**PYTHON**

**UNIT-IV**

**PYTHON DATABASE ACCESS:**

The python standard database interface is the python DB-API. (Application Program Interface). Most python database interfaces adhere to this standard.

The python database API supply with database neutral programming interface for different databases.

These are : MYSQL, SQLite, MSSQL, Oracle, PostgreSQL, Informix, Sybase, Interbase etc. The API supports all these database servers.

The API includes:

* 1. Importing the API Module
  2. Acquiring a connection with the database
  3. Issuing SQL commands and stored procedures.
  4. Closing the connection.

Benefits of Python Database Programming:

1. Programming is simple, efficient compare to the other languages.
2. Python database and python programs both are portable.
3. API of python for the databases is compatible with other databases also.

Some common database connectivity to learn:

1. connecting using mysql using python3 default interface
2. connecting using pymysql module in jupiter notebook
3. connect with SQLite
4. connect using cx\_Oracle module in python
5. Connect with SQL server using pyodbc
6. connect with MS-Access using pyodbc

**Python with MYSQL connectivity using default interface:**

To connect with the MYSQL database we must have MYSQL software and then we need to import mysql.connector module in the program.

The mysql.connector is not a built-in module. So, we need to install it using pip installer.

>>>python -m pip install mysql-connector

Or we can download the file:

mysql-connector-python-8.0.13.tar.gz

and then extract the archived file.

Then open the folder of mysql-connector and execute the code:

$python setup.py install

Once installation is over , we can import the file using statement:

>>>import mysql.connector

**Creating the connection:**

Connect(): To create a connection between the MYSQL database and the python application this method is used. This method is available in the mysql.connector module.

Syntax:

Connectionobject=mysql.connector.connect(host=”hostname”, user=”username”,passwd=”password”, database=”databasename”)

Here the host name is local host or ip of the host.

Example:

import mysql.connector

myconn=mysql.connector.connect(host=”localhost”,user=”root”,passwd=””, database=”db1”)

print(myconn)

o/p:

mysql.connector.connection….

#object created

Creating a Cursor Object:

The MySQLCursor of mysql-connector-python (and similar libraries) is used to execute statements to communicate with the MySQL database.

Cursor(): This method is used to create a cursor object for the mysqlconnector object.

Syntax/example:

mycursor=myconn.cursor()

Now the cursor object mycursor is created and can be used to execute mysql commands.

Fetchall():

The method fetches all rows of a query result set and returns a list of tuples. If no more rows are available, it returns an empty list. You must **fetch all** rows for the current query before executing new statements using the same connection.

**fetchone():**

**fetchall()** fetches all the rows of a query result. An empty list is returned if there is no record to fetch the cursor. **fetchone()** method returns one row or a single record at a time. It will return None if no more rows / records are available.

**Ex1: To see the list of databases: (p1.py)**

import mysql.connector

myconn=mysql.connector.connect(host=”localhost”,user=”root”,passwd=””)

mycursor=myconn.cursor()

mycursor.execute(“show databases”)

dbs=mycursor.fetchall()

for x in dbs:

print(x)

myconn.close()

o/p:

it will show all the list of databases in MYSQL databases.

**Ex1.1: to show tables from a database(p1.1.py)**

**shows the tables list from database db1**

#TO SHOW TABLES FROM DATABASE db1

import mysql.connector

myconn=mysql.connector.connect(host="localhost",user="root",passwd="")

mycursor=myconn.cursor()

mycursor.execute("show databases")

for i in mycursor.fetchall():

print(i)

x=input("enter a database name")

mycursor.execute("use "+x)

mycursor.execute("show tables")

dbs=mycursor.fetchall()

print("tables are...............")

for x in dbs:

print(x)

y= input("enter a table name")

mycursor.execute("select \* from "+y)

dbs=mycursor.fetchall()

print("content is...............")

for x in dbs:

print(x)

myconn.close()

o/p:

**Ex2: to show the content of a table employee from database db1 (p2.py)**

Solution:

import mysql.connector

myconn=mysql.connector.connect(host=”localhost”, user=”root”, passwd=””, database=”db1”)

mycursor=myconn.cursor()

mycursor.execute(“select \* from employee”)

rows=mycursor.fetchall()

for x in rows:

print(x)

o/p:

(1, 'sree', 'lakshmi', 'nbihar', 'null')

(2, 'ree', 'hmi', 'nihar', 'null')

(3, 'reena', 'hunmi', 'vinihar', 'null')

(4, 'reema', 'hnmi', 'vainihar', 'london')

(5, 'rema', 'ahnmi', 'nihar', 'london')

**EX2.1:**

**#DISPLAYING THE CONTENT OF EMPLOYEE TABLE with formatting(p2.1.py)**

import mysql.connector

con = mysql.connector.connect( host="localhost",user="root",password="",database="db1")

mycursor=con.cursor()

mycursor.execute("desc employee")

mycursor.fetchall()

mycursor.execute("select \* from employee")

desc=mycursor.description

print('{:<7s} {:<7s} {:<7s} {:<7s}'. format(desc[0][0],desc[1][0],desc[2][0],desc[3][0]))

print('{:<7s} {:<7s} {:<7s} {:<7s}'.format('-'\*7, '-'\*7,'-'\*7, '-'\*7))

result= mycursor.fetchall()

for row in result:

print("%d\t%s\t%s\t%s"%(row[0],row[1],row[2],row[3]))

O/P:

empid fname lname address

------- ------- ------- -------

1 sree lakshmi nbihar

2 ree hmi nihar

3 reena hunmi vinihar

4 reema hnmi vainihar

5 rema ahnmi nihar

**EX2.2:**

**#DISPLAYING THE CONTENT OF EMPLOYEE TABLE with formatting(p2.2.py)**

#reading the content of table:employee with formatting from database: db1

import mysql.connector

myconn=mysql.connector.connect(host="localhost",user="root",passwd="",database="db1")

mycursor=myconn.cursor()

mycursor.execute("select empid,fname,lname from employee")

result=mycursor.fetchall()

print("employee details: \nid\t\tfname\t\tlname\n.......................................")

for row in result:

print("%d\t\t%s\t\t%s"%(row[0],row[1],row[2]))

myconn.close()

o/p:

employee details:

id fname lname

.......................................

1 sree lakshmi

2 ree hmi

3 reena hunmi

4 reema hnmi

5 rema ahnmi

**ex3: create a database using python mysql connector.(p3.py)**

Solution:

import mysql.connector

myconn=mysql.connector.connect(host="localhost", user="root", passwd="")

mycursor=myconn.cursor()

x=input("enter name for new database to create:")

mycursor.execute("create database "+x)

print("database created: "+x)

myconn.close()

o/p:

enter name for new database to create:hello

database created: hello

**ex4: to connect multiple databases using mysql connector.(p4.py)**

Solution:

import mysql.connector

myconn1=mysql.connector.connect(host=”localhost”, user=”root”, passwd=””, database=”db1”)

myconn2=mysql.connector.connect(host=”localhost”, user=”root”, passwd=””, database=”college”)

mycursor1=myconn1.cursor()

mycursor2=myconn2.cursor()

mycursor1.execute(“select \* from employee”)

rows1=mycursor1.fetchall()

for x in rows1:

print(x)

print(“end of employee”)

mycursor2.execute(“select \* from animals”)

rows2=mycursor2.fetchall()

for y in rows2:

print(y)

print(“end of animals”)

myconn1.close()

myconn2.close()

o/p:

(1, 'sree', 'lakshmi', 'nbihar', 'null')

(2, 'ree', 'hmi', 'nihar', 'null')

(3, 'reena', 'hunmi', 'vinihar', 'null')

(4, 'reema', 'hnmi', 'vainihar', 'london')

(5, 'rema', 'ahnmi', 'nihar', 'london')

end of employee

(1, 'dog')

(2, 'cat')

(3, 'penguin')

(4, 'lax')

(5, 'whale')

(6, 'ostrich')

end of animals

**ex5: creating a table P5.py)**

solution:

#CREATING A TABLE

import mysql.connector

#creating the connection object

myconn=mysql.connector.connect(host="localhost", user="root", passwd="", database="pythondb1")

#creating the cursor object

mycursor=myconn.cursor()

try:

#creating a table with name employee having four coulmns i.e: name, id, salary

x=input("enter name of employee table")

Dbs=mycursor.execute("create table "+x+ "(name varchar(20) not null, id int(20) not null primary key, salary float not null)")

mycursor.execute("show tables")

tab=mycursor.fetchall()

for i in tab:

print(i)

print("table created: ",x)

print("table structure is:")

mycursor.execute("desc "+x)

des=mycursor.fetchall()

for i in des:

print(i)

mycursor.execute("drop table " +x)

print("table dropped: ",x)

except:

myconn.rollback()

myconn.close()

o/p:

enter name of employee tableg

('a',)

('b',)

('c',)

('d',)

('emp',)

('emp1',)

('emp2',)

('emp5',)

('emp6',)

('emplo',)

('g',)

('tab1',)

table created: g

table structure is:

('name', 'varchar(20)', 'NO', '', None, '')

('id', 'int(20)', 'NO', 'PRI', None, '')

('salary', 'float', 'NO', '', None, '')

table dropped: g

**ex5.1 dropping a table (p5.1.py)**

#DROPPING A TABLE

import mysql.connector

#creating the connection object

myconn=mysql.connector.connect(host="localhost", user="root", passwd="", database="pythondb1")

#creating the cursor object

mycursor=myconn.cursor()

try:

#ENTER TABLE name to drop

mycursor.execute("show tables")

s=mycursor.fetchall()

for i in s:

print(i)

x=input("enter name of table")

mycursor.execute("drop table " +x)

print("table dropped: ",x)

except:

myconn.rollback()

myconn.close()

o/p:

('b',)

('c',)

('d',)

('emp',)

('emp1',)

('emp2',)

('emp5',)

('emp6',)

('emplo',)

('tab1',)

enter name of tableb

table dropped: b

**Ex6: alter table(p6.py)**

#it is to alter a table emp in pythondb1

import mysql.connector

myconn=mysql.connector.connect(host="localhost",user="root",passwd="", database="pythondb1")

mycursor=myconn.cursor();

try:

mycursor.execute("show tables")

s=mycursor.fetchall()

for i in s:

print(i)

x=input("enter name of table")

mycursor.execute("alter table "+x+" add branch varchar(20) not null")

print("\n\ntable structure adding branch field is:")

mycursor.execute("desc "+x)

des=mycursor.fetchall()

for i in des:

print(i)

mycursor.execute("alter table "+x+" drop branch")

print("\n\ntable structure is after removing branch field:")

mycursor.execute("desc "+x)

des=mycursor.fetchall()

for i in des:

print(i)

except:

mycursor.rollback()

myconn.close()

o/p:

('c',)

('d',)

('emp',)

('emp1',)

('emp2',)

('emp5',)

('emp6',)

('emplo',)

('tab1',)

enter name of tablec

table structure adding branch field is:

('name', 'varchar(20)', 'NO', '', None, '')

('id', 'int(20)', 'NO', 'PRI', None, '')

('salary', 'float', 'NO', '', None, '')

('branch', 'varchar(20)', 'NO', '', None, '')

table structure is after removing branch field:

('name', 'varchar(20)', 'NO', '', None, '')

('id', 'int(20)', 'NO', 'PRI', None, '')

('salary', 'float', 'NO', '', None, '')

**Ex7:inserting data into a table (p7.py)**

#it is to insert into a table emp1 in pythondb1

import mysql.connector

myconn=mysql.connector.connect(host="localhost",user="root",passwd="", database="pythondb1")

mycursor=myconn.cursor();

q1="insert into emp1(name, id, salary, deptid, branchname) values(%s,%s,%s,%s,%s)"

nm=input("enter name")

iden=int(input("enter id"))

sal=float(input("enter salary"))

did=int(input("enter dept id"))

bname=input("enter branch name")

v1=(nm,iden,sal,did,bname)

try:

mycursor.execute(q1,v1)

myconn.commit()

except:

myconn.rollback()

print(mycursor.rowcount, "record inserted")

mycursor.execute("select \* from emp1")

tbs=mycursor.fetchall()

print("content in employee table is........")

for x in tbs:

print(x)

myconn.close()

o/p:

enter namea

enter id999

enter salary909090

enter dept id11

enter branch namecs

1 record inserted

content in employee table is........

('john', 110, 25000.0, 201, 'cse', '')

('sam', 111, 55000.0, 201, 'cse', '')

('a', 999, 909090.0, 11, 'cs', '')

**Ex8: reading the content of table:employee from database: db1 (p8.py)**

import mysql.connector

myconn=mysql.connector.connect(host="localhost",user="root",passwd="",database="db1")

mycursor=myconn.cursor()

try:

mycursor.execute("select \* from employee")

result=mycursor.fetchall()

print("employee details........................................")

for x in result:

print(x)

except:

myconn.rollback()

mycursor.execute("select empid,fname from employee")

result=mycursor.fetchall()

print("employee :......name.........id.......................")

for x in result:

print(x)

myconn.close()

o/p:

employee details........................................

(1, 'sree', 'lakshmi', 'nbihar', 'null')

(2, 'ree', 'hmi', 'nihar', 'null')

(3, 'reena', 'hunmi', 'vinihar', 'null')

(4, 'reema', 'hnmi', 'vainihar', 'london')

(5, 'rema', 'ahnmi', 'nihar', 'london')

employee :......name.........id.......................

(1, 'sree')

(2, 'ree')

(3, 'reena')

(4, 'reema')

(5, 'rema')

**Ex9: updating a content of a table**

#updating the content of table:employee from database: db1

import mysql.connector

myconn=mysql.connector.connect(host="localhost",user="root",passwd="",database="db1")

mycursor=myconn.cursor()

mycursor.execute("select \* from employee")

t=mycursor.fetchall()

for i in t:

print(i)

x=input("enter 1st firstname")

y=input("enter city for the name")

#mycursor.execute("update employee set city=\"gunupur\" where fname=\"sam1\" ")

mycursor.execute("update employee set city=\""+y+"\" where fname=\""+x+"\"")

#mycursor.reset()

mycursor.execute("select \* from employee")

result=mycursor.fetchall()

print("employee details: \nempid\tfname\tlname\taddress\tcity\n......................................")

for row in result:

print("%d\t%s\t%s\t%s\t%s"%(row[0],row[1],row[2],row[3],row[4]))

myconn.close()

o/p:

(1, 'sree', 'lakshmi', 'nbihar', 'null')

(2, 'ree', 'hmi', 'nihar', 'null')

(3, 'reena', 'hunmi', 'vinihar', 'null')

(4, 'reema', 'hnmi', 'vainihar', 'london')

(5, 'rema', 'ahnmi', 'nihar', 'london')

enter 1st firstnamesree

enter city for the namevizag

employee details:

empid fname lname address city

....... .......... ......... . ...... .....

