Python IO: -

**Files**

Files are named locations on disk to store related information. They are 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 the computer is turned off), we use files for future use of the data by permanently storing them.

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 the 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

**Opening Files in Python**

Python has a built-in open() function 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:/Python38/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 can 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 images or executable files.

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

**Closing Files in Python**

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

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

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](https://www.programiz.com/python-programming/exception-handling) block.

try:

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

# perform file operations

finally:

f.close()

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

The best way to close a file is by using the with statement. This ensures that the file is closed when the block inside the with statement 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

**Writing to Files in 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. Due to this, all the previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using the 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 in the current directory if it does not exist. If it does exist, it is overwritten.

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

**Reading Files in Python**

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

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

We can read the text.txt file we wrote in the above section in the following way:

>>> 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 a newline as '\n'. Once the end of the file is reached, we get an 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

In this program, the lines in the file itself include a newline character \n. So, we use the end parameter of the print() function to avoid two newlines when printing.

Alternatively, we can use the 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 methods return empty values when the 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 the above examples.

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

|  |  |
| --- | --- |
| Method | Description |
| close() | Closes an opened file. It has no effect if the file is already closed. |
| detach() | Separates the underlying binary buffer from the TextIOBase and returns it. |
| fileno() | Returns an integer number (file descriptor) of the file. |
| flush() | Flushes the write buffer of the file stream. |
| isatty() | Returns True if the file stream is interactive. |
| read(n) | Reads at most n characters from 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) | Reads and returns one line from the file. Reads in at most n bytes if specified. |
| readlines(n=-1) | Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified. |
| seek(offset,from=SEEK\_SET) | Changes 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) | Resizes the file stream to size bytes. If size is not specified, resizes to current location. |
| writable() | Returns True if the file stream can be written to. |
| write(s) | Writes the string s to the file and returns the number of characters written. |
| writelines(lines) | Writes a list of lines to the file. |

TC1

TC2

TC3

TC4

Smoke Suite -Trigger-Alure, Any other report-Run another python file-pick report and send to configured email

Sending Email using Python:-

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers.

Python provides **smtplib** module, which defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon.

Here is a simple syntax to create one SMTP object, which can later be used to send an e-mail −

import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local\_hostname]]] )

Here is the detail of the parameters −

* **host** − This is the host running your SMTP server. You can specify IP address of the host or a domain name like tutorialspoint.com. This is optional argument.
* **port** − If you are providing *host* argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.
* **local\_hostname** − If your SMTP server is running on your local machine, then you can specify just *localhost* as of this option.

An SMTP object has an instance method called **sendmail**, which is typically used to do the work of mailing a message. It takes three parameters −

* The *sender* − A string with the address of the sender.
* The *receivers* − A list of strings, one for each recipient.
* The *message* − A message as a string formatted as specified in the various RFCs.

## Example

Here is a simple way to send one e-mail using Python script. Try it once −

#!/usr/bin/python

import smtplib

sender = 'from@fromdomain.com'

receivers = ['to@todomain.com']

message = """From: From Person <from@fromdomain.com>

To: To Person <to@todomain.com>

Subject: SMTP e-mail test

This is a test e-mail message.

"""

try:

smtpObj = smtplib.SMTP('localhost')

smtpObj.sendmail(sender, receivers, message)

print "Successfully sent email"

except SMTPException:

print "Error: unable to send email"

Here, you have placed a basic e-mail in message, using a triple quote, taking care to format the headers correctly. An e-mail requires a **From**, **To**, and **Subject** header, separated from the body of the e-mail with a blank line.

To send the mail you use *smtpObj* to connect to the SMTP server on the local machine and then use the *sendmail* method along with the message, the from address, and the destination address as parameters (even though the from and to addresses are within the e-mail itself, these aren't always used to route mail).

If you are not running an SMTP server on your local machine, you can use *smtplib* client to communicate with a remote SMTP server. Unless you are using a webmail service (such as Hotmail or Yahoo! Mail), your e-mail provider must have provided you with outgoing mail server details that you can supply them, as follows −

smtplib.SMTP('mail.your-domain.com', 25)

## Sending an HTML e-mail using Python

When you send a text message using Python, then all the content are treated as simple text. Even if you include HTML tags in a text message, it is displayed as simple text and HTML tags will not be formatted according to HTML syntax. But Python provides option to send an HTML message as actual HTML message.

