Advanced Python Programming

E box

A Project Report

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor Of Technology

Computer Science and Engineering

Submitted to

LOVELY PROFESSIONAL UNIVERSITY

Phagwara,Punjab

From 30/May/2022 to 05/July/2022

Submitted By
Name of Student : Chinthoti.Praveen
Registration No: 12009790

Student Declaration

To whom so ever it may concern

I hereby declare that I have completed my six weeks summer training at E box Learning from 30th May 2022 to 5th July 2022 under the guidance of E box amphisoft . I have declared that I have worked with full dedication during these six weeks of training and my learning outcomes fulfil the requirements of training for the award of degree of B. Tech CSE, Lovely Professional University, Phagwara.

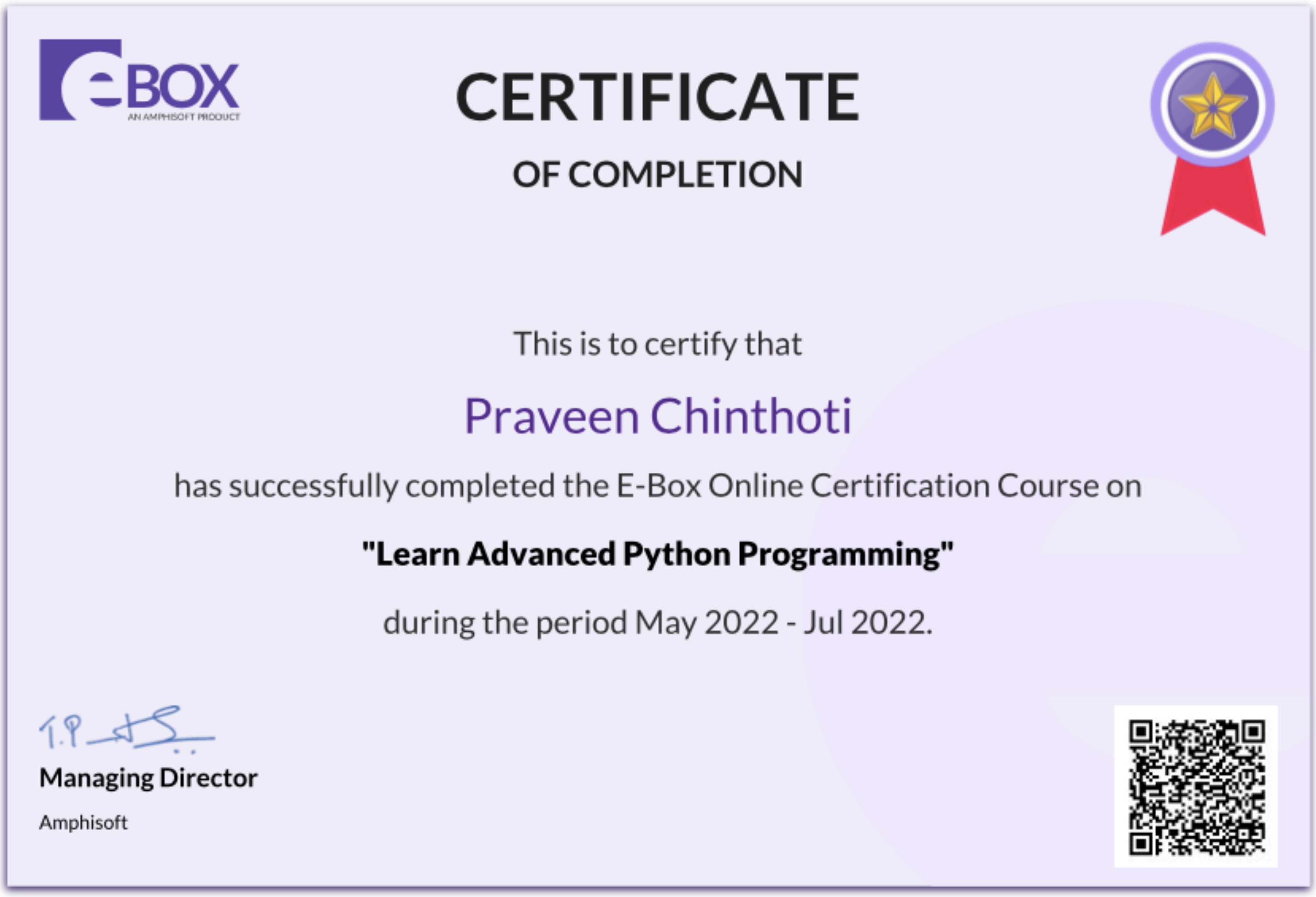
Name of Student:Chinthoti.Praveen
Registration no: 12009790

Date: 10th July 2022

ACKNOWLEDGEMENT

I would like to express my gratitude towards my University as well as E box for providing me the golden opportunity to do this wonderful summer training regarding python , which also helped me in doing a lot of homework and learning.As a result, I came to know about so many new things. So, I am really thank full to them. Moreover I would like to thank my friends who helped me a lot whenever I got stuck in some problem related to my course. I am really thankful to have such a good support of them as they always have my back whenever I need. Also,I would like to mention the support system and consideration of my parents who have always been there in my life to make me choose right thing and oppose the wrong. Without them I could never had learned and became a person who I am now. I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organization. I would like to extend my sincere thanks to all of them.

CERTIFICATE



CERTIFICATE

OF COMPLETION

This is to certify that

Praveen Chinthoti

has successfully completed the E-Box Online Certification Course on

"Learn Advanced Python Programming"

during the period May 2022 - Jul 2022.

[illegible]

Managing Director

Amphisoft

CONTENTS

Introduction to Advance Python

Python is currently the most famous programming language currently beingused in Variety of different fields be it the Complex field of Machine Learning, to Automation of some simple tasks using bot's made using Python and 3d Industry for addon development and game development for the use of AI enhancement. I started python Internship course on E box for the sole purpose to get acquainted with Python for it to use in 3D and Gamedevelopment and this course does introduces some the basic and core concepts used in Python in day-to-day basis regardless of the field you wantto work in.

This section of our tutorial has Python topics that are too difficult for beginners. This is also reflected in our image. The trail requires experience andbeginners with insufficient experience could easily get dizzy. Those who have successfully completed our beginner's tutorial or who have acquired sufficient Python experience elsewhere should not have any problems. As everywhere inour tutorial, we introduce the topics as gently as possible. This part of our Python tutorial is aimed at advanced programmer

1.Class and Objects

What is python class ?

A class is a blueprint for objects- one class for any number of objects of that type. You can also call it an abstract data type.
Interestingly, it contains no values itself, but it is like a prototype for objects.

Let’s see the Python classes explained in detail.

Python Class Syntax

1. Defining a Python Class
To define a class in python programming, we use the ‘class’ keyword. This is like we use ‘def’ to define a function in python.
And like a function, a Python3 class may have a docstring as well.
We can do this to write a couple of lines explaining what the class does. To focus on the syntax here, we will pass in a ‘pass’ statement in its body for now.

```
>>> class fruit:
        """
        This Python3 class creates instances of fruits
        """
```

Output

```
Pass
```

Some points on Python class:

- 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

OBJECTS :

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. Python is object-oriented programming language that stresses on objects i.e. it mainly emphasizes functions. Objects are basically an encapsulation of data variables and methods acting on that data into a single entity.

```
class Cars:
  def __init__(self, m, p):
    self.model = m
    self.price = p

Audi = Cars("R8", 100000)

print(Audi.model)
print(Audi.price)
```

Output

```
R8
100000
```

The first method __init__() is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')

num1 = ComplexNumber(2, 3)
num1.get_data()
num2 = ComplexNumber(5)
num2.attr = 10
print((num2.real, num2.imag, num2.attr))
print(num1.attr)
```

Output

```
2+3j
(5, 0, 10)
Traceback (most recent call last):
  File "main.py", line 14, in <module>
    print(num1.attr)
AttributeError: 'ComplexNumber' object has no attribute ‘attr
```

- The **getattr**(obj, name[, default]) − to access the attribute of object.
- The **hasattr**(obj,name) − to check if an attribute exists or not.
- The **setattr**(obj,name,value) − to set an attribute. If attribute does not exist, then it would be created.
- The **delatt**r(obj, name) − to delete an attribute.
-

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

- **__dict__** − Dictionary containing the class's namespace.
- **__doc__** − Class documentation string or none, if undefined.
- **__name__** − Class name.
- **__module__** − Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

```
class Employee:
  'Common base class for all employees'
  empCount = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1

  def displayCount(self):
   print "Total Employee %d" % Employee.empCount

  def displayEmployee(self):
    print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.
The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

Output

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Encapsulation in Python

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as **private variables.**

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

Protected members

Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow the convention by prefixing the name of the member by a single underscore "_".
Although the protected variable can be accessed out of the class as well as in the derived class(modified too in derived class), it is customary(convention not a rule) to not access the protected out the class body.
Note : The __init__ method is a constructor and runs as soon as an object of a class is instantiated.

```
class Base:
    def __init__(self):

        # Protected member
        self._a = 2

    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ",
            self._a)

        # Modify the protected variable:
        self._a = 3
        print("Calling modified protected member outside class:
",
            self._a)


obj1 = Derived()

obj2 = Base()

# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
```

Output

```
Calling protected member of base class:  2
Calling modified protected member outside class:  3
Accessing protected member of obj1:  3
Accessing protected member of obj2:  2
```

Private members

Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of **Private** instance variables that cannot be accessed except inside a class.
However, to define a private member prefix the member name with double underscore “__”.
Note: Python’s private and protected members can be accessed outside the class through python name mangling.

```
class Base:
   def __init__(self):
      self.a = "Praveen"
      self.__c = "Praveen"

# Creating a derived class
class Derived(Base):
   def __init__(self):

      # Calling constructor of
      # Base class
      Base.__init__(self)
      print("Calling private member of base class: ")
      print(self.__c)


# Driver code
obj1 = Base()
print(obj1.a)
```

Output

Praveen

2.Relationship with classes

we will look at different ways that classes can be related to each other. In Python, there are two main types of relationships between classes: *composition* and *inheritance*.

Association:

An **association** is defined as an organization of people with a common purpose and having a formal structure. It represents a binary relationship between two objects that describes an activity. It is a relationship between objects. For example, A doctor can be associated with multiple patients.

Aggregation:

An **aggregation** is a collection, or the gathering of things together. This relationship is represented by a “has a” relationship. In other words, aggregation is a group, body, or mass composed of many distinct parts or individuals For example, phone number list is an example of aggregation.

Composition

Composition is a concept that models a has a relationship. It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component. This relationship means that a Composite has a Component.

One to One Relationship

When a row in a table is related to only one role in another table and vice versa, we say that is a one to one relationship. This relationship can be created using Primary Key-Unique foreign key constraints.

iThis is where a row from one table can have multiple matching rows in another table this relationship is defined as a one-to-many relationship. This type of relationship can be created using Primary Key-Foreign key relationship

```
import pandas as pd

dataFrame1 = pd.DataFrame(
   {
      "Car": ['BMW', 'Lexus', 'Audi', 'Mustang', 'Bentley', 'Jaguar'],"Units":
[100, 150, 110, 80, 110, 90]
   }
)

print("DataFrame1 ...\n",dataFrame1)


dataFrame2 = pd.DataFrame(
   {
      "Car": ['BMW', 'Lexus', 'Tesla', 'Mustang', 'Mercedes',
'Jaguar'],"Reg_Price": [7000, 1500, 5000, 8000, 9000, 6000]

   }
)

print("\nDataFrame2 ...\n",dataFrame2)

mergedRes = pd.merge(dataFrame1, dataFrame2, validate ="one_to_one")
print("\nMerged dataframe with one-to-one relation...\n", mergedRes)
```

Output

```
DataFrame1 ...
Car  Units
0      BMW    100
1    Lexus    150
2     Audi    110
3  Mustang     80
4  Bentley    110
5   Jaguar     90
DataFrame2 ...
Car  Reg_Price
0       BMW       7000
1     Lexus       1500
2     Tesla       5000
3   Mustang       8000
4  Mercedes       9000
5    Jaguar       6000
Merged dataframe with one-to-one relation...
Car  Units  Reg_Price
0      BMW    100       7000
1    Lexus    150       1500
2  Mustang     80       8000
3   Jaguar     90       6000
```

One to Many Relationship

When a row in a table is related to only one role in another table

and vice versa, we say that is a one to one relationship. This

relationship can be created using Primary Key-Unique foreign key

constraints.

```
import pandas as pd
dataFrame1 = pd.DataFrame(
   {
      "Car": ['BMW', 'Lexus', 'Audi', 'Mustang', 'Bentley',
'Jaguar'],"Units": [100, 150, 110, 80, 110, 90]
   }
)

print("DataFrame1 ...\n",dataFrame1)

dataFrame2 = pd.DataFrame(
   {
      "Car": ['BMW', 'Lexus', 'Tesla', 'Mustang', 'Mercedes',
'Jaguar'],"Reg_Price": [7000, 1500, 5000, 8000, 9000, 6000]

   }
)

print("\nDataFrame2 ...\n",dataFrame2)


mergedRes = pd.merge(dataFrame1, dataFrame2, validate
="one_to_many")
```

```
print("\nMerged dataframe with one-to-many relation...\n",
mergedRes)
```

Output

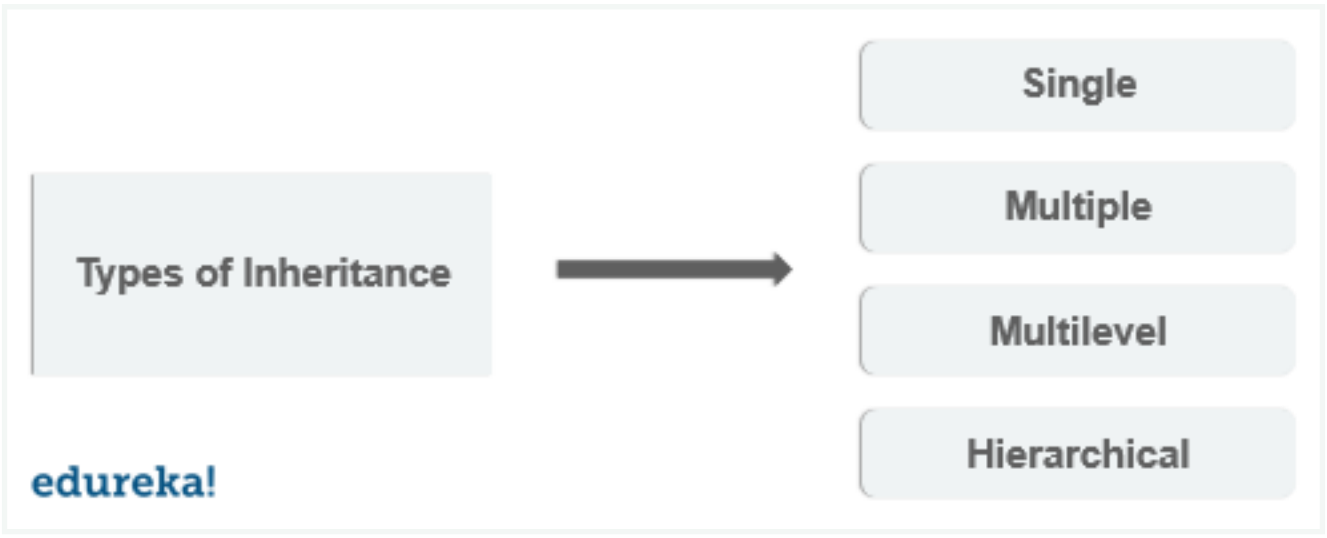
```
DataFrame1 ...
Car  Units
0      BMW    100
1    Lexus    150
2     Audi    110
3  Mustang     80
4  Bentley    110
5   Jaguar     90
DataFrame2 ...
Car  Reg_Price
0       BMW       7000
1     Lexus       1500
2     Tesla       5000
3   Mustang       8000
4  Mercedes       9000
5    Jaguar       6000
Merged dataframe with one-to-many relation...
Car  Units  Reg_Price
0      BMW    100       7000
1    Lexus    150       1500
2  Mustang     80       8000
3   Jaguar     90       6000
```

3.Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.
In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

Types Of Inheritance

Depending upon the number of child and parent classes involved, there are four types of inheritance in python.



Single Inheritance

When a child class inherits only a single parent class.

```
class Parent:
    def func1(self):
        print("this is function one")
class Child(Parent):
    def func2(self):
        print(" this is function 2 ")
ob = Child()
ob.func1()
ob.func2()
```

Output

```
this is function one
 this is function 2
```

Multiple Inheritance

When a child class inherits from more than one parent class.

```
class Parent:
   def func1(self):
       print("this is function 1")
class Parent2:
   def func2(self):
       print("this is function 2")
class Child(Parent , Parent2):
    def func3(self):
       print("this is function 3")

ob = Child()
ob.func1()
ob.func2()
ob.func3()
```

Output

```
this is function 1
this is function 2
this is function 3
```

Multilevel Inheritance

When a child class becomes a parent class for another child class.

```
class Parent:
    def func1(self):
        print("this is function 1")
class Child(Parent):

    def func2(self):
        print("this is function 2")
class Child2(Child):

    def func3("this is function 3")
ob = Child2()
ob.func1()
ob.func2()
ob.func3()
```

Python super()

The super() builtin returns a proxy object (temporary object of the superclass) that allows us to access methods of the base class.

```
class Animal(object):
  def __init__(self, animal_type):
    print('Animal Type:', animal_type)

class Mammal(Animal):
  def __init__(self):

    # call superclass
    super().__init__('Mammal')
    print('Mammals give birth directly')

dog = Mammal()
```

Output

```
Animal Type: Mammal
Mammals give birth directly
```

Use of super()

In Python, super() has two major use cases:

- Allows us to avoid using the base class name explicitly
- Working with Multiple Inheritance

super() with Single Inheritance

```
class Mammal(object):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')

class Dog(Mammal):
  def __init__(self):
    print('Dog has four legs.')
    super().__init__('Dog')

d1 = Dog()
```

Output

```
Dog has four legs.
Dog is a warm-blooded animal.
```

Here, we called the __init__() method of the Mammal class (from the Dog class) using code

```
super().__init__('Dog')
```

instead of

```
Mammal.__init__(self, ‘Dog')
```

Since we do not need to specify the name of the base class when we call its members, we can easily change the base class name (if we need to).

```
class Dog(CanidaeFamily):
  def __init__(self):
    print('Dog has four legs.')

    # no need to change this
    super().__init__(‘Dog’)
```

The super() builtin returns a proxy object, a substitute object that can call methods of the base class via delegation. This is called indirection (ability to reference base object with super())
Since the indirection is computed at the runtime, we can use different base classes at different times (if we need to).

super() with Multiple Inheritance

```
class Animal:
  def __init__(self, Animal):
    print(Animal, 'is an animal.');

class Mammal(Animal):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')
    super().__init__(mammalName)

class NonWingedMammal(Mammal):
  def __init__(self, NonWingedMammal):
    print(NonWingedMammal, "can't fly.")
    super().__init__(NonWingedMammal)

class NonMarineMammal(Mammal):
  def __init__(self, NonMarineMammal):
    print(NonMarineMammal, "can't swim.")
    super().__init__(NonMarineMammal)
```

```
class Dog(NonMarineMammal, NonWingedMammal):
  def __init__(self):
    print('Dog has 4 legs.');
    super().__init__('Dog')

d = Dog()
print('')
bat = NonMarineMammal('Bat')
```

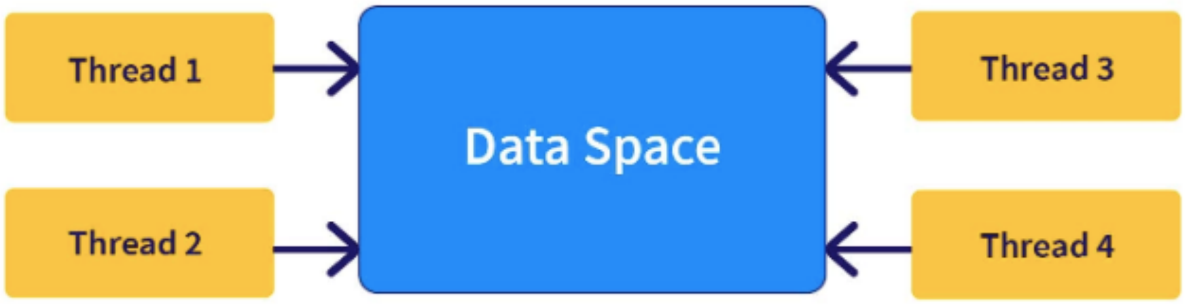
Output

```
Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.

Bat can't swim.
Bat is a warm-blooded animal.
Bat is an animal.
```

4.Multi-Threading

What is Python Multithreading?

