B.E (CSE) VI SEMESTER

1608PC602 – PYTHON PROGRAMMING

UNIT – IV

File and Exception Handling

File - Use to store data permanently.

Exception Handling - Use to make your program reliable and robust.

- Generally data used in program is temporary, but unless data is specifically saved otherwise data will be lost when terminated
- We can Permanently store the data in program by creating in a file and save it in a disk or some other permanent storage device.
- File can be transported and can be read later by the program.

So here we are going to read and write data from and to a file.

1. Text input and output

To read and write data from and to a file, we have to use open function to create a file objects and use the objects to read and write method to read and write data. File is placed in a directory of the file system. An absolute filename contains a filename with its complete path and drive letter. For example, C:\pybook\Scores.txt is the absolute filename for the file Scores.txt on the Windows operating system. Here, c:\pybook is referred to as the directory path to the file.

Absolute filenames are machine dependent. On the UNIX platform, the absolute filename may be /home/liang/pybook/Scores.txt, where /home/liang/pybook is the directory path to the file Scores.txt.

A relative filename is relative to its current working directory. The complete directory path for a relative file name is omitted. For example, Scores.py is a relative filename. If its current working directory is c:\pybook, the absolute filename would be c:\pybook\Scores.py.

Files can be classified into text or binary files. A file that can be processed (that is, read, created, or modified) using a text editor such as Notepad on

Windows or vi on UNIX is called a text file. All the other files are called binary files. For example, Python source programs are stored in text files and can be processed by a text editor, but Microsoft Word files are stored in binary files and are processed by the Microsoft Word program. **text file** as consisting of a sequence of characters and **a binary file** as consisting of a sequence of bits. Characters in a text file are encoded using a character encoding scheme such as ASCII and Unicode. For example, the decimal integer 199 is stored as the sequence of the three characters 1, 9, and 9, in a text file, and the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals hex C7 The advantage of binary files is that they is that they are more efficient to process than text files.

1.1 Opening a File

How do you write data to a file and read the data back from a file. We need to first create a file object that is associated with a physical file. This is called opening a file. The syntax for

opening a file is: fileVariable = open(filename, mode)

The open function returns a file object for filename. The mode parameter is a string that specifies how the file will be used (for reading or writing) is shown here :

"r" Opens a file for reading.

"w" Opens a new file for writing. If the file already exists, its old contents are destroyed.

"a" Opens a file for appending data from the end of the file.

"rb" Opens a file for reading binary data.

"wb" Opens a file for writing binary data.

For example, the following statement opens a file named ABC.txt in the current directory for reading:

```
input = open("ABC.txt", "r")
```

You can also use the absolute filename to open the file in Windows, as follows:

```
input = open(r"c:\pybook\ABC.txt", "r")
```

The statement opens the file ABC.txt that is in the c:\pybook directory for reading. The **r** prefix before the absolute filename specifies that the string is a raw string, which causes the Python interpreter to treat backslash characters as literal backslashes. Without the r prefix, we would have to write the statement using an escape sequence as:

```
input = open("c:\\pybook\\ABC.txt", "r")
```

1.2 Writing Data

The open function creates a file object, which is an instance of the _io.TextIOWrapper class. In io,TextIOWrapper contain the following :

read([number.int): str	Returns the specified number of characters from the file. If the argument is omitted, the entire remaining contents in the file are read.
readline(): str	Returns the next line of the file as a string.
readlines(): list	Returns a list of the remaining lines in the file.
write(s: str): None	Writes the string to the file.
close(): None	Closes the file.

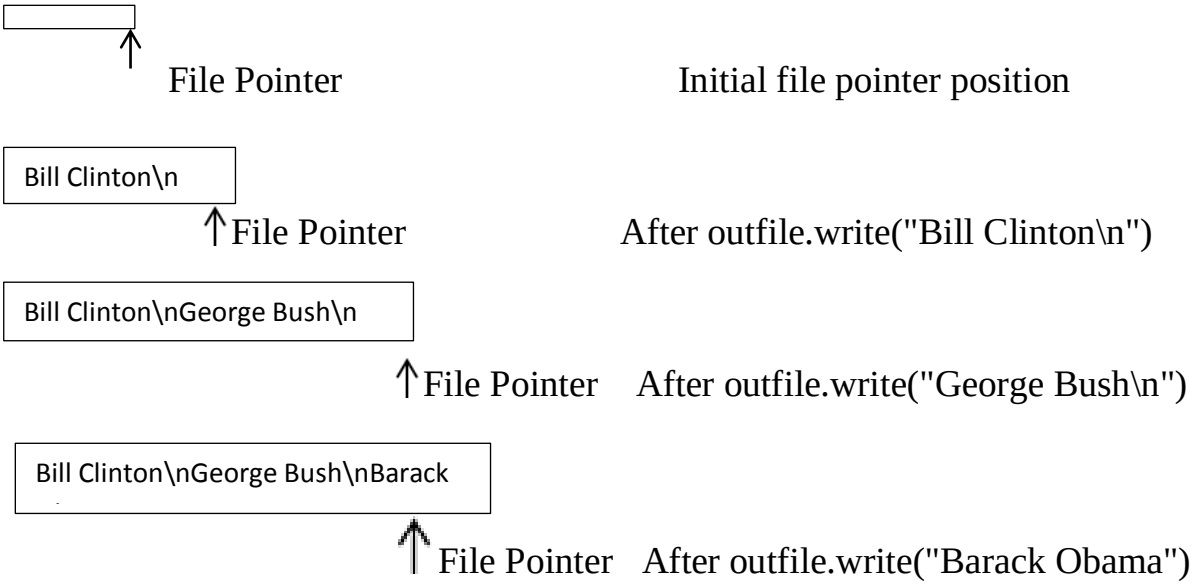
This class contains the methods for reading and writing data and for closing the file. After a file is opened for writing data, we can use the write method to write a string to the file. The simple program writes three strings to the file Presidents.txt.

WriteDemo.py

```
def main():
# Open file for output
outfile = open("Presidents.txt", "w")
# Write data to the file
  outfile.write("Bill Clinton\n")
  outfile.write("George Bush\n")
  outfile.write("Barack Obama")
  outfile.close() # Close the output file
```

main() # Call the main function

The program opens a file named Presidents.txt using the w mode for writing data. If the file does not exist, the open function creates a new file. If the file already exists, the contents of the file will be overwritten with new data. We can now write data to the file. When a file is opened for writing or reading, a special marker called a file pointer is positioned internally in the file. A read or write operation takes place at the pointer's location. When a file is opened, the file pointer is set at the beginning of the file. When you read or write data to the file, the file pointer moves forward. The program invokes the write method on the file object to write three strings . The position of the file pointer after each write.



