Python File Handling

Normally we take the input from the console and writing it back to the console to interact with the user. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is **volatile**, it is impossible to recover the programmatically generated data again and again.

The **file handling** plays an **important role when the data needs to be stored permanently into the file.** A ***file is a named location on disk to store related information.*** We can access the stored information (**non-volatile**) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in **two modes** as **text** or **binary**. The file may be in the text or binary format, and each line of a file is **ended with the special character.**

Hence, a file operation can be done in the following order.

1. **Open a file**
2. **Read or write - Performing operation**
3. **Close the file**

Opening a file

Python provides an **open()** function that accepts **two arguments**, **file name** and **access mode** in which the file is accessed. The

function **returns a file object which can be used to perform various operations like reading, writing, etc.**

Syntax:

file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the access mode to open a file.

SN	Access mode	Description
1	**r**	It **opens the file to read-only mode**. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	**rb**	It opens the file to **read-only in binary format**. The **file pointer exists at the beginning of the file**.
3	**r+**	It **opens the file to read and write both**. The file **pointer exists at the beginning of the file**.
4	**rb+**	It **opens the file to read and write both in binary format.** The **file pointer exists at the beginning of the file.**
5	**w**	It opens the file to **write only.** It **overwrites the file if previously exists or creates a new one if no file exists with the same name.** The **file pointer exists at the beginning of the file.**

6	**wb**	It **opens the file to write only in binary format**. It **overwrites the file if it exists previously or creates a new one if no file exists.** The **file pointer exists at the beginning of the file.**
7	**w+**	It **opens the file to write and read both**. ***It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists.*** The **file pointer exists at the beginning of the file.**
8	**wb+**	It **opens the file to write and read both in binary format.** The **file pointer exists at the beginning of the file.**
9	**a**	It **opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.**
10	**ab**	It **opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.**
11	**a+**	It **opens a file to append and read both. The file**

		pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	**ab+**	It **opens a file to append and read both in binary format. The file pointer remains at the end of the file.**

Example

```
# mydemo.txt file will open in write mode
# we will assign file name and opening mode
# through open function in file pointer object(fp1)
# syntax:
# file pointer = open("filename.txt","file opening mode")
fp1=open("mydemo.txt","w")
#above is a open() function
if fp1:
    print("mydemo file is opened in write mode.")
fp1.close()

#mydeo.txt file will open in read mode
fp2=open("mydemo.txt","r")
if fp2:
    print("mydemo file is opened in read mode")
fp2.close()
```

```
mydemo file is opened in write mode.
mydemo file is opened in read mode
```

In the above code, we have passed **filename as a first argument** and **opened file in read mode** as we mentioned **r as the second argument**. The **fp (file pointer) holds the file object and if the file is opened successfully, it will execute the print statement.**

close() method

Once all the operations are done on the file, we must close it through our Python script using the close() method. Any unwritten information gets destroyed once the close() method is called on a file object.

We can **perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.**

Syntax:

```
fileobject.close()
```

Example:

```
1  # mydemo.txt file will open in write mode
2  # we will assign file name and opening mode
3  # through open function in file pointer object(fp1)
4  # syntax:
5  # file pointer = open("filename.txt","file opening mode")
6  fp1=open("mydemo.txt","w")
7  #above is a open() function
8  if fp1:
9      print("mydemo file is opened in write mode.")
10 fp1.close() #close() function to close opened file
11
12 #mydeo.txt file will open in read mode
13 fp2=open("mydemo.txt","r")
14 if fp2:
15     print("mydemo file is opened in read mode")
16 fp2.close()#close() function to close opened file
```

```
mydemo file is opened in write mode.
mydemo file is opened in read mode
```

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

with statement

The **with** statement was introduced in **python 2.5**. The with statement is **useful in the case of manipulating the files**. **It is used in the scenario where a pair of statements is to be executed with a block of code in between.**

Syntax

```
with open(<file name>, <access mode>) as <file-pointer>:
#statement suite
```

The **advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.**

It is always suggestible to use with statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the close () function. It doesn't let the file to corrupt.

Example

```
#Example of with statement
fp1 = open("mydemo2.txt",'w')
if fp1:
    print("we are writing content through 'with statement'")
    print("This is an example of 'with statement'")
with open("mydemo2.txt",'r') as myfile:
    content = myfile.read();
    print(content)
```

```
we are writing content through 'with statement'
This is an example of 'with statement'
```

Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes:

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

```
# Open the mydemo4.txt in write mode.
# Create a new file if no such file exists.
fileptr = open("mydemo4.txt", "w")

# appending the content to the file
fileptr.write('''1. Python is the modern day language.''')

# closing the opened the file
fileptr.close()

with open("mydemo4.txt",'r') as myfile:
    content = myfile.read();
    print(content)
```

```
1. Python is the modern day language.
```

We have opened the file in **w mode**. The **mydemo4.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.

```
#open the mydemo4.txt in append mode.
fileptr = open("mydemo4.txt","a")

#overwriting the content of the file
fileptr.write("\n2. Python has an easy syntax and user-friendly interaction

#closing the opened file
fileptr.close()
with open("mydemo4.txt",'r') as myfile:
    content = myfile.read();
    print(content)

```

```
1. Python is the modern day language.
2. Python has an easy syntax and user-friendly interaction.
```

We can see that the **content of the file is modified**. We have opened the file in **'a' mode** and **it appended the content in the existing mydemo4.txt**.

To read a file using the Python script, the Python provides the read() method. The read() method reads a string from the file. It can read the data in the text as well as a binary format.

Syntax:

fileobj.read(<count>)

Here, the **count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.**

```python
# open the mydemo4.txt in read mode.
# causes error if no such file exists.
fp = open("mydemo4.txt","r")

# stores all the data of the file into the variable content
content = fp.read(20)

# prints the type of the data stored in the file
print(type(content))

# prints the content of the file
print(content)

# closes the opened file
fp.close()
```

```
<class 'str'>
1. Python is the mod
```

In the above code, we have read the content of **mydemo4.txt** by using the **read()** function. We have passed count value as **20** which means it will **read the first 20 characters from the file**.

