1. Introduction to Object-Oriented Programming

Definition and Benefits Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" – data structures consisting of fields, often known as attributes; and methods, which are procedures associated with the objects. OOP languages are designed to facilitate software development and maintenance by providing ways to structure programs as collections of objects that can be packaged and reused in different parts of the program.

Real-World Analogies Imagine a car. A car has attributes like color, brand, and model, and methods like start, stop, and drive. In OOP terms, the car is an object, the attributes are the data, and the methods are the functions.

2. Core Concepts of OOP

Classes and Objects

- **Class**: A blueprint for creating objects. Defines a type of object according to the attributes and methods.
- **Object**: An instance of a class.

Example: Basic Class and Object

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start(self):
        print(f"{self.brand} {self.model} is starting.")

# Creating an object of the Car class
my_car = Car("Toyota", "Corolla")
my_car.start()  # Output: Toyota Corolla is starting.
```

Attributes and Methods

- **Attributes**: Variables that belong to an object or class.
- **Methods**: Functions that belong to an object or class.

Encapsulation Encapsulation is the bundling of data with the methods that operate on that data. It restricts direct access to some of the object's components, which can prevent the accidental modification of data.

Example: Encapsulation

```
class Car:
    def __init__(self, brand, model):
        self.__brand = brand
        self.__model = model

    def get_brand(self):
        return self.__brand

    def set_brand(self, brand):
        self.__brand = brand

# Creating an object
my_car = Car("Toyota", "Corolla")
print(my_car.get_brand())  # Output: Toyota
my_car.set_brand("Honda")
print(my_car.get_brand())  # Output: Honda
```

Inheritance Inheritance allows a class to inherit attributes and methods from another class.

Example: Inheritance

```
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start(self):
        print(f"{self.brand} {self.model} is starting.")

class Car(Vehicle):
    def drive(self):
        print(f"{self.brand} {self.model} is driving.")

# Creating an object of the Car class
my_car = Car("Toyota", "Corolla")
my_car.start()  # Output: Toyota Corolla is starting.
my_car.drive()  # Output: Toyota Corolla is driving.
```

Polymorphism Polymorphism allows methods to do different things based on the object it is acting upon, even though they share the same name.

Example: Polymorphism

```
class Bird:
    def move(self):
        print("I am flying")

class Fish:
    def move(self):
        print("I am swimming")

# Polymorphic behavior
def make_move(animal):
    animal.move()

bird = Bird()
fish = Fish()

make_move(bird)  # Output: I am flying
make_move(fish)  # Output: I am swimming
```

Abstraction Abstraction means hiding the complex implementation details and showing only the necessary features of the object.

Example: Abstraction

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        pass

class Bird(Animal):
    def move(self):
        print("I am flying")
```

```
class Fish(Animal):
    def move(self):
        print("I am swimming")

# Abstract class cannot be instantiated
# animal = Animal()  # This will raise an error

bird = Bird()
fish = Fish()

bird.move()  # Output: I am flying
fish.move()  # Output: I am swimming
```

3. Detailed Explanations and Examples

Class and Object

- A class is a blueprint for creating objects.
- An object is an instance of a class.

Example:

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age}
years old.")

# Creating an object of the Person class
person1 = Person("Alice", 30)
person1.greet()  # Output: Hello, my name is Alice and I am 30 years
old.
```

Encapsulation

- Encapsulation protects object integrity by preventing unintended or harmful modifications.
- Use of private attributes (conventionally by prefixing with double underscore).

Example:

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number
        self.__balance = balance

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient balance")

    def get_balance(self):
        return self.__balance

# Creating an object of the BankAccount class
account = BankAccount("123456789", 1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
account.withdraw(2000)  # Output: Insufficient balance
print(account.get_balance())  # Output: 1500
```

Inheritance

- Allows the creation of a new class based on an existing class.
- The new class (child class) inherits attributes and methods of the existing class (parent class).

Example:

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

    def sound(self):
        pass

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

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

# Creating objects of Dog and Cat classes
dog = Dog("Buddy")
cat = Cat("Kitty")

dog.sound()  # Output: Bark
cat.sound()  # Output: Meow
```

Polymorphism

- The ability to present the same interface for different underlying forms (data types).

Example:

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

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side
```

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

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

# Using polymorphism
def print_area(shape):
    print(shape.area())

square = Square(4)
circle = Circle(3)

print_area(square)  # Output: 16
print_area(circle)  # Output: 28.26
```

Abstraction

- The concept of hiding the complex implementation details and showing only the essential features of the object.

Example:

```
from abc import ABC, abstractmethod

class Appliance(ABC):
    @abstractmethod
    def turn_on(self):
        pass

class WashingMachine(Appliance):
    def turn_on(self):
        print("Washing machine is now on")

class Refrigerator(Appliance):
    def turn_on(self):
        print("Refrigerator is now on")
```

```
# Cannot instantiate abstract class
# appliance = Appliance()  # This will raise an error

washing_machine = WashingMachine()
refrigerator = Refrigerator()

washing_machine.turn_on()  # Output: Washing machine is now on
refrigerator.turn_on()  # Output: Refrigerator is now on
```

4. Practical Exercises

Exercise 1: Create a Simple Class and Object

1. **Create a `Book` class** with attributes like title, author, and pages.
2. **Instantiate the `Book` class** and print the book details.

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def get_info(self):
        return f"'{self.title}' by {self.author}, {self.pages} pages"

# Create an instance of Book
book = Book("1984", "George Orwell", 328)
print(book.get_info())  # Output: '1984' by George Orwell, 328 pages
```

Exercise 2: Implement Inheritance

1. **Create a base class `Employee`** with attributes name and salary.
2. **Create a derived class `Manager`** that inherits from `Employee` and adds an additional attribute department.

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

```
        self.salary = salary

class Manager(Employee):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department

    def get_details(self):
        return f"Manager {self.name}, Department: {self.department},
Salary: {self.salary}"

# Create an instance of Manager
manager = Manager("Alice", 90000, "HR")
print(manager.get_details())  # Output: Manager Alice, Department: HR,
Salary: 90000
```

Exercise 3: Polymorphism with a Function

1. **Create a base class `Shape`** with a method `area`.
2. **Create derived classes `Rectangle` and `Circle`** each with their own implementation of the `area` method.
3. **Write a function that takes a `Shape` object** and prints its area.

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

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

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

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

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

# Polymorphic function
def print_area(shape):
    print(shape.area())

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

print_area(rectangle)  # Output: 20
print_area(circle)     # Output: 28.26
```

Exercise 4: Develop a Simple Application (Zoo Management System)

1. **Create a base class `Animal`** with attributes and methods.
2. **Create derived classes for different animals (e.g., `Lion`, `Elephant`)**.
3. **Create a `Zoo` class** to manage the collection of animals.
4. **Implement methods to add animals to the zoo and display all animals**.

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

    def __str__(self):
        return f"{self.name} the {self.species}"

class Lion(Animal):
    def __init__(self, name):
        super().__init__(name, "Lion")

class Elephant(Animal):
    def __init__(self, name):
        super().__init__(name, "Elephant")

class Zoo:
    def __init__(self):
```

```
        self.animals = []

    def add_animal(self, animal):
        self.animals.append(animal)

    def show_animals(self):
        for animal in self.animals:
            print(animal)

# Create instances of animals and zoo
zoo = Zoo()
lion = Lion("Simba")
elephant = Elephant("Dumbo")

zoo.add_animal(lion)
zoo.add_animal(elephant)
zoo.show_animals()
```