PYTHON PROGRAMMING

Presented By
POTU NARAYANA
Research Scholar
Osmania University
Dept. of CSE
B.Tech, M.Tech, M.A, PGDHR, MIAENG, MIS,(Ph.D)
Email : pnarayana@osmania.ac.in
Contact : +91 9704868721

UNIT 4

GUI Programming

•Here, we will give you a brief introduction to the subject of graphical user interface (GUI) programming. If you are somewhat new to this area or want to learn more about it, or if you want to see how it is done in Python, then this chapter is for you. We cannot show you everything about GUI application development here in this one chapter, but we will give you a very solid introduction to it.

•The primary GUI toolkit we will be using is Tk, Python's default GUI, and we will access Tk from its Python interface called Tkinter (short for "Tk interface").

What Are Tcl, Tk, and Tkinter?

- Tkinter is Python's default GUI library. It is based on the Tk toolkit, originally designed for the Tool Command Language (Tcl). Due to Tk's popularity, it has been ported to a variety of other scripting languages, including Perl (Perl/Tk), Ruby (Ruby/Tk), and Python (Tkinter). With the GUI development portability and flexibility of Tk, along with the simplicity of scripting language integrated with the power of systems language, you are given the tools to rapidly design and implement a wide variety of commercial-quality GUI applications

Getting Tkinter Installed and Working

- Like threading, Tkinter is not necessarily turned on by default on your system. You can tell whether Tkinter is available for your Python interpreter by attempting to import the Tkinter module. If Tkinter is available, then no errors occur:

```
>>> import Tkinter
```

- If your Python interpreter was *not compiled with Tkinter enabled, the module import fails:*

```
>>> import Tkinter
Traceback (innermost last):
File "<stdin>", line 1, in ?
File "/usr/lib/python1.5/lib-tk/Tkinter.py", line 8, in ?
import _tkinter # If this fails your Python may not
        be configured for Tk
ImportError: No module named _tkinter
```

Client/Server Architecture Take Two

- In the earlier chapter on network programming, we introduced the notion of client/server computing. A windowing system is another example of a software server. These run on a machine with an attached display, such as a monitor of some sort. There are clients, too programs that require a windowing environment to execute, also known as GUI applications. Such applications cannot run without a windows system.
- The architecture becomes even more interesting when networking comes into play. Usually when a GUI application is executed, it displays to the machine that it started on (via the windowing server), but it is possible in some networked windowing environments, such as the X Window system on Unix, to choose another machine's window server to display to. In such situations, you can be running a GUI program on one machine, but have it displayed on another

Tkinter and Python Programming

- **Tkinter Module: Adding Tk to your Applications**

  So what do you need to do to have Tkinter as part of your application? Well, first of all, it is not necessary to have an application already. You can create a pure GUI if you want, but it probably isn't too useful without some underlying software that does something interesting.

- There are basically five main steps that are required to get your GUI up and running:

  1. Import the Tkinter module (or from Tkinter import *).

  2. Create a top-level windowing object that contains your entire GUI application.

  3. Build all your GUI components (and functionality) on top (or "inside") of your top-level windowing object.

  4. Connect these GUI components to the underlying application code.

  5. Enter the main event loop.

Introduction to GUI Programming

- Setting up a GUI application is similar to an artist's producing a painting. Conventionally, there is a single canvas onto which the artist must put all the work. The way it works is like this: You start with a clean slate, a "top-level" windowing object on which you build the rest of your components. Think of it as a foundation to a house or the easel for an artist. In other words, you have to pour the concrete or set up your easel before putting together the actual structure or canvas on top of it. In Tkinter, this foundation is known as the top-level window object.
- In GUI programming, a top-level root windowing object contains all of the little windowing objects that will be part of your complete GUI application. These can be text labels, buttons, list boxes, etc. These individual little GUI components are known as *widgets*. *So when we say create a top-level window*, *we* just mean that you need such a thing as a place where you put all your widgets. In Python, this would typically look like this line:

  top = Tkinter.Tk() # or just Tk() with "**from Tkinter import ***"

- The object returned by Tkinter.Tk() is usually referred to as the *root window, hence the reason why* some applications use root rather than top to indicate as such. Top-level windows are those that show up standalone as part of your application. You may have more than one top-level window for your GUI, but only one of them should be your root window. You may choose to completely design all your widgets first, then add the real functionality, or do a little of this and a little of that along the way.
- Widgets may be standalone or be containers. If a widget "contains" other widgets, it is considered the *parent of those widgets. Accordingly, if a widget is "contained" in another widget, it's considered a child* of the parent, the parent being the next immediate enclosing container widget.
- Usually, widgets have some associated behaviors, such as when a button is pressed, or text is filled into a text field. These types of user behaviors are called *events, and the actions that the GUI takes to* respond to such events are known as *callbacks.*

- Actions may include the actual button press (and release), mouse movement, hitting the RETURN or Enter key, etc. All of these are known to the system literally as *events. The entire system of events that* occurs from the beginning to the end of a GUI application is what drives it. This is known as event driven processing.
- The event-driven processing nature of GUIs fits right in with client/server architecture. When you start a GUI application, it must perform some setup procedures to prepare for the core execution, just as when a network server has to allocate a socket and bind it to a local address. The GUI application must establish all the GUI components, then draw (aka render or paint) them to the screen. Tk has a couple of geometry managers that help position the widget in the right place; the main one that you will use is called Pack, aka the *packer. Another geometry manager is Gridthis is where you specify GUI widgets to* be placed in grid coordinates, and Grid will render each object in the GUI in their grid position. For us, we will stick with the packer.

- Once the packer has determined the sizes and alignments of all your widgets, it will then place them on the screen for you. When all of the widgets, including the top-level window, finally appear on your screen, your GUI application then enters a "server-like" infinite loop. This infinite loop involves waiting for a GUI event, processing it, then going back to wait for the next event.
- The final step we described above says to enter the main loop once all the widgets are ready. This is the "server" infinite loop we have been referring to. In Tkinter, the code that does this is:

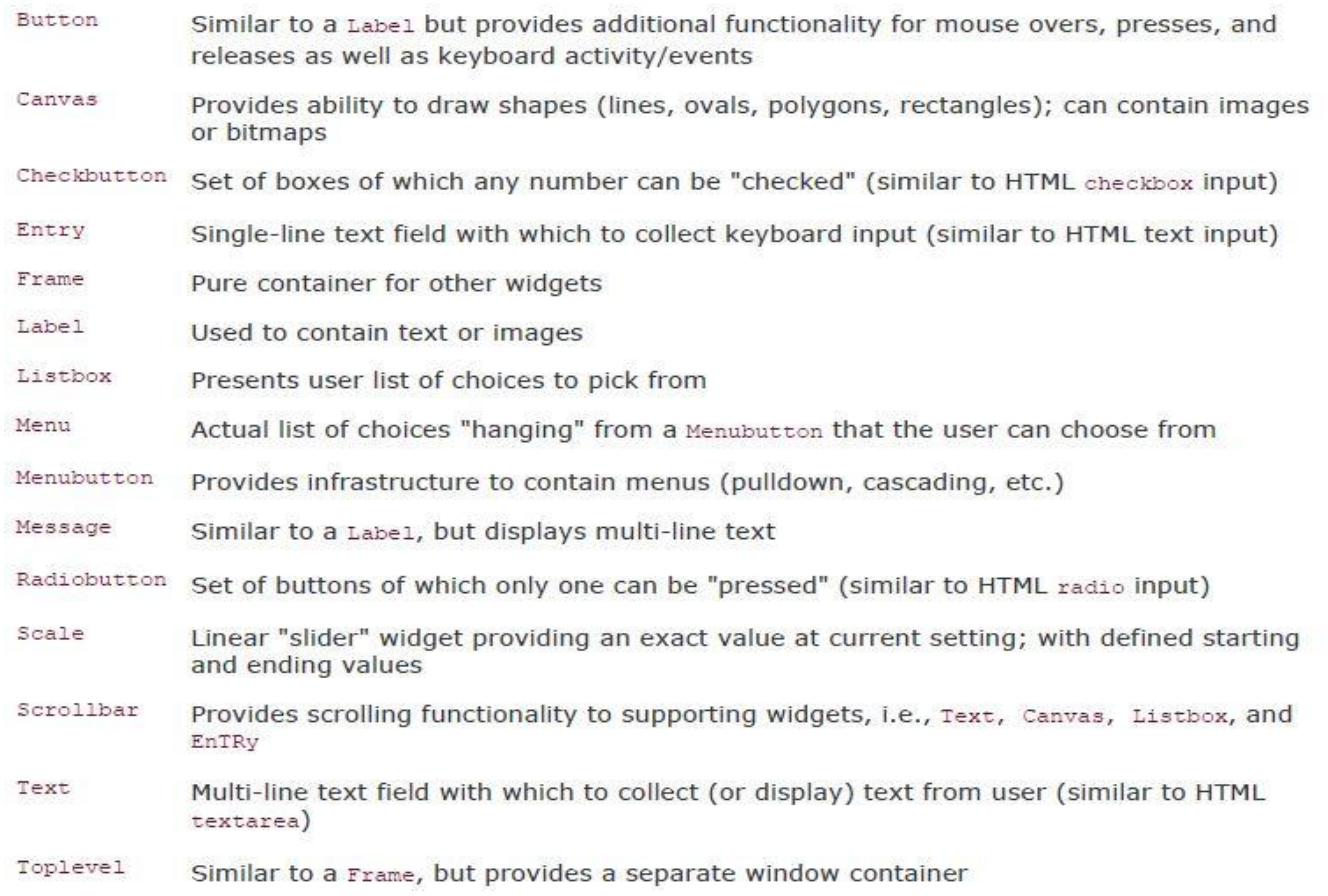
```
Tkinter.mainloop()
```

- This is normally the last piece of sequential code your program runs. When the main loop is entered, the GUI takes over execution from there. All other action is via callbacks, even exiting your application.When you pull down the File menu to click on the Exit menu option or close the window directly, a callback must be invoked to end your GUI application

Top-Level Window: Tkinter.Tk()

- We mentioned above that all main widgets are built into the top-level window object. This object is created by the Tk class in Tkinter and is created via the normal instantiation:
- >>> **import Tkinter**
- >>> top = Tkinter.Tk()
- Within this window, you place individual widgets or multiple-component pieces together to form your GUI. So what kinds of widgets are there? We will now introduce the Tk widgets.

