UNIT-V

OOPS using Python: Classes and Objects, Inheritance and Polymorphism, Abstract Classes and Interfaces.

Regular Expressions: Introduction, Special Symbols and Characters, Res and Python Multithreaded Programming: Introduction, Threads and Processes, Python, Threads, and the Global Interpreter Lock, Thread Module, Threading Module, Related Modules.

Python OOPs Concepts

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming.

The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Inheritance
- Polymorphism
- Abstract Classes
- Interfaces

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.

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.

- 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

Syntax:

class ClassName:

```
    #Statement-1
    .

    .

    .

    #Statement-N
```

In [1]:
```
# Python3 program to demonstrate defining a class

class Dog:
    pass
```

Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You’ve been using objects all along and may not even realize it.

An object consists of:

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

Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

In [2]:
```
obj = Dog()
```

Creating Classes and objects with methods

Here, The Dog class is defined with two attributes:

- attr1 is a class attribute set to the value “mammal”. Class attributes are shared by all instances of the class.
- **init** is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters:self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog.

The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The **init** method is called for each instance to initialize their name attributes with the provided names. The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

In [9]:
```
class Dog:

# class attribute
    attr1 = "mammal"

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

    def speak(self):
        print("My name is {}".format(self.name))

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

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

```
My name is Rodger
My name is Tommy
```

In [12]:
```
class Parrot:

    # class attribute
    name = ""
    age = 0

# create parrot1 object
parrot1 = Parrot()
parrot1.name = "Blu"
parrot1.age = 10

# create another object parrot2
parrot2 = Parrot()
parrot2.name = "Woo"
parrot2.age = 15

# access attributes
print(f"{parrot1.name} is {parrot1.age} years old")
print(f"{parrot2.name} is {parrot2.age} years old")
```

```
Blu is 10 years old
Woo is 15 years old
```

The Python self

- Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
- If we have a method that takes no arguments, then we still have to have one argument.
- This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2)

self represents the instance of the class. By using the “self” we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

In [3]:
```
#it is clearly seen that self and obj is referring to the same object

class check:
    def __init__(self):
        print("Address of self = ",id(self))

obj = check()
print("Address of class object = ",id(obj))
```

```
Address of self =  2009824513760
Address of class object =  2009824513760
```

In [4]:
```
# Write Python3 code here

class car():

# init method or constructor
    def __init__(self, model, color):
        self.model = model
        self.color = color

    def show(self):
        print("Model is", self.model )
        print("color is", self.color )

# both objects have different self which contain their attributes
audi = car("audi a4", "blue")
ferrari = car("ferrari 488", "green")

audi.show() # same output as car.show(audi)
ferrari.show() # same output as car.show(ferrari)

#note:we can also do like this
print("Model for audi is ",audi.model)
print("Colour for ferrari is ",ferrari.color)
#this happens because after assigning in the constructor the attributes are li
#here attributes(model,colour) are linked to objects(audi,ferrari) as we initi
# Behind the scene, in every instance method
# call, python sends the instances also with
# that method call like car.show(audi)
```

```
Model is audi a4
color is blue
Model is ferrari 488
color is green
Model for audi is  audi a4
Colour for ferrari is  green
```

Self is the first argument to be passed in Constructor and Instance Method.

Self must be provided as a First parameter to the Instance method and constructor. If you don't provide it, it will cause an error.

In [6]:
```
# Self is always required as the first argument
class check:
    def __init__():
        print("This is Constructor")

object = check()
print("Worked fine")


# Following Error is produced if Self is not passed as an argument
Traceback (most recent call last):
File "/home/c736b5fad311dd1eb3cd2e280260e7dd.py"
    object = check()
TypeError: __init__() takes 0 positional arguments but 1 was given
# this code is Contributed by Samyak Jain
```

```
  Cell In[6], line 11
    Traceback (most recent call last):
                      ^
SyntaxError: invalid syntax
```

Self is a convention and not a Python keyword .

self is parameter in Instance Method and user can use another parameter name in place of it. But it is advisable to use self because it increases the readability of code, and it is also a good programming practice.

In [7]:
```
# Write Python3 code here

class this_is_class:
    def __init__(in_place_of_self):
        print("we have used another "
        "parameter name in place of self")

object = this_is_class()
```

```
we have used another parameter name in place of self
```

The Python init Method

The **init** method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Now let us define a class and create some objects using the self and **init** method.

In [8]:
```
#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.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

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

```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy
```

Python program to demonstrate error if we forget to invoke init() of the parent If you forget to invoke the **init**() of the parent class then its instance variables would not be available to the child class. The following code produces an error for the same reason.

In [18]:
```
class A:
    def __init__(self, n='Rahul'):
        self.name = n

class B(A):
    def __init__(self, roll):
        self.roll = roll

object = B(23)
print(object.name)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[18], line 10
      7         self.roll = roll
      9 object = B(23)
---> 10 print(object.name)

AttributeError: 'B' object has no attribute 'name'
```

The super() Function

The super() function is a built-in function that returns the objects that represent the parent class. It allows to access the parent class's methods and attributes in the child class.

Example: super() function with simple Python inheritance

In this example, we created the object 'obj' of the child class. When we called the constructor of the child class 'Student', it initialized the data members to the values passed during the object creation.

Then using the super() function, we invoked the constructor of the parent class.