While sending an e-mail message, you can specify a Mime version, content type and character set to send an HTML e-mail.

### **Example**

Following is the example to send HTML content as an e-mail. Try it once −

#!/usr/bin/python

import smtplib

message = """From: From Person <from@fromdomain.com>

To: To Person <to@todomain.com>

MIME-Version: 1.0

Content-type: text/html

Subject: SMTP HTML e-mail test

This is an e-mail message to be sent in HTML format

<b>This is HTML message.</b>

<h1>This is headline.</h1>

"""

try:

smtpObj = smtplib.SMTP('localhost')

smtpObj.sendmail(sender, receivers, message)

print "Successfully sent email"

except SMTPException:

print "Error: unable to send email"

## Sending Attachments as an E-mail

To send an e-mail with mixed content requires to set **Content-type** header to **multipart/mixed**. Then, text and attachment sections can be specified within **boundaries**.

A boundary is started with two hyphens followed by a unique number, which cannot appear in the message part of the e-mail. A final boundary denoting the e-mail's final section must also end with two hyphens.

Attached files should be encoded with the **pack("m")** function to have base64 encoding before transmission.

### **Example**

Following is the example, which sends a file **/tmp/test.txt** as an attachment. Try it once −

#!/usr/bin/python

import smtplib

import base64

filename = "/tmp/test.txt"

# Read a file and encode it into base64 format

fo = open(filename, "rb")

filecontent = fo.read()

encodedcontent = base64.b64encode(filecontent) # base64

sender = 'webmaster@tutorialpoint.com'

reciever = 'amrood.admin@gmail.com'

marker = "AUNIQUEMARKER"

body ="""

This is a test email to send an attachement.

"""

# Define the main headers.

part1 = """From: From Person <me@fromdomain.net>

To: To Person <amrood.admin@gmail.com>

Subject: Sending Attachement

MIME-Version: 1.0

Content-Type: multipart/mixed; boundary=%s

--%s

""" % (marker, marker)

# Define the message action

part2 = """Content-Type: text/plain

Content-Transfer-Encoding:8bit

%s

--%s

""" % (body,marker)

# Define the attachment section

part3 = """Content-Type: multipart/mixed; name=\"%s\"

Content-Transfer-Encoding:base64

Content-Disposition: attachment; filename=%s

%s

--%s--

""" %(filename, filename, encodedcontent, marker)

message = part1 + part2 + part3

try:

smtpObj = smtplib.SMTP('localhost')

smtpObj.sendmail(sender, reciever, message)

print "Successfully sent email"

except Exception:

print "Error: unable to send email"

# **Python - Multithreaded Programming**

Running several threads is similar to running several different programs concurrently, but with the following benefits −

* Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
* Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

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

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

## Starting a New Thread

To spawn another thread, you need to call following method available in *thread* module −

thread.start\_new\_thread ( function, args[, kwargs] )

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of *args*. When function returns, the thread terminates.

Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs*is an optional dictionary of keyword arguments.

### **Example**

#!/usr/bin/python

import time

import \_thread as thread # In Python 3, \_thread is renamed to \_thread

# Define a function for the thread

def print\_time(threadName, delay):

count = 0

while count < 5:

time.sleep(delay)

count += 1

print("%s: %s" % (threadName, time.ctime(time.time())))

# Create two threads as follows

try:

thread.start\_new\_thread(print\_time, ("Thread-1", 2, ))

thread.start\_new\_thread(print\_time, ("Thread-2", 4, ))

except:

print("Error: unable to start thread")

while True:

pass

When the above code is executed, it produces the following result −

Thread-1: Thu Jan 22 15:42:17 2009

Thread-1: Thu Jan 22 15:42:19 2009

Thread-2: Thu Jan 22 15:42:19 2009

Thread-1: Thu Jan 22 15:42:21 2009

Thread-2: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:25 2009

Thread-2: Thu Jan 22 15:42:27 2009

Thread-2: Thu Jan 22 15:42:31 2009

Thread-2: Thu Jan 22 15:42:35 2009

Although it is very effective for low-level threading, but the *thread* module is very limited compared to the newer threading module.