Tk Widgets



`Button`	Similar to a `Label` but provides additional functionality for mouse overs, presses, and releases as well as keyboard activity/events
`Canvas`	Provides ability to draw shapes (lines, ovals, polygons, rectangles); can contain images or bitmaps
`Checkbutton`	Set of boxes of which any number can be "checked" (similar to HTML `checkbox` input)
`Entry`	Single-line text field with which to collect keyboard input (similar to HTML text input)
`Frame`	Pure container for other widgets
`Label`	Used to contain text or images
`Listbox`	Presents user list of choices to pick from
`Menu`	Actual list of choices "hanging" from a `Menubutton` that the user can choose from
`Menubutton`	Provides infrastructure to contain menus (pulldown, cascading, etc.)
`Message`	Similar to a `Label`, but displays multi-line text
`Radiobutton`	Set of buttons of which only one can be "pressed" (similar to HTML `radio` input)
`Scale`	Linear "slider" widget providing an exact value at current setting; with defined starting and ending values
`Scrollbar`	Provides scrolling functionality to supporting widgets, i.e., `Text`, `Canvas`, `Listbox`, and `EnTRy`
`Text`	Multi-line text field with which to collect (or display) text from user (similar to HTML `textarea`)
`Toplevel`	Similar to a `Frame`, but provides a separate window container

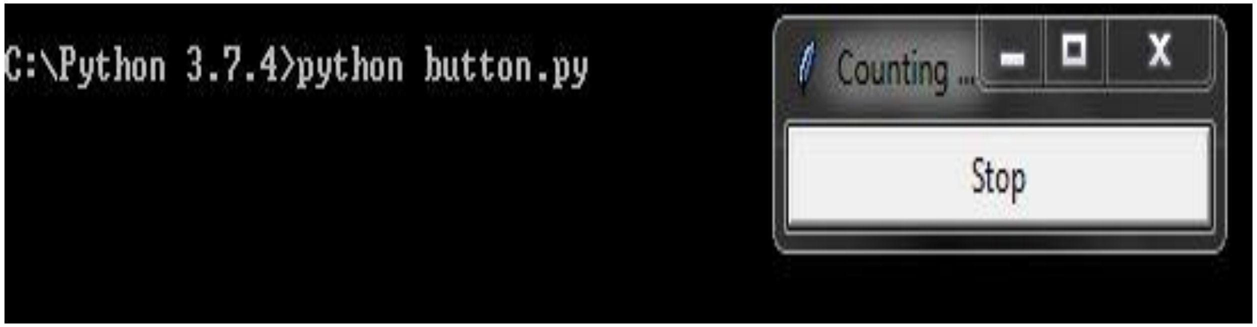
- **Button**:To add a button in your application, this widget is used. The general syntax is:

  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()

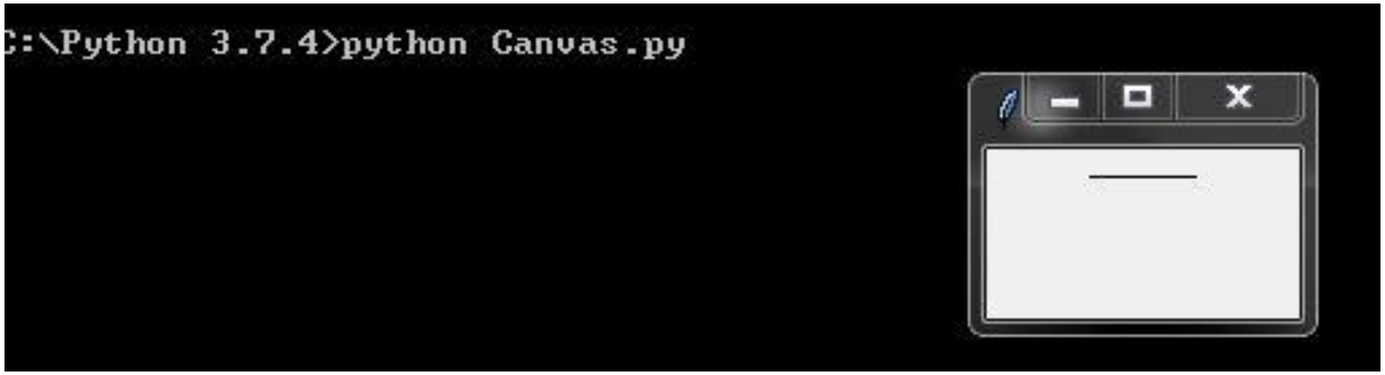
mainloop(): There is a method known by the name mainloop() is used when you are ready for the application to run. mainloop() is an infinite loop used to run the application, wait for an event to occur and process the event till the window is not closed.



```
C:\Python 3.7.4>python button.py
```

- **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.

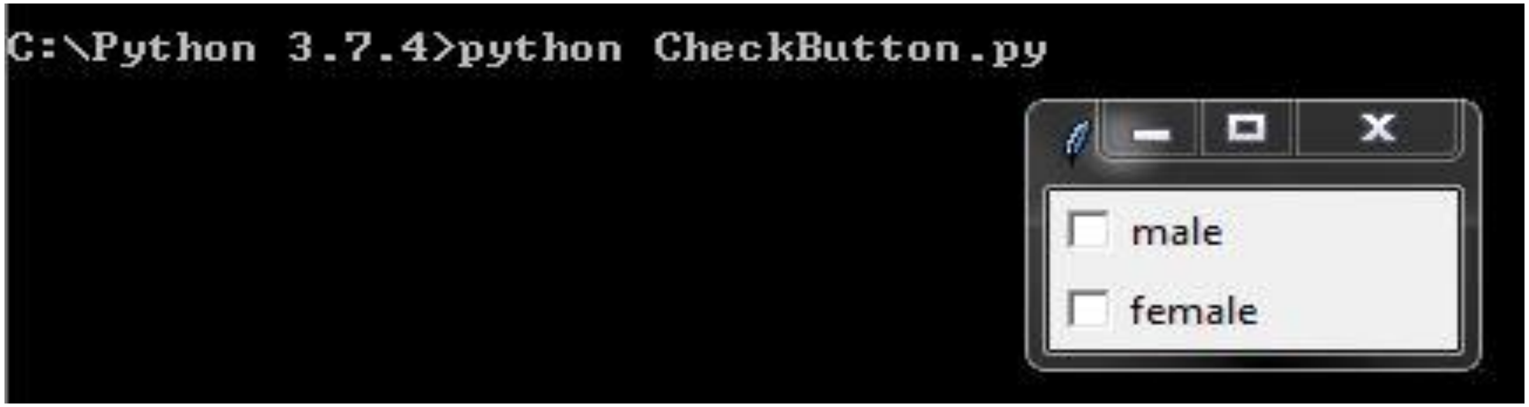
- from tkinter import *
- master = Tk()
- w = Canvas(master, width=40, height=60)
- w.pack()
- canvas_height=20
- canvas_width=200
- y = int(canvas_height / 2)
- w.create_line(0, y, canvas_width, y)
- mainloop()



```
C:\Python 3.7.4>python Canvas.py
```

- **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 backgrouSteganographyBreak
- 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()



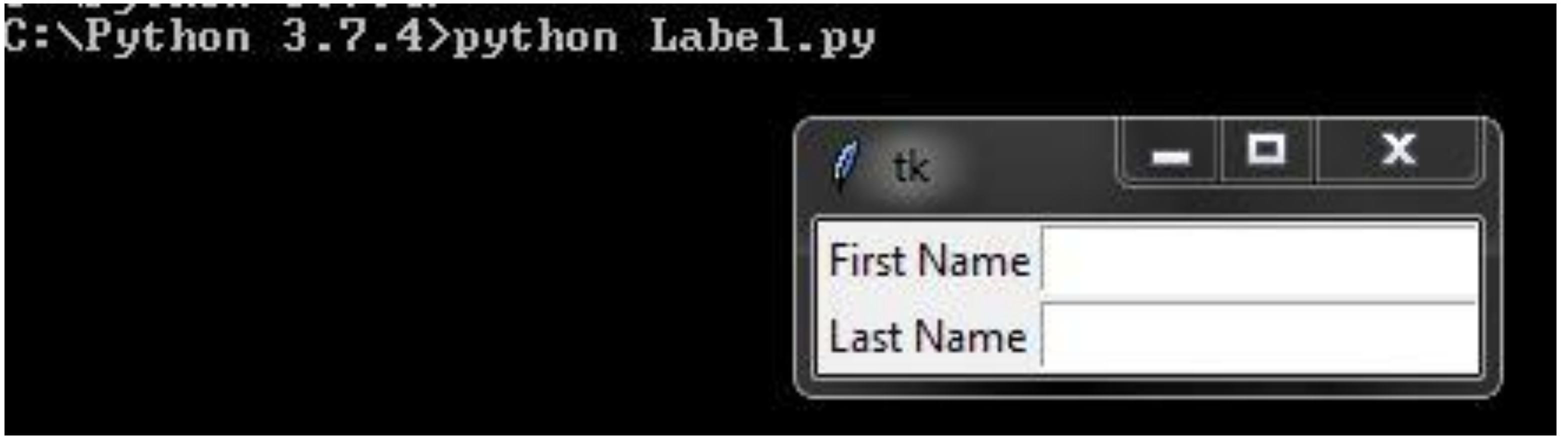
```
C:\Python 3.7.4>python CheckButton.py
```

- **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.

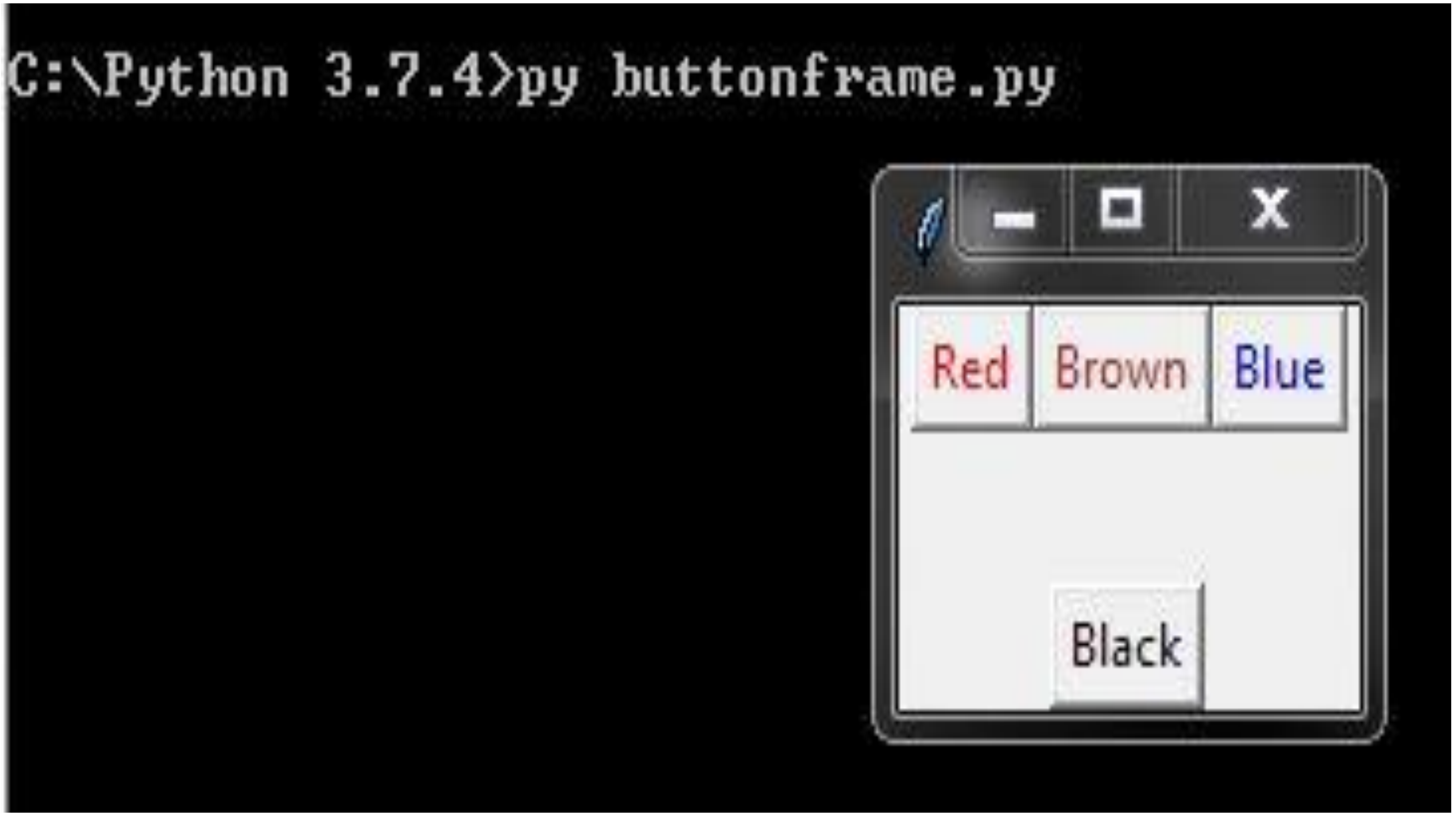
- from tkinter import *
- master = Tk()
- Label(master, text='First Name').grid(row=0)
- Label(master, text='Last Name').grid(row=1)
- e1 = Entry(master)
- e2 = Entry(master)
- e1.grid(row=0, column=1)
- e2.grid(row=1, column=1)
- mainloop()



```
C:\Python 3.7.4>python Label.py
```

- **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()
```

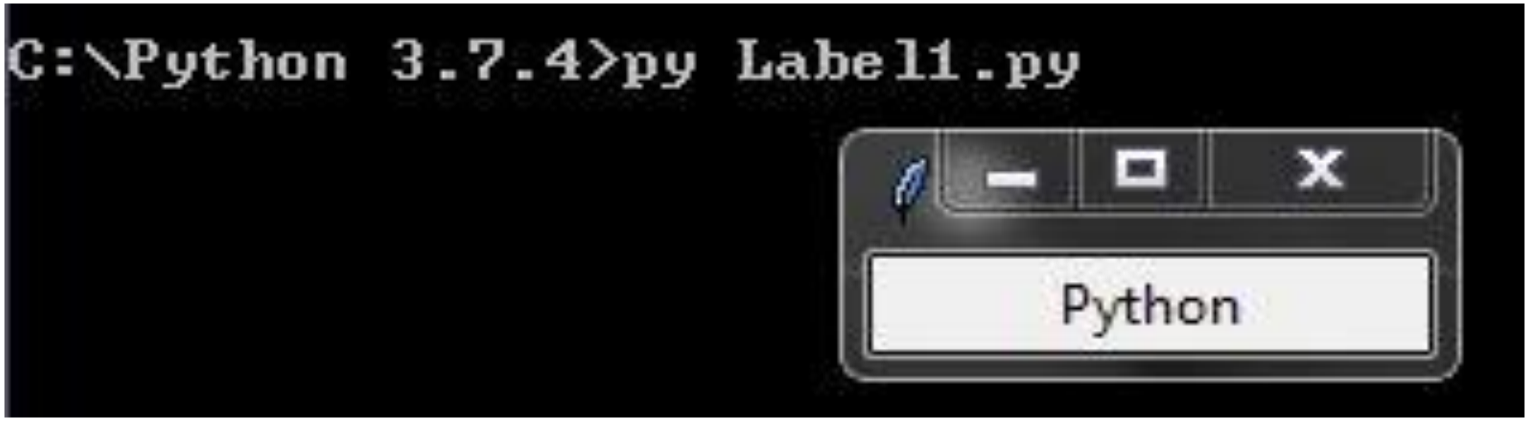


```
C:\Python 3.7.4>py buttonframe.py
```

- **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=Python)
- w.pack()
- root.mainloop()



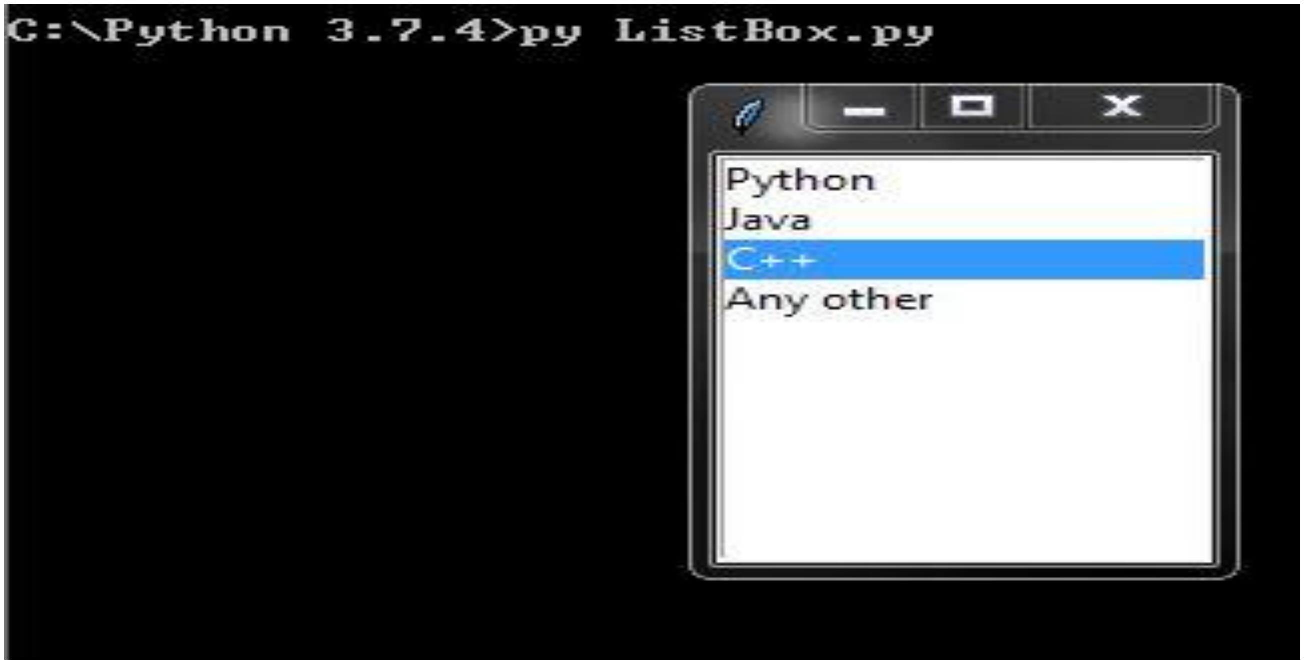
```
C:\Python 3.7.4>py Label1.py
```

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()
```



```
C:\Python 3.7.4>py ListBox.py
```

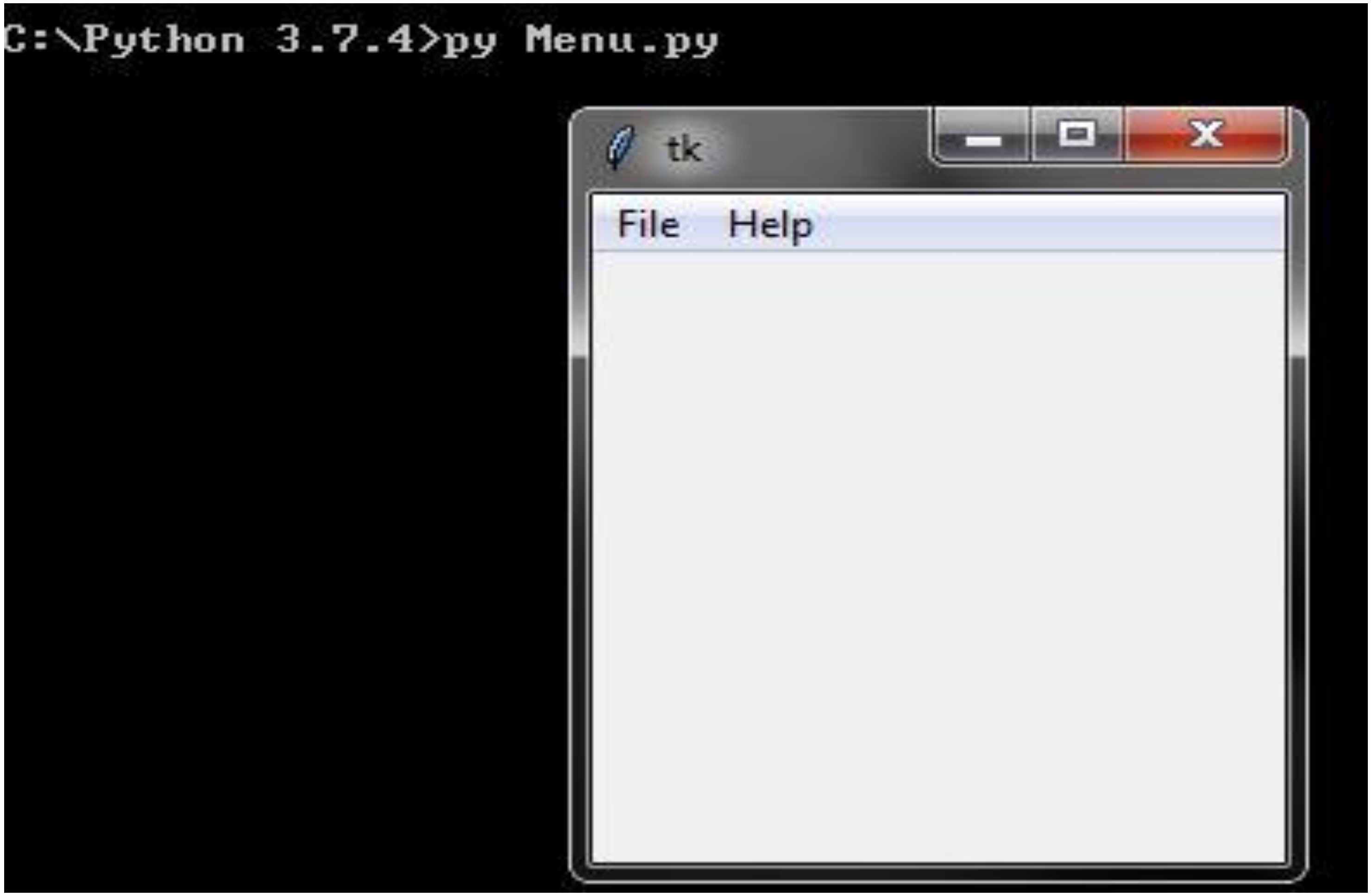
Menu

- **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 widge

- 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()