If we want to print all content of the file, then we will use read() function in following way-

```
# open the mydemo4.txt in read mode.
# causes error if no such file exists.
fp = open("mydemo4.txt","r")

# stores all the data of the file into the variable content
content = fp.read()

# prints the type of the data stored in the file
print(type(content))

# prints the content of the file
print(content)

# closes the opened file
fp.close()
```

```
<class 'str'>
1. Python is the modern day language.
2. Python has an easy syntax and user-friendly interaction.
```

Read file through for loop

We can read the file using for loop. Consider the following example.

```
# open the mydemo4.txt in read mode.
# causes an error if no such file exists.
fp = open("mydemo4.txt","r");
# running a for loop
for i in fp:
    print(i) # i contains each line of the file

```

```
1. Python is the modern day language.

2. Python has an easy syntax and user-friendly interaction.
```

Read Lines of the file

Python facilitates to **read the file line by line** by using a function **readline()** method. The **readline()** method **reads the lines of the file from the beginning,** i.e., if we use the **readline()** method **two times, then we can get the first two lines of the file**.

Consider the following example which contains a function **readline()** that **reads the first line of our file** "**mydemo4.txt**" which is containing two lines.

Example

```
# open the mydemo4.txt in read mode.
# causes error if no such file exists.
fp = open("mydemo4.txt","r");

# stores all the data of the file into the variable content
content = fp.readline()
content1 = fp.readline()

# prints the content of the file
print(content)
print(content1)

# closes the opened file
fp.close()
```

```
1. Python is the modern day language.

2. Python has an easy syntax and user-friendly interaction.
```

We called the **readline()** function **two times that's why it read two lines from the file.**

Python provides also the **readlines()** method which is **used for the reading lines.** It **returns the list of the lines till the end of file(EOF) is reached.**

Example : Reading Lines Using readlines() function

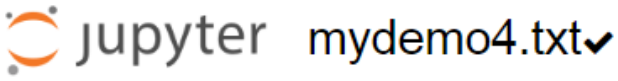
```
# open the mydemo4.txt in read mode.
# causes error if no such file exists.
fp = open("mydemo4.txt","r");

# stores all the data of the file
# into the variable content
content = fp.readlines()

# prints the content of the file
print(content)

# closes the opened file
fp.close()
```

```
['1. Python is the modern day language.\n', '2. Python has an easy syntax and u
ser-friendly interaction.']
```



File Edit View Language

```
1. Python is the modern day language.
2. Python has an easy syntax and user-friendly interaction.
```

Creating a new file

The new file can be **created by using one of the following access modes with the function open().**

x: It creates a new file with the specified name. It causes an error a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Example

```
1  # open the mydemo5.txt in read mode.
2  # causes error if no such file exists.
3  fp = open("mydemo5.txt","x")
4  print(fp)
5  if fp:
6      print("File created successfully")
7
```

```
<_io.TextIOWrapper name='mydemo5.txt' mode='x' encoding='cp1252'>
File created successfully
```

```
1  # open the mydemo4.txt in read mode.
2  # causes an error a file exists with the same name.
3  fp = open("mydemo4.txt","x")
4  print(fp)
5  if fp:
6      print("File created successfully")
7
```

```
---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
<ipython-input-35-5a19c8cae96a> in <module>
      1 # open the mydemo4.txt in read mode.
      2 # causes an error a file exists with the same name.
----> 3 fp = open("mydemo4.txt","x")
      4 print(fp)
      5 if fp:

FileExistsError: [Errno 17] File exists: 'mydemo4.txt'
```

File Pointer positions

Python provides the **tell() method** which is **used to print the byte number at which the file pointer currently exists**. Consider the following example.

```
1  # open the file mydemo4.txt in read mode
2  fp = open("mydemo4.txt","r")
3
4  #initially the filepointer is at 0
5  print("The filepointer is at byte :",fp.tell())
6
7  #reading the content of the file
8  content = fp.read();
9
10 #after the read operation file pointer modifies.
11 #tell() returns the location of the fp.
12
13 print("After reading, the filepointer is at:",fp.tell())
14
```

```
The filepointer is at byte : 0
After reading, the filepointer is at: 98
```

Modifying file pointer position

In real-world applications, sometimes we **need to change the file pointer location externally since we may need to read or write the content at various locations.**

For this purpose, the Python provides us the **seek() method** which **enables us to modify the file pointer position externally.**

Syntax:

<file-ptr>.seek(offset[, from])

The seek() method accepts two parameters:

offset: It **refers to the new position of the file pointer within the file.**

from: It **indicates the reference position from where the bytes are to be moved.** If it is set to **0**, the **beginning of the file is used as the reference position**. If it is set to **1**, **the current position of the file pointer is used as the reference position.** If it is set to **2**, the **end of the file pointer is used as the reference position.**

```
1  # open the file mydemo4.txt in read mode
2  fp = open("mydemo4.txt","r")
3
4  #initially the filepointer is at 0
5  print("The filepointer is at byte :",fp.tell())
6
7  #changing the file pointer location to 10.
8  fp.seek(10,0);
9
10 #tell() returns the location of the fp.
11 print("After reading, the filepointer is at:",fp.tell())
```

```
The filepointer is at byte : 0
After reading, the filepointer is at: 10
```

File related methods

The file object provides the following methods to manipulate the files on various operating systems.

SN	Method	Description
1	file.close()	It closes the opened file. The file once closed, it can't be read or write anymore.
2	File.fush()	It flushes the internal buffer.
3	File.fileno()	It returns the file descriptor used by the underlying implementation to

		request I/O from the OS.
4	**File.isatty()**	It returns true if the file is connected to a TTY(Teletype) device, otherwise returns false.
5	**File.next()**	It returns the next line from the file.
6	**File.read([size])**	It reads the file for the specified size.
7	**File.readline([size])**	It reads one line from the file and places the file pointer to the beginning of the new line.
8	**File.readlines([sizehint])**	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function.
9	**File.seek(offset[,from)**	It modifies the position of the file pointer to a specified offset with the specified reference.
10	**File.tell()**	It returns the current position of the file pointer within the file.
11	**File.truncate([size])**	It truncates the file to the optional specified size.
12	**File.write(str)**	It writes the specified string to a file
13	**File.writelines(seq)**	It writes a sequence of the strings to a file.

Command line Arguments

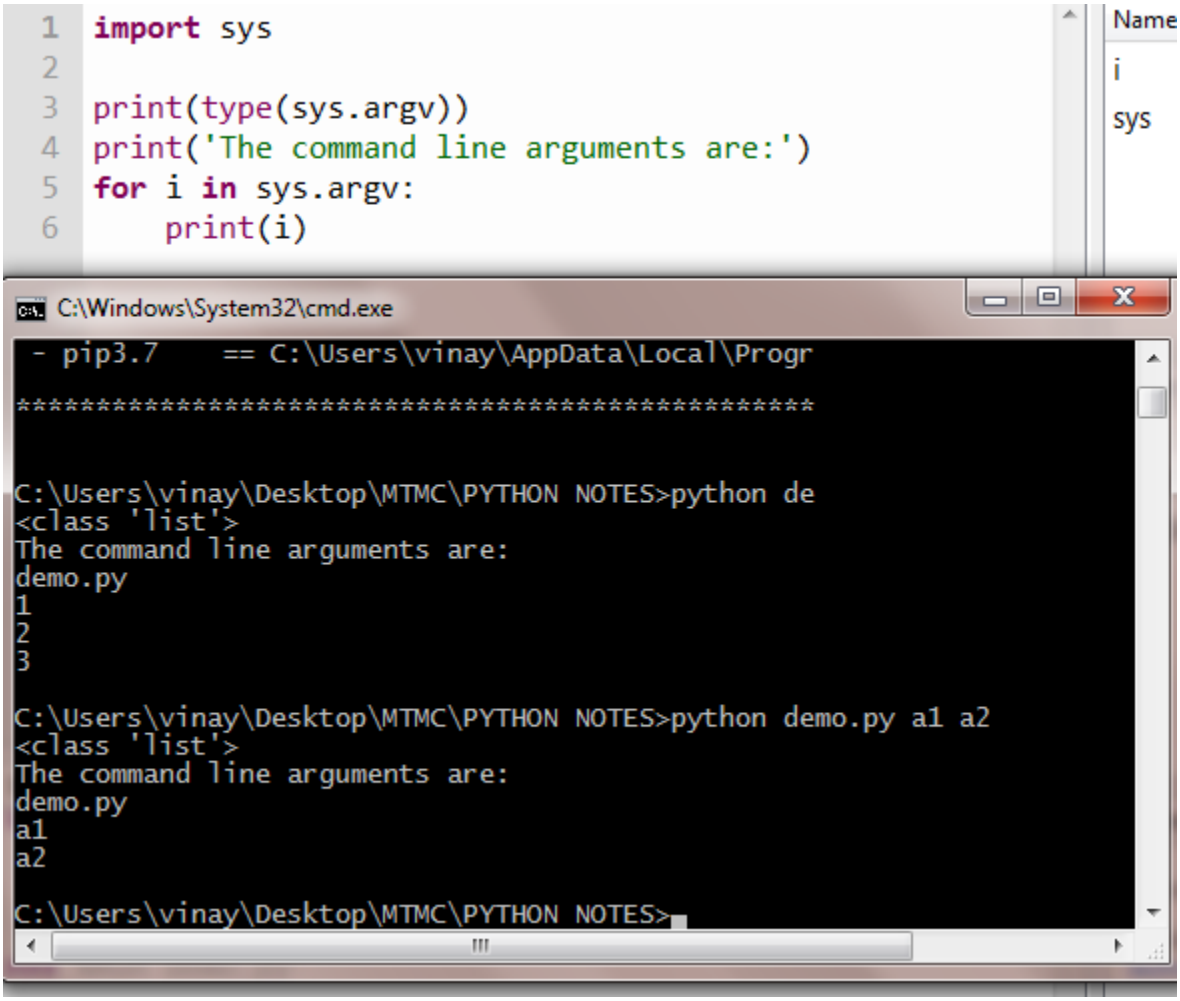
- **Python Command line arguments are input parameters passed to the script when executing them.**
- **A command line interface (CLI) is a program that is designed to allow users to interact with software on their computer.**
- **Many kinds of software have specific tasks that run and perform based on keywords and variables that together create a command line argument.**
- **A command-line argument is an argument that gets passed into a program through the CLI by the user.**

Python sys module

Python sys module stores the command line arguments into a list, we can access it using sys.argv. This is very useful and simple way to read command line arguments as String.

The Python sys module provides access to any command-line arguments via the sys.argv. This serves two purposes –

- **sys.argv is the list of command-line arguments.**
- **len(sys.argv) is the number of command-line arguments**



Pickle module (Object Persistence)

Python pickle module is **used for serializing and de-serializing python object structures. The process to converts any kind of python objects (list, dict, etc.) into byte streams (0s and 1s)** is called **pickling** or **serialization** or **flattening** or **marshalling**. We can **convert the byte stream (generated through pickling) back into python objects by a process called as unpickling.**

Pickle: Need

In real world sceanario, the use pickling and unpickling are widespread as they **allow us to easily transfer data from one server/system to another and then store it in a file or database.**

It is advisable not to unpickle data received from an untrusted source as they may pose security threat. However, the pickle module has no way of knowing or raise alarm while pickling malicious data.

Only after importing pickle module we can do pickling and unpickling.

Importing pickle can be done using the following command –

import pickle

Example

```
# Example for pickle
import pickle
mylist = ['a', 'b', 'c', 'd']
with open('demo3.txt', 'wb') as p:
    pickle.dump(mylist, p)
```

In the above code, list – "mylist" contains four elements ('a', 'b', 'c', 'd'). We open the file in "wb" mode instead of "w" as all the operations are done using bytes in the current working directory. A new file named "demo3.txt" is created, which converts the mylist data in the byte stream.

