Object-Oriented Programming with python

Index

Introduction to Object Oriented Programming in Python

Object-Oriented programming is a widely used concept to write powerful applications. As a data scientist, you will be required to write applications to process your data, among a range of other things. OOP uses the concept of objects and classes. A class can be thought of as a 'blueprint' for objects. These can have their own attributes (characteristics they possess), and methods (actions they perform).

You will learn now:

- How to create a class ?
- Instantiating objects
- Adding attributes to a class
- Defining methods within a class
- Passing arguments to methods

Syntax of class and object declaration

```
# Creating Class
class MyClass:
    # Class attributes
    class_variable = "I am a class variable"

    # Constructor method (called when an object is created)
    def __init__(self, attribute1, attribute2):
        # Instance attributes
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    # Instance method
    def instance_method(self):
        print(f"Instance method called. Attributes: {self.attribute1}, {self.attribute2}")

    # Another instance method
    def another_method(self, parameter):
        print(f"Another method called with parameter: {parameter}")

# Creating an instance of MyClass
my_instance = MyClass(attribute1="Value1", attribute2="Value2")
# Accessing attributes
print(my_instance.attribute1)
```

How to create class?

- A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.
- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute
- Class Definition Syntax:

```
class ClassName:
  # Statement-1
  .
  .
  # Statement-N
```

- To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.
- Creating an Empty Class in Python

  In the above example, we have created a class named Dog using the class keyword.

  # Python3 program to demonstrate defining a class

```
class Dog:
   pass
```

How to create Objects?

- The object is an entity that has a state and behaviour associated with it.
- Real world objects are a mouse, keyboard, chair, table, pen, etc.
- As python is object oriented programming language, Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.
- An object consists of:

  - **State:** It is represented by the attributes of an object. It also reflects the properties of an object.

  - **Behaviour:** It is represented by the methods of an object. It also reflects the response of an object to other objects.

  - **Identity:** It gives a unique name to an object and enables one object to interact with other objects.
- To understand the state, behavior, and identity let us take the example of the class dog (explained above).

  - The identity can be considered as the name of the dog.

  - State or Attributes can be considered as the breed, age, or color of the dog.

  - The behavior can be considered as to whether the dog is eating or sleeping.
- **Example: obj = Dog()**

Creating a class and object with class and instance attributes

```
class Dog:
    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.attr1))
print("Tommy is also a {}".format(Tommy.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Defining methods in a class

- Now that you have a Dog class, it does have a name and age, which you can keep track of, but it doesn't actually do anything. This is where instance methods come in. You can rewrite the class to now include a bark() method. Notice how the def keyword is used again, as well as the self argument.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def bark(self):
        print("bark bark!")
```

- The bark method can now be called using the dot notation, after instantiating a new jonny object. The method should print "bark bark!" to the screen. Notice the parentheses (curly brackets) in .bark(). These are always used when calling a method. They're empty in this case, since the bark() method does not take any arguments.

```
jonny = Dog("Jonny", 2)
jonny.bark()
```

Defining methods in a class (Complete example)

- Recall how you printed jonny earlier (Slide no 4)? The code below now implements this functionality in the Dog class, with the doginfo() method. You then instantiate some objects with different properties, and call the method on them.

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

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

  def doginfo(self):
    print(self.name + " is " + str(self.age) + " year(s) old.")

jonny = Dog("Jonny", 2)
skippy = Dog("Skippy", 12)
filou = Dog("Filou", 8)

Jonny.doginfo()
skippy.doginfo()
filou.doginfo()
```

Passing arguments to methods

You would like for our dogs to have a buddy. This should be optional since not all dogs are as sociable. Take a look at the setBuddy() method below. It takes self, as per usual, and buddy as arguments. In this case, buddy will be another Dog object. Set the self.buddy attribute to buddy, and the buddy. buddy attribute to self. This means that the relationship is reciprocal; you are your buddy's buddy. In this case, Filou will be Ozzy's buddy, which means that Ozzy automatically becomes Filou's buddy. You could also set these attributes manually instead of defining a method, but that would require more work (writing two lines of code instead of one) every time you want to set a buddy. Notice that in Python, you don't need to specify what type the argument is. If this were Java, it would be required.

Passing arguments to methods

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

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

    def doginfo(self):
        print(self.name + " is " + str(self.age) + " year(s) old.")

    def birthday(self):
        self.age +=1

    def setBuddy(self, buddy):
        self.buddy = buddy
        buddy.buddy = self

jonny = Dog("Jonny", 2)
filou = Dog("Filou", 8)

jonny.setBuddy(filou)
```

Difference between Object Oriented programming and Procedural Oriented Programming