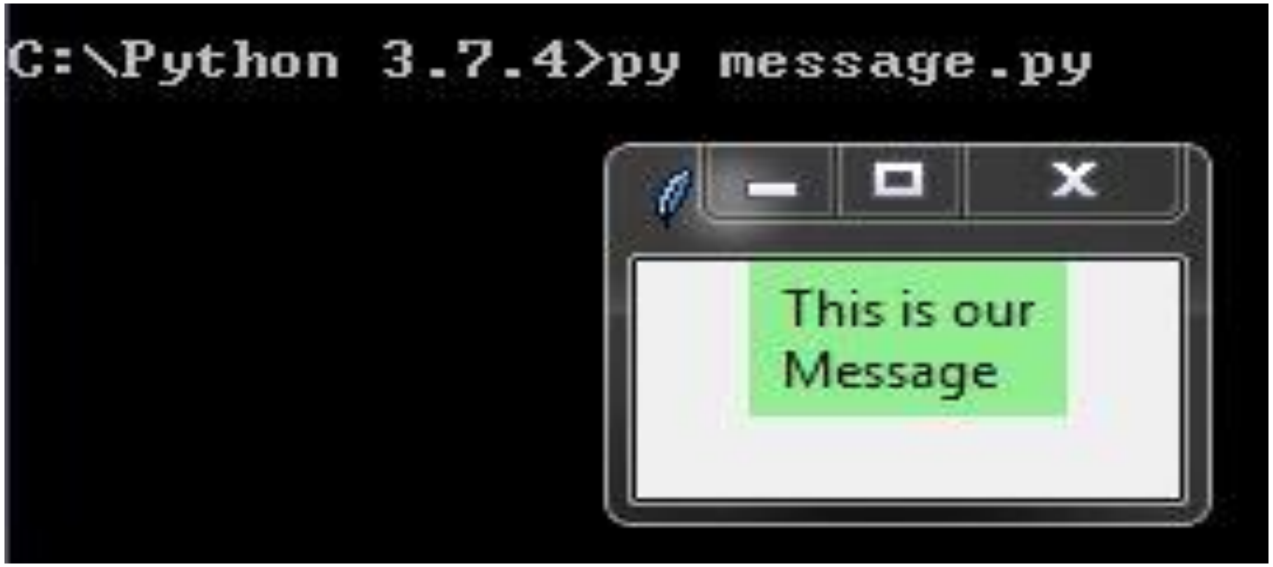


```
C:\Python 3.7.4>py Menu.py
```

Message

- **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( )



```
C:\Python 3.7.4>py message.py
```

This is our
Message

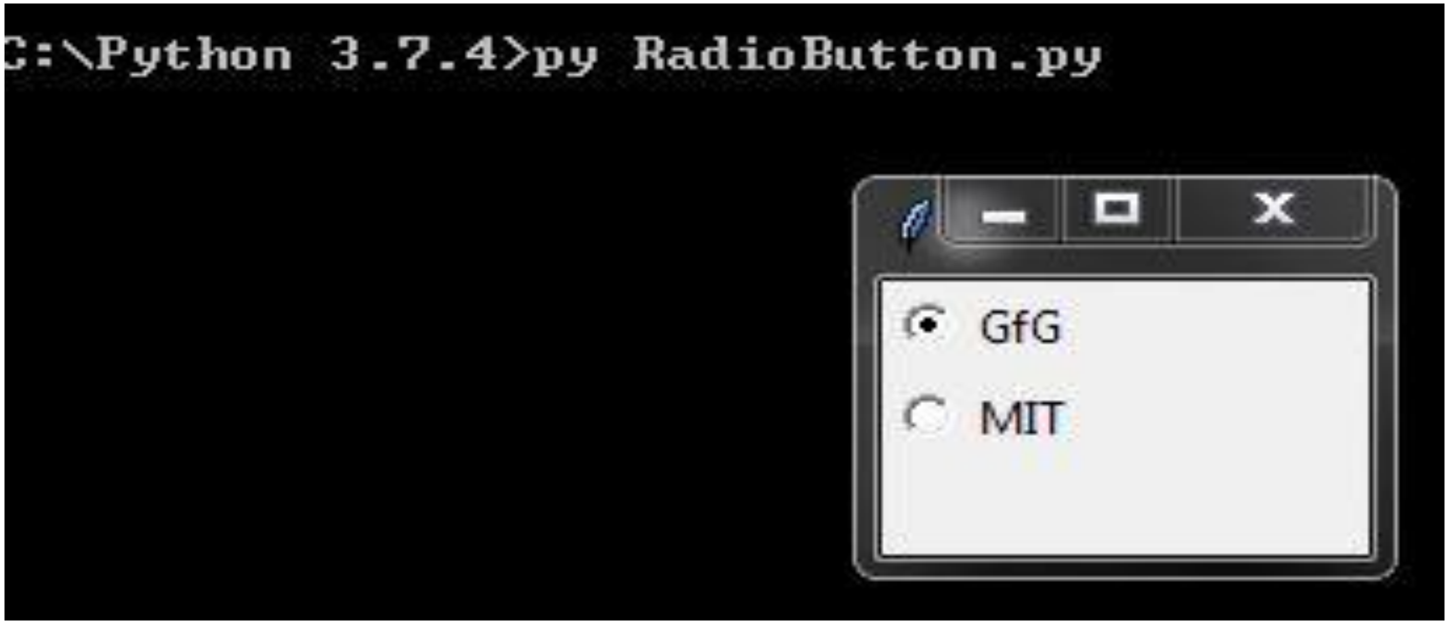
RadioButton

- **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()

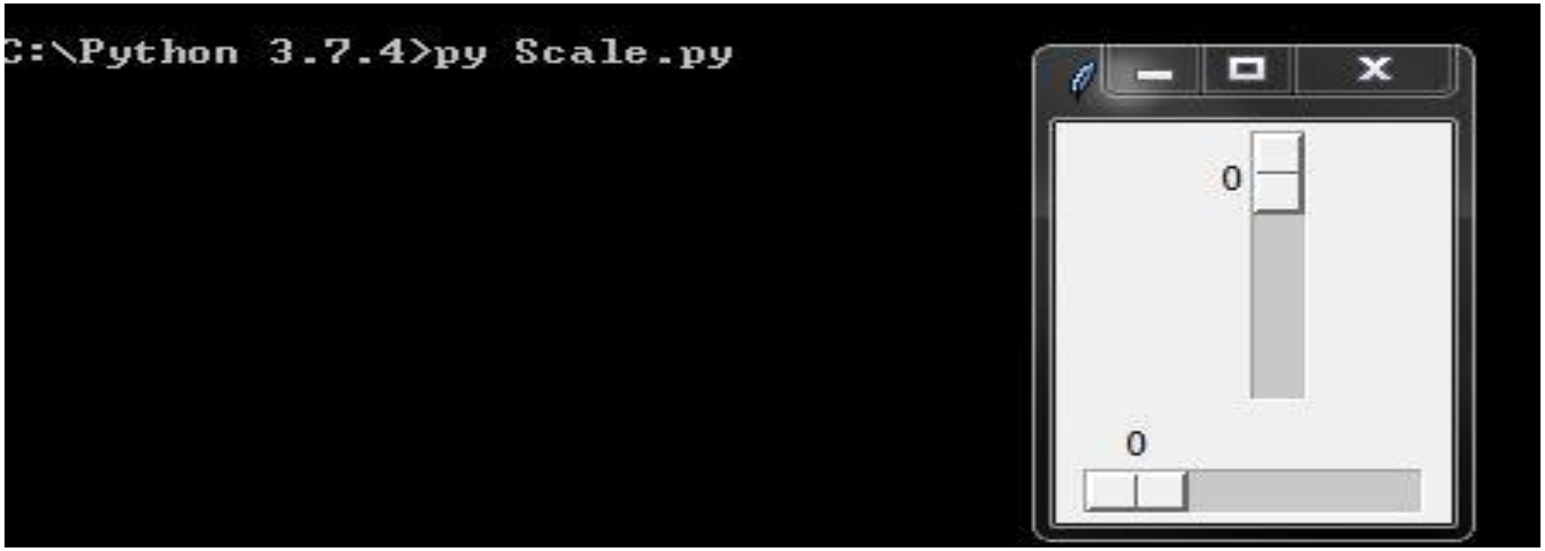


```
C:\Python 3.7.4>py RadioButton.py
```

Scale

- **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()

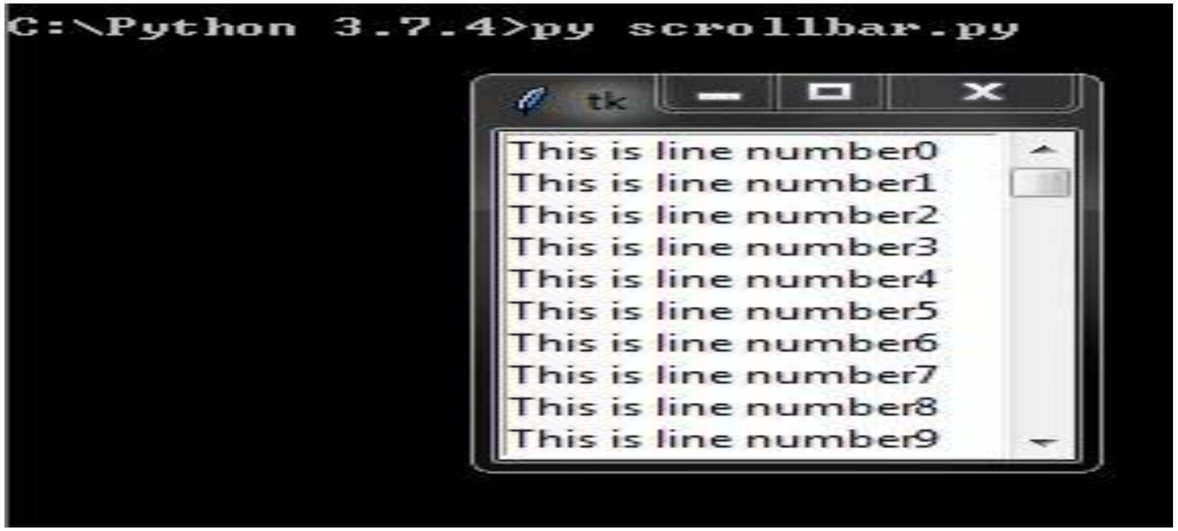


```
C:\Python 3.7.4>py Scale.py
```

Scrollbar

- **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()