1.3 Reading Data

After a file is opened for reading data, we can use the read method to read a specified number of characters or all characters from the file and return them as a string, the readline() method to read the next line, and the readlines() method to read all the lines into a list of strings. Suppose the file Presidents.txt contains the three lines as shown previously. The program that reads the data from the file is shown here.

```
def main():
 # Open file for input                                open file for reading
infile = open("Presidents.txt", "r")
 print("(1) Using read(): ")
 print ( infile.read() )                                    read all data
 infile.close() # Close the input file                      close file

 # Open file for input                                open file for reading
 infile = open("Presidents.txt", "r")
 print("\n(2) Using read(number): ")                        read characters
 s1 = infile.read(4)
 print(s1)
 s2 = infile.read(10)                                       raw strings
 print(repr(s2))
 infile.close() # Close the input file

 # Open file for input
infile = open("Presidents.txt", "r")
print("\n(3) Using readline(): ")                   read a line
line1 =  infile.readline()
line2 = infile.readline()
line3 = infile.readline()
line4 = infile.readline()
print(repr(line1))
print(repr(line2))
print(repr(line3))
```

```
 print(repr(line4))
 infile.close() # Close the input file

# Open file for input
infile = open("Presidents.txt", "r")
print("\n(4) Using readlines(): ")
 print(infile.readlines())
 infile.close() # Close the input file

 main() # Call the main function
```

Output :

```
(1) Using read():
Bill Clinton
George Bush
Barack Obama
(2) Using read(number):
Bill
' Clinton\nG'
(3) Using readline():
'Bill Clinton\n'
'George Bush\n'
'Barack Obama'
''
(4) Using readlines():
['Bill Clinton\n', 'George Bush\n', 'Barack Obama']
```

1.4 Reading All Data from a File

Programs often need to read all data from a file. Here are two common approaches to accomplishing this task:

1. Use the read() method to read all data from the file and return it as one string.

2. Use the readlines() method to read all data and return it as a list of strings.

These two approaches are simple and appropriate for small files, but what happens if the file is so large that its contents cannot be stored in the memory? We can write the following loop to read one line at a time, process it, and continue reading the next line until it reaches the end of the file:

```
line = infile.readline() # Read a line
while line != ' ': # Process the line here ...
# Read next line
line = infile.readline()
```

Python also read all lines by using a for loop, as follows:

```
for line in infile:   # Process the line here ...
```

This is much simpler than using a while loop.

Here is an simple example illustrates a program that copies data from a source file to a target file and counts the number of lines and characters in the file.

CopyFile.py

```
import os.path
import sys
def main():
# Prompt the user to enter filenames
f1 = input("Enter a source file: ").strip()
f2 = input("Enter a target file: ").strip()
```

```
 # Check if target file exists
  if : os.path.isfile(f2) :
 print(f2 + " already exists")
         sys.exit()
 # Open files for input and output
 infile = open(f1, "r")
outfile = open(f2, "w")
# Copy from input file to output file
countLines = countChars = 0
 for line in infile:
 countLines += 1
 countChars += len(line)
 outfile.write(line)
 print(countLines, "lines and", countChars, "chars copied")
 infile.close() # Close the input file
 outfile.close() # Close the output file
 main() # Call the main function
```

Output :

```
Enter a source file: input.txt
Enter a target file: output1.txt
output1.txt already exists
output1.txt
```

```
Enter a source file: input.txt
Enter a target file:output2.txt
3 lines and 73 characters copied
```

1.5.Appending Data

We can use the a mode to open a file for appending data to the end of an existing file. Example of appending two new lines into a file named Info.txt.

```
appendDemo.py
 def main():
 # Open file for appending data
 outfile = open("Info.txt", "a")
outfile.write("\nPython is interpreted\n")
outfile.close()  # Close the file
 main() # Call the main function
```

2. File Dialogs

The tkinter.filedialog module contains the functions askopenfilename and asksaveasfilename for displaying the file Open and Save As dialog boxes. Tkinter provides the tkinter.filedialog module with the following two functions:

```
# Display a file dialog box for opening an existing file
    filename = askopenfilename()
# Display a file dialog box for specifying a file for saving data
    filename = asksaveasfilename()
```

Both functions return a filename. If the dialog is cancelled by the user, the function returns None. Here is an example of using these two functions:

```
from tkinter.filedialog import askopenfilename
from tkinter.filedialog import asksaveasfilename
filenameforReading = askopenfilename()          file dialog for opening
print("You can read from " + filenameforReading)
filenameforWriting = asksaveasfilename()        file dialog for saving
print("You can write data to " + filenameforWriting)
```

When you run this code, the askopenfilename() function displays the Open dialog box for specifying a file to open, as shown in Figure 2.2. The asksaveasfilename() function displays the Save As dialog for specifying the name of the file to save, as shown in Figure 2.1.

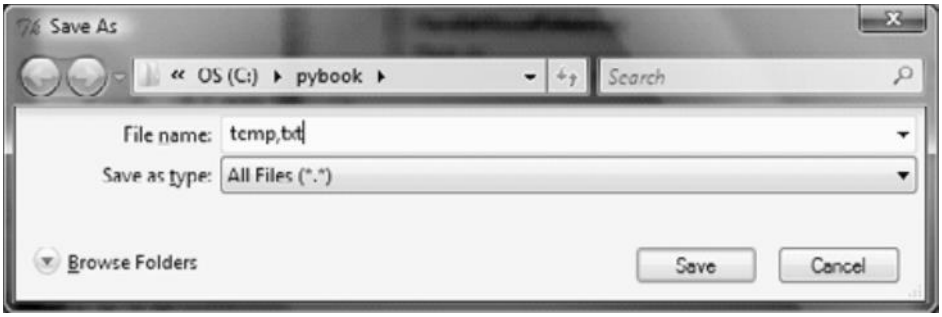


Fig 2.1 The asksaveasfilename() function displays the Save As dialog .