1 sree lakshmi nbihar vizag

2 ree hmi nihar null

3 reena hunmi vinihar null

4 reema hnmi vainihar london

5 rema ahnmi nihar london

**Python with pymysql module using Jupiter notebook:**

Ex 10#connecting using pymysql module in jupiter notebook(p10.py)

import pymysql

from pymysql import Error

con = pymysql.connect(host="localhost",user="root",password="", database="db1")

mycursor=con.cursor()

mycursor.execute("select \* from employee")

result=mycursor.fetchall()

for row in result:

print(row)

**Python with pyodbc module using MSACCESS:**

Ex11: #it is used to connect with MSACCESS

import sys

import pyodbc

#conn = pyodbc.connect(r'Driver={Microsoft Access Driver(\*.mdb,\*.accdb)}; DBQ=D:\DATABASE\db1.emp;')

conn=pyodbc.connect(dsn='test1') #to move control panel,administrative tools, data sources,systemDSN,add "test1"

mycursor = conn.cursor()

query="select \* from emp"

mycursor.execute(query)

result=mycursor.fetchall()

for row in result:

print(row)

**Python with sqlite3 module using SQLite3:**

Ex12:

#connect with SQLite

import sqlite3

conn = sqlite3.connect('db1')

mycursor=conn.cursor()

mycursor.execute("select \* from employee")

result=mycursor.fetchall()

for row in result:

print(row)

conn.close()

**Python with pyodbc module using SQLServer:**

Ex13:

#it is used to connect with SQL server

import sys

import pyodbc

conn = pyodbc.connect('Driver={SQL Server};'

'Server=DESKTOP-NTMMPFV;'

'Database=db1;'

'Trusted\_Connection=yes;')

mycursor = conn.cursor()

query="select \* from Student where Branch\_Id like 'CSE' and vcStatus='ACTIVE'"

mycursor.execute(query)

desc= mycursor.description

print('{:<10s} {:<20s} {:<20s} {:<6s} {:<20s}'.format(desc[0][0],desc[1][0], desc[2][0], desc[3][0],desc[4][0]))

print('{:<10s} {:<20s} {:<20s} {:<6s} {:<20s}'.format('-'\*20, '-'\*20, '-'\*20,'-'\*10, '-'\*6)) # Print divider

result=mycursor.fetchall()

for row in result:

print(row)

**Python with odbc module using Oracle:**

Ex14: connecting to oracle database

import cx\_Oracle

my\_dsn = cx\_Oracle.makedsn("localhost",port='8080',sid="apex")

myconn=cx\_Oracle.connect(user='system', password='manager', dsn='my\_dsn')

mycursor=myconn.cursor()

mycursor.execute(“show databases”)

dbs=mycursor.fetchall()

for x in dbs:

print(x)

myconn.close()

**PYTHON GUI Programming**

**Python - GUI Programming (Tkinter)**

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. We would look this option in this chapter.
* **wxPython** − This is an open-source Python interface for wxWindows [http://wxpython.org](http://wxpython.org/).
* **JPython** − JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine [http://www.jython.org](http://www.jython.org/).

There are many other interfaces available, which you can find them on the net.

## 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.

## Example

import tkinter

top = tkinter.Tk()

# Code to add widgets will go here...

top.mainloop()

This would create a following window −



Python offers multiple options for developing GUI (Graphical User Interface). Out of all the GUI methods, tkinter is most commonly used method. It is a standard Python interface to the Tk GUI toolkit shipped with Python. Python with tkinter outputs the fastest and easiest way to create the GUI applications. Creating a GUI using tkinter is an easy task.

## Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table −

|  |  |
| --- | --- |
| **Sr.No.** | **Operator & Description** |
| 1 | [**Button**](https://www.tutorialspoint.com/python/tk_button.htm)  The Button widget is used to display buttons in your application. |
| 2 | [**Canvas**](https://www.tutorialspoint.com/python/tk_canvas.htm)  The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application. |
| 3 | [**Checkbutton**](https://www.tutorialspoint.com/python/tk_checkbutton.htm)  The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time. |
| 4 | [**Entry**](https://www.tutorialspoint.com/python/tk_entry.htm)  The Entry widget is used to display a single-line text field for accepting values from a user. |
| 5 | [**Frame**](https://www.tutorialspoint.com/python/tk_frame.htm)  The Frame widget is used as a container widget to organize other widgets. |
| 6 | [**Label**](https://www.tutorialspoint.com/python/tk_label.htm)  The Label widget is used to provide a single-line caption for other widgets. It can also contain images. |
| 7 | [**Listbox**](https://www.tutorialspoint.com/python/tk_listbox.htm)  The Listbox widget is used to provide a list of options to a user. |
| 8 | [**Menubutton**](https://www.tutorialspoint.com/python/tk_menubutton.htm)  The Menubutton widget is used to display menus in your application. |
| 9 | [**Menu**](https://www.tutorialspoint.com/python/tk_menu.htm)  The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton. |
| 10 | [**Message**](https://www.tutorialspoint.com/python/tk_message.htm)  The Message widget is used to display multiline text fields for accepting values from a user. |
| 11 | [**Radiobutton**](https://www.tutorialspoint.com/python/tk_radiobutton.htm)  The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time. |
| 12 | [**Scale**](https://www.tutorialspoint.com/python/tk_scale.htm)  The Scale widget is used to provide a slider widget. |
| 13 | [**Scrollbar**](https://www.tutorialspoint.com/python/tk_scrollbar.htm)  The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes. |
| 14 | [**Text**](https://www.tutorialspoint.com/python/tk_text.htm)  The Text widget is used to display text in multiple lines. |
| 15 | [**Toplevel**](https://www.tutorialspoint.com/python/tk_toplevel.htm)  The Toplevel widget is used to provide a separate window container. |
| 16 | [**Spinbox**](https://www.tutorialspoint.com/python/tk_spinbox.htm)  The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values. |
| 17 | [**PanedWindow**](https://www.tutorialspoint.com/python/tk_panedwindow.htm)  A PanedWindow is a container widget that may contain any number of  panes, arranged horizontally or vertically. |
| 18 | [**LabelFrame**](https://www.tutorialspoint.com/python/tk_labelframe.htm)  A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. |
| 19 | [**tkMessageBox**](https://www.tutorialspoint.com/python/tk_messagebox.htm)  This module is used to display message boxes in your applications. |

Let us study these widgets in detail −

## Standard attributes

Let us take a look at how some of their common attributes.such as sizes, colors and fonts are specified.

[Dimensions](https://www.tutorialspoint.com/python/tk_dimensions.htm)

[Colors](https://www.tutorialspoint.com/python/tk_colors.htm)

[Fonts](https://www.tutorialspoint.com/python/tk_fonts.htm)

[Anchors](https://www.tutorialspoint.com/python/tk_anchors.htm)

[Relief styles](https://www.tutorialspoint.com/python/tk_relief.htm)

[Bitmaps](https://www.tutorialspoint.com/python/tk_bitmaps.htm)

[Cursors](https://www.tutorialspoint.com/python/tk_cursors.htm)

Let us study them briefly −

**To create a tkinter:**

1. Importing the module – tkinter
2. Create the main window (container)
3. Add any number of widgets to the main window
4. Apply the event Trigger on the widgets.

Importing tkinter is same as importing any other module in the python code. Note that the name of the module in Python 2.x is ‘Tkinter’ and in Python 3.x is ‘tkinter’.

import tkinter

There are two main methods used you the user need to remember while creating the Python application with GUI.

1. **Tk(screenName=None,  baseName=None,  className=’Tk’,  useTk=1):** To create a main window, tkinter offers a method ‘Tk(screenName=None,  baseName=None,  className=’Tk’,  useTk=1)’. To change the name of the window, you can change the className to the desired one. The basic code used to create the main window of the application is:

m=tkinter.Tk() where m is the name of the main window object

1. **mainloop():** There is a method known by the name mainloop() is used when you are ready for the application to run. mainloop() is an infinite loop used to run the application, wait for an event to occur and process the event till the window is not closed.

m.mainloop()

|  |
| --- |
| import tkinter  m = tkinter.Tk()  '''  widgets are added here  '''  m.mainloop() |

tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

1. **pack() method:**It organizes the widgets in blocks before placing in the parent widget.
2. **grid() method:**It organizes the widgets in grid (table-like structure) before placing in the parent widget.
3. **place() method:**It organizes the widgets by placing them on specific positions directed by the programmer.

There are a number of widgets which you can put in your tkinter application. Some of the major widgets are explained below:

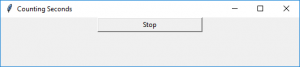
1. **Button**:To add a button in your application, this widget is used.  
   The general syntax is:

w=Button(master, option=value)

master is the parameter used to represent the parent window.  
There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **activebackground**: to set the background color when button is under the cursor.
  + **activeforeground**: to set the foreground color when button is under the cursor.
  + **bg**: to set he normal background color.
  + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the button.
  + **width**: to set the width of the button.
  + **height**: to set the height of the button.

|  |
| --- |
| import tkinter as tk  r = tk.Tk()  r.title('Counting Seconds')  button = tk.Button(r, text='Stop', width=25, command=r.destroy)  button.pack()  r.mainloop() |

Output:  


1. **Canvas:**It is used to draw pictures and other complex layout like graphics, text and widgets.  
   The general syntax is:

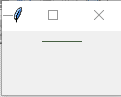
w = Canvas(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bd**: to set the border width in pixels.
  + **bg**: to set the normal background color.
  + **cursor**: to set the cursor used in the canvas.
  + **highlightcolor**: to set the color shown in the focus highlight.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*  master = Tk()  w = Canvas(master, width=40, height=60)  w.pack()  canvas\_height=20  canvas\_width=200  y = int(canvas\_height / 2)  w.create\_line(0, y, canvas\_width, y )  mainloop() |

Output:  


1. **CheckButton:**To select any number of options by displaying a number of options to a user as toggle buttons. The general syntax is:

w = CheckButton(master, option=value)

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **Title**: To set the title of the widget.
  + **activebackground**: to set the background color when widget is under the cursor.
  + **activeforeground**: to set the foreground color when widget is under the cursor.
  + **bg**: to set he normal backgrouSteganography

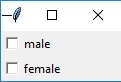
Break

Secret Code:

Attach a File:nd color.

* + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.

|  |
| --- |
| from tkinter import \*  master = Tk()  var1 = IntVar()  Checkbutton(master, text='male', variable=var1).grid(row=0,  sticky=W)  var2 = IntVar()  Checkbutton(master, text='female', variable=var2).grid(row=1,  sticky=W)  mainloop() |

Output:  


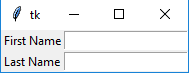
1. **Entry:**It is used to input the single line text entry from the user.. For multi-line text input, Text widget is used.  
   The general syntax is:

w=Entry(master, option=value)

master is the parameter used to represent the parent window.  
There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bd**: to set the border width in pixels.
  + **bg**: to set the normal background color.
  + **cursor**: to set the cursor used.
  + **command**: to call a function.
  + **highlightcolor**: to set the color shown in the focus highlight.
  + **width**: to set the width of the button.
  + **height**: to set the height of the button.

|  |
| --- |
| from tkinter import \*  master = Tk()  Label(master, text='First Name').grid(row=0)  Label(master, text='Last Name').grid(row=1)  e1 = Entry(master)  e2 = Entry(master)  e1.grid(row=0, column=1)  e2.grid(row=1, column=1)  mainloop() |

Output:  


1. **Frame:** It acts as a container to hold the widgets. It is used for grouping and organizing the widgets. The general syntax is:

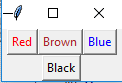
w = Frame(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
  + **bd**: to set the border width in pixels.
  + **bg**: to set the normal background color.
  + **cursor**: to set the cursor used.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*    root = Tk()  frame = Frame(root)  frame.pack()  bottomframe = Frame(root)  bottomframe.pack( side = BOTTOM )  redbutton = Button(frame, text = 'Red', fg ='red')  redbutton.pack( side = LEFT)  greenbutton = Button(frame, text = 'Brown', fg='brown')  greenbutton.pack( side = LEFT )  bluebutton = Button(frame, text ='Blue', fg ='blue')  bluebutton.pack( side = LEFT )  blackbutton = Button(bottomframe, text ='Black', fg ='black')  blackbutton.pack( side = BOTTOM)  root.mainloop() |

Output:  


1. **Label**: It refers to the display box where you can put any text or image which can be updated any time as per the code.  
   The general syntax is:

w=Label(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bg**: to set he normal background color.
  + **bg** to set he normal background color.
  + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the button.
  + **width**: to set the width of the button.
  + **height**” to set the height of the button.

|  |
| --- |
| from tkinter import \*  root = Tk()  w = Label(root, text='GeeksForGeeks.org!')  w.pack()  root.mainloop() |

Output:  
http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Screenshot-68-5.png

1. **Listbox**: It offers a list to the user from which the user can accept any number of options.  
   The general syntax is:

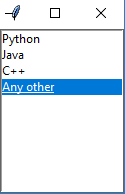
w = Listbox(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
  + **bg**: to set he normal background color.
  + **bd**: to set the border width in pixels.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*    top = Tk()  Lb = Listbox(top)  Lb.insert(1, 'Python')  Lb.insert(2, 'Java')  Lb.insert(3, 'C++')  Lb.insert(4, 'Any other')  Lb.pack()  top.mainloop() |

Output:  


1. **MenuButton**: It is a part of top-down menu which stays on the window all the time. Every menubutton has its own functionality. The general syntax is:

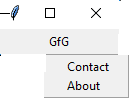
w = MenuButton(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **activebackground**: To set the background when mouse is over the widget.
  + **activeforeground**: To set the foreground when mouse is over the widget.
  + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.
  + **highlightcolor**: To set the color of the focus highlight when widget has to be focused.

|  |
| --- |
| from tkinter import \*    top = Tk()  mb =  Menubutton ( top, text = &quot;GfG&quot;)  mb.grid()  mb.menu  =  Menu ( mb, tearoff = 0 )  mb[&quot;menu&quot;]  =  mb.menu  cVar  = IntVar()  aVar = IntVar()  mb.menu.add\_checkbutton ( label ='Contact', variable = cVar )  mb.menu.add\_checkbutton ( label = 'About', variable = aVar )  mb.pack()  top.mainloop() |

Output:  


1. **Menu**: It is used to create all kinds of menus used by the application.  
   The general syntax is:

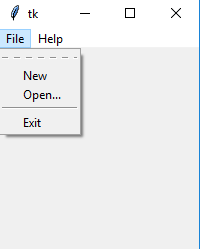
w = Menu(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **title**: To set the title of the widget.
  + **activebackground**: to set the background color when widget is under the cursor.
  + **activeforeground**: to set the foreground color when widget is under the cursor.
  + **bg**: to set he normal background color.
  + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.

|  |
| --- |
| from tkinter import \*    root = Tk()  menu = Menu(root)  root.config(menu=menu)  filemenu = Menu(menu)  menu.add\_cascade(label='File', menu=filemenu)  filemenu.add\_command(label='New')  filemenu.add\_command(label='Open...')  filemenu.add\_separator()  filemenu.add\_command(label='Exit', command=root.quit)  helpmenu = Menu(menu)  menu.add\_cascade(label='Help', menu=helpmenu)  helpmenu.add\_command(label='About')  mainloop() |

Output:  


1. **Message**: It refers to the multi-line and non-editable text. It works same as that of Label.  
   The general syntax is:

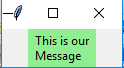
w = Message(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bd**: to set the border around the indicator.
  + **bg**: to set he normal background color.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*  main = Tk()  ourMessage ='This is our Message'  messageVar = Message(main, text = ourMessage)  messageVar.config(bg='lightgreen')  messageVar.pack( )  main.mainloop( ) |

Output:  


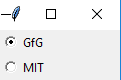
1. **RadioButton:**It is used to offer multi-choice option to the user. It offers several options to the user and the user has to choose one option.  
   The general syntax is:

w = RadioButton(master, option=value)

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **activebackground**: to set the background color when widget is under the cursor.
  + **activeforeground**: to set the foreground color when widget is under the cursor.
  + **bg**: to set he normal background color.
  + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the label in characters.
  + **height**: to set the height of the label in characters.

|  |
| --- |
| from tkinter import \*  root = Tk()  v = IntVar()  Radiobutton(root, text='GfG', variable=v, value=1).pack(anchor=W)  Radiobutton(root, text='MIT', variable=v, value=2).pack(anchor=W)  mainloop() |

