Unit-V: Errors and Exceptions and Graphical User Interfaces

Errors and Exceptions:

When writing a program, errors may be encountered. There are two distinguishable kinds of errors:

1. syntax errors and

2. exceptions.

Syntax error:

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

Ex:

```
>>> if a < 3
  File "<interactive input>", line 1
  if a < 3
         ^
SyntaxError: invalid syntax
```

Here a colon is missing in the if statement. Syntax errors are easy to fix.

Exceptions:

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions. For example, dividing a number by zero or trying to write to a file which is read-only raises an exception.

When a Python script raises exception, it creates an Exception object. If the script raises an exception, it must handle the exception; otherwise, if it doesn't handle the exception the program will terminate abruptly.

Handling Exceptions:

Python handles exceptions using try and except blocks. In try block you can write the code which is suspicious to raise an exception, and in except block, you can write the code which will handle this exception.

Syntax:

```
try:
  statements

except ExceptionName:
  statements    #If there is Exception, then execute this block.
```

The try statement works as follows.

• First, the try clause (the statement(s) between the try and except keywords) is executed.

• If no exception occurs, the except clause is skipped and execution of the try statement is finished.

• If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.

• If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message.

A try clause can have any number of except clause to handle them differently, but only one will be executed in case an exception occurs.

Ex:

```
try:
   numerator=int(input("enter a value:"))
   denominator=int(input("enter a value:"))

   res=numerator/denominator
   print("result=",res)

except ArithmeticError:
   print("Division by zero is not possible")
```

```
Output 1:
enter a value:3
enter a value:6
result= 0.5
```

```
Output 2:
enter a value:3
enter a value:0
Division by zero is not possible
```

Multiple except blocks:

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed.

Syntax:

```
try:
  statements

except Exception1:
  statements    #If there is Exception1, then execute this block.

except Exception2:
  statements   #If there is Exception2, then execute this block.

  .......................
  .......................
 .......................
```

Ex:

```
try:
   numerator=int(input("enter a value:"))
   denominator=int(input("enter a value:"))

   res=numerator/denominator
   print("result=",res)

except ArithmeticError:

   print("Division by zero is not possible")

except ValueError:
   print("enter numeric values only")
```

```
Output1:

enter a value:4
enter a value:2
result= 2.0
```

```
Output2:

enter a value:3
enter a value:0
Division by zero is not possible
```

```
Output3:

enter a value:pavan
enter numeric values only
```

<u>Except clause with multiple exceptions:</u>

If you want to write a single except clause to handle multiple exceptions, this can be achieved by writing names of exception classes in except clause separated by comma.

<u>Syntax</u>:

```
 try:
   statements
 except (Exception1[, Exception2[,...ExceptionN]]]):
   statements  #If there is any exception from the given exception list,  then execute this
```

Ex:

try:
  numerator = int(input("enter a value:"))
  denominator = int(input("enter a value:"))
	Output1:
	enter a value:4
	enter a value:2
	result= 2.0

res=numerator/denominator
  print("result=",res)
 	
except (ArithmeticError,ValueError):
	Output 2:
	enter a value:pavan
	improper values entered. enter correct values

print("improper values entered. enter correct values")	Output 3: enter a value:3 enter a value:0 improper values entered. enter correct values

Except Clause with No Exceptions:

We can specify an except block without mentioning any exception name. **This type of except block must be last one if present**.

This default except block can be used along with other exception handlers, which handle some specific type of Exceptions. But the exceptions that are not handled by these specific exception handlers can be handled by this default except block.

Syntax:

```
try:
    statements
except (Exception1[, Exception2[,...ExceptionN]]]):
    statements  #If there is any exception from the given exception list, then execute this
except :
    statements     #If there is any other exception not listed above, then execute this
```

Ex:

```
try:
   numerator = int(input("enter a value:"))
   denominator = int(input("enter a value:"))

   res=numerator/denominator
   print("result=",res)



except ArithmeticError:

   print("division by zero is not possible")

except:
   print("enter numeric integer values only")
```

```
Output 1:

enter a value:3
enter a value:6
result= 0.5
```

```
Output2:

enter a value:3
enter a value:0
division by zero is not possible
```

```
Output 3:

enter a value:pavan
enter numeric integer values only
```

else clause:

The try … except statement has an optional else clause, which, when present, must follow all except clauses. The code in the else clause executes if the code in the try: block does not raise an exception.

