

Unit -6

Topics to be covered:

Brief Tour of the Standard Library - Operating System Interface - String Pattern Matching, Mathematics, Internet Access, Dates and Times, Data Compression, Multithreading, GUI Programming, Turtle Graphics

Testing: Why testing is required ?, Basic concepts of testing, Unit testing in Python, Writing Test cases, Running Tests.

Introduction to tkinter package

- Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.
- **tkinter:** Tkinter is the Python interface to the Tk GUI toolkit shipped with Python.
- Tkinter is distributed along with Python software.
- It is a platform independent package.
- It has variety of GUI elements such as Label, Button, Menu, Frame..etc.
- These GUI elements are called “Widgets”

tkinter programming

- Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.
- Creating a GUI application using Tkinter is an easy task.
 - import the *Tkinter* module.
 - Create the GUI application main window.
 - Add one or more of the above-mentioned widgets to the GUI application.
 - Enter the main event loop to take action against each event triggered by the user.

How to arrange these widgets in window

- All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.
- [The pack\(\) Method](#) - This geometry manager organizes widgets in blocks before placing them in the parent widget.

Syntax

- widget.pack(pack_options)
- Options are {expand, fill, side}
- expand –used to expand the widget space, not used by parent
- fill
 - -NONE –default
 - X – fills Horizontally ,
 - Y – fills Veritcally
- side
 - TOP-default
 - BOTTOM-
 - LEFT
 - RIGHT
- [The grid\(\) Method](#) - This geometry manager organizes widgets in a table-like structure in the parent widget.
 - Syntax
 - widget.grid(grid_options)
 - **row**: The row to put widget in; default the first row that is still empty.
 - **column** : The column to put widget in; default 0 (leftmost column)
 - [The place\(\) Method](#) -This geometry manager organizes widgets by placing them in a specific position in the parent widget.
- This geometry manager organizes widgets by placing them in a specific position in the parent widget.
- Syntax - widget.place(place_options)
- **height, width** : Height and width in pixels.
- Example – l.place(height=100,width=200) # where l is widget

Label widget

- This is used to create static element which does not do anything except displaying the content
- Syntax:
 - L=Label(parent,text="Hello",)

Example Program:

```
from tkinter import*
root=Tk()
root.title("KSR window") #giving the title to the window
l1=Label(root,text="Hello ....")
l1.config(bg='green',fg='red',font=('calbri',30,'italic'))
l1.pack(side=LEFT) #setting the widget on left side of the root window
root.mainloop()
```