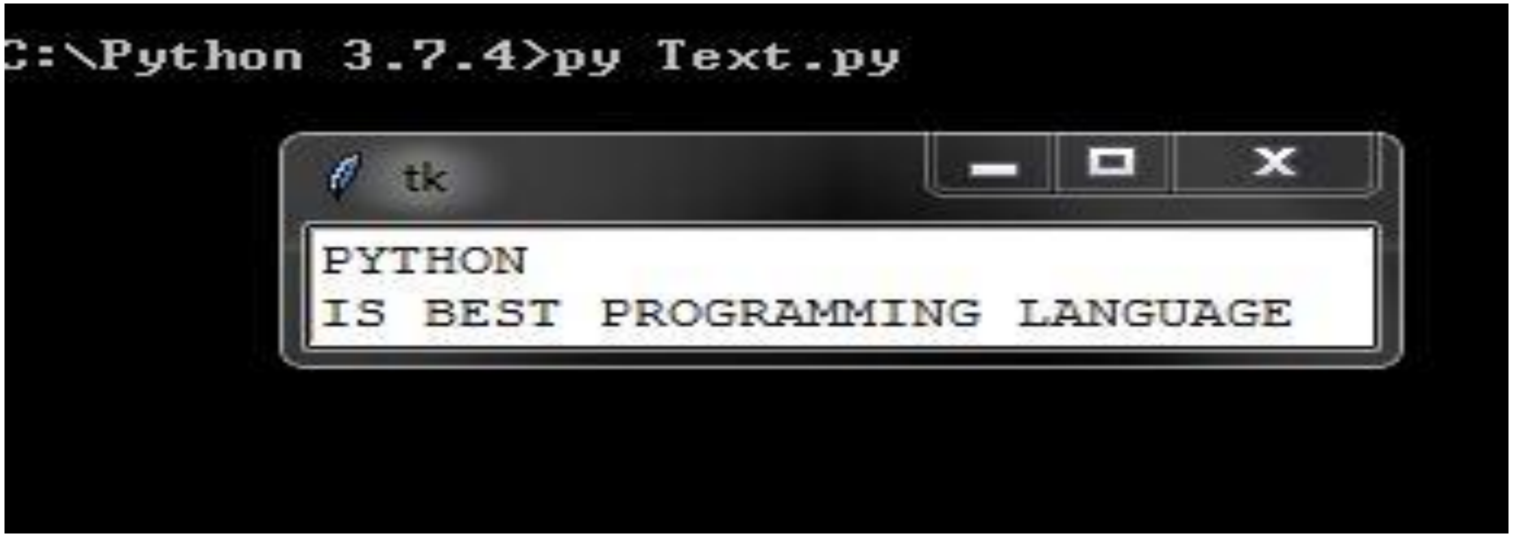
```
C:\Python 3.7.4>py scrollbar.py
```

```
This is line number0
This is line number1
This is line number2
This is line number3
This is line number4
This is line number5
This is line number6
This is line number7
This is line number8
This is line number9
```

Text

- **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, 'PYTHON\nIS BEST PROGRAMMING LANGUAGE\n')
- mainloop()



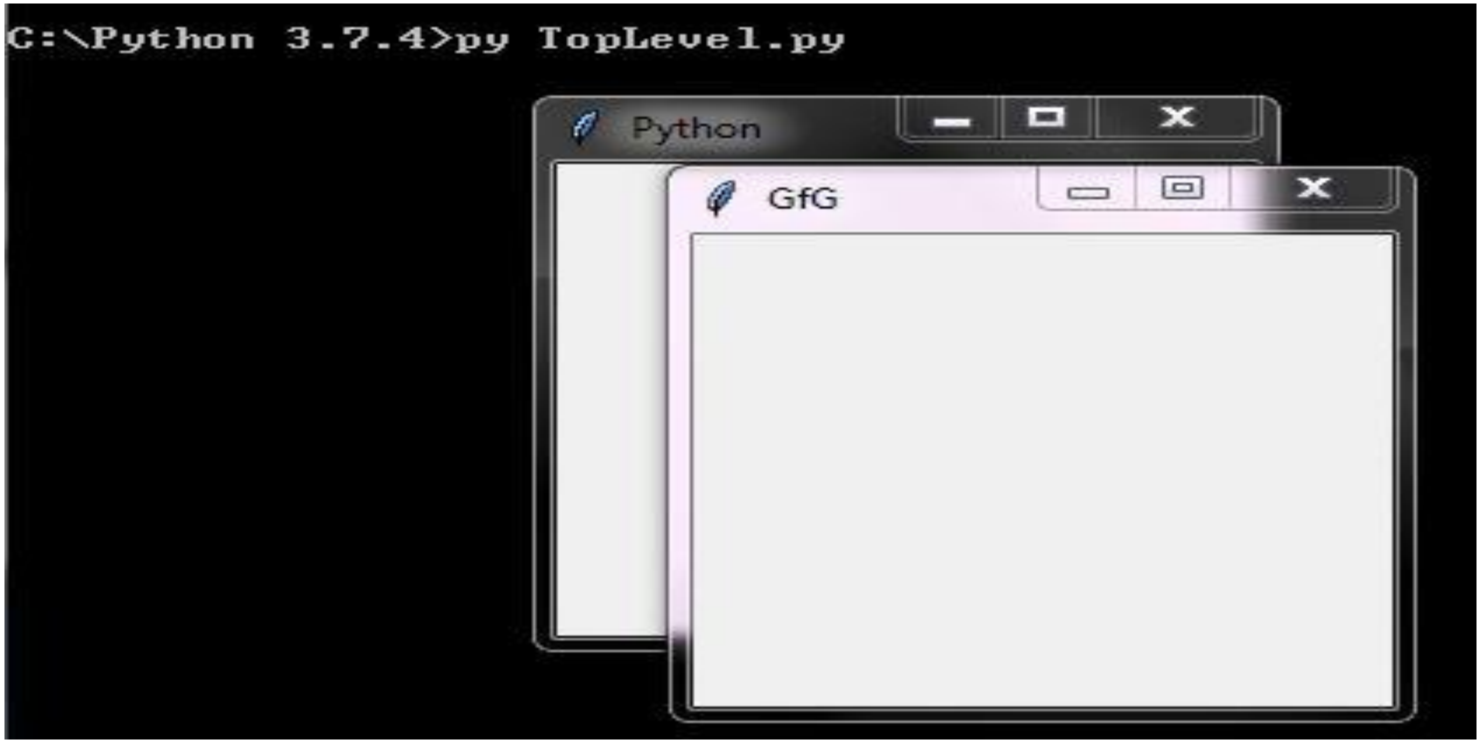
```
C:\Python 3.7.4>py Text.py
```

```
PYTHON
IS BEST PROGRAMMING LANGUAGE
```

Top Level

- **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.

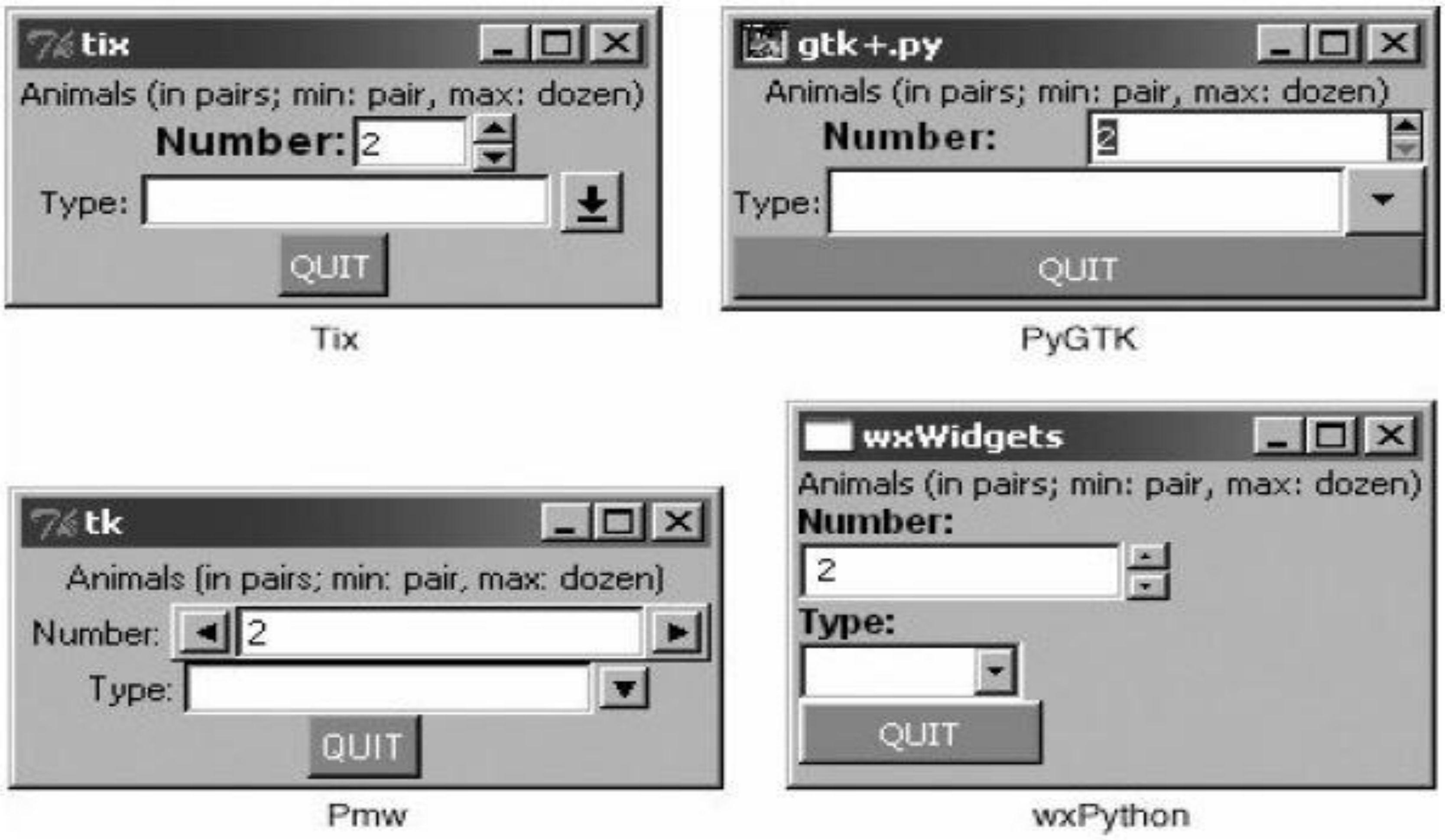
- from tkinter import *
- root = Tk()
- root.title('GfG')
- top = Toplevel()
- top.title('Python')
- top.mainloop()



```
C:\Python 3.7.4>py TopLevel.py
```