Syntax:

```
try:

      statements

except (Exception1[, Exception2[,...ExceptionN]]]):

     statements  #If there is any exception from the given exception list, then execute this

except :

     statements     #If there is any other exception not listed above, then execute this

else:

     statements     #If there is no exception then execute this block.
```

try: fh = open("testfile", "w") fh.write("This is my test file for exception handling")	Output 1: Written content in the file successfully
except IOError: print ("Error: can\'t find file or read data") else: print ("Written content in the file successfully") fh.close()	Output 2: Error: can't find file or read data

finally clause:

The try statement in Python can have an optional finally clause. In case if there is any code which you want to be executed, whether exception occurs or not, then that code can be placed inside the finally clause. The finally clause runs whether or not the try statement produces an exception.

```
try:
   # block of code
   # this may raise an exception
finally:
   # block of code
   # this will always be executed
```

```
try:
        numerator = int(input("enter a value:"))
        denominator = int(input("enter a value:"))


        res=numerator/denominator
        print("result=",res)


except ArithmeticError:


        print("division by zero is not possible")


except:
        print("enter numeric integer values only")


finally:
        print("executing finally clause")
```

Output:

```
enter a value:3
enter a value:0
division by zero is not possible
executing finally clause
```

<u>Raising exceptions:</u>

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the **<u>keyword raise.</u>**

We can also optionally pass in value to the exception to clarify why that exception was raised

try:
 	
    a = int(input("Enter a positive integer: "))
 	
    if a <= 0:
 	
        raise ValueError("That is not a positive number!")
 	
except ValueError as ve:
 	
    print(ve)
	Output:
	
	Enter a positive integer: -4
	
	That is not a positive number!

User defined exceptions:

Python has many built-in exceptions which you can use in your program, but sometimes you may need to create custom exceptions with custom messages to serve your purpose.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from **Exception class**. This new exception can be raised, like other exceptions, using the raise statement with an optional error message.

Syntax for creating the user exception class:

```
class UserExceptionName(Exception):
   statements
```

for raising the exception

```
raise UserExceptionName("optional message")
```

Ex:

```
# defining Python user-defined exceptions
class Error(Exception):
  """Base class for other exceptions"""
  pass

class ValueTooSmallError(Error):
  """Raised when the input value is too small"""
  pass

class ValueTooLargeError(Error):
  """Raised when the input value is too large"""
  pass

# our main program
# user guesses a number until he/she gets it right
# you need to guess this number 10

number = 10
while True:
  try:
     num = int(input("Enter a number: "))
     if num < number:
        raise ValueTooSmallError
     elif num > number:
        raise ValueTooLargeError
     break
  except ValueTooSmallError:
     print("This value is small, try again!")

  except ValueTooLargeError:
     print("This value is large, try again!")

print("Congratulations! You guessed it correctly.")
```

Output:

```
Enter a number: 15
This value is large, try again!
Enter a number: 1
This value is small, try again!
Enter a number: 10
Congratulations! You guessed it correctly.
```

GRAPHICAL USER INTERFACE:

A graphical user interface allows the user to interact with the operating system and other programs using graphical elements such as icons, buttons, and dialog boxes.

For many years, the only way that the user could interact with an operating system was through a command line interface. A command line interface(or terminal based interface) typically displays a prompt, and the user types a command, which is then executed.

A graphical user interface (GUI), allows the user to interact with the operating system and other programs through graphical elements on the screen. Instead of requiring the user to type commands on the keyboard, GUIs allow the user to point at graphical elements and click the mouse button to activate them.

The Behavior of Terminal-Based Programs and GUI-Based Programs:

Terminal-Based Program:

A terminal-based program prompts users to enter successive inputs. In this, programs determine the order in which things happen . The user has no choice but to enter the data in the order that it is requested.

For example, consider the following program that calculates the area of the rectangle.

Program	Output

length=int(input("Enter the length"))
 	
breadth=int(input("Enter the breadth"))
 	
area=length*breadth
 	
print('Area of the rectangle is: {:.1f}'.format(area))
	Enter the length3
	
	Enter the breadth4
	
	Area of the rectangle is: 12.0

Here, in the terminal based program, the program prompts the user to enter the length. The user enters the length and then the program prompts the user to enter the breadth. The user enters the breadth and then the program calculates the area. The user has no choice but to enter the data in the order that it is requested.