Output:**Canvas widget**

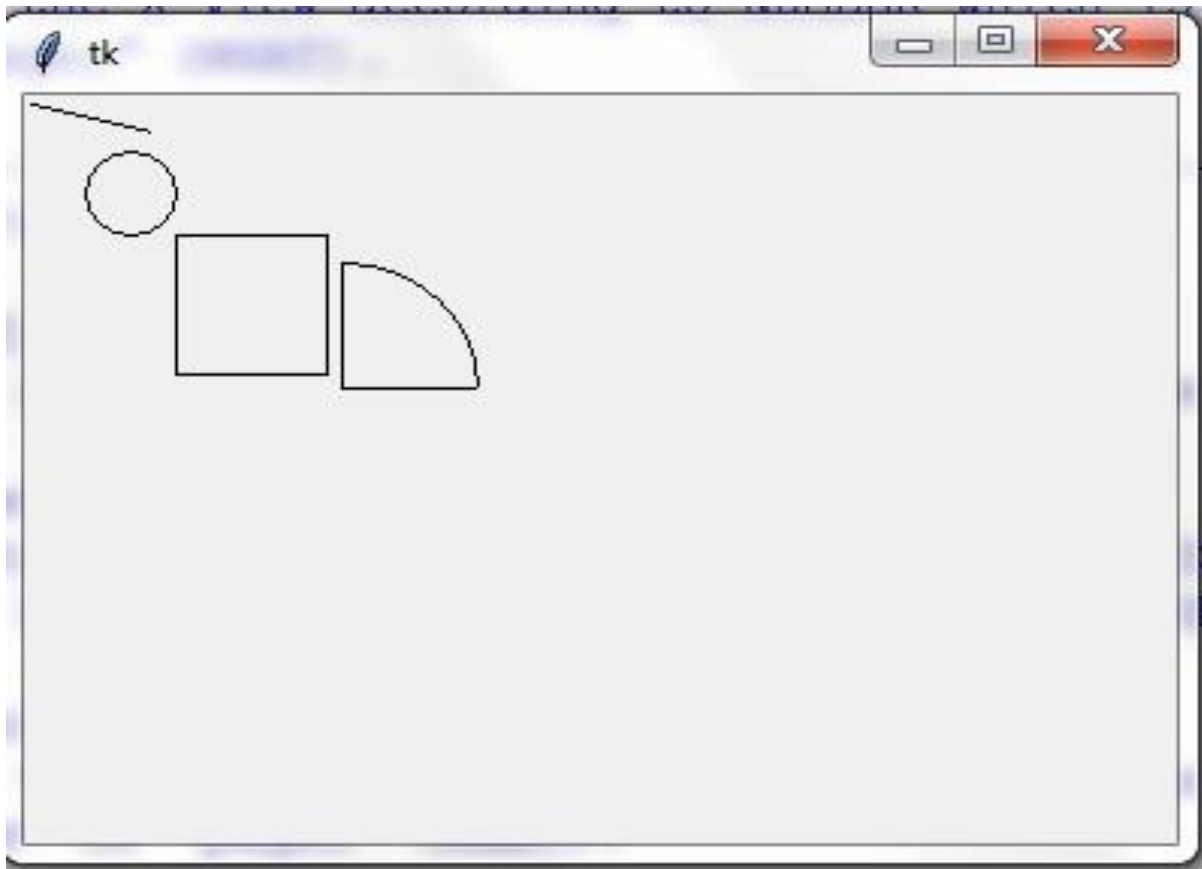
This used to draw shapes like lines, ovals, rectangles, and arcs.

Example program:

#drwing the shapes

```
from tkinter import*
root=Tk()
c=Canvas(root)
c.create_line(2,3,42,13) #drawing the line
c.create_oval(20,20,50,50) #drawing the oval or circle
c.create_arc(150,150,60,60) #drawing arc
c.create_rectangle(100,100,50,50) #rectangle
c.pack()
```

Output:



Button widget

- The Button widget is used to display buttons in your application.
- Syntax:

```
B=Button(parent, text=' name',command=fun_name)
```

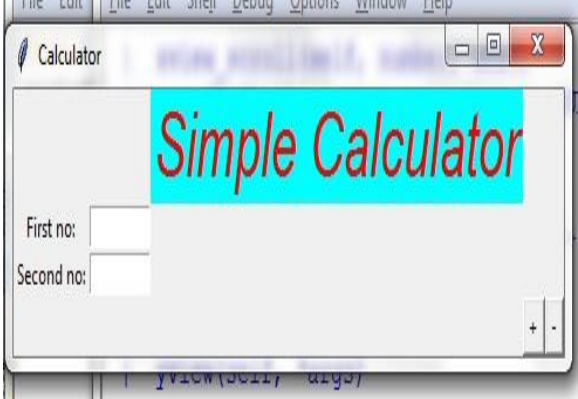
Example program to print calculator

Entry widget

- The Entry widget is used to display a single-line text field for accepting values from a user.
- Syntax:

```
a=Button(root,text='+',command=printadd)
```

