GUI

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.

- **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

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.

Tkinter Widget Classes

Widget Class	Description
Button	A simple button, used to execute a command.
Canvas	Structured graphics, used to draw graphs and plots, create graphics editors, and implement custom widgets.
Checkbutton	Clicking a check button toggles between the values.
Entry	A text entry field, also called a text field or a text box.
Frame	A container widget for containing other widgets.
Label	Displays text or an image.
Menu	A menu pane, used to implement pull-down and popup menus.

Menubutton	A menu button, used to implement pull-down menus.
Message	Displays a text. Similar to the label widget, but can automatically wrap text to a given width or aspect ratio.
Radiobutton	Clicking a radio button sets the variable to that value, and clears all other radio buttons associated with the same variable.
Text	Formatted text display. Allows you to display and edit text with various styles and attributes. Also supports embedded images and windows.

There are many options for creating widgets from these classes. The first argument is always the parent container. You can specify a foreground color, background color, font, and cursor style when constructing a widget.

First example

```
1. from tkinter import *
2. window = Tk()
3. label = Label(window, text = "Welcome to Python")
4. button = Button(window, text = "Click Me")
5. label.pack()
6. button.pack()
7. window.mainloop()
```

Whenever you create a GUI-based program in Tkinter, you need to import the tkinter module (line 1) and create a window by using the Tk class (line 2). Recall that the asterisk (*) in line 1 imports all definitions for classes, functions, and constants from the tkinter module to the program.Tk() creates an instance of a window. Label and Button are Python Tkinter widget classes for creating labels and buttons. The first argument of a widget class is always the parent container (i.e., the container in which the widget will be placed). The statement (line 3)

```
label = Label(window, text = "Welcome to Python")
```

constructs a label with the text Welcome to Python that is contained in the window. The statement (line5)

label.pack()

places label in the container using a pack manager. In this example, the pack manager packs the widget in the window row by row. For now, you can use the pack manager without knowing its full details. Tkinter GUI programming is event driven. After the user interface is displayed, the program waits for user interactions such as mouse clicks and key presses. This is specified in the following statement (line 7)

window.mainloop()

The statement creates an event loop. The event loop processes events continuously until you close the main window .

Processing events (Handling events when user clicks a button)

The Button widget is a good way to demonstrate the basics of event-driven programming, so we'll use it in the following example. When the user clicks a button, your program should process this event. You enable this action by defining a processing function and binding the function to the button,

```
from tkinter import * # Import all definitions from tkinter

class ProcessButtonEvent:
    def __init__(self):
        window = Tk()
        btOK = Button(window, text = "OK", fg = "red",command =self.processOK )
        btCancel = Button(window, text = "Cancel", bg = "yellow",  command =self.processCancel)
        btOK.pack() # Place the OK button in the window
        btCancel.pack() # Place the Cancel button in the window
        window.mainloop() # Create an event loop
    def processOK(self):
        print("OK button is clicked")
    def processCancel(self):
        print("Cancel button is clicked")
```

ProcessButtonEvent()

Example 3 : Finding sum of two numbers by taking input from text boxes.

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.**

```
from tkinter import *

root = Tk()

e1 = Entry(root)

e2 = Entry(root)

l = Label(root)

def callback():

    #total = sum(int(e.get()) for e in (e1, e2))

     a= int(e1.get())

     b= int(e2.get())

     total=a+b

     l.config(text="answer = %s" % total)

b = Button(root, text="add them", command=callback)

for widget in (e1, e2, l, b):

 widget.pack()

b.mainloop()
```

Radio button

This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.

In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another.

Syntax

Here is the simple syntax to create this widget −

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

```
from tkinter import *

def sel():

   selection = "You selected the option " + str(var.get())

   label.config(text = selection)

root = Tk()

var = IntVar()

R1 = Radiobutton(root, text="Option 1", variable=var, value=1,                command=sel)

R1.pack()

R2 = Radiobutton(root, text="Option 2", variable=var, value=2, command=sel)

R2.pack()

R3 = Radiobutton(root, text="Option 3", variable=var, value=3, command=sel)

R3.pack()

label = Label(root)

label.pack()

root.mainloop()
```

CheckBox

The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option.

