inherits** the `eat()` method from the `Animal` class, and it also **adds** its own method `bark()`.

Summary:

- **Base Class**: The class from which other classes inherit. It provides common functionality.
- **Derived Class**: The class that inherits from the base class. It can reuse, extend, or override the functionality of the base class.

In inheritance, the derived class can inherit and/or modify the behavior of the base class to create specialized versions of the original class.

Q4.

The **"protected"** access modifier plays a significant role in **inheritance** in object-oriented programming (OOP) by controlling the visibility and accessibility of class members (attributes and methods) within the class hierarchy. It is used to define class members that should be accessible to the class itself, its **derived (child) classes**, and **subclasses**, but **not accessible from outside the class hierarchy**.

Significance of the "Protected" Access Modifier in Inheritance:

1. **Controlled Visibility**:
   - **Protected** members are **invisible to external classes** (i.e., classes that do not inherit from the class containing the protected member).
   - They are only **accessible within the class and its subclasses** (derived classes), allowing for a controlled way of sharing data or behavior with child classes while keeping it hidden from external code.
2. **Inheritance Use**:
   - The **protected** modifier is especially useful in inheritance scenarios. Derived classes can access and **reuse** or **modify** the protected attributes and methods of the parent class.
   - This supports the concept of **encapsulation**, allowing the base class to expose some internal members for use by its subclasses, but hiding those members from the outside world.
3. **Encapsulation and Extensibility**:
   - It allows **encapsulation** because the base class hides internal details from the outside world while still enabling its subclasses to extend or modify behaviors.
   - At the same time, it provides **extensibility** because subclasses can access and extend the functionality of protected members.

How "Protected" Differs from "Private" and "Public" Modifiers:

To understand the significance of "protected," we should also compare it with the **private** and **public** access modifiers, which define how class members can be accessed in different contexts.

1. Public Modifier:

- **Access**: Public members are accessible **from anywhere** — both inside the class and outside it.
- **Visibility**: Public members are fully exposed, meaning they can be accessed by any other code that has visibility of the class.

Example in Java:

```
class Animal {
    public String name;

    public void speak() {
        System.out.println(name + " makes a sound");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println(name + " barks");
    }
}
```

- In this example, `name` and `speak()` are **public** and can be accessed by any external code.

Advantages of Public:

- Full access to the class members from any other code.

Disadvantages of Public:

- Less control over how class members are accessed and modified.
- Potential for misuse or accidental modification, violating the encapsulation principle.

2. Private Modifier:

- **Access**: Private members are **only accessible within the class** where they are defined. They are not accessible in subclasses or from outside the class.
- **Visibility**: Private members are **completely hidden** from external code and even from derived classes.

Example in Java:

```
class Animal {
    private String name;  // Private attribute

    private void speak() {  // Private method
        System.out.println(name + " makes a sound");
    }

    public void setName(String name) {
        this.name = name;  // Public setter to modify the private attribute
    }
}

class Dog extends Animal {
    public void bark() {
        // Cannot access name or speak() directly because they are private in
Animal class
    }
}
```

- Here, `name` and `speak()` are **private** and cannot be accessed from the `Dog` class or from external code. Only public methods (like `setName`) can be used to interact with them.

Advantages of Private:

- Full encapsulation: Private members cannot be accessed directly by other classes, ensuring that only methods in the class can modify the internal state.

Disadvantages of Private:

- Prevents inheritance and reuse, as derived classes cannot access private members of the base class.
- It can be restrictive, especially if subclasses need to use or modify the private attributes/methods.

3. Protected Modifier:

- **Access**: Protected members are accessible within the class where they are defined, and **in any subclass** (derived class), but **not from external code**.
- **Visibility**: Protected members are **visible to derived classes** but hidden from outside the class hierarchy.

Example in Java:

```
class Animal {
    protected String name;  // Protected attribute

    protected void speak() {  // Protected method
        System.out.println(name + " makes a sound");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println(name + " barks");  // Accessing protected member
in the subclass
    }
}
```

- Here, `name` and `speak()` are **protected**, so they can be accessed and modified within the `Dog` class (which is a subclass of `Animal`), but they cannot be accessed from outside the `Animal` or `Dog` class.

Advantages of Protected:

- **Controlled inheritance**: Allows subclasses to inherit and modify behaviors without exposing class details to external code.
- Provides **encapsulation** while allowing derived classes to interact with or extend the base class's functionality.