Procedural Oriented Programming	Object Oriented Programming
In procedural programming, the program is divided into small parts called functions.	In object-oriented programming, the program is divided into small parts called objects.
Procedural programming follows a top-down approach.	Object-oriented programming follows a bottom-up approach.
Procedural programming does not have any proper way of hiding data so it is less secure.	Object-oriented programming provides data hiding so it is more secure.
Procedural programming is based on the unreal world.	Object-oriented programming is based on the real world.
Doesn’t use Access modifiers	Makes use of Access modifiers ‘public’, private’, protected’.
Program is divided into functions	Program is divided into objects
Object can move freely within member functions	Data can move freely from function to function within programs
Examples: C, FORTRAN, Pascal, Basic, etc	**Examples: C++, Java, Python, C#, etc.**

Object-Oriented Programming methodologies

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

Inheritance

- Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

  - It represents real-world relationships well.

  - It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.

  - It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

- **Types of Inheritance**

  1. Single Inheritance: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

  2. Multilevel Inheritance: Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

  3. Hierarchical Inheritance: Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.

  4. Multiple Inheritance: Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Single Inheritance

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

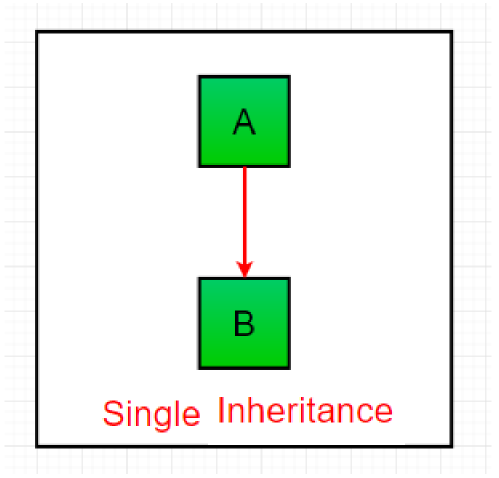
```
# Python program to demonstrate single inheritance
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class
class Child(Parent):
    def func2(self):
        print("This function is in child class.")

# Driver's code
object = Child()
object.func1()
object.func2()
```

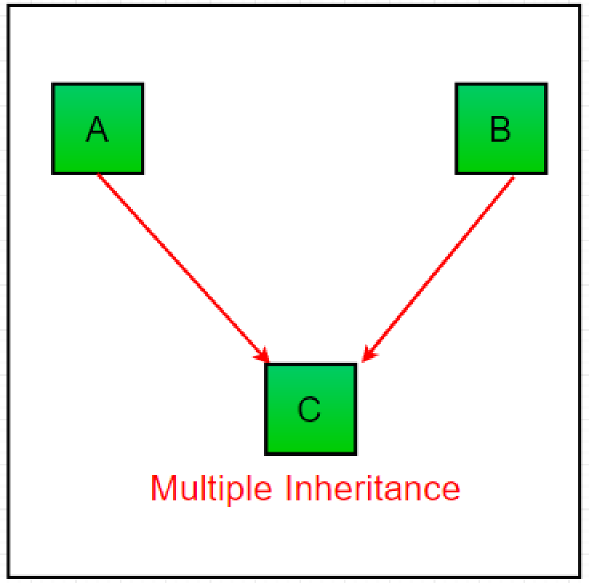
Output:

```
This function is in parent class.
This function is in child class.
```