This terminal-based user interface has several obvious effects on its users:

- The user is constrained to reply to a definite sequence of prompts for inputs. Once an input is entered, there is no way to back up and change it.

- To obtain results for a different set of input data, the user must run the program again. At that point, all of the inputs must be re-entered.

Each of these effects poses a problem for users that can be solved by converting the interface to a GUI.

GUI-Based Program:

But a GUI program allows the users to enter inputs in any order and waiting for them to press a command button or select a menu option.

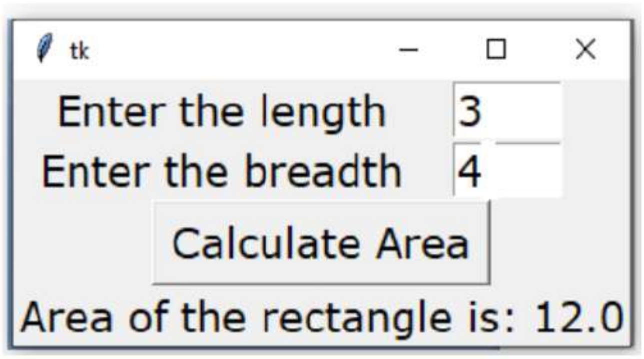
For example, the following figure shows a GUI program that calculates the area of a rectangle.

Program

```
from tkinter import *


def calculate():

    length = int(entry1.get())

    breadth = int(entry2.get())

    area=length*breadth

    output_label1.configure(text = 'Area of the rectangle is: {:.1f}'.format(area))

root = Tk()

message_label1 = Label(text='Enter the length',

                font=('Verdana', 16))

entry1 = Entry(font=('Verdana', 16), width=4)



message_label2 = Label(text='Enter the breadth',

                font=('Verdana', 16))

entry2 = Entry(font=('Verdana', 16), width=4)
```

```
calc_button = Button(text='Calculate Area', font=('Verdana', 16),
                    command=calculate)
output_label1 = Label(font=('Verdana', 16))


message_label1.grid(row=0, column=0)
message_label2.grid(row=1, column=0)


entry1.grid(row=0, column=1)
entry2.grid(row=1, column=1)


calc_button.grid(row=2, column=0,columnspan=3)
output_label1.grid(row=3, column=0, columnspan=3)


mainloop()
```

Output:



```
Area of the rectangle is: 12.0
```

The user can enter the length and the breadth in any order. If a mistake is made, the user can erase the data that was entered and retype it. When the user is ready to calculate the area, he or she clicks the Calculate Area button and the program performs the calculation.

This GUI-based user interface has several advantages on its users:

- The user is not constrained to enter inputs in a particular order. Before she presses the Compute button, she can edit any of the data in the two input fields.
- Running different data sets does not require re-entering all of the data. The user can edit just one value and press the Compute button to observe different results.

<u>Event-Driven Programming:</u>

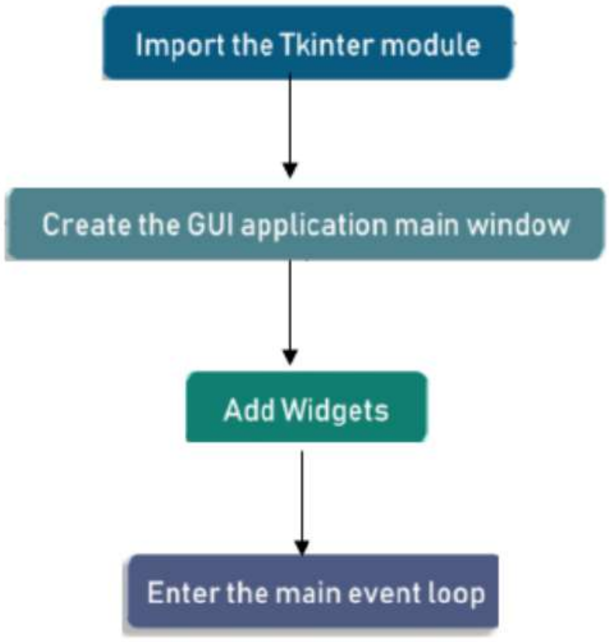
Because GUI programs must respond to the actions of the user, it is said that they are event-driven.