In [20]:
```
# parent class
class Person():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(self.name, self.age)

# child class
class Student(Person):
    def __init__(self, name, age):
        self.sName = name
        self.sAge = age
# inheriting the properties of parent class
        super().__init__("Rahul", age)

    def displayInfo(self):
        print(self.sName, self.sAge)

obj = Student("Mayank", 23)
obj.display()
obj.displayInfo()
```

```
Rahul 23
Mayank 23
```

Adding Properties

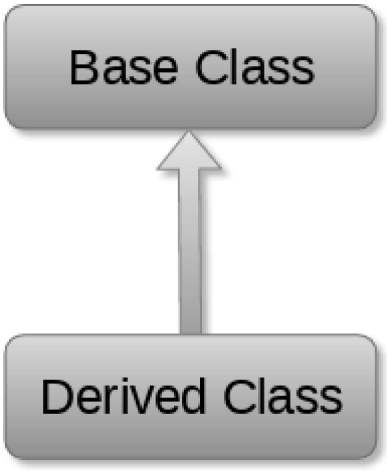
One of the features that inheritance provides is inheriting the properties of the parent class as well as adding new properties of our own to the child class.

In [21]:
```
# parent class
class Person():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(self.name, self.age)

# child class
class Student(Person):
    def __init__(self, name, age, dob):
        self.sName = name
        self.sAge = age
        self.dob = dob
# inheriting the properties of parent class
        super().__init__("Rahul", age)

    def displayInfo(self):
        print(self.sName, self.sAge, self.dob)

obj = Student("Mayank", 23, "16-03-2000")
obj.display()
obj.displayInfo()
```

```
Rahul 23
Mayank 23 16-03-2000
```

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.



Syntax

class derived-class(base 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

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **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.
- **Hierarchical Inheritance**: Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance**: Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

In [11]:
```
# Python code to demonstrate how parent constructors are called.

# parent class
class Person():

# __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

# invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))

# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using its instance
a.display()
a.details()
```

```
Rahul
886012
My name is Rahul
IdNumber: 886012
Post: Intern
```

Single inheritance

When a child class inherits from only one parent class, it is called single inheritance.

In [17]:
```
# parent class
class Person():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(self.name, self.age)

# child class
class Student(Person):
    def __init__(self, name, age, dob):
        self.sName = name
        self.sAge = age
        self.dob = dob
# inheriting the properties of parent class
        super().__init__("Rahul", age)

    def displayInfo(self):
        print(self.sName, self.sAge, self.dob)

obj = Student("Mayank", 23, "16-03-2000")
obj.display()
obj.displayInfo()
```

```
Rahul 23
Mayank 23 16-03-2000
```

In [13]:
```
# base class
class Animal:

    def eat(self):
        print( "I can eat!")

    def sleep(self):
        print("I can sleep!")

# derived class
class Dog(Animal):

    def bark(self):
        print("I can bark! Woof woof!!")

# Create object of the Dog class
dog1 = Dog()

# Calling members of the base class
dog1.eat()
dog1.sleep()

# Calling member of the derived class
dog1.bark();
```

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class.

There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.
python-inheritance2.png

Syntax

class class1:

```
    <class-suite>
```

class class2(class1):

```
    <class suite>
```

class class3(class2):

```
    <class suite>
```

.

.

In [26]:
```
# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.
# (Generally, object is made ancestor of all classes)
# In Python 3.x "class Person" is
# equivalent to "class Person(object)"

class Base():

# Constructor
    def __init__(self, name):
        self.name = name

# To get name
    def getName(self):
        return self.name


# Inherited or Sub class (Note Person in bracket)
class Child(Base):

# Constructor
    def __init__(self, name, age):
        Base.__init__(self, name)
        self.age = age

# To get name
    def getAge(self):
        return self.age

# Inherited or Sub class (Note Person in bracket)


class GrandChild(Child):

# Constructor
    def __init__(self, name, age, address):
        Child.__init__(self, name, age)
        self.address = address

# To get address
    def getAddress(self):
        return self.address


# Driver code
g = GrandChild("CMR TC", 501401, "Hyderabad")
print(g.getName(), g.getAge(), g.getAddress())
```

```
CMR TC 501401 Hyderabad
```

In [14]:
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

```
dog barking
Animal Speaking
Eating bread...
```

Hierarchical Inheritance

Hierarchical inheritance is a type in Python where you can inherit more than one class from the base or parent class.

In [27]:
```
# base class

class Animal():

    def animal(self):

        print("I'm an Animal")


# child class 1

class Cat(Animal):

    def cat(self):

        print("I'm a cat Meow Meow!")


# child class 2

class Dog(Animal):

    def dog(self):

        print("I'm a dog Brak Bark!")


# create object of child classes

cat = Cat()

dog = Dog()

print("Cat")

cat.animal()

cat.cat()


print("\nDog")

dog.animal()

dog.dog()
```

```
Cat
I'm an Animal
I'm a cat Meow Meow!

Dog
I'm an Animal
I'm a dog Brak Bark!
```

Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.
 python-inheritance3.png

Syntax

class Base1:

```
    <class-suite>
```

class Base2:

```
    <class-suite>
```

.

.

.

class BaseN:

```
    <class-suite>
```

class Derived(Base1, Base2, ... BaseN):

```
    <class-suite>
```

In [24]:
```
# Python example to show the working of multiple inheritance

class Base1():
    def __init__(self):
        self.str1 = "Geek1"
        print("Base1")


class Base2():
    def __init__(self):
        self.str2 = "Geek2"
        print("Base2")


class Derived(Base1, Base2):
    def __init__(self):

# Calling constructors of Base1 and Base2 classes
        Base1.__init__(self)
        Base2.__init__(self)
        print("Derived")

    def printStrs(self):
        print(self.str1, self.str2)


ob = Derived()
ob.printStrs()
```

```
Base1
Base2
Derived
Geek1 Geek2
```

In [15]:
```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

```
30
200
0.5
```

Polymorphism

Polymorphism is the principle that one kind of thing can take a variety of forms. In the context of programming, this means that a single entity in a programming language can behave in multiple ways depending on the context.

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.

inbuilt polymorphic functions

In [28]:
```
# Python program to demonstrate in-built poly-morphic functions

# len() being used for a string
print(len("CMR Technical Campus"))

# len() being used for a list
print(len([10, 20, 30]))
```

```
20
3
```

user-defined polymorphic functions

In [29]:
```
# 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))
```

```
5
9
```

Polymorphism with class methods

The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

In [30]:
```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Polymorphism with Inheritance

In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class.