Program to Create Simple Calculator

Calc.py	Output
<pre>from tkinter import* root=Tk() root.title('Calculator') def printadd(): x=float(t1.get()) y=float(t2.get()) s1=x+y s=str(s1) res=Label(root,text="Result is:"+s) res.grid(row=4,column=0) def printsub(): x=float(t1.get()) y=float(t2.get()) s1=x-y s=str(s1) res=Label(root,text="Result is:"+s) res.grid(row=4,column=0) #creating widgets l=Label(root,text="Simple Calculator") l.config(font=('calibri',30,'italic'),bg='cyan',fg='red') l1=Label(root,text="First no:") l2=Label(root,text="Second no:") t1=Entry(root,width=8) t2=Entry(root,width=8) a=Button(root,text='+',command=printadd) s=Button(root,text='-',command=printsub) #attaching he widgets to the window l.grid(row=0,column=2) l1.grid(row=1,column=0) t1.grid(row=1,column=1) l2.grid(row=2,column=0) t2.grid(row=2,column=1) a.grid(row=3,column=4) s.grid(row=3,column=5)</pre>	

Checkbutton, Radiobutton widgets


Checkbutton

- The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
- Syntax:
- `Cb= Checkbutton(parent,text=' ')`

Radiobutton

- The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time.
- Syntax:

Example program

ButtonTest.py	Output
<pre> from tkinter import* root=Tk() root.title("KSR window") #giving the title to the window cb1=Checkbutton(root,text='Python') cb2=Checkbutton(root,text='Graphics') m=Radiobutton(root,text='Male',value =0) f=Radiobutton(root,text='Female',valu e=1) cb1.pack() cb2.pack() m.pack() f.pack() root.mainloop() </pre>	

Menu widget

The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.

Syntax:

```
menu_bar=Menu(root)
```

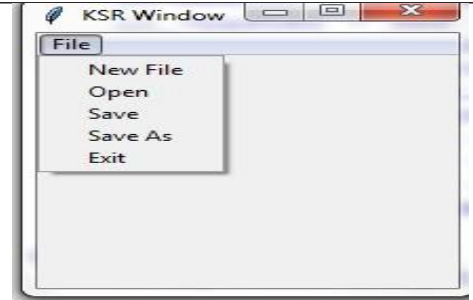
Menu example

Menutest.py	Output
-------------	--------

```

from tkinter import*
root=Tk()
root.title("KSR Window")
menu_bar=Menu(root)
file_menu=Menu(menu_bar,tearoff=0)
file_menu.add_command(label='New
File',command=root.destroy)
file_menu.add_command(label='Open
',command=root.destroy)
file_menu.add_command(label='Save',command=root.destroy)
file_menu.add_command(label='Save
As',command=root.destroy)
file_menu.add_command(label='Exit',command=root.destroy)
menu_bar.add_cascade(label='File',menu=file_menu)
root.config(menu=menu_bar)
root.mainloop()

```



Turtle - Introduction

- Turtle graphics is a popular way for introducing programming to kids.
- Virtual turtles can be programmed to move around the screen.
- The turtle draws lines as it moves.
- The user can write turtle programs that draw beautiful shapes and learn to program at the same time.

Example to draw 'square'

Sq.py	Output
<pre> from turtle import * forward(100) left(90) forward(100) left(90) forward(100) left(90) forward(100) </pre>	A screenshot of a Python Turtle Graphics window. The window title is "Python Turtle Graphics". The main area is a white canvas where a square has been drawn. The square is centered on the canvas. The turtle's head is visible at the bottom-left corner of the square, pointing downwards.

The turtle draws a line behind it as it moves. This program draws a square. The steps given to the program are:

- Move forward 100 steps. (In the beginning, the turtle is facing to the right.)
- Turn 90 degrees to the left.
- Move forward 100 steps.

- Turn 90 degrees to the left.
- Move forward 100 steps.
- Turn 90 degrees to the left.
- Move forward 100 steps. The turtle has ended up where it started.