A GUI-based program opens a window and waits for the user to manipulate window components with the mouse. These user-generated events, such as mouse clicks, trigger operations in the program to respond by pulling in inputs, processing them, and displaying results. This type of software system is event-driven, and the type of programming used to create it is called event-driven programming.

Fundamentals of tkinter:

tkinter is actually an inbuilt **Python** module used to create simple **GUI** apps. It is the most commonly used module for **GUI** apps in the **Python**.

The following diagram shows how an application actually executes in tkinter. we first import the tkinter module. Followed by that, we create the main window. It is in this window that we are performing operations and displaying visuals and everything basically. Later, we add the widgets like labels, buttons, frames, etc. to the window and lastly we enter the main event loop.

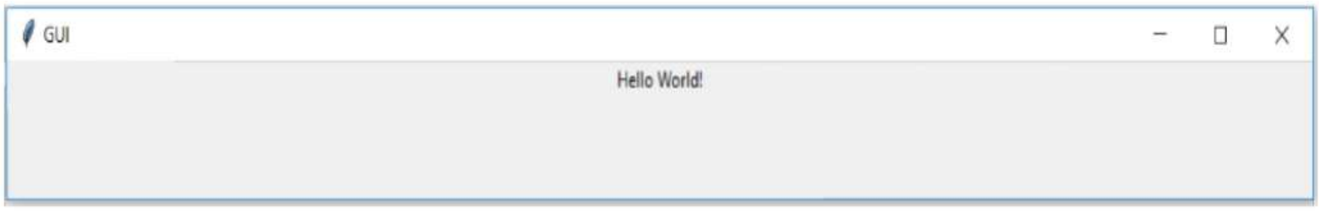

Import the Tkinter module
Create the GUI application main window
Add Widgets
Enter the main event loop

An event loop is basically telling the code to keep displaying the window until we manually close it. It runs in an infinite loop in the back-end.

Ex1:

```
import tkinter


window = tkinter.Tk( )


# to rename the title of the window

window.title("GUI")


# pack is used to show the object in the window

label = tkinter.Label(window, text = "Hello World!").pack()


window.mainloop( )
```

output:



Tkinter widgets:

There are various widgets like button, canvas, checkbutton, entry, etc. that are used to build the python GUI applications.

S.No.	widget	Description
1	Button	The Button is used to add various kinds of buttons to the python application.
2	Canvas	The canvas widget is used to draw the canvas on the window.
3	Checkbutton	The Checkbutton is used to display the CheckButton on the window.
4	Entry	The entry widget is used to display the single-line text field to the user. It is commonly used to accept user values.
5	Frame	It can be defined as a container to which, another widget can be added and organized.
6	Label	A label is a text used to display some message or information about the other widgets.
7	ListBox	The ListBox widget is used to display a list of options to the user.
8	Menubutton	The Menubutton is used to display the menu items to the user.
9	Menu	It is used to add menu items to the user.
10	Message	The Message widget is used to display the message-box to the user.
11	Radiobutton	The Radiobutton is different from a checkbutton. Here, the user is provided with various options and the user can select only one option among them.
12	Scale	It is used to provide the slider to the user.
13	Scrollbar	It provides the scrollbar to the user so that the user can scroll the window up and down.
14	Text	It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it.

14	Toplevel	It is used to create a separate window container.
15	Spinbox	It is an entry widget used to select from options of values.
16	PanedWindow	It is like a container widget that contains horizontal or vertical panes.
17	LabelFrame	A LabelFrame is a container widget that acts as the container
18	MessageBox	This module is used to display the message-box in the desktop based applications.

Python Tkinter Geometry:

The **Tkinter geometry specifies** the method by using which, the **widgets are represented on display**. The python Tkinter provides the following geometry methods.

1. The pack() method
2. The grid() method
3. The place() method

pack( ) method:

The pack( ) geometry manager is used to organize widget in the block. It can place widgets on top of each other or place them side by side.

Syntax:

widget.pack(options)

A list of possible options that can be passed in pack() is given below.

- expand − When set to True, widget expands to fill any space not otherwise used in widget's parent.