However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class.

In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

In [31]:
```
class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Polymorphism with a Function and objects

It is also possible to create a function that can take any object, allowing for polymorphism.

In this example, let's create a function called "func()" which will take an object which we will name "obj". Though we are using the name 'obj', any instantiated object will be able to be called into this function. Next, let's give the function something to do that uses the 'obj' object we passed to it.

In this case, let's call the three methods, viz., capital(), language() and type(), each of which is defined in the two classes 'India' and 'USA'. Next, let's create instantiations of both the 'India' and 'USA' classes if we don't have them already. With those, we can call their action using the same func() function

In [32]:
```
def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Implementing Polymorphism with a Function

In [33]:
```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

    def func(obj):
        obj.capital()
        obj.language()
        obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

polymorphism using inheritance and method overriding

In [34]:
```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement this method")

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

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

# Create a list of Animal objects
animals = [Dog(), Cat()]

# Call the speak method on each object
for animal in animals:
    print(animal.speak())
```

```
Woof!
Meow!
```

Abstract Classes

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class.

A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

Why use Abstract Base Classes : By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

How Abstract Base classes work : By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is **ABC**.

ABC works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword **@abstractmethod**.

In [35]:
```
# Python program showing abstract base class work

from abc import ABC, abstractmethod

class Polygon(ABC):

    @abstractmethod
    def noofsides(self):
        pass

class Triangle(Polygon):

# overriding abstract method
    def noofsides(self):
        print("I have 3 sides")

class Pentagon(Polygon):

# overriding abstract method
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(Polygon):

# overriding abstract method
    def noofsides(self):
        print("I have 6 sides")

class Quadrilateral(Polygon):

# overriding abstract method
    def noofsides(self):
        print("I have 4 sides")

# Driver code
R = Triangle()
R.noofsides()

K = Quadrilateral()
K.noofsides()

R = Pentagon()
R.noofsides()

K = Hexagon()
K.noofsides()
```

```
I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides
```

In [36]:
```
# Python program showing abstract base class work

from abc import ABC, abstractmethod
class Animal(ABC):

    def move(self):
        pass

class Human(Animal):

    def move(self):
        print("I can walk and run")

class Snake(Animal):

    def move(self):
        print("I can crawl")

class Dog(Animal):

    def move(self):
        print("I can bark")

class Lion(Animal):

    def move(self):
        print("I can roar")

# Driver code
R = Human()
R.move()

K = Snake()
K.move()

R = Dog()
R.move()

K = Lion()
K.move()
```

```
I can walk and run
I can crawl
I can bark
I can roar
```

Implementation Through Subclassing

By subclassing directly from the base, we can avoid the need to register the class explicitly.

In this case, the Python class management is used to recognize PluginImplementation as implementing the abstract PluginBase.

In [37]:
```
# Python program showing implementation of abstract class through subclassing

import abc

class parent:
    def geeks(self):
        pass

class child(parent):
    def geeks(self):
        print("child class")

# Driver code
print( issubclass(child, parent))
print( isinstance(child(), parent))
```

```
True
True
```

- A side-effect of using direct subclassing is, it is possible to find all the implementations of your plugin by asking the base class for the list of known classes derived from it.

Concrete Methods in Abstract Base Classes

Concrete classes contain only concrete (normal)methods whereas abstract classes may contain both concrete methods and abstract methods.

The concrete class provides an implementation of abstract methods, the abstract base class can also provide an implementation by invoking the methods via **super()**.

In [38]:
```
# Python program invoking a method using super()

import abc
from abc import ABC, abstractmethod

class R(ABC):
    def rk(self):
        print("Abstract Base Class")

class K(R):
    def rk(self):
        super().rk()
        print("subclass ")

# Driver code
r = K()
r.rk()
```

```
Abstract Base Class
subclass
```

Abstract Properties

Abstract classes include attributes in addition to methods, you can require the attributes in concrete classes by defining them with **@abstractproperty**.

In [39]:
```
# Python program showing abstract properties

import abc
from abc import ABC, abstractmethod

class parent(ABC):
    @abc.abstractproperty
    def cmrtc(self):
        return "parent class"
class child(parent):

    @property
    def cmrtc(self):
        return "child class"


try:
    r =parent()
    print( r.cmrtc)
except Exception as err:
    print (err)

r = child()
print (r.cmrtc)
```

```
Can't instantiate abstract class parent with abstract method cmrtc
child class
```

- In the above example, the Base class cannot be instantiated because it has only an abstract version of the property getter method.

Abstract Class Instantiation

Abstract classes are incomplete because they have methods that have nobody. If python allows creating an object for abstract classes then using that object if anyone calls the abstract method, but there is no actual implementation to invoke.

So we use an abstract class as a template and according to the need, we extend it and build on it before we can use it. Due to the fact, an abstract class is not a concrete class, it cannot be instantiated. When we create an object for the abstract class it raises an **error**.

In [40]:
```
# Python program showing abstract class cannot be an instantiation
from abc import ABC,abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        pass
class Human(Animal):
    def move(self):
        print("I can walk and run")

class Snake(Animal):
    def move(self):
        print("I can crawl")

class Dog(Animal):
    def move(self):
        print("I can bark")

class Lion(Animal):
    def move(self):
        print("I can roar")

c=Animal()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[40], line 24
     21     def move(self):
     22         print("I can roar")
---> 24 c=Animal()