You can also display images in place of text.

Syntax

Here is the simple syntax to create this widget −

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

```
from tkinter import *
master = Tk()

def var_states():
   print("male: %d,\nfemale: %d" % (var1.get(), var2.get()))

Label(master, text="Your sex:").grid(row=0, sticky=W)
var1 = IntVar()
Checkbutton(master, text="male", variable=var1).grid(row=1, sticky=W)
var2 = IntVar()
Checkbutton(master, text="female", variable=var2).grid(row=2, sticky=W)
Button(master, text='Quit', command=master.quit).grid(row=3, sticky=W, pady=4)
Button(master, text='Show', command=var_states).grid(row=4, sticky=W, pady=4)
mainloop()
```

Standard Dilog Boxes

You can use standard dialog boxes to display message boxes or to prompt the user to enter numbers and strings.

The following program invokes the showinfo, showwarning, and showerror functions to display an information message , a warning , and an error . These functions are defined in the tkinter.messagebox module . The askyesno function displays the Yes and No buttons in the dialog box . The function returns True if the Yes button is clicked or False if the No button is clicked. The askokcancel function displays the OK and Cancel buttons in the dialog box . The function returns True if the OK button is clicked or False if the Cancel button is clicked. The askyesnocancel function displays the Yes, No, and Cancelbuttons in the dialog box . The function returns True if the Yes button is clicked, False if the No button is clicked or None if the Cancel button is clicked. The askstring function returns the string entered from the dialog box after the OK button is clicked or None if the Cancel button is clicked. The askinteger function returns the integer entered from the dialog box after the OK button is clicked or None if the Cancel button is clicked. The askfloatfunction returns the float entered from the dialog box after the OK button is clicked or None if the Cancel button is clicked. All the dialog boxes are modal windows, which means that the program cannot continue until a dialog box is dismissed.

Example

```
import tkinter.messagebox
import tkinter.simpledialog
import tkinter.colorchooser
tkinter.messagebox.showwarning("showwarning", "This is a warning")
tkinter.messagebox.showerror("showerror", "This is an error")
isYes = tkinter.messagebox.askyesno("askyesno", "Continue?")
print(isYes)
isOK = tkinter.messagebox.askokcancel("askokcancel", "OK?")
print(isOK)
isYesNoCancel = tkinter.messagebox.askyesnocancel( "askyesnocancel", "Yes, No, Cancel?")
print(isYesNoCancel)
name = tkinter.simpledialog.askstring( "askstring", "Enter your name")
print(name)
age = tkinter.simpledialog.askinteger(  "askinteger", "Enter your age")
print(age)
weight = tkinter.simpledialog.askfloat(   "askfloat", "Enter your weight")
print(weight)
from tkinter import *
def donothing():
   filewin = Toplevel(root)
   button = Button(filewin, text="Do nothing button")
   button.pack()
```

Menu

Menus make selection easier and are widely used in windows. You can use the Menu class to create a menu bar and a menu, and use the add_command method to add items to the menu. The program creates a menu bar and the menu bar is added to the window. To

display the menu, use the config method to add the menu bar to the container . To create a menu inside a menu bar, use the menu bar as the parent container and invoke the menu bar's add_cascade method to set the menu label .You can then use the add_command method to add items to the menu . Note that the tearoff is set to 0, which specifies that the menu cannot be moved out of the window. If this option is not set, the menu can be moved out of the window.