- fill − Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically).

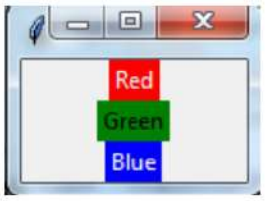
- side − Determines which side of the parent widget packs against: TOP (default), BOTTOM, LEFT, or RIGHT.

Note:

- The fill option tells the manager that the widget wants fill the entire space assigned to it. The value controls how to fill the space; BOTH means that the widget should expand both horizontally and vertically, X means that it should expand only horizontally, and Y means that it should expand only vertically.
- The expand option tells the manager to assign additional space to the widget box. If the parent widget is made larger than necessary to hold all packed widgets, any exceeding space will be distributed among all widgets that have the expand option set to a non-zero value.

```
from tkinter import *

root = Tk()

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

w.pack()

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

w.pack()

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

w.pack(fill=BOTH)

mainloop()
```

Output:



grid() method:

The grid( ) geometry manager organizes the widgets in the tabular form. We can specify the rows and columns as the options in the method call. We can also specify the column span (width) or rowspan(height) of a widget.

This is a more organized way to place the widgets to the python application. The syntax to use the grid() is given below.

Syntax:

widget.grid(options)

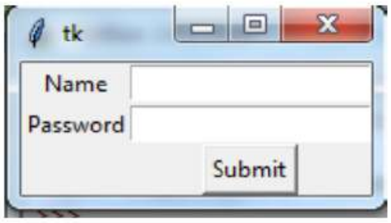
A list of possible options that can be passed inside the grid() method is given below.

- **Column**
  The column number in which the widget is to be placed. The leftmost column is represented by 0.

- **Columnspan**
  The width of the widget. It represents the number of columns up to which, the column is expanded.
- **ipadx, ipady**
  It represents the number of pixels to pad the widget inside the widget's border.
- **padx, pady**
  It represents the number of pixels to pad the widget outside the widget's border.
- **row**
  The row number in which the widget is to be placed. The topmost row is represented by 0.
- **rowspan**
  The height of the widget, i.e. the number of the row up to which the widget is expanded.
- **Sticky**
  If the cell is larger than a widget, then sticky is used to specify the position of the widget inside the cell. It may be the concatenation of the sticky letters representing the position of the widget. It may be N, E, W, S, NE, NW, NS, EW, ES.

```
from tkinter import *

parent = Tk()

name = Label(parent,text = "Name").grid(row = 0, column = 0)

e1 = Entry(parent).grid(row = 0, column = 1)

password = Label(parent,text = "Password").grid(row = 1, column = 0)

e2 = Entry(parent).grid(row = 1, column = 1)

submit = Button(parent, text = "Submit").grid(row = 2, column = 1)

parent.mainloop()
```

Output:



Place() method:

The place() geometry manager organizes the widgets to the specific x and y coordinates.

Syntax

widget.place(options)

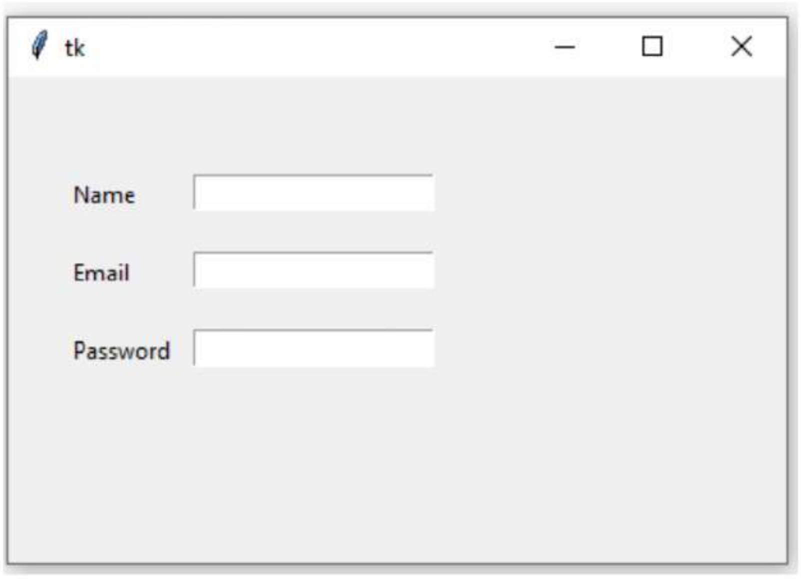
A list of possible options is given below.