TypeError: Can't instantiate abstract class Animal with abstract method move
```

Interfaces

In object-oriented languages like Python, the interface is a collection of method signatures that should be provided by the implementing class. Implementing an interface is a way of writing an organized code and achieve abstraction.

The package **zope.interface** provides an implementation of “object interfaces” for Python. It is maintained by the Zope Toolkit project. The package exports two objects, ‘Interface’ and ‘Attribute’ directly. It also exports several helper methods. It aims to provide stricter semantics and better error messages than Python’s built-in abc module.

Declaring interface In python, interface is defined using python class statements and is a subclass of interface.Interface which is the parent interface for all interfaces.

Syntax :

class IMyInterface(zope.interface.Interface):

```
    # methods and attributes
```

In [41]:
```
import zope.interface


class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute("foo")
    def method1(self, x):
        pass
    def method2(self):
        pass

print(type(MyInterface))
print(MyInterface.__module__)
print(MyInterface.__name__)

# get attribute
x = MyInterface['x']
print(x)
print(type(x))
```

```
<class 'zope.interface.interface.InterfaceClass'>
__main__
MyInterface
__main__.MyInterface.foo
<class 'zope.interface.interface.Attribute'>
```

Implementing interface

Interface acts as a blueprint for designing classes, so interfaces are implemented using implementer decorator on class. If a class implements an interface, then the instances of the class provide the interface. Objects can provide interfaces directly, in addition to what their classes implement.

Syntax :

@zope.interface.implementer(*interfaces)

class Class_name:

```
    # methods
```

In [42]:
```
import zope.interface

class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute("foo")
    def method1(self, x):
        pass
    def method2(self):
        pass

@zope.interface.implementer(MyInterface)
class MyClass:
    def method1(self, x):
        return x**2
    def method2(self):
        return "foo"
```

We declared that MyClass implements MyInterface. This means that instances of MyClass provide MyInterface.

Methods

- **implementedBy(class) –** returns a boolean value, True if class implements the interface else False
- **providedBy(object) –** returns a boolean value, True if object provides the interface else False
- **providedBy(class) –** returns False as class does not provide interface but implements it
- **list(zope.interface.implementedBy(class)) –** returns the list of interfaces implemented by a class
- **list(zope.interface.providedBy(object)) –** returns the list of interfaces provided by an object.
- **list(zope.interface.providedBy(class)) –** returns empty list as class does not provide interface but implements it.

In [43]:
```
import zope.interface

class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute('foo')
    def method1(self, x, y, z):
        pass
    def method2(self):
        pass

@zope.interface.implementer(MyInterface)
class MyClass:
    def method1(self, x):
        return x**2
    def method2(self):
        return "foo"
obj = MyClass()

# ask an interface whether it is implemented by a class:
print(MyInterface.implementedBy(MyClass))

# MyClass does not provide MyInterface but implements it:
print(MyInterface.providedBy(MyClass))

# ask whether an interface is provided by an object:
print(MyInterface.providedBy(obj))

# ask what interfaces are implemented by a class:
print(list(zope.interface.implementedBy(MyClass)))

# ask what interfaces are provided by an object:
print(list(zope.interface.providedBy(obj)))

# class does not provide interface
print(list(zope.interface.providedBy(MyClass)))
```

```
True
False
True
[<InterfaceClass __main__.MyInterface>]
[<InterfaceClass __main__.MyInterface>]
[]
```

Interface Inheritance

Interfaces can extend other interfaces by listing the other interfaces as base interfaces.

Functions

- **extends(interface) –** returns boolean value, whether one interface extends another.
- **isOrExtends(interface) –** returns boolean value, whether interfaces are same or one extends another.
- **isEqualOrExtendedBy(interface) –** returns boolean value, whether interfaces are same or one is extended by another.

In [44]:
```
import zope.interface

class BaseI(zope.interface.Interface):
    def m1(self, x):
        pass
    def m2(self):
        pass

class DerivedI(BaseI):
    def m3(self, x, y):
        pass

@zope.interface.implementer(DerivedI)
class cls:
    def m1(self, z):
        return z**3
    def m2(self):
        return 'foo'
    def m3(self, x, y):
        return x ^ y

# Get base interfaces
print(DerivedI.__bases__)

# Ask whether baseI extends DerivedI
print(BaseI.extends(DerivedI))

# Ask whether baseI is equal to or is extended by DerivedI
print(BaseI.isEqualOrExtendedBy(DerivedI))

# Ask whether baseI is equal to or extends DerivedI
print(BaseI.isOrExtends(DerivedI))

# Ask whether DerivedI is equal to or extends BaseI
print(DerivedI.isOrExtends(DerivedI))
```

```
(<InterfaceClass __main__.BaseI>,)
False
True
False
True
```

Regular Expressions

A Regular Expressions (RegEx) is a special sequence of characters that uses a search pattern to find a string or set of strings. It can detect the presence or absence of a text by matching it with a particular pattern, and also can split a pattern into one or more sub-patterns.

Python provides a re module that supports the use of regex in Python. Its primary function is to offer a search, where it takes a regular expression and a string. Here, it either returns the first match or else none.

In [45]:
```
import re

s = 'CMR Technical Campus, Hyderabad'

match = re.search(r'amp', s)

print('Start Index:', match.start())
print('End Index:', match.end())
```

```
Start Index: 15
End Index: 18
```

The above code gives the starting index and the ending index of the string portal.

Note: Here r character (r'amp') stands for raw, not regex. The raw string is slightly different from a regular string, it won't interpret the \ character as an escape character. This is because the regular expression engine uses \ character for its own escaping purpose.

Before starting with the Python regex module let's see how to actually write regex using metacharacters or special sequences.

MetaCharacters

To understand the RE analogy, MetaCharacters are useful, important, and will be used in functions of module re. Below is the list of metacharacters.

MetaCharacters	Description
\	Used to drop the special meaning of character following it

\ – Backslash

The backslash () makes sure that the character is not treated in a special way. This can be considered a way of escaping metacharacters.

For example, if you want to search for the dot(.) in the string then you will find that dot(.) will be treated as a special character as is one of the metacharacters (as shown in the above table).

So for this case, we will use the backslash() just before the dot(.) so that it will lose its specialty.

In [46]:
```
import re

s = 'CMR Technical.Campus'