Disadvantages of Protected:

- **Limited visibility**: While protected members are accessible to subclasses, they still cannot be accessed directly by external code, which can sometimes limit flexibility when external code needs to interact with these members.

Summary of Differences:

Modifier	**Access within the class**	**Access in subclass**	**Access from external code**	**Purpose**
Public	Yes	Yes	Yes	Fully accessible, no restrictions
Private	Yes	No	No	Only accessible within the class, strict encapsulation
Protected	Yes	Yes	No	Accessible within the class and by subclasses, but hidden from outside code

Conclusion:

- **Protected** members provide a balance between **public** and **private** members, offering visibility to **subclasses** while maintaining some level of encapsulation from external code.
- **Private** members are used when you want to fully hide data and methods from both external code and subclasses.
- **Public** members are exposed for unrestricted access, which is useful in cases where external access is needed but can lead to weaker encapsulation.

The choice between **protected**, **private**, and **public** depends on the level of encapsulation and access control you want to enforce in your class design, especially in the context of inheritance.

Q.4 5. What is the purpose of the "super" keyword in inheritance? Provide an example.

The **`super`** keyword in object-oriented programming (OOP) is used to refer to the **parent (base) class** from within a **child (derived) class**. It serves several important purposes in the context of inheritance, such as:

Purposes of the `super` Keyword:

1. **Accessing Parent Class Methods:**
   - The `super` keyword allows a child class to **call methods** that are defined in its parent class. This is useful when the child class wants to **extend** or **override** a method but still needs to invoke the method from the parent class.
2. **Accessing Parent Class Constructors:**
   - It can be used to **call the constructor** of the parent class from within the child class, which is particularly useful when you need to initialize the parent class before using it in the child class. This ensures that the parent class is properly set up before the child class can take over.
3. **Avoiding Method Name Conflicts (Overriding):**
   - In case the child class overrides a method of the parent class, the `super` keyword can be used to invoke the **parent class's version** of the method within the overridden method. This is helpful for combining both the parent's and child's behavior.

Example in Python:

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

    def speak(self):
        print(f"{self.name} makes a sound")

# Derived class (Child class)
class Dog(Animal):
    def __init__(self, name, breed):
        # Call the constructor of the parent class using super()
        super().__init__(name)
        self.breed = breed

    def speak(self):
        # Call the speak method of the parent class using super()
        super().speak()  # This calls Animal's speak method
        print(f"{self.name} barks")

# Create an object of the Dog class
dog = Dog("Buddy", "Golden Retriever")
dog.speak()
```

Explanation:

1. **Constructor Call Using `super()`:**

    - In the `Dog` class, the `__init__` method uses `super().__init__(name)` to call the **constructor** of the `Animal` class. This ensures that the `name` attribute is initialized in the parent class (`Animal`), avoiding the need to duplicate code in the child class (`Dog`).
2. **Method Call Using `super()`:**
    - In the `Dog` class, the `speak()` method calls `super().speak()` to invoke the `speak()` method from the parent class (`Animal`). This allows the child class to add its own functionality (e.g., `print(f"{self.name} barks")`) while still using the method of the parent class.

Output of the Example:

```
Buddy makes a sound
Buddy barks
```

- First, the `super().speak()` invokes the `speak()` method from the parent class, which prints `"Buddy makes a sound"`.
- Then, the child class adds its behavior by printing `"Buddy barks"`.

Key Points About `super`:

- `super()` is often used in **multiple inheritance** to manage the method resolution order (MRO), ensuring that the correct methods from the base classes are called in the proper order.
- It allows for **code reuse** by enabling the child class to build on the behavior of the parent class rather than completely overriding or duplicating functionality.

Summary:

The `super` keyword is a powerful tool in inheritance that allows a child class to:

- Call the **parent class constructor** to initialize the inherited properties.
- Invoke the **parent class methods**, enabling method extension and proper method overriding.

It is essential for writing efficient and maintainable object-oriented code, especially when dealing with class hierarchies and ensuring proper initialization and behavior propagation from parent to child classes.

Q.6 Create a base class called "Vehicle" with attributes like "make", "model", and "year". Then, create a derived class called "Car" that inherits from "Vehicle" and adds an attribute called "fuel_type". Implement appropriate methods in both classes.

To create a base class called `Vehicle` with the attributes "make", "model", and "year", and a derived class called `Car` that inherits from `Vehicle` and adds the "fuel_type" attribute, we can define both classes in Python. The base class will include methods to display the details of the vehicle, while the derived class will extend this functionality by adding the `fuel_type` attribute.

Code Implementation:

```
# Base class
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make        # Make of the vehicle
        self.model = model      # Model of the vehicle
        self.year = year        # Year of manufacture

    def display_details(self):
        # Method to display vehicle details
        print(f"Vehicle Details:\nMake: {self.make}\nModel:
{self.model}\nYear: {self.year}")

# Derived class
class Car(Vehicle):
    def __init__(self, make, model, year, fuel_type):
        # Call the parent class (Vehicle) constructor using super()
        super().__init__(make, model, year)
        self.fuel_type = fuel_type  # Additional attribute specific to Car

    def display_details(self):
        # Call the parent class method to display basic vehicle details
        super().display_details()
        # Display the additional Car attribute
        print(f"Fuel Type: {self.fuel_type}")

# Create an instance of the Car class
car = Car("Toyota", "Corolla", 2020, "Petrol")

# Display the details of the car
car.display_details()
```

Explanation of the Code:

1. **Base Class (`Vehicle`)**:
    - The `Vehicle` class has three attributes: `make`, `model`, and `year`, which are initialized via the constructor `__init__()`.
    - The `display_details()` method prints the details of the vehicle.
2. **Derived Class (`Car`)**:
    - The `Car` class inherits from the `Vehicle` class, and adds an additional attribute `fuel_type`, which represents the type of fuel the car uses (e.g., Petrol, Diesel, Electric, etc.).
    - In the `__init__()` method, we use `super().__init__(make, model, year)` to call the constructor of the parent class `Vehicle` and initialize the common attributes. Then, we initialize the `fuel_type` attribute, which is specific to the `Car` class.

    - The `display_details()` method in `Car` calls the `display_details()` method of the parent class to display the basic vehicle details and then prints the additional `fuel_type` attribute.
3. **Creating an Object**:
    - We create an instance of the `Car` class, passing values for all attributes (make, model, year, and fuel_type).
    - Calling `car.display_details()` will display the complete details of the car, including the inherited attributes from the `Vehicle` class and the additional `fuel_type` attribute from the `Car` class.

Output:

```
Vehicle Details:
Make: Toyota
Model: Corolla
Year: 2020
Fuel Type: Petrol
```

Summary:

- The `Vehicle` class serves as the **base class** with common attributes like make, model, and year.
- The `Car` class is the **derived class** that inherits from `Vehicle` and adds a new attribute `fuel_type`.
- The `super()` function is used to call the parent class constructor to initialize the inherited attributes.
- The `display_details()` method in the derived class extends the functionality of the base class method, adding specific details about the fuel type.

Q7. Create a base class called "Employee" with attributes like "name" and "salary." Derive two classes, "Manager" and "Developer," from "Employee." Add an additional attribute called "department" for the "Manager" class and "programming_language" for the "Developer" class.

Here is the implementation of the **Employee** base class, and the **Manager** and **Developer** derived classes. We'll define the attributes `name` and `salary` in the base class, and add the `department` attribute for the `Manager` class and `programming_language` attribute for the `Developer` class.

Code Implementation:

```
# Base class
class Employee:
    def __init__(self, name, salary):
        self.name = name      # Name of the employee
        self.salary = salary  # Salary of the employee

    def display_details(self):
        # Method to display basic employee details
        print(f"Employee Name: {self.name}")
        print(f"Salary: {self.salary}")

# Derived class for Manager
class Manager(Employee):
    def __init__(self, name, salary, department):
        # Call the parent class constructor using super()
        super().__init__(name, salary)
        self.department = department  # Additional attribute specific to
Manager

    def display_details(self):
        # Call the parent class method to display basic employee details
        super().display_details()
        # Display the additional Manager-specific attribute
        print(f"Department: {self.department}")

# Derived class for Developer
class Developer(Employee):
    def __init__(self, name, salary, programming_language):
        # Call the parent class constructor using super()
        super().__init__(name, salary)
        self.programming_language = programming_language  # Additional
attribute specific to Developer

    def display_details(self):
        # Call the parent class method to display basic employee details
        super().display_details()
        # Display the additional Developer-specific attribute
        print(f"Programming Language: {self.programming_language}")

