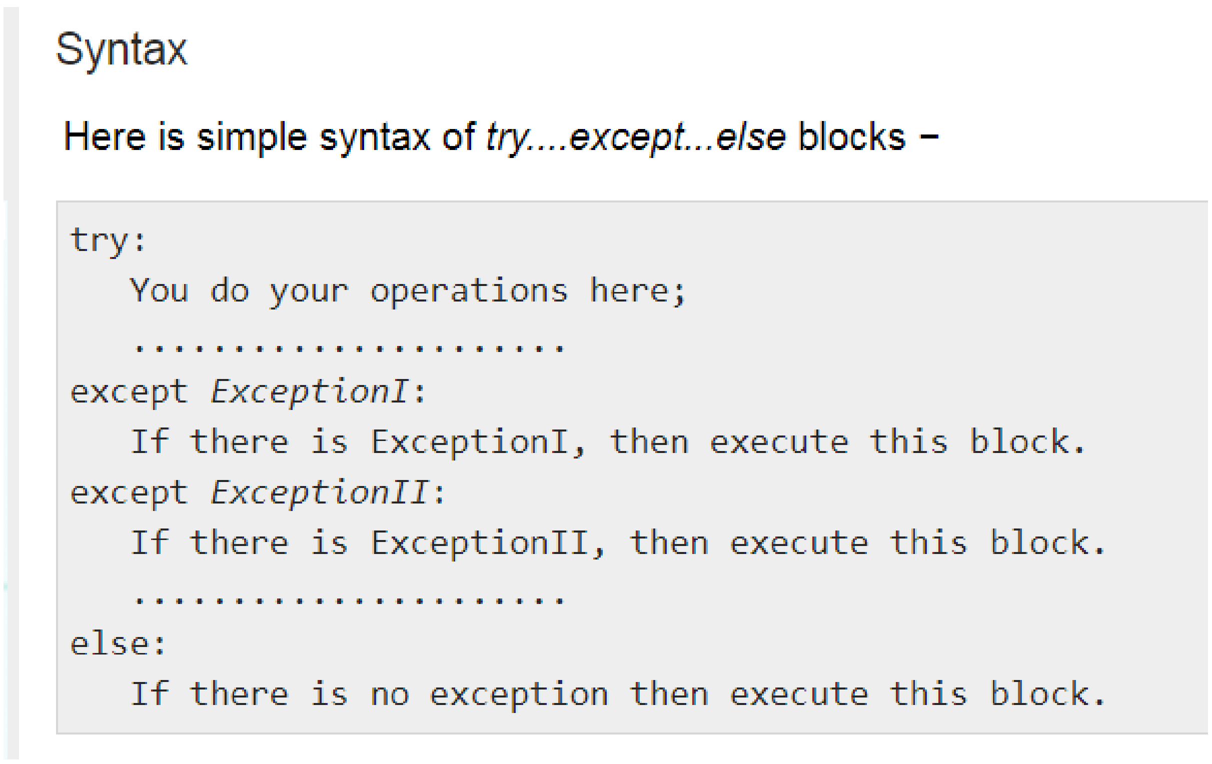
EXCEPTION HANDLING IN PYTHON

- Errors are the problems in a program due to which the program will stop the execution.

Two types of Error occurs in python.

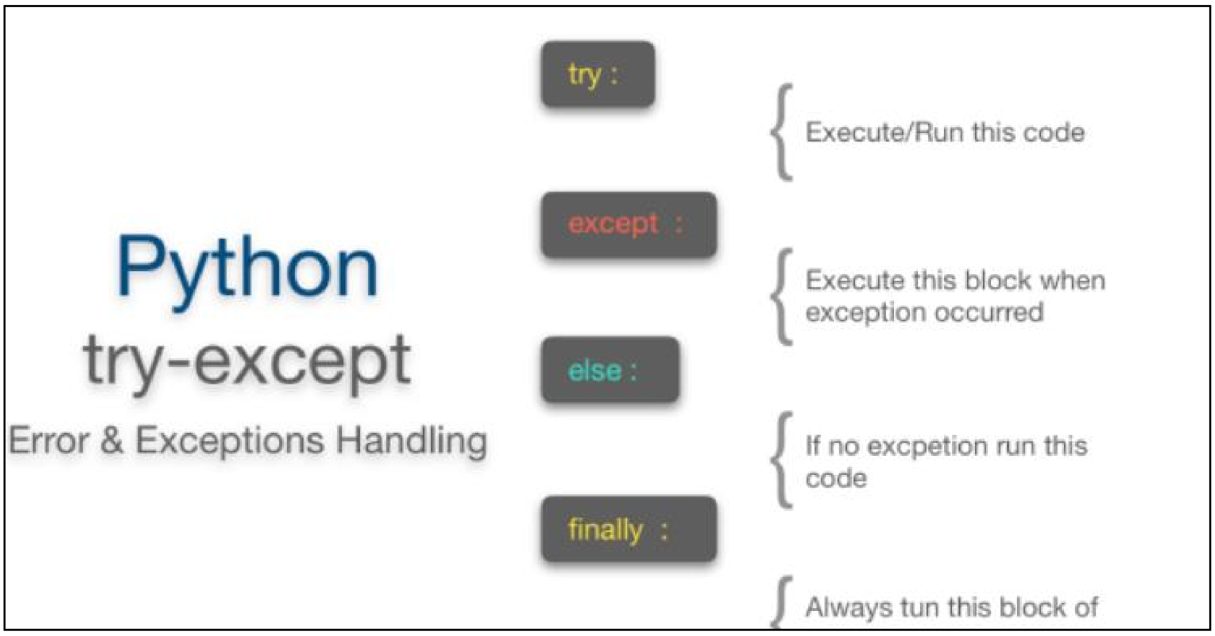
- Syntax errors(parsing errors)
- Logical errors (Exceptions)
- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.



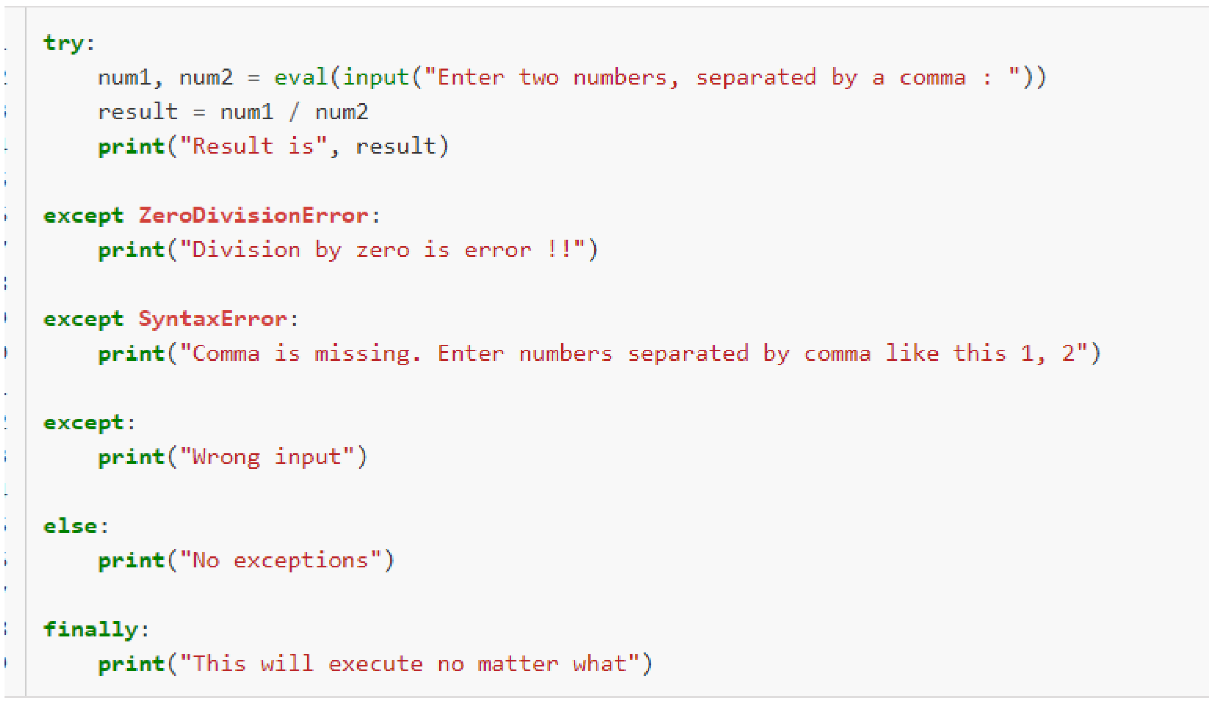
Syntax

Here is simple syntax of *try....except...else* blocks −

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```


Python
try-except
Error & Exceptions Handling
try :
Execute/Run this code
except :
Execute this block when exception occurred
else :
If no excpetion run this code
finally :
Always tun this block of

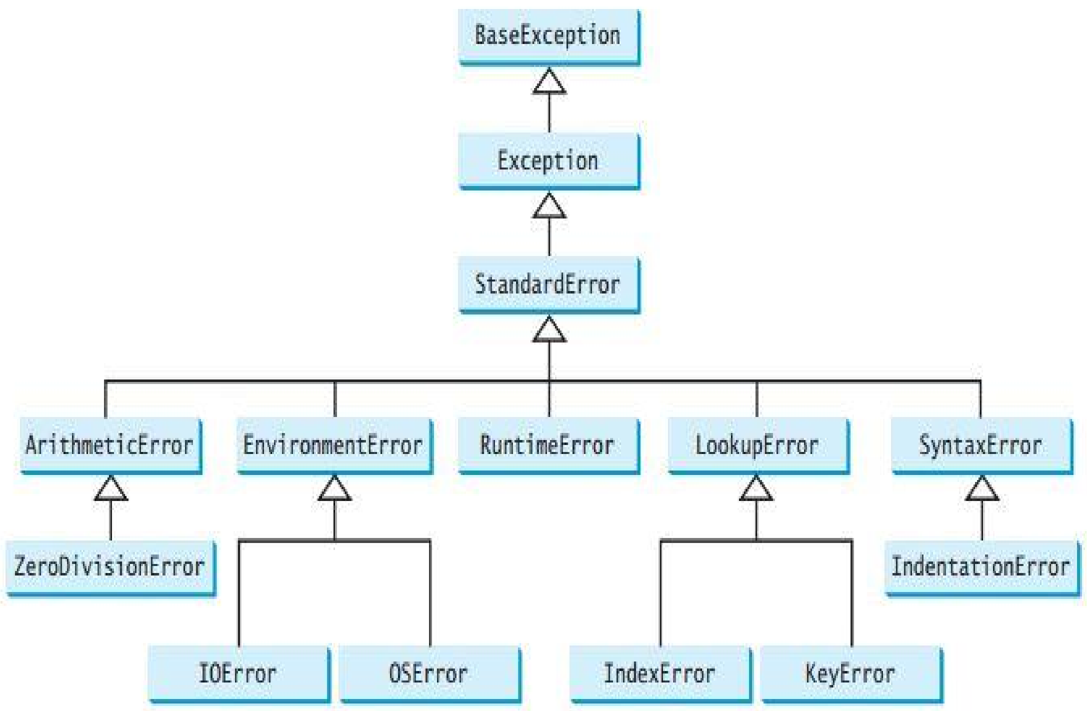
```python
try:
    f = open('somefile.txt', 'r')
    print(f.read())
    f.close()
except IOError:
    print('file not found')
```



```python
try:
    num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
    result = num1 / num2
    print("Result is", result)

except ZeroDivisionError:
    print("Division by zero is error !!")

except SyntaxError:
    print("Comma is missing. Enter numbers separated by comma like this 1, 2")

except:
    print("Wrong input")

else:
    print("No exceptions")

finally:
    print("This will execute no matter what")
```

COMMON EXCEPTIONS:

- **ZeroDivisionError**: Occurs when a number is divided by zero.
- **NameError:** It occurs when a name is not found. It may be local or global.
- **IndentationError:** If incorrect indentation is given.
- **IOError:** It occurs when Input Output operation fails.
- **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.



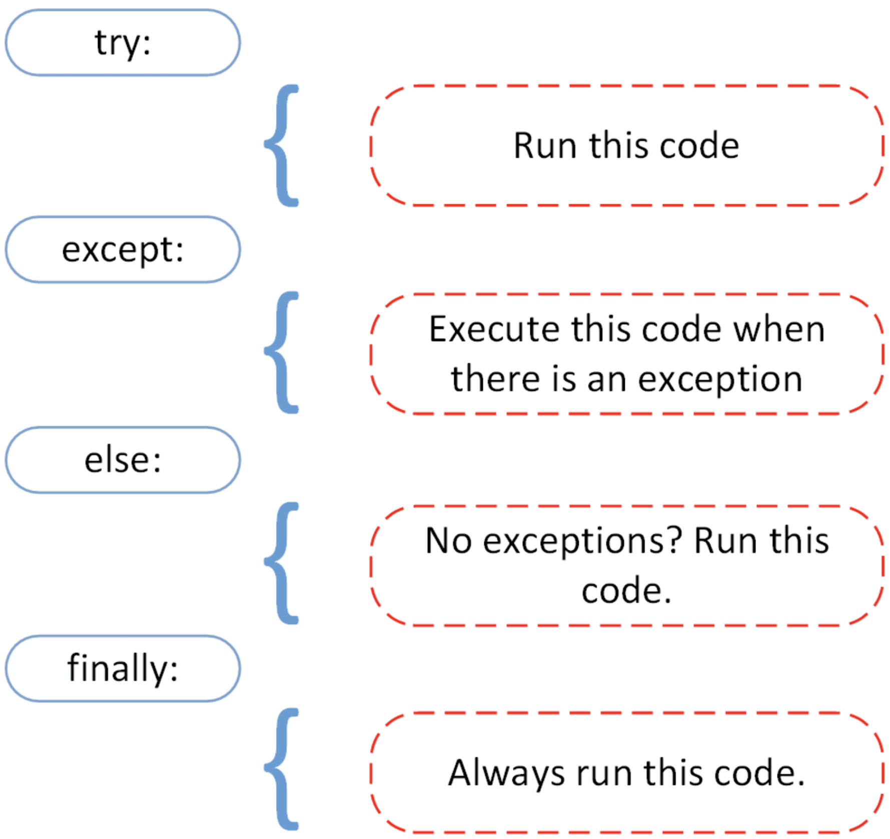
Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

EXAMPLE:

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

Output:

```
Enter a:10
Enter b:0
Can't divide with zero
```


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

RAISE AN EXCEPTION:

- As a Python developer you can choose to throw an exception if a condition occurs.
- To throw (or raise) an exception, use the raise keyword.
- We can use raise to throw an exception if a condition occurs. The statement can be complemented with a custom exception.

-
  ```
  x = -1

  if x < 0:
    raise Exception("Sorry, no numbers below zero")
  ```

-
  ```
  x = "hello"

  if not type(x) is int:
    raise TypeError("Only integers are allowed")
  ```

```
# Program to depict else clause with try-except
# Function which returns a/b
```

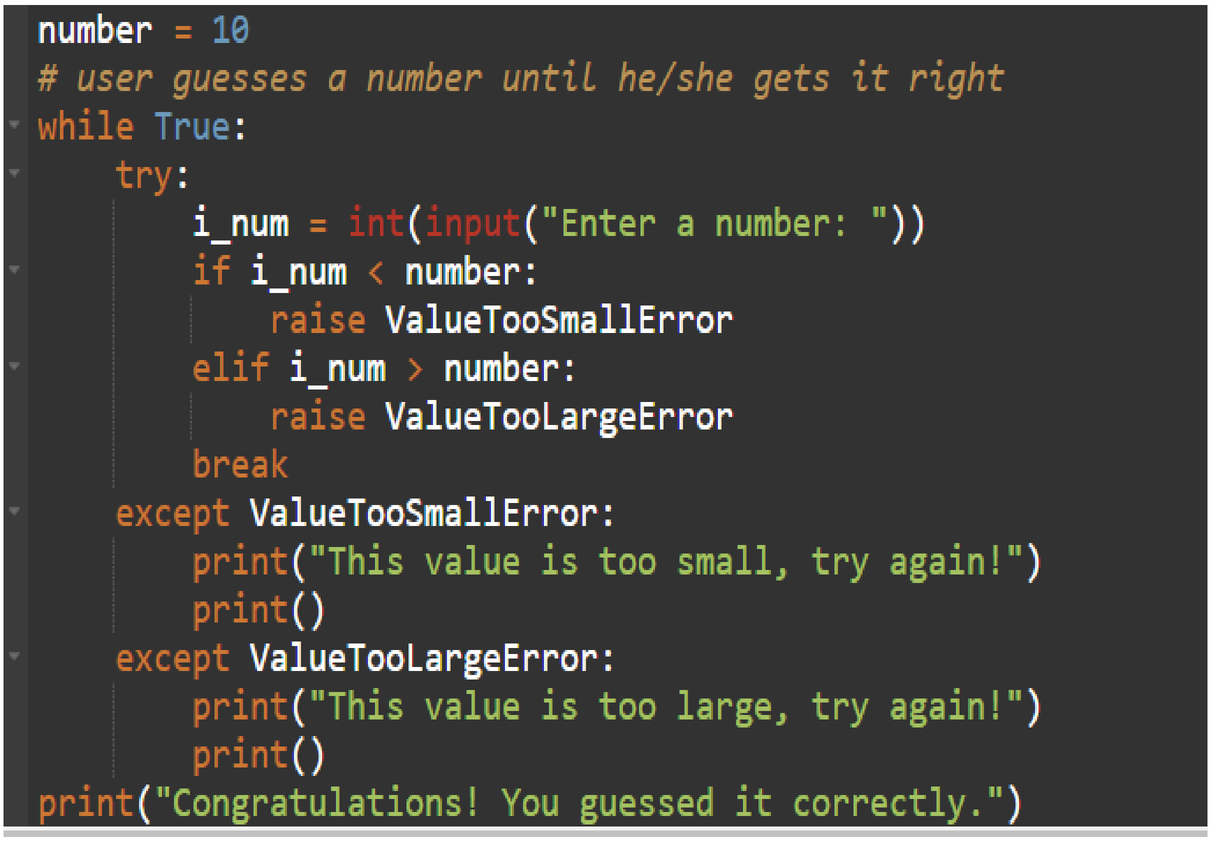
```
def AbyB(a , b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print "a/b result is 0"
    else:
        print c
# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

OUTPUT:

```
-5.0

a/b result is 0
```

USER DEFINED EXCEPTION

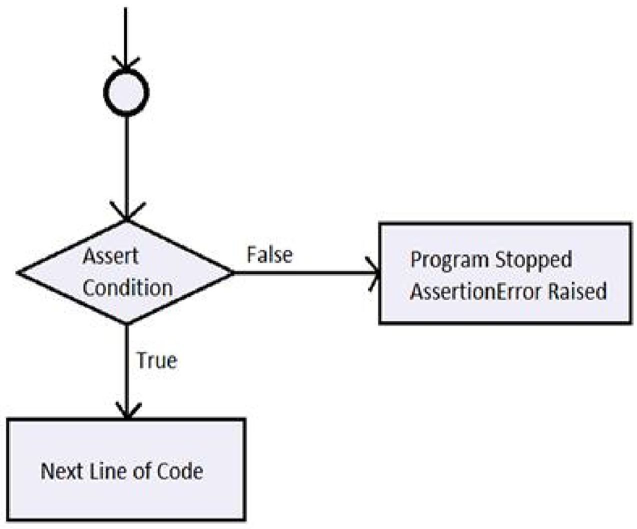
- Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class, either directly or indirectly.



```
number = 10
# user guesses a number until he/she gets it right
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()
print("Congratulations! You guessed it correctly.")
```

ASSERTION IN PYTHON

- An Assertion in Python or a Python Assert Statement is one which asserts (or tests the trueness of) a condition in your code. This is a Boolean expression that confirms the Boolean output of a condition.



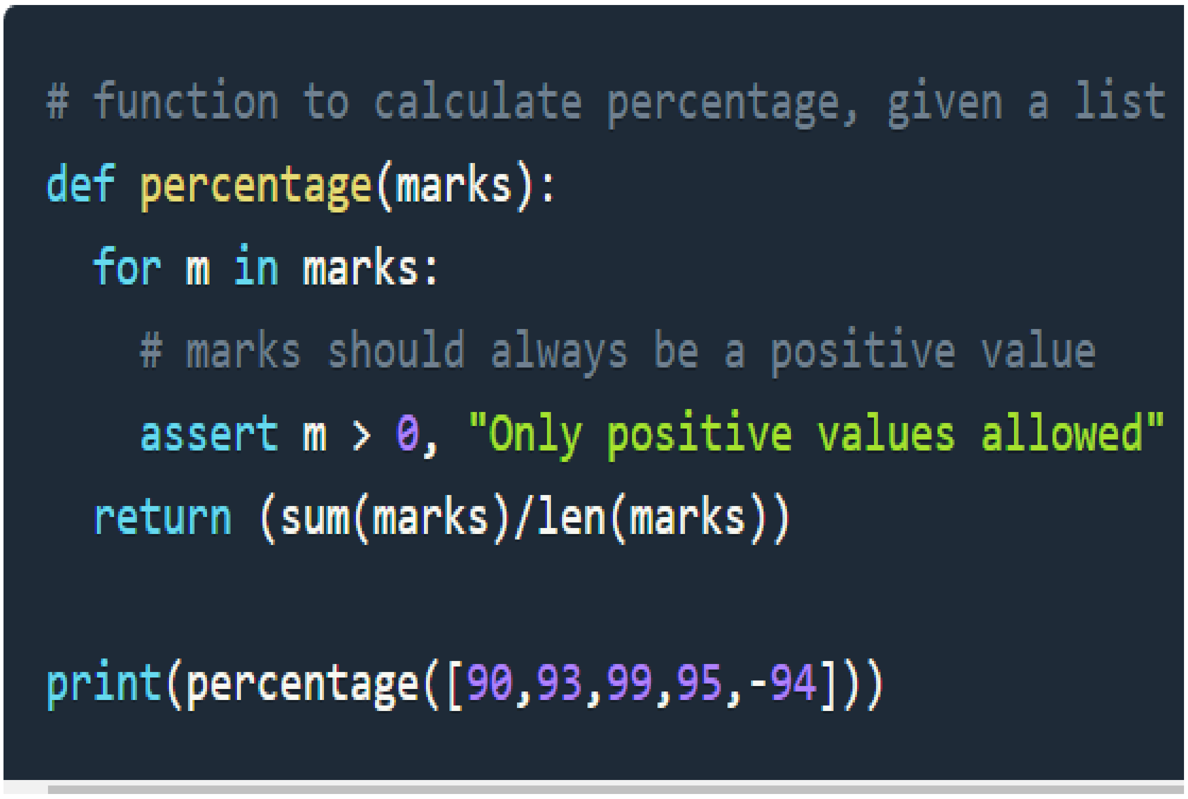
Where Assertion in Python used?

- In checking types/ in checking valid input.
- In checking values of arguments.
- Checking outputs of functions.
- As a debugger to halt where an error occurs.
- In testing code.
- In detecting abuse of an interface by another programmer.

Using assert without Error Message:

```
def avg(marks):
        assert len(marks) != 0
        return sum(marks)/len(marks)
mark1 = []
 print("Average of mark1:",avg(mark1))
```

Using assert with Error Message



```
# function to calculate percentage, given a list
def percentage(marks):
  for m in marks:
    # marks should always be a positive value
    assert m > 0, "Only positive values allowed"
  return (sum(marks)/len(marks))

print(percentage([90,93,99,95,-94]))
```

LOGGING AN EXCEPTION

- Logging, in software applications, is a way to track events. Before we can proceed, telling you more about it, we want to exemplify.
- To log an exception in Python we can use logging module and through that we can log the error.
- Logging an exception in python with an error can be done in the logging. exception() method. This function logs a message with level ERROR on this logger. The arguments are interpreted as for debug(). Exception info is added to the logging message. This method should only be called from an exception handler

PURPOSES OF LOGGING IN PYTHON

- Diagnostic Logging- To record events that revolve around the application's operation.
- Audit Logging- To record events for business analysis.

- Logging module provides a set of functions for simple logging and for following purposes

➢ DEBUG

➢ INFO

➢ WARNING

➢ ERROR

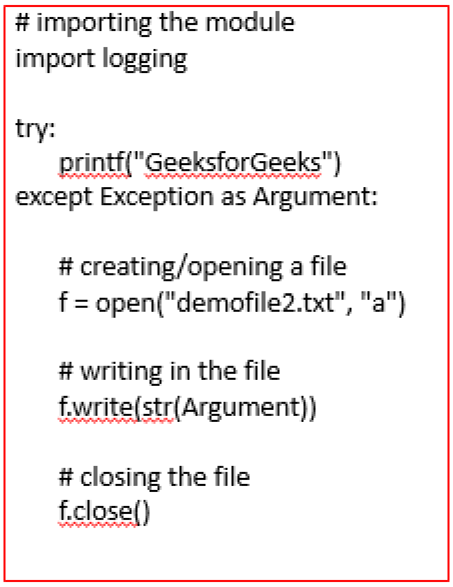
➢ CRITICAL

PYTHON LOGGING FUNCTIONS

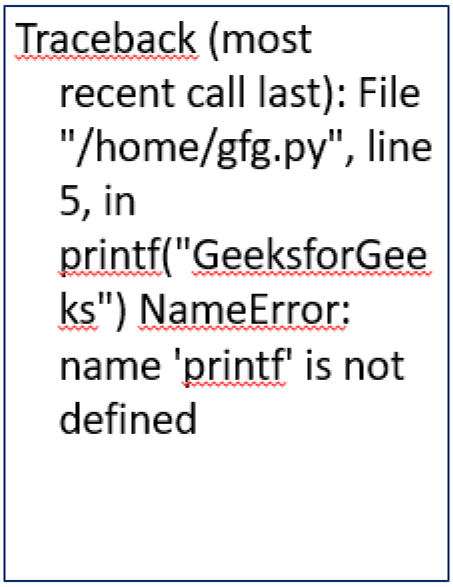
- logging.info() or logging.debug() for the detailed output of events that occur during normal operation of a program.
- warnings.warn() issues a warning for a runtime event if the issue is avoidable.
- logging.warning() issues a warning for a runtime event if we need to note the event even when the client can do nothing about it.
- logging.error(), logging.exception(), or logging.critical() report the suppression of an error without raising an exception.

```
# importing the module
import logging
try:
    printf(“Hello")
except Exception as Argument:
    logging.exception("Error occured while printing Hello")
ERROR:root:Error occured while printing GeeksforGeeks Traceback (most
recent call last): File "/home/gfg.py", line 3, in printf("GeeksforGeeks")
NameError: name 'printf' is not defined
```

We can also log the error message into different file without showing error in the console by the following method:



```
# importing the module
import logging

try:
    printf("GeeksforGeeks")
except Exception as Argument:

    # creating/opening a file
    f = open("demofile2.txt", "a")

    # writing in the file
    f.write(str(Argument))

    # closing the file
    f.close()
```



```
Traceback (most
    recent call last): File
    "/home/gfg.py", line
    5, in
    printf("GeeksforGee
    ks") NameError:
    name 'printf' is not
    defined
```

Logging Variable Data:
dynamic information from application in the logs.
import logging
 name = 'John‘
 logging.error('%s raised an error', name)
ERROR:root:John raised an error
Displaying Date/Time For Python Logging:

- logging.basicConfig(format=’%(asctime)s %(message)s’)

FILE IN PYTHON

- A file is a chunk of logically related data or information which can be used by computer programs.
- Files on most modern file systems are composed of three main parts:
- Header: metadata about the contents of the file (file name, size, type, and so on)
- Data: contents of the file as written by the creator or editor

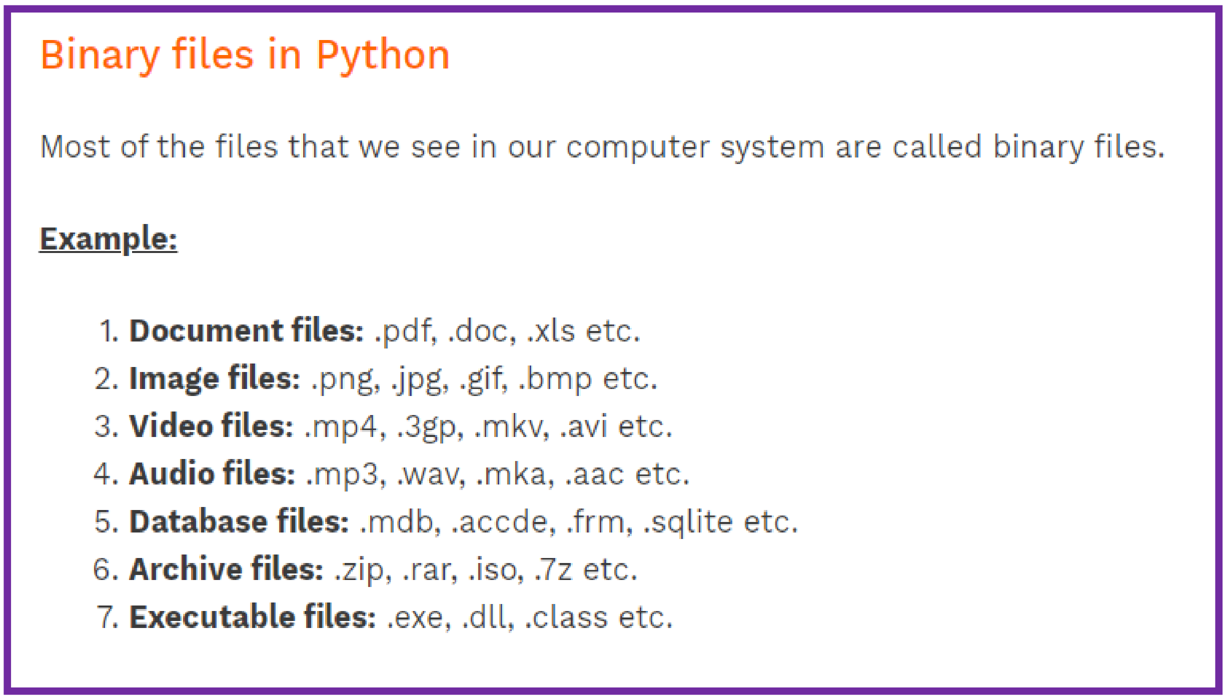
- End of file (EOF): special character that indicates the end of the file

In Python, there is no need for importing external library to read and write files. Python provides an inbuilt function for creating, writing, and reading files.

- Python has several functions for creating, reading, updating, and deleting files.
- There are two types of files in Python

❖ Binary file

❖ Text file

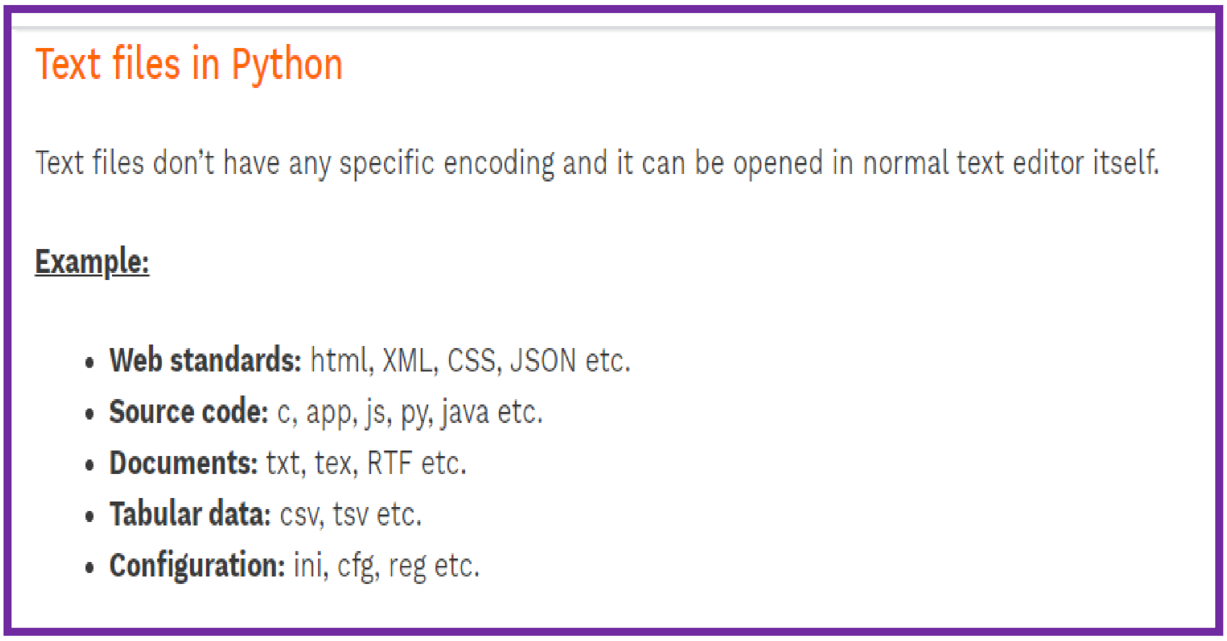


Binary files in Python

Most of the files that we see in our computer system are called binary files.

Example:

1. **Document files:** .pdf, .doc, .xls etc.
2. **Image files:** .png, .jpg, .gif, .bmp etc.
3. **Video files:** .mp4, .3gp, .mkv, .avi etc.
4. **Audio files:** .mp3, .wav, .mka, .aac etc.
5. **Database files:** .mdb, .accde, .frm, .sqlite etc.
6. **Archive files:** .zip, .rar, .iso, .7z etc.
7. **Executable files:** .exe, .dll, .class etc.

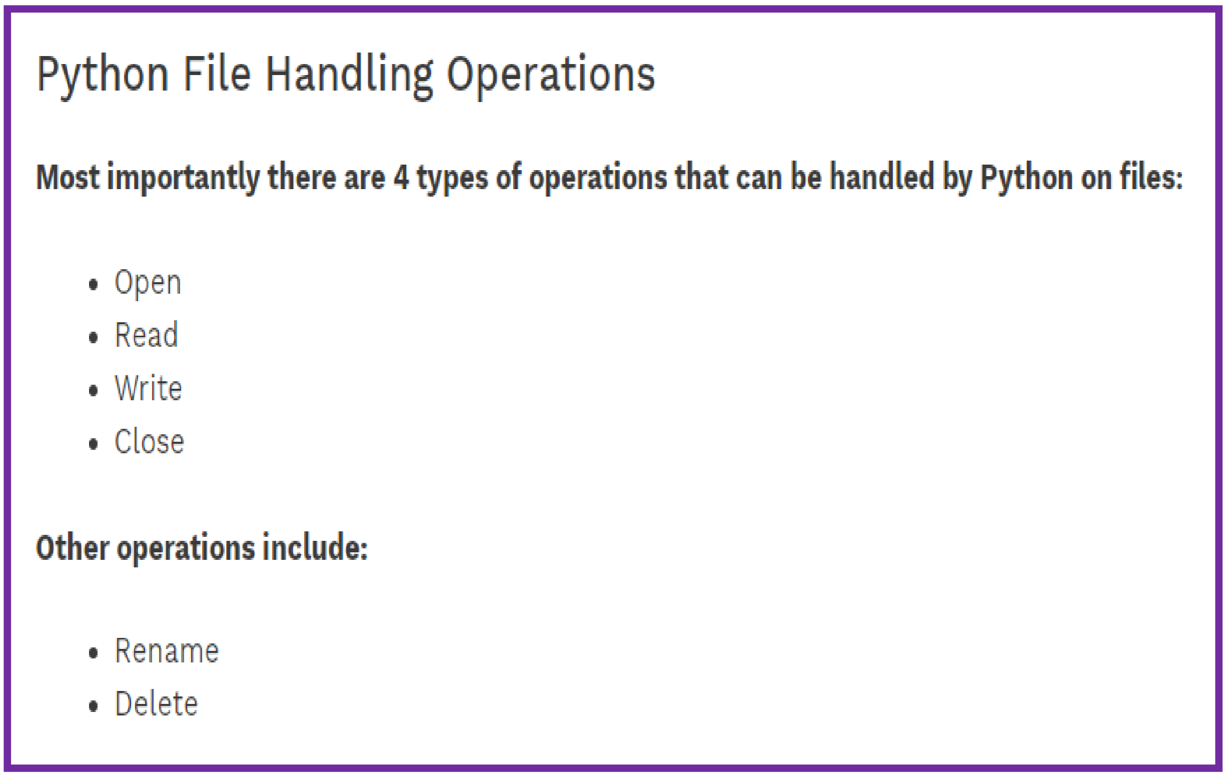


Text files in Python

Text files don't have any specific encoding and it can be opened in normal text editor itself.

Example:

- **Web standards:** html, XML, CSS, JSON etc.
- **Source code:** c, app, js, py, java etc.
- **Documents:** txt, tex, RTF etc.
- **Tabular data:** csv, tsv etc.
- **Configuration:** ini, cfg, reg etc.



Python File Handling Operations

Most importantly there are 4 types of operations that can be handled by Python on files:

- Open
- Read
- Write
- Close

Other operations include:

- Rename
- Delete

***FILE** open() function*

The Python file open function returns a file object that contains methods and attributes to perform various operations for opening files in Python.

file_object = open("filename", "mode")
Here,

- **filename:** gives name of the file that the file object has opened.
- **mode:** attribute of a file object tells you which mode a file was opened in.

File Modes:

MODES	DESCRIPTION
<r>	It opens a file in read-only mode while the file offset stays at the root.
<rb>	It opens a file in (binary + read-only) modes. And the offset remains at the root level.
<r+>	It opens the file in both (read + write) modes while the file offset is again at the root level.
<rb+>	It opens the file in (read + write + binary) modes. The file offset is again at the root level.
<w>	It allows write-level access to a file. If the file already exists, then it'll get overwritten. It'll create a new file if the same doesn't exist.
<wb>	Use it to open a file for writing in binary format. Same behavior as for write-only mode.
<w+>	It opens a file in both (read + write) modes. Same behavior as for write-only mode.
<wb+>	It opens a file in (read + write + binary) modes. Same behavior as for write-only mode.

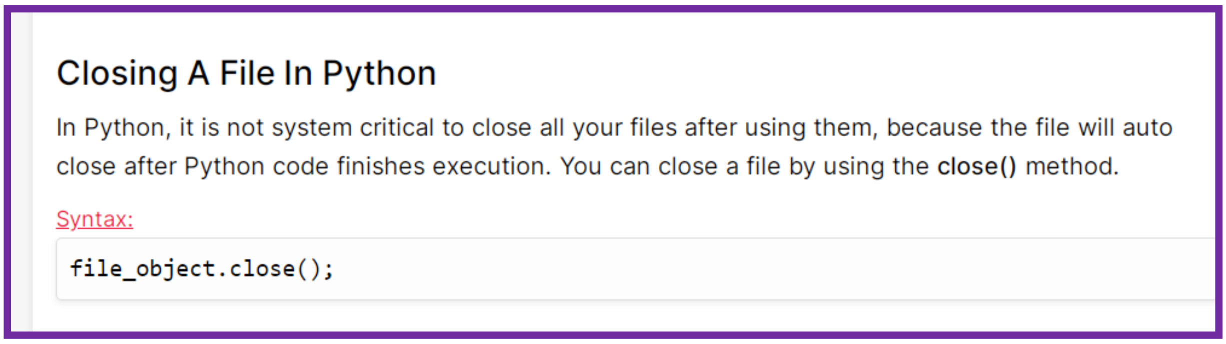
<a>	It opens the file in append mode. The offset goes to the end of the file. If the file doesn't exist, then it gets created.
<ab>	It opens a file in (append + binary) modes. Same behavior as for append mode.
<a+>	It opens a file in (append + read) modes. Same behavior as for append mode.
<ab+>	It opens a file in (append + read + binary) modes. Same behavior as for append mode.

Syntax:

- **file_object = open(file_name, mode)**

EG

- **f = open("demofile.txt", "r")**

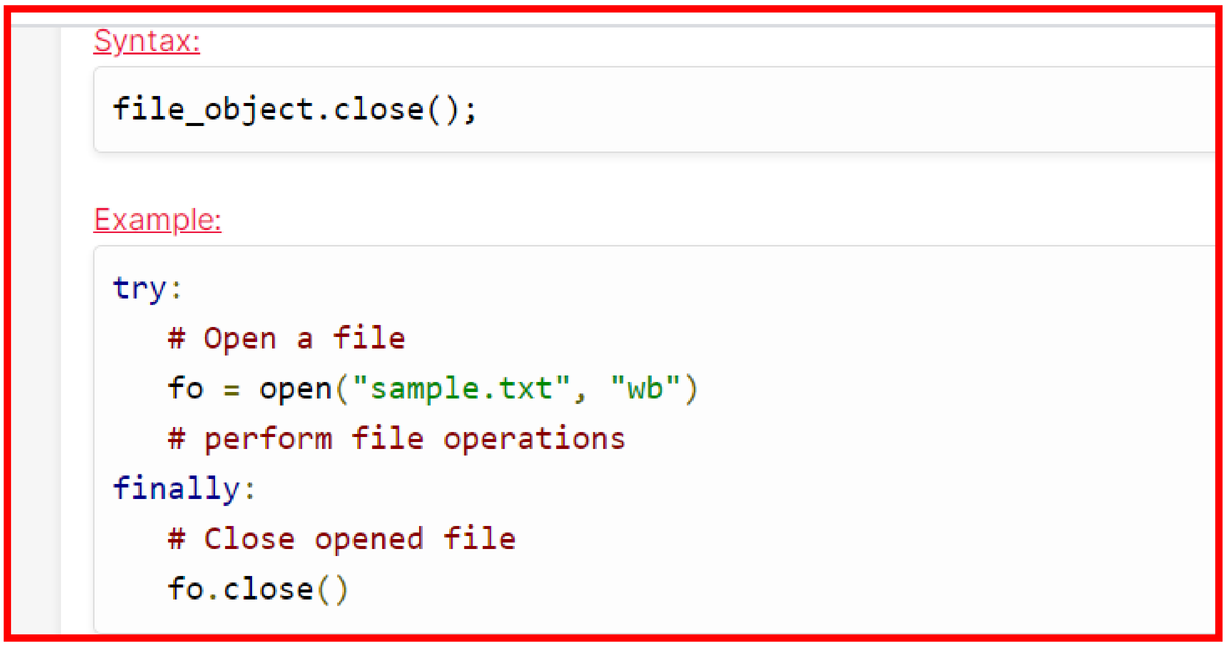


Closing A File In Python

In Python, it is not system critical to close all your files after using them, because the file will auto close after Python code finishes execution. You can close a file by using the **close()** method.

Syntax:

```
file_object.close();
```


Syntax:

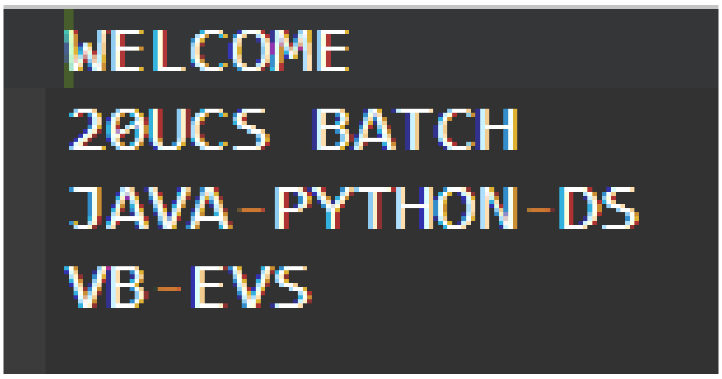
```
file_object.close();
```

Example:

```
try:
   # Open a file
   fo = open("sample.txt", "wb")
   # perform file operations
finally:
   # Close opened file
   fo.close()
```

EXAMPLE:

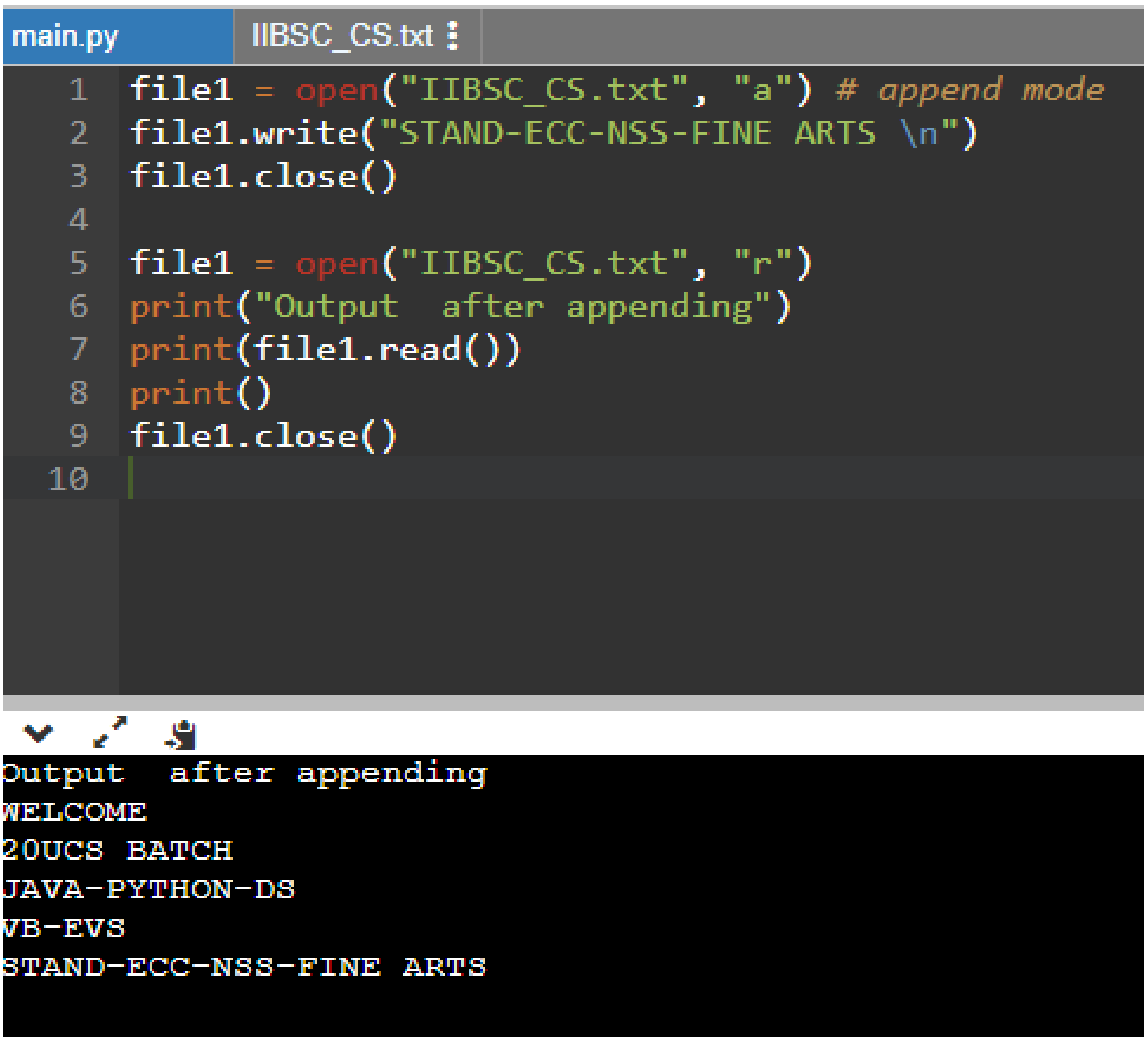
```
# Python program to demonstrate File Concept
file1 = open('IIBSC_CS.txt', 'w')
L = [“20UCS BATCH \n", "JAVA-PYTHON-DS \n", "VB-EVS \n"]
s = "WELCOME\n"
# Writing a string to file
file1.write(s)
# Writing multiple strings  at a time
file1.writelines(L)
# Closing file
file1.close()
file1 = open('IIBSC_CS.txt', 'r')
print(file1.read())
file1.close()
```



```
WELCOME
20UCS BATCH
JAVA-PYTHON-DS
VB-EVS
```

APPEND

```
# Python program to illustrate # Append
 # Append-adds at last
file1 = open("IIBSC_CS.txt", "a") # append mode
file1.write("STAND-ECC-NSS-FINE ARTS \n")
file1.close()
file1 = open("IIBSC_CS.txt", "r")
print("Output  after appending")
print(file1.read())
print()
file1.close()
```



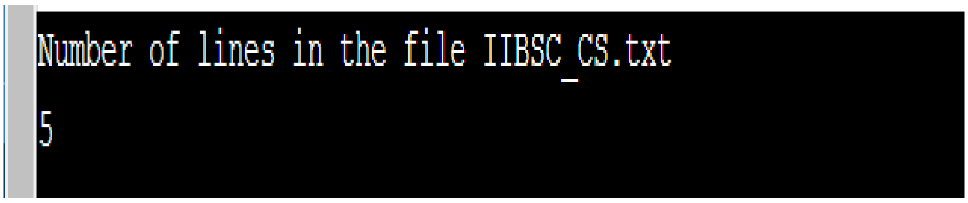
```
file1 = open("IIBSC_CS.txt", "a") # append mode
file1.write("STAND-ECC-NSS-FINE ARTS \n")
file1.close()

file1 = open("IIBSC_CS.txt", "r")
print("Output  after appending")
print(file1.read())
print()
file1.close()
```

```
Output  after appending
WELCOME
20UCS BATCH
JAVA-PYTHON-DS
VB-EVS
STAND-ECC-NSS-FINE ARTS
```

EXAMPLE:

Python program to count the number of lines in a text file

```
# Opening a file
file = open("IIBSC_CS.txt","r")
Counter = 0
# Reading from file
Content = file.read()
CoList = Content.split("\n")
for i in CoList:
    if i:
        Counter += 1
print("Number of lines in the file IIBSC_CS.txt")
print(Counter)
```

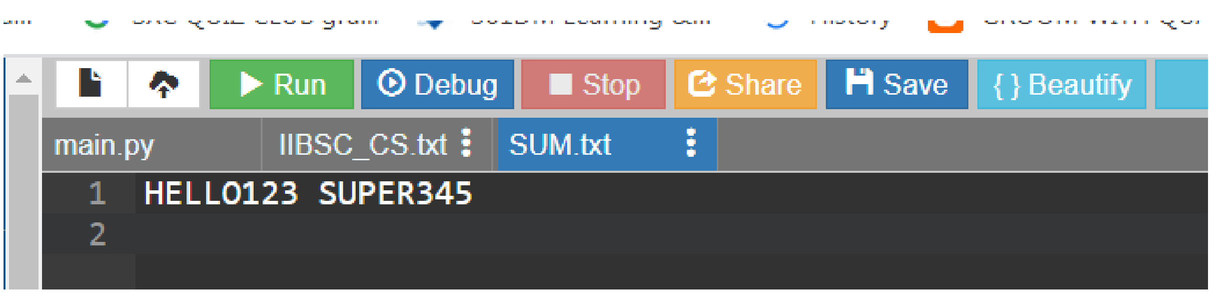


```
Number of lines in the file IIBSC_CS.txt
5
```

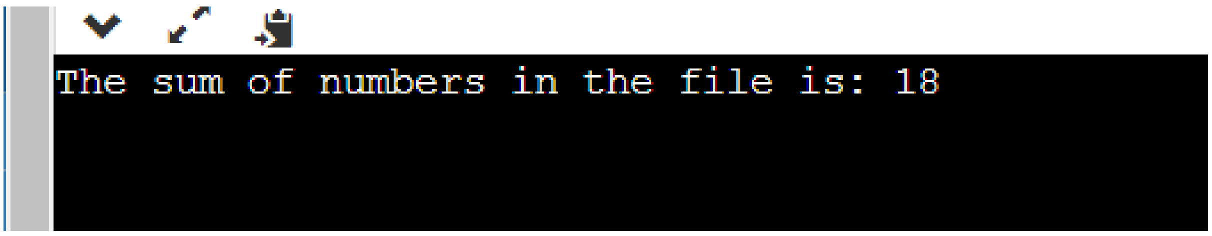
EXAMPLE:

PROGRAM TO DISPLAY SUM OF DIGITS IN THE TEXT FILE

```
# Python program for writing to file
file = open('SUM.txt', 'w')
data ='HELLO123 SUPER345'
file.write(data)
file.close()
# Python program for reading from file
h = open('SUM.txt', 'r')
content = h.readlines()
a = 0
for line in content:
    for i in line:
        if i.isdigit() == True:
            a += int(i)
print("The sum of numbers in the file is:", a)
```



```
HELLO123 SUPER345
```



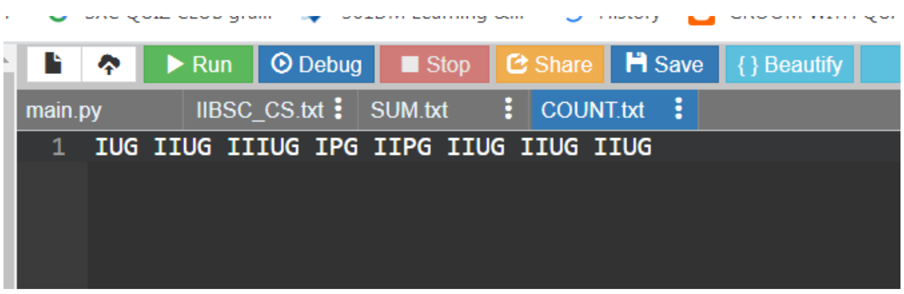
```
The sum of numbers in the file is: 18
```

EXAMPLE

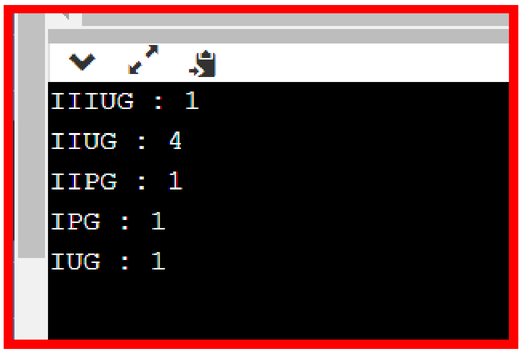
Count the number of occurrence of the words in text file

```
file = open('COUNT.txt', 'w')
data ='IUG IIUG IIIUG IPG IIPG IIUG IIUG IIUG'
file.write(data)
file.close()
text = open("COUNT.txt", "r")
d = dict()
for line in text:
    line = line.strip()
    line = line.upper()
    words = line.split(" ")
    for word in words:
        if word in d:
            d[word] = d[word] + 1
        else:
            d[word] = 1
```

```
for key in list(d.keys()):
        print(key, ":", d[key])
```



```
IUG IIUG IIIUG IPG IIPG IIUG IIUG IIUG
```



```
IIIUG : 1
IIUG : 4
IIPG : 1
IPG : 1
IUG : 1
```

FILE METHODS:

close()	Closes the file
detach()	Returns the separated raw stream from the buffer
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not

read()	Returns the file content
readable()	Returns whether the file stream can be read or not
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file
seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writable()	Returns whether the file can be written to or not
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file

FILE:RANDOM ACCESS

FILE

The `tell()` method returns the current file position in a file stream.

file.tell()

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.tell())
```

seek()

move the file pointer to another position. fileObject.seek(offset,from_what))

offset – A number of positions will move.

from_what – defines your point of reference. (Optional)

```
f = open("testFile.txt", "r")

f.seek(9)

print(f.readline())
```

Using the `mmap` module allows the user to randomly access locations in a file by mapping the file into memory. This is an alternative to using normal file operations.

```
import mmap

with open('filename.ext', 'r') as fd:
    # 0: map the whole file
    mm = mmap.mmap(fd.fileno(), 0)

    # print characters at indices 5 through 10
    print mm[5:10]
```

```
# print the line starting from mm's current position
print mm.readline()

# write a character to the 5th index
mm[5] = 'a'

# return mm's position to the beginning of the file
mm.seek(0)

# close the mmap object
mm.close()
```

Zipping and Unzipping files in Python

ZIP is an archive file format that supports lossless data compression. By lossless compression, it means that the

compression algorithm used allows the original data to be perfectly reconstructed from the compressed data.

Also, a **ZIP** file may contain one or more files or directories that may have been compressed.

Why do we need zip files?

- To reduce storage requirements.
- To improve transfer speed over standard connections.

```
# importing required modules

from zipfile import ZipFile


# specifying the zip file name

file_name = "my_python_files.zip"


# opening the zip file in READ mode

with ZipFile(file_name, 'r') as zip:

        # printing all the contents of the zip file

        zip.printdir()
```

```
    # extracting all the files
    print('Extracting all the files now...')
    zip.extractall()
    print('Done!')
```

Unzipping the File

```
import zipfile
def un_zipFiles(path):
    files=os.listdir(path)
    for file in files:
        if file.endswith('.zip'):
            filePath=path+'/'+file
            zip_file = zipfile.ZipFile(filePath)
            for names in zip_file.namelist():
                zip_file.extract(names,path)
            zip_file.close()
```