## The *Threading* Module

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

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

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

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

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

## Creating Thread Using *Threading* Module

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

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

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

### **Example**

#!/usr/bin/python

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print("Starting " + self.name)

print\_time(self.name, 5, self.counter)

print("Exiting " + self.name)

def print\_time(threadName, counter, delay):

while counter:

if exitFlag:

threadName.exit()

time.sleep(delay)

print("%s: %s" % (threadName, time.ctime(time.time())))

counter -= 1

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

print("Exiting Main Thread")

When the above code is executed, it produces the following result −

Starting Thread-1

Starting Thread-2

Exiting Main Thread

Thread-1: Thu Mar 21 09:10:03 2013

Thread-1: Thu Mar 21 09:10:04 2013

Thread-2: Thu Mar 21 09:10:04 2013

Thread-1: Thu Mar 21 09:10:05 2013

Thread-1: Thu Mar 21 09:10:06 2013

Thread-2: Thu Mar 21 09:10:06 2013

Thread-1: Thu Mar 21 09:10:07 2013

Exiting Thread-1

Thread-2: Thu Mar 21 09:10:08 2013

Thread-2: Thu Mar 21 09:10:10 2013

Thread-2: Thu Mar 21 09:10:12 2013

Exiting Thread-2

## Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object is used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.

If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The *release()* method of the new lock object is used to release the lock when it is no longer required.

### **Example**

#!/usr/bin/python

import threading

import time

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print("Starting " + self.name)

# Get lock to synchronize threads

threadLock.acquire()

print\_time(self.name, self.counter, 3)

# Free lock to release next thread

threadLock.release()

def print\_time(threadName, delay, counter):

while counter:

time.sleep(delay)

print("%s: %s" % (threadName, time.ctime(time.time())))

counter -= 1

threadLock = threading.Lock()

threads = []

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

# Add threads to thread list

threads.append(thread1)

threads.append(thread2)

# Wait for all threads to complete

for t in threads:

t.join()

print("Exiting Main Thread")

When the above code is executed, it produces the following result −

Starting Thread-1

Starting Thread-2

Thread-1: Thu Mar 21 09:11:28 2013

Thread-1: Thu Mar 21 09:11:29 2013

Thread-1: Thu Mar 21 09:11:30 2013

Thread-2: Thu Mar 21 09:11:32 2013

Thread-2: Thu Mar 21 09:11:34 2013

Thread-2: Thu Mar 21 09:11:36 2013

Exiting Main Thread

## Multithreaded Priority Queue

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue −

* **get()** − The get() removes and returns an item from the queue.
* **put()** − The put adds item to a queue.
* **qsize()** − The qsize() returns the number of items that are currently in the queue.
* **empty()** − The empty( ) returns True if queue is empty; otherwise, False.
* **full()** − the full() returns True if queue is full; otherwise, False.

### **Example**

#!/usr/bin/python

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print("Starting " + self.name)

print\_time(self.name, 5, self.counter)

print("Exiting " + self.name)

def print\_time(threadName, counter, delay):

while counter:

if exitFlag:

threadName.exit()

time.sleep(delay)

print("%s: %s" % (threadName, time.ctime(time.time())))

counter -= 1

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

print("Exiting Main Thread")

When the above code is executed, it produces the following result −

Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1 processing One

Thread-2 processing Two

Thread-3 processing Three

Thread-1 processing Four

Thread-2 processing Five

Exiting Thread-3

Exiting Thread-1

Exiting Thread-2

Exiting Main Thread

## What Is Itertools and Why Should You Use It?

According to the [itertools docs](https://docs.python.org/3/library/itertools.html), it is a “module [that] implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML… Together, they form an ‘iterator algebra’ making it possible to construct specialized tools succinctly and efficiently in pure Python.”

Loosely speaking, this means that the functions in itertools “operate” on iterators to produce more complex iterators. Consider, for example, the [built-in zip() function](https://docs.python.org/3/library/functions.html#zip), which takes any number of iterables as arguments and returns an iterator over tuples of their corresponding elements:

>>>

>>> list(zip([1, 2, 3], ['a', 'b', 'c']))

[(1, 'a'), (2, 'b'), (3, 'c')]

The arguments that are given after the name of the program in the command line shell of the operating system are known as [Command Line Arguments](https://www.geeksforgeeks.org/command-line-arguments-in-python/). Python provides various ways of dealing with these types of arguments. One of them is sys module.

## sys Module

A [module](https://www.geeksforgeeks.org/python-modules/) is a file containing Python definitions and statements. The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

#### sys.argv

sys.argv is used in python to retrieve command line arguments at runtime. For a program to be able to use it, it has to be imported from the “sys” module. The arguments so obtained are in the form of an array named sys.argv.

