Lab 2: Python Basics Part II

Overview:

In this lab session, we will delve into various important concepts in Python programming, including error handling, file handling, functional programming, and object-oriented programming (OOP) principles. The lab has the following two main sections:

1. Section I: Demonstration of the aforementioned topics using sample codes. In addition, simple practice exercises are provided to obtain hands on these concepts.
2. Section II: Lab task related to file handling

By the end of this lab, you will have a solid understanding of these topics and how they are applied in Python.

Topics Covered:

Errors and Exceptions: Learn about handling errors and exceptions gracefully in Python programs to ensure smooth execution and better user experience.

File Handling: Understand how to work with files in Python, including creating, reading, writing, and manipulating file contents using various file handling techniques.

Functional Programming: Explore the functional programming paradigm in Python, including pure functions, recursion, map and filter functions, and list comprehensions, to write clean, concise, and efficient code.

OOP Principles: Gain insight into object-oriented programming principles in Python, including encapsulation, abstraction, inheritance, and polymorphism, to create modular, reusable, and maintainable code.

Prerequisites:

Before starting this lab, ensure that you have a basic understanding of Python programming concepts, including variables, data types, control structures, and functions.

Lab Format:

Each topic will be covered through a series of explanations, sample code demonstrations, and practice exercises. You will have the opportunity to apply what you've learned through hands-on coding exercises.

Lab Environment:

You can use any Python IDE or text editor of your choice to write and execute Python code. Additionally, you may utilize online Python interpreters or environments if you do not have Python installed locally.

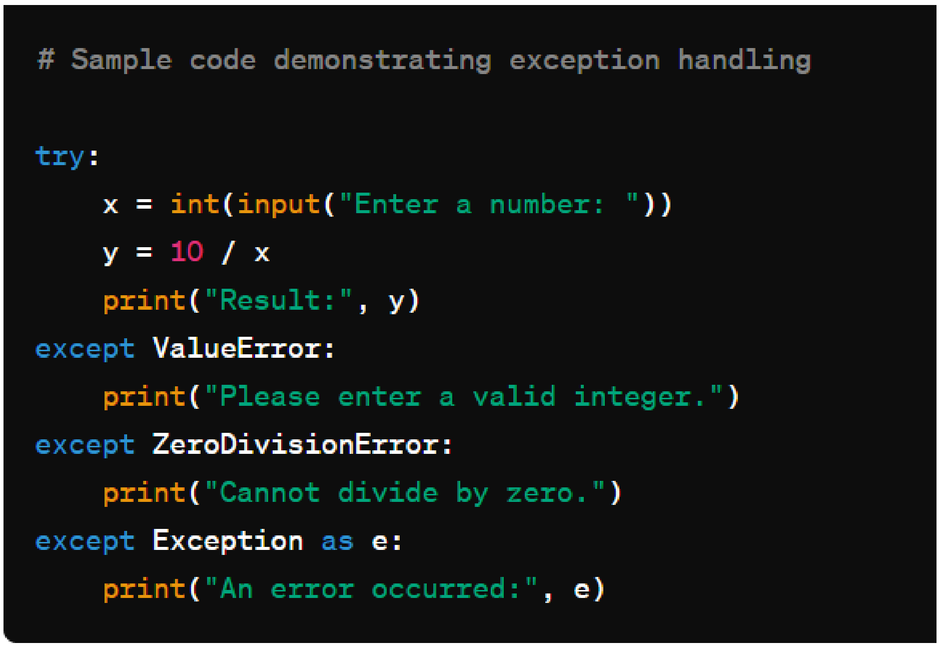
Section I: Exceptions, File Handling, Functional Programming, and OOP Concepts

1: Errors and Exceptions

1.1. Description:

In this section, you will learn about exception handling in Python. You will explore how to handle errors and exceptions gracefully in your Python programs.