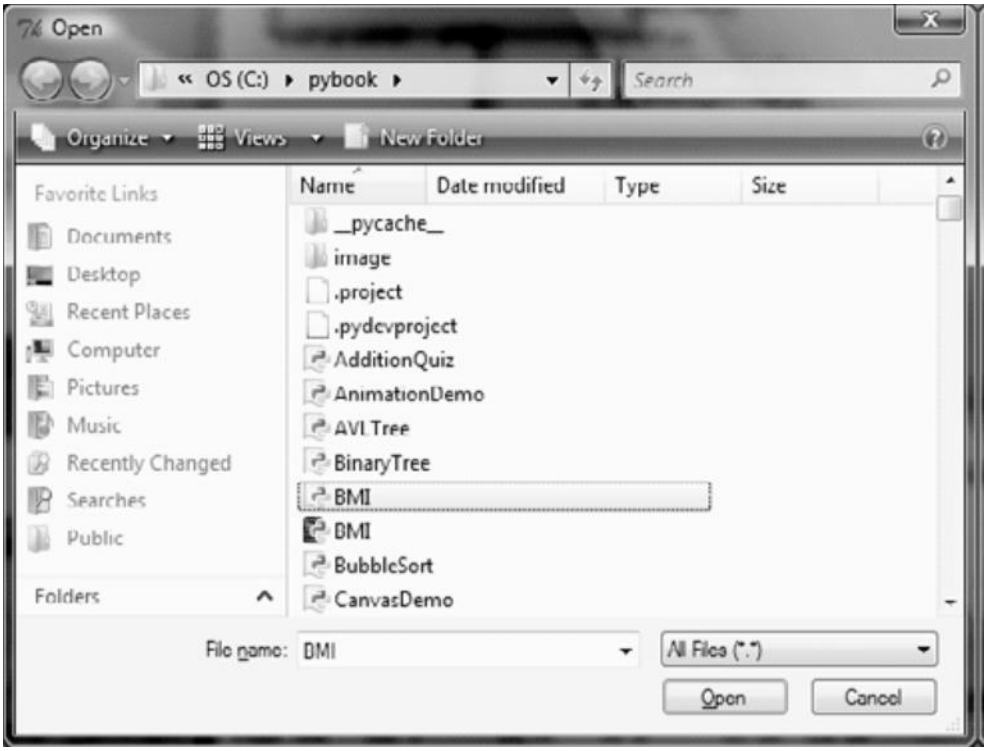


Fig 2.2 The askopenfilename() function displays the Open dialog

Now let's create a simple text editor that uses menus, toolbar buttons, and file dialogs, as shown Figure 2.3. The editor enables the user to open and save text files shows the program below.

FileEditor.py

```
from tkinter import *
from tkinter.filedialog import askopenfilename
from tkinter.filedialog import asksaveasfilename
class FileEditor:
def __init__(self):
        window = Tk()
        window.title("Simple Text Editor")
# Create a menu bar
        menubar = Menu(window)
        window.config(menu = menubar) # Display the menu bar

# Create a pull-down menu and add it to the menu bar
        operationMenu = Menu(menubar, tearoff = 0)
        menubar.add_cascade(label = "File", menu = operationMenu)
        operationMenu.add_command(label = "Open", command
                = self.openFile )
        operationMenu.add_command(label = "Save", command =
                self.saveFile)

# Add a tool bar frame
        frame0 = Frame(window) # Create and add a frame to window
        frame0.grid(row = 1, column = 1, sticky = W)
```

```
# Create images
  openImage = PhotoImage(file = "image/open.gif")
  saveImage = PhotoImage(file = "image/save.gif")
  Button(frame0, image = openImage, command =
  self.openFile).grid(row = 1, column = 1, sticky  = W)
  Button(frame0, image = saveImage, command =
          self.saveFile).grid(row = 1, column = 2)
  frame1 = Frame(window) # Hold editor pane
  frame1.grid(row = 2, column = 1)
  scrollbar = Scrollbar(frame1)
  scrollbar.pack(side = RIGHT, fill = Y)
  self.text = Text(frame1, width = 40, height = 20, wrap = WORD,
                   yscrollcommand = scrollbar.set)
  self.text.pack()
  scrollbar.config(command = self.text.yview)
  window.mainloop() # Create an event loop


def openFile(self):
filenameforReading = askopenfilename()
infile = open(filenameforReading, "r")
self.text.insert(END, infile.read()) # Read all from the file
 infile.close() # Close the input file


def saveFile(self):
 filenameforWriting = asksaveasfilename()
 outfile = open(filenameforWriting, "w")
```

```
# Write to the file
 outfile.write(self.text.get(1.0, END))
 outfile.close() # Close the output file


 FileEditor()# Create GUI
```

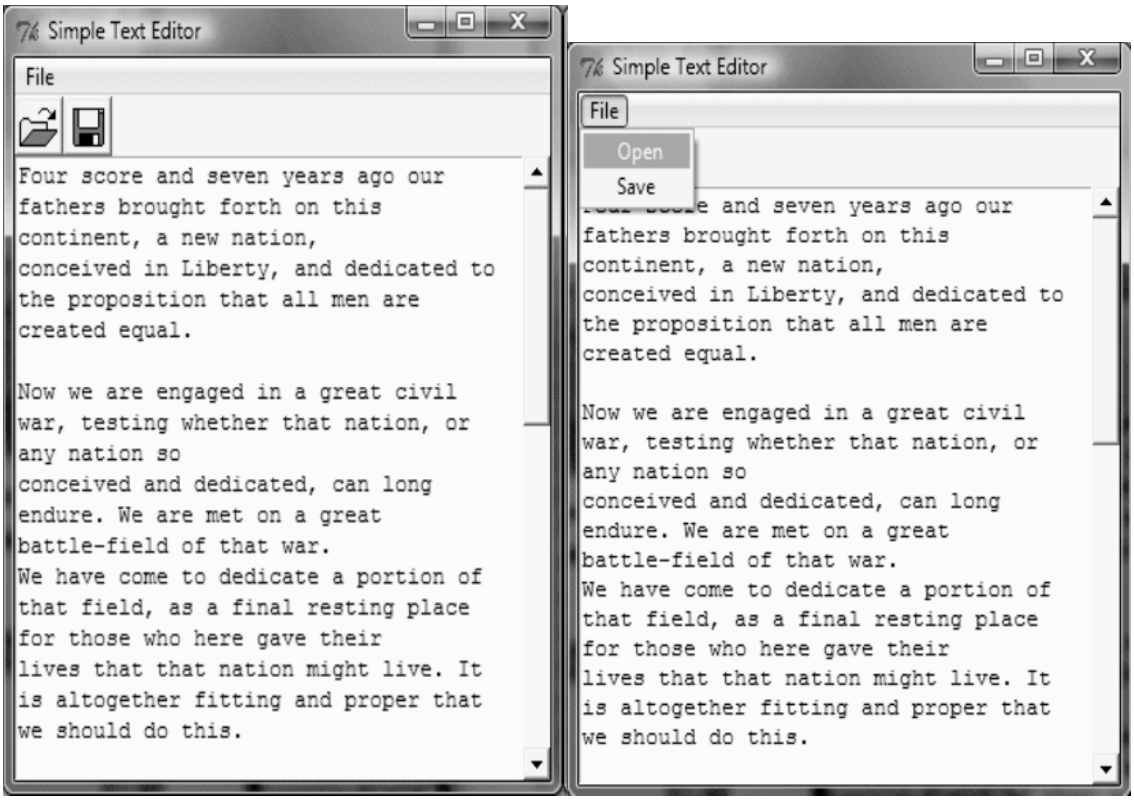


Fig 2.3 The editor enables you to open and save files from the File menu or from the toolbar