Brief Tour of Other GUIs

- We hope to eventually develop an independent chapter on general GUI development using many of the abundant number of graphical toolkits that exist under Python, but alas, that is for the future. As a proxy, we would like to present a single simple GUI application written using four of the more popular and available toolkits out there: Tix (Tk Interface eXtensions), Pmw (Python MegaWidgets Tkinter extension), wxPython (Python binding to wxWidgets), and PyGTK (Python binding to GTK+).
- The Tix module is already available in the Python standard library. You must download the others, which are third party. Since Pmw is just an extension to Tkinter, it is the easiest to install (just extract into your site packages). wxPython and PyGTK involve the download of more than one file and building.


tix
Animals (in pairs; min: pair, max: dozen)
Number:
2
Type:
QUIT
Tix
gtk+.py
Animals (in pairs; min: pair, max: dozen)
Number:
2
Type:
QUIT
PyGTK
tk
Animals (in pairs; min: pair, max: dozen)
Number:
2
Type:
QUIT
Pmw
wxWidgets
Animals (in pairs; min: pair, max: dozen)
Number:
2
Type:
QUIT
wxPython

Application using various GUIs under Win32 (animal*.pyw)

- **Tk Interface eXtensions (Tix)** Tix is an extension library for Td/T that adds many new widgets, image types, and other commands that keep Tk a viable GUI development todkit. Let's take look at how to use Tix with Python.
- **Python MegaWidgets (PMW)** This module was created to
- address the aging Tkinter. It basically helps the extend its longevity by adding more modern widgets to the GUI palette.
- **wxWidgets and wxPython** wxWidgets (formerly known as wxWindows) is a cross-platform toolkit used to build graphical user
- applications. It is implemented using C++ and is available on a wide number of platforms to which wxWidgets defines a consistent and common API. The best part of all is that wxWidgets uses the native
- GUI on each platform, so your program will have the same look-and-feel as all the other applications on your desktop. Another feature is that you are not restricted to developing wxWidgets applications in C++.

GTK+ and PyGTK

- Finally, we have the PyGTK version, which is quite similar to the wxPython GUI.
- The biggest difference is that we use only one class, and it seems more tedious to set the foreground and background colors of objects, buttons in particular.

Related Modules and Other GUIs

GUI Systems Available for Python

Tk-Related Modules

S.NO	GUI Module or System	Description
1	Tkinter	TK INTERface: Python's default GUI toolkit
2	Pmw	Python MegaWidgets (Tkinter extension)
3	Tix	Tk Interface eXtension (Tk extension)
4	TkZinc (Zinc)	Extended Tk canvas type (Tk extension)
5	EasyGUI (easygui)	Very simple non-event-driven GUIs (Tkinter extension) easygui
6	TIDE + (IDE Studio)	Tix Integrated Development Environment (including IDE Studio, a Tixenhanced version of the standard IDLE IDE)

wxWidgets-Related Modules

S.NO	GUI Module or System	Description
1	wxPython	Python binding to wxWidgets, a cross-platform GUI framework (formerly known as wxWindows)
2	Boa Constructor	Python IDE and wxPython GUI builder
3	PythonCard	wxPython-based desktop application GUI construction kit (inspired by HyperCard)
4	wxGlade	another wxPython GUI designer (inspired by Glade, the GTK+/GNOME GUI builder)

GTK+/GNOME-Related Modules		
S.NO	GUI Module or System	Description
1	PyGTK	Python wrapper for the GIMP Toolkit (GTK+) library
2	GNOME-Python	Python binding to GNOME desktop and development libraries
3	Glade	a GUI builder for GTK+ and GNOME
4	PyGUI(GUI)	cross-platform "Pythonic" GUI API (built on Cocoa [MacOS X] and GTK+ [POSIX/X11 and Win32])

Qt/KDE-Related Modules		
S.NO	GUI Module or System	Description
1	PyQt	Python binding for the Qt GUI/XML/SQL C++ toolkit from Trolltech (partially open source [dual-license])
2	PyKDE	Python binding for the KDE desktop environment
3	eric	Python IDE written in PyQt using QScintilla editor widget
4	PyQtGPL	Qt (Win32 Cygwin port), Sip, QScintilla, PyQt bundle

Other Open Source GUI Toolkits

S.NO	GUI Module or System	Description
1	FXPy	Python binding to FOX toolkit
2	pyFLTK (fltk)	Python binding to FLTK toolkit
3	PyOpenGL (OpenGL)	Python binding to OpenGL

Commercial

S.NO	GUI Module or System	Description
1	win32ui	Microsoft MFC (via Python for Windows Extensions
2	swing	Sun Microsystems Java/Swing (via Jython)

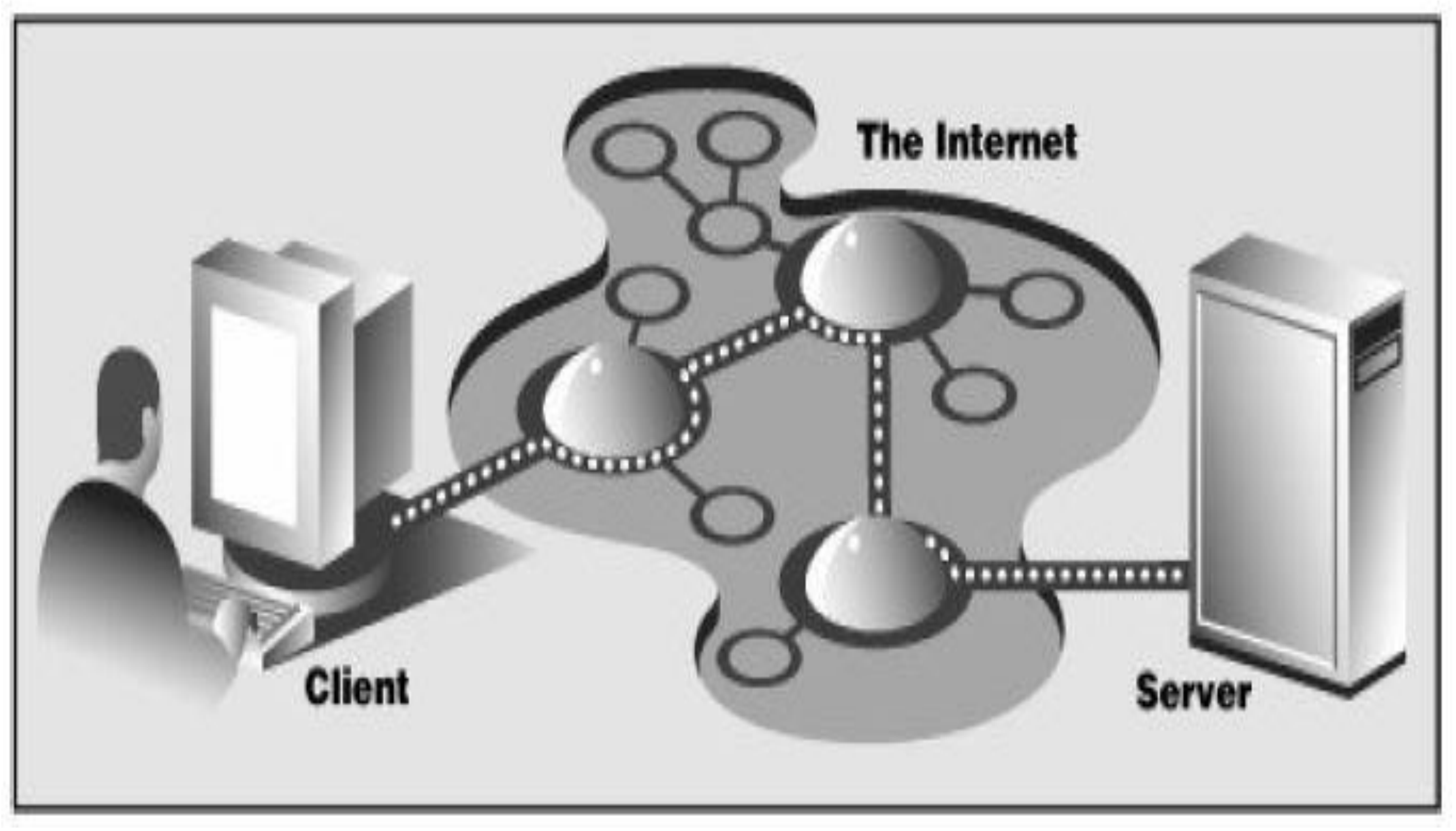
WEB Programming

- **Introduction**

  This chapter on Web programming will give you a quick and high-level overview of the kinds of things you can do with Python on the Internet, from Web surfing to creating user feedback forms, from recognizing Uniform Resource Locators to generating dynamic Web page output.

Web Surfing: Client/Server Computing

- Web surfing falls under the same client/server architecture umbrella that we have seen repeatedly. This time, Web *clients are browsers, applications that allow users to seek documents on the World Wide* Web. On the other side are Web *servers, processes that run on an information provider's host* computers. These servers wait for clients and their document requests, process them, and return the requested data. As with most servers in a client/server system, Web servers are designed to run "forever."



Web client and Web server on the Internet. A client sends a request out over the Internet to the server, which then responds with the requested data back to the client.

- Clients may issue a variety of requests to Web servers. Such requests may include obtaining a Web page for viewing or submitting a form with data for processing. The request is then serviced by the Web server, and the reply comes back to the client in a special format for display purposes.
- The "language" that is spoken by Web clients and servers, the standard protocol used for Web communication, is called *HTTP, which stands for HyperText Transfer Protocol. HTTP is written "on top of"* the TCP and IP protocol suite, meaning that it relies on TCP and IP to carry out its lower-level communication functionality. Its responsibility is not to route or deliver messagesTCP and IP handle thatbut to respond to client requests (by sending and receiving HTTP messages).
- HTTP is known as a "stateless" protocol because it does not keep track of information from one client request to the next, similar to the client/server architecture we have seen so far. The server stays running, but client interactions are singular events structured in such a way that once a client request is serviced, it quits. New requests can always be sent, but they are considered separate service requests. Because of the lack of context per request, you may notice that some URLs have a long set of variables and values chained as part of the request to provide some sort of state information. Another alternative is the use of "cookies"static data stored on the client side which generally contain state information as well. In later parts of this chapter, we will look at how to use both long URLs and cookies to maintain state information.

The Internet

- The Internet is a moving and fluctuating "cloud" or "pond" of interconnected clients and servers scattered around the globe. Communication between client and server consists of a series of connections from one lily pad on the pond to another, with the last step connecting to the server. As a client user, all this detail is kept hidden from your view. The abstraction is to have a direct connection between you the client and the server you are "visiting," but the underlying HTTP, TCP, and IP protocols are hidden underneath, doing all of the dirty work. Information regarding the intermediate "nodes" is of no concern or consequence to the general user anyway, so it's good that the implementation is hidden.