# without using \
match = re.search(r'.', s)
print(match)

# using \
match = re.search(r'\.', s)
print(match)
```

```
<re.Match object; span=(0, 1), match='C'>
<re.Match object; span=(13, 14), match='.'>
```

[] – Square Brackets

Square Brackets ([]) represent a character class consisting of a set of characters that we wish to match.

For example, the character class [abc] will match any single a, b, or c.

We can also specify a range of characters using – inside the square brackets. For example,

- [0, 3] is sample as [0123]
- [a-c] is same as [abc]

We can also invert the character class using the caret(^) symbol. For example,

- [^0-3] means any number except 0, 1, 2, or 3
- [^a-c] means any character except a, b, or c

In [47]:
```
import re

string = "The quick brown fox jumps over the lazy dog"
pattern = "[a-m]"
result = re.findall(pattern, string)

print(result)
```

```
['h', 'e', 'i', 'c', 'k', 'b', 'f', 'j', 'm', 'e', 'h', 'e', 'l', 'a', 'd',
'g']
```

^ – Caret

Caret (^) symbol matches the beginning of the string i.e. checks whether the string starts with the given character(s) or not.

For example –

- ^c will check if the string starts with c such as cmr, cmrtc, cmrcet, c, etc.
- ^cmr will check if the string starts with cmr such as cmrtc, cmrit, cmrcet etc.

In [49]:
```
# import re

# Match strings starting with "The"
regex = r'^The'
strings = ['The quick brown fox', 'The lazy dog', 'A quick brown fox']
for string in strings:
    if re.match(regex, string):
        print(f'Matched: {string}')
    else:
        print(f'Not matched: {string}')
```

```
Matched: The quick brown fox
Matched: The lazy dog
Not matched: A quick brown fox
```

Dollar

Dollar($) symbol matches the end of the string i.e checks whether the string ends with the given character(s) or not.

For example –

- r$ will check for the string that ends with a such as cmr, vmr, r, etc.
- mr$ will check for the string that ends with mr such as cmr, vmr, gmr, etc.

In [50]:
```
import re

string = "Hello World!"
pattern = r"World!$"

match = re.search(pattern, string)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

```
Match found!
```

. – Dot

Dot(.) symbol matches only a single character except for the newline character (\n).

For example –

- a.b will check for the string that contains any character at the place of the dot such as acb, acbd, abbb, etc
- .. will check if the string contains at least 2 characters

In [51]:
```
import re

string = "The quick brown fox jumps over the lazy dog."
pattern = r"brown.fox"

match = re.search(pattern, string)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

```
Match found!
```

| – Or Or symbol works as the or operator meaning it checks whether the pattern before or after the or symbol is present in the string or not.

For example –

a|b will match any string that contains a or b such as acd, bcd, abcd, etc.

? – Question Mark Question mark(?) checks if the string before the question mark in the regex occurs at least once or not at all.

For example –

ab?c will be matched for the string ac, acb, dabc but will not be matched for abbc because there are two b. Similarly, it will not be matched for abdc because b is not followed by c.

Star(*) Star (*) symbol matches zero or more occurrences of the regex preceding the * symbol.

For example –

ab*c will be matched for the string ac, abc, abbbc, dabc, etc. but will not be matched for abdc because b is not followed by c.

+ – Plus Plus (+) symbol matches one or more occurrences of the regex preceding the + symbol.

For example –

ab+c will be matched for the string abc, abbc, dabc, but will not be matched for ac, abdc because there is no b in ac and b is not followed by c in abdc.

{m, n} – Braces Braces match any repetitions preceding regex from m to n both inclusive.

For example –

a{2, 4} will be matched for the string aaab, baaaac, gaad, but will not be matched for strings like abc, bc because there is only one a or no a in both the cases.

Group Group symbol is used to group sub-patterns.

For example –

(a|b)cd will match for strings like acd, bcd, gacd, etc.

Special Sequences

Special sequences do not match for the actual character in the string instead it tells the specific location in the search string where the match must occur. It makes it easier to write commonly used patterns.

- **\A**: Matches if the string begins with the given character
- **\b**: Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.
- **\B**: It is the opposite of the \b i.e. the string should not start or end with the given regex.
- **\d**: Matches any decimal digit, this is equivalent to the set class [0-9]
- **\D**: Matches any non-digit character, this is equivalent to the set class [^0-9]
- **\s**: Matches any whitespace character.
- **\S**: Matches any non-whitespace character
- **\w**: Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].
- **\W**: Matches any non-alphanumeric character.
- **\Z**: Matches if the string ends with the given regex

Regex Module in Python

Python has a module named re that is used for regular expressions in Python. We can import this module by using the import statement.

re.findall()

Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.

Example: Finding all occurrences of a pattern

In [52]:
```
# A Python program to demonstrate working of findall()
import re

# A sample text string where regular expression is searched.
string = """Hello my Number is 123456789 and
            my friend's number is 987654321"""

# A sample regular expression to find digits.
regex = '\d+'

match = re.findall(regex, string)
print(match)
```

```
['123456789', '987654321']
```

re.compile()

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

In [53]:
```
# Module Regular Expression is imported using __import__().
import re

# compile() creates regular expression character class [a-e],
# which is equivalent to [abcde].
# class [abcde] will match with string with 'a', 'b', 'c', 'd', 'e'.
p = re.compile('[a-e]')

# findall() searches for the Regular Expression and return a list upon finding
print(p.findall("Aye, said Mr. Gibenson Stark"))
```

```
['e', 'a', 'd', 'b', 'e', 'a']
```

In [54]:
```
#Set class [\s,.] will match any whitespace character,  ‘,’,  or, ‘.’ .
import re

# \d is equivalent to [0-9].
p = re.compile('\d')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))

# \d+ will match a group on [0-9], group of one or greater size
p = re.compile('\d+')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
```

```
['1', '1', '4', '1', '8', '8', '6']
['11', '4', '1886']
```

In [55]:
```
import re

# \w is equivalent to [a-zA-Z0-9_].
p = re.compile('\w')
print(p.findall("He said * in some_lang."))

# \w+ matches to group of alphanumeric character.
p = re.compile('\w+')
print(p.findall("I went to him at 11 A.M., he \
said *** in some_language."))