3. Exception handling

Exception handling enables a program to deal with exceptions and continue its normal execution. When running the programs in the previous sections, what happens if the user enters a file or a URL that does not exist? The program would be aborted and raise an error. For example, if you try to run by entering a nonexistent filename, the program would report this IOError:

```
c:\pybook\python CountEachLetter.py
Enter a filename: NonexistentOrIncorrectFile.txt
Traceback (most recent call last):
File "C:\pybook\CountEachLetter.py", line 23, in <module>
main()
File "C:\pybook\CountEachLetter.py", line 4, in main
infile = open(filename, "r") # Open the file
IOError: [Errno 22] Invalid argument: 'NonexistentOrIncorrectFile.txt\r'
NonexistentOrIncorrectFile.txt
```

The lengthy error message is called a **stack traceback** or **traceback**. The traceback gives information on the statement that caused the error by tracing back to the function calls that led to this statement. The line numbers of the function calls are displayed in the error message for tracing the errors.

An error that occurs at runtime is also called an **exceptio**n. An exception the program can catch the error and prompt the user to enter a correct filename? This can be done using Python's exception handling syntax.

The syntax for exception handling is to wrap the code that might raise (or throw) an exception in a try clause, as follows:

```
try:
<body>
except <ExceptionType>:
<handler>
```

Here, **<body>** contains the code that may raise an exception. When an exception occurs, the rest of the code in <body> is skipped. If the exception matches an exception type, the corresponding handler is executed. **<handler>** is the code that processes the exception. Now we see a simple exception handling program in that user enter a new filename if the input is incorrect it is shown error.

```
def main():
while True:
    try:
        filename = input("Enter a filename: ").strip()
        infile = open(filename, "r") # Open the file
        break
    except IOError:
        print("File " + filename + " does not exist. Try again")
```

The try/except block works as follows:

- First, the statements in the body between try and except are executed.
- If no exception occurs, the except clause is skipped. In this case, the break statement is executed to exit the while loop.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. In this case, if the file does not exist, the open function raises an exception and the break statement is skipped.
- When an exception occurs, if the exception type matches the exception name after the except keyword, the except clause is executed, and then the execution continues after the try statement.
- If an exception occurs and it does not match the exception name in the except clause, the exception is passed on to the caller of this function; if no handler is found, it is an unhandled exception and execution stops with an error message displayed.

A try statement can have more than one except clause to handle different exceptions. The statement can also have an optional else and/or finally statement, in a syntax like this:

```
try:
    <body>
except <ExceptionType1>:
    <handler1>.
```

```
except <ExceptionTypeN>:
 <handlerN>
 except:
        <handlerExcept>
 else:
        <process_else>
 finally:
        <process_finally>
```

The multiple excepts are similar to elifs. When an exception occurs, it is checked to match an exception in an except clause after the try clause sequentially. If a match is found, the handler for the matching case is executed and the rest of the except clauses are skipped. Note that the <ExceptionType> in the last except clause may be omitted. If the exception does not match any of the exception types before the last except clause , the <handlerExcept> for the last except clause is executed. A try statement may have an optional else clause, which is executed if no exception is raised in the try body.

A try statement may have an optional finally clause, which is intended to define cleanup actions that must be performed under all circumstances, an example of using exception handling is given below :

```
def main():
        try :
number1, number2 = eval( input("Enter two numbers, separated by a
                                        comma: "))
result = number1 / number2
 print("Result is", result)
 except ZeroDivisionError:
 print("Division by zero!")
except SyntaxError:
```

```
 print("A comma may be missing in the input")
except:
print("Something wrong in the input")
else:
print("No exceptions")
finally:
        print("The finally clause is executed")
main() # Call the main function
```

Output :

```
Enter two numbers, separated by a comma: 3,4
Result is 0.75
No exceptions
The finally clause is executed
```

```
Enter two numbers, separated by a comma: 2,0
Division by zero!
The finally clause is executed
```

```
Enter two numbers, separated by a comma: 2,3
A comma may be missing in the input
The finally clause is executed
```

```
Enter two numbers, separated by a comma: a,v
Something wrong in the input
The finally clause is executed
```

4. **Raising Exception :**

Exceptions are wrapped in objects, and objects are created from classes. An exception is raised from a function. Previous we learned how to write the code to handle exceptions. He we are going to see where does an exception come from ? How is an exception created? -- The information pertaining to

an exception is wrapped in an object. An exception is raised from a function. When a function detects an error, it creates an object from an appropriate exception class and throws the exception to the caller of the function, using the following syntax:

```
raise ExceptionClass("Something is wrong")
```

Suppose the program detects that an argument passed to a function violates the function's contract; for example, the argument must be nonnegative, but a negative argument is passed. The program can create an instance of RuntimeError and raise the exception, as follows:

```
ex = RuntimeError("Wrong argument")
raise ex
```

Or, we can combine the preceding two statements in one like this:

```
raise RuntimeError("Wrong argument")
```

Now we see an example for raise a RuntimeError exception if the radius is negative.

```
from GeometricObject import GeometricObject
import math
class Circle(GeometricObject):
 def __init__(self, radius):
        super().__init__()
        self.setRadius(radius)
 def getRadius(self):
    return self.__radius
 def setRadius(self, radius):
    if radius < 0:
        raise RuntimeError("Negative radius")
    else:
        self.__radius = radius
```

```
def getArea(self):
  return self.__radius * self.__radius * math.pi
def getDiameter(self):
  return 2 * self.__radius
def getPerimeter(self):
  return 2 * self.__radius * math.pi
def printCircle(self):
  print(self.__str__() + " radius: " + str(self.__radius))
 try:
      c1 = Circle(5)
      print("c1's area is", c1.getArea())
      c2 = Circle(-5)
      print("c2's area is", c2.getArea())
      c3 = Circle(0)
      print("c3's area is", c3.getArea())
except RuntimeException:
        print("Invalid radius")
```