Output:  


1. **Scale:**It is used to provide a graphical slider that allows to select any value from that scale. The general syntax is:

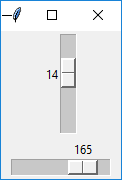
w = Scale(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **cursor**: To change the cursor pattern when the mouse is over the widget.
  + **activebackground**: To set the background of the widget when mouse is over the widget.
  + **bg**: to set he normal background color.
  + **orient**: Set it to HORIZONTAL or VERTICAL according to the requirement.
  + **from\_**: To set the value of one end of the scale range.
  + **to**: To set the value of the other end of the scale range.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.

|  |
| --- |
| from tkinter import \*  master = Tk()  w = Scale(master, from\_=0, to=42)  w.pack()  w = Scale(master, from\_=0, to=200, orient=HORIZONTAL)  w.pack()  mainloop() |

Output:  


1. **Scrollbar**: It refers to the slide controller which will be used to implement listed widgets.  
   The general syntax is:

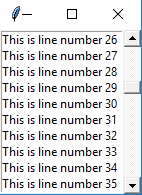
w = Scrollbar(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **width**: to set the width of the widget.
  + **activebackground**: To set the background when mouse is over the widget.
  + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.

|  |
| --- |
| from tkinter import \*  root = Tk()  scrollbar = Scrollbar(root)  scrollbar.pack( side = RIGHT, fill = Y )  mylist = Listbox(root, yscrollcommand = scrollbar.set )  for line in range(100):     mylist.insert(END, 'This is line number' + str(line))  mylist.pack( side = LEFT, fill = BOTH )  scrollbar.config( command = mylist.yview )  mainloop() |

Output:  


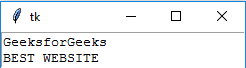
1. **Text:**To edit a multi-line text and format the way it has to be displayed.  
   The general syntax is:

w =Text(master, option=value)

There are number of options which are used to change the format of the text. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
  + **insertbackground**: To set the background of the widget.
  + **bg**: to set he normal background color.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*  root = Tk()  T = Text(root, height=2, width=30)  T.pack()  T.insert(END, 'GeeksforGeeks\nBEST WEBSITE\n')  mainloop() |

Output:  


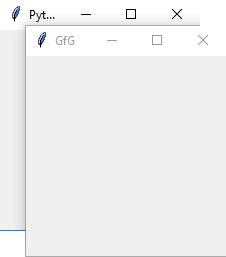
1. **TopLevel:**This widget is directly controlled by the window manager. It don’t need any parent window to work on.The general syntax is:

w = TopLevel(master, option=value)

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*  root = Tk()  root.title('GfG')  top = Toplevel()  top.title('Python')  top.mainloop() |

Output:  


1. **SpinBox:**It is an entry of ‘Entry’ widget. Here, value can be input by selecting a fixed value of numbers.The general syntax is:

w = SpinBox(master, option=value)

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.
  + **command**: To call a function.
  + **width**: to set the width of the widget.
  + **activebackground**: To set the background when mouse is over the widget.
  + **disabledbackground**: To disable the background when mouse is over the widget.
  + **from\_**: To set the value of one end of the range.
  + **to**: To set the value of the other end of the range.

|  |
| --- |
| from tkinter import \*  master = Tk()  w = Spinbox(master, from\_ = 0, to = 10)  w.pack()  mainloop() |

Output:  
http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Screenshot-68-16.png

1. **PannedWindow**It is a container widget which is used to handle number of panes arranged in it. The general syntax is:

w = PannedWindow(master, option=value)

master is the parameter used to represent the parent window.  
There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

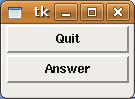
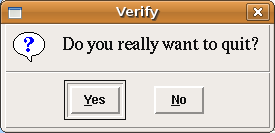
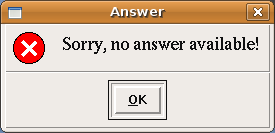
|  |
| --- |
| from tkinter import \*  m1 = PanedWindow()  m1.pack(fill = BOTH, expand = 1)  left = Entry(m1, bd = 5)  m1.add(left)  m2 = PanedWindow(m1, orient = VERTICAL)  m1.add(m2)  top = Scale( m2, orient = HORIZONTAL)  m2.add(top)  mainloop() |

Output:  


### The Options in Detail

|  |  |
| --- | --- |
| **Option** | **Meaning** |
| Anchor | The position, where the text should be placed in the message widget: N, NE, E, SE, S, SW, W, NW, or CENTER. The Default is CENTER. |
| Aspect | Aspect ratio, given as the width/height relation in percent. The default is 150, which means that the message will be 50% wider than it is high. Note that if the width is explicitly set, this option is ignored. |
| Background | The background color of the message widget. The default value is system specific. |
| Bg | Short for background. |
| Borderwidth | Border width. Default value is 2. |
| Bd | Short for borderwidth. |
| Cursor | Defines the kind of cursor to show when the mouse is moved over the message widget. By default the standard cursor is used. |
| Font | Message font. The default value is system specific. |
| Foreground | Text color. The default value is system specific. |
| Fg | Same as foreground. |
| highlightbackground | Together with highlightcolor and highlightthickness, this option controls how to draw the highlight region. |
| Highlightcolor | See highlightbackground. |
| Highlightthickness | See highlightbackground. |
| Justify | Defines how to align multiple lines of text. Use LEFT, RIGHT, or CENTER. Note that to position the text inside the widget, use the anchor option. Default is LEFT. |
| Padx | Horizontal padding. Default is -1 (no padding). |
| Pady | Vertical padding. Default is -1 (no padding). |
| Relief | Border decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE. |
| Takefocus | If true, the widget accepts input focus. The default is false. |
| Text | Message text. The widget inserts line breaks if necessary to get the requested aspect ratio. (text/Text) |
| Textvariable | Associates a Tkinter variable with the message, which is usually a StringVar. If the variable is changed, the message text is updated. |
| Width | Widget width given in character units. A suitable width based on the aspect setting is automatically chosen, if this option is not given. |

## Dialogues and Message Boxes

Tkinter (and TK of course) provides a set of dialogues (dialogs in American English spelling), which can be used to display message boxes, showing warning or errors, or widgets to select files and colours. There are also simple dialogues, asking the user to enter string, integers or float numbers.   
  
Let's look at a typical GUI Session with Dialogues and Message boxes. There might be a button starting the dialogue, like the "quit" button in the following window:   
  
   
  
Pushing the "quit" button raises the Verify window:   
  
   
  
Let's assume that we want to warn users that the "quit" functionality is not yet implemented. In this case we can use the warning message to inform the user, if he or she pushes the "yes" button:   
  
   
  
If somebody types the "No" button, the "Cancel" message box is raised:   
  
   
  
Let's go back to our first Dialogue with the "quit" and "answer" buttons. If the "Answer" functionality is not implemented, it might be useful to use the following error message box:   
  
   
  
  
Python script, which implements the previous dialogue widges: 

from Tkinter import \*

from tkMessageBox import \*

def answer():

showerror("Answer", "Sorry, no answer available")

def callback():

if askyesno('Verify', 'Really quit?'):

showwarning('Yes', 'Not yet implemented')

else:

showinfo('No', 'Quit has been cancelled')

Button(text='Quit', command=callback).pack(fill=X)

Button(text='Answer', command=answer).pack(fill=X)

mainloop()

### Message Boxes

The message dialogues are provided by the tkMessageBox module.   
  
The tkMessageBox consists of the following functions, which correspond to dialog windows:

* askokcancel(title=None, message=None, \*\*options)  
  Ask if operation should proceed; return true if the answer is ok
* askquestion(title=None, message=None, \*\*options)  
  Ask a question
* askretrycancel(title=None, message=None, \*\*options)  
  Ask if operation should be retried; return true if the answer is yes
* askyesno(title=None, message=None, \*\*options)  
  Ask a question; return true if the answer is yes
* askyesnocancel(title=None, message=None, \*\*options)  
  Ask a question; return true if the answer is yes, None if cancelled.
* showerror(title=None, message=None, \*\*options)  
  Show an error message
* showinfo(title=None, message=None, \*\*options)  
  Show an info message
* showwarning(title=None, message=None, \*\*options)  
  Show a warning message

### Open File Dialogue

There is hardly any serious application, which doesn't need a way to read from a file or write to a file. Furthermore, such an application might have to choose a directory. Tkinter provides the module tkFileDialog for these purposes.

from Tkinter import \*

from tkFileDialog import askopenfilename

def callback():

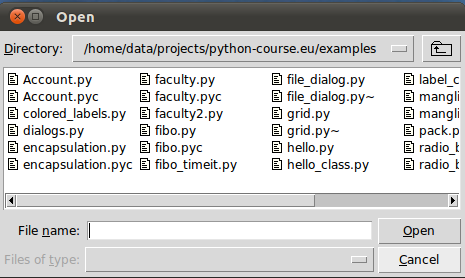
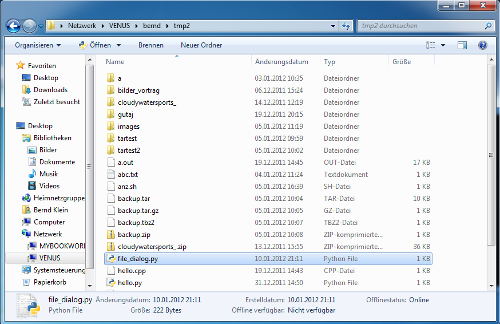
name= askopenfilename()

print name

errmsg = 'Error!'

Button(text='File Open', command=callback).pack(fill=X)

mainloop()

The code above creates a window with a single button with the text "File Open". If the button is pushed, the following window appears:   
  
   
  
The look-and-feel of the file-open-dialog depends on the GUI of the operating system. The above example was created using a gnome desktop under Linux. If we start the same program under Windows 7, it looks like this:   
  


### Choosing a Colour

There are applications where the user should have the possibility to select a colour. Tkinter provides a pop-up menu to choose a colour. To this purpose we have to import the tkColorChooser module and have to use the method askColor:

result = tkColorChooser.askColor ( color, option=value, ...)

If the user clicks the OK button on the pop-up window, respectively, the return value of askColor() is a tuple with two elements, both a representation of the chosen colour, e.g. ((106, 150, 98), '#6a9662')   
The first element return[0] is a tuple (R, G, B) with the RGB representation in decimal values (from 0 to 255). The second element return[1] is a hexadecimal representation of the chosen colour.   
If the user clicks "Cancel" the method returns the tuple (None, None).   
  
The optional keyword parameters are:

|  |  |
| --- | --- |
| Color | The variable color is used to set the default colour to be displayed. If color is not set, the initial colour will be grey. |
| Title | The text assigned to the variable title will appear in the pop-up window's title area. The default title is "Color". |
| parent | Make the pop-up window appear over window W. The default behaviour is that it appears over the root window. |

Let's have a look at an example:

from Tkinter import \*

from tkColorChooser import askcolor

def callback():

result = askcolor(color="#6A9662",

title = "Bernd's Colour Chooser")

print result

root = Tk()

Button(root,

text='Choose Color',

fg="darkgreen",

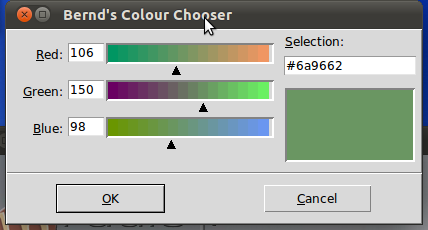
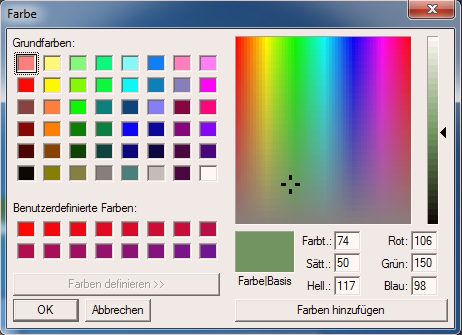
command=callback).pack(side=LEFT, padx=10)

Button(text='Quit',

command=root.quit,

fg="red").pack(side=LEFT, padx=10)

mainloop()

The look and feel depends on the operating system (e.g. Linux or Windows) and the chosen GUI (GNOME, KDE and so on). The following windows appear, if you use Gnome:   
  
Choosing a Colour Startmenu   
  
   
Using the same script under Windows 7 gives us the following result:   


## Layout Managers / Geometry Manager

In this chapter of our Python-Tkinter tutorial we will introduce the layout managers or geometry managers, as they are sometimes called as well. Tkinter possess three layout managers:

* pack
* grid
* place

The three layout managers pack, grid, and place should never be mixed in the same master window! Geometry managers serve various functions. They:

* arrange widgets on the screen
* register widgets with the underlying windowing system
* manage the display of widgets on the screen

Arranging widgets on the screen includes determining the size and position of components. Widgets can provide size and alignment information to geometry managers, but the geometry managers has always the final say on the positioning and sizing.

### Pack

Pack is the easiest to use of the three geometry managers of Tk and Tkinter. Instead of having to declare precisely where a widget should appear on the display screen, we can declare the positions of widgets with the pack command relative to each other. The pack command takes care of the details. Though the pack command is easier to use, this layout managers is limited in its possibilities compared to the grid and place mangers. For simple applications it is definitely the manager of choice. For example simple applications like placing a number of widgets side by side, or on top of each other.   
  
Example: 

from Tkinter import \*

root = Tk()

Label(root, text="Red Sun", bg="red", fg="white").pack()

Label(root, text="Green Grass", bg="green", fg="black").pack()

Label(root, text="Blue Sky", bg="blue", fg="white").pack()

mainloop()

#### fill Option

In our example, we have packed three labels into the parent widget "root". We used pack() without any options. So pack had to decide which way to arrange the labels. As you can see, it has chosen to place the label widgets on top of each other and centre them. Furthermore, we can see that each label has been given the size of the text. If you want to make the widgets as wide as the parent widget, you have to use the fill=X option:   
  
from Tkinter import \*

root = Tk()

w = Label(root, text="Red Sun", bg="red", fg="white")

w.pack(fill=X)

w = Label(root, text="Green Grass", bg="green", fg="black")

w.pack(fill=X)

w = Label(root, text="Blue Sky", bg="blue", fg="white")

w.pack(fill=X)

mainloop()

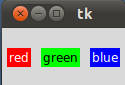
#### Padding

The pack() manager knows four padding options, i.e. internal and external padding and padding in x and y direction: 

|  |  |
| --- | --- |
| Padx | External padding, horizontally   Packing labels with the option padx   The code for the window above:   from Tkinter import \*  root = Tk()  w = Label(root, text="Red Sun", bg="red", fg="white")  w.pack(fill=X,padx=10)  w = Label(root, text="Green Grass", bg="green", fg="black")  w.pack(fill=X,padx=10)  w = Label(root, text="Blue Sky", bg="blue", fg="white")  w.pack(fill=X,padx=10)  mainloop() |
| Pady | External padding, vertically  Packing labels with the option padx   The code for the window above:   from Tkinter import \*  root = Tk()  w = Label(root, text="Red Sun", bg="red", fg="white")  w.pack(fill=X,pady=10)  w = Label(root, text="Green Grass", bg="green", fg="black")  w.pack(fill=X,pady=10)  w = Label(root, text="Blue Sky", bg="blue", fg="white")  w.pack(fill=X,pady=10)  mainloop() |
| Ipadx | Internal padding, horizontally.   In the following example, we change only the label with the text "Green Grass", so that the result can be easier recognized. We have also taken out the fill option.   Packing labels using ipadx   from Tkinter import \*  root = Tk()  w = Label(root, text="Red Sun", bg="red", fg="white")  w.pack()  w = Label(root, text="Green Grass", bg="green", fg="black")  w.pack(ipadx=10)  w = Label(root, text="Blue Sky", bg="blue", fg="white")  w.pack()  mainloop() |
| ipady | Internal padding, vertically   We will change the last label of our previous example to ipady=10.   Packing labels using ipadx   from Tkinter import \*  root = Tk()  w = Label(root, text="Red Sun", bg="red", fg="white")  w.pack()  w = Label(root, text="Green Grass", bg="green", fg="black")  w.pack(ipadx=10)  w = Label(root, text="Blue Sky", bg="blue", fg="white")  w.pack(ipady=10)  mainloop() |
|  |  |

The default value in all cases is 0. 