1.2. Sample Code:



```python
# Sample code demonstrating exception handling

try:
    x = int(input("Enter a number: "))
    y = 10 / x
    print("Result:", y)
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
except Exception as e:
    print("An error occurred:", e)
```

1.3. Practice Exercise:

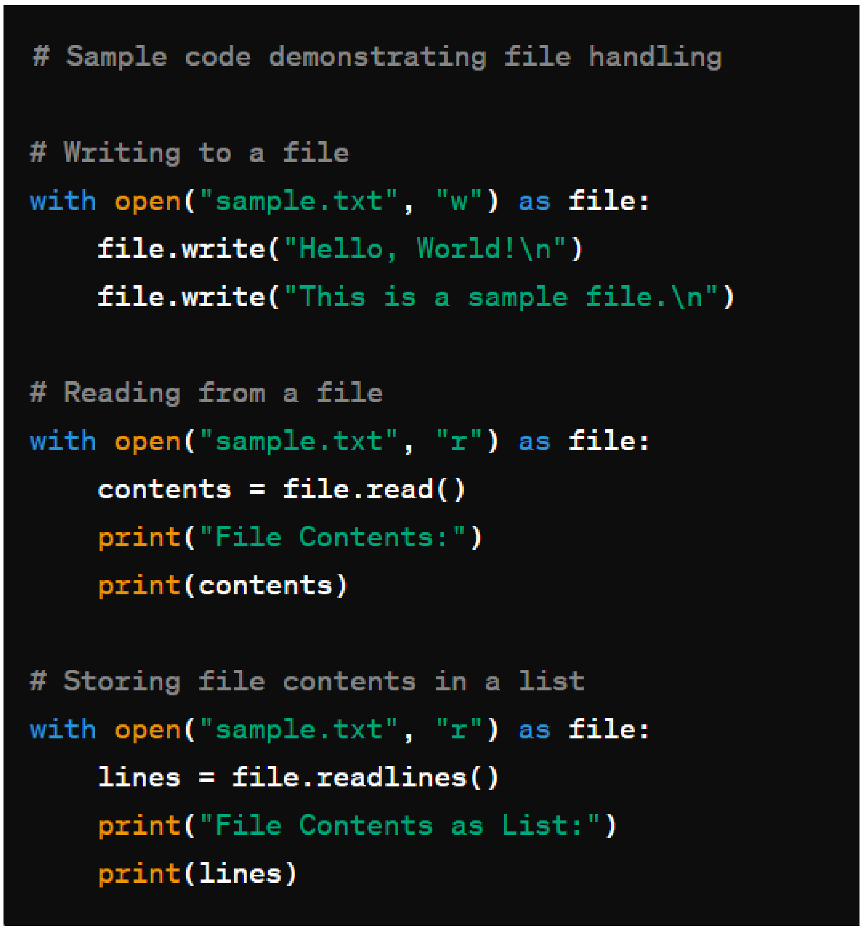
1. Write a program that asks the user to enter two numbers and performs division. Handle the ZeroDivisionError and ValueError exceptions.
2. Modify the sample code to include a custom exception called CustomError. Raise this exception when the user enters a negative number.

2: File Handling

2.1. Description:

This section introduces you to file handling in Python. You will learn how to create, read, and manipulate files, as well as how to store file contents in data structures.

2.2. Sample Code:



```python
# Sample code demonstrating file handling

# Writing to a file
with open("sample.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a sample file.\n")

# Reading from a file
with open("sample.txt", "r") as file:
    contents = file.read()
    print("File Contents:")
    print(contents)

# Storing file contents in a list
with open("sample.txt", "r") as file:
    lines = file.readlines()
    print("File Contents as List:")
    print(lines)
```