# \W matches to non alphanumeric characters.
p = re.compile('\W')
print(p.findall("he said *** in some_language."))
```

```
['H', 'e', 's', 'a', 'i', 'd', 'i', 'n', 's', 'o', 'm', 'e', '_', 'l', 'a',
'n', 'g']
['I', 'went', 'to', 'him', 'at', '11', 'A', 'M', 'he', 'said', 'in', 'some_la
nguage']
[' ', ' ', '*', '*', '*', ' ', ' ', '.']
```

In [56]:
```
import re

# '*' replaces the no. of occurrence of a character.
p = re.compile('ab*')
print(p.findall("ababbaabbb"))
```

```
['ab', 'abb', 'a', 'abbb']
```

re.split()

Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.

Syntax :

re.split(pattern, string, maxsplit=0, flags=0)

The First parameter, pattern denotes the regular expression, string is the given string in which pattern will be searched for and in which splitting occurs, maxsplit if not provided is considered to be zero ‘0’, and if any nonzero value is provided, then at most that many splits occur.

If maxsplit = 1, then the string will split once only, resulting in a list of length 2. The flags are very useful and can help to shorten code, they are not necessary parameters,

eg: flags = re.IGNORECASE, in this split, the case, i.e. the lowercase or the uppercase will be ignored

In [57]:
```
from re import split

# '\W+' denotes Non-Alphanumeric Characters or group of characters Upon findin
# or whitespace ' ', the split(), splits the string from that point
print(split('\W+', 'Words, words , Words'))
print(split('\W+', "Word's words Words"))

# Here ':', ' ' ,',' are not AlphaNumeric thus,the point where splitting occur
print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))

# '\d+' denotes Numeric Characters or group of characters Splitting occurs at
# '11', '02' only
print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

```
['Words', 'words', 'Words']
['Word', 's', 'words', 'Words']
['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']
['On ', 'th Jan ', ', at ', ':', ' AM']
```

In [58]:
```
import re

# Splitting will occurs only once, at '12', returned list will have length 2
print(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))

# 'Boy' and 'boy' will be treated same when flags = re.IGNORECASE
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here', flags=re.IGNORECASE))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here'))
```

```
['On ', 'th Jan 2016, at 11:02 AM']
['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']
['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']
```

re.sub()

The ‘sub’ in the function stands for SubString, a certain regular expression pattern is searched in the given string(3rd parameter), and upon finding the substring pattern is replaced by repl(2nd parameter), count checks and maintains the number of times this occurs.

Syntax:

re.sub(pattern, repl, string, count=0, flags=0)

In [59]:
```
import re

# Regular Expression pattern 'ub' matches the
# string at "Subject" and "Uber". As the CASE
# has been ignored, using Flag, 'ub' should
# match twice with the string Upon matching,
# 'ub' is replaced by '~*' in "Subject", and
# in "Uber", 'Ub' is replaced.
print(re.sub('ub', '~*', 'Subject has Uber booked already',
             flags=re.IGNORECASE))

# Consider the Case Sensitivity, 'Ub' in "Uber", will not be replaced.
print(re.sub('ub', '~*', 'Subject has Uber booked already'))

# As count has been given value 1, the maximum times replacement occurs is 1
print(re.sub('ub', '~*', 'Subject has Uber booked already',
             count=1, flags=re.IGNORECASE))

# 'r' before the pattern denotes RE, \s is for start and end of a String.
print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam',
             flags=re.IGNORECASE))
```

```
S~*ject has ~*er booked already
S~*ject has Uber booked already
S~*ject has Uber booked already
Baked Beans & Spam
```

re.subn()

subn() is similar to sub() in all ways, except in its way of providing output. It returns a tuple with a count of the total of replacement and the new string rather than just the string.

Syntax:

re.subn(pattern, repl, string, count=0, flags=0)

In [60]:
```
import re

print(re.subn('ub', '~*', 'Subject has Uber booked already'))

t = re.subn('ub', '~*', 'Subject has Uber booked already',
             flags=re.IGNORECASE)
print(t)
print(len(t))

# This will give same output as sub() would have
print(t[0])
```

```
('S~*ject has Uber booked already', 1)
('S~*ject has ~*er booked already', 2)
2
S~*ject has ~*er booked already
```

re.escape()

Returns string with all non-alphanumerics backslashed, this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

Syntax:

re.escape(string)

In [61]:
```
import re

# escape() returns a string with BackSlash '\',
# before every Non-Alphanumeric Character
# In 1st case only ' ', is not alphanumeric
# In 2nd case, ' ', caret '^', '-', '[]', '\'
# are not alphanumeric
print(re.escape("This is Awesome even 1 AM"))
print(re.escape("I Asked what is this [a-9], he said \t ^WoW"))
```

```
This\ is\ Awesome\ even\ 1\ AM
I\ Asked\ what\ is\ this\ \[a\-9\],\ he\ said\ \        \ \^WoW
```

re.search()

This method either returns None (if the pattern doesn't match), or a re.MatchObject contains information about the matching part of the string. This method stops after the first match, so this is best suited for testing a regular expression more than extracting data.

