OOPs

Python supports OOPs(Object oriented programming) paradigm.It allows us to develop applications using an Object-Oriented approach. In Python, we can easily create and use classes and objects.

object-oriented programming (OOP) is a programming paradigm that is based on the concept of objects. Objects interact with each other and that is how the program functions. Objects have states and behaviors. Many objects can have similar characteristics and if you need to work with them in your program it may be easier to create a class for similar objects. Classes represent the common structure of similar objects: their data and their behavior.

Class: The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an Car class, then it should contain an attribute and method, i.e. an model_no, mileage, No_of_gears, Fuel_capacity, etc.

Syntax of declaring classes

In [2]:

```
class MyClass:
    pass
```

Classes are declared with a keyword class and the name of the class.

Generally, class name starts with a capital letter and it is usually a noun or a noun phrase.

Classes allow you to define the data and the behaviors of similar objects. The behavior is described by methods. A method is similar to a function in that it is a block of code that has a name and performs a certain action. Methods, however, are not independent since they are defined within a class. Data within classes are stored in the attributes (variables) and there are two kinds of them: class attributes and instance attributes. The difference between those two is:.

1. Class attribute

A class attribute is an attribute shared by all instances of the class. Let's consider the class Book as an example

Example:

In [4]:

```
# Book class
class Book:
    material = "paper"
    cover = "paperback"
    all_books = []
```

This class has a string variable material with the value "paper", a string variable cover with the value "paperback" and an empty list as an attribute all_books. All those variables are class attributes and they can be accessed using the dot notation with the name of the class:

Book.material # "paper" Book.cover # "paperback" Book.all_books # []

As for the instance variables, they store the data unique to each object of the class. They are defined within the class methods, notably, within the init method

2. Class instance

Now, let's create an instance of a Book class. For that we would need to execute this code:

In [7]:

In [7]:

```
# Book instance
my_book = Book()
```

Here we not only created an instance of a Book class but also assigned it to the variable my_book. The syntax of instantiating a class object resembles the function notation: after the class name, we write parentheses.

Our my_book object has access to the contents of the class Book: its attributes and methods.

init() function:

The init method is a constructor. Constructors are a concept from the object-oriented programming. A class can have one and only one constructor. If init is defined within a class, it is automatically invoked when we create a new class instance.

Example:

In [17]:

```
class River:

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

volga = River("Volga", 3530)
print(volga.name)
print(volga.length)

seine = River("Seine", 776)
print(seine.name)
print(seine.length)

nile = River("Nile", 6852)
print(seine.name)
print(seine.length)
```

```
Volga
3530
Seine
776
Seine
776
```

Self Parameter:

You may have noticed that our init method had another argument besides name and length: self. The self argument represents a particular instance of the class and allows us to access its attributes and methods. In the example with init, we basically create attributes for the particular instance and assign the values of method arguments to them. It is important to use the self parameter inside the method if we want to save the values of the instance for the later use.

Most of the time we also need to write the self parameter in other methods because when the method is called the first argument that is passed to the method is the object itself. Let's add a method to our River class and see how it works. The syntax of the methods is not of importance at the moment, just pay attention to the use of the self:

Example:

In [19]:

```
class River:
    all_rivers = []

    def __init__(self, name, length):
        self.name = name
```

```
        self.length = length

    def get_info(self):
        print("The length of the {0} is {1} km".format(self.name, self.length))


volga = River("Volga", 3530)
print(volga.get_info())

seine = River("Seine", 776)
print(seine.get_info())

nile = River("Nile", 6852)
print(nile.get_info())
```

```
The length of the Volga is 3530 km
None
The length of the Seine is 776 km
None
The length of the Nile is 6852 km
None
```

Instance attribute:

In the class River, the attributes name and length are instance attributes since they are defined within a method (init) and have self before them. Usually, instance attributes are created within the init method since it's the constructor, but you can define instance attributes in other methods as well.