Web Surfing with Python: Creating Simple Web Clients

- One thing to keep in mind is that a browser is only one type of Web client. Any application that makes a request for data from a Web server is considered a "client." Yes, it is possible to create other clients that retrieve documents or data off the Internet. One important reason to do this is that a browser provides only limited capacity, i.e., it is used primarily for viewing and interacting with Web sites. A client program, on the other hand, has the ability to do moreit can not only download data, but it can also store it, manipulate it, or perhaps even transmit it to another location or application.
- Applications that use the urllib module to download or access information on the Web [using either urllib.urlopen() or urllib.urlretrieve()] can be considered a simple Web client. All you need to do is provide a valid Web address.

Uniform Resource Locators

- Simple Web surfing involves using Web addresses called *URLs (Uniform Resource Locators). Such* addresses are used to locate a document on the Web or to call a CGI program to generate a document for your client. URLs are part of a larger set of identifiers known as *URIs (Uniform Resource Identifiers).*
- This superset was created in anticipation of other naming conventions that have yet to be developed. A URL is simply a URI which uses an existing protocol or scheme (i.e., http, ftp, etc.) as part of its addressing. To complete this picture, we'll add that non-URL URIs are sometimes known as *URNs* (Uniform Resource Names), but because URLs are the only URIs in use today, you really don't hear much about URIs or URNs, save perhaps as XML identifiers.
- Like street addresses, Web addresses have some structure. An American street address usually is of the form "number street designation," i.e., 123 Main Street. It differs from other countries, which have their own rules.

  A URL uses the format:

  prot_sch://net_loc/path;params?query#frag

Web Address Components

URL Component	Description
prot_sch	Network protocol or download scheme
net_loc	Location of server (and perhaps user information)
path	Slash (/) delimited path to file or CGI application
params	Optional parameters
query	Ampersand (&) delimited set of "key=value" pairs
frag	Fragment to a specific anchor within document

Network Location Components

net_loc Component	Description
User	User name or login
passwd	User password
host	Name or address of machine running Web server [required]
port	Port number (if not 80, the default)

urlparse Module

- The urlparse module provides basic functionality with which to manipulate URL strings.

  These functions include urlparse(), urlunparse(), and urljoin().
- **urlparse.urlparse()**

  urlparse() breaks up a URL string into some of the major components described above. It has the

  following syntax:

  urlparse(*urlstr, defProtSch=None, allowFrag=None)*
- urlparse() parses *urlstr into a 6-tuple (prot_sch, net_loc, path, params, query, frag). Each of these* components has been described above. *defProtSch indicates a default network protocol or download* scheme in case one is not provided in urlstr.*allowFrag is a flag that signals whether or not a fragment*
- part of a URL is allowed. Here is what urlparse() outputs when given a URL:
- >>> urlparse.urlparse('http://www.python.org/doc/FAQ.html')
- ('http', 'www.python.org', '/doc/FAQ.html', '', '', '')

- **urlparse.urlunparse()**
- urlunparse() does the exact opposite of urlparse()it merges a 6-tuple (prot_sch, net_loc, path, params,query, frag)*urltup, which could be the output of urlparse(), into a single URL string and returns it.*
- Accordingly, we state the following equivalence:

  urlunparse(urlparse(*urlstr)) urlstr*
- You may have already surmised that the syntax of urlunparse() is as follows:

  urlunparse(*urltup)*
- **urlparse.urljoin()**
- The urljoin() function is useful in cases where many related URLs are needed, for example, the URLs for a set of pages to be generated for a Web site.
- The syntax for urljoin() is:

  urljoin(*baseurl, newurl, allowFrag=None)*
- urljoin() takes *baseurl and joins its base path (net_loc plus the full path up to, but not including, a file* at the end) with *newurl. For example:*
- >>> urlparse.urljoin('http://www.python.org/doc/FAQ.html', \
- ... 'current/lib/lib.htm')
- 'http://www.python.org/doc/current/lib/lib.html'

Core urlparse Module Functions

urlparse Functions	Description
urlparse(*urlstr, defProtSch=None, allowFrag=None)*	Parses *urlstr into separate components, using defProtSch if the protocol or scheme is not given* in *urlstr; allowFrag determines whether a URL* fragment is allowed
urlunparse(*urltup)*	Unparses a tuple of URL data (*urltup) into a* single URL string
urljoin(*baseurl,newurl,allowFrag=None)*	Merges the base part of the *baseurl URL with newurl to form a complete URL; allowFrag is the* same as for urlparse()

urllib Module

- The urllib module provides functions to download data from given URLs as well as encoding and decoding strings to make them suitable for including as part of valid URL strings.

The functions we will be looking at in this upcoming section include: urlopen(), urlretrieve(), quote(), unquote(), quote_plus(), unquote_plus(), and urlencode().

We will also look at some of the methods available to the file-like

object returned by urlopen().

- **urllib.urlopen()**
- urlopen() opens a Web connection to the given URL string and returns a file-like object. It has the following syntax:

  urlopen(*urlstr, postQueryData=None)*
- urlopen() opens the URL pointed to by *urlstr. If no protocol or download scheme is given, or if a "file"* scheme is passed in, urlopen() will open a local file.
- For all HTTP requests, the normal request type is "GET." In these cases, the query string provided to the Web server (key-value pairs encoded or "quoted," such as the string output of the urlencode() function [see below), should be given as part of *urlstr.*

- If the "POST" request method is desired, then the query string (again encoded) should be placed in the *postQueryData variable. (For more information regarding the GET and POST request methods, refer to* any general documentation or texts on programming CGI applications which we will also discuss below.GET and POST requests are the two ways to "upload" data to a Web server.
- When a successful connection is made, urlopen() returns a file-like object as if the destination was a file opened in read mode. If our file object is f, for example, then our "handle" would support the expected read methods such as f.read(), f.readline(), f.readlines(), f.close(), and f.fileno().
- In addition, a f.info() method is available which returns the *MIME (Multipurpose Internet Mail* Extension) headers. Such headers give the browser information regarding which application can view returned file types. For example, the browser itself can view *HTML (HyperText Markup Language), plain* text files, and render *PNG (Portable Network Graphics) and JPEG (Joint Photographic Experts Group) or* the old *GIF (Graphics Interchange Format) graphics files. Other files such as multimedia or specific* document types require external applications in order to view.

urllib.urlopen() *File-like Object Methods*

urlopen() Object Methods	Description
f.read([bytes])	Reads all or bytes bytes from f
f.readline()	Reads a single line from f
f.readlines()	Reads a all lines from f into a list
f.close()	Closes URL connection for f
f.fileno()	Returns file number of f
f.info()	Gets MIME headers of f
f.geturl()	Returns true URL opened for f

- **urllib.urlretrieve()**
- urlretrieve() will do some quick and dirty work for you if you are interested in working with a URL document as a whole.

  Here is the syntax for urlretrieve():

  urlretrieve(*urlstr, localfile=None, downloadStatusHook=*None)

Core urllib Module Functions

urllib Functions	Description
urlopen(*urlstr, postQueryData=None)*	Opens the URL *urlstr, sending the query* data in *postQueryData if a POST request*
urlretrieve(*urlstr, localfile=None, downloadStatusHook=None)*	Downloads the file located at the *urlstr* URL to *localfile or a temporary file if localfile not given; if present, downloaStatusHook is a function that can* receive download statistics
quote(*urldata, safe='/'*)	Encodes invalid URL characters of *urldata; characters in safe string are not* Encoded
quote_plus(*urldata, safe='/'*)	Same as quote() except encodes spaces as plus (+) signs
unquote(*urldata)*	Decodes encoded characters of *urldata*
unquote_plus(*urldata)*	Same as unquote() but converts plus signs to spaces
urlencode(*dict)*	Encodes the key-value pairs of *dict into* a valid string for CGI queries and encodes the key and value strings with quote_plus ()

Advanced Web Clients

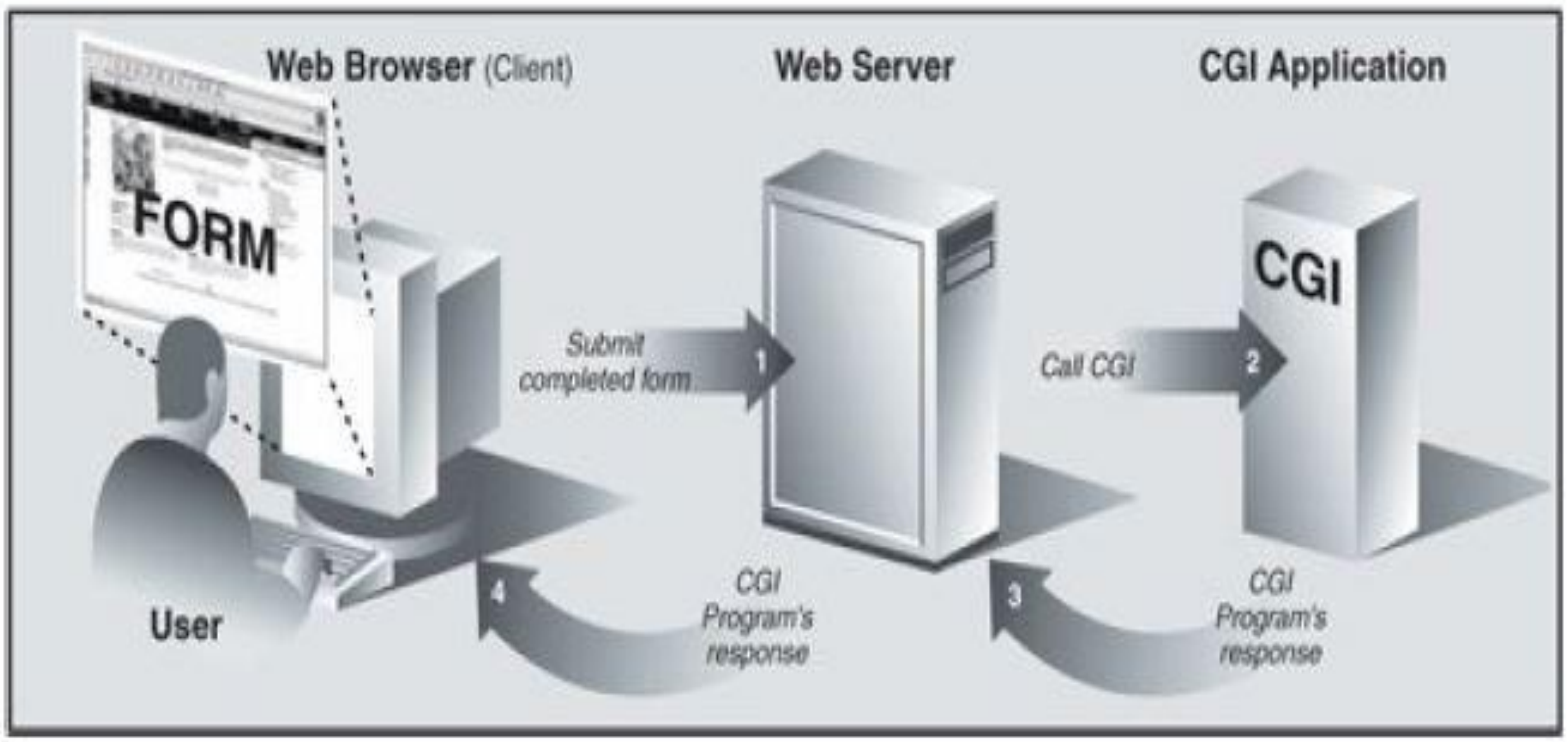
- Web browsers are basic Web clients. They are used primarily for searching and downloading documents from the Web. Advanced clients of the Web are those applications that do more than download single documents from the Internet.
- One example of an advanced Web client is a *crawler (aka spider, robot). These are programs that* explore and download pages from the Internet for different reasons, some of which include:
- ● Indexing into a large search engine such as Google or Yahoo!
- ● Offline browsingdownloading documents onto a local hard disk and rearranging hyperlinks to
- create almost a mirror image for local browsing
- ● Downloading and storing for historical or archival purposes, or
- ● Web page caching to save superfluous downloading time on Web site revisits.
- The crawler we present below, crawl.py, takes a starting Web address (URL), downloads that page and all other pages whose links appear in succeeding pages, but only those that are in the same domain as the starting page. Without such limitations, you will run out of disk space!

