**# 1.Inheritance:**

**# Scenario:**

# Imagine that you're building a software system for a car dealership.

You need to track all the cars that are sold, and you also need to keep track of which cars are in inventory.

You decide to use object-oriented programming to model the cars as objects, with different properties and methods depending on whether they're sold or in inventory.

1. Car.py
2. SoldCar.py
3. InventoryCar.py

# first, let's defines a base class called Car that contains the common properties and methods that all cars have:

# This Car class has an \_\_init\_\_ method that takes in the make, model, and year of the car, and initializes those values as instance variables.

# next, let's defines a SoldCar class that inherits from the Car class.

This class will represent cars that have been sold:

This SoldCar class also has an \_\_init\_\_ method that takes in the make, model, year, and sale price of the car, and initializes those values as instance variables. Notice that we call the \_\_init\_\_ method of the Car class using the super() function. This is because we want to reuse the initialization logic of the Car class in the SoldCar class.

# now let's defines an InventoryCar class that also inherits from the Car class.

This class will represent cars that are still in inventory:

# The InventoryCar class is very similar to the SoldCar class, but with a few The InventoryCar class is very similar to the SoldCar class, but with a few enhancements:

# In this example, we create two car objects: car1 is a SoldCar object representing a 2018 Toyota Camry that was sold for $15,000, and car2 is an InventoryCar object representing a 2020 Honda Accord that was purchased for $20,000 and is still in inventory.

# We then call the get\_description method on each object, which returns a string describing the car, including whether it is sold or in inventory.

# Finally, we call the get\_profit method on each object, passing in the purchase price or sale price of the car as an argument. This method calculates and returns the profit that the dealership made or will make on the sale of the car.

# This example shows how inheritance can be used to model real-world objects with different properties and methods, while still reusing common logic defined in a base class.

**# 2.Encapsulation:**

**# Real-Time Scenario: A library management system**

# in a library management system, there are various entities such as books, authors, and borrowers.

Each entity has different attributes, methods, and behaviors.

Let's create a simple library management system that illustrates the concept of abstraction.

1. Book.py
2. Borrower.py
3. Library.py

# In this program, we have defined three classes: Book, Borrower, and Library. The Book and Borrower classes represent the entities of books and borrowers, respectively, while the Library class is the main class that manages the books and borrowers.

# In conclusion, abstraction is an important concept in object-oriented programming that allows us to create programs that are easy to use and maintain. By hiding unnecessary complexity from the user and focusing on essential details, we can create programs that are more efficient, scalable, and robust.

**# 3.Encapsulation:**

**# Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to the practice of hiding the internal details of an object and only exposing the necessary information through well-defined interfaces. In Python, encapsulation can be achieved using access modifiers.**

**# lets take a real-time scenario of a bank account class to demonstrate encapsulation.**

In this scenario, we will create a bank account class that has two attributes –

account number and balance.

We will also define two methods - deposit and withdraw - to update the balance.

1. BankAccount.py

# As we can see, encapsulation allows us to hide the internal details of the BankAccount class and only expose the necessary information through well-defined interfaces.

This makes the code more secure and easier to maintain.

**# 4.Polymorphism:**

**# Polymorphism is a concept in object-oriented programming that allows objects of different types to be treated as if they are of the same type. In Python, polymorphism can be achieved through method overriding and method overloading.**

**# Let's take a real-time scenario to understand polymorphism.**

**Consider a scenario where we have different types of animals such as dogs, cats, and birds, and each animal can make a sound.**

**We can implement this scenario using polymorphism as follows:**

1. Animal.py
2. Dog.py
3. Cat.py
4. Bird.py

# In the above code, we have a parent class Animal with a method make\_sound() that is overridden by the child classes Dog, Cat, and Bird. Each child class has its own implementation of the make\_sound() method, which allows objects of different types to be treated as if they are of the same type.

# When we create objects of different animals and call the make\_sound() method, the appropriate implementation of the method is executed based on the type of object. For example, when we call dog.make\_sound(), the make\_sound() method of the Dog class is executed, which prints "Tommy says woof woof!" to the console.

# This is an example of run-time polymorphism, where the appropriate method implementation is determined at runtime based on the type of the object. The Animal class provides a common interface for all the child classes, allowing us to treat objects of different types as if they are of the same type.

**# 5.Method Overloading:**

**# # Method Overloading in Python is a concept where multiple methods in a class have the same name but different parameters. It is a form of polymorphism that allows a class to have multiple methods with the same name but different signature (parameter types, number of parameters, or order of parameters).**

**# A real-time scenario where method overloading could be used is in a banking application where different types of accounts have different requirements for opening an account. For example, a savings account requires a minimum deposit, while a current account requires a letter of credit from a reputable organization.**

1. BankAccount.py

# In this implementation, the BankAccount class has an \_\_init\_\_ method that takes an account number and an optional balance parameter. The default value of the balance parameter is 0.

# The class also has three other methods: deposit, withdraw, and get\_balance. The deposit method takes an amount parameter and adds it to the account's balance if the amount is greater than zero. If the amount is zero or negative, the method prints an error message.

# The withdraw method takes an amount parameter and subtracts it from the account's balance if there are sufficient funds and the amount is greater than zero. If there are insufficient funds, the method prints an error message. If the amount is zero or negative, the method also prints an error message.

# The get\_balance method simply prints the current balance of the account.

**# 6.Method Overriding:**

**# Method Overriding is a feature of object-oriented programming (OOP) that allows a subclass to provide its own implementation of a method that is already defined in its superclass. In this way, the subclass can modify or extend the behavior of the method inherited from the superclass.**

**# Let's take a real-time scenario to explain Method Overriding. Consider the following code snippet:**

1. **Animal.py**
2. **Dog.py**
3. **Cat.py**

# In this example, we have defined a class Animal that has a method make\_sound which simply prints the name of the animal and the sound it makes.

We then define two subclasses Dog and Cat which inherit from Animal and each of them provides their own implementation of the make\_sound method.

# As we can see, the make\_sound method of the Animal class is overridden by the make\_sound methods of the Dog and Cat subclasses.

# In summary, Method Overriding is a powerful feature of OOP that allows subclasses to modify or extend the behavior of a method inherited from a superclass.

This enables us to write more flexible and maintainable code that can be customized for specific use cases.