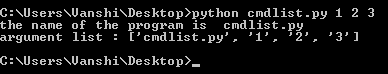
**Note:** sys.argv[0] gives the name of the program and the following indices work like members of an array.

**Approach:**  
The name of the program is “cmdlist.py”.

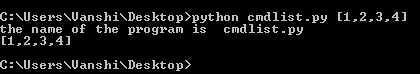
Python CMD Line Argument: -

1. **Working with command line:** Consider the below code written in cmdlis.py

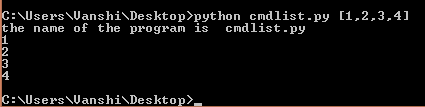
|  |
| --- |
| import sys      print("the name of the program is ", sys.argv[0])  print("argument list :", sys.argv) |

1. **Output:**
2. 
3. **Calling a list using the command line**

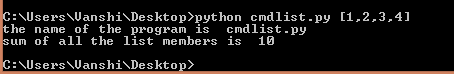
|  |
| --- |
| import sys      print ("the name of the program is ", sys.argv[0])    a = sys.argv[1]  print (a) |

1. **Output:**
2. 
3. **Working with the list called by command line.**  
   sys.argv is a string array. Thus, any argument passed to it is a string so in order to properly put in use we have to convert it to an appropriate list form.
   * **Program to take list as command line argument and convert it to a proper list form**

|  |
| --- |
| import sys      print("the name of the program is ", sys.argv[0])    n = len(sys.argv[1])  a = sys.argv[1][1:n-1]  a = a.split(', ')    for i in a:      print(i) |

* + **Output:**  
    
  + **Program to find the sum of all the members of the list**  
    Also, keep in find to change the members to the required data type if not string.

|  |
| --- |
| import sys      print ("the name of the program is", sys.argv[0])    n=len(sys.argv[1])  a=sys.argv[1][1:n-1]  a=a.split(',')    A = [int(i) for i in a]    b = 0    for i in A:      b += i  print("sum of all the list members is ", b) |

* + **Output:**
  + 

1. **Working with multiple list retrieved through command line.**

|  |
| --- |
| import sys      print ("the name of the program is ", sys.argv[0])    n = len(sys.argv[1])  a = sys.argv[1][1:n-1]  a = a.split(', ')    A = [int(i) for i in a]  b = 0    for i in A:      b += i  print("sum of all the list members is ", b)    n = len(sys.argv[2])  c = sys.argv[2][1:n-1]  c = c.split(', ')    d = ""    for i in c:      d = d+i    print ("sum of all the list members is ", d) |

1. **Output:**  
   

# **Assertions in Python**

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

## The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError exception*.

The syntax for assert is −

assert Expression[, Arguments]

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

## Example

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature −

#!/usr/bin/python

def KelvinToFahrenheit(Temperature):

assert (Temperature >= 0),"Colder than absolute zero!"

return ((Temperature-273)\*1.8)+32

print KelvinToFahrenheit(273)

print int(KelvinToFahrenheit(505.78))

print KelvinToFahrenheit(-5)

When the above code is executed, it produces the following result −

32.0

451

Traceback (most recent call last):

File "test.py", line 9, in <module>

print KelvinToFahrenheit(-5)

File "test.py", line 4, in KelvinToFahrenheit

assert (Temperature >= 0),"Colder than absolute zero!"

AssertionError: Colder than absolute zero!

**Python Lambda Functions**

In Python, an anonymous function means that a function is without a name. As we already know that the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in Python. It has the following syntax:

**Syntax:** lambda arguments: expression

* This function can have any number of arguments but only one expression, which is evaluated and returned.
* One is free to use lambda functions wherever function objects are required.
* You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
* It has various uses in particular fields of programming besides other types of expressions in functions.