Output:

```
c1's area is 78.53981633974483
Invalid radius
```

Now you know how to raise exceptions and how to handle exceptions. The benefits of using exception handling are it enables a function to throw an exception to its caller. The caller can handle this exception. Without this capability, the called function itself must handle the exception or terminate the program. Often the called function does not know what to do in case of an error. This is typically the case for library functions. The library function can detect the error, but only the caller knows what needs to be done when an error occurs. The essential benefit of exception handling is to separate the

detection of an error (done in a called function) from the handling of an error (done in the calling method). Many library functions raise exceptions, such as ZeroDivisionError, TypeError, and IndexError. You can use the try-except syntax to catch and process the exceptions. Functions may invoke other functions in a chain of function calls. Consider an example involving multiple function calls. Suppose the main function invokes function1, function1 invokes function2, function 2 invokes function3, and function3 raises an exception, as shown figure 4.1

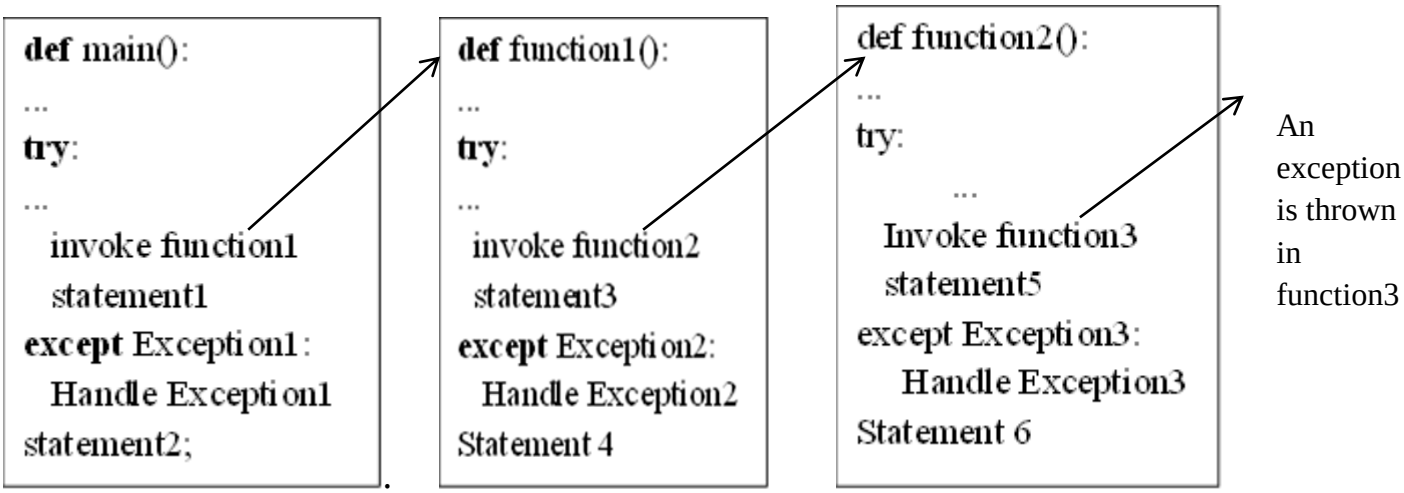


Fig 4.1 If an exception is not caught in the current function, it is passed to its caller. The process is repeated until the exception is caught or passed to the main function

Consider the following scenario:

- If the exception type is Exception 3, it is caught by the except block for handling this exception in function 2. Statement 5 is skipped, and statement 6 is executed.

- If the exception type is Exception 2, function 2 is aborted, the control is returned to function 1, and the exception is caught by the except block for handling Exception 2 in function 1. Statement 3 is skipped, and statement 4 is executed.
- If the exception type is Exception 1, function 1 is aborted, the control is returned to the main function, and the exception is caught by the except block for handling Exception 1 in the main function. Statement 1 is skipped, and statement 2 is executed.
- If the exception is not caught in function 2, function 1, or main, the program terminates, and statement1 and statement 2 are not executed.

5. Processing Exception Using Exception Objects

You can access an exception object in the except clause. As stated earlier, an exception is wrapped in an object. To throw an exception, we have to first create an exception object and then use the raise keyword to throw it. Whetehr this exception object be accessed from the except clause? Yes. You can use the following syntax to assign the exception object to a variable:

```
try
<body>
<handler>
```

With this syntax, when the except clause catches the exception, the exception object is assigned to a variable named ex. You can now use the object in the handler.

Here we have shown an example for Exception objects. In the prompts the user have to enter a number and displays the number if the input is correct. Otherwise, the program displays an error message.

```
try:
        number = eval(input("Enter a number: "))
        print("The number entered is", number)
```

except NameError as :

print("Exception:", ex)

Output :

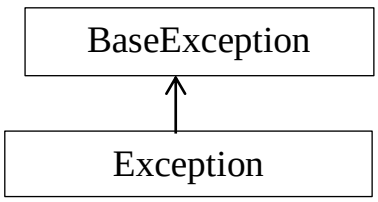
```
Enter a number: 34
The number entered is 34
```