#### Placing widgets side by side

We want to place the three label side by side now and shorten the text slightly:   
  
   
  
The corresponding code looks like this: 

from Tkinter import \*

root = Tk()

w = Label(root, text="red", bg="red", fg="white")

w.pack(padx=5, pady=10, side=LEFT)

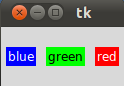
w = Label(root, text="green", bg="green", fg="black")

w.pack(padx=5, pady=20, side=LEFT)

w = Label(root, text="blue", bg="blue", fg="white")

w.pack(padx=5, pady=20, side=LEFT)

mainloop()

If we change LEFT to RIGHT in the previous example, we get the colours in reverse order:   
  
 

### Place Geometry Manager

The Place geometry manager allows you explicitly set the position and size of a window, either in absolute terms, or relative to another window. The place manager can be accessed through the place method. It can be applied to all standard widgets.   
  
We use the place geometry manager in the following example. We are playing around with colours in this example, i.e. we assign to every label a different colour, which we randomly create using the randrange method of the random module. We calculate the brightness (grey value) of each colour. If the brightness is less than 120, we set the foreground colour (fg) of the label to White otherwise to black, so that the text can be easier read. 

import Tkinter as tk

import random

root = tk.Tk()

# width x height + x\_offset + y\_offset:

root.geometry("170x200+30+30")

languages = ['Python','Perl','C++','Java','Tcl/Tk']

labels = range(5)

for i in range(5):

ct = [random.randrange(256) for x in range(3)]

brightness = int(round(0.299\*ct[0] + 0.587\*ct[1] + 0.114\*ct[2]))

ct\_hex = "%02x%02x%02x" % tuple(ct)

bg\_colour = '#' + "".join(ct\_hex)

l = tk.Label(root,

text=languages[i],

fg='White' if brightness < 120 else 'Black',

bg=bg\_colour)

l.place(x = 20, y = 30 + i\*30, width=120, height=25)

root.mainloop()

### Grid Manager

The first geometry manager of Tk had been pack. The algorithmic behaviour of pack is not easy to understand and it can be difficult to change an existing design. Grid was introduced in 1996 as an alternative to pack. Though grid is easier to learn and to use and produces nicer layouts, lots of developers keep using pack.   
  
Grid is in many cases the best choice for general use. While pack is sometimes not sufficient for changing details in the layout, place gives you complete control of positioning each element, but this makes it a lot more complex than pack and grid.   
  
The Grid geometry manager places the widgets in a 2-dimensional table, which consists of a number of rows and columns. The position of a widget is defined by a row and a column number. Widgets with the same column number and different row numbers will be above or below each other. Correspondingly, widgets with the same row number but different column numbers will be on the same "line" and will be beside of each other, i.e. to the left or the right.   
  
Using the grid manager means that you create a widget, and use the grid method to tell the manager in which row and column to place them. The size of the grid doesn't have to be defined, because the manager automatically determines the best dimensions for the widgets used.

#### Example with grid

from Tkinter import \*

colours = ['red','green','orange','white','yellow','blue']

r = 0

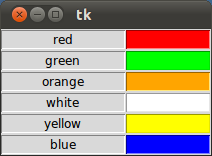
for c in colours:

Label(text=c, relief=RIDGE,width=15).grid(row=r,column=0)

Entry(bg=c, relief=SUNKEN,width=10).grid(row=r,column=1)

r = r + 1

mainloop()



**Python - GUI Programming (Tkinter)**

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. We would look this option in this chapter.
* **wxPython** − This is an open-source Python interface for wxWindows [http://wxpython.org](http://wxpython.org/).
* **JPython** − JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine [http://www.jython.org](http://www.jython.org/).

There are many other interfaces available, which you can find them on the net.

## 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. All you need to do is perform the following steps −

* 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.

## Example

#!/usr/bin/python

import tkinter

top = tkinter.Tk()

# Code to add widgets will go here...

top.mainloop()

This would create a following window −



Python offers multiple options for developing GUI (Graphical User Interface). Out of all the GUI methods, tkinter is most commonly used method. It is a standard Python interface to the Tk GUI toolkit shipped with Python. Python with tkinter outputs the fastest and easiest way to create the GUI applications. Creating a GUI using tkinter is an easy task.

## Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table −

|  |  |
| --- | --- |
| **Sr.No.** | **Operator & Description** |
| 1 | [**Button**](https://www.tutorialspoint.com/python/tk_button.htm)  The Button widget is used to display buttons in your application. |
| 2 | [**Canvas**](https://www.tutorialspoint.com/python/tk_canvas.htm)  The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application. |
| 3 | [**Checkbutton**](https://www.tutorialspoint.com/python/tk_checkbutton.htm)  The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time. |
| 4 | [**Entry**](https://www.tutorialspoint.com/python/tk_entry.htm)  The Entry widget is used to display a single-line text field for accepting values from a user. |
| 5 | [**Frame**](https://www.tutorialspoint.com/python/tk_frame.htm)  The Frame widget is used as a container widget to organize other widgets. |
| 6 | [**Label**](https://www.tutorialspoint.com/python/tk_label.htm)  The Label widget is used to provide a single-line caption for other widgets. It can also contain images. |
| 7 | [**Listbox**](https://www.tutorialspoint.com/python/tk_listbox.htm)  The Listbox widget is used to provide a list of options to a user. |
| 8 | [**Menubutton**](https://www.tutorialspoint.com/python/tk_menubutton.htm)  The Menubutton widget is used to display menus in your application. |
| 9 | [**Menu**](https://www.tutorialspoint.com/python/tk_menu.htm)  The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton. |
| 10 | [**Message**](https://www.tutorialspoint.com/python/tk_message.htm)  The Message widget is used to display multiline text fields for accepting values from a user. |
| 11 | [**Radiobutton**](https://www.tutorialspoint.com/python/tk_radiobutton.htm)  The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time. |
| 12 | [**Scale**](https://www.tutorialspoint.com/python/tk_scale.htm)  The Scale widget is used to provide a slider widget. |
| 13 | [**Scrollbar**](https://www.tutorialspoint.com/python/tk_scrollbar.htm)  The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes. |
| 14 | [**Text**](https://www.tutorialspoint.com/python/tk_text.htm)  The Text widget is used to display text in multiple lines. |
| 15 | [**Toplevel**](https://www.tutorialspoint.com/python/tk_toplevel.htm)  The Toplevel widget is used to provide a separate window container. |
| 16 | [**Spinbox**](https://www.tutorialspoint.com/python/tk_spinbox.htm)  The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values. |
| 17 | [**PanedWindow**](https://www.tutorialspoint.com/python/tk_panedwindow.htm)  A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically. |
| 18 | [**LabelFrame**](https://www.tutorialspoint.com/python/tk_labelframe.htm)  A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. |
| 19 | [**tkMessageBox**](https://www.tutorialspoint.com/python/tk_messagebox.htm)  This module is used to display message boxes in your applications. |

Let us study these widgets in detail −

## Standard attributes

Let us take a look at how some of their common attributes.such as sizes, colors and fonts are specified.

* [Dimensions](https://www.tutorialspoint.com/python/tk_dimensions.htm)
* [Colors](https://www.tutorialspoint.com/python/tk_colors.htm)
* [Fonts](https://www.tutorialspoint.com/python/tk_fonts.htm)
* [Anchors](https://www.tutorialspoint.com/python/tk_anchors.htm)
* [Relief styles](https://www.tutorialspoint.com/python/tk_relief.htm)
* [Bitmaps](https://www.tutorialspoint.com/python/tk_bitmaps.htm)
* [Cursors](https://www.tutorialspoint.com/python/tk_cursors.htm)

Let us study them briefly −

## Geometry Management

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](https://www.tutorialspoint.com/python/tk_pack.htm) − This geometry manager organizes widgets in blocks before placing them in the parent widget.
* [The *grid()* Method](https://www.tutorialspoint.com/python/tk_grid.htm) − This geometry manager organizes widgets in a table-like structure in the parent widget.
* [The *place()* Method](https://www.tutorialspoint.com/python/tk_place.htm) − This geometry manager organizes widgets by placing them in a specific position in the parent widget.

**To create a tkinter:**

1. Importing the module – tkinter
2. Create the main window (container)
3. Add any number of widgets to the main window
4. Apply the event Trigger on the widgets.

Importing tkinter is same as importing any other module in the python code. Note that the name of the module in Python 2.x is ‘Tkinter’ and in Python 3.x is ‘tkinter’.

import tkinter

There are two main methods used you the user need to remember while creating the Python application with GUI.

1. **Tk(screenName=None,  baseName=None,  className=’Tk’,  useTk=1):** To create a main window, tkinter offers a method ‘Tk(screenName=None,  baseName=None,  className=’Tk’,  useTk=1)’. To change the name of the window, you can change the className to the desired one. The basic code used to create the main window of the application is:

m=tkinter.Tk() where m is the name of the main window object

1. **mainloop():** There is a method known by the name mainloop() is used when you are ready for the application to run. mainloop() is an infinite loop used to run the application, wait for an event to occur and process the event till the window is not closed.

m.mainloop()

|  |
| --- |
| import tkinter  m = tkinter.Tk()  '''  widgets are added here  '''  m.mainloop() |

tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

1. **pack() method:**It organizes the widgets in blocks before placing in the parent widget.
2. **grid() method:**It organizes the widgets in grid (table-like structure) before placing in the parent widget.
3. **place() method:**It organizes the widgets by placing them on specific positions directed by the programmer.

There are a number of widgets which you can put in your tkinter application. Some of the major widgets are explained below:

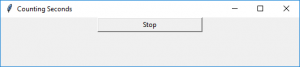
1. **Button**:To add a button in your application, this widget is used.  
   The general syntax is:

w=Button(master, option=value)

master is the parameter used to represent the parent window.  
There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **activebackground**: to set the background color when button is under the cursor.
  + **activeforeground**: to set the foreground color when button is under the cursor.
  + **bg**: to set he normal background color.
  + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the button.
  + **width**: to set the width of the button.
  + **height**: to set the height of the button.

|  |
| --- |
| import tkinter as tk  r = tk.Tk()  r.title('Counting Seconds')  button = tk.Button(r, text='Stop', width=25, command=r.destroy)  button.pack()  r.mainloop() |

Output:  


1. **Canvas:**It is used to draw pictures and other complex layout like graphics, text and widgets.  
   The general syntax is:

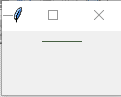
w = Canvas(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bd**: to set the border width in pixels.
  + **bg**: to set the normal background color.
  + **cursor**: to set the cursor used in the canvas.
  + **highlightcolor**: to set the color shown in the focus highlight.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*  master = Tk()  w = Canvas(master, width=40, height=60)  w.pack()  canvas\_height=20  canvas\_width=200  y = int(canvas\_height / 2)  w.create\_line(0, y, canvas\_width, y )  mainloop() |

Output:  


1. **CheckButton:**To select any number of options by displaying a number of options to a user as toggle buttons. The general syntax is:

w = CheckButton(master, option=value)

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **Title**: To set the title of the widget.
  + **activebackground**: to set the background color when widget is under the cursor.
  + **activeforeground**: to set the foreground color when widget is under the cursor.
  + **bg**: to set he normal backgrouSteganography

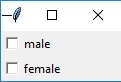
Break

Secret Code:

Attach a File:nd color.

* + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.

|  |
| --- |
| from tkinter import \*  master = Tk()  var1 = IntVar()  Checkbutton(master, text='male', variable=var1).grid(row=0, sticky=W)  var2 = IntVar()  Checkbutton(master, text='female', variable=var2).grid(row=1, sticky=W)  mainloop() |

Output:  


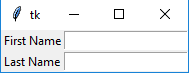
1. **Entry:**It is used to input the single line text entry from the user.. For multi-line text input, Text widget is used.  
   The general syntax is:

w=Entry(master, option=value)

master is the parameter used to represent the parent window.  
There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bd**: to set the border width in pixels.
  + **bg**: to set the normal background color.
  + **cursor**: to set the cursor used.
  + **command**: to call a function.
  + **highlightcolor**: to set the color shown in the focus highlight.
  + **width**: to set the width of the button.
  + **height**: to set the height of the button.

|  |
| --- |
| from tkinter import \*  master = Tk()  Label(master, text='First Name').grid(row=0)  Label(master, text='Last Name').grid(row=1)  e1 = Entry(master)  e2 = Entry(master)  e1.grid(row=0, column=1)  e2.grid(row=1, column=1)  mainloop() |

Output:  


1. **Frame:** It acts as a container to hold the widgets. It is used for grouping and organizing the widgets. The general syntax is:

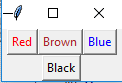
w = Frame(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
  + **bd**: to set the border width in pixels.
  + **bg**: to set the normal background color.
  + **cursor**: to set the cursor used.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*    root = Tk()  frame = Frame(root)  frame.pack()  bottomframe = Frame(root)  bottomframe.pack( side = BOTTOM )  redbutton = Button(frame, text = 'Red', fg ='red')  redbutton.pack( side = LEFT)  greenbutton = Button(frame, text = 'Brown', fg='brown')  greenbutton.pack( side = LEFT )  bluebutton = Button(frame, text ='Blue', fg ='blue')  bluebutton.pack( side = LEFT )  blackbutton = Button(bottomframe, text ='Black', fg ='black')  blackbutton.pack( side = BOTTOM)  root.mainloop() |

Output:  


1. **Label**: It refers to the display box where you can put any text or image which can be updated any time as per the code.  
   The general syntax is:

w=Label(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bg**: to set he normal background color.
  + **bg** to set he normal background color.
  + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the button.
  + **width**: to set the width of the button.
  + **height**” to set the height of the button.

|  |
| --- |
| from tkinter import \*  root = Tk()  w = Label(root, text='GeeksForGeeks.org!')  w.pack()  root.mainloop() |

Output:  
http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Screenshot-68-5.png

1. **Listbox**: It offers a list to the user from which the user can accept any number of options.  
   The general syntax is:

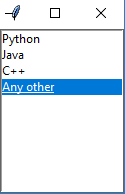
w = Listbox(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
  + **bg**: to set he normal background color.
  + **bd**: to set the border width in pixels.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*    top = Tk()  Lb = Listbox(top)  Lb.insert(1, 'Python')  Lb.insert(2, 'Java')  Lb.insert(3, 'C++')  Lb.insert(4, 'Any other')  Lb.pack()  top.mainloop() |

Output:  


1. **MenuButton**: It is a part of top-down menu which stays on the window all the time. Every menubutton has its own functionality. The general syntax is:

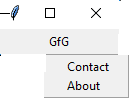
w = MenuButton(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **activebackground**: To set the background when mouse is over the widget.
  + **activeforeground**: To set the foreground when mouse is over the widget.
  + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.
  + **highlightcolor**: To set the color of the focus highlight when widget has to be focused.