Example: Searching for an occurrence of the pattern

In [62]:
```
# A Python program to demonstrate working of re.match().
import re

# Lets use a regular expression to match a date string
# in the form of Month name followed by day number
regex = r"([a-zA-Z]+) (\d+)"

match = re.search(regex, "I was born on June 24")

if match != None:

# We reach here when the expression "([a-zA-Z]+) (\d+)" matches the date strin
# This will print [14, 21), since it matches at index 14 and ends at 21.
    print ("Match at index %s, %s" % (match.start(), match.end()))

# We use group() method to get all the matches and
# captured groups. The groups contain the matched values.
# In particular:
# match.group(0) always returns the fully matched string
# match.group(1) match.group(2), ... return the capture
# groups in order from left to right in the input string
# match.group() is equivalent to match.group(0)

# So this will print "June 24"
    print ("Full match: %s" % (match.group(0)))

# So this will print "June"
    print ("Month: %s" % (match.group(1)))

# So this will print "24"
    print ("Day: %s" % (match.group(2)))

else:
    print ("The regex pattern does not match.")
```

```
Match at index 14, 21
Full match: June 24
Month: June
Day: 24
```

Match Object

A Match object contains all the information about the search and the result and if there is no match found then None will be returned. Let's see some of the commonly used methods and attributes of the match object.

Getting the string and the regex

match.re attribute returns the regular expression passed and match.string attribute returns the string passed.

Example: Getting the string and the regex of the matched object

In [65]:
```
import re

s = "Welcome to CMR Technical Campus"

# here x is the match object
res = re.search(r"\bC", s)

print(res.re)
print(res.string)
```

```
re.compile('\\bC')
Welcome to CMR Technical Campus
```

Getting index of matched object

- **start()** method returns the starting index of the matched substring
- **end()** method returns the ending index of the matched substring
- **span()** method returns a tuple containing the starting and the ending index of the matched substring

Example: Getting index of matched object

In [66]:
```
import re

s = "Welcome to CMR Technical Campus"

# here x is the match object
res = re.search(r"\bCam", s)

print(res.start())
print(res.end())
print(res.span())
```

```
25
28
(25, 28)
```

Getting matched substring

group() method returns the part of the string for which the patterns match.

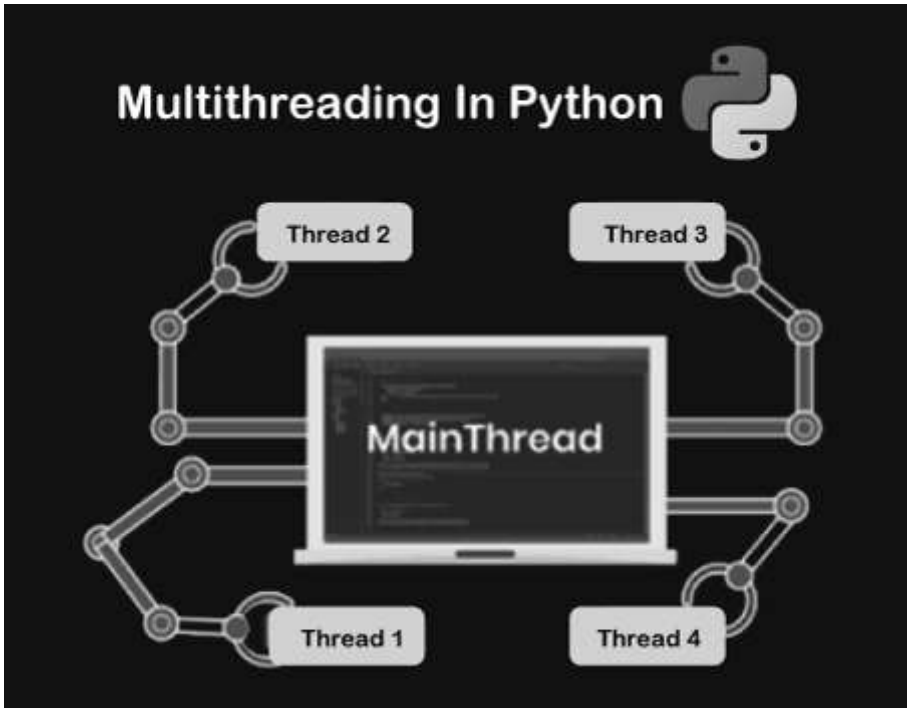
Example: Getting matched substring

In [8]:
```
import re

s = "Welcome to CMR Technical Campus"

# here x is the match object
res = re.search(r"\D{2} C", s)

print(res.group())
```

```
to C
```

Multithreaded Programming

A program or process's smallest unit is called a thread, and it can run on its own or as part of a schedule set by the Operating System. Multitasking in a computer system is achieved by dividing a process into threads by an operating system. A string is a lightweight cycle that guarantees the execution of the interaction independently on the framework. When multiple processors are running on a program in Python 3, each processor runs simultaneously to carry out its own tasks.



Multithreading is a stringing procedure in Python programming to run various strings simultaneously by quickly exchanging between strings with a central processor help (called setting exchanging). In addition, it makes it possible to share its data space with the primary threads of a process, making it easier than with individual processes to share information and communicate with other threads. The goal of multithreading is to complete multiple tasks at the same time, which improves application rendering and performance.

Note: The Python Global Interpreter Lock (GIL) allows running a single thread at a time, even the machine has multiple processors.

Benefits of Using Python for Multithreading

The following are the advantages of using Python for multithreading:

- It guarantees powerful usage of PC framework assets.
- Applications with multiple threads respond faster.
- It is more cost-effective because it shares resources and its state with sub-threads (child).
- It makes the multiprocessor engineering more viable because of closeness.
- By running multiple threads simultaneously, it cuts down on time.
- To store multiple threads, the system does not require a lot of memory.

When to use Multithreading in Python?