Let’s look at this example and try to understand the **difference between a normal def defined function and lambda function**. This is a program that returns the cube of a given value:

* Python

|  |
| --- |
| # Python code to illustrate cube of a number  # showing difference between def() and lambda().  def cube(y):      return y\*y\*y    lambda\_cube = lambda y: y\*y\*y    # using the normally  # defined function  print(cube(5))    # using the lambda function  print(lambda\_cube(5)) |

**Output:**

125

125

As we can see in the above example both the cube() function and lambda\_cube() function behave the same and as intended. Let’s analyze the above example a bit more:

* **Without using Lambda:** Here, both of them return the cube of a given number. But, while using def, we needed to define a function with a name cube and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the *return* keyword.
* **Using Lambda:** Lambda definition does not include a “return” statement, it always contains an expression that is returned. We can also put a lambda definition anywhere a function is expected, and we don’t have to assign it to a variable at all. This is the simplicity of lambda functions.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

**Using lambda() Function with filter()**

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True. Here is a small program that returns the odd numbers from an input list: 

**Example 1:**

* Python

|  |
| --- |
| # Python code to illustrate  # filter() with lambda()  li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]    finalist = list(filter(lambda x: (x%2 != 0) , li))  print(finalist) |

**Output:**

[5, 7, 97, 77, 23, 73, 61]

**Example 2:**

* Python3

|  |
| --- |
| # Python 3 code to people above 18 yrs  ages = [13, 90, 17, 59, 21, 60, 5]    adults = list(filter(lambda age: age>18, ages))    print(adults) |

**Output:**

[90, 59, 21, 60]

**Using lambda() Function with map()**

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

**Example 1:**

* Python

|  |
| --- |
| # Python code to illustrate  # map() with lambda()  # to get double of a list.  li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]    final\_list = list(map(lambda x: x\*2, li))  print(final\_list) |

**Output:**

[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]

**Example 2:**

* Python3

|  |
| --- |
| # Python program to demonstrate  # use of lambda() function  # with map() function  animals = ['dog', 'cat', 'parrot', 'rabbit']    # here we intend to change all animal names  # to upper case and return the same  uppered\_animals = list(map(lambda animal: str.upper(animal), animals))    print(uppered\_animals) |

**Output:**

['DOG', 'CAT', 'PARROT', 'RABBIT']

**Using lambda() Function with reduce()**

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

**Example 1:**

* Python

|  |
| --- |
| # Python code to illustrate  # reduce() with lambda()  # to get sum of a list    from functools import reduce  li = [5, 8, 10, 20, 50, 100]  sum = reduce((lambda x, y: x + y), li)  print (sum) |

**Output:**

193

Here the results of previous two elements are added to the next element and this goes on till the end of the list like (((((5+8)+10)+20)+50)+100).

**Example 2:**

* Python3

|  |
| --- |
| # python code to demonstrate working of reduce()  # with a lambda function    # importing functools for reduce()  import functools    # initializing list  lis = [ 1 , 3, 5, 6, 2, ]    # using reduce to compute maximum element from list  print ("The maximum element of the list is : ",end="")  print (functools.reduce(lambda a,b : a if a > b else b,lis)) |

**Output:**

The maximum element of the list is : 6

# Read and Write to an excel file using Python openpyxl module

Python provides openpyxl module for operating with Excel files.

How to create Excel files, how to write, read etc. can be implemented by this module.

For installing openpyxl module, we can write this command in command prompt. Alt + F12

pip install openpyxl

## If we want to give a sheet title name

## Example code

import openpyxl

my\_wb = openpyxl.Workbook()

my\_sheet = my\_wb.active

my\_sheet\_title = my\_sheet.title

print("My sheet title: " + my\_sheet\_title)

## Output

My sheet title:Sheet

## To change Title Name

## Example code

import openpyxl

my\_wb = openpyxl.Workbook()

my\_sheet = my\_wb.active

my\_sheet.title = "My New Sheet"

print("sheet name is : " + my\_sheet.title)

## Output

sheet name is : My New Sheet

## Insert data or to write to an Excel sheet

## Example code

import openpyxl

my\_wb = openpyxl.Workbook()

my\_sheet = my\_wb.active

c1 = my\_sheet.cell(row=1, column=1)

c1.value = "Aadrika"

c2 = my\_sheet.cell(row=1, column=2)

c2.value = "Adwaita"

c3 = my\_sheet['A2']

