Standard Library and Testing

Introduction to Standard Library

A library is a set of files which contains pre-defined code that we can use in our own programs. For example in our Python programs we are using *print()* and *input()* for printing and reading input. These functions are a part of Python's default library.

Python's standard library is a collection of several modules which will be installed by default when you install Python. There are other third party libraries which need to be installed separately after installing Python. Some examples of third party libraries are: *NumPy, SciPy, ScikitLearn* etc.

Python standard library contains several modules for performing several operations from sending email messages to creating a basic HTTP server. Some of the tasks that can be done using Python's standard library are:

- Text manipulation
- Creating and working with data structures
- Date and time manipulation
- Mathematical calculations
- File system access
- Data access and storage
- Data compression
- Cryptography
- Process and thread management
- Networking
- Internet access and management
- Email functionality
- Internationalization and Localization
- Testing
- Operating system interfacing

Operating System Interface

The *os* module provides a portable interface to most of the operating systems like Windows, Linux, Mac OS X etc. It provides functions for creating and managing processes, file system access, and other functionality. Following are some of the functions available in *os* module:

Function	**Description**
getuid( )	Returns the user id for the process
geteuid( )	Returns the effective user id for the process
getgid( )	Returns the group id to which the user belongs
getegid( )	Returns the effective group id to which the user belongs
getenv(variable_name)	Returns the value of specified environment variable
getcwd( )	Returns the current working directory
chdir(new_dir)	Moves the current working directory to the specified new directory
stat(filename)	Returns information of a file like file size, permissions, owner, etc.
chmod(filename, new_permissions)	Assigns new permissions to the specified filename

String Pattern Matching

We can search for a specific pattern in a given string using regular expressions. Regular expressions provide a formal syntax for searching a given pattern in a string. Regular expressions in Python follows Perl syntax. Regular expressions are implemented in Python by the *re* module.

The function *start()* takes in two arguments. First argument is the pattern we want to search and the second argument is the text in which we want to search. The *start()* functions returns a *Match* object.

The *Match* object contains *start()* and *end()* functions which returns the starting and ending index of the pattern in the string. Consider the following example:

```
import re
result = re.search('aa', 'My name is aakaash')
print(result.start(), result.end())
```

The above code prints *11 13* as the output. The pattern in this example is *'aa'* and the text is *'My name is aakaash'*.

The function *findall()* takes pattern and text as input and returns all substrings matching pattern in the given text. The function *finditer()* also takes pattern and text as input and returns an iterator that produces *Match* objects. Consider the following example which returns the multiple occurrences of the pattern:

```
import re
pattern = 'aa'
text = 'My name is aakaash'
for match in re.finditer(pattern, text):
	print(match.start(), match.end())
```

Output of the above code is as follows:

```
11 13
14 16
```

Mathematics

It is common in Python programs to perform mathematical operations. Operators provide support for performing basic mathematical operations. Python's standard library provides support for performing advanced mathematical operations like generating random numbers, finding logarithmic values, performing trigonometric calculations etc.

To generate pseudo random numbers, we can use the *random* module. The pseudo random number generator of Python is based on *Mersenne Twister* algorithm. Following are some of the functions available in random module:

Function	**Description**
random( )	Returns a floating-point value in the range 0 to 1.0
uniform(start, end)	Returns a floating-point value in the range start to end
seed(value)	Sets the seed value of random number generator
randint(start, end)	Returns an integer in the range start to end
randrange(start, end, step)	Returns an integer in the range start to end. It is equivalent to selecting a random number from range(start, end, step)

choice(sequence)	Selects a random item from the given sequence
sample(sequence, n)	Generates n samples from the sequence without repeating values

The *math* module contains several functions which are useful for performing advanced mathematical computations. Following are some of the constants and functions available in *math* module:

Element	**Description**
pi	Prints the mathematical value of π.
e	Prints the mathematical value of *e*.
trunc(float)	Returns an integer value without the decimal part
ceil(float)	Returns the integer greater than or equal to the given number
floor(float)	Returns the integer lesser than or equal to the given number
modf(float)	Returns a tuple containing the fractional part and the whole part
fabs(float)	Returns the absolute value of the given floating-point value
fsum(list of float values)	Returns the sum of floating-point values
factorial(int)	Returns the factorial of given integer value
pow(x, y)	Returns a floating-point value which represents the x^y
sqrt(x)	Returns a floating-point value which represents the square root of x
log(x)	Returns a floating-point value which represents natural log. value of x
Log(x, b)	Returns a floating-point value which represents log. value with base b

Internet Access

Internet access is required for almost any kind of real world application. Even a single script can access a remote server and retrieve information or store data to it. Python provides several modules to create web-based applications.

The *urlparse* module allows to manipulate URLs, i.e., splitting them or combining the individual components together to form an URL. It can be useful either in a client program or in a server program. Following are some of the functions available in *urlparse* module:

Function	**Description**
urlparse(string)	Takes an URL as a string and returns a tuple containing the individual components of an URL like: scheme, netloc, path, params, query, and fragment
urlsplit(string)	This function is an alternative to *urlpase( )* function
geturl( )	Returns the parsed URL
urlunparse(tuple)	Takes a tuple and combines the elements in the tuple in to an URL

The *urllib* module provides various functions that allows our script to access network resources that do not need any authentication. It also provides support for encoding and appending arguments to be passed over HTTP to a server. Some of the functions in the *urllib* module are as follows:

Function	**Description**
urlretrieve( )	Takes four arguments: temporary filename an URL, a function to report the download progress, and data to pass if the URL refers to a form.
urlcleanup( )	Removes the temporary files.
urlencode( )	Takes a dictionary containing data in the form of key value pairs and appends them to the URL.
urlopen( )	Takes an URL as a string and returns a handle to the remote resource.

Dates and Times

Python does not provide native types to handle dates and times. Python's standard library provides three modules which allows us to work with dates and times.

The *time* module contains functions to work with clock time and processor runtime. The *datetime* modules provides classes and functions which provide an interface to work with date, time, and combined values. The *calendar* module allows us to create formatted representation of weeks, months, and years.

Following are some of the functions from *time* module:

Function	**Description**
time( )	Returns the number of seconds since the start of the epoch as a floating-point value.
ctime( )	Displays the date and time in a human readable format.
clock( )	Returns the processor's clock time as a float.

Following are some of the functions from *datetime* module:

Function	**Description**
today( )	This function which belongs to *date* class returns date of current day.
replace( )	Used to change the day, month, or a year in a date.
now( )	Returns current day date and time.
strftime(format)	Displays the date and time based on the given format.

Data Compression

To save the storage area in the hard disk we can compress the existing files. Python's standard library provides various modules to work with various popular compression libraries.

Python's *zlib* and *gzip* modules provides an interface to the GNU zip library. The module *bz2* supports the recent bzip2 format. We can use these modules for reading and writing compressed files.

The *zlib* module provides a low-level interface to many of the functions in the *zlib* compression library from the GNU project. For working with *zlib*, the data that need to be compressed or decompressed needs to be in the memory. The *zlib* module contains a function *compress( )* to compress the data and another function *decompress( )* to decompress the data. Following is an example which demonstrates compressing and decompressing data using the *zlib* module:

```
import zlib
import binascii
original_data = ’This is the original text.’
print ’Original :’, len(original_data), original_data
compressed = zlib.compress(original_data)
print ’Compressed :’, len(compressed), binascii.hexlify(compressed)
decompressed = zlib.decompress(compressed)
print ’Decompressed :’, len(decompressed), decompressed
```

The output for the above code is as follows:

```
Original : 26 This is the original text.
Compressed : 32 789c0bc9c82c5600a2928c5485fca2ccf4ccbcc41c8592d48a123d007f2f097e
```

```
Decompressed : 26 This is the original text.
```

Multithreading

Threading allows several parts of a program to run in parallel (concurrently). This allows the program to execute faster and to keep the CPU busy. The *threading* module of Python's standard library allows us to easily manage several threads of execution. This module builds on the low-level features provided by *thread*.

The simplest way to create a thread is to create an object for the *Thread* class with a target function and call the *start()* function to start the execution of the thread. Following program demonstrates creating and executing a thread:

```
import threading

def worker():
        """thread worker function"""
        print ’Worker’
        return

threads = []
for i in range(5):
t = threading.Thread(target=worker)
threads.append(t)
t.start()
```

Output of the above program is as follows:

```
Worker
Worker
Worker
Worker
Worker
```

We can pass information to a thread as arguments. Any type of object can be passed as an argument to the thread. Following example demonstrates passing arguments to the function:

```
import threading

def worker(num):
        """thread worker function"""
        print ’Worker: %s’ % num
        return

threads = []
for i in range(5):
t = threading.Thread(target=worker, args=(i,))
threads.append(t)
t.start()
```

Output for the above program is as follows:

```
Worker: 0
```

```
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

Turtle Graphics

Turtle graphics is a popular way to introduce programming. It was part of the original *Logo* programming language. Python's standard library provides the *turtle* module to access the graphics package. To use the *turtle* module, write the following:

```
import turtle
```

Now, we can use the functions available in the turtle module. For example, we can move the turtle forward by 200 steps by using the function *forward( )* as follows:

```
turtle.forward(200)
```

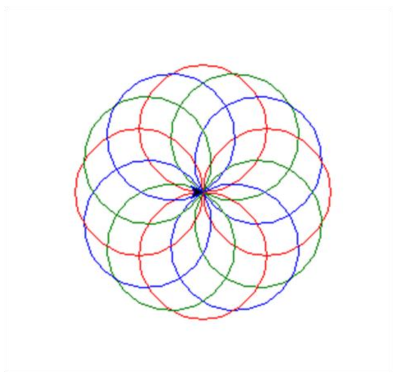
Following are some of the functions available in *turtle* module:

Function	**Description**
forward(distance)	Moves the turtle forward by specified distance
backward(distance)	Moves the turtle backward by specified distance
right(angle)	Turn the turtle right by specified angle
left(angle)	Turn the turtle left by specified angle
setpos(x, y)	Set the position of turtle based on the given x and y co-ordinates
setx(x)	Set the x coordinate of turtle
sety(y)	Set the y coordinate of turtle
home( )	Move the turtle to the origin (0, 0)
circle(radius)	Draw circle with the given radius

Some interesting programs can be written to generate graphics using the turtle graphics module. Following is one such program to draw circle pattern with different colors:

```
import turtle
x = 0
for _ in range(30, 361, 30):
	if(x % 3 == 0): turtle.pencolor("red")
	if(x % 3 == 1): turtle.pencolor("green")
	if(x % 3 == 2): turtle.pencolor("blue")
	turtle.circle(50)
	turtle.right(30)
	x = x + 1
n = input()
```

Output of the above program is as follows:



GUI Programming

Python provides *tkinter* module for GUI programming. The type of programming in which a user can see an interface containing different widgets (controls) and interact with them using mouse is known as GUI programming.

The *tkinter* module is an interface to the popular Tk GUI toolkit which was developed as an extension to the Tcl scripting language. Tkinter is an acronym for *Tk interface*. Tk toolkit provides the following widgets:

- button
- canvas
- checkbutton
- combobox
- entry
- frame
- label
- labelframe
- listbox
- menu
- menubutton
- message
- notebook
- tk_optionMenu
- panedwindow
- progressbar
- radiobutton
- scale
- scrollbar
- separator
- sizegrip
- spinbox
- text
- treeview

It provides the following top-level windows:

- tk_chooseColor - pops up a dialog box for the user to select a color.
- tk_chooseDirectory - pops up a dialog box for the user to select a directory.
- tk_dialog - creates a modal dialog and waits for a response.
- tk_getOpenFile - pops up a dialog box for the user to select a file to open.

- tk_getSaveFile - pops up a dialog box for the user to select a file to save.
- tk_messageBox - pops up a message window and waits for a user response.
- tk_popup - posts a popup menu.
- toplevel - creates and manipulates toplevel widgets.

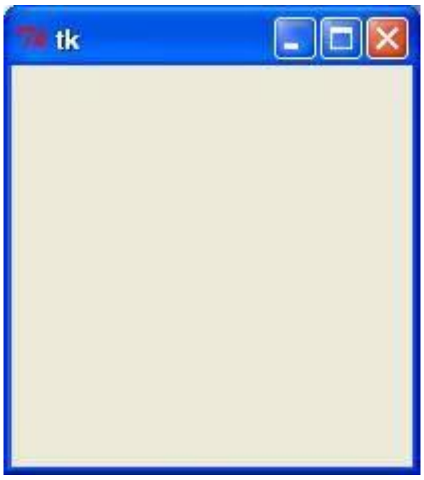
Tk also provides three geometry managers:

- place - which positions widgets at absolute locations
- grid - which arranges widgets in a grid
- pack - which packs widgets into a cavity

Following is a simple program to create a window using *tkinter* module:

```
import tkinter
top = tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```

The output of the above code is a window as follows:



Testing

Introduction

Testing is the practice of writing code that helps to find if there are any errors in the actual logic of the program. It does not prove that our logic is correct. It only reports if the conditions given by the tester are handled correctly or not. Testing is generally used to find out *logical errors*, as *syntax errors* will be reported by the Python runtime itself.

Unit testing is about specifically testing a *unit*. A unit in a Python program or script can be an entire module, a single class, a single class, or almost anything in between. Following are some of the reasons why testing is needed:

- Testing makes sure that our codes work properly under a given set of conditions.
- Testing allows us to make sure that changes to the code did not break existing functionality.
- Testing forces us to think about the code under unusual conditions, possibly revealing errors in the process.

Unit Testing in Python

Python's standard library provides *unittest* module for performing unit testing on our Python code. A unit test consists of one or more assertions. Assertion is statement which is supposed to be always true. The *unittest* module contains many assert functions. These functions are available in *unittest. TestCase* class. One such function is *assertTrue( )* which takes an argument and asserts it to be *True*.

Writing Test Cases

Consider the following program which contains two functions for printing the next prime number:

```
def is_prime(number):
    """Return True if *number* is prime."""
    for element in range(number):
        if number % element == 0:
            return False

    return True

def print_next_prime(number):
    """Print the closest prime number larger than *number*."""
    index = number
    while True:
        index += 1
        if is_prime(index):
            print(index)
```

We have two functions, *is_prime* and *print_next_prime*. If we wanted to test *print_next_prime*, we would need to be sure that *is_prime* is correct, as *print_next_prime* makes use of it. In this case, the function *print_next_prime* is one unit, and *is_prime* is another.

Let's assume that the above Python code is saved in a file named *primes.py*. Now, let's write our test code inside another file named *test_primes.py*. The test case for checking the function *is_prime* is as follows:

```
import unittest
from primes import is_prime

class PrimesTestCase(unittest.TestCase):
    """Tests for `primes.py`."""

    def test_is_five_prime(self):
        """Is five successfully determined to be prime?"""
        self.assertTrue(is_prime(5))

if __name__ == '__main__':
    unittest.main()
```

The file creates a unit test with a single test case: *test_is_five_prime*. Using Python's built-in *unittest* framework, any member function whose name begins with *test* in a class deriving from *unittest.TestCase* will be run, and its assertions checked, when *unittest.main()* is called.

Running Test Cases

We can run the test cases by running the Python code which contains the test cases. In out example we will use the command *python test_primes.py* to run the test cases. We'll see the output of the *unittest* framework printed on the console:

```
$ python test_primes.py
E
======================================================================
ERROR: test_is_five_prime (__main__.PrimesTestCase)
----------------------------------------------------------------------
...
```

The single "E" represents the results of our single test (if it was successful, a "." would have been printed). We can see that our test failed, the line that caused the failure, and any exceptions raised.