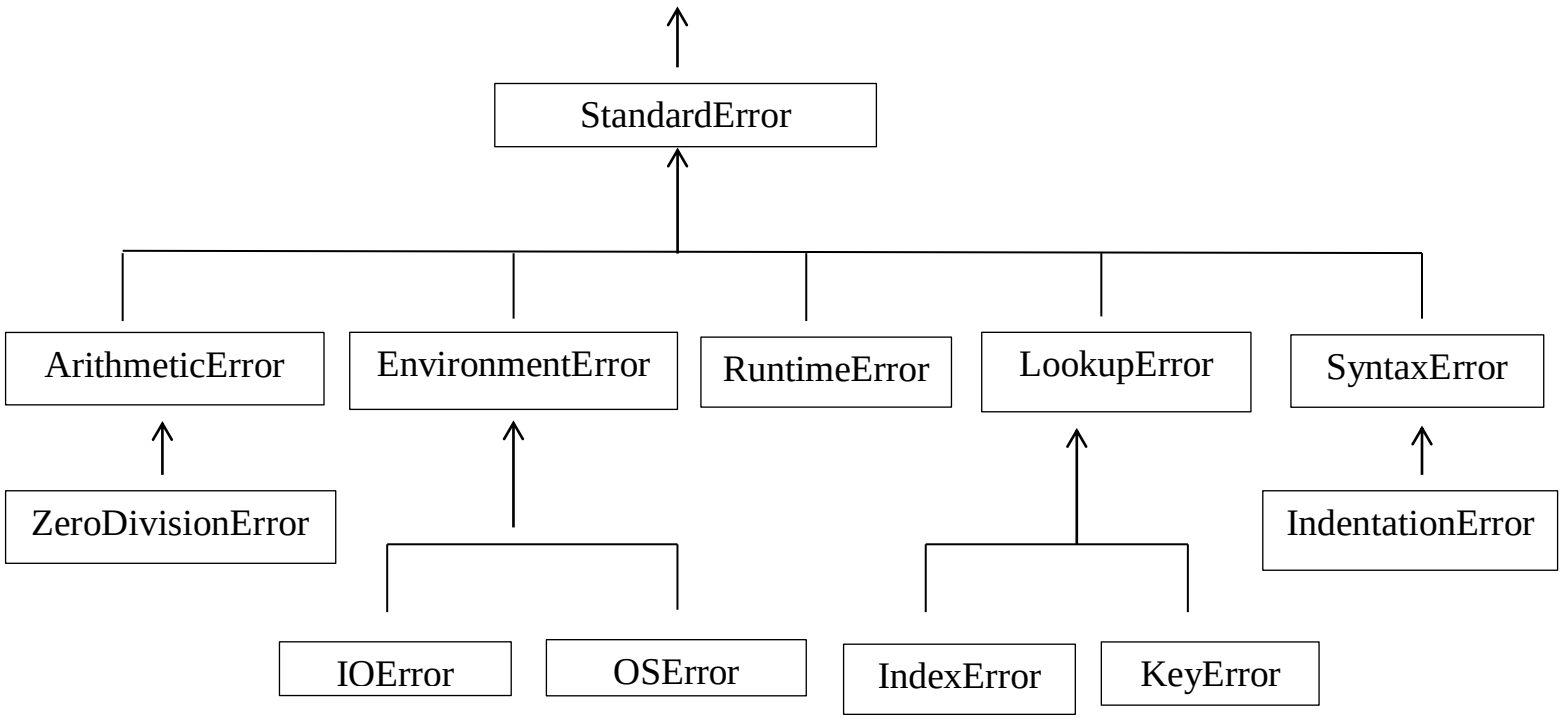
```
Enter a number: one
Exception: name 'one' is not defined
```

When we enter a nonnumeric value, an object of NameError is thrown. This object is assigned to variable ex. So, you can access it to handle the exception. The __str__() method in ex is invoked to return a string that describes the exception. In this case the string is name 'one' is not defined.

6. Defining Custom Exception Classes

We can define a custom exception class by extending BaseException or a subclass of BaseException. So far we have used Python's built-in exception classes such as ZeroDivisionError, SyntaxError, RuntimeError, and NameError Here we are going see other types of exceptions. Python has many more built-in exceptions. The BaseException class is the root of exception classes. All Python exception classes inherit directly or indirectly from BaseException. As we have see, Python provides quite a few exception classes. We can also define our own exception classes, derived from BaseException or from a subclass of BaseException, such as RuntimeError.





The setRadius method in the Circle class throws a RuntimeError exception if the radius is negative. The caller can catch this exception, but the caller does not know what radius caused this exception. To fix this problem, you can define a custom exception class to store the radius. as shown below.

```
class InvalidRadiusException(RuntimeError) :
        def __init__(self, radius):
            super().__init__()
            self.radius = radius
```

This custom exception class extends RuntimeError. The initializer simply invokes the superclass's initializer and sets the radius in the data field. Now let's modify the setRadius(radius) method in the Circle class to raise an InvalidRadiusException if the radius is negative, as shown below :

```
from GeometricObject import GeometricObject
from InvalidRadiusException import InvalidRadiusException
import math
class Circle(GeometricObject):
```

```
    def __init__(self, radius):
            super().__init__()
             self.setRadius(radius)
  def getRadius(self):
            return self.__radius
def setRadius(self, radius):
 if radius >= 0:
      self.__radius = radius
    else :
       raise InvalidRadiusException(radius)
def getArea(self):
  return self.__radius * self.__radius * math.pi
def getDiameter(self):
   return 2 * self.__radius
def getPerimeter(self):
   return 2 * self.__radius * math.pi
def printCircle(self):
  print(self.__str__(), "radius:", self.__radius)
try:
  c1 = Circle(5)
  print("c1's area is", c1.getArea())
 c2 = Circle(-5)
 print("c2's area is", c2.getArea())
 c3 = Circle(0)
  print("c3's area is", c3.getArea())
except InvalidRadiusException as ex:
```

```
        print("The radius", , "is invalid")
except Exception:
        print("Something is wrong")
```

Output :

```
c1's area is 78.53981633974483
The radius -5 is invalid
```

When creating a Circle object with a negative radius, an InvalidRadiusException is raised. The exception is caught in the except clause. The order in which exceptions are specified in except blocks is important, because Python finds a handler in this order. If an except block for a superclass type appears before an except block for a subclass type, the except block for the subclass type will never be executed. Thus, it would be wrong to write the code as follows

```
    try:
        ....
except Exception:
        print("Something is wrong")
except InvalidRadiusException:
        print("Invalid radius")
```

7. Binary I/O Using Pickling

To perform binary IO using pickling, open a file using the mode rb or wb for reading binary or writing binary and invoke the pickle module's dump and load functions to write and read data. We can write strings and numbers to a file and also we can write any object such as a list directly to a file. This would require binary IO. There are many ways to perform binary IO in Python. This section introduces binary IO using the dump and load functions in the pickle module. The Python pickle module implements the powerful

and efficient algorithms for serializing and deserializing objects. Serializing is the process of converting an object into a stream of bytes that can be saved to a file or transmitted on a network. Deserializing is the opposite process that extracts an object from a stream of bytes. Serializing/deserializing is also known as pickling/unpickling or dumping/loading objects in Python.

7.1 Dumping and Loading Objects

As we know, all data in Python are objects. The pickle module enables as to write and read any data using the dump and load functions. We have demonstrates these functions below :

```
import pickle
def main():
  # Open file for writing binary
      outfile = open("pickle.dat", "wb")
       pickle.dump(45, outfile)
      pickle.dump(56.6, outfile)
      pickle.dump("Programming is fun", outfile)
      pickle.dump([1, 2, 3, 4], outfile)
      outfile.close() # Close the output file

  # Open file for reading binary
      infile = open("pickle.dat", "rb")
      print(pickle.load(infile))
      print(pickle.load(infile))
      print(pickle.load(infile))
       print(pickle.load(infile))
      infile.close() # Close the input file

main() # Call the main function
```

Output :

```
45
56.6
Programming is fun
[1, 2, 3, 4]
```

To use pickle, you need to import the pickle module. To write objects to a file, open the file using the mode wb for writing binary and use the dump(object) method to write the object into the file. This method serializes the object into a stream of bytes and stores them in the file. The program closes the file and opens it for reading binary. The load method is used to read the objects. This method reads a stream of bytes and deserializes them into an object.