Moving turtle

- By calling these functions, the turtle can be made to move around the screen. Imagine the turtle holding a pen down on the ground and drawing a line as it moves around.
- The turtle's position is two numbers: the X coordinate and Y coordinate.
- **forward(*distance*)**
The forward() function moves the turtle *distance* number of steps in the current direction. If the pen is down (see pendown() and penup()) a line will be drawn as the turtle moves forward. If *distance* is a negative number, the turtle will move backwards.
- **backward(*distance*)**
The backward() function moves the turtle *distance* number of steps in **opposite direction** the current direction. If the pen is down (see pendown() and penup()) a line will be drawn as the turtle moves backward. If *distance* is a negative number, the turtle will move forward.
- **right(*angle*)**
The right() function will change the current direction clockwise by *angle* degrees. If you imagine being above the turtle looking down, the turtle turning right looks like it is turning clockwise. The turtle will not move; it will only change the direction it is facing.
- **left(*angle*)**
The left() function will change the current direction counter-clockwise or anti-clockwise by *angle* degrees. If you imagine being above the turtle looking down, the turtle turning left looks like it is turning counter-clockwise or anti-clockwise. The turtle will not move; it will only change the direction it is facing.
- **goto(*x*, *y*)**
The goto() function will immediately move the turtle to the given *x* and *y* coordinates. If the pen is down (see pendown() and penup()) a line will be drawn from the previous coordinates to the new coordinates.

This example moves the to several x and y coordinates while drawing a line behind it:

```
from turtle import *
goto(50, 50)
goto(-50, 50)
goto(100, -50)
goto(-50, -50)
```

- **setx(*x*)**
The goto() function will immediately move the turtle to the given *x* coordinate. The turtle's y coordinate will stay the same. If the pen is down (see pendown() and penup()) a line will be drawn from the previous coordinates to the new coordinates.
- **sety(*y*)**

The `goto()` function will immediately move the turtle to the given **y* coordinate. The turtle's x coordinate will stay the same. If the pen is down (see `pendown()` and `penup()`) a line will be drawn from the previous coordinates to the new coordinates.

- **setheading(*heading*)**

The `setheading()` function will change the current direction to the *heading* angle. If you imagine being above the turtle looking down, the turtle turning left looks like it is turning counter-clockwise or anti-clockwise. The turtle will not move; it will only change the heading it is facing.

To draw 'triangle'

```
import turtle
turtle.forward(100)
turtle.right(90)
turtle.forward(100)
turtle.home()
```

Drawing

- **pendown()**

The `pendown()` function will cause the turtle to draw as it moves around. The line it draws can be set with the `pencolor()` and `pensize()` functions.

- **penup()**

The `penup()` function will cause the turtle to draw as it moves around. The line it draws can be set with the `pencolor()` and `pensize()` functions.

- **pensize(*size*)**

The `pensize()` function sets the width of the line that the turtle draws as it moves.

- **clear()**

The `clear()` function will erase all the line drawings on the screen. This function does not move the turtle.

- **reset()**

The `reset()` function will erase all the line drawings on the screen and return the turtle to the (0, 0) coordinate and facing 0 degrees. This function does the same thing as calling the `clear()` and `home()` function.

Example Program:

```
import turtle
turtle.forward(100)
turtle.right(90)
turtle.forward(100)
turtle.home()
turtle.reset() #clears and sets the turtle to (0,0) position
#turtle.clear()
```

Mathematics

Provides functions for specialized mathematical operations. The “math” module provides

The following functions are provided by this module:

- ✓ Number-theoretic and representation functions
- ✓ Power and logarithmic function
- ✓ Trigonometric function
- ✓ Angular conversion
- ✓ Hyperbolic functions
- ✓ Special functions
- ✓ Constants

math.ceil(x)

- Return the ceiling of x, the smallest integer greater than or equal to x.