|  |
| --- |
| from tkinter import \*    top = Tk()  mb =  Menubutton ( top, text = &quot;GfG&quot;)  mb.grid()  mb.menu  =  Menu ( mb, tearoff = 0 )  mb[&quot;menu&quot;]  =  mb.menu  cVar  = IntVar()  aVar = IntVar()  mb.menu.add\_checkbutton ( label ='Contact', variable = cVar )  mb.menu.add\_checkbutton ( label = 'About', variable = aVar )  mb.pack()  top.mainloop() |

Output:  


1. **Menu**: It is used to create all kinds of menus used by the application.  
   The general syntax is:

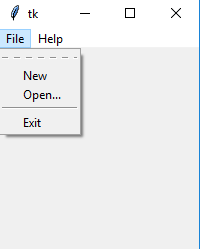
w = Menu(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **title**: To set the title of the widget.
  + **activebackground**: to set the background color when widget is under the cursor.
  + **activeforeground**: to set the foreground color when widget is under the cursor.
  + **bg**: to set he normal background color.
  + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.

|  |
| --- |
| from tkinter import \*    root = Tk()  menu = Menu(root)  root.config(menu=menu)  filemenu = Menu(menu)  menu.add\_cascade(label='File', menu=filemenu)  filemenu.add\_command(label='New')  filemenu.add\_command(label='Open...')  filemenu.add\_separator()  filemenu.add\_command(label='Exit', command=root.quit)  helpmenu = Menu(menu)  menu.add\_cascade(label='Help', menu=helpmenu)  helpmenu.add\_command(label='About')  mainloop() |

Output:  


1. **Message**: It refers to the multi-line and non-editable text. It works same as that of Label.  
   The general syntax is:

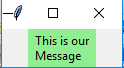
w = Message(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bd**: to set the border around the indicator.
  + **bg**: to set he normal background color.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*  main = Tk()  ourMessage ='This is our Message'  messageVar = Message(main, text = ourMessage)  messageVar.config(bg='lightgreen')  messageVar.pack( )  main.mainloop( ) |

Output:  


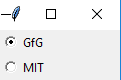
1. **RadioButton:**It is used to offer multi-choice option to the user. It offers several options to the user and the user has to choose one option.  
   The general syntax is:

w = RadioButton(master, option=value)

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **activebackground**: to set the background color when widget is under the cursor.
  + **activeforeground**: to set the foreground color when widget is under the cursor.
  + **bg**: to set he normal background color.
  + **command**: to call a function.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the label in characters.
  + **height**: to set the height of the label in characters.

|  |
| --- |
| from tkinter import \*  root = Tk()  v = IntVar()  Radiobutton(root, text='GfG', variable=v, value=1).pack(anchor=W)  Radiobutton(root, text='MIT', variable=v, value=2).pack(anchor=W)  mainloop() |

Output:  


1. **Scale:**It is used to provide a graphical slider that allows to select any value from that scale. The general syntax is:

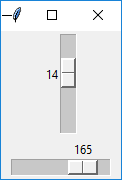
w = Scale(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **cursor**: To change the cursor pattern when the mouse is over the widget.
  + **activebackground**: To set the background of the widget when mouse is over the widget.
  + **bg**: to set he normal background color.
  + **orient**: Set it to HORIZONTAL or VERTICAL according to the requirement.
  + **from\_**: To set the value of one end of the scale range.
  + **to**: To set the value of the other end of the scale range.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.

|  |
| --- |
| from tkinter import \*  master = Tk()  w = Scale(master, from\_=0, to=42)  w.pack()  w = Scale(master, from\_=0, to=200, orient=HORIZONTAL)  w.pack()  mainloop() |

Output:  


1. **Scrollbar**: It refers to the slide controller which will be used to implement listed widgets.  
   The general syntax is:

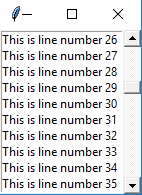
w = Scrollbar(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **width**: to set the width of the widget.
  + **activebackground**: To set the background when mouse is over the widget.
  + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.

|  |
| --- |
| from tkinter import \*  root = Tk()  scrollbar = Scrollbar(root)  scrollbar.pack( side = RIGHT, fill = Y )  mylist = Listbox(root, yscrollcommand = scrollbar.set )  for line in range(100):     mylist.insert(END, 'This is line number' + str(line))  mylist.pack( side = LEFT, fill = BOTH )  scrollbar.config( command = mylist.yview )  mainloop() |

Output:  


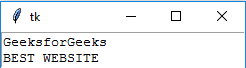
1. **Text:**To edit a multi-line text and format the way it has to be displayed.  
   The general syntax is:

w =Text(master, option=value)

There are number of options which are used to change the format of the text. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
  + **insertbackground**: To set the background of the widget.
  + **bg**: to set he normal background color.
  + **font**: to set the font on the button label.
  + **image**: to set the image on the widget.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*  root = Tk()  T = Text(root, height=2, width=30)  T.pack()  T.insert(END, 'GeeksforGeeks\nBEST WEBSITE\n')  mainloop() |

Output:  


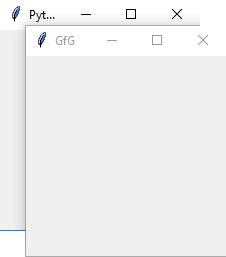
1. **TopLevel:**This widget is directly controlled by the window manager. It don’t need any parent window to work on.The general syntax is:

w = TopLevel(master, option=value)

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

|  |
| --- |
| from tkinter import \*  root = Tk()  root.title('GfG')  top = Toplevel()  top.title('Python')  top.mainloop() |

Output:  


1. **SpinBox:**It is an entry of ‘Entry’ widget. Here, value can be input by selecting a fixed value of numbers.The general syntax is:

w = SpinBox(master, option=value)

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.
  + **command**: To call a function.
  + **width**: to set the width of the widget.
  + **activebackground**: To set the background when mouse is over the widget.
  + **disabledbackground**: To disable the background when mouse is over the widget.
  + **from\_**: To set the value of one end of the range.
  + **to**: To set the value of the other end of the range.

|  |
| --- |
| from tkinter import \*  master = Tk()  w = Spinbox(master, from\_ = 0, to = 10)  w.pack()  mainloop() |

Output:  
http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Screenshot-68-16.png

1. **PannedWindow**It is a container widget which is used to handle number of panes arranged in it. The general syntax is:

w = PannedWindow(master, option=value)

master is the parameter used to represent the parent window.  
There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

* + **bg**: to set he normal background color.
  + **bd**: to set the size of border around the indicator.
  + **cursor**: To appear the cursor when the mouse over the menubutton.
  + **width**: to set the width of the widget.
  + **height**: to set the height of the widget.

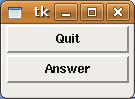
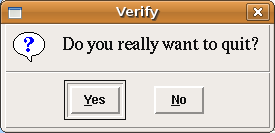
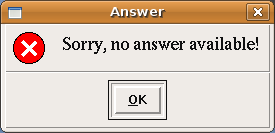
|  |
| --- |
| from tkinter import \*  m1 = PanedWindow()  m1.pack(fill = BOTH, expand = 1)  left = Entry(m1, bd = 5)  m1.add(left)  m2 = PanedWindow(m1, orient = VERTICAL)  m1.add(m2)  top = Scale( m2, orient = HORIZONTAL)  m2.add(top)  mainloop() |

Output:  


### The Options in Detail

|  |  |
| --- | --- |
| **Option** | **Meaning** |
| Anchor | The position, where the text should be placed in the message widget: N, NE, E, SE, S, SW, W, NW, or CENTER. The Default is CENTER. |
| Aspect | Aspect ratio, given as the width/height relation in percent. The default is 150, which means that the message will be 50% wider than it is high. Note that if the width is explicitly set, this option is ignored. |
| Background | The background color of the message widget. The default value is system specific. |
| Bg | Short for background. |
| Borderwidth | Border width. Default value is 2. |
| Bd | Short for borderwidth. |
| Cursor | Defines the kind of cursor to show when the mouse is moved over the message widget. By default the standard cursor is used. |
| Font | Message font. The default value is system specific. |
| Foreground | Text color. The default value is system specific. |
| Fg | Same as foreground. |
| highlightbackground | Together with highlightcolor and highlightthickness, this option controls how to draw the highlight region. |
| Highlightcolor | See highlightbackground. |
| Highlightthickness | See highlightbackground. |
| Justify | Defines how to align multiple lines of text. Use LEFT, RIGHT, or CENTER. Note that to position the text inside the widget, use the anchor option. Default is LEFT. |
| Padx | Horizontal padding. Default is -1 (no padding). |
| Pady | Vertical padding. Default is -1 (no padding). |
| Relief | Border decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE. |
| Takefocus | If true, the widget accepts input focus. The default is false. |
| Text | Message text. The widget inserts line breaks if necessary to get the requested aspect ratio. (text/Text) |
| Textvariable | Associates a Tkinter variable with the message, which is usually a StringVar. If the variable is changed, the message text is updated. |
| Width | Widget width given in character units. A suitable width based on the aspect setting is automatically chosen, if this option is not given. |

## Dialogues and Message Boxes

Tkinter (and TK of course) provides a set of dialogues (dialogs in American English spelling), which can be used to display message boxes, showing warning or errors, or widgets to select files and colours. There are also simple dialogues, asking the user to enter string, integers or float numbers.   
  
Let's look at a typical GUI Session with Dialogues and Message boxes. There might be a button starting the dialogue, like the "quit" button in the following window:   
  
   
  
Pushing the "quit" button raises the Verify window:   
  
   
  
Let's assume that we want to warn users that the "quit" functionality is not yet implemented. In this case we can use the warning message to inform the user, if he or she pushes the "yes" button:   
  
   
  
If somebody types the "No" button, the "Cancel" message box is raised:   
  
   
  
Let's go back to our first Dialogue with the "quit" and "answer" buttons. If the "Answer" functionality is not implemented, it might be useful to use the following error message box:   
  
   
  
  
Python script, which implements the previous dialogue widges: 

from Tkinter import \*

from tkMessageBox import \*

def answer():

showerror("Answer", "Sorry, no answer available")

def callback():

if askyesno('Verify', 'Really quit?'):

showwarning('Yes', 'Not yet implemented')

else:

showinfo('No', 'Quit has been cancelled')

Button(text='Quit', command=callback).pack(fill=X)

Button(text='Answer', command=answer).pack(fill=X)

mainloop()

### Message Boxes

The message dialogues are provided by the tkMessageBox module.   
  
The tkMessageBox consists of the following functions, which correspond to dialog windows:

* askokcancel(title=None, message=None, \*\*options)  
  Ask if operation should proceed; return true if the answer is ok
* askquestion(title=None, message=None, \*\*options)  
  Ask a question
* askretrycancel(title=None, message=None, \*\*options)  
  Ask if operation should be retried; return true if the answer is yes
* askyesno(title=None, message=None, \*\*options)  
  Ask a question; return true if the answer is yes
* askyesnocancel(title=None, message=None, \*\*options)  
  Ask a question; return true if the answer is yes, None if cancelled.
* showerror(title=None, message=None, \*\*options)  
  Show an error message
* showinfo(title=None, message=None, \*\*options)  
  Show an info message
* showwarning(title=None, message=None, \*\*options)  
  Show a warning message

### Open File Dialogue

There is hardly any serious application, which doesn't need a way to read from a file or write to a file. Furthermore, such an application might have to choose a directory. Tkinter provides the module tkFileDialog for these purposes.

from Tkinter import \*

from tkFileDialog import askopenfilename

def callback():

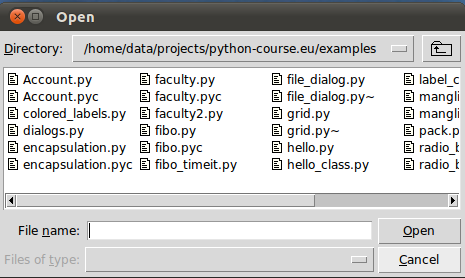
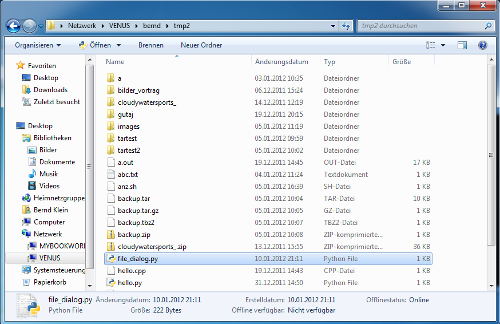
name= askopenfilename()

print name

errmsg = 'Error!'

Button(text='File Open', command=callback).pack(fill=X)

mainloop()

The code above creates a window with a single button with the text "File Open". If the button is pushed, the following window appears:   
  
   
  
The look-and-feel of the file-open-dialog depends on the GUI of the operating system. The above example was created using a gnome desktop under Linux. If we start the same program under Windows 7, it looks like this:   
  


### Choosing a Colour

There are applications where the user should have the possibility to select a colour. Tkinter provides a pop-up menu to choose a colour. To this purpose we have to import the tkColorChooser module and have to use the method askColor:

result = tkColorChooser.askColor ( color, option=value, ...)

If the user clicks the OK button on the pop-up window, respectively, the return value of askColor() is a tuple with two elements, both a representation of the chosen colour, e.g. ((106, 150, 98), '#6a9662')   
The first element return[0] is a tuple (R, G, B) with the RGB representation in decimal values (from 0 to 255). The second element return[1] is a hexadecimal representation of the chosen colour.   
If the user clicks "Cancel" the method returns the tuple (None, None).   
  
The optional keyword parameters are:

|  |  |
| --- | --- |
| Color | The variable color is used to set the default colour to be displayed. If color is not set, the initial colour will be grey. |
| Title | The text assigned to the variable title will appear in the pop-up window's title area. The default title is "Color". |
| parent | Make the pop-up window appear over window W. The default behaviour is that it appears over the root window. |

Let's have a look at an example:

from Tkinter import \*

from tkColorChooser import askcolor

def callback():

result = askcolor(color="#6A9662",

title = "Bernd's Colour Chooser")

print result

root = Tk()

Button(root,

text='Choose Color',

fg="darkgreen",

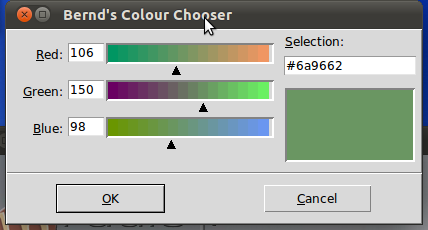
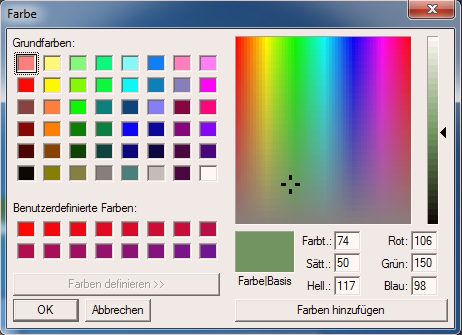
command=callback).pack(side=LEFT, padx=10)

Button(text='Quit',

command=root.quit,

fg="red").pack(side=LEFT, padx=10)

mainloop()

The look and feel depends on the operating system (e.g. Linux or Windows) and the chosen GUI (GNOME, KDE and so on). The following windows appear, if you use Gnome:   
  
Choosing a Colour Startmenu   
  
   
Using the same script under Windows 7 gives us the following result:   


## Layout Managers / Geometry Manager

In this chapter of our Python-Tkinter tutorial we will introduce the layout managers or geometry managers, as they are sometimes called as well. Tkinter possess three layout managers:

* pack
* grid
* place

The three layout managers pack, grid, and place should never be mixed in the same master window! Geometry managers serve various functions. They:

* arrange widgets on the screen
* register widgets with the underlying windowing system
* manage the display of widgets on the screen

Arranging widgets on the screen includes determining the size and position of components. Widgets can provide size and alignment information to geometry managers, but the geometry managers has always the final say on the positioning and sizing.