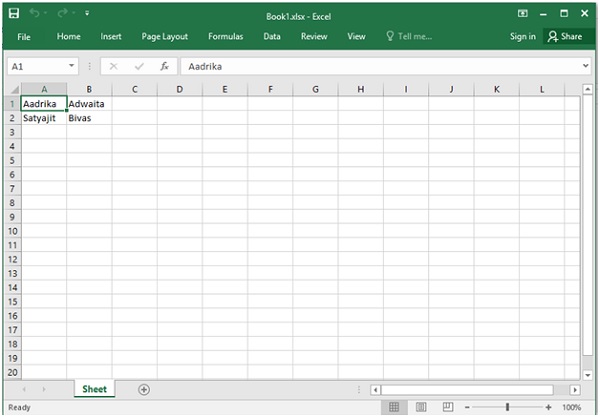
c3.value = "Satyajit"

c4 = my\_sheet['B2']

c4.value = "Bivas"

my\_wb.save(r"C:\Users\Pragyandeep\Desktop\Book1.xlsx")

## Output



## To add Sheets in the Workbook

## Example code

import openpyxl

my\_wb = openpyxl.Workbook()

my\_sheet = my\_wb.active

my\_wb.create\_sheet(index = 1 , title = "new sheet")

my\_wb.save("C:\Users\TP\Desktop\Book1.xlsx")

## Output



## Display Total number of rows.

## Example code

import openpyxl

my\_path = r"C:\Users\TP\Desktop\Book1.xlsx"

my\_wb\_obj = openpyxl.load\_workbook(my\_path)

my\_sheet\_obj = my\_wb\_obj.active

print(my\_sheet\_obj.max\_row)

## Output

2

## Display a particular cell value

## Example code

import openpyxl

# Give the location of the file

My\_path = "C:\Users\TP\Desktop\Book1.xlsx"

wb\_obj = openpyxl.load\_workbook(my\_path)

my\_sheet\_obj = my\_wb\_obj.active

my\_cell\_obj = my\_sheet\_obj.cell(row = 1, column = 1)

print(my\_cell\_obj.value)

## Output

Aadrika

## Display total number of columns

## Example code

import openpyxl

# Give the location of the file

my\_path = "C:\Users\TP\Desktop\Book1.xlsx"

my\_wb\_obj = openpyxl.load\_workbook(path)

my\_sheet\_obj = my\_wb\_obj.active

print(my\_sheet\_obj.max\_column)

## Output

2

## Display all columns name

## Example code

import openpyxl

# Give the location of the file

my\_path = "C:\Users\TP\Desktop\Book1.xlsx"

# workbook object is created

my\_wb\_obj = openpyxl.load\_workbook(my\_path)

my\_sheet\_obj = my\_wb\_obj.active

my\_max\_col = my\_sheet\_obj.max\_column

for i in range(1, my\_max\_col + 1):

my\_cell\_obj = my\_sheet\_obj.cell(row = 1, column = i)

print(my\_cell\_obj.value)

## Output

Aadrika

Adwaita

## Display first column value

## Example code

import openpyxl

# Give the location of the file

my\_path = "C:\Users\TP\Desktop\Book1.xlsx"

my\_wb\_obj = openpyxl.load\_workbook(my\_path)

my\_sheet\_obj = my\_wb\_obj.active

my\_row = my\_sheet\_obj.max\_row

for i in range(1, my\_row + 1):

cell\_obj = my\_sheet\_obj.cell(row = i, column = 1)

print(cell\_obj.value)

## Output

Aadrika

Satyajit

## Print a particular row value

## Example code

import openpyxl

# Give the location of the file

my\_path = "C:\Users\TP\Desktop\Book1.xlsx"

my\_wb\_obj = openpyxl.load\_workbook(my\_path)

my\_sheet\_obj = my\_wb\_obj.active

my\_max\_col = my\_sheet\_obj.max\_column

for i in range(1, my\_max\_col + 1):

cell\_obj = my\_sheet\_obj.cell(row = 2, column = i)

print(cell\_obj.value, end = " ")

## Output

Satyajit Bivas

# **Python - Random Module**

The random module is a built-in module to generate the pseudo-random variables. It can be used perform some action randomly such as to get a random number, selecting a random elements from a list, shuffle elements randomly, etc.

## Generate Random Floats

The random.random() method returns a random float number between 0.0 to 1.0. The function doesn't need any arguments.

Example: random()

>>> import random

>>> random.random()

0.645173684807533

## Generate Random Integers

The random.randint() method returns a random integer between the specified integers.

Example: randint()

>>> import random

>>> random.randint(1, 100)

95

>>> random.randint(1, 100)

49

## Generate Random Numbers within Range

The random.randrange() method returns a randomly selected element from the range created by the start, stop and step arguments. The value of start is 0 by default. Similarly, the value of step is 1 by default.