2.3. Practice Exercise:

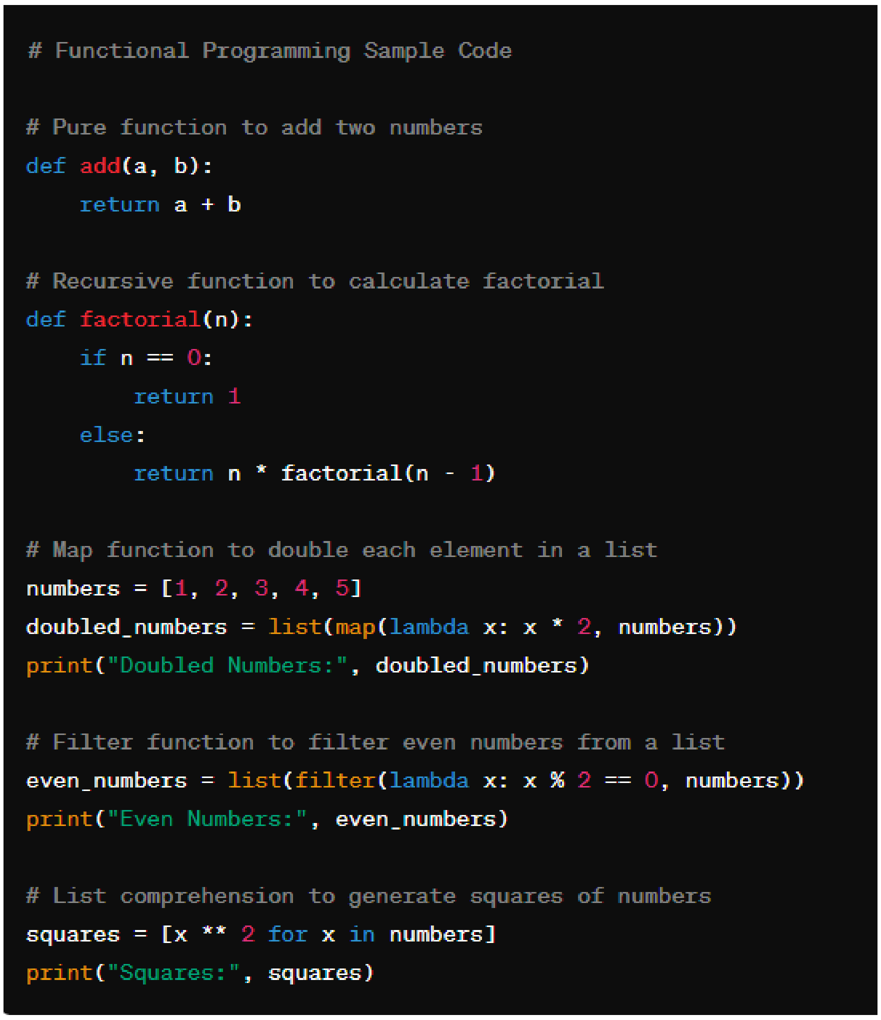
1. Write a program that reads a text file (data.txt) line by line and prints each line.
2. Modify the sample code to append additional text to the existing file instead of overwriting it.

3: Functional Programming

3.1. Description:

In this section, you will explore functional programming concepts in Python, including pure functions, recursion, map and filter functions, and comprehensions.

3.2. Syntax and Sample Code:



```python
# Functional Programming Sample Code

# Pure function to add two numbers
def add(a, b):
    return a + b

# Recursive function to calculate factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# Map function to double each element in a list
numbers = [1, 2, 3, 4, 5]
doubled_numbers = list(map(lambda x: x * 2, numbers))
print("Doubled Numbers:", doubled_numbers)

# Filter function to filter even numbers from a list
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print("Even Numbers:", even_numbers)

# List comprehension to generate squares of numbers
squares = [x ** 2 for x in numbers]
print("Squares:", squares)
```

4. OOP Principles:

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data in the form of fields (attributes) and code in the form of procedures (methods). The four main principles of OOP are:

Encapsulation: Encapsulation refers to the bundling of data (attributes) and methods that operate on that data into a single unit, i.e., an object. This helps in hiding the internal state of an object and only exposing necessary functionalities.

Abstraction: Abstraction is the concept of simplifying complex systems by modeling classes based on real-world entities. It involves focusing only on the relevant attributes and behaviors of objects while hiding unnecessary details.

Inheritance: Inheritance allows a class (child class) to inherit attributes and methods from another class (parent class). This promotes code reuse and facilitates the creation of hierarchical relationships between classes.