# Creating instances of Manager and Developer classes
manager = Manager("Alice", 95000, "HR")
developer = Developer("Bob", 80000, "Python")

# Display details of Manager and Developer
print("\nManager Details:")
manager.display_details()

print("\nDeveloper Details:")
developer.display_details()
```

Explanation of the Code:

1. **Base Class (`Employee`)**:
    - The `Employee` class has two attributes: `name` (the name of the employee) and `salary` (the salary of the employee).
    - The `display_details()` method in `Employee` prints the `name` and `salary` of the employee.
2. **Derived Class (`Manager`)**:
    - The `Manager` class inherits from `Employee` and adds an additional attribute: `department`, which indicates the department the manager is responsible for (e.g., HR, Sales, etc.).

    - The `__init__()` method in `Manager` uses `super().__init__(name, salary)` to initialize the inherited `name` and `salary` attributes, and also initializes the `department` attribute.
    - The `display_details()` method in `Manager` calls the `display_details()` method of the base class to print the employee details, and then prints the manager's department.
3. **Derived Class (`Developer`)**:
    - The `Developer` class inherits from `Employee` and adds an additional attribute: `programming_language`, which specifies the language the developer works with (e.g., Python, Java, etc.).
    - The `__init__()` method in `Developer` uses `super().__init__(name, salary)` to initialize the inherited `name` and `salary` attributes, and also initializes the `programming_language` attribute.
    - The `display_details()` method in `Developer` calls the `display_details()` method of the base class to print the employee details, and then prints the developer's programming language.
4. **Creating Objects**:
    - We create an instance of the `Manager` class (with name "Alice", salary 95000, and department "HR").
    - We also create an instance of the `Developer` class (with name "Bob", salary 80000, and programming language "Python").
    - Finally, we call `display_details()` for both the `Manager` and `Developer` instances to print their details.

Output:

```
Manager Details:
Employee Name: Alice
Salary: 95000
Department: HR

Developer Details:
Employee Name: Bob
Salary: 80000
Programming Language: Python
```

Summary:

- **Base Class (`Employee`)**: Contains shared attributes (`name`, `salary`) for all employee types and a method to display these details.
- **Derived Class (`Manager`)**: Adds a `department` attribute, representing the department of the manager, and overrides the `display_details()` method to show this information.
- **Derived Class (`Developer`)**: Adds a `programming_language` attribute, representing the programming language a developer specializes in, and overrides the `display_details()` method to display this attribute.

This structure demonstrates basic inheritance and how subclasses can extend the functionality of a base class by adding additional attributes and methods.

Q.8. Design a base class called "Shape" with attributes like "colour" and "border_width." Create derived classes, "Rectangle" and "Circle," that inherit from "Shape" and add specific attributes like "length" and "width" for the "Rectangle" class and "radius" for the "Circle" class.

To design a base class called `Shape` with attributes like `colour` and `border_width`, and then create two derived classes, `Rectangle` and `Circle`, that inherit from `Shape` and add specific attributes like `length` and `width` for the `Rectangle` class and `radius` for the `Circle` class, we'll implement the following:

Code Implementation:

```
# Base class
class Shape:
    def __init__(self, colour, border_width):
        self.colour = colour          # Colour of the shape
        self.border_width = border_width  # Border width of the shape

    def display_details(self):
        # Method to display common shape details
        print(f"Colour: {self.colour}")
        print(f"Border Width: {self.border_width}")

# Derived class for Rectangle
class Rectangle(Shape):
    def __init__(self, colour, border_width, length, width):
        # Call the parent class constructor using super()
        super().__init__(colour, border_width)
        self.length = length      # Length of the rectangle
        self.width = width        # Width of the rectangle

    def display_details(self):
        # Call the parent class method to display common shape details
        super().display_details()
        # Display the additional Rectangle-specific attributes
        print(f"Length: {self.length}")
        print(f"Width: {self.width}")

    def area(self):
        # Calculate and return the area of the rectangle
        return self.length * self.width

# Derived class for Circle
class Circle(Shape):
    def __init__(self, colour, border_width, radius):
        # Call the parent class constructor using super()
        super().__init__(colour, border_width)
        self.radius = radius      # Radius of the circle

    def display_details(self):
        # Call the parent class method to display common shape details
        super().display_details()
        # Display the additional Circle-specific attribute
        print(f"Radius: {self.radius}")

    def area(self):
        # Calculate and return the area of the circle
        import math
        return math.pi * (self.radius ** 2)