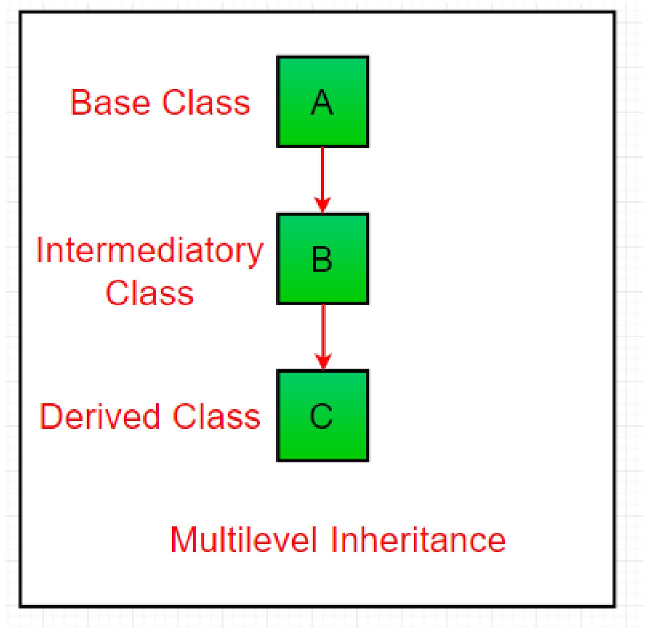
Multiple Inheritance

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



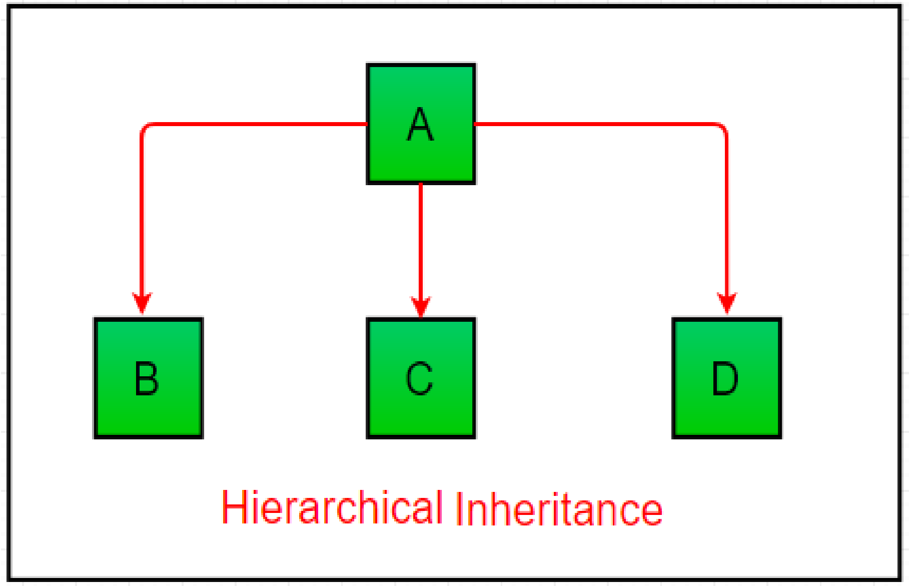
Multilevel Inheritance

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



Hierarchical Inheritance

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



Polymorphism

Polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

```
# A simple Python function to demonstrate

# Polymorphism


def add(x, y, z = 0):

    return x + y+ z


# Driver code

print(add(2, 3))

print(add(2, 3, 4))
```

Steps to write object oriented program in Python

1 **Step 1: Class Definition**

Create a class with attributes and methods that represent a real-world object or concept.

2 **Step 2: Object Instantiation**

Create objects (instances) of the class, each with its own set of attribute values.

3 **Step 3: Accessing Methods and Attributes**

Invoke methods and access attributes of objects to perform desired operations.

File handling in python

I/O Operations

Read, write, and manipulate files to store and retrieve data in your Python programs.

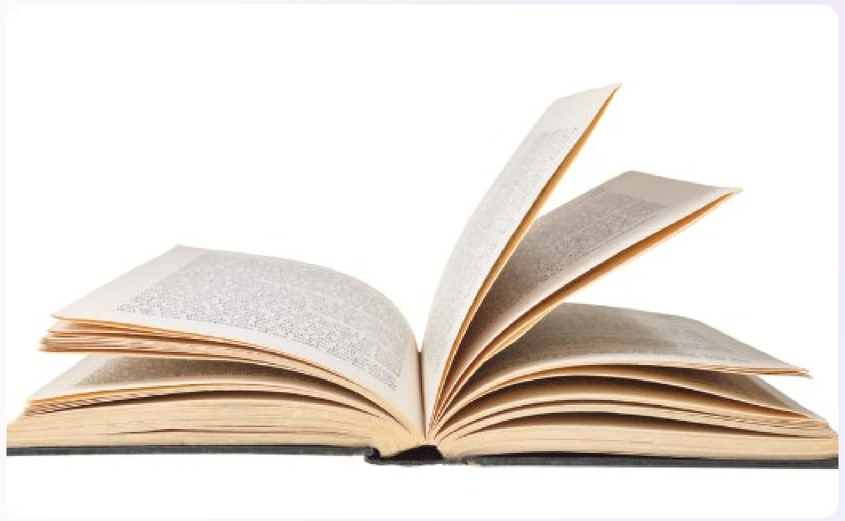
File Handling

Perform essential operations like opening, closing, and deleting files on your system.

Error Handling

Implement robust error handling techniques to gracefully deal with file-related issues.

Code Examples: Exploring Python's File System



Reading Files

```
with open("example.txt", "r") as file:
print(file.read())
```

Writing Files

```
with open("example.txt", "w") as file:
file.write("Hello, world!")
```

Appending Files

```
with open("example.txt", "a") as file:
file.write("\nWelcome to Python
Programming!")
```