### Pack

Pack is the easiest to use of the three geometry managers of Tk and Tkinter. Instead of having to declare precisely where a widget should appear on the display screen, we can declare the positions of widgets with the pack command relative to each other. The pack command takes care of the details. Though the pack command is easier to use, this layout managers is limited in its possibilities compared to the grid and place mangers. For simple applications it is definitely the manager of choice. For example simple applications like placing a number of widgets side by side, or on top of each other.   
  
Example: 

from Tkinter import \*

root = Tk()

Label(root, text="Red Sun", bg="red", fg="white").pack()

Label(root, text="Green Grass", bg="green", fg="black").pack()

Label(root, text="Blue Sky", bg="blue", fg="white").pack()

mainloop()

#### fill Option

In our example, we have packed three labels into the parent widget "root". We used pack() without any options. So pack had to decide which way to arrange the labels. As you can see, it has chosen to place the label widgets on top of each other and centre them. Furthermore, we can see that each label has been given the size of the text. If you want to make the widgets as wide as the parent widget, you have to use the fill=X option:   
  
from Tkinter import \*

root = Tk()

w = Label(root, text="Red Sun", bg="red", fg="white")

w.pack(fill=X)

w = Label(root, text="Green Grass", bg="green", fg="black")

w.pack(fill=X)

w = Label(root, text="Blue Sky", bg="blue", fg="white")

w.pack(fill=X)

mainloop()

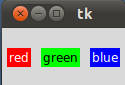
#### Padding

The pack() manager knows four padding options, i.e. internal and external padding and padding in x and y direction: 

|  |  |
| --- | --- |
| Padx | External padding, horizontally   Packing labels with the option padx   The code for the window above:   from Tkinter import \*  root = Tk()  w = Label(root, text="Red Sun", bg="red", fg="white")  w.pack(fill=X,padx=10)  w = Label(root, text="Green Grass", bg="green", fg="black")  w.pack(fill=X,padx=10)  w = Label(root, text="Blue Sky", bg="blue", fg="white")  w.pack(fill=X,padx=10)  mainloop() |
| Pady | External padding, vertically  Packing labels with the option padx   The code for the window above:   from Tkinter import \*  root = Tk()  w = Label(root, text="Red Sun", bg="red", fg="white")  w.pack(fill=X,pady=10)  w = Label(root, text="Green Grass", bg="green", fg="black")  w.pack(fill=X,pady=10)  w = Label(root, text="Blue Sky", bg="blue", fg="white")  w.pack(fill=X,pady=10)  mainloop() |
| Ipadx | Internal padding, horizontally.   In the following example, we change only the label with the text "Green Grass", so that the result can be easier recognized. We have also taken out the fill option.   Packing labels using ipadx   from Tkinter import \*  root = Tk()  w = Label(root, text="Red Sun", bg="red", fg="white")  w.pack()  w = Label(root, text="Green Grass", bg="green", fg="black")  w.pack(ipadx=10)  w = Label(root, text="Blue Sky", bg="blue", fg="white")  w.pack()  mainloop() |
| ipady | Internal padding, vertically   We will change the last label of our previous example to ipady=10.   Packing labels using ipadx   from Tkinter import \*  root = Tk()  w = Label(root, text="Red Sun", bg="red", fg="white")  w.pack()  w = Label(root, text="Green Grass", bg="green", fg="black")  w.pack(ipadx=10)  w = Label(root, text="Blue Sky", bg="blue", fg="white")  w.pack(ipady=10)  mainloop() |
|  |  |

The default value in all cases is 0. 

#### Placing widgets side by side

We want to place the three label side by side now and shorten the text slightly:   
  
   
  
The corresponding code looks like this: 

from Tkinter import \*

root = Tk()

w = Label(root, text="red", bg="red", fg="white")

w.pack(padx=5, pady=10, side=LEFT)

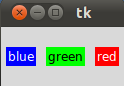
w = Label(root, text="green", bg="green", fg="black")

w.pack(padx=5, pady=20, side=LEFT)

w = Label(root, text="blue", bg="blue", fg="white")

w.pack(padx=5, pady=20, side=LEFT)

mainloop()

If we change LEFT to RIGHT in the previous example, we get the colours in reverse order:   
  
 

### Place Geometry Manager

The Place geometry manager allows you explicitly set the position and size of a window, either in absolute terms, or relative to another window. The place manager can be accessed through the place method. It can be applied to all standard widgets.   
  
We use the place geometry manager in the following example. We are playing around with colours in this example, i.e. we assign to every label a different colour, which we randomly create using the randrange method of the random module. We calculate the brightness (grey value) of each colour. If the brightness is less than 120, we set the foreground colour (fg) of the label to White otherwise to black, so that the text can be easier read.   
  
import Tkinter as tk

import random

root = tk.Tk()

# width x height + x\_offset + y\_offset:

root.geometry("170x200+30+30")

languages = ['Python','Perl','C++','Java','Tcl/Tk']

labels = range(5)

for i in range(5):

ct = [random.randrange(256) for x in range(3)]

brightness = int(round(0.299\*ct[0] + 0.587\*ct[1] + 0.114\*ct[2]))

ct\_hex = "%02x%02x%02x" % tuple(ct)

bg\_colour = '#' + "".join(ct\_hex)

l = tk.Label(root,

text=languages[i],

fg='White' if brightness < 120 else 'Black',

bg=bg\_colour)

l.place(x = 20, y = 30 + i\*30, width=120, height=25)

root.mainloop()

### Grid Manager

The first geometry manager of Tk had been pack. The algorithmic behaviour of pack is not easy to understand and it can be difficult to change an existing design. Grid was introduced in 1996 as an alternative to pack. Though grid is easier to learn and to use and produces nicer layouts, lots of developers keep using pack.   
  
Grid is in many cases the best choice for general use. While pack is sometimes not sufficient for changing details in the layout, place gives you complete control of positioning each element, but this makes it a lot more complex than pack and grid.   
  
The Grid geometry manager places the widgets in a 2-dimensional table, which consists of a number of rows and columns. The position of a widget is defined by a row and a column number. Widgets with the same column number and different row numbers will be above or below each other. Correspondingly, widgets with the same row number but different column numbers will be on the same "line" and will be beside of each other, i.e. to the left or the right.   
  
Using the grid manager means that you create a widget, and use the grid method to tell the manager in which row and column to place them. The size of the grid doesn't have to be defined, because the manager automatically determines the best dimensions for the widgets used.

#### Example with grid

from Tkinter import \*

colours = ['red','green','orange','white','yellow','blue']

r = 0

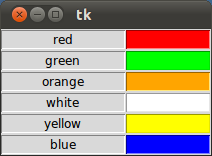
for c in colours:

Label(text=c, relief=RIDGE,width=15).grid(row=r,column=0)

Entry(bg=c, relief=SUNKEN,width=10).grid(row=r,column=1)

r = r + 1

mainloop()



**Turtle Programming in Python**

**turtle module**

“Turtle” is a Python feature like a drawing board, which lets us command a turtle to draw all over it! We can use functions like turtle.forward(…) and turtle.right(…) which can move the turtle around.Commonly used turtle methods are :

| METHOD | PARAMETER | DESCRIPTION |
| --- | --- | --- |
| Turtle() | None | Creates and returns a new turtle object |
| forward() | amount | Moves the turtle forward by the specified amount |
| backward() | amount | Moves the turtle backward by the specified amount |
| right() | angle | Turns the turtle clockwise |
| left() | angle | Turns the turtle counter clockwise |
| penup() or up() | None | Picks up the turtle’s Pen |
| pendown() or down() | None | Puts down the turtle’s Pen |
| color() | Color name | Changes the color of the turtle’s pen |
| fillcolor() | Color name | Changes the color of the turtle will use to fill a polygon |
| heading() | None | Returns the current heading |
| position() | None | Returns the current position |
| goto() | x, y | Move the turtle to position x,y |
| begin\_fill() | None | Remember the starting point for a filled polygon |
| end\_fill() | None | Close the polygon and fill with the current fill color |
| dot() | None | Leave the dot at the current position |
| stamp() | None | Leaves an impression of a turtle shape at the current location |
| shape() | shapename | Should be ‘arrow’, ‘classic’, ‘turtle’ or ‘circle’ |

To make use of the turtle methods and functionalities, we need to import turtle.”turtle” comes packed with the standard Python package and need not be installed externally.

The roadmap for executing a turtle program follows 4 steps:

1. Import the turtle module
2. Create a turtle to control.
3. Draw around using the turtle methods.
4. Run turtle.done().

So as stated above, before we can use turtle, we need to import it.We import it as :

from turtle import \*

# or

import turtle

After importing the turtle library and making all the turtle functionalities available to us, we need to create a new drawing board(window) and a turtle. Let’s call the window as wn and the turtle as skk. So we code as:

wn = turtle.Screen()

wn.bgcolor("light green")

wn.title("Turtle")

skk = turtle.Turtle()

Now that we have created the window and the turtle, we need to move the turtle. To move forward 100 pixels in the direction skk is facing, we code:

skk.forward(100)

We have moved skk 100 pixels forward, Awesome! Now we complete the program with the done() function and We’re done!

turtle.done()

So, we have created a program that draws a line 100 pixels long. We can draw various shapes and fill different colors using turtle methods. There’s plethora of functions and programs to be coded using the turtle library in python. Let’s learn to draw some of the basic shapes.

EX1: **Shape 1: Square**

# Python program to draw square

# using Turtle Programming

import turtle

skk = turtle.Turtle()

for i in range(4):

    skk.forward(50)

    skk.right(90)

turtle.done()

EX2: **Shape 2: Star**

# Python program to draw star

# using Turtle Programming

import turtle

star = turtle.Turtle()

for i in range(50):

    star.forward(50)

    star.right(144)

turtle.done()

EX3: **Shape 3: Hexagon**

# Python program to draw hexagon

# using Turtle Programming

import turtle

polygon = turtle.Turtle()

num\_sides = 6

side\_length = 70

angle = 360.0 / num\_sides

for i in range(num\_sides):

    polygon.forward(side\_length)

    polygon.right(angle)

turtle.done()

EX4: **1. Spiral Square Outside In and Inside Out**

# Python program to draw

# Spiral Square Outside In and Inside Out

# using Turtle Programming

import turtle   #Outside\_In

wn = turtle.Screen()

wn.bgcolor("light green")

wn.title("Turtle")

skk = turtle.Turtle()

skk.color("blue")

def sqrfunc(size):

    for i in range(4):

        skk.fd(size)

        skk.left(90)

        size = size-5

sqrfunc(146)

sqrfunc(126)

sqrfunc(106)

sqrfunc(86)

sqrfunc(66)

sqrfunc(46)

sqrfunc(26)

EX5: **2. Spiral Square Outside In and Inside Out**

|  |
| --- |
| import turtle  #Inside\_Out  wn = turtle.Screen()  wn.bgcolor("light green")  skk = turtle.Turtle()  skk.color("blue")    def sqrfunc(size):      for i in range(4):          skk.fd(size)          skk.left(90)          size = size + 5    sqrfunc(6)  sqrfunc(26)  sqrfunc(46)  sqrfunc(66)  sqrfunc(86)  sqrfunc(106)  sqrfunc(126)  sqrfunc(146) |

**Output:**  
  
**EX6: User Input Pattern**

# Python program to user input pattern

# using Turtle Programming

import turtle   #Outside\_In

import turtle

import time

import random

print ("This program draws shapes based on the number you enter in a uniform pattern.")

num\_str = input("Enter the side number of the shape you want to draw: ")

if num\_str.isdigit():

    squares = int(num\_str)

angle = 180 - 180\*(squares-2)/squares

turtle.up

x = 0

y = 0

turtle.setpos(x, y)

numshapes = 8

for x in range(numshapes):

    turtle.color(random.random(), random.random(), random.random())

    x += 5

    y += 5

    turtle.forward(x)

    turtle.left(y)

    for i in range(squares):

        turtle.begin\_fill()

        turtle.down()

        turtle.forward(40)

        turtle.left(angle)

        turtle.forward(40)

        print (turtle.pos())

        turtle.up()

        turtle.end\_fill()

time.sleep(11)

turtle.bye()

EX7: **Spiral Helix Pattern**

# Python program to draw

# Spiral  Helix Pattern

# using Turtle Programming

import turtle

loadWindow = turtle.Screen()

turtle.speed(2)

for i in range(100):

    turtle.circle(5\*i)

    turtle.circle(-5\*i)

    turtle.left(i)

turtle.exitonclick()

EX8:**Rainbow Benzene**

# Python program to draw

# Rainbow Benzene

# using Turtle Programming

import turtle

colors = ['red', 'purple', 'blue', 'green', 'orange', 'yellow']

t = turtle.Pen()

turtle.bgcolor('black')

for x in range(360):

    t.pencolor(colors[x%6])

    t.width(x/100 + 1)

    t.forward(x)

    t.left(59)

Multithreading in Python

**Thread**

In computing, a **process** is an instance of a computer program that is being executed. Any process has 3 basic components:

* An executable program.
* The associated data needed by the program (variables, work space, buffers, etc.)
* The execution context of the program (State of process)

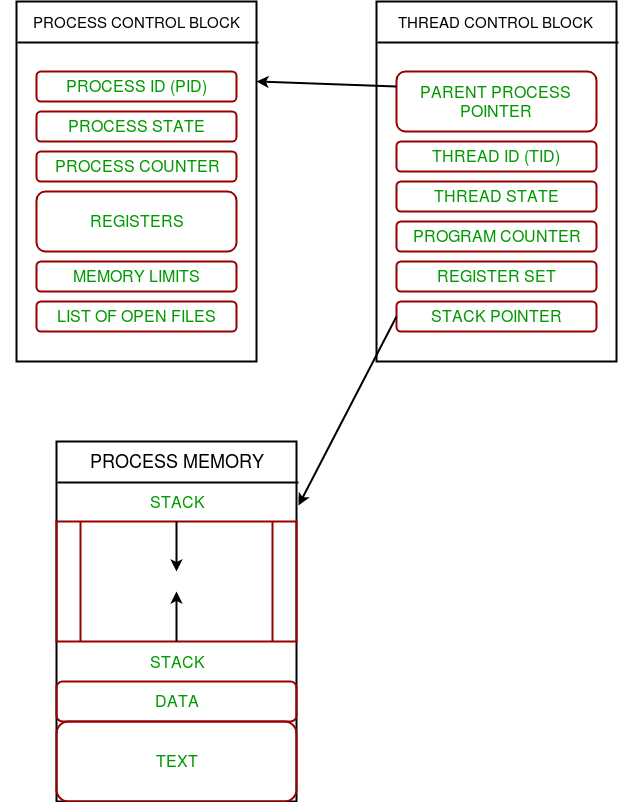
A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

In simple words, a **thread** is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!

