UNIT -4 (Network Programming & GUI Using Python)

Network Programming

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols. Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

Protocol: In networking, a protocol is a set of rules for formatting and processing data. Network protocols are like a common language for computers. The computers within a network may use vastly different software and hardware; however, the use of protocols enables them to communicate with each other regardless. Standardized protocols are like a common language that computers can use, similar to how two people from different parts of the world may not understand each other's native languages, but they can communicate using a shared third language. If one computer uses the Internet Protocol (IP) and a second computer does as well, they will be able to communicate — just as the United Nations relies on its 6 official languages to communicate amongst representatives from all over the globe. But if one computer uses IP and the other does not know this protocol, they will be unable to communicate. On the Internet, there are different protocols for different types of processes. Protocols are often discussed in terms of which OSI model layer they belong to.

Sockets: Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents. Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The socket library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

To create a socket, you must use the socket.socket() function available in socket module, which has the general syntax:

```
s = socket.socket (socket_family, socket_type, protocol=0)
```

Here is the description of the parameters:

- **socket_family**– This is either AF_UNIX or AF_INET, as explained earlier.
- **socket_type** – This is either SOCK_STREAM or SOCK_DGRAM.
- **protocol** – This is usually left out, defaulting to 0.

Once you have socket object, then you can use required functions to create your client or server program. Following is the list of functions required:

Server Socket Methods

Sr.	Method & Description
1	**s.bind()** This method binds address (hostname, port number pair) to socket.
2	**s.listen()** This method sets up and start TCP listener.
3	**s.accept()** This passively accept TCP client connection, waiting until connection arrives (blocking).

Client Socket Methods

Sr.	Method & Description
1	**s.connect()** This method actively initiates TCP server connection.

UNIT -4 (Network Programming & GUI Using Python)

General Socket Methods

Sr.	Method & Description
1	**s.recv()** This method receives TCP message
2	**s.send()** This method transmits TCP message
3	**s.recvfrom()** This method receives UDP message
4	**s.sendto()** This method transmits UDP message
5	**s.close()** This method closes socket
6	**socket.gethostname()** Returns the hostname.

Python Internet modules:

A list of some important modules in Python Network/Internet programming.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

- Knowing IP Address
- URL, Reading the Source Code of a Web Page
- Downloading a Web Page from Internet
- Downloading an Image from Internet
- A TCP/IP Server, A TCP/IP Client
- A UDP Server, A UDP Client
- File Server, File Client

Two-Way Communication between Server and Client

A Simple Server: To write Internet servers, we use the socket function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server. Now call bind(hostname, port) function to specify a port for your service on the given host. Next, call the accept method of the returned object. This method waits until a client connects to the port you specified, and then returns a connection object that represents the connection to that client.

```
import socket               # Import socket module

s = socket.socket()         # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                # Reserve a port for your service.
s.bind((host, port))        # Bind to the port
```

```
s.listen(5)                 # Now wait for client connection.
while True:
   c, addr = s.accept()     # Establish connection with client.
   print 'Got connection from', addr
c.send('Thank you for connecting')
c.close()
```

A Simple Client: Let us write a very simple client program which opens a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's socket module function. The socket.connect(hosname, port) opens a TCP connection to hostname on the port. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file. The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits:

```
import socket               # Import socket module

s = socket.socket()         # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                # Reserve a port for your service.

s.connect((host, port))
print s.recv(1024)
s.close()
```

Sending a Simple Mail: Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers. Python provides smtplib module, which defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. Here is a simple syntax to create one SMTP object, which can later be used to send an e-mail:

```
import smtplib
smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

Here is the detail of the parameters:

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

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

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

Example...

```
import smtplib

sender = 'from@fromdomain.com'
receivers = ['to@todomain.com']
```

```
message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
Subject: SMTP e-mail test

This is a test e-mail message.
"""

try:
smtpObj = smtplib.SMTP('localhost')
smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

Here, you have placed a basic e-mail in message, using a triple quote, taking care to format the headers correctly. An e-mail requires a From, To, and Subject header, separated from the body of the e-mail with a blank line. To send the mail you use smtpObj to connect to the SMTP server on the local machine and then use the sendmail method along with the message, the from address, and the destination address as parameters (even though the from and to addresses are within the e-mail itself, these aren't always used to route mail). If you are not running an SMTP server on your local machine, you can use smtplib client to communicate with a remote SMTP server.

GUI Programming

Event-driven programming paradigm: Eventually, the flow of a program depends upon the events, and programming which focuses on events is called Event-Driven programming. We were only dealing with either parallel or sequential models, but now we will discuss the asynchronous model. The programming model following the concept of Event-Driven programming is called the Asynchronous model. The working of Event-Driven programming depends upon the events happening in a program.Other than this, it depends upon the program's event loops that always listen to a new incoming event in the program. Once an event loop starts in the program, then only the events will decide what will execute and in which order.

Creating simple GUI

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:

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

UNIT -4 (Network Programming & GUI Using Python)

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 different types of widgets in Tkinter. We present these widgets as well as a brief description in the following sections:

Buttons: The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button which is called automatically when you click the button. Here is the simple syntax to create this widget:

w = Button (master, option=value, ...)

Parameters

- **master** – This represents the parent window.
- **options** – Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.	Option & Description
1	**activebackground** Background color when the button is under the cursor.
2	**activeforeground** Foreground color when the button is under the cursor.
3	**bd** Border width in pixels. Default is 2.
4	**bg** Normal background color.
5	**command** Function or method to be called when the button is clicked.
6	**fg** Normal foreground (text) color.
7	**font** Text font to be used for the button's label.
8	**height** Height of the button in text lines (for textual buttons) or pixels (for images).
9	**highlightcolor** The color of the focus highlight when the widget has focus.
10	**image** Image to be displayed on the button (instead of text).
11	**justify** How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
12	**padx** Additional padding left and right of the text.
13	**pady** Additional padding above and below the text.
14	**relief** Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
15	**state** Set this option to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
16	**underline** Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
17	**width** Width of the button in letters (if displaying text) or pixels (if displaying an image).

18	**wraplength** If this value is set to a positive number, the text lines will be wrapped to fit within this length.

Following are commonly used methods for this widget:

Sr.	Method & Description
1	**flash()** Causes the button to flash several times between active and normal colors. Leaves the button in the state it was in originally. Ignored if the button is disabled.
2	**invoke()** Calls the button's callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

Example:

```
import Tkinter
import tkMessageBox

top = Tkinter.Tk()

def helloCallBack():
tkMessageBox.showinfo( "Hello Python", "Hello World")

B = Tkinter.Button(top, text ="Hello", command = helloCallBack)
B.pack()
top.mainloop()
```

Labels: This widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time you want. It is also possible to underline part of the text (like to identify a keyboard shortcut) and span the text across multiple lines. Here is the simple syntax to create this widget:

```
w = Label ( master, option, ... )
```

Parameters:

- **master** – This represents the parent window.
- **options** – Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.	Option & Description
1	**anchor** This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
2	**bg** The normal background color displayed behind the label and indicator.
3	**bitmap** Set this option equal to a bitmap or image object and the label will display that graphic.
4	**bd** The size of the border around the indicator. Default is 2 pixels.
5	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
6	**font** If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.

7	**fg** If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
8	**height** The vertical dimension of the new frame.
9	**image** To display a static image in the label widget, set this option to an image object.
10	**justify** Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
11	**padx** Extra space added to the left and right of the text within the widget. Default is 1.
12	**pady** Extra space added above and below the text within the widget. Default is 1.
13	**relief** Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
14	**text** To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
15	**textvariable** To slave the text displayed in a label widget to a control variable of class *StringVar*, set this option to that variable.
16	**underline** You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
17	**width** Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
18	**wraplength** You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example:

```
from Tkinter import *

root = Tk()
var = StringVar()
label = Label( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

Entry: The Entry widget is used to accept single-line text strings from a user. If you want to display multiple lines of text that can be edited, then you should use the Text widget. If you want to display one or more lines of text that cannot be modified by the user, then you should use the Label widget. Here is the simple syntax to create this widget:

```
w = Entry( master, option, ... )
```

Parameters:

- **master** – This represents the parent window.
- **options** – Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.	Option & Description
1	**bg** The normal background color displayed behind the label and indicator.
2	**bd** The size of the border around the indicator. Default is 2 pixels.
3	**command** A procedure to be called every time the user changes the state of this checkbutton.
4	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
5	**font** The font used for the text.
6	**exportselection** By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0.
7	**fg** The color used to render the text.
8	**highlightcolor** The color of the focus highlight when the checkbutton has the focus.
9	**justify** If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
10	**relief** With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
11	**selectbackground** The background color to use displaying selected text.
12	**selectborderwidth** The width of the border to use around selected text. The default is one pixel.
13	**selectforeground** The foreground (text) color of selected text.
14	**show** Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
15	**state** The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
16	**textvariable** In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
17	**width** The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
18	**xscrollcommand** If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar.

Following are commonly used methods for this widget:

Sr.	Method & Description
1	**delete (first, last=None)** Deletes characters from the widget, starting with the one at index first, up to but not including the character at position last. If the second argument is omitted, only the single character at position first is deleted.

2	**get()** Returns the entry's current text as a string.
3	**icursor(index)** Set the insertion cursor just before the character at the given index.
4	**index (index)** Shift the contents of the entry so that the character at the given index is the leftmost visible character. Has no effect if the text fits entirely within the entry.
5	**insert (index, s)** Inserts string s before the character at the given index.
6	**select_adjust(index)** This method is used to make sure that the selection includes the character at the specified index.
7	**select_clear()** Clears the selection. If there isn't currently a selection, has no effect.
8	**select_from(index)** Sets the ANCHOR index position to the character selected by index, and selects that character.
9	**select_present()** If there is a selection, returns true, else returns false.
10	**select_range(start, end)** Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position.
11	**select_to(index)** Selects all the text from the ANCHOR position up to but not including the character at the given index.
12	**xview(index)** This method is useful in linking the Entry widget to a horizontal scrollbar.
13	**xview_scroll(number, what)** Used to scroll the entry horizontally. The what argument must be either UNITS, to scroll by character widths, or PAGES, to scroll by chunks the size of the entry widget. The number is positive to scroll left to right, negative to scroll right to left.

Example:

```
from Tkinter import *

top = Tk()
L1 = Label(top, text="User Name")
L1.pack( side = LEFT)
E1 = Entry(top, bd =5)
E1.pack(side = RIGHT)