Multithreading in Python is a popular technique that enables multiple tasks to be executed simultaneously. In simple words, the ability of a processor to execute multiple threads simultaneously is known as multithreading.
Python multithreading facilitates sharing data space and resources of multiple threads with the main thread. It allows efficient and easy communication between the threads.



Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than
processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

Benefits of Multithreading in python

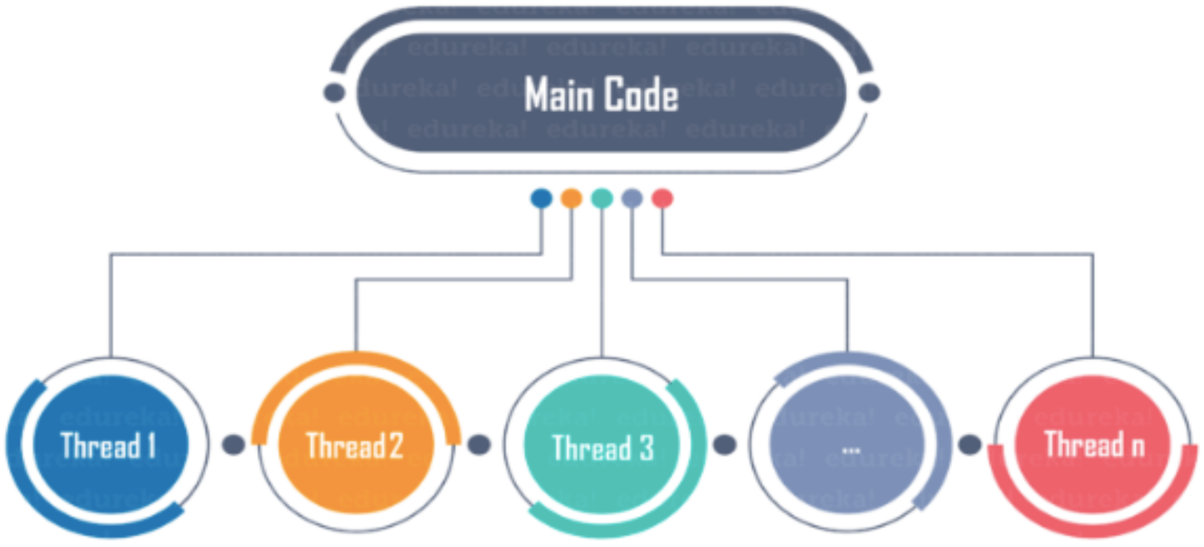
Following are the benefits to create a multithreaded application in Python, as follows:

1. It ensures effective utilization of computer system resources.
2. Multithreaded applications are more responsive.
3. It shares resources and its state with sub-threads (child) which makes it more economical.
4. It makes the multiprocessor architecture more effective due to similarity.
5. It saves time by executing multiple threads at the same time.
6. The system does not require too much memory to store multiple threads.

When to use Multithreading in Python?

It is a very useful technique for time-saving and improving the performance of an application. Multithreading allows the programmer to divide application **tasks** into sub-tasks and simultaneously run them in a program. It allows threads to communicate and share resources such as files, data, and memory to the same processor. Furthermore, it increases the user's responsiveness to continue running a program even if a part of the application is the length or blocked.

What is a thread?



A thread is basically an ***independent*** flow of execution. A single process can consist of multiple threads. Each thread in a program performs a particular task. *For Example,* when you are playing a game say FIFA on your PC, the game as a whole is a single process*,* but it consists of several threads responsible for playing the music, taking input from the user, running the opponent synchronously, etc. All these are separate threads responsible for carrying out these different tasks in the same program.
Every process has one thread that is always running. This is the main thread. This main thread actually creates the child thread objects. The child thread is also initiated by the main thread. I will show you all further in this article how to check the current running thread.
So with this, I hope you have clearly understood what is a thread. Moving on, let's see what is Multithreading in Python

Thread Lifecycle

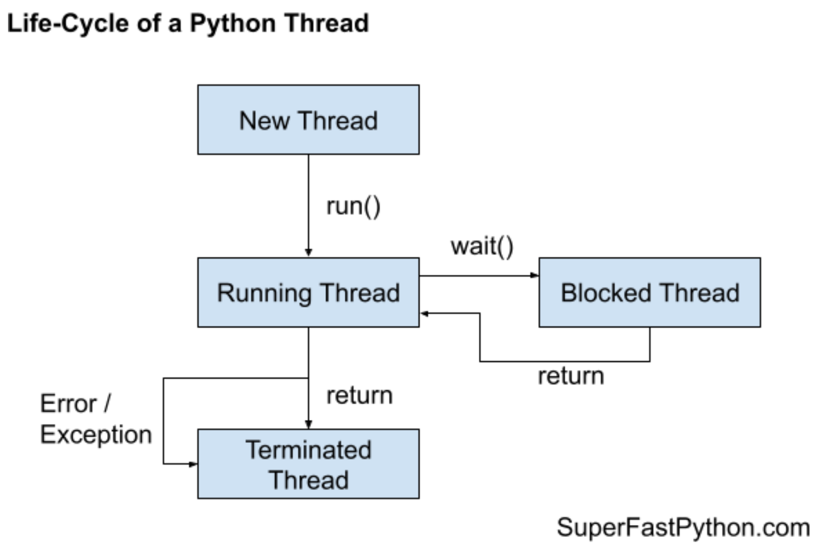
A thread in Python is represented as an instance of the threading. Thread class.

Once a thread is started, the Python interpreter will interface with the underlying operating system andrequest that a new native thread be created. The threading. Thread instance then provides a Python- based reference to this underlying native thread.

Each thread follows the same life cycle. Understanding the stages of this life cycle can help when gettingstarted with concurrent programming in Python.

For example:

· The difference between creating and starting a thread.
· The difference between run and start.
· The difference between blocked and terminated



How to implement Threads using Thread Class?

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 whenstarted.

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

```
import threading
import time

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
      # Get lock to synchronize threads
      threadLock.acquire()
      print_time(self.name, self.counter, 3)
      # Free lock to release next thread
      threadLock.release()

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

threadLock = threading.Lock()
threads = []

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

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

# Add threads to thread list
threads.append(thread1)
threads.append(thread2)

# Wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread”
```

Output

```
Starting Thread-1
Starting Thread-2
Thread-1: Thu Mar 21 09:11:28 2013
Thread-1: Thu Mar 21 09:11:29 2013
Thread-1: Thu Mar 21 09:11:30 2013
Thread-2: Thu Mar 21 09:11:32 2013
Thread-2: Thu Mar 21 09:11:34 2013
Thread-2: Thu Mar 21 09:11:36 2013
Exiting Main Thread
```

Thread API methods

This module-level function is used to open a new thread in the current process. The function takes a function objectas an argument. This function gets invoked on successful creation of the new thread. The span of this function corresponds to the lifespan of the thread. The thread may be blocked by calling sleep () function.

Synchronizing Threads in Python

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the Lock () method, which returns the new lock.
The acquire(blocking) method of the new lock object is used to force threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the lock.
If blocking is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.
The release () method of the new lock object is used to release the lock when it is no longer required.

```
import Queue
import threading
import time

exitFlag = 0

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

def process_data(threadName, q):
   while not exitFlag:
      queueLock.acquire()
         if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print "%s processing %s" % (threadName, data)
         else:
            queueLock.release()
         time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = Queue.Queue(10)
threads = []
threadID = 1

# Create new threads
for tName in threadList:
   thread = myThread(threadID, tName, workQueue)
   thread.start()
   threads.append(thread)
   threadID += 1

# Fill the queue
queueLock.acquire()
for word in nameList:
   workQueue.put(word)
queueLock.release()

# Wait for queue to empty
while not workQueue.empty():
   pass

# Notify threads it's time to exit
exitFlag = 1

# Wait for all threads to complete
for t in threads:
   t.join()
print "Exiting Main Thread"
```

Output

```
Starting Thread-1
Starting Thread-2
Starting Thread-3
Thread-1 processing One
Thread-2 processing Two
Thread-3 processing Three
Thread-1 processing Four
Thread-2 processing Five
Exiting Thread-3
Exiting Thread-1
Exiting Thread-2
Exiting Main Thread
```

Inter Thread communication by using Queue in python:
Queues Concept is the most enhanced Mechanism for inter thread communication and to share data between threads. Queue internally has Condition functionality which inturn includes the Lock functionality. Hence whenever we are using Queue we are not required to worry about Synchronization.

We can create Queue object as: q = queue. Queue ()
Two important methods of the queue module

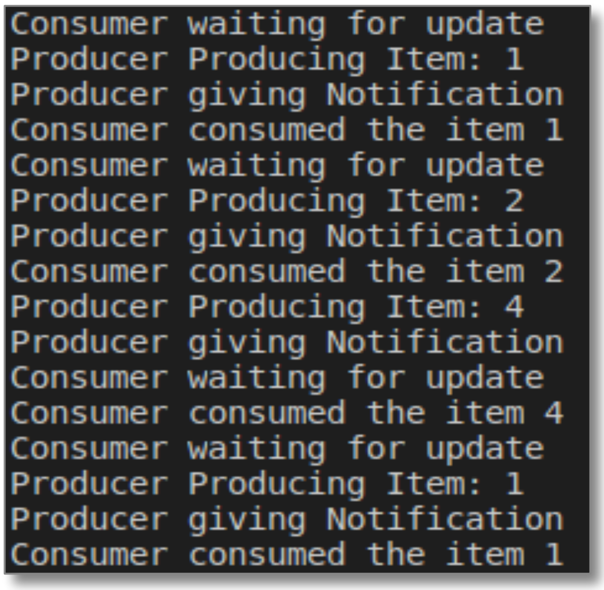
1. **put ():** To put an item into the queue.
2. **get ():** To remove and return an item from the queue.

Producer Thread can use the put () method to insert data in the queue. Whenever the producer method calls theput method, internally, first it will acquire the lock and then insert the data into the queue and then release the lock automatically. The put method also checks whether the queue is full or not and if queue is full then the Producer thread will enter into the waiting state by calling wait () method internally.