A thread contains all this information in a **Thread Control Block (TCB)**:

* **Thread Identifier:** Unique id (TID) is assigned to every new thread
* **Stack pointer:** Points to thread’s stack in the process. Stack contains the local variables under thread’s scope.
* **Program counter:** a register which stores the address of the instruction currently being executed by thread.
* **Thread state:** can be running, ready, waiting, start or done.
* **Thread’s register set:** registers assigned to thread for computations.
* **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

Consider the diagram below to understand the relation between process and its thread:

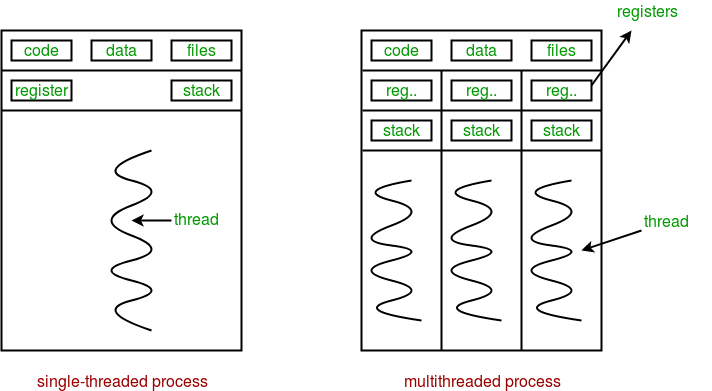


**Multithreading**

Multiple threads can exist within one process where:

* Each thread contains its own **register set** and **local variables (stored in stack)**.
* All thread of a process share **global variables (stored in heap)** and the **program code**.

Consider the diagram below to understand how multiple threads exist in memory:

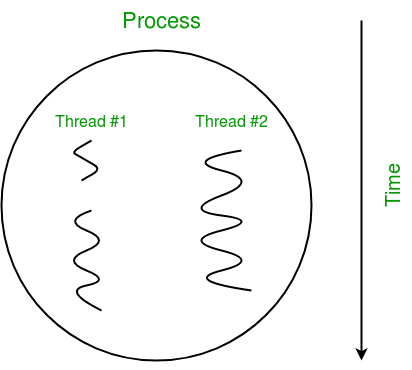


**Multithreading** is defined as the ability of a processor to execute multiple threads concurrently.

*In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed as****context switching****.*

*In context switching, the state of a thread is saved and state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallely (this is termed as****multitasking****).*

Consider the diagram below in which a process contains two active threads:



**Multithreading in Python**

In Python, the **threading** module provides a very simple and intuitive API for spawning multiple threads in a program.

Let us consider a simple example using **threading module**:

|  |
| --- |
| # Python program to illustrate the concept  # of threading  # importing the threading module  import threading    def print\_cube(num):      """      function to print cube of given num      """      print("Cube: {}".format(num \* num \* num))    def print\_square(num):      """      function to print square of given num      """      print("Square: {}".format(num \* num))    if \_\_name\_\_ == "\_\_main\_\_":      # creating thread      t1 = threading.Thread(target=print\_square, args=(10,))      t2 = threading.Thread(target=print\_cube, args=(10,))        # starting thread 1      t1.start()      # starting thread 2      t2.start()        # wait until thread 1 is completely executed      t1.join()      # wait until thread 2 is completely executed      t2.join()        # both threads completely executed      print("Done!") |

Square: 100

Cube: 1000

Done!

Let us try to understand the above code:

* To import the threading module, we do:
* import threading
* To create a new thread, we create an object of **Thread** class. It takes following arguments:
  + **target**: the function to be executed by thread
  + **args**: the arguments to be passed to the target function

In above example, we created 2 threads with different target functions:

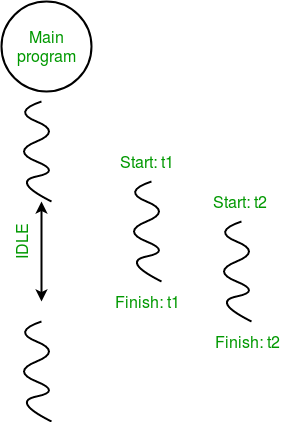
t1 = threading.Thread(target=print\_square, args=(10,))

t2 = threading.Thread(target=print\_cube, args=(10,))

* To start a thread, we use **start** method of **Thread** class.
* t1.start()
* t2.start()
* Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop execution of current program until a thread is complete, we use **join** method.
* t1.join()
* t2.join()

As a result, the current program will first wait for the completion of **t1** and then **t2**. Once, they are finished, the remaining statements of current program are executed.

Consider the diagram below for a better understanding of how above program works:



Consider the python program given below in which we print thread name and corresponding process for each task:(multi threading)

|  |
| --- |
| # Python program to illustrate the concept  # of threading  import threading  import os    def task1():      print("Task 1 assigned to thread: {}".format(threading.current\_thread().name))      print("ID of process running task 1: {}".format(os.getpid()))    def task2():      print("Task 2 assigned to thread: {}".format(threading.current\_thread().name))      print("ID of process running task 2: {}".format(os.getpid()))    if \_\_name\_\_ == "\_\_main\_\_":        # print ID of current process      print("ID of process running main program: {}".format(os.getpid()))        # print name of main thread      print("Main thread name: {}".format(threading.main\_thread().name))        # creating threads      t1 = threading.Thread(target=task1, name='t1')      t2 = threading.Thread(target=task2, name='t2')        # starting threads      t1.start()      t2.start()        # wait until all threads finish      t1.join()      t2.join() |

ID of process running main program: 11758

Main thread name: MainThread

Task 1 assigned to thread: t1

ID of process running task 1: 11758

Task 2 assigned to thread: t2

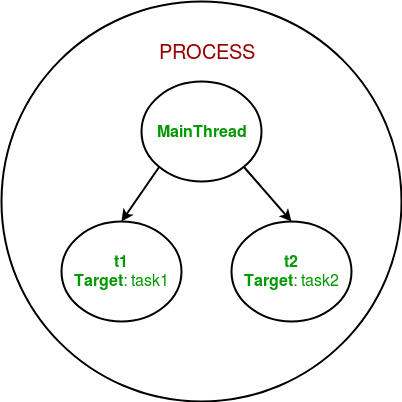
ID of process running task 2: 11758

Let us try to understand the above code:

* We use **os.getpid()** function to get ID of current process.
* print("ID of process running main program: {}".format(os.getpid()))

As it is clear from the output, the process ID remains same for all threads.

* We use **threading.main\_thread()** function to get the main thread object. In normal conditions, the main thread is the thread from which the Python interpreter was started. **name** attribute of thread object is used to get the name of thread.
* print("Main thread name: {}".format(threading.main\_thread().name))
* We use the **threading.current\_thread()** function to get the current thread object.
* print("Task 1 assigned to thread: {}".format(threading.current\_thread().name))

The diagram given below clears the above concept:  


So, this was a brief introduction to multithreading in Python.

**synchronization between multiple threads**.

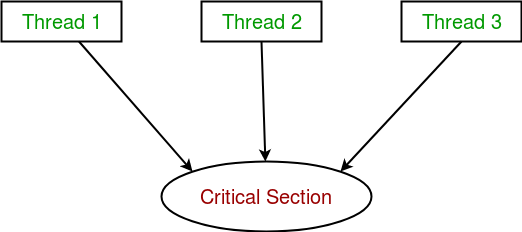
[Multithreading in Python (Synchronization)](https://www.geeksforgeeks.org/multithreading-in-python-set-2-synchronization/)

**Synchronization between threads**

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as **critical section**.

|  |  |  |
| --- | --- | --- |
| X=10  Reg1=10+1=11  Reg2=10+1=11  X=11  X=11 | T1  1.Reg1=x  2.Reg1=reg1+1  3.X=reg1 | T2  1.Reg2=x  2.Reg2=reg2+1  3.X=reg2 |

*Critical section refers to the parts of the program where the shared resource is accessed.*

For example, in the diagram below, 3 threads try to access shared resource or critical section at the same time.  


Concurrent accesses to shared resource can lead to **race condition**.

*A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.*

**Consider the program below to understand the concept of race condition:**

|  |
| --- |
| import threading    # global variable x  x = 0    def increment():      """      function to increment global variable x      """      global x      x += 1    def thread\_task():      """      task for thread      calls increment function 100000 times.      """      for \_ in range(100000):          increment()    def main\_task():      global x      # setting global variable x as 0      x = 0        # creating threads      t1 = threading.Thread(target=thread\_task)      t2 = threading.Thread(target=thread\_task)        # start threads      t1.start()      t2.start()        # wait until threads finish their job      t1.join()      t2.join()    if \_\_name\_\_ == "\_\_main\_\_":      for i in range(10):          main\_task()          print("Iteration {0}: x = {1}".format(i,x)) |

Output:

Iteration 0: x = 175005

Iteration 1: x = 200000

Iteration 2: x = 200000

Iteration 3: x = 169432

Iteration 4: x = 153316

Iteration 5: x = 200000

Iteration 6: x = 167322

Iteration 7: x = 200000

Iteration 8: x = 169917

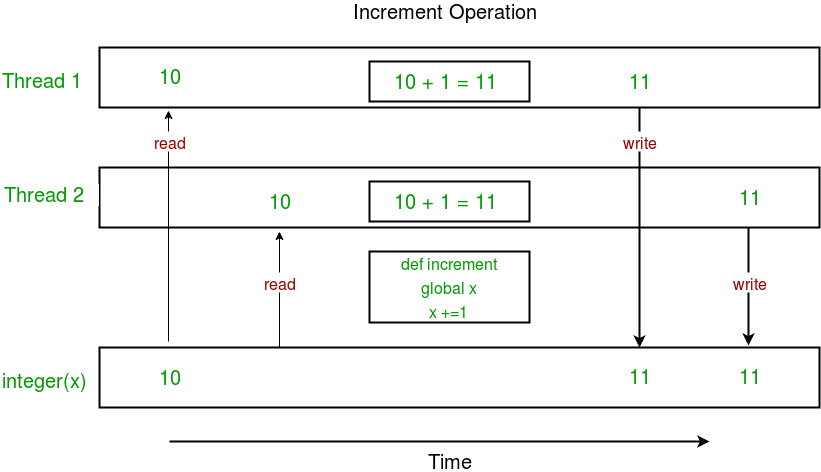
Iteration 9: x = 153589

In above program:

* Two threads **t1** and **t2** are created in **main\_task** function and global variable **x** is set to 0.
* Each thread has a target function **thread\_task** in which **increment** function is called 100000 times.
* **increment** function will increment the global variable **x** by 1 in each call.

The expected final value of **x** is 200000 but what we get in 10 iterations of **main\_task** function is some different values.

This happens due to concurrent access of threads to the shared variable **x**. This unpredictability in value of **x** is nothing but **race condition**.

Given below is a diagram which shows how can **race condition** occur in above program:  


Notice that expected value of **x** in above diagram is 12 but due to race condition, it turns out to be 11!  
  
Hence, we need a tool for proper synchronization between multiple threads.

**Using Locks**

**threading** module provides a **Lock** class to deal with the race conditions. Lock is implemented using a **Semaphore** object provided by the Operating System.

A semaphore is a synchronization object that controls access by multiple processes/threads to a common resource in a parallel programming environment.

It is simply a value in a designated place in operating system (or kernel) storage that each process/thread can check and then change. Depending on the value that is found, the process/thread can use the resource or will find that it is already in use and must wait for some period before trying again.

Semaphores can be binary (0 or 1) or can have additional values. Typically, a process/thread using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

**Lock** class provides following methods:

* **acquire([blocking]) :** To acquire a lock. A lock can be blocking or non-blocking.
  + When invoked with the blocking argument set to **True** (the default), thread execution is blocked until the lock is unlocked, then lock is set to locked and return **True**.
  + When invoked with the blocking argument set to **False**, thread execution is not blocked. If lock is unlocked, then set it to locked and return **True** else return **False** immediately.
* **release() :** To release a lock.
  + When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.
  + If lock is already unlocked, a **ThreadError** is raised.

Consider the example given below:( semaphore)

|  |
| --- |
| import threading    # global variable x  x = 0    def increment():      """      function to increment global variable x      """      global x      x += 1    def thread\_task(lock):      """      task for thread      calls increment function 100000 times.      """      for \_ in range(100000):          lock.acquire()          increment()          lock.release()    def main\_task():      global x      # setting global variable x as 0      x = 0        # creating a lock      lock = threading.Lock()        # creating threads      t1 = threading.Thread(target=thread\_task, args=(lock,))      t2 = threading.Thread(target=thread\_task, args=(lock,))        # start threads      t1.start()      t2.start()        # wait until threads finish their job      t1.join()      t2.join()    if \_\_name\_\_ == "\_\_main\_\_":      for i in range(10):          main\_task()          print("Iteration {0}: x = {1}".format(i,x)) |

Output:

Iteration 0: x = 200000

Iteration 1: x = 200000

Iteration 2: x = 200000

Iteration 3: x = 200000

Iteration 4: x = 200000

Iteration 5: x = 200000

Iteration 6: x = 200000

Iteration 7: x = 200000

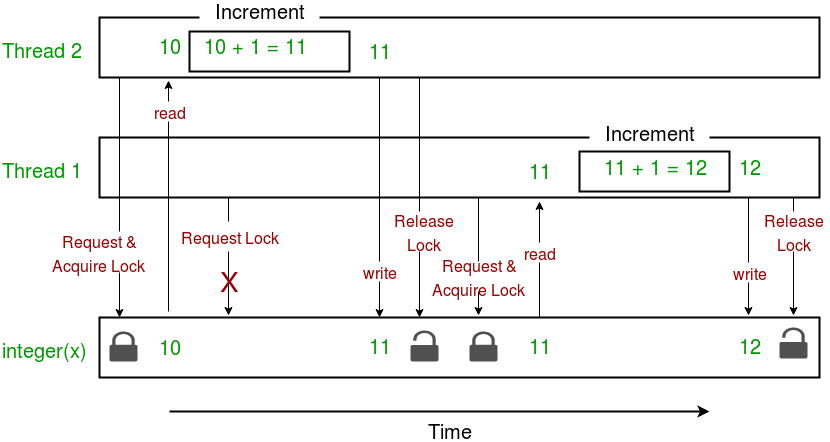
Iteration 8: x = 200000

Iteration 9: x = 200000

Let us try to understand the above code step by step:

* Firstly, a **Lock** object is created using:
* lock = threading.Lock()
* Then, **lock** is passed as target function argument:
* t1 = threading.Thread(target=thread\_task, args=(lock,))
* t2 = threading.Thread(target=thread\_task, args=(lock,))
* In the critical section of target function, we apply lock using **lock.acquire()** method. As soon as a lock is acquired, no other thread can access the critical section (here, **increment** function) until the lock is released using **lock.release()** method.
* lock.acquire()
* increment()
* lock.release()

As you can see in the results, the final value of **x** comes out to be 200000 every time (which is the expected final result).

Here is a diagram given below which depicts the implementation of locks in above program:  


This brings us to the end of **Multithreading in Python**.

Finally, here are a few advantages and disadvantages of multithreading:

**Advantages:**

* It doesn’t block the user. This is because threads are independent of each other.
* Better use of system resources is possible since threads execute tasks parallely.
* Enhanced performance on multi-processor machines.
* Multi-threaded servers and interactive GUIs use multithreading exclusively.

**Disadvantages:**

* As number of threads increase, complexity increases.
* Synchronization of shared resources (objects, data) is necessary.
* It is difficult to debug, result is sometimes unpredictable.
* Potential deadlocks which leads to starvation, i.e. some threads may not be served with a bad design
* Constructing and synchronizing threads is CPU/memory intensive.