Instance attributes, naturally, are used to distinguish objects: their values are different for different instances.

In [21]:

```
print(volga.name)
print(seine.name)
print(nile.name)
```

```
Volga
Seine
Nile
```

Object method:

Object can also contain methods. Methods in objects are functions that belong to that object.

Example:

In [24]:

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

    def show(self):
        print("The length of the {0} is {1} km".format(self.name, self.length))

volga = River("Volga", 3530)
print(volga.show())

seine = River("Seine", 776)
print(seine.show())

nile = River("Nile", 6852)
print(nile.show())
```

```
The length of the Volga is 3530 km
None
```

```
The length of the Seine is 776 km
None
The length of the Nile is 6852 km
None
```

Inheritance:

Inheritance is a mechanism that allows classes to inherit methods or properties from other classes. Or, in other words, inheritance is a mechanism of deriving new classes from existing ones.

The purpose of inheritance is to reuse existing code. Often, objects of one class may resemble objects of another class, so instead of rewriting the same methods and attributes, we can make it so that a class inherits those methods and attributes from another class.

When we talk about inheritance, the terminology resembles biological inheritance: we have child classes (or subclasses, derived classes) that inherit methods or variables from parent classes (or base classes, superclasses). Child classes can also redefine methods of the parent class if necessary.

Class object:

Inheritance is very easy to implement in your programs. Any class can be a parent class, so all we need to do is to write in the definition of the child class the name of the parent class in parentheses after the child class:

Syntax:

 # inheritance syntax class ChildClass(ParentClass): # methods and attributes ..

NOTE:

The definition of the parent class should precede the definition of the child class, otherwise, you'll get a NameError! If a class has several subclasses, its definition should precede them all. The "sibling" classes can be defined in any order.

 # parent class is explicit class SomeClass(object): # methods and attributes ... # parent class is implicit class SomeClass: # methods and attributes ...

Example:

In [34]:

```
#Definition of parent class

class Person:
    def __init__(self,Fname,Lname):
        self.firstname=Fname
        self.lastname=Lname

    def show(self):
        print("The first name is {0} and last name is {1}".format(self.firstname,self.lastname))

p1=Person("Sachin","Kapoor")
p1.show()

p2=Person("Ranjeet","Kumar")
p2.show()

#Definition of child class

class Student(Person):
    pass

s1=Student("Abhishek","Jaiswal")
s1.show()

s1=Student("Rishav","Goyal")
s1.show()
```

```
The first name is Sachin and last name is Kapoor
The first name is Ranjeet and last name is Kumar
The first name is Abhishek and last name is Jaiswal
The first name is Rishav and last name is Goyal
```

Polymorphism:

The literal meaning of polymorphism is the condition of occurence in different forms.

There are four types of polymorphism:

1. **Duck Typing**
2. **Operator overloading**
3. **method overloading**
4. **method overriding**

Duck typing: is a type of system used in dynamic language. For example, Python ,perl ,Ruby, Javascript, etc. Where the type or the class of an object is less important than the method it defines. Using "duck typing", we do not check types at all.

Example:

In [40]:

```
class Pycharm:
    def execute(self):
        print("Compiling")
        print("Running")

class Myeditor:
    def execute(self):
        print("Spell check")
        print("conventional typing")

class laptop:
    def code(self,ide):
        ide.execute()

ide=Pycharm()
lap1=laptop()
lap1.code()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-45d81922f4f3> in <module>
     14
     15 ide=Pycharm()
---> 16 lap1=laptop(ide)
     17 lap1.code()
     18

TypeError: laptop() takes no arguments
```

Method overriding:

Overriding is the ability of a class to change the implementation of the methods inherited from its ancestor classes.

This feature is extremely useful as it allows us to explore inheritance to its full potential. We can not only reuse existing code and method implementations but also upgrade and advance them if needed.

Overriding is a concept applicable only to class hierarchies: without inheritance, we cannot talk about method overriding. Let's consider an example of a class hierarchy:

Example:

In [49]:

```
class Parent:
    def do_something(self):
        print("Did something")


class Child(Parent):
    def do_something(self):
        print("Did something else")


parent = Parent()
child = Child()

parent.do_something()
child.do_something()
```

```
Did something
Did something else
```

Code Explanation: Here, the method do_something is overridden in the class Child. If we hadn't overridden it, the method would have the same implementation as in the class Parent. The code child.do_something() would then print Did something.

super(): Python has a special function for calling the method of the parent class inside the methods of the child class: the super() function. It returns a proxy, a temporary object of the parent class, and allows us to call a method of the parent class using this proxy. Let's take a look at the following example:

Example:

In [44]:

```
class Parent:
    def __init__(self, name):
        self.name = name
        print("Called Parent __init__")


class Child(Parent):
    def __init__(self, name):
        super().__init__(name)
        print("Called Child __init__")

jack = Child("Jack")
```

```
Called Parent __init__
Called Child __init__
```

Multiple Inheritance:

Multiple inheritance is when a class has two or more parent classes.

In the code, multiple inheritance looks similar to the single inheritance. Only now, in brackets after the child class, you need to write all parent classes instead of just one:

Example:

class Person: ... class Student(Person): ... class Programmer(Person): ... class StudentProgrammer(Student, Programmer): ...

Code Explanation: As you can see, there are a basic parent class Person and classes Student and Programmer that inherit from it. The class StudentProgrammer, in its turn, inherits from both Student and Programmer classes which makes it a case of multiple inheritance. This way, we can say that StudentProgrammer has two parent classes, Student and Programmer while Person can be regarded as a "grandparent" class.

Diamond problem: So, we have a class hierarchy with one superclass, two classes that inherit from it, and a class that has those child classes as parents. As you can see from the hierarchy scheme above, the whole structure is shaped like a

diamond

Example:

class Person: def print_message(self): print("Message from Person") class Student(Person): def print_message(self): print("Message from Student") class Programmer(Person): def print_message(self): print("Message from Programmer") class StudentProgrammer(Student, Programmer): . . . . . .

Class Person has a method print_message that classes Student and Programmer override to print their own messages. The class StudentProgrammer doesn't override this method.

The question is, then: if we create an instance of the class StudentProgrammer and call print_message method, which message will be printed?

This is the crux of the diamond problem: how to choose an implementation when we have several alternatives.

MRO: Different programming languages use different techniques for dealing with the diamond problem. Basically, what we need to do is to somehow transform the diamond shape (or any complicated hierarchy) into a single line so that we know in which order to look for the necessary method. Python uses the C3 Linearization algorithm that calculates the Method Resolution Order (MRO).

MRO tells us how the particular class hierarchy looks in a linear form and how we should navigate this hierarchy. Two basic rules are that child classes precede parent classes and parent classes are placed in the order they were listed in.

In [51]:

```
print(StudentProgrammer.__mro__)
# (<class '__main__.StudentProgrammer'>, <class '__main__.Student'>, <class
'__main__.Programmer'>, <class '__main__.Person'>, <class 'object'>)
```

```
(<class '__main__.StudentProgrammer'>, <class '__main__.Student'>, <class '__main__.Programmer'>,
<class '__main__.Person'>, <class 'object'>)
```

super() with multiple inheritance:

Example:

In [48]:

```
class Person:
    def print_message(self):
        print("Message from Person")


class Student(Person):
    def print_message(self):
        print("Message from Student")
        super().print_message()


class Programmer(Person):
    def print_message(self):
        print("Message from Programmer")
        super().print_message()


class StudentProgrammer(Student, Programmer):
    def print_message(self):
        super().print_message()

jack = StudentProgrammer()
jack.print_message()
```

```
Message from Student
Message from Programmer
Message from Person
```