CGI: Helping Web Servers Process Client Data

- **Introduction to CGI**
- The Web was initially developed to be a global online repository or archive of (mostly educational and research-oriented) documents. Such pieces of information generally come in the form of static text and usually in HTML.
- HTML is not as much a *language as it is a text formatter, indicating changes in font types, sizes, and* styles. The main feature of HTML is in its hypertext capability, text that is in one way or another highlighted to point to another document in a related context to the original. Such a document can be accessed by a mouse click or other user selection mechanism. These (static) HTML documents live on the Web server and are sent to clients when and if requested.
- As the Internet and Web services evolved, there grew a need to process user input. Online retailers needed to be able to take individual orders, and online banks and search engine portals needed to create accounts for individual users. Thus fill-out forms were invented, and became the only way a Web site can get specific information from users (until Java applets came along). This, in turn, required the HTML now be generated on the fly, for each client submitting user-specific data.

Now, Web servers are only really good at one thing, getting a user request for a file and returning that file (i.e., an HTML file) to the client. They do not have the "brains" to be able to deal with user-specific data such as those which come from fields. Not being their responsibility, Web servers farm out such requests to external applications which create the dynamically generated HTML that is returned to the client.

The entire process begins when the Web server receives a client request (i.e., GET or POST) and calls the appropriate application. It then waits for the resulting HTML meanwhile, the client also waits. Once the application has completed, it passes the dynamically generated HTML back to the server, who then (finally) forwards it back to the user. This process of the server receiving a form, contacting an external application, and receiving and returning the newly-generated HTML takes place through what is called
the Web server's *CGI (Common Gateway Interface)*.



Overview of how CGI works. CGI represents the interaction between a Web server and the application that is required to process a user's form and generate the dynamic HTML that is eventually returned

- Forms input from the client sent to a Web server may include processing and perhaps some form of storage in a backend database. Just keep in mind that any time there are any user-filled fields and/or a Submit button or image, it most likely involves some sort of CGI activity.
- CGI applications that create the HTML are usually written in one of many higher-level programming languages that have the ability to accept user data, process it, and return HTML back to the server.
- Currently used programming languages include Perl, PHP, C/C++, or Python, to name a few. Before we take a look at CGI, we have to provide the caveat that the typical production Web application is no longer being done in CGI anymore.
- Because of its significant limitations and limited ability to allow Web servers to process an abundant number of simultaneous clients, CGI is a dinosaur. Mission-critical Web services rely on compiled languages like C/C++ to scale. A modern-day Web server is typically composed of Apache and integrated components for database access (MySQL or PostgreSQL), Java (Tomcat), PHP, and various modules for Perl, Python, and SSL/security. However, if you are working on small personal Web sites or ones for small organizations and do not need the power and complexity required by mission critical Web services, CGI is the perfect tool for your simple Web sites.

CGI Applications

- A CGI application is slightly different from a typical program. The primary differences are in the input, output, and user interaction aspects of a computer program. When a CGI script starts, it needs to retrieve the user-supplied form data, but it has to obtain this data from the Web client, not a user on the server machine nor a disk file.
- The output differs in that any data sent to standard output will be sent back to the connected Web client rather than to the screen, GUI window, or disk file. The data sent back must be a set of valid headers followed by HTML. If it is not and the Web client is a browser, an error (specifically, an Internal Server Error) will occur because Web clients such as browsers understand only valid HTTP data (i.e., MIME headers and HTML).
- Finally, as you can probably guess, there is no user interaction with the script. All communication occurs among the Web client (on behalf of a user), the Web server, and the CGI application.

cgi Module

- There is one primary class in the cgi module that does all the work: the FieldStorage class. This class should be instantiated when a Python CGI script begins, as it will read in all the pertinent user information from the Web client (via the Web server). Once this object has been instantiated, it will consist of a dictionary-like object that has a set of key-value pairs. The keys are the names of the form items that were passed in through the form while the values contain the corresponding data.
- These values themselves can be one of three objects. They can be FieldStorage objects (instances) as well as instances of a similar class called MiniFieldStorage, which is used in cases where no file uploads or multiple-part form data is involved. MiniFieldStorage instances contain only the key-value pair of the name and the data. Lastly, they can be a list of such objects. This occurs when a form contains more than one input item with the same field name.
- For simple Web forms, you will usually find all MiniFieldStorage instances. All of our examples below pertain only to this general case.

Building CGI Applications

- In order to play around with CGI development in Python, you need to first install a Web server, configure it for handling Python CGI requests, and then give the Web server access to your CGI scripts. Some of these tasks may require assistance from your system administrator.
- If you want a real Web server, you will likely download and install Apache. There are Apache plug-ins or modules for handling Python CGI, but they are not required for our examples. You may wish to install those if you are planning on "going live" to the world with your service. Even this may be overkill.

- If you want to just start up the most basic Web server, just execute it directly with Python:

```
$ python -m CGIHTTPServer
```

Advanced CGI

- We will now take a look at some of the more advanced aspects of CGI programming. These include the use of *cookiescached data saved on the client sidemultiple values for the same CGI field and file upload* using multipart form submissions. To save space, we will show you all three of these features with a single application.
- **Multipart Form Submission and File Uploading**
- Currently, the CGI specifications only allow two types of form encodings, "application/x-www-formurlencoded”and "multipart/form-data." Because the former is the default, there is never a need to state the encoding in the FORM tag like this:
- <FORM enctype="application/x-www-form-urlencoded" ...>
- But for multipart forms, you must explicitly give the encoding as:
- <FORM enctype="multipart/form-data" ...>
- You can use either type of encoding for form submissions, but at this time, file uploads can only be performed with the multipart encoding. Multipart encoding was invented by Netscape in the early days but has since been adopted by Microsoft (starting with version 4 of Internet Explorer) as well as other browsers.

- File uploads are accomplished using the file input type:

  `<INPUT type=file name=...>`

  This directive presents an empty text field with a button on the side which allows you to browse your file directory structure for a file to upload. When using multipart, your Web client's form submission to the server will look amazingly like (multipart) e-mail messages with attachments. A separate encoding was needed because it just would not be necessarily wise to "urlencode" a file, especially a binary file. The information still gets to the server, but it is just "packaged" in a different way.

Multivalued Fields

- In addition to file uploads, we are going to show you how to process fields with multiple values. The most common case is when you have a set of checkboxes allowing a user to select from various choices.
- Each of the checkboxes is labeled with the same field name, but to differentiate them, each will have a different value associated with a particular checkbox. As you know, the data from the user are sent to the server in key-value pairs during form submission.

Cookies

- Cookies : they are just bits of data information which a server at a Web site will request to be saved on the client side, e.g., the browser
- cookies.py is a Python module for working with HTTP cookies: parsing and rendering ‘Cookie:’ request headers and ‘Set-Cookie:’ response headers, and exposing a convenient API for creating and modifying cookies. It can be used as a replacement of Python’s Cookie.py
  (aka http.cookies).

Web (HTTP) Servers

- we have been discussing the use of Python in creating Web clients and performing tasks to aid Web servers in CGI request processing. We know that Python can be used to create both simple and complex Web clients. Complexity of CGI requests goes without saying.
- However, we have yet to explore the creation of Web *servers, and that is the focus of this section. If the* Firefox, Mozilla, IE, Opera, Netscape, AOL, Safari, Camino, Epiphany, Galeon, and Lynx browsers are among the most popular Web clients, then what are the most common Web servers? They are Apache, Netscape, IIS, thttpd, Zeus, and Zope. In situations where these servers may be overkill for your desired application, Python can be used to create simple yet useful Web servers.

Creating Web Servers in Python

- Since you have decided on building such an application, you will naturally be creating all the custom stuff, but all the base code you will need is already available in the Python Standard Library. To create a Web server, a base server and a "handler" are required.
- The base (Web) server is a boilerplate item, a must have. Its role is to perform the necessary HTTP communication between client and server. The base server class is (appropriately) named HTTPServer and is found in the BaseHTTPServer module.
- The handler is the piece of software that does the majority of the "Web serving." It processes the client request and returns the appropriate file, whether static or dynamically generated by CGI. The complexity of the handler determines the complexity of your Web server. The Python standard library provides three different handlers.

- The most basic, plain, vanilla handler, named BaseHTTPRequestHandler, is found in the BaseHTTPServer module, along with the base Web server. Other than taking a client request, no other handling is implemented at all, so you have to do it all yourself, such as in our myhttpd.py server coming up.
- The SimpleHTTPRequestHandler, available in the SimpleHTTP-Server module, builds on BaseHTTPRequestHandler by implementing the standard GET and HEAD requests in a fairly straightforward manner. Still nothing sexy, but it gets the simple jobs done.
- Finally, we have the CGIHTTPRequestHandler, available in the CGIHTTPServer module, which takes the SimpleHTTPRequestHandler and adds support for POST requests. It has the ability to call CGI scripts to

  perform the requested processing and can send the generated HTML back to the client.

Web Server Modules and Classes

Module	Description
BaseHTTPServer	Provides the base Web server and base handler classes, HTTPServer and BaseHTTPRequestHandler, respectively
SimpleHTTPServer	Contains the SimpleHTTPRequestHandler class to perform GET and HEAD requests
CGIHTTPServer	Contains the CGIHTTPRequestHandler class to process POST requests and perform CGI execution

THANK YOU