Example: for unpickling

```
# example for unpickling
import pickle
pickle_off = open ("demo3.txt", "rb")
emp = pickle.load(pickle_off)
print(emp)
```

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

On running above scripts, we can see our mylist data again as output.

['a', 'b', 'c', 'd']

Example:

```
import pickle
EmpID = {1:"shailendra",2:"40000",3:"CS",4:"46"}
pickling_on = open("EmpID.pickle","wb")
pickle.dump(EmpID, pickling_on)
pickling_on.close()
```

```
#Unpickle a dictionary
import pickle
pickle_off = open("EmpID.pickle", 'rb')
EmpID = pickle.load(pickle_off)
print(EmpID)
```

```
{1: 'shailendra', 2: '40000', 3: 'CS', 4: '46'}
```

On running above script (unpickle) we get our dictionary back as we initialized earlier. Also, please note because we are reading bytes here, we have used “rb” instead of “r”.

Pickle Exceptions

Below are some of the common exceptions raised while dealing with pickle module −

- **Pickle.PicklingError**: If the pickle object doesn't support pickling, this exception is raised.
- **Pickle.UnpicklingError**: In case the file contains bad or corrupted data.
- **EOFError**: In case the end of file is detected, this exception is raised.

Advantages:

- Comes handy to save complicated data.
- Easy to use, lighter and doesn't require several lines of code.
- The pickled file generated is not easily readable and thus provide some security.

Disadvantages:

- Languages other than python may not able to reconstruct pickled python objects.
- Risk of unpickling data from malicious sources.

Python object persistence (shelve)

The **shelve module** in Python's standard library is a simple yet effective tool for **persistent data storage** when using a relational database solution is not required. The shelf object defined in this module is **dictionary-like object** which is **persistently stored in a**

disk file. This creates a file **similar to dbm database on UNIX like systems.** Only string data type can be used as key in this special dictionary object, whereas any picklable object can serve as value.

The shelve module defines three classes as follows –

Sr.No.	Module & Description
1	**Shelf** This is the **base class for shelf implementations**. It is initialized with dict-like object.
2	**BsdDbShelf** This is a subclass of Shelf class. The dict object passed to its constructor must support first(), next(), previous(), last() and set_location() methods.
3	**DbfilenameShelf** This is also a subclass of Shelf but accepts a filename as parameter to its constructor rather than dict object.

Easiest way to form a **Shelf object** is to use **open() function** defined in shelve module which return a **DbfilenameShelf object**.

open(filename, flag = 'c', protocol=None, writeback = False)

The filename parameter is assigned to the database created.

Default value for flag parameter is 'c' for read/write access. Other flags are 'w' (write only) 'r' (read only) and 'n' (new with read/write)

Protocol parameter denotes pickle protocol writeback parameter by default is false. If set to true, the accessed entries

are cached. Every access calls sync() and close() operations hence process may be slow.

Following code creates a database and stores dictionary entries in it.

```
# example for shelves
import shelve
s = shelve.open("test")
s['name'] = "Amit"
s['age'] = 23
s['marks'] = 80
s.close()
```

```
s=shelve.open('test')
s['age']
```

```
23
```

```
s['age']=25
s.get('age')
```

```
25
```

This will create test.dir file in current directory and store key-value data in hashed form. The Shelf object has following methods available –

Sr.No.	Method & Description
1	**close()** synchronise and close persistent dict object.
2	**sync()** Write back all entries in the cache if shelf was opened with writeback set to True.
3	**get()** returns value associated with key
4	**items()** list of tuples – each tuple is key value pair
5	**keys()** list of shelf keys
6	**pop()** remove specified key and return the corresponding value.
7	**update()** Update shelf from another dict/iterable
8	**values()** list of shelf values

The items(), keys() and values() methods return view objects.

```
list(s.items())
```

```
[('name', 'Amit'),
 ('age', 25),
 ('salary', 10000),
 ('designation', 'manager'),
 ('marks', 80)]
```

```
list(s.keys())
```

```
['name', 'age', 'salary', 'designation', 'marks']
```

```
list(s.values())
```

```
['Amit', 25, 10000, 'manager', 80]
```

To remove a key-value pair from shelf

```
s.pop('marks')
```

```
80
```

```
list(s.items())
```

```
[('name', 'Amit'), ('age', 25), ('salary', 10000), ('designation', 'manager')]
```

Notice that key-value pair of marks-80 has been removed.

To merge items of another dictionary with shelf use update() method

```
d={'salary':10000, 'designation':'manager'}
s.update(d)
list(s.items())
```

```
[('name', 'Amit'), ('age', 25), ('salary', 10000), ('designation', 'manager')]
```

Lambda Functions

These are **anonymous functions**, implying they **don't have a name**. The **def** keyword is needed **to create a typical function** in Python whereas **lambda** keyword used in Python **to define an unnamed function.**

Syntax

lambda arguments: expression

This function **accepts any count of inputs but only evaluates and returns one expression.**

Lambda functions can be **used whenever function arguments are necessary.** Lambda functions are limited to a single statement.

```
1  # typical function
2
3  def add(a,b):
4      return a+b
5  print("addition is:",add(6,9))
```

```
addition is: 15
```

```
1  # Lambda Function
2  # no name function
3  add2=lambda a,b:a+b
4  print("addition is:",add2(4,8))
```

```
addition is: 12
```

Using Lambda Function with filter()

The filter() method **accepts two arguments in Python: a function and an iterable such as a list.**

The **function is called for every item of the list, and a new iterable or list is returned that holds just those elements that returned True when supplied to the function.**

```
# Code to filter odd numbers from a given list
mylist = [11, 12, 13, 14, 15,16,17]
odd_list = list(filter( lambda num: (num % 2 != 0) , mylist ))
print("filtered odd numbers are:",odd_list)
```

```
filtered odd numbers are: [11, 13, 15, 17]
```

Using Lambda Function with map()

A **method and a list are passed to Python's map() function.**

The **function is executed for all of the elements within the list, and a new list is produced with elements generated by the given function for every item.**

```
#Code to calculate the square of each number
#of a list using the map() function
num_list = [1,2,3,4,5]
squared_list = list(map( lambda num: num ** 2 , num_list ))
print( "Square of elements are",squared_list )
```

```
Square of elements are [1, 4, 9, 16, 25]
```

zip() Function

Python zip() function **returns a zip object, which maps a similar index of multiple containers.**

It **takes iterables (can be zero or more), makes it an iterator that aggregates the elements based on iterables passed, and returns an iterator of tuples.**

Syntax:

zip(iterator1, iterator2, iterator3 ...)

iterator1, iterator2, iterator3: These are **iterator objects that joined together.**

```
# zip function
u_id=['user1','user2','user3']
names=['priya','priyanka','aishwarya']
print(zip(u_id,names)) # returns identity
print(list(zip(u_id,names)))
print(type(list(zip(u_id,names))))
print(dict(zip(u_id,names)))
print(type(dict(zip(u_id,names))))
```

```
<zip object at 0x037C0C68>
[('user1', 'priya'), ('user2', 'priyanka'), ('user3', 'aishwarya')]
<class 'list'>
{'user1': 'priya', 'user2': 'priyanka', 'user3': 'aishwarya'}
<class 'dict'>
```

Regular Expression (RegEx)

- **It is a sequence of characters that forms a search pattern.**
- **It can be used to check if a string contains the specified search pattern.**
- **To use RegEx, we hve to use python built in package i.e. re so we have to import re module.**

import re

Meta characters:

- **Characters with special meaning**

1. [ ] A set of characters "[a-m]"

```
# 1. [ ] set of characters  "[a-s]"
import re
txt = "we are students of MTMC"
#Find all lower case characters alphabetically between "a" and "s":
x = re.findall("[a-s]", txt)
print(x)
```

```
['e', 'a', 'r', 'e', 's', 'd', 'e', 'n', 's', 'o', 'f']
```

2. \ Signals a special sequence (can also be used to escape special characters) "\d"

```
# 2.\ Signals a special sequence
#(can also be used to escape special characters)
# "\d"
import re
txt = "we are 14 students in MTMC"
#Find all digit characters:
x = re.findall("\d", txt)
print(x)
```

```
['1', '4']
```

3. . Any character (except newline character) "he..o"

```
# 3. .(DOT) Any character (except newline character)——»"he..o"
import re
txt = "MTMC students"
# Search for a sequence that starts with "st",
# followed by two (any) characters, and an "e":
x = re.findall("st..e", txt)
print(x)
```

```
['stude']
```

4. ^ Starts with "^hello"

```
# 4. ^ Starts with "^welcome"
import re
txt = "welcome students"
#Check if the string starts with 'welcome':
x = re.findall("^welcome", txt)
if x:
  print("Yes, the string starts with 'welcome'")
else:
  print("No match")

```

```
Yes, the string starts with 'welcome'
```

5. $ ends with "students$"

```
#5. $  ends with  "students$"
import re
txt = "we are MTMC students"
#Check if the string ends with 'students':
x = re.findall("students$", txt)
if x:
  print("Yes, the string ends with 'students'")
else:
  print("No match")

```

```
Yes, the string ends with 'students'
```

6. * zero or more occurrences "st.*s"

```
#6.*  zero or more occurrences    "st.*s"
import re
txt = "we are MTMC students"
#Search for a sequence that starts with "st",
#followed by 0 or more  (any) characters, and an "s":
x = re.findall("st.*s", txt)
print(x)
```

```
['students']
```

7. + one or more occurrences "st.+s"

```
#7. +  one or more occurrences   "st.+s"
import re
txt = "we are MTMC students"
#Search for a sequence that starts with "st",
#followed by 1 or more  (any) characters, and an "s":
x = re.findall("st.+s", txt)
y = re.findall("st.+d",txt)
print("x:",x)
print("y:",y)
```

```
x: ['students']
y: ['stud']
```

8. ? zero or one occurrence “st.?s”

```
#8. ?  zero or one occurrence “st.?s”
import re
txt = "we are MTMC students"
#Search for a sequence that starts with "st",
#followed by 0 or 1  (any) character, and an "s":
x = re.findall("st.?s", txt)
#This time we got no match, because there were not zero,
#not one, but two characters between "st" and the "s"
print(x)
y = re.findall("st.?d", txt)
print(y)
```

```
[]
['stud']
```

9. { } exactly specified number of occurrences “st.{2}s”

```
#9. { } exactly specified number of occurrences    “st.{2}s”
import re
txt = "we are MTMC students"
#Search for a sequence that starts with "st",
#followed excactly 2 (any) characters, and an "s":
x = re.findall("st.{2}s", txt)
y = re.findall("st.{2}e", txt)
print(x)
print(y)
```

```
[]
['stude']
```

10. | either or "falls | stays"

```
#10. | either or  "we | MTMC"
import re
txt = "we are MTMC students"
#Check if the string contains either "we" or "MTMC":
x = re.findall("we|MTMC", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['we', 'MTMC']
Yes, there is at least one match!
```

Special Sequences

A special sequence is a \ followed by one of the characters in the list below and has a special meaning.

1. **\A : returns a match if the specified characters are at the beginning of the string**

```
import re
txt = "we are MTMC students"
#Check if the string starts with "we":
x = re.findall("\Awe", txt)
print(x)
if x:
  print("Yes, there is a match!")
else:
  print("No match")
```

```
['we']
Yes, there is a match!
```

```
import re
txt = "we are MTMC students"
#Check if the string starts with "are":
x = re.findall("\Aare", txt)
print(x)
if x:
  print("Yes, there is a match!")
else:
  print("No match")
```

```
[]
No match
```

2. **\b : Returns a match where the specified characters are at the beginning or at the end of a word**

   (The "r" in the beginning is making sure that the string is being treated as a "raw string")

```
import re
txt = "we are MTMC students"
#Check if "MTM" is present at the beginning of a WORD:
x = re.findall(r"\bMTM", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['MTM']
Yes, there is at least one match!
```

```
import re
txt = "we are MTMC students"
#Check if "TMC" is present at the end of a WORD:
x = re.findall(r"TMC\b", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['TMC']
Yes, there is at least one match!
```

3. **\B: Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word**

(The "r" in the beginning is making sure that the string is being treated as a "raw string")

```
import re
txt = "we are MTMC students"
#Check if "TMC" is present, but NOT at the beginning of a word:
x = re.findall(r"\BTMC", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['TMC']
Yes, there is at least one match!
```

4. \d Returns a match where the string contains digits (numbers from 0-9)

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string contains any digits (numbers from 0-9):
x = re.findall("\d", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['1', '4']
Yes, there is at least one match!
```

5. \D Returns a match where the string DOES NOT contain digits

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Return a match at every no-digit character:
x = re.findall("\D", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['w', 'e', ' ', 'a', 'r', 'e', ' ', 'M', 'T', 'M', 'C', ' ', 's', 't', 'u',
'd', 'e', 'n', 't', 's', ' ', 'a', 'n', 'd', ' ', 'a', 'r', 'e', ' ', ' ', 'i',
'n', ' ', 'n', 'u', 'm', 'b', 'e', 'r', 's']
Yes, there is at least one match!
```

6. \s Returns a match where the string contains a white space character

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Return a match at every white-space character:
x = re.findall("\s", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
Yes, there is at least one match!
```

7. \S Returns a match where the string DOES NOT contain a white space character

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Return a match at every NON white-space character:
x = re.findall("\S", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['w', 'e', 'a', 'r', 'e', 'M', 'T', 'M', 'C', 's', 't', 'u', 'd', 'e', 'n',
't', 's', 'a', 'n', 'd', 'a', 'r', 'e', '1', '4', 'i', 'n', 'n', 'u', 'm', 'b',
'e', 'r', 's']
Yes, there is at least one match!
```

8. \w Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)

```
1  import re
2  txt = "we are MTMC students and are 14 in numbers"
3  #Return a match at every word character
4  #(characters from a to Z, digits from 0-9, and the underscore _ character):
5  x = re.findall("\w", txt)
6  print(x)
7  if x:
8    print("Yes, there is at least one match!")
9  else:
10   print("No match")
```

```
['w', 'e', 'a', 'r', 'e', 'M', 'T', 'M', 'C', 's', 't', 'u', 'd', 'e', 'n',
't', 's', 'a', 'n', 'd', 'a', 'r', 'e', '1', '4', 'i', 'n', 'n', 'u', 'm', 'b',
'e', 'r', 's']
Yes, there is at least one match!
```

9. \W Returns a match where the string DOES NOT contain any word characters

```
1  import re
2  txt = "we are MTMC students and are 14 in numbers"
3  #Return a match at every NON word character
4  #(characters NOT between a and Z. Like "!", "?" white-space etc.):
5  x = re.findall("\W", txt)
6  print(x)
7  if x:
8    print("Yes, there is at least one match!")
9  else:
10   print("No match")
```

```
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
Yes, there is at least one match!
```

10. **\Z Returns a match if the specified characters are at the end of the string**

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string ends with "numbers":
x = re.findall("numbers\Z", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['numbers']
Yes, there is at least one match!
```

Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:

1. **[arn] Returns a match where one of the specified characters (a, r, or n) is present**

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string has any a, b, c or d characters:
x = re.findall("[abcd]", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['a', 'd', 'a', 'd', 'a', 'b']
Yes, there is at least one match!
```

2. **[a-n] Returns a match for any lower case character, alphabetically between a and n**

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string has any characters between a and d:
x = re.findall("[a-d]", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['a', 'd', 'a', 'd', 'a', 'b']
Yes, there is at least one match!
```

3. **[^arn] Returns a match for any character EXCEPT a, r, and n**

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string has other characters than a, b, c or d:
x = re.findall("[^abcd]", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['w', 'e', ' ', 'r', 'e', ' ', 'M', 'T', 'M', 'C', ' ', 's', 't', 'u', 'e',
'n', 't', 's', ' ', 'n', ' ', 'r', 'e', ' ', '1', '4', ' ', 'i', 'n', ' ', 'n',
'u', 'm', 'e', 'r', 's']
Yes, there is at least one match!
```

4. **[0123] Returns a match where any of the specified digits (0, 1, 2, or 3) are present**

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string has any 0, 1, 2, or 3 digits:
x = re.findall("[1234]", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['1', '4']
Yes, there is at least one match!
```

5. **[0-9] Returns a match for any digit between 0 and 9**

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string has any digits:
x = re.findall("[0-9]", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['1', '4']
Yes, there is at least one match!
```

6. [0-5][0-9] Returns a match for any two-digit numbers from 00 and 59

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string has any two-digit numbers, from 00 to 59:
x = re.findall("[1-3][4-9]", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['14']
Yes, there is at least one match!
```

7. [a-zA-Z] Returns a match for any character alphabetically between a and z, lower case OR upper case

```
import re
txt = "we are MTMC students and are 14 in numbers"
#Check if the string has any characters from a to z lower case,
#and A to Z upper case:
x = re.findall("[a-zA-Z]", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['w', 'e', 'a', 'r', 'e', 'M', 'T', 'M', 'C', 's', 't', 'u', 'd', 'e', 'n',
't', 's', 'a', 'n', 'd', 'a', 'r', 'e', 'i', 'n', 'n', 'u', 'm', 'b', 'e', 'r',
's']
Yes, there is at least one match!
```

8. **[+] In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string**

```
1  import re
2  txt = "we are MTMC students + are 14 in numbers"
3  #Check if the string has any + characters:
4  x = re.findall("[+]", txt)
5  print(x)
6  if x:
7    print("Yes, there is at least one match!")
8  else:
9    print("No match")
```

```
['+']
Yes, there is at least one match!
```

findall() Function

The findall() function returns a list containing all matches.

Example

```
1  import re
2  txt = "we are MTMC students + are 14 in numbers"
3  x = re.findall("re", txt)
4  print(x)
```

```
['re', 're']
```

The list contains the matches in the order they are found.
If no matches are found, an empty list is returned:

```
1  import re
2  txt = "we are MTMC students and are 14 in numbers"
3  x = re.findall("ea", txt)
4  print(x)
```

```
[]
```

search() Function

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

```
import re
txt1 = "we are MTMC students and are 14 in numbers"
txt2 = "weare MTMC students and are 14 in numbers"
x = re.search("\s", txt1)
print("The first white-space character is located in position:", x.start())
y = re.search("\s", txt2)
print("The first white-space character is located in position:", y.start())
```

```
The first white-space character is located in position: 2
The first white-space character is located in position: 5
```

split() Function

The split() function returns a list where the string has been split at each match:

```
import re
txt = "we are MTMC students and are 14 in numbers"
x = re.split("\s", txt)
print(x)
```

```
['we', 'are', 'MTMC', 'students', 'and', 'are', '14', 'in', 'numbers']
```

sub() Function

The sub() function replaces the matches with the text of your choice:

```
import re
txt = "we are MTMC students and are 14 in numbers"
x = re.sub("\s", "-", txt)
print(x)
```

```
we-are-MTMC-students-and-are-14-in-numbers
```

Match Object

A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value None will be returned, instead of the Match Object.

```
1  import re
2
3  txt = "we are MTMC students and are 14 in numbers"
4  x = re.search("ud", txt)
5  print(x) #this will print an object
```

```
<re.Match object; span=(14, 16), match='ud'>
```

The Match object has properties and methods used to retrieve information about the search, and the result:

.span() returns a tuple containing the start-, and end positions of the match.

.string returns the string passed into the function

.group() returns the part of the string where there was a match

```
1  import re
2
3  txt = "we are MTMC students and are 14 in numbers"
4  x = re.search("ud", txt)
5  print(x.span()) #span
```

```
(14, 16)
```

```
1  import re
2
3  txt = "we are MTMC students and are 14 in numbers"
4  x = re.search("ud", txt)
5  print(x.string) # string
```

```
we are MTMC students and are 14 in numbers
```

```
1  import re
2
3  txt = "we are MTMC students and are 14 in numbers"
4  x = re.search("ud", txt)
5  print(x.group()) #group
```

```
ud
```

Note: If there is no match, the value None will be returned, instead of the Match Object.

```
1  import re
2
3  txt = "we are MTMC students and are 14 in numbers"
4  x = re.search("pp", txt)
5  print(x) #group
```

```
None
```

Python Exception

An exception can be defined as **an unusual condition in a program resulting in the interruption in the flow of the program.**

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to

Python script. An exception is a Python object that represents an error

Python **provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.**

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

1. **Common Exceptions**

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list 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. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** 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.

The problem without handling exceptions

We know that the **exception is an abnormal condition that halts the execution of the program.**

Suppose we have two variables a and b, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a ZeroDivision exception.

Example

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print("a/b = %d"%c)

#other code:
print("Hi I am other part of the program")
```

```
Enter a:6
Enter b:5
a/b = 1
Hi I am other part of the program
```

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print("a/b = %d"%c)

#other code:
print("Hi I am other part of the program")
```

```
Enter a:3
Enter b:0

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-9-c3b3474c804b> in <module>
      1 a = int(input("Enter a:"))
      2 b = int(input("Enter b:"))
----> 3 c = a/b
      4 print("a/b = %d"%c)
      5

ZeroDivisionError: division by zero
```

The above program is **syntactically correct**, but raising the **error** because of **unusual input**.

Such kind of **programming may not be suitable or recommended for the projects** because these **projects are required uninterrupted execution.**

That's why an exception-handling plays an essential role in handling these unexpected exceptions.

Exception handling in python

The try-except statement

If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block. The **try** block must be followed with the **except** statement, which

contains a block of code that will be executed if there is some exception in the try block.

Syntax

```
try:
    #block of code
except Exception1:
    #block of code
except Exception2:
    #block of code
#other code
```

Example 1

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```

```
Enter a:4
Enter b:0
Can't divide with zero
```

We can also use the **else** statement with the **try-except** statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

Syntax

```
try:
    #block of code

except Exception1:
    #block of code

else:
    #this code executes if no except block is executed
```

Example 2

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
# Using Exception with except statement.
# If we print(Exception) it will return exception class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```

```
Enter a:7
Enter b:0
can't divide by zero
<class 'Exception'>
```

The except statement with no exception

Python **provides the flexibility not to specify the name of exception with the exception statement.**

Example

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except:
    print("can't divide by zero")
else:
    print("Hi I am else block")
```

```
Enter a:8
Enter b:0
can't divide by zero
```

The except statement using with exception variable

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. This object will **return the cause of the exception.** Consider the following example:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
    # Using exception object with the except statement
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

```
Enter a:5
Enter b:0
can't divide by zero
division by zero
```

Points to remember

1. Python **facilitates us to not specify the exception with the except statement.**
2. We can **declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.**
3. We can also **specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.**
4. The statements that **don't throw the exception should be placed inside the else block.**

Example

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("demo.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    fileptr.close()
```

```
The file opened successfully
```

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("demo1.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    fileptr.close()
```

```
File not found
```

Declaring Multiple Exceptions

The Python **allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions.** The syntax is given below.

Syntax

```
try:
    #block of code
except (<Exception 1>,<Exception 2>,<Exception 3>,...
<Exception n>)
    #block of code
else:
    #block of code
```

Example

```
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

```
Arithmetic Exception
```

The try...finally block

Python provides the optional **finally** statement, which is used with the **try** statement. **It is executed no matter what exception occurs and used to release the external resource.** The **finally block provides a guarantee of the execution.**

We can **use the finally block with the try block in which we can place the necessary code, which must be executed before the try statement throws an exception.**

Syntax

```
try:
```

```
    # block of code
    # this may throw an exception
finally:
    # block of code
    # this will always be executed
```

Example

```
try:
    fileptr = open("demo.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

```
file closed
Error
```

2. Raising exceptions

An exception can be raised forcefully by using the raise clause in Python. It is useful in that scenario where we need to raise an exception to stop the execution of the program.

For example, ***there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.***

Syntax

raise Exception_class,<value>

Points to remember

1. **To raise an exception, the raise statement is used. The exception class name follows it.**
2. An **exception can be provided with a value that can be given in the parenthesis.**
3. To **access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.**
4. We **can pass the value to an exception to specify the exception type.**

Example

```
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

```
Enter the age:12
The age is not valid
```

```
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

```
Enter the age:22
the age is valid
```

Example : Raise the exception with message

```
try:
     num = int(input("Enter a positive integer: "))
     if(num <= 0):
# we can pass the message in the raise statement
          raise ValueError("That is  a negative number!")
except ValueError as e:
     print(e)
```

```
Enter a positive integer: 2
```

```
try:
     num = int(input("Enter a positive integer: "))
     if(num <= 0):
# we can pass the message in the raise statement
          raise ValueError("That is  a negative number!")
except ValueError as e:
     print(e)
```

```
Enter a positive integer: -4
That is  a negative number!
```

Example

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    if b == 0:
        raise ArithmeticError
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

```
Enter a:12
Enter b:0
The value of b can't be 0
```

Assertions in Python

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.

The **simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.**

Assertions are made via the assert statement, which was added in **Python 1.5** as the **latest keyword**.

Assertions are commonly **used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.**

assert Statement

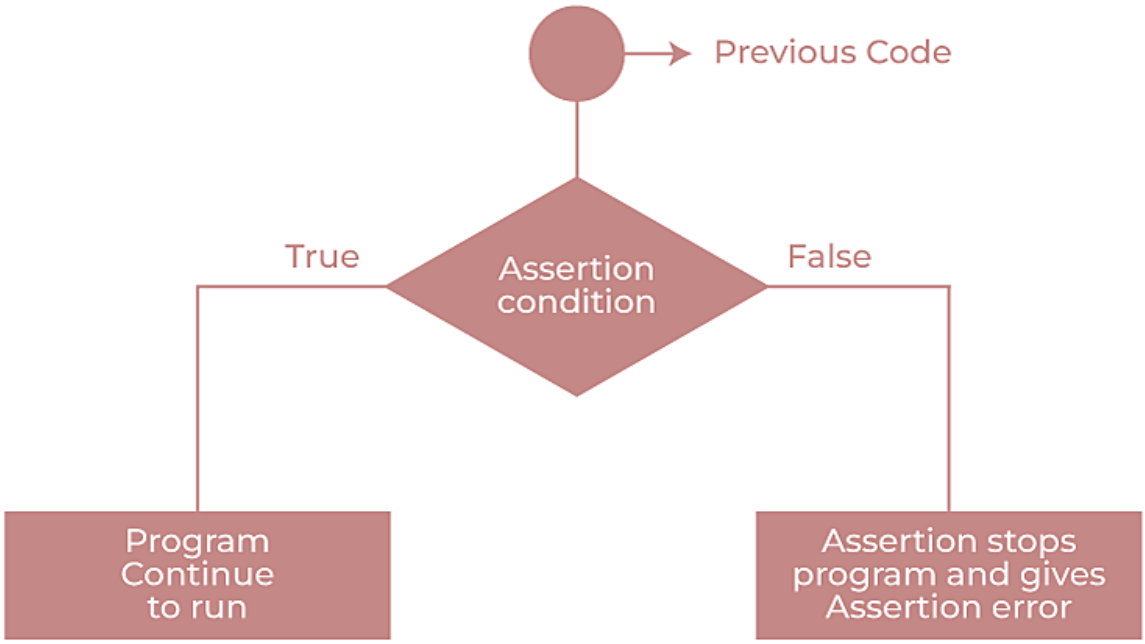
Python **examines the adjacent expression, preferably true when it finds an assert statement.** Python **throws an AssertionError exception if the result of the expression is false.**

Syntax:

assert Expressions[, Argument]

Python **uses ArgumentException, if the assertion fails, as the argument for the AssertionError. We can use the try-except clause to catch and handle AssertionError exceptions, but if they aren't, the program will stop, and the Python interpreter will generate a traceback.**

The assert keyword is used during debugging.



```python
#Python program to show how to use assert keyword
# defining a function
def div(a,b):
    assert ( b!= 0), "denominator should be > 0"
    return a/b

#Calling function and passing the values
print(div(4,2))
#print( square_root(-4))
```

```
2.0
```

Example:

```python
#Python program to show how to use assert keyword
# defining a function
def div(a,b):
    assert ( b!= 0), "denominator should be > 0"
    return a/b

#Calling function and passing the values
print(div(4,0))
#print( square_root(-4))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-9-d233c3eb09fc> in <module>
      6
      7 #Calling function and passing the values
----> 8 print(div(4,0))
      9 #print( square_root(-4))

<ipython-input-9-d233c3eb09fc> in div(a, b)
      2 # defining a function
      3 def div(a,b):
----> 4     assert ( b!= 0), "denominator should be > 0"
      5     return a/b
      6

AssertionError: denominator should be > 0
```

Example:

```
1  # initializing list of foods temperatures
2  batch = [ 40, 26, 39, 30, 25, 21]
3
4  # initializing cut temperature
5  cut = 26
6
7  # using assert to check for temperature greater than cut
8  for i in batch:
9      assert i >= cut, "Batch is Rejected"
10     print (str(i) + " is O.K" )
```

```
40 is O.K
26 is O.K
39 is O.K
30 is O.K

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-7-8b65fde3ddf5> in <module>
      7 # using assert to check for temperature greater than cut
      8 for i in batch:
----> 9     assert i >= cut, "Batch is Rejected"
     10     print (str(i) + " is O.K" )

AssertionError: Batch is Rejected
```