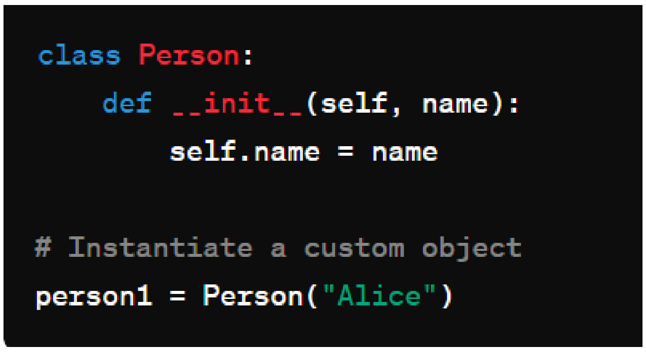
Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different types of objects and allows for more flexible and modular code.

4.1. Python Classes and Instances:

In Python, a class is a blueprint for creating objects. It defines the attributes and methods that will be associated with the objects created from that class. An instance, also known as an object, is a specific realization of a class.

4.2. Instantiate a Custom Object:

Instantiating a custom object involves creating an instance of a class. This is done by calling the class name followed by parentheses. For example:



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

# Instantiate a custom object
person1 = Person("Alice")
```

4.3. Instance Methods:

Instance methods are functions defined within a class that operate on instance attributes. They are called on instances of the class and have access to the instance itself via the self parameter.

4.4. Parent Classes vs Child Classes:

In Python, classes can inherit attributes and methods from other classes. A parent class (also known as a base class or superclass) is the class from which attributes and methods are inherited. A child class (also known as a derived class or subclass) is a class that inherits from another class.

4.5. Inheritance and Multiple Inheritance:

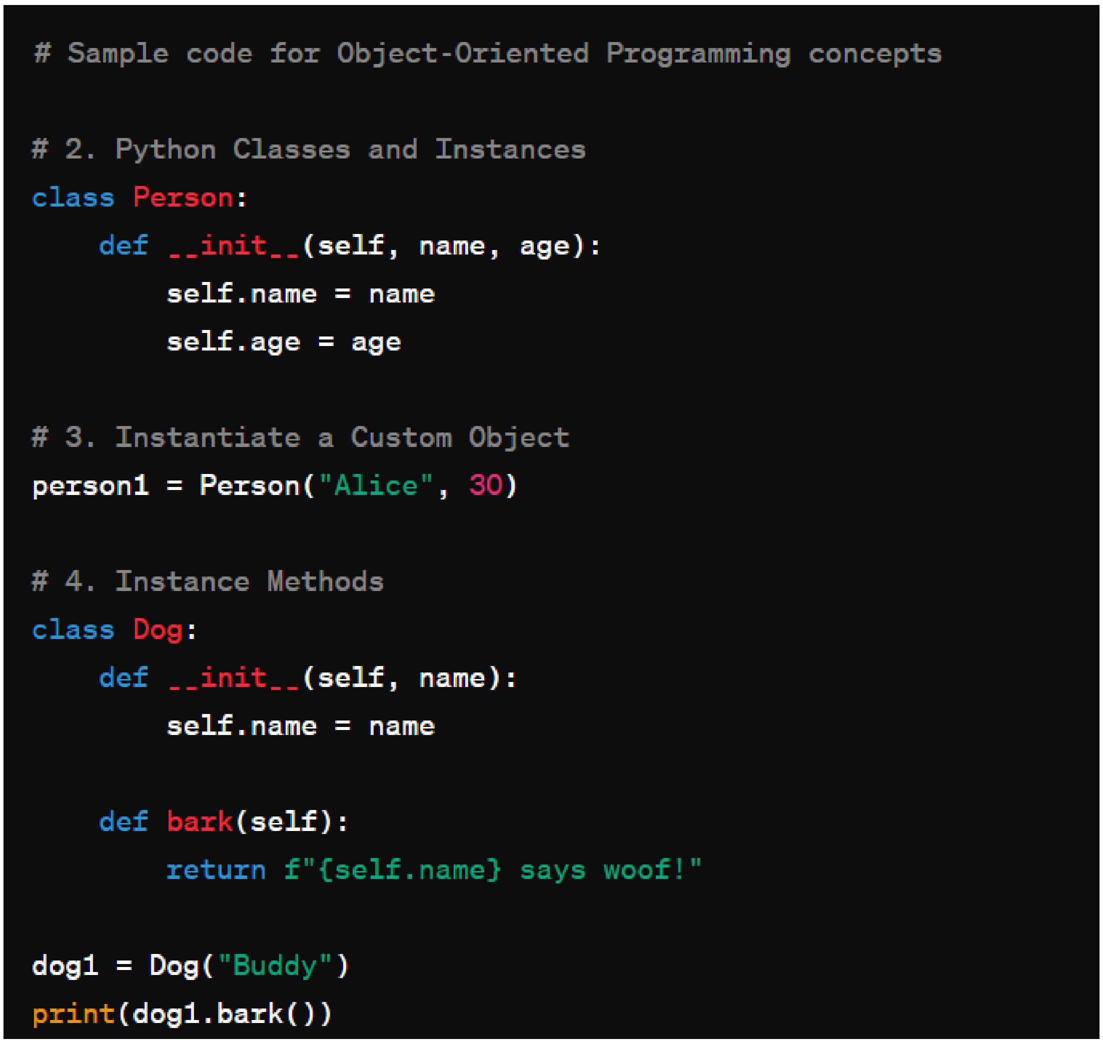
Inheritance is the mechanism by which a class can inherit attributes and methods from another class. Python supports multiple inheritance, where a class can inherit from multiple parent classes. This allows for the creation of complex class hierarchies and promotes code reuse.