Example:

>>> random.randrange(1, 10)

2

>>> random.randrange(1, 10, 2)

5

>>> random.randrange(0, 101, 10)

80

## Select Random Elements

The random.choice() method returns a randomly selected element from a non-empty sequence. An empty sequence as argument raises an IndexError.

Example:

>>> import random

>>> random.choice('computer')

't'

>>> random.choice([12,23,45,67,65,43])

45

>>> random.choice((12,23,45,67,65,43))

67

## Shuffle Elements Randomly

The random.shuffle() method randomly reorders the elements in a [list](https://www.tutorialsteacher.com/python/python-list).

Example:

>>> numbers=[12,23,45,67,65,43]

>>> random.shuffle(numbers)

>>> numbers

[23, 12, 43, 65, 67, 45]

>>> random.shuffle(numbers)

>>> numbers

[23, 43, 65, 45, 12, 67]

# **Python - OS Module**

It is possible to automatically perform many operating system tasks. The OS module in Python provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory, etc.

You first need to import the os module to interact with the underlying operating system. So, import it using the import os statement before using its functions.

## Getting Current Working Directory

The getcwd() function confirms returns the current working directory.

Example: Get Current Working Directory

>>> import os

>>> os.getcwd()

'C:\\Python311'

## Creating a Directory

We can create a new directory using the os.mkdir() function, as shown below.

Example: Create a Physical Directory

>>> import os

>>> os.mkdir("C:\MyPythonProject")

A new directory corresponding to the path in the string argument of the function will be created. If you open the C:\ drive, then you will see the MyPythonProject folder has been created.

By default, if you don't specify the whole path in the mkdir() function, it will create the specified directory in the current working directory or drive. The following will create MyPythonProject in the C:\Python37 directory.

Example: Create a Physical Directory

>>> import os

>>> os.getcwd()

'C:\Python311'

>>> os.mkdir("MyPythonProject")

## Changing the Current Working Directory

We must first change the current working directory to a newly created one before doing any operations in it. This is done using the chdir() function. The following change current working directory to C:\MyPythonProject.

Example: Change Working Directory

>>> import os

>>> os.chdir("C:\MyPythonProject") # changing current workign directory

>>> os.getcwd()

'C:\MyPythonProject'

You can change the current working directory to a drive. The following makes the C:\ drive as the current working directory.

Example: Change Directory to Drive

>>> os.chdir("C:\\")

>>> os.getcwd()

'C:\\'

In order to set the current directory to the parent directory use ".." as the argument in the chdir() function.

Example: Change CWD to Parent

>>> os.chdir("C:\\MyPythonProject")

>>> os.getcwd()

'C:\\MyPythonProject'

>>> os.chdir("..")

>>> os.getcwd()

'C:\\'

ADVERTISEMENT

## Removing a Directory

The rmdir() function in the OS module removes the specified directory either with an absolute or relative path. Note that, for a directory to be removed, it should be empty.

Example: Remove Directory

>>> import os

>>> os.rmdir("C:\\MyPythonProject")

However, you can not remove the current working directory. To remove it, you must change the current working directory, as shown below.

Example: Remove Directory

>>> import os

>>> os.getcwd()

'C:\\MyPythonProject'

>>> os.rmdir("C:\\MyPythonProject")

PermissionError: [WinError 32] The process cannot access the file because it is being used by another process: 'd:\\MyPythonProject'

>>> os.chdir("..")

>>> os.rmdir("MyPythonProject")

Above, the MyPythonProject will not be removed because it is the current directory. We changed the current working directory to the parent directory using os.chdir("..") and then remove it using the rmdir() function.

## List Files and Sub-directories

The listdir() function returns the list of all files and directories in the specified directory.

Example: List Directories

>>> import os

>>> os.listdir("c:\python311")

['DLLs', 'Doc', 'fantasy-1.py', 'fantasy.db', 'fantasy.py', 'frame.py',

'gridexample.py', 'include', 'Lib', 'libs', 'LICENSE.txt', 'listbox.py', 'NEWS.txt',

'place.py', 'players.db', 'python.exe', 'python3.dll', 'python36.dll', 'pythonw.exe',

'sclst.py', 'Scripts', 'tcl', 'test.py', 'Tools', 'tooltip.py', 'vcruntime140.dll',

'virat.jpg', 'virat.py']

If we don't specify any directory, then list of files and directories in the current working directory will be returned.