Example:

```
>>> math.ceil(12.3)
13
```

math.floor(x)

- Return the floor of x, the largest integer less than or equal to x.

```
>>> math.floor(12.3)
12
```

math.factorial(x)

- Return x factorial.

```
>>> math.factorial(5)
120
```

math.gcd(a, b)

- Return the greatest common divisor of the integers a and b.

```
>>> math.gcd(12,26)
2
```

math.exp(x)

- Return e^{**x}

```
>>> math.exp(5)
148.4131591025766
```

math.log(x, base)

- With one argument, return the natural logarithm of x (to base e).
- With two arguments, return the logarithm of x to the given base.

```
>>> math.log(10,10)
1.0
>>> math.log(10,3)
2.095903274289385
```

math.pow(x, y)

- Return x raised to the power y.

```
>>> math.pow(2,3)
```

```
8.0
```

math.sqrt(x)

- Return the square root of x.

```
>>> math.sqrt(16)
```

```
4.0
```

math.cos(x)

- Return the cosine of x radians.

math.sin(x)

- Return the sine of x radians.

math.tan(x)

- Return the tangent of x radians.

math.degrees(x)

- Convert angle x from radians to degrees.

math.radians(x)

- Convert angle x from degrees to radians

math.cos(x)

- Return the cosine of x radians.

math.sin(x)

- Return the sine of x radians.

math.tan(x)

- Return the tangent of x radians.

math.degrees(x)

- Convert angle x from radians to degrees.

math.radians(x)

- Convert angle x from degrees to radians

math.pi

- The mathematical constant $\pi = 3.141592\dots$, to available precision.

math.e

- The mathematical constant $e = 2.718281\dots$, to available precision.

math.inf

- A floating-point positive infinity. (For negative infinity, use `-math.inf`)

Assertion

- Assertion is a conditional statement.
- This is used to test an assumption in the program.
- It verifies a conditional statement must be always TRUE.
- Syntax:

- `assert True==fun(x)`
- Here `assert` is keyword, `fun()` is user defined function, `x` is argument
- `fun(x)` returns a Boolean value which is either `True` or `False`.
- If the returned value is `True` continues, otherwise Raises an `AssertionError`

Data Compression

- The `zlib` module provides a lower-level interface to many of the functions in the `zlib` compression library from the GNU project.

`zlib.compress(data, level=-1)`

- Compresses the bytes in `data`, returning a bytes object containing compressed data.
- `level` is an integer from 0 to 9 or -1 controlling the level of compression
- 1 is fastest and produces the least compression, 9 is slow

`zlib.decompress(data, wbits=MAX_WBITS, bufsize=DEF_BUF_SIZE)`

- Decompresses the bytes in `data`, returning a bytes object containing the uncompressed data.

Example:

```
a=b'i want to meet you on sunday'
>>> a
b'i want to meet you on sunday'
>>> c=zlib.compress(a,level=2)
>>> c
b'x^\xcbT(O\xcc+Q(\xc9W\xc8MM-Q\xa8\xcc/U\xc8\xcfS(\.xcdKI\xac\x04\x00\x90\xbf\n@'
p=zlib.decompress(c)
>>> p
b'i want to meet you on sunday'
```

`zlib.adler32(data, value)`

- Computes an Adler-32 checksum of data. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) The result is an unsigned 32-bit integer.
- If `value` is present, it is used as the starting value of the checksum; otherwise, a default value of 1 is used.
- Passing in `value` allows computing a running checksum over the concatenation of several inputs.
- `cr=zlib.adler32(a,12)`
- `>>> cr`
- 2448624203

`zlib.crc32(data, value)`

- Computes a CRC (Cyclic Redundancy Check) checksum of data.
- The result is an unsigned 32-bit integer.

- If value is present, it is used as the starting value of the checksum; otherwise, a default value of 0 is used.
- Passing in value allows computing a running checksum over the concatenation of several inputs.
- `cr=zlib.crc32(a,12)`
- `>>> cr`
- `1321257511`

