Introduction to OOP in Python:

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into "objects," which are instances of classes. These classes define the properties (attributes) and behaviors (methods) of the objects. Python is an object-oriented programming language, meaning it fully supports the principles of OOP.

The four key principles of OOP are:

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

Let's go through each of these principles in detail with examples and real-world scenarios.

---

1. Encapsulation

Definition: Encapsulation is the concept of bundling the data (attributes) and the methods (functions) that operate on the data into a single unit or class. It also restricts direct access to some of the object's components, which is usually done by making certain attributes or methods private.

- **Private attributes** are variables that cannot be accessed directly from outside the class.
- **Public attributes** are variables that can be accessed directly.

Example of Encapsulation in Python:

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner         # Public attribute
        self.__balance = balance    # Private attribute (can't be accessed
directly)

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > 0 and self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Insufficient funds or invalid amount.")
    def get_balance(self):
        return  self.__balance    #  Getter  method  to  access  the  private
attribute

# Creating an object
account = BankAccount("Alice", 1000)

# Accessing public attribute
print(account.owner)  # Output: Alice

# Accessing private attribute directly would raise an error
# print(account.__balance)  # AttributeError

# Using getter to access private attribute
print(account.get_balance())  # Output: 1000

# Performing deposit and withdrawal
account.deposit(500)
account.withdraw(300)
print(account.get_balance())  # Output: 1200
```

Scenario:
In a bank system, you might want to encapsulate the account balance because you don't want users to directly change it. The `deposit` and `withdraw` methods are the only ways to update the balance, ensuring that business logic (like preventing withdrawals that exceed the balance) is respected.

---

2. Inheritance

Definition: Inheritance allows one class (child class) to inherit the attributes and methods from another class (parent class). This helps in code reusability and establishing a relationship between the parent and child classes.

Example of Inheritance in Python:

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

    def speak(self):
        pass  # Base method, to be overridden in child classes

class Dog(Animal):
    def speak(self):
        return f"{self.name} barks."

class Cat(Animal):
    def speak(self):
        return f"{self.name} meows."
```

```
# Creating objects of the child classes
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Calling the overridden methods
print(dog.speak())  # Output: Buddy barks.
print(cat.speak())  # Output: Whiskers meows.
```

Scenario:
In a zoo management system, you can create an `Animal` class with common attributes and methods (like `name` and `speak`). Specific animals like `Dog` and `Cat` can then inherit from `Animal` and have their own `speak` method, which is overridden to match the sound each animal makes.

---

3. Polymorphism

Definition: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It also allows methods to have the same name but behave differently based on the object that calls them. This is most commonly achieved through method overriding (as seen in inheritance).

Example of Polymorphism in Python:

```
class Bird:
    def fly(self):
        print("Birds can fly.")

class Airplane:
    def fly(self):
        print("Airplanes can fly.")

# Both classes have the same method name "fly"
def make_it_fly(obj):
    obj.fly()

# Creating objects
bird = Bird()
airplane = Airplane()

# Polymorphism: The same method name, but different behavior based on object
type
make_it_fly(bird)      # Output: Birds can fly.
make_it_fly(airplane)  # Output: Airplanes can fly.
```

Scenario:
In a transportation simulation system, you can have different objects like `Bird` and `Airplane`. Both can `fly`, but the implementation of `fly` is different for each. Polymorphism allows you to treat both as objects that can `fly`, even though the behavior differs.

---

4. Abstraction

Definition: Abstraction is the process of hiding the complex implementation details and showing only the essential features. In Python, abstraction is achieved using abstract classes and methods. An abstract class cannot be instantiated directly and must be subclassed. Abstract methods are defined in the parent class but must be implemented by the child class.

Example of Abstraction in Python:

```
from abc import ABC, abstractmethod

class Shape(ABC):  # Abstract base class
    @abstractmethod
    def area(self):
        pass  # Abstract method

    @abstractmethod
    def perimeter(self):
        pass  # Abstract method

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

    def area(self):
        return 3.14 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14 * self.radius

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

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

    def perimeter(self):
        return 2 * (self.width + self.height)

# Creating objects of concrete classes
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Accessing methods
print(circle.area())      # Output: 78.5
print(rectangle.area())   # Output: 24
```

Scenario:
In a drawing application, you can define a `Shape` abstract class with abstract methods for calculating the area and perimeter. Concrete classes like `Circle` and `Rectangle` will implement

these methods, but the user of the system doesn't need to worry about the details—just call the `area()` method and get the result.

Real-World Scenario: Building a Simple Inventory Management System

Let's put all the concepts together by creating a basic inventory management system using OOP principles.

Scenario Breakdown:

1. We have products in the inventory (represented by objects).
2. Different types of products (e.g., `Electronics`, `Groceries`) can inherit common features like `product_name`, `price`, etc.
3. The system should allow adding, removing, and displaying products in the inventory, hiding the internal details of inventory management.

```
from abc import ABC, abstractmethod

# Abstract class for common product functionality
class Product(ABC):
    def __init__(self, name, price):
        self.name = name
        self.price = price

    @abstractmethod
    def display(self):
        pass

# Concrete class for Electronics
class Electronics(Product):
    def __init__(self, name, price, warranty_years):
        super().__init__(name, price)
        self.warranty_years = warranty_years

    def display(self):
        print(f"Electronics   Product:   {self.name},   Price:   {self.price},
Warranty: {self.warranty_years} years")

# Concrete class for Groceries
class Groceries(Product):
    def __init__(self, name, price, expiration_date):
        super().__init__(name, price)
        self.expiration_date = expiration_date

    def display(self):
        print(f"Groceries  Product:  {self.name},  Price:  {self.price},  Expiry
Date: {self.expiration_date}")

# Inventory management class
class Inventory:
    def __init__(self):
        self.products = []

    def add_product(self, product):
        self.products.append(product)

    def remove_product(self, product_name):
        self.products = [p for p in self.products if p.name != product_name]

    def display_inventory(self):
        for product in self.products:
            product.display()

# Creating products and adding to inventory
inventory = Inventory()

laptop = Electronics("Laptop", 1200, 2)
apple = Groceries("Apple", 3, "2025-12-31")

inventory.add_product(laptop)
inventory.add_product(apple)

# Displaying the inventory
inventory.display_inventory()

# Removing a product
inventory.remove_product("Apple")

# Displaying the updated inventory
inventory.display_inventory()
```

Output:

```
Electronics Product: Laptop, Price: 1200, Warranty: 2 years
Groceries Product: Apple, Price: 3, Expiry Date: 2025-12-31
Electronics Product: Laptop, Price: 1200, Warranty: 2 years
```

Explanation:

1. **Inheritance:** `Electronics` and `Groceries` inherit from `Product` and have their own implementation of the `display` method.
2. **Polymorphism:** Both `Electronics` and `Groceries` share the `display` method, but it behaves differently for each.
3. **Encapsulation:** `Inventory` manages the list of products internally and exposes methods like `add_product` and `remove_product` to interact with it.
4. **Abstraction:** The user interacts with the `Product` objects without needing to understand the details of how inventory management is implemented.