4.6. Abstract Classes and Methods:

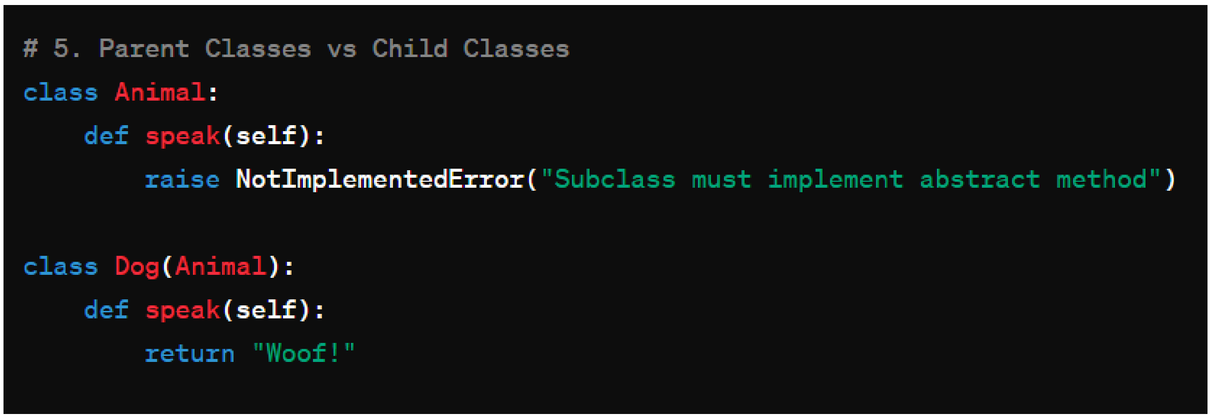
Abstract classes are classes that cannot be instantiated and are meant to be subclassed. They may contain one or more abstract methods, which are methods declared in the abstract class but not implemented. Subclasses of abstract classes must implement all abstract methods defined in the parent class.

4.7. Method Resolution Order (MRO):

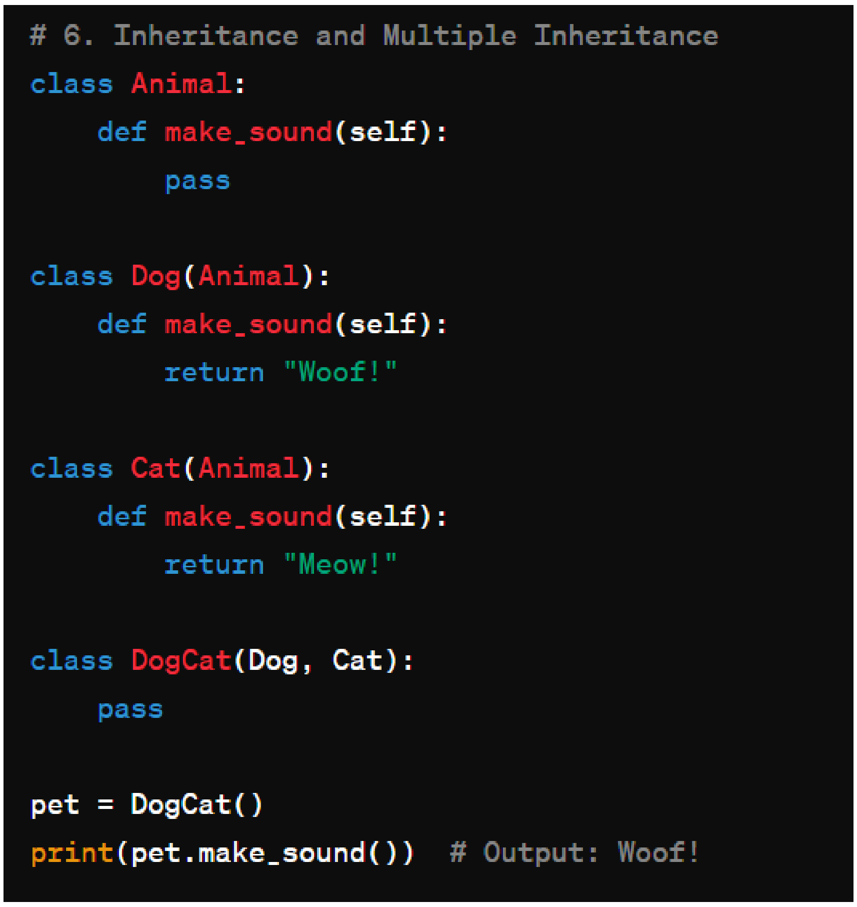
Method Resolution Order (MRO) is the order in which Python looks for methods and attributes in a class hierarchy. It follows a depth-first, left-to-right traversal of the class hierarchy. The MRO can be accessed using the __mro__ attribute or the mro() method.



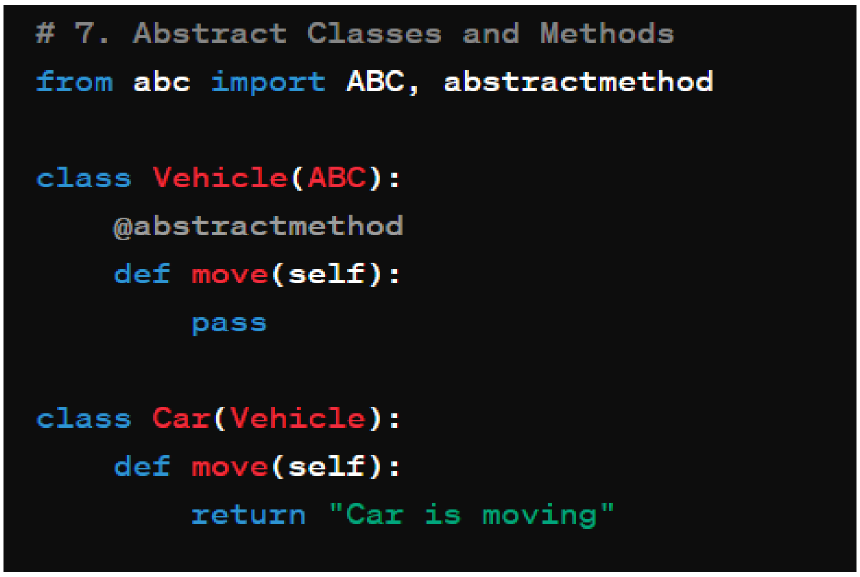
```python
# Sample code for Object-Oriented Programming concepts