It is an exceptionally valuable strategy for efficient and working on the presentation of an application. Programmers can run multiple subtasks of an application at the same time by using multithreading. It lets threads talk to the same processor and share resources like files, data, and memory. In addition, it makes it easier for the user to continue running a program even when a portion of it is blocked or too long.

How to achieve multithreading in Python?

There are two main modules of multithreading used to handle threads in Python.

- The thread module
- The threading module

Thread modules

It is started with Python 3, designated as obsolete, and can only be accessed with **_thread** that supports backward compatibility.

Syntax:

```
thread.start_new_thread ( function_name, args[, kwargs] )
```

To implement the thread module in Python, we need to import a **thread** module and then define a function that performs some action by setting the target with a variable.

In [20]:
```
import _thread
import time

# Define a function for the thread
def print_time( threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print("%s: %s" % ( threadName, time.ctime(time.time()) ))

# Create two threads as follows
try:
    _thread.start_new_thread( print_time, ("Thread-1", 2, ) )
    _thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
    print("Error: unable to start thread")

while 1:
    pass
```

```
Thread-1: Fri Jun 30 19:07:37 2023
Thread-2: Fri Jun 30 19:07:39 2023
Thread-1: Fri Jun 30 19:07:39 2023
Thread-1: Fri Jun 30 19:07:41 2023
Thread-2: Fri Jun 30 19:07:43 2023
Thread-1: Fri Jun 30 19:07:43 2023
Thread-1: Fri Jun 30 19:07:45 2023
Thread-2: Fri Jun 30 19:07:47 2023
Thread-2: Fri Jun 30 19:07:51 2023
Thread-2: Fri Jun 30 19:07:55 2023

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[20], line 20
     17     print("Error: unable to start thread")
     19 while 1:
---> 20     pass

KeyboardInterrupt:
```

In [10]:
```
import _thread # import the thread module
import time # import time module

def cal_sqre(num): # define the cal_sqre function
    print(" Calculate the square root of the given number")
    for n in num:
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(' Square is : ', n * n)

def cal_cube(num): # define the cal_cube() function
    print(" Calculate the cube of  the given number")
    for n in num:
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(" Cube is : ", n * n *n)

arr = [4, 5, 6, 7, 2] # given array

t1 = time.time() # get total time to execute the functions
cal_sqre(arr) # call cal_sqre() function
cal_cube(arr) # call cal_cube() function

print(" Total time taken by threads is :", time.time() - t1) # print the total
```

```
 Calculate the square root of the given number
 Square is :  16
 Square is :  25
 Square is :  36
 Square is :  49
 Square is :  4
 Calculate the cube of  the given number
 Cube is :  64
 Cube is :  125
 Cube is :  216
 Cube is :  343
 Cube is :  8
 Total time taken by threads is : 3.0379602909088135
```

Threading Module

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.

The threading module exposes all the methods of the thread module and provides some additional methods −

- **threading.activeCount()** − Returns the number of thread objects that are active.
- **threading.currentThread()** − Returns the number of thread objects in the caller's thread control.
- **threading.enumerate()** − Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the Thread class that implements threading. The methods provided by the Thread class are as follows −

- **run()** − The run() method is the entry point for a thread.
- **start()** − The start() method starts a thread by calling the run method.
- **join([time])** − The join() waits for threads to terminate.
- **isAlive()** − The isAlive() method checks whether a thread is still executing.
- **getName()** − The getName() method returns the name of a thread.
- **setName()** − The setName() method sets the name of a thread.

Creating Thread Using Threading Module

To implement a new thread using the threading module, you have to do the following −

- Define a new subclass of the Thread class.
- Override the **init**(self [,args]) method to add additional arguments.
- Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the **start()**, which in turn calls **run()** method.

In [16]:
```
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print("Starting " + self.name)
        print_time(self.name, 5, self.counter)
        print("Exiting " + self.name)

def print_time(threadName, counter, delay):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

print("Exiting Main Thread")
```

```
Starting Thread-1Starting Thread-2
Exiting Main Thread

Thread-1: Fri Jun 30 18:58:33 2023
Thread-2: Fri Jun 30 18:58:34 2023
Thread-1: Fri Jun 30 18:58:34 2023
Thread-1: Fri Jun 30 18:58:35 2023
Thread-2: Fri Jun 30 18:58:36 2023
Thread-1: Fri Jun 30 18:58:36 2023
Thread-1: Fri Jun 30 18:58:37 2023
Exiting Thread-1
Thread-2: Fri Jun 30 18:58:38 2023
Thread-2: Fri Jun 30 18:58:40 2023
Thread-2: Fri Jun 30 18:58:42 2023
Exiting Thread-2
```

Global Interpreter Lock (GIL)

Python Global Interpreter Lock (GIL) is a type of process lock which is used by python whenever it deals with processes. Generally, Python only uses only one thread to execute the set of written statements. This means that in python only one thread will be executed at a time. The performance of the single-threaded process and the multi-threaded process will be the same in python and this is because of GIL in python. We can not achieve multithreading in python because we have global interpreter lock which restricts the threads and works as a single thread.

Python has something that no other language has that is a reference counter. With the help of the reference counter, we can count the total number of references that are made internally in python to assign a value to a data object. Due to this counter, we can count the references and when this count reaches to zero the variable or data object will be released automatically.

In [21]:
```
# Python program showing use of reference counter
import sys

cmr_var = "CMR"
print(sys.getrefcount(cmr_var))

string_gfg = cmr_var
print(sys.getrefcount(string_gfg))
```

```
3
4
```

- This reference counter variable needed to be protected, because sometimes two threads increase or decrease its value simultaneously by doing that it may lead to memory leaked so in order to protect thread we add locks to all data structures that are shared across threads but sometimes by adding locks there exists a multiple locks which lead to another problem that is deadlock. In order to avoid memory leaked and deadlocks problem, we used single lock on the interpreter that is Global Interpreter Lock(GIL).

In [ ]:
```

```