Example: List Directories of CWD

>>> import os

>>>os.listdir()

['.config', '.dotnet', 'python']

# **Python - sys Module**

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. You will learn some of the important features of this module here.

## sys.argv

sys.argv returns a list of command line arguments passed to a Python script. The item at index 0 in this list is always the name of the script. The rest of the arguments are stored at the subsequent indices.

Here is a Python script (test.py) consuming two arguments from the command line.

test.py

import sys

print("You entered: ",sys.argv[1], sys.argv[2], sys.argv[3])

This script is executed from command line as follows:

C:\python36> python test.py Python C# Java  
You entered: Python C# Java

Above, sys.argv[1] contains the first argument 'Python', sys.argv[2] contains the second argument 'Python', and sys.argv[3] contains the third argument 'Java'. sys.argv[0] contains the script file name test.py.

## sys.exit

This causes the script to exit back to either the Python console or the command prompt. This is generally used to safely exit from the program in case of generation of an exception.

## sys.maxsize

Returns the largest integer a variable can take.

Example: sys.maxsize

>>> import sys

>>>sys.maxsize

9223372036854775807

## sys.path

This is an environment variable that is a search path for all Python modules.

Example: sys.path

>>> import sys

>>>sys.path

['', 'C:\\python36\\Lib\\idlelib', 'C:\\python36\\python36.zip',

'C:\\python36\\DLLs', 'C:\\python36\\lib', 'C:\\python36',

'C:\\Users\\acer\\AppData\\Roaming\\Python\\Python36\\site-packages',

'C:\\python36\\lib\\site-packages']

## sys.version

This attribute displays a string containing the version number of the current Python interpreter.

Example: sys.version

>>> import sys

>>>sys.version

'3.11.4 (v3.11.4:f59c0932b4, Mar 28 2024, 17:00:18) [MSC v.1900 64 bit (AMD64)]'

# **Python statistics module**

Python statistics module provides the functions to mathematical statistics of numeric data. There are some popular statistical functions defined in this module.

## mean() function

The mean() function is used to calculate the arithmetic mean of the numbers in the list.

**Example**

1. **import** statistics
2. # list of positive integer numbers
3. datasets = [5, 2, 7, 4, 2, 6, 8]
4. x = statistics.mean(datasets)
5. # Printing the mean
6. **print**("Mean is :", x)

**Output:**

Mean is : 4.857142857142857

## median() function

The median() function is used to return the middle value of the numeric data in the list.

**Example**

Skip Ad

1. **import** statistics
2. datasets = [4, -5, 6, 6, 9, 4, 5, -2]
3. # Printing median of the
4. # random data-set
5. **print**("Median of data-set is : % s "
6. % (statistics.median(datasets)))

**Output:**

Median of data-set is : 4.5

## mode() function – Maximum Occurrence

The mode() function returns the most common data that occurs in the list.

**Example**

1. **import** statistics
2. # declaring a simple data-set consisting of real valued positive integers.
3. dataset =[2, 4, 7, 7, 2, 2, 3, 6, 6, 8]
4. # Printing out the mode of given data-set
5. **print**("Calculated Mode % s" % (statistics.mode(dataset)))

**Output:**

Calculated Mode 2

## stdev() function

The stdev() function is used to calculate the standard deviation on a given sample which is available in the form of the list.

**Example**

1. **import** statistics
2. # creating a simple data - set
3. sample = [7, 8, 9, 10, 11]
4. # Prints standard deviation
5. **print**("Standard Deviation of sample is % s "
6. % (statistics.stdev(sample)))

**Output:**

Standard Deviation of sample is 1.5811388300841898

## median\_low()

The median\_low function is used to return the low median of numeric data in the list.

**Example**

1. **import** statistics
2. # simple list of a set of integers
3. set1 = [4, 6, 2, 5, 7, 7]
4. # Note: low median will always be a member of the data-set.
5. # Print low median of the data-set
6. **print**("Low median of data-set is % s "
7. % (statistics.median\_low(set1)))

**Output:**

Low median of the data-set is 5

## median\_high()

The median\_high function is used to return the high median of numeric data in the list.

**Example**

1. **import** statistics
2. # list of set of the integers
3. dataset = [2, 1, 7, 6, 1, 9]
4. **print**("High median of data-set is %s "
5. % (statistics.median\_high(dataset)))

**Output:**

High median of the data-set is 6