Consumer Thread can use the get () method to remove and get data from the queue. Whenever the consumer method calls the get method, internally, first it will acquire the lock and then get/ remove the data from the queue and then release the lock automatically. If the queue is empty, then the consumer thread will enter into the waiting state by calling wait () method internally. Once the queue is updated with data then the thread will be notified automatically.

```
from threading import *
import time
import random
import queue

items=[]
def produce(c):
    while True:
        item=random.randint(1,10) #Step 1.2
        print("Producer Producing Item:", item)
        q.put(item)
        print("Producer giving Notification")
        time.sleep(5)

def consume(c):
    while True:
        print("Consumer waiting for update")
        print("Consumer consumed the item", q.get())
        time.sleep(5)

q=queue.Queue()
t1=Thread(target=consume, args=(q,))
t2=Thread(target=produce, args=(q,))
```

Output



```
Consumer waiting for update
Producer Producing Item: 1
Producer giving Notification
Consumer consumed the item 1
Consumer waiting for update
Producer Producing Item: 2
Producer giving Notification
Consumer consumed the item 2
Producer Producing Item: 4
Producer giving Notification
Consumer waiting for update
Consumer consumed the item 4
Consumer waiting for update
Producer Producing Item: 1
Producer giving Notification
Consumer consumed the item 1
```

5.Exception Handling

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception. We will see what an exception is. Also, we will see the difference between a syntax error and an exception in this tutorial. Following that, we will learn about trying and except blocks and how to raise exceptions and make assertions. After that, we will see the Python exceptions list.

What is an Exception?

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. Alist of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **Name Error:** It occurs when a name is not found. It may be local or global.
3. **Indentation Error:** If incorrect indentation is given.
4. **IO Error:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

```
a,b=1,0
try:
   print(a/b)
except:
   print("You can't divide by 0")
print("Will this be printed?")
```

Output

```
You can't divide by 0
Will this be printed?
```

Important Python Errors

Error Type	Description
ArithmeticError	ArithmeticError act as a base class for all arithmetic exceptions. It is raised for errors in arithmetic operations.
ImportError	ImportError is raised when you are trying to import a module which does not present. This kind of exception occurs if you have made typing mistake in the module name or the module which is not present in the standard path.

IndexError	An IndexErroris raised when you try to refer a sequence which is out of range.
KeyError	When a specific key is not found in a dictionary, a KeyError exception is raised.
NameError	A NameError is raised when a name is referred to in code which never exists in the local or global namespace.
ValueError	Value error is raised when a function or built-in operation receives an argument which may be of correct type but does not have suitable value.
EOFerror	This kind of error raises when one of the built-in functions (input() or raw_input()) reaches an EOF condition without reading any data.

ZeroDivisonError	This type of error raised when division or module by zero takes place for all numeric types.
IOError-	This kind of error raised when an input/output operation fails.
syntaxError	SyntaxErrors raised when there is an error in Python syntax.
IndentationError	This error raised when indentation is not properly defined

Other Important Python Exceptions

Exception	**Description**
ArithmeticException	Arithmetic error, such as divide-by-zero.

ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment helps you to the array element of an incompatible type.
ClassCastException	Invalid cast
MlegalMonitorStateException	Illegal monitor operation, like waiting on an unlocked thread.
MlegalStateException	Environment or application is in wrong state.
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object which does not implement the Cloneable interface.
Illegal AccessException	Access to a class is denied.
InstantiationException	Occurs when you attempt to create an object of an interface or abstract class.
CloneNotSupportedException	Attempt to clone an object which does not implement the interface.

Error vs. Exceptions

All errors in Python are the unchecked type.	Exceptions include both checked and unchecked type.
Errors occur at run time which unknown to the compiler.	Exceptions can be recover by handling them with the help of try-catch blocks.
Errors are mostly caused by the environment in which an application is running.	The application itself causes exceptions.
Examples: OutofMemoryError	Examples: Checked Exceptions, SQL exception, NullPointerException,etc.

Classes involved in exception handling

Like other high-level languages, there are some exceptions in python also. When a problem occurs, it raises an exception. There are different kind of exceptions like **ZeroDivisionError, Assertion Error** etc. All exception classes are derived from the Base Exception class.
The code can run built in exceptions, or we can also raise these exceptions in the code. User can derive their ownexception from the **Exception** class, or from any other child class of **Exception** class.
The Base Exception is the base class of all other exceptions. User defined classes cannot be directly derived fromthis class, to derive user defied class, we need to use Exception class.

```
class LowAgeError (Exception):
def__init_(self):
pass


def__str_(self):
return 'The age must be greater than 18 years'


class Employee:
def__init_(self, name, age):
self.name = nameif age < 18:
raise LowAgeErrorelse:
self.age = age

def display(self):
print('The name of the employee: ' + self.name + ', Age: ' +
str(self.age) +' Years')

try:
e1 = Employee('Subhas', 25)e1.display()

e2 = Employee('Anupam', 12)e1.display()
```

```
except LowAgeError as e:
print ('Error Occurred: ' + str(e))
```

Output

```
The name of the employee: Subhas, Age: 25 Years
 Error Occurred the age must be greater than 18 years
```

Python Exception Handling Mechanism

Exception handling is managed by the following 5 keywords:

1. try
2. catch
3. finally
4. throw

Python Try Statement

A try statement includes keyword try, followed by a colon (:) and a suite of code in which exceptions may occur. It has one or more clauses.

During the execution of the try statement, if no exceptions occurred then, the interpreter ignores the exception handlers for that specific try statement.
In case, if any exception occurs in a try suite, the try suite expires and program control transfers to the matching except handler following the try suite.

Syntax:

```
try:
statement(s)
```

The catch Statement

Catch blocks take one argument at a time, which is the type of exception that it is likely to catch. These arguments may range from a specific type of exception which can be varied to a catch-all category of exceptions.
Rules for catch block:

- You can define a catch block by using the keyword catch
- Catch Exception parameter is always enclosed in parentheses
- It always represents the type of exception that catch block handles.
- An exception handling code is written between two {} curly braces.
- You can place multiple catch block within a single try block.
- You can use a catch block only after the try block.
- All the catch block should be ordered from subclass to superclass exception.

Example:

```
try
}
catch (ArrayIndexOutOfBoundsException e) {
System.err.printin("Caught first " + e.getMessage()); } catch (IOException
e) {
System.err.printin("Caught second " + e.getMessage());
}
```

Finally Statement in Python

Finally block always executes irrespective of an exception being thrown or not. The final keyword allows you to create a block of code that follows a try-catch block.
Finally, clause is optional. It is intended to define clean-up actions which should be that executed in all conditions.

```
try:
   raise KeyboardInterrupt
finally:
   print 'welcome, world!'
Output
Welcome, world!
KeyboardInterrupt
```

Finally, clause is executed before try statement.

User-Defined Exceptions

By inheriting classes from the typical built-in exceptions, Python also lets us design our customized exceptions.
Here is an illustration of a RuntimeError. In this case, a class that derives from RuntimeError is produced. Once an exception is detected, we can use this to display additional detailed information.
We raise a user-defined exception in the try block and then handle the exception in the except block. An example of the class EmptyError is created using the variable var.

```
class EmptyError( RuntimeError ):
    def __init__(self, argument):
        self.arguments = argument
```

Once the preceding class has been created, the following is how to raise an exception:

Code

```
var = " "
try:
    raise EmptyError( "The variable is empty" )
except (EmptyError, var):
    print( var.arguments )
```

Creating custom exceptions

In Python, users can define custom exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in Exception class. Most of the built-in exceptions are also derived from this class.

```
>>> class CustomError(Exception):
...     pass
...
>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError
>>> raise CustomError("An error occurred")
Traceback (most recent call last):
...
__main__.CustomError: An error occurred
```

Here, we have created a user-defined exception called CustomError which inherits from the Exception class. This new exception, like other exceptions, can be raised using the raise statement with an optional error message.

When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file. Many standard modules do this. They define their exceptions separately as exceptions.py or errors.py (generally but not always).

User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise. Most implementations declare a custom base class and derive others exception classes from this base class. This concept is made clearer in the following example.

```
class SalaryNotInRangeError (Exception):
"""Exception raised for errors in the input salary.
Attributes:
salary -- input salary which caused the error
message -- explanation of the error
"""
def __init__(self, salary, message="Salary is not in (5000, 15000) range"):
self.salary = salary
self.message = message
super().__init__(self.message)
salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
raise SalaryNotInRangeError(salary)
```

