Notebook For Learning Python Programming Language

Module 3

Classes and Objects

Classes and objects are fundamental concepts in object-oriented programming (OOP). Classes are used to create user-defined data types that can contain attributes (data) and methods (functions) for manipulating the data. An object is an instance of a class.

Here's an example to explain classes and objects in Python:

In [ ]:
```
class Car:
  def __init__(self, make, model, year):
    self.make = make
    self.model = model
    self.year = year

  def start(self):
    print(f"Starting the {self.make} {self.model}")

car = Car("Toyota", "Camry", 2020)
print(car.make)   # Output: Toyota
print(car.model) # Output: Camry
print(car.year)   # Output: 2020
car.start()       # Output: Starting the Toyota Camry
```

In the example, we have defined a class `Car` with three attributes `make`, `model`, and `year`. The `__init__` method is the constructor of the class, it is called when an object is created from the class and it is used to initialize the attributes of the object. The class `Car` also has a method `start` that prints a message indicating that the car is starting.

We then create an object `car` from the class `Car` and assign values to its attributes. We can access the attributes of the object using the dot (.) operator and call the methods of the object using parentheses.

Another definition of Class and objects. </br> Classes and objects are fundamental concepts in Object-Oriented Programming (OOP), which is a programming paradigm based on the concept of objects.

A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

An object is an instance of a class. When you define a class, you define a blueprint for a data structure. This doesn’t actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Here's an example of how to define a simple class in Python:

In [ ]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Woof!")

dog = Dog("Max", 5)
print(dog.name)
print(dog.age)
dog.bark()
```

In this example, `Dog` is a class. The `__init__` method is a special method that is executed when an object of the class is created. The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The `bark` method is a behavior of the `Dog` class. When we create an instance of the `Dog` class, we can access the `bark` method using the dot (.) operator.

Inheritance and polymorphism

Inheritance and polymorphism are two important concepts in Object-Oriented Programming (OOP) in Python.

Inheritance is a mechanism that allows creating a new class based on an existing class. The new class inherits the properties and methods of the existing class. This allows you to create new classes that are derived from existing classes, thereby making it easier to reuse code.

Here's an example to demonstrate inheritance in Python:

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

    def __str__(self):
        return f'{self.name} is a {self.species}'

class Dog(Animal):
    def __init__(self, name, breed):
        Animal.__init__(self, name, species='Dog')
        self.breed = breed

    def __str__(self):
        return f'{self.name} is a {self.breed} {self.species}'

dog = Dog('Rocky', 'Labrador')
print(dog)
# Output: Rocky is a Labrador Dog
```

Polymorphism is a concept in OOP that allows an object to take on many forms. Polymorphism allows objects of different classes to respond to the same method call. In other words, the same method name can perform different actions based on the class of the object.

Here's an example to demonstrate polymorphism in Python:

In [ ]:
```
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!'

animals = [Dog(), Cat()]
for animal in animals:
    print(animal.make_sound())
# Output:
# Woof!
# Meow!
```

In this example, `Dog` and `Cat` classes inherit from the `Animal` class and override the `make_sound` method to return their respective sounds. In the loop, both `Dog` and `Cat` objects respond to the same method call `make_sound` and produce different results based on the class of the object.

Python Exception Handling

Python Exception Handling is a process of handling and catching errors or exceptions in a Python program. When a program encounters an error, an exception is raised, and the program execution stops. To prevent the program from stopping and to handle the exception, we can use the try-except block in Python.

The try-except block consists of two parts:

- The try block is used to enclose the code that may raise an exception.
- The except block is used to handle the exception if it occurs.

Here's a simple example of how to handle an exception in Python:

In [ ]:
```
try:
    # code that may raise an exception
    number = int(input("Enter a number: "))
    print(10/number)
except ZeroDivisionError:
    # code to handle the exception
    print("Cannot divide by zero")
```

In this example, the input from the user is converted to an integer and stored in the `number` variable. Then, we perform the division operation, which may raise a `ZeroDivisionError` exception if the user enters zero. The except block is used to catch the exception and display a message to the user, telling them that division by zero is not allowed.

By using exception handling, we can ensure that our program does not stop unexpectedly, and we can provide meaningful feedback to the user in case of an error.

Exercises

Here are a few exercise scripts for practicing classes and objects in Python:

1. Define a class named "Person" with attributes "name", "age", and "gender". Define a method called "introduction" which returns a string introducing the person. Create an instance of the class and call the

"introduction" method.

2. Define a class named "Rectangle" with attributes "length" and "width". Define methods "area" and "perimeter" which return the area and perimeter of the rectangle respectively. Create an instance of the class and call both methods.

3. Define a class named "Student" with attributes "name", "age", and "grades". Define a method "average_grade" which returns the average of the grades. Create an instance of the class and call the method.

4. Create a base class called `Animal` with a method called `make_sound()` that prints "Animal making noise". Then create two subclasses `Dog` and `Cat` that both inherit from `Animal`. Override the `make_sound()` method in both subclasses to print "Woof" for `Dog` and "Meow" for `Cat`.

5. Create a class `Rectangle` with methods to calculate the area and perimeter. Then create a class `Square` that inherits from `Rectangle`. Override the methods in `Square` to only use one side length as the width and height.

6. Create a class `Person` with attributes `name` and `age`. Then create a subclass `Student` that inherits from `Person`. Add an attribute `student_id` to the `Student` class. Override the `__str__` method in both classes to return a custom string representation of each class.

7. Write a function to divide two numbers. If the denominator is 0, raise an exception with the message "division by zero error".

8. Write a function to calculate the square root of a number. If the number is negative, raise an exception with the message "negative number error".

9. Write a function to read a file and count the number of lines in it. If the file does not exist, raise an exception with the message "file not found error".

In [ ]: