**What is a file?**

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

**How to open a file?**

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

>>> f = open("test.txt") # open file in current directory

>>> f = open("C:/Python33/README.txt") # specifying full path

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

|  |  |
| --- | --- |
| Python File Modes | |
| Mode | Description |
| 'r' | Open a file for reading. (default) |
| 'w' | Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists. |
| 'x' | Open a file for exclusive creation. If the file already exists, the operation fails. |
| 'a' | Open for appending at the end of the file without truncating it. Creates a new file if it does not exist. |
| 't' | Open in text mode. (default) |
| 'b' | Open in binary mode. |
| '+' | Open a file for updating (reading and writing) |

f = open("test.txt") # equivalent to 'r' or 'rt'

f = open("test.txt",'w') # write in text mode

f = open("img.bmp",'r+b') # read and write in binary mode

Unlike other languages, the character 'a' does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).

Moreover, the default encoding is platform dependent. In windows, it is 'cp1252' but 'utf-8' in Linux.

So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

f = open("test.txt",mode = 'r',encoding = 'utf-8')

## The *file* Object Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object −

|  |  |
| --- | --- |
| **Sr.No.** | **Attribute & Description** |
| 1 | **file.closed**  Returns true if file is closed, false otherwise. |
| 2 | **file.mode**  Returns access mode with which file was opened. |
| 3 | **file.name**  Returns name of the file. |
| 4 | **file.softspace**  Returns false if space explicitly required with print, true otherwise. |

### Example

#!/usr/bin/python

# Open a file

fo = open("foo.txt", "wb")

print ("Name of the file: ", fo.name)

print ("Closed or not : ", fo.closed)

print ("Opening mode : ", fo.mode)

print ("Softspace flag : ", fo.softspace)

This produces the following result −

Name of the file: foo.txt

Closed or not : False

Opening mode : wb

Softspace flag : 0

## The *close()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

### Syntax

fileObject.close();

### Example

#!/usr/bin/python

# Open a file

fo = open("foo.txt", "wb")

print ("Name of the file: ", fo.name)

# Close opend file

fo.close()

This produces the following result −

Name of the file: foo.txt

## Reading and Writing Files

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read()* and *write()* methods to read and write files.

## The *write()* Method

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string −

### Syntax

fileObject.write(string);

Here, passed parameter is the content to be written into the opened file.

### Example

#!/usr/bin/python

# Open a file

fo = open("foo.txt", "wb")

fo.write( "Python is a great language.\nYeah its great!!\n");

# Close opend file

fo.close()

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

Python is a great language.

Yeah its great!!

## The *read()* Method

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

### Syntax

fileObject.read([count]);

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

### Example

Let's take a file *foo.txt*, which we created above.

#!/usr/bin/python

# Open a file

fo = open("foo.txt", "r+")

str = fo.read(10);

print "Read String is : ", str

# Close opend file

fo.close()

This produces the following result −

Read String is : Python is

## File Positions

The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek(offset[, from])* method changes the current file position. The *offset*argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

### Example

Let us take a file *foo.txt*, which we created above.

#!/usr/bin/python

# Open a file

fo = open("foo.txt", "r+")

str = fo.read(10);

print "Read String is : ", str

# Check current position

position = fo.tell();

print "Current file position : ", position

# Reposition pointer at the beginning once again

position = fo.seek(0, 0);

str = fo.read(10);

print "Again read String is : ", str

# Close opend file

fo.close()

This produces the following result −

Read String is : Python is

Current file position : 10

Again read String is : Python is

## Renaming and Deleting Files

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

## The rename() Method

The *rename()* method takes two arguments, the current filename and the new filename.

### Syntax

os.rename(current\_file\_name, new\_file\_name)

### Example

Following is the example to rename an existing file *test1.txt* −

#!/usr/bin/python

import os

# Rename a file from test1.txt to test2.txt

os.rename( "test1.txt", "test2.txt" )

## The *remove()* Method

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

### Syntax

os.remove(file\_name)

### Example

Following is the example to delete an existing file *test2.txt* −

#!/usr/bin/python

import os

# Delete file test2.txt

os.remove("text2.txt")

## Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

## The *mkdir()* Method

You can use the *mkdir()* method of the **os** module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

### Syntax

os.mkdir("newdir")

### Example

Following is the example to create a directory *test* in the current directory −

#!/usr/bin/python

import os

# Create a directory "test"

os.mkdir("test")

## The *chdir()* Method

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

### Syntax

os.chdir("newdir")

### Example

Following is the example to go into "/home/newdir" directory −

#!/usr/bin/python

import os

# Changing a directory to "/home/newdir"

os.chdir("/home/newdir")

## The *getcwd()* Method

The *getcwd()* method displays the current working directory.

### Syntax

os.getcwd()

### Example

Following is the example to give current directory −

#!/usr/bin/python

import os

# This would give location of the current directory

os.getcwd()

## The *rmdir()* Method

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

### Syntax

os.rmdir('dirname')

### Example

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

#!/usr/bin/python

import os

# This would remove "/tmp/test" directory.

os.rmdir( "/tmp/test" )

## File & Directory Related Methods

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows −

* [File Object Methods](https://www.tutorialspoint.com/python/file_methods.htm): The *file* object provides functions to manipulate files.
* [OS Object Methods](https://www.tutorialspoint.com/python/os_file_methods.htm): This provides methods to process files as well as directories.

**How to close a file Using Python?**

When we are done with operations to the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file and is done using Python close() method.

Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

f = open("test.txt",encoding = 'utf-8')

# perform file operations

f.close()

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

try:

f = open("test.txt",encoding = 'utf-8')

# perform file operations

finally:

f.close()

This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.

The best way to do this is using the with statement. This ensures that the file is closed when the block inside with is exited.

We don't need to explicitly call the close() method. It is done internally.

with open("test.txt",encoding = 'utf-8') as f:

# perform file operations

**How to write to File Using Python?**

In order to write into a file in Python, we need to open it in write 'w', append 'a' or exclusive creation 'x' mode.

We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using write() method. This method returns the number of characters written to the file.

with open("test.txt",'w',encoding = 'utf-8') as f:

f.write("my first file\n")

f.write("This file\n\n")

f.write("contains three lines\n")

This program will create a new file named 'test.txt' if it does not exist. If it does exist, it is overwritten.

We must include the newline characters ourselves to distinguish different lines.

**How to read files in Python?**

To read a file in Python, we must open the file in reading mode.

There are various methods available for this purpose. We can use the read(size) method to read in size number of data. If size parameter is not specified, it reads and returns up to the end of the file.

>>> f = open("test.txt",'r',encoding = 'utf-8')

>>> f.read(4) # read the first 4 data

'This'

>>> f.read(4) # read the next 4 data

' is '

>>> f.read() # read in the rest till end of file

'my first file\nThis file\ncontains three lines\n'

>>> f.read() # further reading returns empty sting

''

We can see that, the read() method returns newline as '\n'. Once the end of file is reached, we get empty string on further reading.

We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).

>>> f.tell() # get the current file position

56

>>> f.seek(0) # bring file cursor to initial position

0

>>> print(f.read()) # read the entire file

This is my first file

This file

contains three lines

We can read a file line-by-line using a [for loop](https://www.programiz.com/python-programming/for-loop). This is both efficient and fast.

>>> for line in f:

... print(line, end = '')

...

This is my first file

This file

contains three lines

The lines in file itself has a newline character '\n'.

Moreover, the print() end parameter to avoid two newlines when printing.

Alternately, we can use readline() method to read individual lines of a file. This method reads a file till the newline, including the newline character.

>>> f.readline()

'This is my first file\n'

>>> f.readline()

'This file\n'

>>> f.readline()

'contains three lines\n'

>>> f.readline()

''

Lastly, the readlines() method returns a list of remaining lines of the entire file. All these reading method return empty values when end of file (EOF) is reached.

>>> f.readlines()

['This is my first file\n', 'This file\n', 'contains three lines\n']

**Python File Methods**

There are various methods available with the file object. Some of them have been used in above examples.

Here is the complete list of methods in text mode with a brief description.

|  |  |
| --- | --- |
| Python File Methods | |
| Method | Description |
| close() | Close an open file. It has no effect if the file is already closed. |
| detach() | Separate the underlying binary buffer from the TextIOBaseand return it. |
| fileno() | Return an integer number (file descriptor) of the file. |
| flush() | Flush the write buffer of the file stream. |
| isatty() | Return True if the file stream is interactive. |
| read(n) | Read atmost n characters form the file. Reads till end of file if it is negative or None. |
| readable() | Returns True if the file stream can be read from. |
| readline(n=-1) | Read and return one line from the file. Reads in at most nbytes if specified. |
| readlines(n=-1) | Read and return a list of lines from the file. Reads in at most n bytes/characters if specified. |
| seek(offset,from=SEEK\_SET) | Change the file position to offset bytes, in reference to from (start, current, end). |
| seekable() | Returns True if the file stream supports random access. |
| tell() | Returns the current file location. |
| truncate(size=None) | Resize the file stream to size bytes. If size is not specified, resize to current location. |
| writable() | Returns True if the file stream can be written to. |
| write(s) | Write string s to the file and return the number of characters written. |
| writelines(lines) | Write a list of lines to the file. |

# Python Directory and Files Management

If there are a large number of files to handle in your Python program, you can arrange your code within different directories to make things more manageable.

A directory or folder is a collection of files and sub directories. Python has the os module, which provides us with many useful methods to work with directories (and files as well).

## Get Current Directory

We can get the present working directory using the getcwd() method.

This method returns the current working directory in the form of a string. We can also use the getcwdb() method to get it as bytes object.

>>> import os

>>> os.getcwd()

'C:\\Program Files\\PyScripter'

>>> os.getcwdb()

b'C:\\Program Files\\PyScripter'

The extra backslash implies escape sequence. The print() function will render this properly.

>>> print(os.getcwd())

C:\Program Files\PyScripter

## Changing Directory

We can change the current working directory using the chdir() method.

The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash (\) to separate path elements.

It is safer to use escape sequence when using the backward slash.

>>> os.chdir('C:\\Python33')

>>> print(os.getcwd())

C:\Python33

## List Directories and Files

All files and sub directories inside a directory can be known using the listdir() method.

This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

>>> print(os.getcwd())

C:\Python33

>>> os.listdir()

['DLLs',

'Doc',

'include',

'Lib',

'libs',

'LICENSE.txt',

'NEWS.txt',

'python.exe',

'pythonw.exe',

'README.txt',

'Scripts',

'tcl',

'Tools']

>>> os.listdir('G:\\')

['$RECYCLE.BIN',

'Movies',

'Music',

'Photos',

'Series',

'System Volume Information']

## Making a New Directory

We can make a new directory using the mkdir() method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

>>> os.mkdir('test')

>>> os.listdir()

['test']

## Renaming a Directory or a File

The rename() method can rename a directory or a file.

The first argument is the old name and the new name must be supplies as the second argument.

>>> os.listdir()

['test']

>>> os.rename('test','new\_one')

>>> os.listdir()

['new\_one']

## Removing Directory or File

A file can be removed (deleted) using the remove() method.

Similarly, the rmdir() method removes an empty directory.

>>> os.listdir()

['new\_one', 'old.txt']

>>> os.remove('old.txt')

>>> os.listdir()

['new\_one']

>>> os.rmdir('new\_one')

>>> os.listdir()

[]

However, note that rmdir() method can only remove empty directories.

In order to remove a non-empty directory we can use the rmtree() method inside the shutil module.

>>> os.listdir()

['test']

>>> os.rmdir('test')

Traceback (most recent call last):

...

OSError: [WinError 145] The directory is not empty: 'test'

>>> import shutil

>>> shutil.rmtree('test')

>>> os.listdir()

[]

# Python Errors and Built-in Exceptions

When writing a program, we, more often than not, will encounter errors.

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

>>> if a < 3

File "<interactive input>", line 1

if a < 3

^

SyntaxError: invalid syntax

We can notice here that a colon is missing in the if statement.

Errors can also occur at runtime and these are called exceptions. They occur, for example, when a file we try to open does not exist (FileNotFoundError), dividing a number by zero (ZeroDivisionError), module we try to import is not found (ImportError) etc.

Whenever these type of runtime error occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

>>> 1 / 0

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

ZeroDivisionError: division by zero

>>> open("imaginary.txt")

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'

## Python Built-in Exceptions

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur. We can view all the built-in exceptions using the local() built-in functions as follows.

>>> locals()['\_\_builtins\_\_']

This will return us a dictionary of built-in exceptions, functions and attributes.

Some of the common built-in exceptions in Python programming along with the error that cause then are tabulated below.

|  |  |
| --- | --- |
| Python Built-in Exceptions | |
| Exception | Cause of Error |
| AssertionError | Raised when assert statement fails. |
| AttributeError | Raised when attribute assignment or reference fails. |
| EOFError | Raised when the input() functions hits end-of-file condition. |
| FloatingPointError | Raised when a floating point operation fails. |
| GeneratorExit | Raise when a generator's close() method is called. |
| ImportError | Raised when the imported module is not found. |
| IndexError | Raised when index of a sequence is out of range. |
| KeyError | Raised when a key is not found in a dictionary. |
| KeyboardInterrupt | Raised when the user hits interrupt key (Ctrl+c or delete). |
| MemoryError | Raised when an operation runs out of memory. |
| NameError | Raised when a variable is not found in local or global scope. |
| NotImplementedError | Raised by abstract methods. |
| OSError | Raised when system operation causes system related error. |
| OverflowError | Raised when result of an arithmetic operation is too large to be represented. |
| ReferenceError | Raised when a weak reference proxy is used to access a garbage collected referent. |
| RuntimeError | Raised when an error does not fall under any other category. |
| StopIteration | Raised by next() function to indicate that there is no further item to be returned by iterator. |
| SyntaxError | Raised by parser when syntax error is encountered. |
| IndentationError | Raised when there is incorrect indentation. |
| TabError | Raised when indentation consists of inconsistent tabs and spaces. |
| SystemError | Raised when interpreter detects internal error. |
| SystemExit | Raised by sys.exit() function. |
| TypeError | Raised when a function or operation is applied to an object of incorrect type. |
| UnboundLocalError | Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. |
| UnicodeError | Raised when a Unicode-related encoding or decoding error occurs. |
| UnicodeEncodeError | Raised when a Unicode-related error occurs during encoding. |
| UnicodeDecodeError | Raised when a Unicode-related error occurs during decoding. |
| UnicodeTranslateError | Raised when a Unicode-related error occurs during translating. |
| ValueError | Raised when a function gets argument of correct type but improper value. |
| ZeroDivisionError | Raised when second operand of division or modulo operation is zero. |

# Python Exception Handling - Try, Except and Finally

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.

When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.

If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

**Catching Exceptions in Python**

In Python, exceptions can be handled using a try statement.

A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.

It is up to us, what operations we perform once we have caught the exception. Here is a simple example.

# import module sys to get the type of exception

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except:

print("Oops!",sys.exc\_info()[0],"occured.")

print("Next entry.")

print()

print("The reciprocal of",entry,"is",r)

**Output**

The entry is a

Oops! <class 'ValueError'> occured.

Next entry.

The entry is 0

Oops! <class 'ZeroDivisionError' > occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5

In this program, we loop until the user enters an integer that has a valid reciprocal. The portion that can cause exception is placed inside try block.

If no exception occurs, except block is skipped and normal flow continues. But if any exception occurs, it is caught by the except block.

Here, we print the name of the exception using ex\_info() function inside sys module and ask the user to try again. We can see that the values 'a' and '1.3' causes ValueError and '0' causes ZeroDivisionError.

**Catching Specific Exceptions in Python**

In the above example, we did not mention any exception in the except clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch.

A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

try:

# do something

pass

except ValueError:

# handle ValueError exception

pass

except (TypeError, ZeroDivisionError):

# handle multiple exceptions

# TypeError and ZeroDivisionError

pass

except:

# handle all other exceptions

pass

**Raising Exceptions**

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword raise.

We can also optionally pass in value to the exception to clarify why that exception was raised.

>>> raise KeyboardInterrupt

Traceback (most recent call last):

...

KeyboardInterrupt

>>> raise MemoryError("This is an argument")

Traceback (most recent call last):

...

MemoryError: This is an argument

>>> try:

... a = int(input("Enter a positive integer: "))

... if a <= 0:

... raise ValueError("That is not a positive number!")

... except ValueError as ve:

... print(ve)

...

Enter a positive integer: -2

That is not a positive number!

**try...finally**

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.

For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution.

Here is an example of file operations to illustrate this.

try:

f = open("test.txt",encoding = 'utf-8')

# perform file operations

finally:

f.close()

This type of construct makes sure the file is closed even if an exception occurs.

# Python Custom Exceptions

Python has many [built-in exceptions](https://www.programiz.com/python-programming/exceptions) which forces your program to output an error when something in it goes wrong.

However, sometimes you may need to create custom exceptions that serves your purpose.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived form 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 is derived from the Exception class. This new exception can be raised, like other exceptions, using the raisestatement 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.

## Example: User-Defined Exception in Python

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.

This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, hint is provided whether their guess is greater than or less than the stored number.

# define Python user-defined exceptions

class Error(Exception):

"""Base class for other exceptions"""

pass

class ValueTooSmallError(Error):

"""Raised when the input value is too small"""

pass

class ValueTooLargeError(Error):

"""Raised when the input value is too large"""

pass

# our main program

# user guesses a number until he/she gets it right

# you need to guess this number

number = 10

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.")

Here is a sample run of this program.

Enter a number: 12

This value is too large, try again!

Enter a number: 0

This value is too small, try again!

Enter a number: 8

This value is too small, try again!

Enter a number: 10

Congratulations! You guessed it correctly.

Here, we have defined a base class called Error.

The other two exceptions (ValueTooSmallError and ValueTooLargeError) that are actually raised by our program are derived from this class. This is the standard way to define user-defined exceptions in Python programming, but you are not limited to this way only.

## Source Code to Merge Mails

# Python program to mail merger

# Names are in the file names.txt

# Body of the mail is in body.txt

# open names.txt for reading

with open("names.txt",'r',encoding = 'utf-8') as names\_file:

# open body.txt for reading

with open("body.txt",'r',encoding = 'utf-8') as body\_file:

# read entire content of the body

body = body\_file.read()

# iterate over names

for name in names\_file:

mail = "Hello "+name+body

# write the mails to individual files

with open(name.strip()+".txt",'w',encoding = 'utf-8') as mail\_file:

mail\_file.write(mail)

# Python rogram to find the SHA-1 message digest of a file

# import hashlib module

import hashlib

def hash\_file(filename):

""""This function returns the SHA-1 hash

of the file passed into it"""

# make a hash object

h = hashlib.sha1()

# open file for reading in binary mode

with open(filename,'rb') as file:

# loop till the end of the file

chunk = 0

while chunk != b'':

# read only 1024 bytes at a time

chunk = file.read(1024)

h.update(chunk)

# return the hex representation of digest

return h.hexdigest()

message = hash\_file("track1.mp3")

print(message)

## Source Code of Find Resolution of JPEG Image

def jpeg\_res(filename):

""""This function prints the resolution of the jpeg image file passed into it"""

# open image for reading in binary mode

with open(filename,'rb') as img\_file:

# height of image (in 2 bytes) is at 164th position

img\_file.seek(163)

# read the 2 bytes

a = img\_file.read(2)

# calculate height

height = (a[0] << 8) + a[1]

# next 2 bytes is width

a = img\_file.read(2)

# calculate width

width = (a[0] << 8) + a[1]

print("The resolution of the image is",width,"x",height)

jpeg\_res("img1.jpg")

In this program, we opened the image in binary mode. Non-text files must be open in this mode. The height of the image is at 164th position followed by width of the image. Both are 2 bytes long.

Note that this is true only for JPEG File Interchange Format (JFIF) standard. If your image is encode using other standard (like EXIF), the code will not work.

We convert the 2 bytes into a number using bitwise shifting operator <<. Finally, the resolution is displayed.