Output

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "<string>", line 17, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: Salary is not in (5000,
15000) range
```

6. Database Connection

What is Python Database Access?

A database is a collection of tables related to each other via columns. For most real-world projects, a database is a must.
We can use SQL (Structured Query Language) to create, access, and manipulate data.
We can also make use of normalization to avoid redundancy of data.
For database programming, Python supports many database servers-
MySQL, Oracle, PostgreSQL, SQLite, Sybase, Microsoft SQL Server, mSQL, Microsoft Access, and many more.
It also supports Data Query Statements, Data Definition Language (DDL), and Data Manipulation Language (DML).
The standard database interface for Python is Python DB-API. For that, we have the module MySQLdb for MySQL.
This is independent of database engines; so we can write Python scripts to access any database engine. However, this isn't compatible with Python 3.
So, in this Python Database Access tutorial, we use the module PyMySQL.

Advantages of Database Programming with Python

With Python, we have the following benefits:

- Platform-independent
- Faster and more efficient
- Portable
- Support for relational database systems

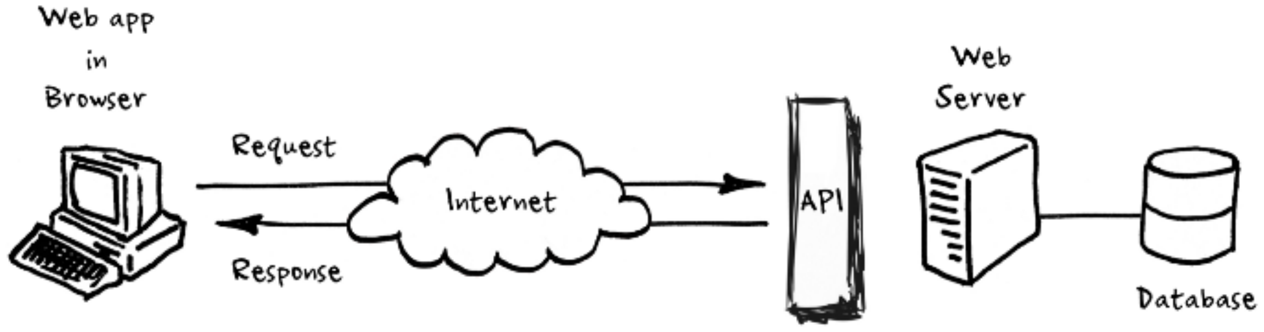
- Easy to migrate and port database application interfaces
- Support for SQL cursors
- It handles open and closed connections

What is Database?

A **database** is an organized collection of data, generally stored and accessed electronically from a computer system. Where databases are more complex they are often developed using formal design and modeling techniques.

The **database management system (DBMS)** is the software that interacts with end users, applications, and the database itself to capture and analyze the data. The DBMS software additionally encompasses the core facilities provided to administer the database. The sum total of the database, the DBMS and the associated applications can be referred to as a "database system". Often the term "database" is also used to loosely refer to any of the DBMS, the database system or an application associated with the database.
Now Lets understand the meaning of API

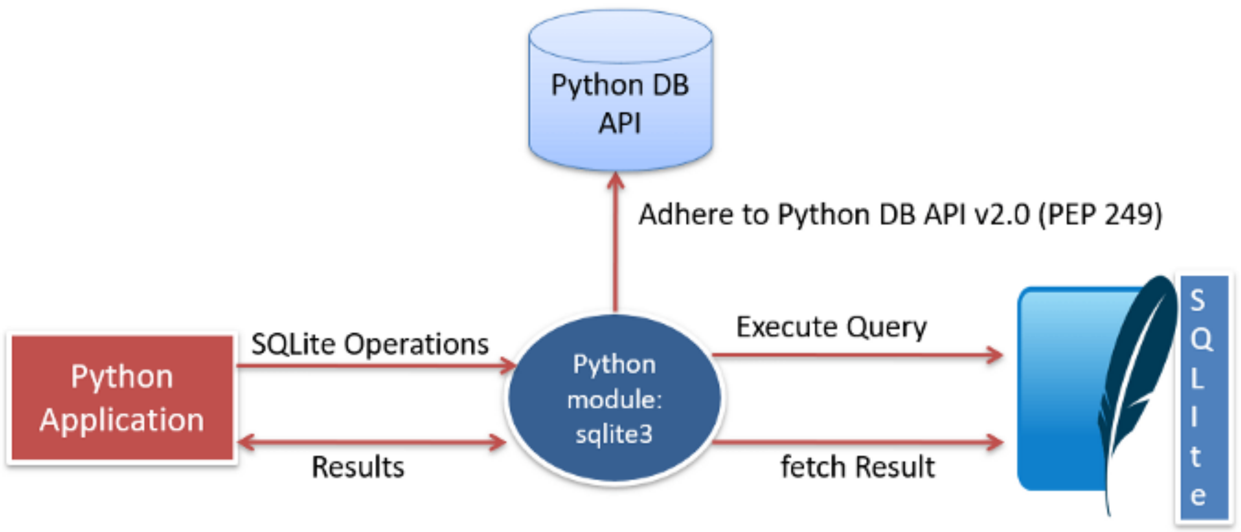
Meaning of API (Application Programming Interface)



(**A**pplication **P**rogramming **I**nterface) API is a language and message format used by an application program to communicate with the operating system or some other control program such as a database management system (DBMS) or communications protocol. APIs are implemented by writing function calls in the program, which provide the linkage to the required subroutine for execution.
Thus, an API implies that a driver or program module is available in the computer to perform the operation or that

software must be linked into the existing program to perform the tasks.

`sqlite3` — DB-API 2.0 interface for SQLite databases



SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle. The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. The database file format is cross-platform — you can freely copy a database between 32-bit and 64-bit systems or between big-endian and little-endian architectures.

SQLite natively supports the following types: `NULL`, `INTEGER`, `REAL`, `TEXT`, `BLOB`. The following Python types can thus be sent to SQLite without any problem:

Python type	SQLite type
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

Concepts of Python DB-API

- **Connection Objects** — Database Connections and Manage Transactions

cursor(*factory=Cursor*)-The cursor method accepts a single optional parameter *factory*. If supplied, this must be a callable returning an instance of Cursor or its subclasses.
commit()-This method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections. If you wonder why you don't see the data you've written to the database, please check you didn't forget to call this method.

rollback()-This method rolls back any changes to the database since the last call to commit().
close()-This closes the database connection. Note that this does not automatically call commit(). If you just close your database connection without calling commit() first, your changes will be lost!
2. **Cursor Objects** — Database Queries
execute(*sql*[, *parameters*])-Executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals).
executemany(*sql*, *seq_of_parameters*)-Executes an SQL command against all parameter sequences or mappings found in the sequence *seq_of_parameters*. The sqlite3 module also allows using an iterator yielding parameters instead of a sequence.
fetchone()-Fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
fetchmany(*size=cursor.arraysize*)-Fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available.
The **number of rows to fetch per call** is specified by the *size* parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.
Note there are **performance considerations** involved with the *size* parameter. For optimal performance, it is usually best to use the arraysize attribute. If the *size* parameter is used, then it is best for it to retain the same value from one fetchmany() call to the next.
fetchall()-Fetches all (remaining) rows of a query result, returning a list. Note that the cursor's arraysize attribute can affect the performance of this operation. An empty list is returned when no rows are available.

close()-Close the cursor now (rather than whenever `__del__` is called).
arraysize-Read/write attribute that controls the number of rows returned by `fetchmany()`. The default value is 1 which means a single row would be fetched per call.

How to Create Tables in Python Database?

Now let's take a look at all operations one by one, starting with creating a table.

```
>>> import pymysql
>>> db=pymysql.connect("localhost","ayushi","yourpassword","demo")
caching sha2: succeeded by fast path.
>>> cursor=db.cursor()
>>> cursor.execute("DROP TABLE IF EXISTS student") #This drops the table and replaces
it
>>> query="""CREATE TABLE student(
fname VARCHAR(20), lname VARCHAR(20),
age INT, enrolment_no VARCHAR(12))"""
>>> cursor.execute(query)
>>> db.close()
```

How to Insert a Record in Python Database?

Let's try inserting a record in 'student'.

```
>>> import pymysql
>>> db=pymysql.connect("localhost","ayushi","yourpassword","demo")
caching sha2: succeeded by fast path.
>>> cursor=db.cursor()
>>> query='INSERT INTO student VALUES("Ayushi","Sharma",22,"0812CS141028")'
>>> try:
            cursor.execute(query)
            db.commit() #Commit writing to the database
except:
            db.rollback() #Rollback the transaction if not complete
1
>>> db.close()
```

Let's check if this makes any changes to the database. In the command prompt:

```
mysql> select * from student;
+---------+--------+------+--------------+
| fname   | lname  | age  | enrolment_no |
+---------+--------+------+--------------+
| Ayushi  | Sharma |   22 | 0812CS141028 |
+---------+--------+------+--------------+
1 row in set (0.00 sec)
```

How to Read Records in Python Database?

Now how can we fetch values from a database?
Let’s take an example to fetch records of students from ‘student’ that are older than 22.
We have added another record for this purpose.

```
>>> import pymysql
>>> db=pymysql.connect("localhost","ayushi","yourpassword","demo")
caching sha2: succeeded by fast path.
>>> cursor=db.cursor()
>>> query="select * from student where age>22"
>>> try:
            cursor.execute(query)
            resultset=cursor.fetchall() #To fetch all records that satisfy
            for record in resultset:
                        fname=record[0]
                        lname=record[1]
                        age=record[2]
                        enrolment_no=record[3]
                        print(f"Student: {fname} {lname}; Enrolment: {enrolment_no}; Age: {age}")
except:
            print("Sorry, we encountered a problem")
1
Student: Megha Sharma; Enrolment: 0812CS141015; Age: 24
>>> db.close()
```

We have the following methods and attributes-

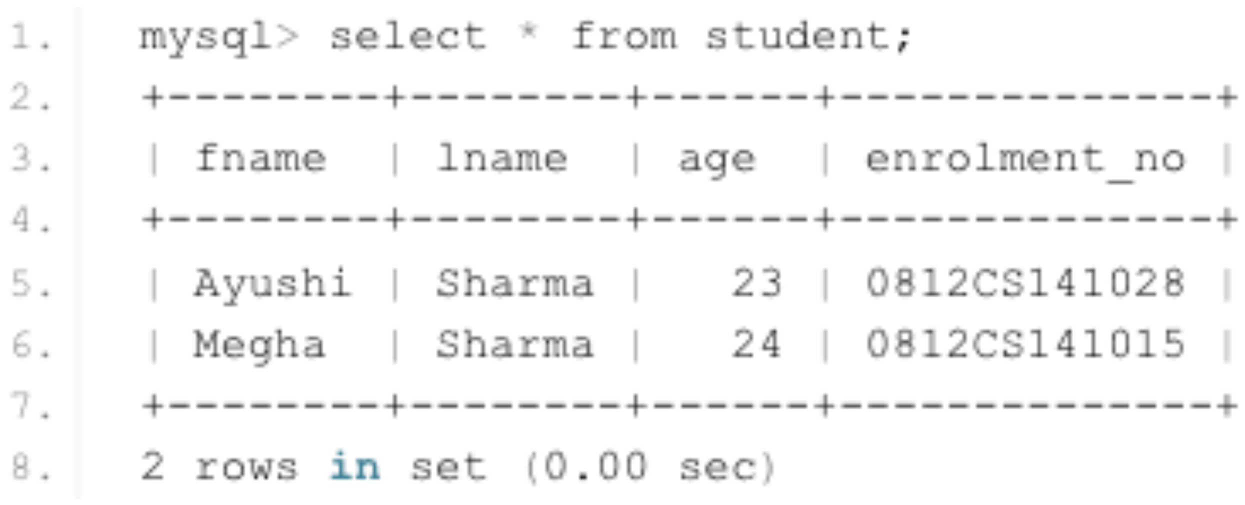
- **fetchone()**– This fetches the immediate next row from the result set of the query.
- **fetchall()**– This fetches the entire result set; it will exclude the records already extracted.
- **rowcount**– This is an attribute. It returns an integer denoting the number of records that a call to execute() affected.

How to Update Records in Python Database?

To update an existing record, we can simply use an SQL query for the same.

```
>>> import pymysql
>>> db=pymysql.connect("localhost","ayushi","yourpassword","demo")
caching sha2: succeeded by fast path.
>>> cursor=db.cursor()
>>> query="update student set age=age+1 where age<=22"
>>> try:
                cursor.execute(query)
                db.commit()
except:
                db.rollback()
1
>>> db.close()
```

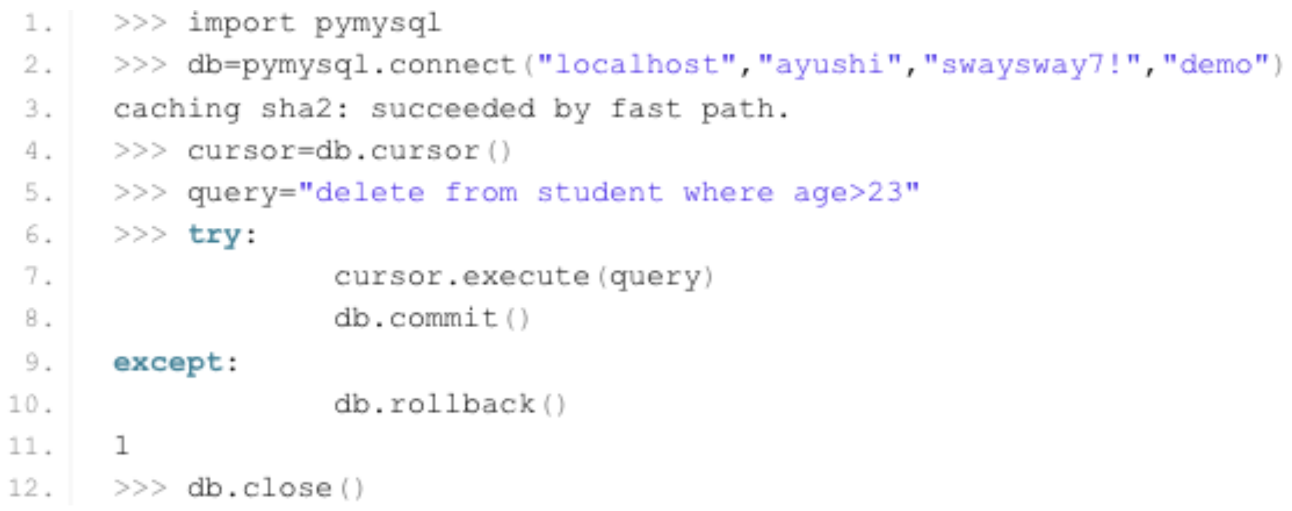
Let’s see if this has made any changes to the actual database. In your command prompt:



```
mysql> select * from student;
+--------+--------+------+---------------+
| fname  | lname  | age  | enrolment_no  |
+--------+--------+------+---------------+
| Ayushi | Sharma |   23 | 0812CS141028  |
| Megha  | Sharma |   24 | 0812CS141015  |
+--------+--------+------+---------------+
2 rows in set (0.00 sec)
```

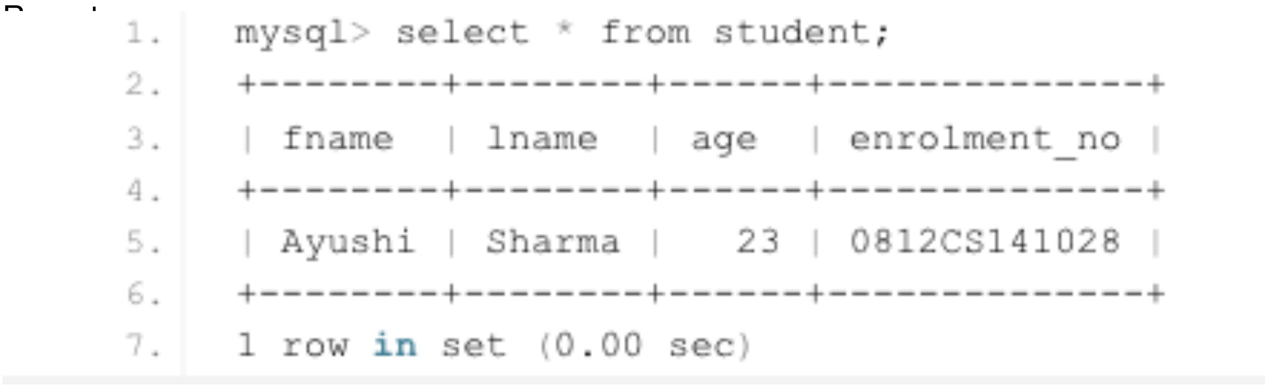
How to Delete Records in Python Database?

We can also delete records from a database using Python.



```
>>> import pymysql
>>> db=pymysql.connect("localhost","ayushi","swaysway7!","demo")
caching sha2: succeeded by fast path.
>>> cursor=db.cursor()
>>> query="delete from student where age>23"
>>> try:
            cursor.execute(query)
            db.commit()
except:
            db.rollback()
1
>>> db.close()
```

And in the command prompt:



```
mysql> select * from student;
+--------+--------+------+--------------+
| fname  | lname  | age  | enrolment_no |
+--------+--------+------+--------------+
| Ayushi | Sharma |   23 | 0812CS141028 |
+--------+--------+------+--------------+
1 row in set (0.00 sec)
```

Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.
The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

Sr.No.	Exception & Description
1	**Warning** Used for non-fatal issues. Must subclass StandardError.
2	**Error** Base class for errors. Must subclass StandardError.

3	**InterfaceError** Used for errors in the database module, not the database itself. Must subclass Error.
4	**DatabaseError** Used for errors in the database. Must subclass Error.
5	**DataError** Subclass of DatabaseError that refers to errors in the data.
6	**OperationalError** Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter.
7	**IntegrityError** Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
8	**InternalError** Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.

9	**ProgrammingError** Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.
10	**NotSupportedError** Subclass of DatabaseError that refers to trying to call unsupported functionality.

DB connection using SQL:

To create a connection between the MySQL database and Python, the connect () method of MySQL. Connector module is used. We pass the database details like Hostname, username, and the password in the method call, and then the method returns the connection object.
The following steps are required to connect SQL with Python:
Step 1: Download and Install the free MySQL database from here.
Step 2: After installing the MySQL database, open your Command prompt.
Step 3: Navigate your Command prompt to the location of PIP. Click here to see, how to install PIP?
Step 4: Now run the commands given below to download and install "MySQL Connector". Here, MySQL. Connector statement will help you to communicate with the MySQL database.

Step 5: To check if the installation was successful, or if you already installed “MySQL Connector”, go to your IDE, and run the given below code:
Step 6: Now to connect SQL with Python, run the code given below in your IDE.

What are SQL operators?

An SQL operator is a special word or character used to perform tasks. These tasks can be anything from complex comparisons, to basic arithmetic operations. Think of an SQL operator as similar to how the different buttons on a calculator function.
SQL operators are primarily used within the WHERE clause of an SQL statement. This is the part of the statementthat is used to filter data by a specific condition or conditions.
There are six types of SQL operators that we are going to cover: Arithmetic, Bitwise, Comparison, Compound,Logical and String.

7.Abstaract Class

A class is called an Abstract class if it contains one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and its abstract methods must be implemented by its subclasses.

Abstract base classes provide a way to define interfaces when other techniques like hasattr() would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by isinstance() and issubclass() functions. There are many built-in ABCs in Python. ABCs for Data structures like Iterator, Generator, Set, mapping etc. are defined in collections.abc module. The numbers module defines numeric tower which is a collection of base classes for numeric data types. The 'abc' module in Python library provides the infrastructure for defining custom abstract base classes.

'abc' works by marking methods of the base class as abstract. This is done by @absttractmethod decorator. A concrete class which is a sub class of such abstract base class then implements the abstract base by overriding its abstract methods

The abc module defines **ABCMeta** class which is a metaclass for defining abstract base class. Following example defines Shape class as an abstract base class using ABCMeta. The shape class has area() method decorated by abstractmethod.

A Rectangle class now uses above Shape class as its parent and implementing the abstract area() method. Since it is a concrete class, it can be instantiated and imlemented area() method can be called.

```
import abc
class Shape(metaclass=abc.ABCMeta):
   @abc.abstractmethod
   def area(self):
      pass
class Rectangle(Shape):
   def __init__(self, x,y):
      self.l = x
      self.b=y
   def area(self):
      return self.l*self.b
r = Rectangle(10,20)
print ('area: ',r.area())
```

Note the abstract base class may have more than one abstract methods. The child class must implement all of them failing which TypeError will be raised.

the abc module also defines **ABC** helper class which can be used instead of ABCMeta class in definition of abstract base class.

```
class Shape(abc.ABC):
@abc.abstractmethod
def area(self):
pass
```

Instead of subclassing from abstract base class, it can be registered as abstract base by register class decorator.

```
class Shape(abc.ABC):
   @abc.abstractmethod
   def area(self):
      pass
@Shape.register
class Rectangle():
   def __init__(self, x,y):
   self.l = x
   self.b=y
   def area(self):
      return self.l*self.b
```

Why Use Abstract Base Classes?

abstract classes are used to create a blueprint of our classes as they don't contain the method implementation. This is a very useful capability especially in situations where child classes should provide their own separate implementation. Also, in complex projects involving large teams and a huge codebase, It is fairly difficult to remember all the class names.

Importance of Abstract Classes

we define a blueprint for our classes using an abstract class. The importance of using abstract class in Python is that **if our subclasses don't follow that blueprint, Python will give an error**. Thus we can make sure that our classes follow the structure and implement all the abstract methods defined in our abstract class.

```
# Python program to define
# abstract class

from abc import ABC

class Polygon(ABC):
    # abstract method
    def sides(self):
        pass

class Triangle(Polygon):
    def sides(self):
        print("Triangle has 3 sides")

class Pentagon(Polygon):
    def sides(self):
        print("Pentagon has 5 sides")

class Hexagon(Polygon):
    def sides(self):
       print("Hexagon has 6 sides")

class square(Polygon):
    def sides(self):
        print("I have 4 sides")

# Driver code
t = Triangle()
t.sides()

s = square()
s.sides()
```

```
p = Pentagon()
p.sides()

k = Hexagon()
k.sides()
```

Output

```
Triangle has 3 sides
I have 4 sides
Pentagon has 5 sides
Hexagon has 6 sides
```

8.Streams

Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned. This session helps you get a clear understanding of streams and writers.File reading & writing forms an integral part of any application. From Simple logging to storing Objects & Content, file processing plays a key role.

StreamWriter

class asyncio.StreamWriter
Represents a writer object that provides APIs to write data to the IO stream.
It is not recommended to instantiate StreamWriter objects directly; use open_connection() and start_server() instead.

write(*data*)

¶

The method attempts to write the *data* to the underlying socket immediately. If that fails, the data is queued in an internal write buffer until it can be sent.
The method should be used along with the drain() method:

```
stream.write(data)
await stream.drain()
```

writelines(*data*)

The method writes a list (or any iterable) of bytes to the underlying socket immediately. If that fails, the data is queued in an internal write buffer until it can be sent.

The method should be used along with the `drain()` method:

```
stream.writelines(lines)
await stream.drain()
```

close()

The method closes the stream and the underlying socket.
The method should be used along with the wait_closed() method:

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

Return `True` if the underlying transport supports the `write_eof()` method, `False` otherwise.

write_eof()

Close the write end of the stream after the buffered write data is flushed.

transport

Return the underlying asyncio transport.

get_extra_info(*name*, *default=None*)

Access optional transport information;
see BaseTransport.get_extra_info() for details.
coroutine drain()
Wait until it is appropriate to resume writing to the stream.
Example:
writer.write(data)
await writer.drain()

This is a flow control method that interacts with the underlying IO write buffer. When the size of the buffer reaches the high watermark, *drain()* blocks until the size of the buffer is drained down to the low watermark and writing can be resumed. When there is nothing to wait for, the drain() returns immediately.

is_closing()

Return True if the stream is closed or in the process of being closed.

coroutine wait_closed()

Wait until the stream is closed.
Should be called after close() to wait until the underlying connection is closed.