top.mainloop()
```

Dialogs

Layouts (What is a layout manager?): When you create graphical interfaces, the widgets in the window must have a way to be arranged relative to each other. For example, placing widgets can be accomplished using their relative positions to other widgets or by defining their positions by specifying pixel locations. In Tkinter there are three types of layout managers -- pack, place, and grid. Each manager uses a different method to help us arrange widgets.

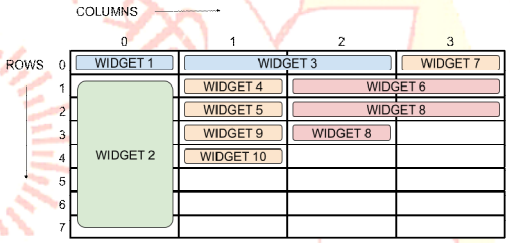
Pack Layout Manager: The simplest way to think about it is that the pack() method turns each individual widget into a block. Each widget has its own size and the pack manager fits them all together just like you would do with real blocks.

Each widget already has its own size and parameters. Of course, you can change these to better fit your needs. Once every widget's size is determined, the manager does its job to arrange them in the window.

Let's take a moment to learn some of the basics of the pack layout manager. With pack, the manager stacks widgets on top of each other vertically like blocks. Of course, you can also achieve a horizontal layout by changing the side parameter to 'left' or 'right'. You can also change the height, width, and locations of the widgets. Some of the pack manager's more useful parameters are listed below:

- **side** -- specifies the general location of the widget in the window, arguments are 'top', 'bottom', 'left', 'right' (default is 'top').
- **fill** -- which directions you want the widget to fill in the parent window, can choose 'x', 'y' directions, or 'both'.
- **padx, pady** -- the number of pixels surrounding the widget to create a padding between other widgets, for horizontal or vertical padding.
- **ipadx, ipady** -- how many pixels to use for padding inside the widget, also for horizontal or vertical padding
- **expand** -- set to True if you want the widget to stretch if the parent window expands. Default is False.
- **anchor** -- where the widget is placed in the parent widget, specified by 'n', 's', 'e', 'w', or some combination of them. Default is 'center'.

Grid Layout Manager: I personally think that using the grid layout manager is the easiest manager out of the three managers that Tkinter offers. The reason is because it works similar to a matrix, with rows and columns. The upper left-most corner has row value 0 and column value 0. So moving around the grid and arranging widgets is quite simple. If you want to place widgets in a horizontal row, then the row value stays the same and the column value increases by 1 for each widget. It is similar if you want to move down a column,



with the column value staying the same and the row value increasing. Check out the image for a visual example. Now let's look at some of the main parameters that can help you arrange widgets using the grid layout manager.

- **row, column** -- the row and column values for the location of the widget. Both start from 0.
- **columnspan, rowspan** -- specifies how many columns or rows a widget will be in. This is very useful to help get the spacing right for your widgets.
- **padx, pady** -- the number of pixels surrounding the widget to create padding between other widgets, for horizontal or vertical padding
- **ipadx, ipady** -- how many pixels to use for padding inside the widget, also for horizontal or vertical padding
- **sticky** -- specifies a value of S, N, E, W, or a combination of them, e.g. NW, NE, SW, or SE. The parameter tells which side of the "cell" the widget will "stick" to. If you use W+E+N+S, then the widget will fill up the "cell". Default is to center the widget within the "cell".

Place Layout Manager: The place layout manager allows for you to have more absolute control about the arrangement of your widgets. With place, you can specify the size of the widget, as well as the exact x and y coordinates to arrange it within the parent window. In many cases, this can prove to be easier when you are thinking about the layout of your GUI. But it also means that you may have to spend a little more time playing around with the x and y values. The place manager is most useful for arranging buttons or other smaller widgets together within a larger dialog window. A few of the parameters you can play around with are listed below.

- **in_** -- specifies the master window for the widget
- **x, y** -- specifies the specific x and y values of the widget in the parent window

- **relx, rely** -- horizontal and vertical offset relative to the size of the parent widget, values between 0.0 and 0.1
- **relwidth, relheight** -- set height and width of widget relative to the size of the parent widget, values between 0.0 and 0.1
- **anchor** -- where the widget is placed in the parent widget, specified by 'n', 's', 'e', 'w', or some combination of them. Default is 'center'

Frames: The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets. It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets. Here is the simple syntax to create this widget:

```
w = Frame ( master, option, ... )
```

Parameters:

- master – This represents the parent window.
- options – Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.	Option & Description
1	**bg** The normal background color displayed behind the label and indicator.
2	**bd** The size of the border around the indicator. Default is 2 pixels.
3	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
4	**height** The vertical dimension of the new frame.
5	**highlightbackground** Color of the focus highlight when the frame does not have focus.
6	**highlightcolor** Color shown in the focus highlight when the frame has the focus.
7	**highlightthickness** Thickness of the focus highlight.
8	**relief** With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
9	**width** The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.

Example:

```
# Import Module
from tkinter import *

# Create Tkinter Object
root = Tk()

# Set Geometry
root.geometry("400x400")

# Frame 1
frame1 = Frame(root,bg="black",width=500,height=300)
frame1.pack()
```

```
# Frame 2
frame2 = Frame(frame1,bg="white",width=100,height=100)
frame2.pack(pady=20,padx=20)

# Execute Tkinter
root.mainloop()
```