Multithreading

- The program in execution is called “process”
- Executing the number of processes simultaneously or concurrently is called “multi processing”
- The light-weight process is called “Thread”
- Executing the number of threads is called “multi threading”
- A process can be divided into several threads, each will do a specific task.
- Multiple threads within a process share same address space, and hence share information and communicate more easily.
- The threads can communicate with each other which is called “Inter thread communication”
- These are cheaper than processes.
- The Thread has three things:
 - It has beginning
 - It has an execution sequence
 - It has conclusion
- A thread can be pre-empted (interrupted)
- It can be temporarily put on hold, while other threads are running. This is called “Yielding”.
- To start new thread we call the following method in Python 2.7*
 - `thread.start_new_thread(function, arguments)`

Creating the New thread

```
import time
import thread
def print_time(threadname,delay): #function definition
    count=0
    while count<5:
        time.sleep(delay)
        count+=1
        print "The thread name is:",threadname,"Delay is:",delay,"\n"
#creating the thread
try:
    thread.start_new_thread(print_time,("Thread-1",2))
    thread.start_new_thread(print_time,("Thread-2",4))
```

```
except:
    print "There is some error in threading"
```

```
>>> The thread name is: Thread-1 Delay is: 2
The thread name is: Thread-2 Delay is: 4
The thread name is: Thread-1 Delay is: 2
The thread name is: Thread-1 Delay is: 2
The thread name is: Thread-2 Delay is: 4
The thread name is: Thread-1 Delay is: 2
The thread name is: Thread-1 Delay is: 2
The thread name is: Thread-2 Delay is: 4
The thread name is: Thread-2 Delay is: 4
The thread name is: Thread-2 Delay is: 4
```

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.

Thread class methods

- 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.

```
import time
```

```

import threading
#defining class
class MyThread(threading.Thread):
    def __init__(self,threadID,n,c):
        threading.Thread.__init__(self)
        self.tID=threadID
        self.name=n
        self.counter=c
    def run(self): #entry point to the thread
        for a in range(1,11):
            print("name",self.name,"ID is:",self.tID," :5 * ",a,"=",5*a,"\n")
            time.sleep(self.counter) #used to delay
#creating the threads
t1=MyThread(1,"Guido Thread",2)
t2=MyThread(2,"Steev Jobs",4)
#starting the threads
t1.start()
t2.start()

```

To know the information of the current threads

Threadtest1.py	Output
<pre> print("The name of the First Thread is",t1.getName(),"\n") print("The Name of the Second Thread is",t2.getName(),"\n") print("The Name of the Second Thread is",t3.getName(),"\n") print("The active threads are :",threading.activeCount()) print("The list of threads is:",threading.enumerate()) </pre>	<pre> The active threads are : 5 The list of threads is: [<_MainThread(MainThread, started 2544)>, <Thread(SocketThread, started daemon 3552)>, <MyThread(Guido Thread, started 2560)>, <MyThread(Steev Jobs, started 2088)>, <MyThread(KSR Thread, started 1488)>] </pre>

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.

Testing

- **Unit Test**
 - This test is invented by Eric Gamma in 1977
 - This test is used to test the individual units or functions of the program.
 - The function may be defined to do some specific task.
 - This functions is tested for all possible value, for knowing where it is performing its intended task correctly or not

Goals of Unit Test

There are three goals of unit test

- To make it easy to write tests
- To make it easy to run tests
- To make it easy to tell if the tests are passed

How many tests should we have

- Test at least one “typical” case.
- The objective of the testing is not to prove that your program is working.
- It is to try to find out where it does not test “Extreme” case you can think.
- For example you want to write test case for “*sort()*” function over a list.
- Then some issues you can consider are:
 - What if the list contains equal number?
 - Do the first element and last element moved to the correct position?
 - Can you sort a 1-element list without getting an error?
 - How about an empty list?

unittest module

- In Python we have a module called “unittest” to support unit test.
- to use these tests we create a class that Inherits the properties from “TestCase” class of this module.
- Define a method in this class and use the different methods of the TestCase class.

Commonly used methods of TestCase class

assertEqual(a,b)

assertNotEqual(a,b)

assertTrue(x)

assertFalse(x)

assertIs(a,b)

assertIsNone(a,b)

assertIn(a,b)

assertNotIn(a,b)