```
# Opening a file
file1 = open('myfile.txt', 'w')
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]
s = "Hello\n"

# Writing a string to file
file1.write(s)

# Writing multiple strings
# at a time
file1.writelines(L)

# Closing file
file1.close()

# Checking if the data is
# written to file or not
file1 = open('myfile.txt', 'r')
print(file1.read())
file1.close()
```

Output

```
Hello
This is Delhi
This is Paris
This is London
```

```
# Creating a file
file1 = open("myfile.txt", "w")
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

# Writing data to a file
file1.write("Hello \n")
file1.writelines(L)
file1.close()  # to change file access modes

file1 = open("myfile.txt", "r+")

print("Output of Read function is ")
print(file1.read())
print()

# seek(n) takes the file handle to the nth
# bite from the beginning.
file1.seek(0)

print("Output of Readline function is ")
print(file1.readline())
print()

file1.seek(0)

# To show difference between read and readline
print("Output of Read(9) function is ")
print(file1.read(9))
print()
```

```
file1.seek(0)

print("Output of Readline(9) function is ")
print(file1.readline(9))
print()

file1.seek(0)

# readlines function
print("Output of Readlines function is ")
print(file1.readlines())
print()
file1.close()
```

Output

```
Output of Read function is
Hello
This is Delhi
This is Paris
This is London


Output of Readline function is
Hello


Output of Read(9) function is
Hello
Th

Output of Readline(9) function is
Hello


Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']
```

9.Lamada

What are lambda functions in Python?

In Python, an anonymous function is a function that is defined without a name.
While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.
Hence, anonymous functions are also called lambda functions.

How to use lambda Functions in Python?

A lambda function in python has the following syntax.

Syntax of Lambda Function in python

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Use of Lambda Function in python

We use lambda functions when we require a nameless function for a short period of time.
In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments).
Lambda functions are used along with built-in functions like filter(), map() etc.

Output

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Using lambda() Function with reduce()

The reduce() function in Python takes in a function and a list as an argument. The function is called with a lambda function and an iterable and a new reduced result is returned. This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the ***functools*** module.

```
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

Output

```
193
```

```
my_List = [ [3, 5, 8, 6], [23, 54, 12, 87], [1, 2, 4, 12, 5] ]
sort_List = lambda num : ( sorted(n) for n in num )
third_Largest = lambda num, func : [ l[ len(l) - 2] for l in
func(num)]
result = third_Largest( my_List, sort_List)

print( result )
```

Using Lambda Function with filter()

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True.

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

Output

```
[4, 6, 8, 12]
```

Using Lambda Function with map()

The map() function in Python takes in a function and a list as an argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

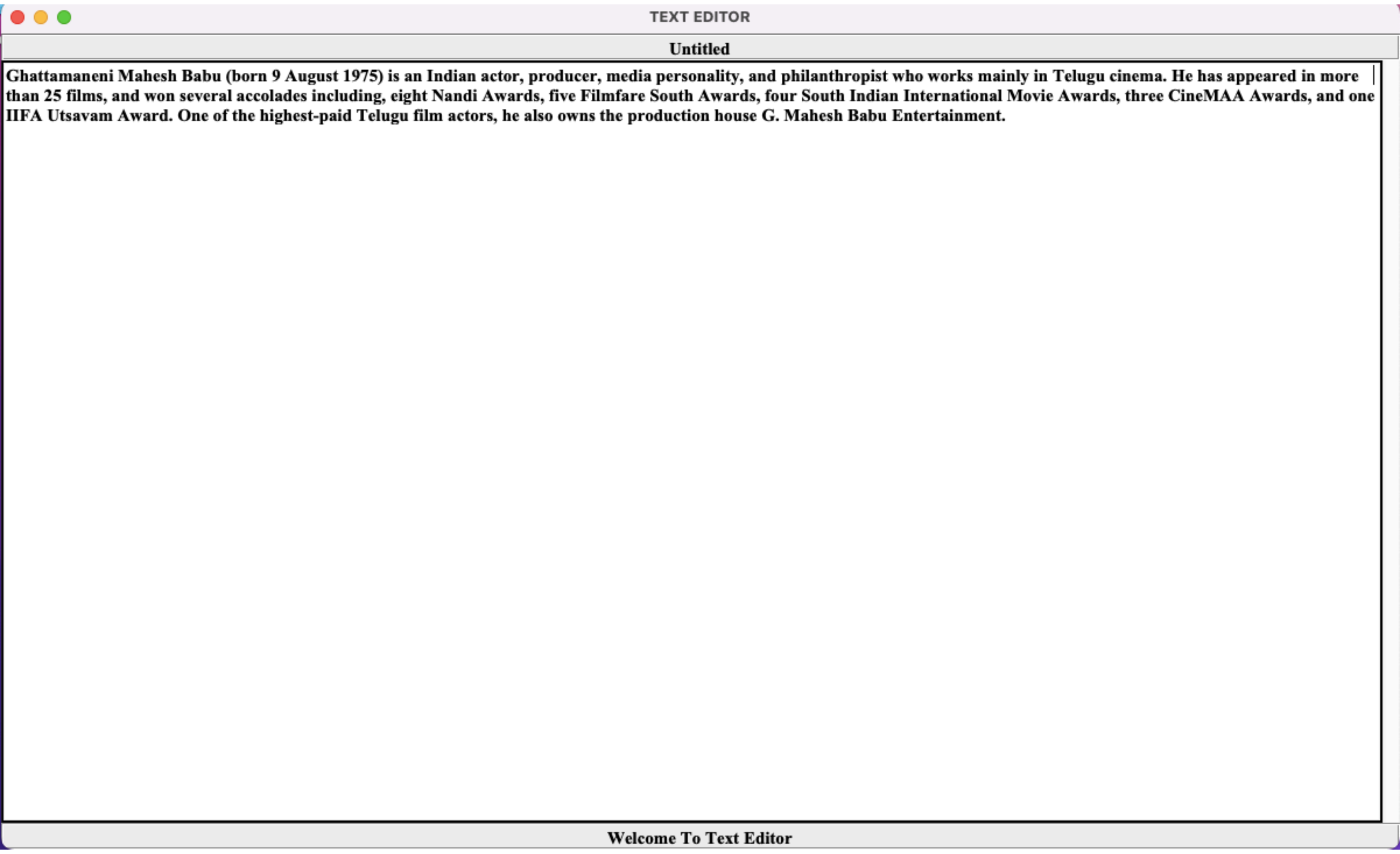
Output

```
[6, 54, 5]
```

MINI PROJECT

(TEXT EDITOR)

Text Editor in Python



Ghattamaneni Mahesh Babu (born 9 August 1975) is an Indian actor, producer, media personality, and philanthropist who works mainly in Telugu cinema. He has appeared in more than 25 films, and won several accolades including, eight Nandi Awards, five Filmfare South Awards, four South Indian International Movie Awards, three CineMAA Awards, and one IIFA Utsavam Award. One of the highest-paid Telugu film actors, he also owns the production house G. Mahesh Babu Entertainment.

About Text Editor

Text Editor is a simple application that helps a user to write texts and save it in a file. It is a basic beginners project but very interesting to make.

Project Details

This is a very interesting project using Python. Here we are going to create a text editor which will have several options like copy, paste, save, save as etc. We will try to include every possible option in it. While creating the project we will be using these modules and libraries –

- Tkinter Module
- Message box
- Filedilog

1. Importing libraries:

```
from tkinter import *
from tkinter import messagebox
from tkinter import filedialog
```

2. Text Editor Class

```
# Defining TextEditor Class
class TextEditor:


  def __init__(self,root):

    self.root = root

    self.root.title("TEXT EDITOR")

    self.root.geometry("1200x700+200+150")

    self.filename = None

    self.title = StringVar()

    self.status = StringVar()
```

3.Creating Title Bar

```
    self.titlebar = 
Label(self.root,textvariable=self.title,font=("times new 
roman",15,"bold"),bd=2,relief=GROOVE)

    self.titlebar.pack(side=TOP,fill=BOTH)

    self.settitle()
```

4.Creating Statusbar

```
    self.statusbar = 
Label(self.root,textvariable=self.status,font=("times new 
roman",15,"bold"),bd=2,relief=GROOVE)

    self.statusbar.pack(side=BOTTOM,fill=BOTH)

    self.status.set("Welcome To Text Editor")
```

5.Creating Menubar

```
    self.menubar = Menu(self.root,font=("times new 
roman",15,"bold"),activebackground="skyblue")

    self.root.config(menu=self.menubar)
```

6.Creating File Menu

```
    self.filemenu = Menu(self.menubar,font=("times new 
roman",12,"bold"),activebackground="skyblue",tearoff=0)


self.filemenu.add_command(label="New",accelerator="Ctrl
+N",command=self.newfile)


self.filemenu.add_command(label="Open",accelerator="Ctr
l+O",command=self.openfile)
```

```
self.filemenu.add_command(label="Save",accelerator="Ctrl
+S",command=self.savefile)

    self.filemenu.add_command(label="Save
As",accelerator="Ctrl+A",command=self.saveasfile)

    self.filemenu.add_separator()


self.filemenu.add_command(label="Exit",accelerator="Ctrl+
E",command=self.exit)

    self.menubar.add_cascade(label="File",
menu=self.filemenu)
```

7.Creating Edit Menu

```
    self.editmenu = Menu(self.menubar,font=("times new
roman",12,"bold"),activebackground="skyblue",tearoff=0)


self.editmenu.add_command(label="Cut",accelerator="Ctrl
+X",command=self.cut)


self.editmenu.add_command(label="Copy",accelerator="Ct
rl+C",command=self.copy)

self.editmenu.add_command(label="Paste",accelerator="Ct
rl+V",command=self.paste)
```

```
    self.editmenu.add_separator()


self.editmenu.add_command(label="Undo",accelerator="Ct
rl+U",command=self.undo)

    self.menubar.add_cascade(label="Edit",
menu=self.editmenu)
```