# 2. Python Classes and Instances
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# 3. Instantiate a Custom Object
person1 = Person("Alice", 30)

# 4. Instance Methods
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        return f"{self.name} says woof!"

dog1 = Dog("Buddy")
print(dog1.bark())
```



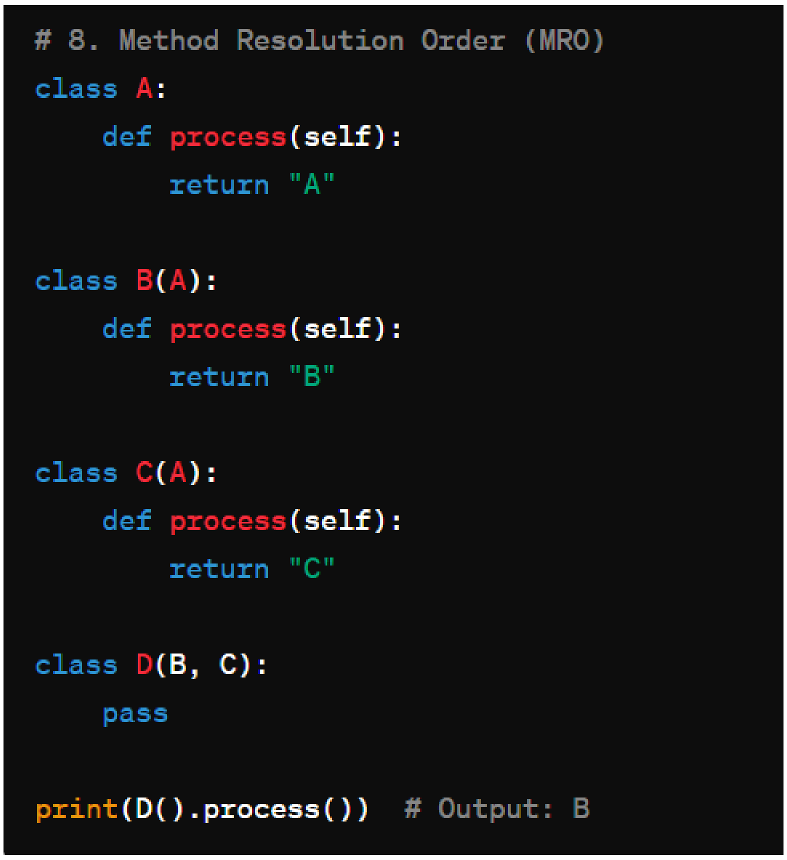
```python
# 5. Parent Classes vs Child Classes
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):
    def speak(self):
        return "Woof!"
```



```python
# 6. Inheritance and Multiple Inheritance
class Animal:
    def make_sound(self):
        pass

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

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

class DogCat(Dog, Cat):
    pass

pet = DogCat()
print(pet.make_sound())  # Output: Woof!
```



```python
# 7. Abstract Classes and Methods
from abc import ABC, abstractmethod

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

class Car(Vehicle):
    def move(self):
        return "Car is moving"
```



```python
# 8. Method Resolution Order (MRO)
class A:
    def process(self):
        return "A"

class B(A):
    def process(self):
        return "B"

class C(A):
    def process(self):
        return "C"

class D(B, C):
    pass

print(D().process())  # Output: B
```

Practice Exercise:

1. Create a class Rectangle with attributes length and width, and a method area to calculate its area.
2. Create a class Square that inherits from Rectangle. Override the area method to calculate the area of a square.
3. Define an abstract class Shape with an abstract method area. Implement concrete classes Circle and Triangle inheriting from Shape and providing implementations for the area method.

Section II:

Lab 2 Task Read in data, store, manipulate and output new data to a file

In this task you'll read the contents of a file and then write the contents to another file.

You'll store the contents of a file into a list so that it can be accessed in different ways.

Objectives:

There are two objectives of this activity:

1. Create a function for reading in a file
2. Create a function for writing files.

Exercise Instructions:

1. Check that the `sampletext.txt` and `file_ops.py` files exist and are present inside the project folder.You can run the `file_ops.py` file by opening a terminal and executing the following command:

```
python3 file_ops.py
```

2. Complete the `read_file()` function to read in the sampletext.txt file using the `open` function and return the entire contents of the file.

3. Complete the `read_file_into_line()` function so that it returns a data structure of all the contents of the file in a line-by-line sequential order.

4. Fill in the `write_first_line_to_file()` that accepts two arguments. This should write only the first line of the file contents into the given output file.

- **Argument 1:** The contents of a file to be written

- **Argument 2:** The name of an output file.

5. Complete the `read_even_numbered_lines()` to return a list of the even-numbered lines of a file (2, 4, 6, etc.)

6. Fill in the `read_file_in_reverse()` function to return a list of the lines of a file in reverse order.