7.2 Detecting the End of File

If we don't know how many objects are in the file, then how do weu read all the objects. We can repeatedly read an object using the load function until it throws an EOFError (end of file) exception. When this exception is raised, catch it and process it to end the file-reading process. The program shown below are stores an unspecified number of integers in a file by using object IO, and then it reads all the numbers back from the file.

```
import pickle
 def main():
   # Open file for writing binary
       outfile = open("numbers.dat", "wb")
       data = eval(input("Enter an integer (the input exits " +  "if the
                 input is 0): "))
while data != 0:
        pickle.dump(data, outfile)
```

```
            data = eval(input("Enter an integer (the input exits " +
                              "if the input is 0): "))
            outfile.close() # Close the output file
        # Open file for reading binary
            infile = open("numbers.dat", "rb")
             end_of_file = False
            while not end_of_file:
            try:
             print(pickle.load(infile), end = " ")
            except EOFError:
                    end_of_file = True
        infile.close() # Close the input file
         print("\nAll objects are read")
main() # Call the main function
```

```
Enter an integer (the input exits if the input is 0): 4
Enter an integer (the input exits if the input is 0): 5
Enter an integer (the input exits if the input is 0): 7
Enter an integer (the input exits if the input is 0): 9
Enter an integer (the input exits if the input is 0): 0
4 5 7 9
All objects are read
```

The program opens the file for writing binary and repeatedly prompts the user to enter an integer and saves it to the file using the dump function until the integer is 0. The program closes the file and reopens it for reading binary. It repeatedly reads an object using the load function in a while loop until an EOFError exception occurs. When an EOFError exception occurs, end_of_file is to set to True, which terminates the while loop. As shown in the sample output, the user entered four integers and they are saved and then read back and displayed on the console.

8. Client Server Architecture

What is client/server architecture?

It means different things to different people, depending on whom you ask as well as whether you are describing a software or a hardware system. In either case, the it is simple: the server—a piece of hardware or software—provides a "service" that is needed by one or more clients (users of the service). The purpose of existence is to wait for (client) requests, respond to those clients (provide the service), and then wait for more requests. Clients, on the other hand, contact a server for a particular request, send over any necessary data, and then wait for the server to reply, either completing the request or indicating the cause of failure. The server runs indefinitely, continually processing requests; clients make a one-time request for service, receive that service, and thus conclude their transaction. A client might make additional requests at some later time, but these are considered separate transactions. The most common notion of the client/server architecture today is illustrated in Figure 8.1, which depicts a user or client computer retrieving information n from a server across the Internet. Although such a system is indeed an example of a client/server architecture, it isn't the only one. Furthermore, client/server architecture can be applied to computer hardware as well as software.

8.1 Hardware Client/Server Architecture

Print(er) servers is the examples of hardware servers. They process incoming print jobs and send them to a printer (or some other printing device) attached to such a system. Such computer is generally network-accessible and client computers would send it print requests Another example of a hardware server is a file server. These are typically computers with large, generalized storage capacity, which is remotely accessible to clients. Client computers mount the disks from the server computer as if the disk itself were on the local computer. One of the most popular network operating systems that support file servers is Network File System (NFS). If you are accessing a networked disk drive and cannot tell whether it is local or on the network, then the client/ server system has done its job. The main goal is for the user

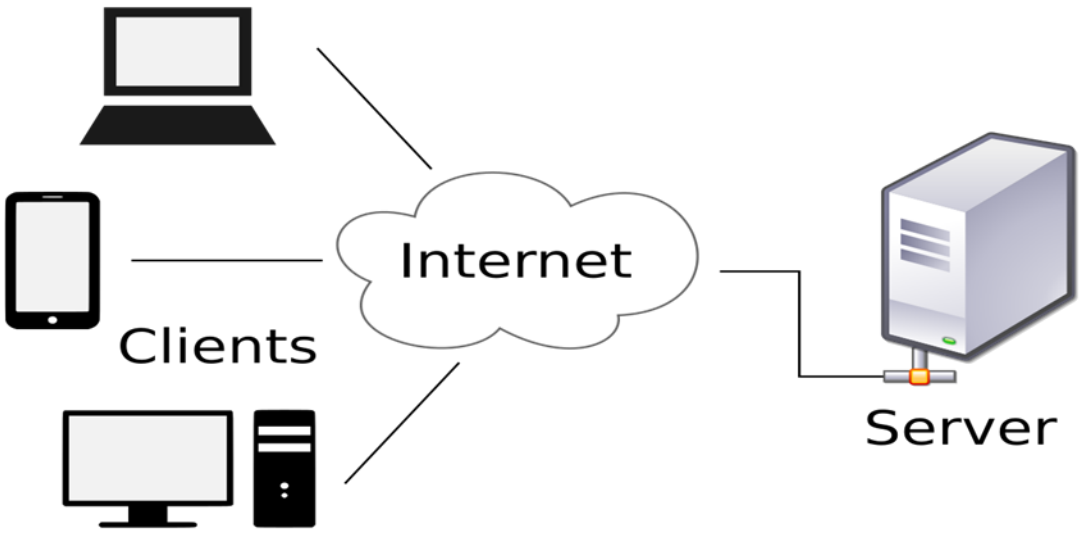


Fig 8.1 Clients/Server Architecture

experience to be exactly the same as that of a local disk—the abstraction is normal disk access.