# Creating instances of Rectangle and Circle
rectangle = Rectangle("Red", 2, 5, 3)
circle = Circle("Blue", 1, 4)

# Display details and area of the rectangle
print("\nRectangle Details:")
rectangle.display_details()
print(f"Area: {rectangle.area()}")

# Display details and area of the circle
print("\nCircle Details:")
circle.display_details()
print(f"Area: {circle.area()}")
```

Explanation of the Code:

1. **Base Class (`Shape`)**:
   - The `Shape` class has two attributes:
     - `colour`: Represents the colour of the shape.
     - `border_width`: Represents the border width of the shape.
   - The `display_details()` method prints the common details (colour and border width) for any shape.
2. **Derived Class (`Rectangle`)**:
   - The `Rectangle` class inherits from `Shape` and adds two specific attributes:
     - `length`: The length of the rectangle.
     - `width`: The width of the rectangle.
   - The `display_details()` method in the `Rectangle` class calls the `display_details()` method from the `Shape` class and prints the additional `length` and `width` attributes specific to the rectangle.
   - The `area()` method calculates and returns the area of the rectangle using the formula `length * width`.
3. **Derived Class (`Circle`)**:
   - The `Circle` class inherits from `Shape` and adds one specific attribute:
     - `radius`: The radius of the circle.
   - The `display_details()` method in the `Circle` class calls the `display_details()` method from the `Shape` class and prints the `radius` attribute specific to the circle.
   - The `area()` method calculates and returns the area of the circle using the formula `π * radius^2`.
4. **Creating Objects**:

- We create an instance of the `Rectangle` class with a red colour, a border width of 2, a length of 5, and a width of 3.
- We create an instance of the `Circle` class with a blue colour, a border width of 1, and a radius of 4.
- Then, we call the `display_details()` method for both objects to print their details and the `area()` method to compute and display their respective areas.

Output:

```
Rectangle Details:
Colour: Red
Border Width: 2
Length: 5
Width: 3
Area: 15

Circle Details:
Colour: Blue
Border Width: 1
Radius: 4
Area: 50.26548245743669
```

Summary:

- The **`Shape`** class is the base class with common attributes (`colour` and `border_width`) and a method (`display_details()`) to display these attributes.
- The **`Rectangle`** class adds specific attributes (`length` and `width`) and includes methods to display rectangle details and compute its area.
- The **`Circle`** class adds a `radius` attribute and includes methods to display circle details and compute its area.
- The `super()` function is used to call the parent class (`Shape`) constructor and methods from within the derived classes (`Rectangle` and `Circle`).

This design demonstrates inheritance and how derived classes can extend and specialize the functionality of a base class while still sharing common attributes and methods.

Q.9 Create a base class called "Device" with attributes like "brand" and "model." Derive two classes, "Phone" and "Tablet," from "Device." Add specific attributes like "screen_size" for the "Phone" class and "battery_capacity" for the "Tablet" class.

To create a base class called `Device` with attributes like `brand` and `model`, and then derive two classes, `Phone` and `Tablet`, each with their specific attributes (`screen_size` for the `Phone` class and `battery_capacity` for the `Tablet` class), we can implement the following code:

Code Implementation:

```
# Base class
class Device:
    def __init__(self, brand, model):
        self.brand = brand    # Brand of the device
        self.model = model    # Model of the device

    def display_details(self):
        # Method to display basic device details
        print(f"Brand: {self.brand}")
        print(f"Model: {self.model}")

# Derived class for Phone
class Phone(Device):
    def __init__(self, brand, model, screen_size):
        # Call the parent class constructor using super()
        super().__init__(brand, model)
        self.screen_size = screen_size  # Screen size of the phone

    def display_details(self):
        # Call the parent class method to display common device details
        super().display_details()
        # Display the additional Phone-specific attribute
        print(f"Screen Size: {self.screen_size} inches")

# Derived class for Tablet
class Tablet(Device):
    def __init__(self, brand, model, battery_capacity):
        # Call the parent class constructor using super()
        super().__init__(brand, model)
        self.battery_capacity = battery_capacity  # Battery capacity of the
tablet

    def display_details(self):
        # Call the parent class method to display common device details
        super().display_details()
        # Display the additional Tablet-specific attribute
        print(f"Battery Capacity: {self.battery_capacity} mAh")