- **Anchor:** It represents the exact position of the widget within the container. The default value (direction) is NW (the upper left corner)
- **bordermode:** The default value of the border type is INSIDE that refers to ignore the parent's inside the border. The other option is OUTSIDE.
- **height, width:** It refers to the height and width in pixels.
- **relheight, relwidth:** It is represented as the float between 0.0 and 1.0 indicating the fraction of the parent's height and width.
- **relx, rely:** It is represented as the float between 0.0 and 1.0 that is the offset in the horizontal and vertical direction.
- **x, y:** It refers to the horizontal and vertical offset in the pixels.

```
from tkinter import *

top = Tk()

top.geometry("400x250")

name = Label(top, text = "Name").place(x = 30,y = 50)
```

```
email = Label(top, text = "Email").place(x = 30, y = 90)

password = Label(top, text = "Password").place(x = 30, y = 130)

e1 = Entry(top).place(x = 95, y = 50)

e2 = Entry(top).place(x = 95, y = 90)

e3 = Entry(top).place(x = 95, y = 130)

top.mainloop()
```

Output:



Tkinter Button:

The Button widget is used to add buttons in a Python application. The user can click the button to cause an action to take place.

When you create a Button widget you can specify the text that is to appear on the face of the button, and the name of a callback function. A callback function is a function or method that executes when the user clicks the button.

A callback function is also known as an event handler because it handles the event that occurs when the user clicks the button.

Syntax:

Here is the simple syntax to create this widget

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

Parameters

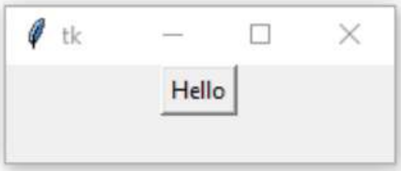
- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

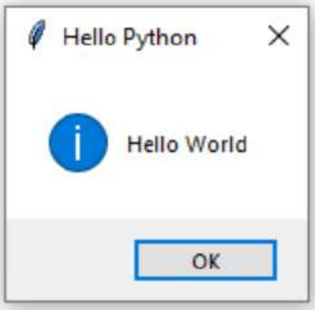
Option	Description
text	The string that should appear on the face of the button
command	Function or method to be called when the button is clicked.
activebackground	Background color when the button is under the cursor.
activeforeground	Foreground color when the button is under the cursor.
bd	Border width in pixels. Default is 2.
bg	Normal background color.
image	Image to be displayed on the button (instead of text).

Ex:

```
from tkinter import *

from tkinter import messagebox

top = Tk()

top.geometry("200x50")

def helloCallBack():

   msg=messagebox.showinfo( "Hello Python", "Hello World")

B = Button(top, text ="Hello", command = helloCallBack)

B.pack()

top.mainloop()
```

Output:





Hello World

Tkinter Entry:

An Entry widget is a rectangular area that the user can type input into. We can use the Entry widget's get method to retrieve the data that has been typed into the widget. The get method returns a string, so it will have to be converted to the appropriate data type if the Entry widget is used for numeric input.

Typically, a program will have one or more Entry widgets in a window, along with a button that the user clicks to submit the data that has typed into the Entry widgets. The button's callback function retrieves data from the window's Entry widgets and processes it.