8.Creating Help Menu

```
    self.helpmenu = Menu(self.menubar,font=("times new
roman",12,"bold"),activebackground="skyblue",tearoff=0)

self.helpmenu.add_command(label="About",command=self
.infoabout)

    self.menubar.add_cascade(label="Help",
menu=self.helpmenu)
```

9.Creating Scroll Bar

```
scrol_y = Scrollbar(self.root,orient=VERTICAL)

    self.txtarea =
Text(self.root,yscrollcommand=scrol_y.set,font=("times new
roman",15,"bold"),state="normal",relief=GROOVE)

    scrol_y.pack(side=RIGHT,fill=Y)

    scrol_y.config(command=self.txtarea.yview)
    self.txtarea.pack(fill=BOTH,expand=1)


    self.shortcuts()
```

10. Setting Function

```
def settitle(self):

    if self.filename:
          self.title.set(self.filename)
    else:
          self.title.set("Untitled")
```

11.New FileFunction

```
def newfile(self,*args):

    self.txtarea.delete("1.0",END)

    self.filename = None

      self.settitle()

    self.status.set("New File Created")
```

12.Open File Function

```
def openfile(self,*args):

    try:
      self.filename = filedialog.askopenfilename(title = "Select 
file",filetypes = (("All Files","*.*"),("Text Files","*.txt"),
("Python Files","*.py")))

     if self.filename:

       infile = open(self.filename,"r")

       self.txtarea.delete("1.0",END)

       for line in infile:
         self.txtarea.insert(END,line)

       infile.close()

       self.settitle()

       self.status.set("Opened Successfully")
    except Exception as e:
      messagebox.showerror("Exception",e)
```

13. Save File Function

```
def savefile(self,*args):

    try:

      if self.filename:

        data = self.txtarea.get("1.0",END)

        outfile = open(self.filename,"w")
        outfile.write(data)

        outfile.close()

        self.settitle()

        self.status.set("Saved Successfully")
      else:
        self.saveasfile()
    except Exception as e:
      messagebox.showerror("Exception",e)
```

14. Save As File Function

```
def saveasfile(self,*args):

    try:

      untitledfile = filedialog.asksaveasfilename(title = "Save
file As",defaultextension=".txt",initialfile =
"Untitled.txt",filetypes = (("All Files","*.*"),("Text
Files","*.txt"),("Python Files","*.py")))

      data = self.txtarea.get("1.0",END)

      outfile = open(untitledfile,"w")

      outfile.write(data)

      outfile.close()

      self.filename = untitledfile
      self.settitle()

           self.status.set("Saved Successfully")
    except Exception as e:
      messagebox.showerror("Exception",e)
```

15. Exit Function

```
 def exit(self,*args):
    op = messagebox.askyesno("WARNING","Your Unsaved 
Data May be Lost!!")
    if op>0:
      self.root.destroy()
    else:
      return
```

16. Cut / Copy /Paste Functions

```
def cut(self,*args):
    self.txtarea.event_generate("<<Cut>>")


  def copy(self,*args):
          self.txtarea.event_generate("<<Copy>>")


  def paste(self,*args):
    self.txtarea.event_generate("<<Paste>>")


  def undo(self,*args):
    # Exception handling
    try:
```

```
    if self.filename:

      self.txtarea.delete("1.0",END)

      infile = open(self.filename,"r")

      for line in infile:
        self.txtarea.insert(END,line)

      infile.close()

      self.settitle()

      self.status.set("Undone Successfully")
    else:

      self.txtarea.delete("1.0",END)

      self.filename = None

      self.settitle()

      self.status.set("Undone Successfully")
  except Exception as e:
    messagebox.showerror("Exception",e)
```

17. About Function

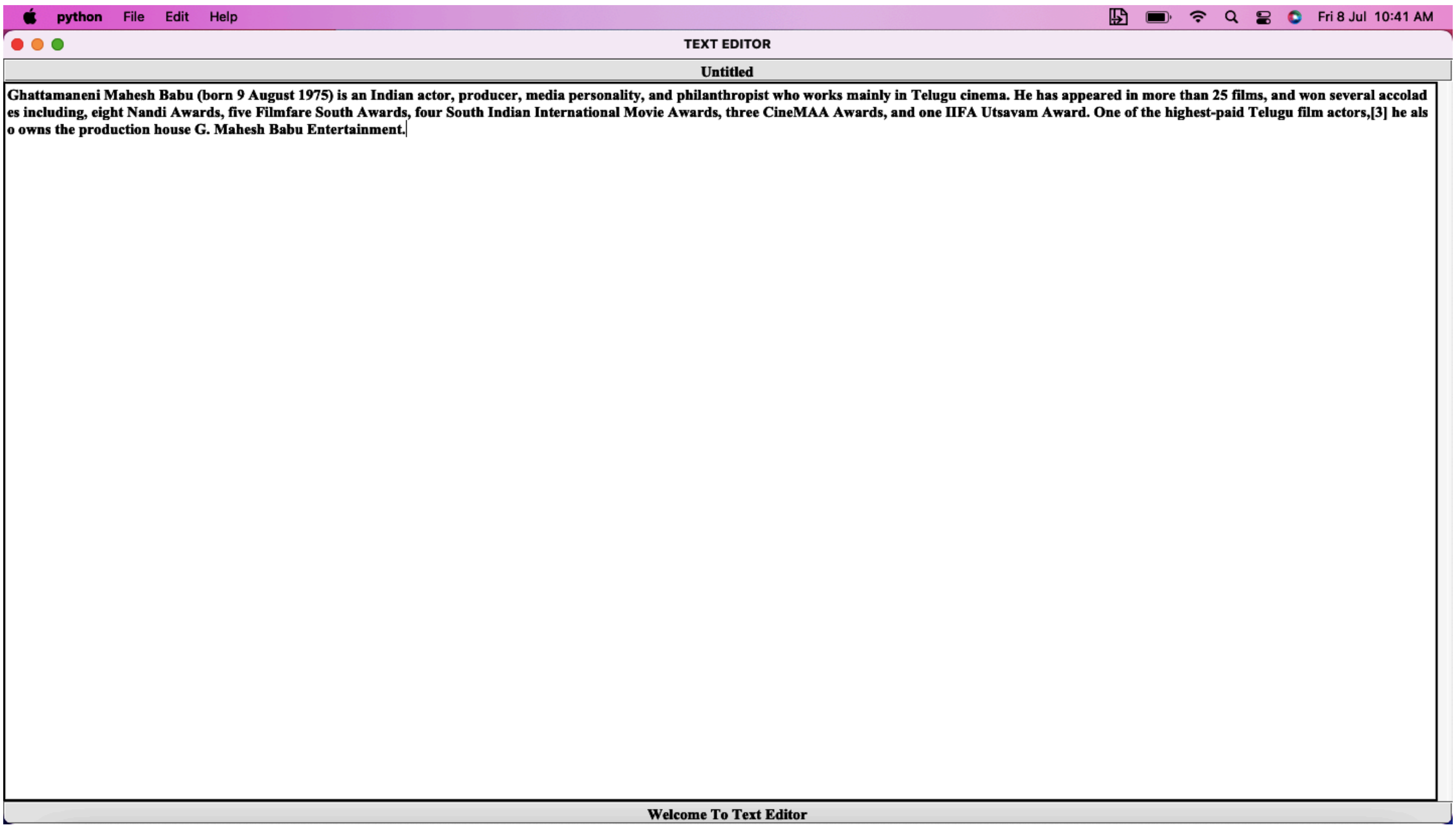
```
def infoabout(self):
  messagebox.showinfo("About Text Editor","A Simple
Text Editor\nCreated using Python.")
```

18. Shortcut Function

```
def shortcuts(self):

    self.txtarea.bind("<Control-n>",self.newfile)

    self.txtarea.bind("<Control-o>",self.openfile)

    self.txtarea.bind("<Control-s>",self.savefile)

   self.txtarea.bind("<Control-a>",self.saveasfile)

    self.txtarea.bind("<Control-e>",self.exit)

    self.txtarea.bind("<Control-x>",self.cut)

    self.txtarea.bind("<Control-c>",self.copy)

    self.txtarea.bind("<Control-v>",self.paste)

    self.txtarea.bind("<Control-u>",self.undo)


root = Tk()

TextEditor(root)

root.mainloop()
```

Python Text Editor Project Output

The below picture shows the python text editor window:



Ghattamaneni Mahesh Babu (born 9 August 1975) is an Indian actor, producer, media personality, and philanthropist who works mainly in Telugu cinema. He has appeared in more than 25 films, and won several accolad
es including, eight Nandi Awards, five Filmfare South Awards, four South Indian International Movie Awards, three CineMAA Awards, and one IIFA Utsavam Award. One of the highest-paid Telugu film actors,[3] he als
o owns the production house G. Mahesh Babu Entertainment.

Summary

We successfully created the Text Editor Project using Python. The modules and libraries we learned during the process of creation are

- Tkinter
- Filedialog
- MessageBox
-

Conclusion

From the report we studied that Understanding the syntax Of Python is great and all , and Python by itself is indeed a great language , but the fundamental of Python aren't why Python is a successful language. The are plenty   of the fun-to-write language that are just like Python , such as Ruby , Julie , even R. The study of wind turbine and its characteristics showed that how it can be properly designed and used to get the maximum output. As Python supports more than one programming paradigm that has functional , object-oriented and procedural programming , it is generally consider as batteries included language due to its standardised library. It was in the late 1980's when Python was conceived as a hit to the complete ABC language. In 2000, after the release of Python 2.0 the feature callas like garbage collection system and list comprehension were able to collect reference styles. This also showed Python is fun and stuff right now , but remember … Python is a fully-fledged programming language . Any code you download from GitHub ,the python Package Index , or anywhere else, can be malicious and would almost certainly go unnoticed by most people's firewalls.

Reference

https://data-flair.training/blogs/python-tutorials-home/

https://www.tutorialspoint.com/python/index.htm

https://www.geeksforgeeks.org/python-programming-language/

https://www.pythontutorial.net/advanced-python/

https://www.geeksforgeeks.org/top-10-advance-python-concepts-that-you-must- know/

https://python-course.eu/advanced-python/

https://www.youtube.com/watch?v=Yrtm7d3TJbs

The End