# Creating instances of Phone and Tablet classes
phone = Phone("Samsung", "Galaxy S21", 6.2)
tablet = Tablet("Apple", "iPad Pro", 9720)

# Display details of the Phone
print("\nPhone Details:")
phone.display_details()

# Display details of the Tablet
print("\nTablet Details:")
tablet.display_details()
```

Explanation of the Code:

1. **Base Class (`Device`)**:
    - The `Device` class has two attributes:
        - `brand`: The brand of the device (e.g., Samsung, Apple).
        - `model`: The model of the device (e.g., Galaxy S21, iPad Pro).

- The `display_details()` method in `Device` prints the `brand` and `model` attributes.

2. **Derived Class (`Phone`)**:
   - The `Phone` class inherits from `Device` and adds the `screen_size` attribute, which represents the size of the phone's screen in inches (e.g., 6.2 inches).
   - The `__init__()` method in `Phone` uses `super().__init__(brand, model)` to call the parent class constructor and initialize the inherited attributes (`brand`, `model`). It also initializes the `screen_size` attribute.
   - The `display_details()` method in `Phone` calls the `display_details()` method from the base class to print the `brand` and `model` and then adds the specific `screen_size` attribute.
3. **Derived Class (`Tablet`)**:
   - The `Tablet` class inherits from `Device` and adds the `battery_capacity` attribute, which represents the tablet's battery capacity in milliampere-hours (mAh) (e.g., 9720 mAh).
   - The `__init__()` method in `Tablet` uses `super().__init__(brand, model)` to initialize the inherited attributes (`brand`, `model`). It also initializes the `battery_capacity` attribute.
   - The `display_details()` method in `Tablet` calls the `display_details()` method from the base class to print the `brand` and `model`, and then adds the `battery_capacity` attribute.
4. **Creating Objects**:
   - We create an instance of the `Phone` class with the brand "Samsung", model "Galaxy S21", and screen size of 6.2 inches.
   - We create an instance of the `Tablet` class with the brand "Apple", model "iPad Pro", and battery capacity of 9720 mAh.
   - Then, we call the `display_details()` method for both the `Phone` and `Tablet` objects to print their details.

Output:

```
Phone Details:
Brand: Samsung
Model: Galaxy S21
Screen Size: 6.2 inches

Tablet Details:
Brand: Apple
Model: iPad Pro
Battery Capacity: 9720 mAh
```

Summary:

- The **`Device`** class is the base class with common attributes (`brand` and `model`) and a method (`display_details()`) to display these attributes.
- The **`Phone`** class adds the `screen_size` attribute and overrides the `display_details()` method to display the phone's details.

- The **`Tablet`** class adds the `battery_capacity` attribute and overrides the `display_details()` method to display the tablet's details.
- The `super()` function is used to call the parent class constructor and methods from within the derived classes (`Phone` and `Tablet`).

This design demonstrates how inheritance allows derived classes to extend and specialize the functionality of a base class while still sharing common properties and methods.

Q10. 10. Create a base class called "BankAccount" with attributes like "account_number" and "balance." Derive two classes, "SavingsAccount" and "CheckingAccount," from "BankAccount." Add specific methods like "calculate_interest" for the "SavingsAccount" class and "deduct_fees" for the "CheckingAccount" class.

To create a base class called `BankAccount` with attributes like `account_number` and `balance`, and then derive two classes, `SavingsAccount` and `CheckingAccount`, each with specific methods (`calculate_interest` for `SavingsAccount` and `deduct_fees` for `CheckingAccount`), we can follow the steps below.

Code Implementation:

```
# Base class
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number  # Account number of the bank
account
        self.balance = balance                # Balance of the bank account

    def display_account_details(self):
        # Method to display the account details
        print(f"Account Number: {self.account_number}")
        print(f"Balance: ${self.balance:.2f}")

# Derived class for SavingsAccount
class SavingsAccount(BankAccount):
    def __init__(self, account_number, balance, interest_rate):
        # Call the parent class constructor using super()
        super().__init__(account_number, balance)
        self.interest_rate = interest_rate  # Interest rate for savings
account

    def calculate_interest(self):
        # Method to calculate interest based on balance and interest rate
        interest = self.balance * self.interest_rate / 100
        return interest

    def display_account_details(self):
        # Call the parent class method to display account details
        super().display_account_details()
        # Display interest rate
        print(f"Interest Rate: {self.interest_rate}%")
```