Example

```
root = Tk()

menubar = Menu(root)

filemenu = Menu(menubar, tearoff=0)

filemenu.add_command(label="New", command=donothing)

filemenu.add_command(label="Open", command=donothing)

filemenu.add_command(label="Save", command=donothing)

filemenu.add_command(label="Save as...", command=donothing)

filemenu.add_command(label="Close", command=donothing)

filemenu.add_separator()

filemenu.add_command(label="Exit", command=root.quit)

menubar.add_cascade(label="File", menu=filemenu)

editmenu = Menu(menubar, tearoff=0)

editmenu.add_command(label="Undo", command=donothing)


editmenu.add_separator()

editmenu.add_command(label="Cut", command=donothing)

editmenu.add_command(label="Copy", command=donothing)

editmenu.add_command(label="Paste", command=donothing)

editmenu.add_command(label="Delete", command=donothing)

editmenu.add_command(label="Select All", command=donothing)
```

```
menubar.add_cascade(label="Edit", menu=editmenu)

helpmenu = Menu(menubar, tearoff=0)

helpmenu.add_command(label="Help Index", command=donothing)

helpmenu.add_command(label="About...", command=donothing)

menubar.add_cascade(label="Help", menu=helpmenu)

root.config(menu=menubar)

root.mainloop()
```

Canvas

You use the Canvas widget for displaying shapes. You can use the methods create_rectangle, create_oval, create_arc, create_polygon, or create_line to draw a rectangle, oval, arc, polygon, or line on a canvas.

The program displays a line, and a rectangle.

```
from tkinter import *

top = Tk()

C = Canvas(top, bg="blue", height=250, width=300)

line = C.create_line(10,10,200,200,fill='white')

rectangle = C.create_rectangle(20,20,190,90,fill='blue')

C.pack()

top.mainloop()
```

Multithreading

Running several threads is similar to running several different programs concurrently, but with the following benefits −

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.

- Threads sometimes called light-weight processes and they do not require much memory overhead; they care cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

There are two different kind of threads:

- **kernel thread**
- **user thread**

Kernel Threads are part of the operating system, while User-space threads are not implemented in the kernel.

There are two modules which support the usage of threads in Python3:

- **_thread**
- **threading**

The thread module has been "deprecated" for quite a long time. Users are encouraged to use the threading module instead. So,in Python 3 the module "thread" is not available anymore. However, it has been renamed to "_thread" for backwards compatibilities in Python3.

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 *agrs*. 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

```
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")
```

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.

```
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, self.counter, 5)
        print ("Exiting " + self.name)

def print_time(threadName, delay, counter):
    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()
thread1.join()
thread2.join()
```

```
print ("Exiting Main Thread")
```

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.

```
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")
```

What are 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.

Sockets have their own vocabulary:

domain	The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.
type	The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
protocol	Typically zero, this may be used to identify a variant of a protocol within a domain and type.
hostname	The identifier of a network interface: • A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation • A string "<broadcast>", which specifies an INADDR_BROADCAST address. • A zero-length string, which specifies INADDR_ANY, or • An Integer, interpreted as a binary address in host byte order.
port	Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

The *socket* Module

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

Method	Description
s.bind()	This method binds address (hostname, port number pair) to socket.
s.listen()	This method sets up and start TCP listener.

s.accept()	This passively accept TCP client connection, waiting until connection arrives (blocking).

Client Socket Methods

Method	Description
s.connect()	This method actively initiates TCP server connection.

General Socket Methods

Method	Description
s.recv()	This method receives TCP message
s.send()	This method transmits TCP message
s.recvfrom()	This method receives UDP message
s.sendto()	This method transmits UDP message
s.close()	This method closes socket
socket.gethostname()	Returns the hostname.

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.

```
#!/usr/bin/python3           # This is server.py file
import socket

# create a socket object
serversocket = socket.socket(
                socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = socket.gethostname()

port = 9999

# bind to the port
serversocket.bind((host, port))

# queue up to 5 requests
serversocket.listen(5)

while True:
    # establish a connection
    clientsocket,addr = serversocket.accept()

    print("Got a connection from %s" % str(addr))

    msg='Thank you for connecting'+ "\r\n"
    clientsocket.send(msg.encode('ascii'))
```

```
    clientsocket.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 −

```
#!/usr/bin/python3           # This is client.py file
import socket
# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()# get local machine name
port = 9999
s.connect((host, port)) # connection to hostname on the port.
msg = s.recv(1024) # Receive no more than 1024 bytes.
s.close()
print (msg.decode('ascii'))
```

Now run this server.py in background and then run above client.py to see the result.