## Programming on Threading

## 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 thread

import time

# 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 1:

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 Queue

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

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

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

self.threadID = threadID

self.name = name

self.q = q

def run(self):

print "Starting " + self.name

process\_data(self.name, self.q)

print "Exiting " + self.name

def process\_data(threadName, q):

while not exitFlag:

queueLock.acquire()

if not workQueue.empty():

data = q.get()

queueLock.release()

print "%s processing %s" % (threadName, data)

else:

queueLock.release()

time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]

nameList = ["One", "Two", "Three", "Four", "Five"]

queueLock = threading.Lock()

workQueue = Queue.Queue(10)

threads = []

threadID = 1

# Create new threads

for tName in threadList:

thread = myThread(threadID, tName, workQueue)

thread.start()

threads.append(thread)

threadID += 1

# Fill the queue

queueLock.acquire()

for word in nameList:

workQueue.put(word)

queueLock.release()

# Wait for queue to empty

while not workQueue.empty():

pass

# Notify threads it's time to exit

exitFlag = 1

# 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

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

**Use of matplotlib:**

# **Matplotlib (Python Plotting Library)**

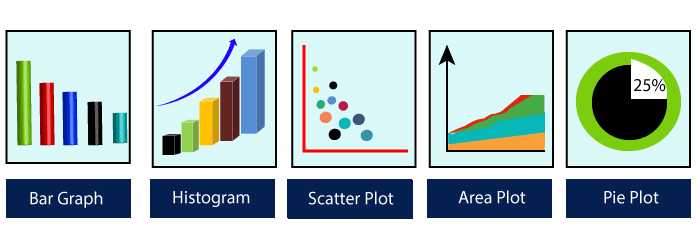
Human minds are more adaptive for the visual representation of data rather than textual data. We can easily understand things when they are visualized.

## Data Visualization

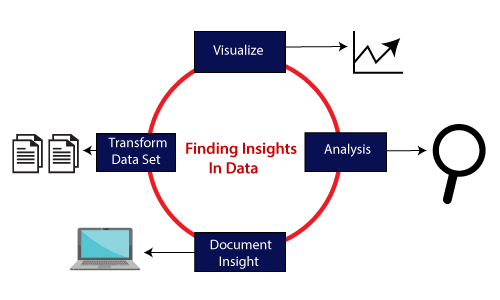
Data visualization is a new term. It expresses the idea that involves more than just representing data in the graphical form

This can be very helpful when discovering and getting to know a dataset and can help with classifying patterns, corrupt data, outliers, and much more.

There are five key plots that are used for data visualization.



There are five phases which are essential to make the decision for the organization:



**Visualize:** We analyze the raw data, which means it makes complex data more accessible, understandable, and more usable.

**Analysis:** Data analysis is defined as cleaning, inspecting, transforming, and modeling data to derive useful information.

**Document Insight:** Document insight is the process where the useful data or information is organized in the document in the standard format.

**Transform Data Set:** Standard data is used to make the decision more effectively.

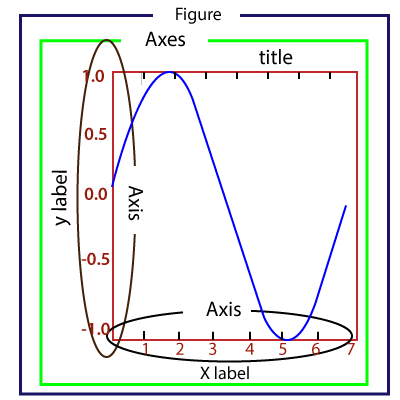
**Data visualization can perform below tasks:**

* It identifies areas that need improvement and attention.
* It clarifies the factors.
* It helps to understand which product to place where.
* Predict sales volumes.

**Matplotlib** is a Python library which is defined as a multi-platform data visualization library built on Numpy array. It can be used in python scripts, shell, web application, and other graphical user interface toolkit.

## The General Concept of Matplotlib

A Matplotlib figure can be categorized into various parts as below:



**Figure:** It is a whole figure which may hold one or more axes (plots). We can think of a Figure as a canvas that holds plots.

**Axes:** A Figure can contain several Axes. It consists of two or three (in the case of 3D) Axis objects. Each Axes is comprised of a title, an x-label, and a y-label.

**Axis:** Axises are the number of line like objects and responsible for generating the graph limits.

**Artist:** An artist is the all which we see on the graph like Text objects, Line2D objects, and collection objects. Most Artists are tied to Axes.

## Installing Matplotlib

In anaconda prompt:

conda install matplotlib

in command prompt:

pip install matplotlib

## Basic Example of plotting Graph

Here is the basic example of generating a simple graph; the program is following:

from matplotlib **import** pyplot as plt

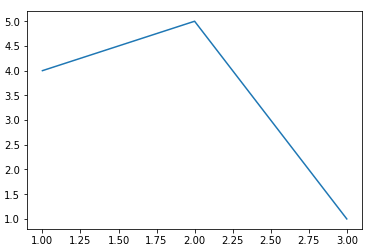
#ploting our canvas

plt.plot([1,2,3],[4,5,1])

#display the graph

plt.show()

**Output:**



e can add titles, labels to our chart which are created by Python matplotlib library to make it more meaningful. The example is the following:

from matplotlib **import** pyplot as plt

x = [5, 2, 7]

y = [1, 10, 4]

plt.plot(x, y)

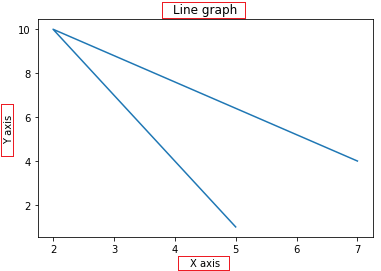
plt.title('Line graph')

plt.ylabel('Y axis')

plt.xlabel('X axis')

plt.show()

**Output:**



## Working with Pyplot

The pyplot functions are used to make some changes to figure such as create a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot including labels, etc.

We can pass the arbitrary number of arguments to the plot(). For example, to plot x versus y, we can do this following way:

from matplotlib **import** pyplot as plt

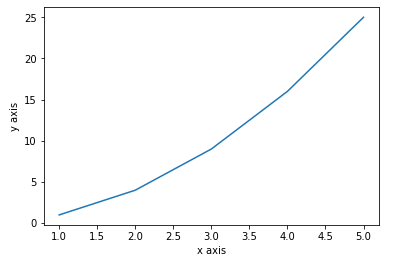
plt.plot([1,2,3,4,5],[1,4,9,16,25])

plt.ylabel("y axis")

plt.xlabel('x axis')

plt.show()

**Output:**



**Formatting the style of the plot**

**Example format String**

|  |  |
| --- | --- |
| **'b'** | Using for the blue marker with default shape. |
| **'ro'** | Red circle |
| **'-g'** | Green solid line |
| **'--'** | A dashed line with the default color |
| **'^k:'** | Black triangle up markers connected by a dotted line |

The matplotlib supports the following color abbreviation:

|  |  |
| --- | --- |
| **Character** | **Color** |
| **'b' Blue** |  |
| **'g'** | Green |
| **'r'** | Red |
| **'c'** | Cyan |
| **'m'** | Magenta |
| **'y'** | Yellow |
| **'k'** | Black |
| **'w'** | White |

### Plotting with categorical variables

Matplotlib allows us to pass categorical variables directly to many plotting functions: consider the following example

from matplotlib **import** pyplot

names = ['Abhishek', 'Himanshu', 'Devansh']

marks= [87,50,98]

plt.figure(figsize=(9,3))

plt.subplot(131)

plt.bar(names, marks)

plt.subplot(132)

plt.scatter(names, marks)

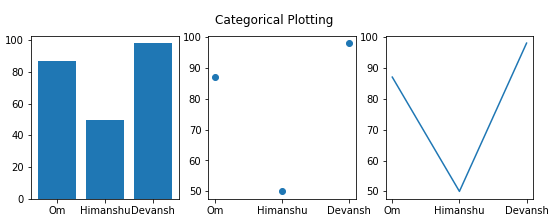
plt.subplot(133)

plt.plot(names, marks)

plt.suptitle('Categorical Plotting')

plt.show()

**Output:**



In the above program, we have plotted the categorical graph using the **subplot()** function. Let's a have a look on the subplot() function.

## What is subplot()

The Matplotlib **subplot()** function is defined as to plot two or more plots in one figure. We can use this method to separate two graphs which plotted in the same axis Matplotlib supports all kinds of subplots, including 2x1 vertical, 2x1 horizontal, or a 2x2 grid.

It accepts the three arguments: they are **nrows, ncols, and index**. It denote the number of rows, number of columns and the index.

The subplot() function can be called in the following way:

subplot(nrows,ncols,index,\*\*kwargs)

subplot(pos,\*\*kwargs)

subplot(ax)

**Parameters:**

* **\*args:**

Three separate integers or three-digit integer describes the position of the subplot. If the three integers are **nrows, ncols, and index** in order, the subplot will take the index position on a grid with **nrows row** and **ncol column**.

The argument **pos** are a three-digit integer, where the first digit is denoted the number of rows, the second digit denoted the number of columns, and the third represents the index of the subplot. **For example,** subplot (1, 3, 2) is the same as the subplot (132).

## Creating different types of graph

### 1. Line graph

The line graph is one of charts which shows information as a series of the line. The graph is plotted by the plot() function. The line graph is simple to plot; let's consider the following example:

from matplotlib **import** pyplot as plt

x = [4,8,9]

y = [10,12,15]

plt.plot(x,y)

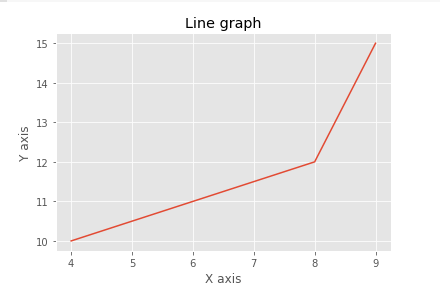
plt.title("Line graph")

plt.ylabel('Y axis')

plt.xlabel('X axis')

plt.show()

**Output:**



We can customize the graph by importing the style module. The style module will be built into a matplotlib installation. It contains the various functions to make the plot more attractive. In the below program, we are using the style module:

from matplotlib **import** pyplot as plt

from matplotlib **import** style

style.use('ggplot')

x = [16, 8, 10]

y = [8, 16, 6]

x2 = [8, 15, 11]

y2 = [6, 15, 7]

plt.plot(x, y, 'r', label='line one', linewidth=5)

plt.plot(x2, y2, 'm', label='line two', linewidth=5)

plt.title('Epic Info')

fig = plt.figure()

plt.ylabel('Y axis')

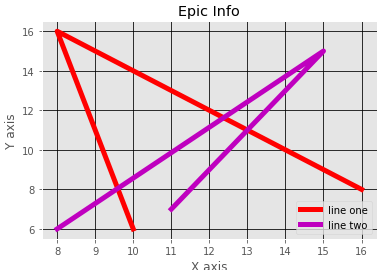
plt.xlabel('X axis')

plt.legend()

plt.grid(True, color='k')

plt.show()

**Output:**



In Matplotlib, the figure (an instance of class plt.Figure) can be supposed of as a single container that consists of all the objects denoting axes, graphics, text, and labels.

**Example-3**

**import** numpy as np

**import** matplotlib.pyplot as plt

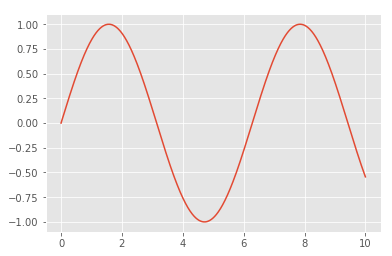
fig = plt.figure()

ax = plt.axes()

x = np.linspace(0, 10, 1000)

ax.plot(x, np.sin(x))

**Output:**



### Bar graphs

Matplotlib provides a **bar()** to make bar graphs which accepts arguments such as: categorical variables, their value and color.

from matplotlib **import** pyplot as plt

players = ['Virat','Rohit','Shikhar','Hardik']

runs = [51,87,45,67]

plt.bar(players,runs,color = 'green')

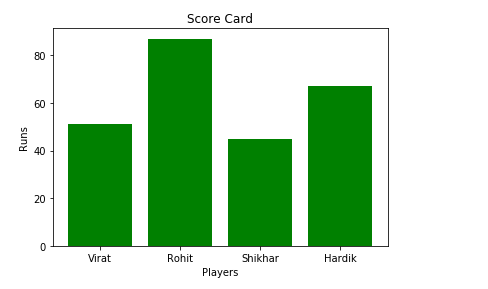
plt.title('Score Card')

plt.xlabel('Players')

plt.ylabel('Runs')

plt.show()

**Output:**



Another function **barh()** is used to make horizontal bar graphs. It accepts **xerr** or **yerr** as arguments (in case of vertical graphs) to depict the variance in our data as follows:

from matplotlib **import** pyplot as plt

players = ['Virat','Rohit','Shikhar','Hardik']

runs = [51,87,45,67]

plt.barh(players,runs, color = 'green')

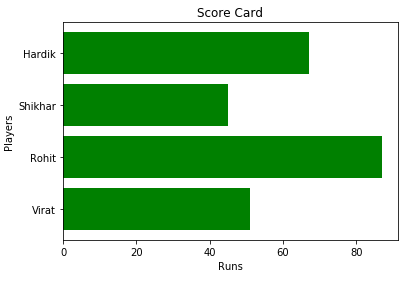
plt.title('Score Card')

plt.xlabel('Players')

plt.ylabel('Runs')

plt.show()

**Output:**



Let's have a look on the other example using the **style()** function:

from matplotlib **import** pyplot as plt

from matplotlib **import** style

style.use('ggplot')

x = [5,8,10]

y = [12,16,6]

x2 = [6,9,11]

y2 = [7,15,7]

plt.bar(x, y, color = 'y', align='center')

plt.bar(x2, y2, color='c', align='center')

plt.title('Information')

plt.ylabel('Y axis')

plt.xlabel('X axis')

**Output:**



### Pie Chart

 It is generally used to represent the percentage or proportional data where each slice of pie represents a particular category.

from matplotlib **import** pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:

Players = 'Rohit', 'Virat', 'Shikhar', 'Yuvraj'

Runs = [45, 30, 15, 10]

explode = (0.1, 0, 0, 0)  # it "explode" the 1st slice

fig1, ax1 = plt.subplots()

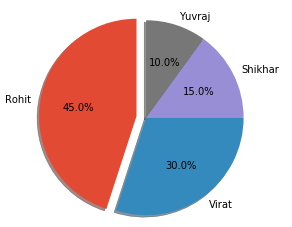
ax1.pie(Runs, explode=explode, labels=Players, autopct='%1.1f%%',

        shadow=True, startangle=90)

ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()

**Output:**



### Histogram

 A histogram is a type of bar plot that shows the frequency of a number of values compared to a set of values ranges.

**For example** we take the data of the different age group of the people and plot a histogram with respect to the bin. Now, bin represents the range of values that are divided into series of intervals. Bins are generally created of the same size.

from matplotlib **import** pyplot as plt

from matplotlib **import** pyplot as plt

population\_age = [21,53,60,49,25,27,30,42,40,1,2,102,95,8,15,105,70,65,55,70,75,60,52,44,43,42,45]

bins = [0,10,20,30,40,50,60,70,80,90,100]

plt.hist(population\_age, bins, histtype='bar', rwidth=0.8)

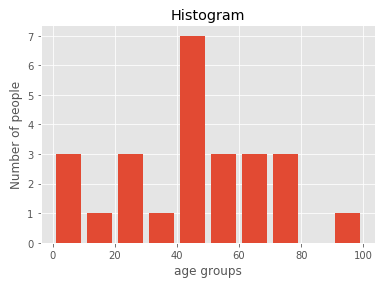
plt.xlabel('age groups')

plt.ylabel('Number of people')

plt.title('Histogram')

plt.show()

**Output:**



Network & Socket Programming

Server Client Programming