```
# Derived class for CheckingAccount
class CheckingAccount(BankAccount):
    def __init__(self, account_number, balance, monthly_fee):
        # Call the parent class constructor using super()
        super().__init__(account_number, balance)
        self.monthly_fee = monthly_fee  # Monthly fee for checking account

    def deduct_fees(self):
        # Method to deduct monthly fees from the account balance
        self.balance -= self.monthly_fee
        if self.balance < 0:
            self.balance = 0  # Ensure balance does not go below zero

    def display_account_details(self):
        # Call the parent class method to display account details
        super().display_account_details()
        # Display monthly fee
        print(f"Monthly Fee: ${self.monthly_fee:.2f}")

# Creating instances of SavingsAccount and CheckingAccount
savings = SavingsAccount("12345", 1000, 2)  # 2% interest rate
checking = CheckingAccount("67890", 500, 10)  # $10 monthly fee

# Display details of the SavingsAccount
print("\nSavings Account Details:")
savings.display_account_details()
interest = savings.calculate_interest()
print(f"Interest Earned: ${interest:.2f}")

# Display details of the CheckingAccount before deducting fees
print("\nChecking Account Details (Before Fees):")
checking.display_account_details()
checking.deduct_fees()  # Deduct fees
print(f"Balance After Deducting Fees: ${checking.balance:.2f}")
```

Explanation of the Code:

1. **Base Class (`BankAccount`)**:
    - The `BankAccount` class contains two attributes:
        - `account_number`: A unique number identifying the bank account.
        - `balance`: The current balance of the bank account.
    - The method `display_account_details()` displays the `account_number` and `balance`.
2. **Derived Class (`SavingsAccount`)**:
    - The `SavingsAccount` class inherits from `BankAccount` and adds an additional attribute:
        - `interest_rate`: The annual interest rate applied to the savings account.
    - The `calculate_interest()` method calculates and returns the interest earned based on the balance and the interest rate (`balance * interest_rate / 100`).
    - The `display_account_details()` method calls the parent class method to display common account details, and then displays the specific `interest_rate` for the savings account.

3. **Derived Class (`CheckingAccount`)**:
    - The `CheckingAccount` class inherits from `BankAccount` and adds an additional attribute:
        - `monthly_fee`: The monthly fee charged to the checking account.
    - The `deduct_fees()` method deducts the monthly fee from the account's balance. If the balance becomes negative after the fee is deducted, the balance is set to zero.
    - The `display_account_details()` method calls the parent class method to display common account details, and then displays the specific `monthly_fee` for the checking account.
4. **Creating Objects**:
    - We create an instance of `SavingsAccount` with an account number "12345", an initial balance of 1000, and an interest rate of 2%.
    - We create an instance of `CheckingAccount` with an account number "67890", an initial balance of 500, and a monthly fee of 10.
    - We call `display_account_details()` for both accounts to print their details.
    - For the `SavingsAccount`, we calculate the interest using `calculate_interest()` and display it.
    - For the `CheckingAccount`, we call `deduct_fees()` to subtract the monthly fee from the balance and display the updated balance.

Output:

```
Savings Account Details:
Account Number: 12345
Balance: $1000.00
Interest Rate: 2%
Interest Earned: $20.00

Checking Account Details (Before Fees):
Account Number: 67890
Balance: $500.00
Monthly Fee: $10.00
Balance After Deducting Fees: $490.00
```

Summary:

- The **`BankAccount`** class serves as the base class with shared attributes (`account_number` and `balance`) and a method to display these details.
- The **`SavingsAccount`** class extends the base class by adding an `interest_rate` attribute and a method (`calculate_interest()`) to compute the interest earned based on the balance.
- The **`CheckingAccount`** class extends the base class by adding a `monthly_fee` attribute and a method (`deduct_fees()`) to subtract the monthly fee from the balance.
- We use inheritance to avoid code duplication, allowing both `SavingsAccount` and `CheckingAccount` to share common attributes and methods from the `BankAccount` class, while also adding their own specific attributes and behaviors.

This design demonstrates how inheritance allows derived classes to build on the functionality of a base class while introducing specialized behaviors.