Syntax

Here is the simple syntax to create this widget

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

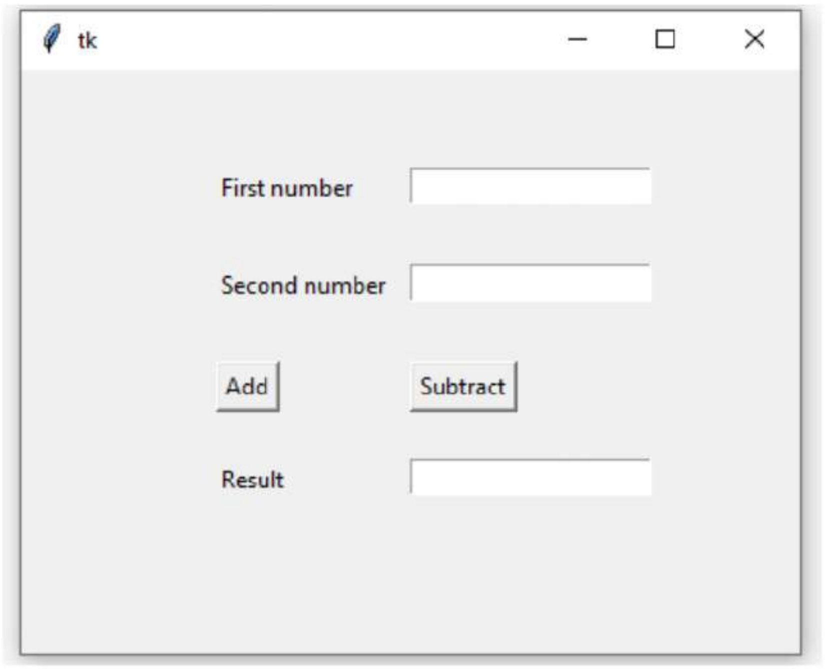
Parameters

- 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.

```
#kinter7.py
from tkinter import *
top = Tk()
top.geometry("400x300+10+10")
lbl1=Label(top, text='First number')
lbl2=Label(top, text='Second number')
lbl3=Label(top, text='Result')
t1=Entry()
t2=Entry()
t3=Entry()
lbl1.place(x=100, y=50)
t1.place(x=200, y=50)
lbl2.place(x=100, y=100)
t2.place(x=200, y=100)
def add():
   t3.delete(0, 'end')      #clear an entry box
   num1=int(t1.get())
   num2=int(t2.get())
   result=num1+num2
   t3.insert(END, str(result))  #insert into entry box
def sub():
   t3.delete(0, 'end')
   num1=int(t1.get())
   num2=int(t2.get())
   result=num1-num2
```

```
    t3.insert(END, str(result))
b1=Button(top, text='Add', command=add)
b2=Button(top, text='Subtract',command=sub)
b1.place(x=100, y=150)
b2.place(x=200, y=150)
lbl3.place(x=100, y=200)
t3.place(x=200, y=200)


top.mainloop()
```

Output:



Radio Buttons:

Radio buttons are useful when you want the user to select one choice from several possible option. A radio button may be selected or deselected. Each radio button has a small circle that appears filled in when the radio button is selected and appears empty when the radio button is deselected.

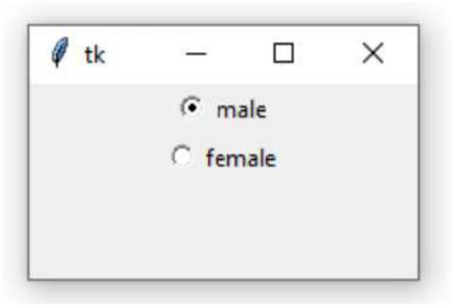
You use the tkinter module's Radiobutton class to create Radiobutton widgets. Only one of the Radiobutton widgets in a container, such as a frame, may be selected at any time. Clicking a Radiobutton selects it and automatically deselects any other Radiobutton in the same container. Because only one Radiobutton in a container can be selected at any given time, they are said to be mutually exclusive.

The tkinter module provides a class named IntVar that can be used along with Radiobutton widgets. When you create a group of Radiobuttons, you associate them all with the same IntVar object. You also assign a unique integer value to each Radiobutton widget. When one of the Radiobutton widgets is selected, it stores its unique integer value in the IntVar object.

```
from tkinter import *


def handler():
  print ("Radio button selected: " + str(var.get()))


top = Tk()


var = StringVar()
var.set('male')  #initial default selection of radiobutton
R1 = Radiobutton(top, text = "male", variable = var, value = 'male',
            command = handler)
R2 = Radiobutton(top, text = "female", variable = var, value = 'female',
            command = handler)
```

```
R1.pack()

R2.pack()



top.mainloop()
```

Output:



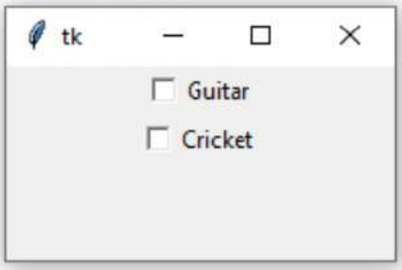
Check Buttons:

A check button appears as a small box with a label appearing next to it. check buttons may be selected or deselected. When a check button is selected, a small check mark appears inside its box. Although check buttons are often displayed in groups, the user is allowed to select **any or all of the check buttons** that are displayed in a group.

You use the tkinter module's Checkbutton class to create Checkbutton widgets. As with Radiobuttons, you can use an IntVar object along with a Checkbutton widget. Unlike Radiobuttons, however, you associate a different IntVar object with each Checkbutton. When a Checkbutton is selected, its associated IntVar object will hold the value 1. When a Checkbutton is selected, its associated IntVar object will hold the value 0.

```
from tkinter import *



def handler():

  if int(v1.get())==1:

    print ("I play Guitar. ", end='')
  if v2.get()==1:
    print ("I play Cricket. ")
  print()

top = Tk()
top.geometry("200x100+10+10")

v1 = StringVar()
v1.set(0)
v2 = IntVar()
C1 = Checkbutton(top, text = "Guitar", variable = v1, command=handler)
C2 = Checkbutton(top, text = "Cricket", variable = v2, command=handler)
C1.pack()
C2.pack()

top.mainloop()
```

Output:



Frame widget:

The frame widget is basically a container whose task is to hold other widgets and arrange them with respect to each other. 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.

Syntax

Here is the simple syntax to create this widget

w = Frame (master, options, ...)

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.

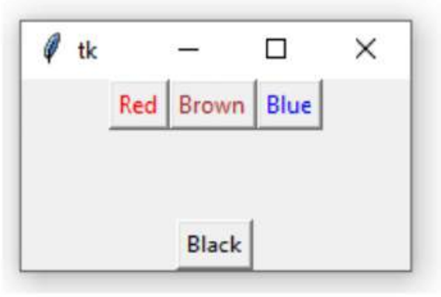
Option	Description
bd	Border width in pixels. Default is 2.
bg	Normal background color.
height	The vertical dimension of the new frame.

Ex:

```
#frames kinter11.py

from tkinter import *

root = Tk()

root.geometry("200x100+10+10")

frame = Frame(root)

frame.pack()

bottomframe = Frame(root)

bottomframe.pack( side = BOTTOM )

redbutton = Button(frame, text="Red", fg="red")
```

```
redbutton.pack( side = LEFT)

greenbutton = Button(frame, text="Brown", fg="brown")

greenbutton.pack( side = LEFT )

bluebutton = Button(frame, text="Blue", fg="blue")

bluebutton.pack( side = LEFT )

blackbutton = Button(bottomframe, text="Black", fg="black")

blackbutton.pack( side = BOTTOM)

root.mainloop()
```

Output:



Dialogs:

Many programs have dialog boxes that allow the user to pick a file to open or to save a file. To use them in Tkinter, we need the following import statement:

from tkinter.filedialog import *

The most useful dialogs are given below.

Dialog	Description
`askopenfilename`	Opens a typical file chooser dialog
`askopenfilenames`	Like previous, but user can pick more than one file
`asksaveasfilename`	Opens a typical file save dialog
`askdirectory`	Opens a directory chooser dialog

The return value of askopenfilename and asksaveasfilename is the name of the file selected. There is no return value if the user does not pick a value. The return value of askopenfilenames is a list of files, which is empty if no files are selected. The askdirectory function returns the name of the directory chosen.

There are some options you can pass to these functions. You can set initialdir to the directory you want the dialog to start in. You can also specify the file types.

Ex:

```
filename=askopenfilename( initialdir='c:\\python31\\',
                                    filetypes=[('Image files', '.jpg .png .gif'), ('All files', '*')])
```

The program that opens a file dialog and allows to select a text file. The program then displays the contents of the file in a textbox.

```
from tkinter import *

from tkinter.filedialog import *

from tkinter.scrolledtext import ScrolledText

root = Tk()

textbox = ScrolledText()

textbox.grid()

filename=askopenfilename(initialdir='D:\\python\\python notes\\2020-21\\lab programs',

                  filetypes=[('Textfile', '.txt')])


root.title(filename)

s = open(filename).read()

textbox.insert(1.0, s)

mainloop()
```