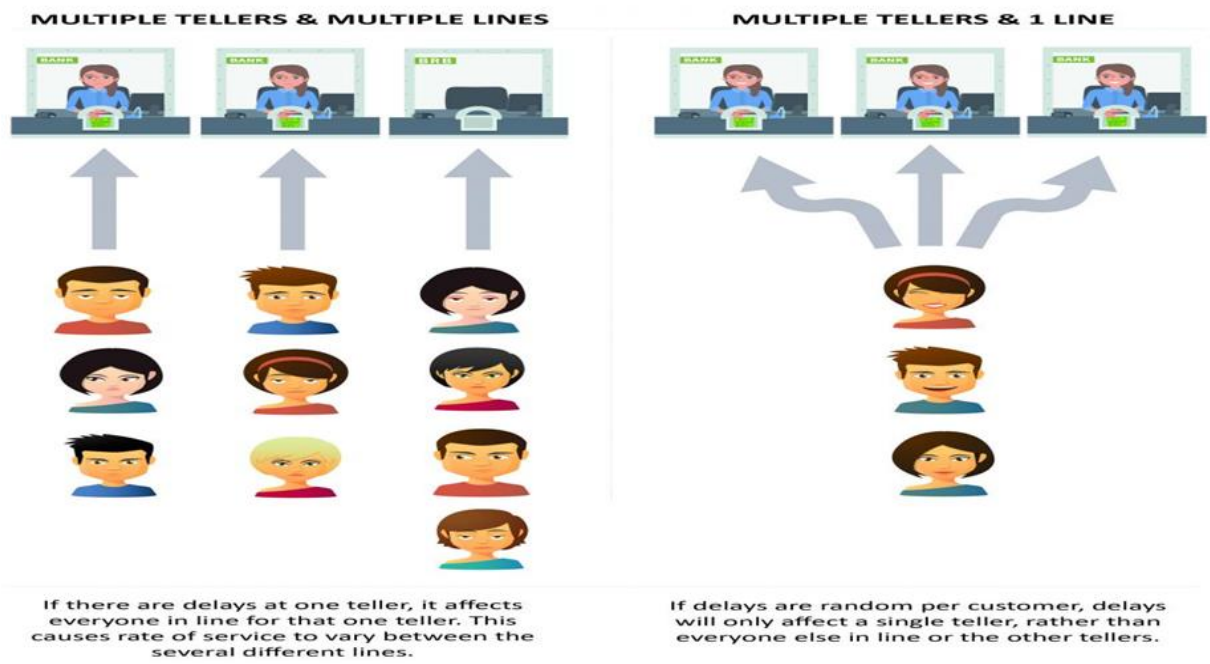
8.2 Software Client/Server Architecture

Software servers also run on a piece of hardware but do not have dedicated peripheral devices as hardware servers do (i.e., printers, disk drives, etc.). The primary services provided by software servers include program execution, data transfer retrieval, aggregation, update, or other types of programmed or data manipulation. One of the more common software servers today is the Web server. Individuals or companies desiring to run their own Web server will get one or more computers, install the Web pages and or Web applications they wish to provide to users, and then start the Web server. The job of such a server is to accept client requests, send back Web pages to (Web) clients, that is, browsers on users' computers, and then wait for the next client request. These servers are started with the expectation of running forever. Although they do not achieve that goal, they go for as long as possible unless stopped by some external force such as being shut down, either explicitly or catastrophically (due to hardware failure). Database servers are another kind of software server. They take client requests for either storage or retrieval, act upon that request, and then wait for more business. They are also designed to run forever. The last type

of software server we will discuss are windows servers. These servers can almost be considered hardware servers. They run on a computer with an attached display, such as a monitor of some sort. Windows clients are actually programs that require a windowing environment in which to execute. These are generally considered graphical user interface (GUI) applications. If they are executed without a window server, meaning, in a text-based environment such as a DOS window or a Unix shell, they are unable to start. Once a windows server is accessible, then things are fine.

8.3 Client/Server Network Programming

Before a server can respond to client requests, some preliminary setup procedures must be performed to prepare it for the work that lies ahead. A communication endpoint is created which allows a server to listen for requests. One can liken our server to a company receptionist or switchboard operator who answers calls on the main corporate line. Once the phone number and equipment are installed and the operator arrives, the service can begin.



This process is the same in the networked world—once a communication endpoint has been established, our listening server can now enter its infinite loop, waiting for clients to connect, and responding to requests. Of course, to

keep our corporate phone receptionist busy, we must not forget to put that phone number on company letterhead, in advertisements, or some sort of press release. Similarly, potential clients must be made aware that this server exists to handle their needs. Imagine creating a brand new Web site. It might be the most super-duper, awesome, amazing, useful, and coolest Web site of all, but if the Web address or URL is never broadcast or advertised in any way, no one will ever know about it, and it will never see the any visitors. Now you have a good idea as to how the server works. You have made it past the difficult part. The client-side stuff is much more simple than that on the server side. All the client has to do is to create its single communication endpoint, and then establish a connection to the server. The client can now make a request, which includes any necessary exchange of data. Once the request has been processed and the client has received the result or some sort of acknowledgement, communication is terminated.

9. Sockets

What Are Sockets?

Sockets are computer networking data structures that embody the concept of the "communication endpoint. Networked applications must create sockets before any type of communication can commence. They can be likened to telephone jacks, without which, engaging in communication is impossible. Sometimes hear these sockets referred to as Berkeley sockets or BSD sockets. Sockets were originally created for same-host applications where they would enable one running program (a.k.a. a process) to communicate with another running program. This is known as interprocess communication, or IPC. There are two types of sockets: file-based and network-oriented.

Unix sockets are the first family of sockets we are looking at and have a "family name" of AF_UNIX, which stands for address family: UNIX. Most popular platforms, including Python, use the term address families and the abbreviation AF. Similarly, AF_LOCAL (standardized in 2000–2001) is supposed to replace AF_UNIX; however, for backward-compatibility, many systems use both and just make them aliases to the same constant. Python

itself still uses AF_UNIX. Because both processes run on the same computer, these sockets are file-based, meaning that their underlying infrastructure is supported by the file system. This makes sense, because the file system is a shared constant between processes running on the same host. The second type of socket is networked-based and has its own family name, AF_INET, or address family: Internet. Another address family, AF_INET6, is used for Internet Protocol version 6 (IPv6) addressing. There are other address families, all of which are either specialized, antiquated, seldom used, or remain unimplemented. Of all address families, AF_INET is now the most widely used. Overall, Python supports only the AF_UNIX, AF_NETLINK, AF_TIPC,and AF _INET families. Because of our focus on network programming, we will be using AF_INET.

9.1 Socket Addresses

A socket is like a telephone jack—a piece of infrastructure that enables communication—then a hostname and port number are like an area code and telephone number combination. Having the hardware and ability to communicate doesn't do any good unless you know to whom and how to "dial." An Internet address is comprised of a hostname and port number pair, which is required for networked communication. Valid port numbers range from 0–65535, although those less than 1024 are reserved for the system. If you are using a POSIX-compliant system (e.g., Linux, Mac OS X, etc.), the list of reserved port numbers (along with servers/protocols and socket types) is found in the /etc/services file. A list of well-known port numbers is accessible at this Web site:

http://www.iana.org/assignments/port-numbers

9.2 Connection-Oriented Sockets vs. Connectionless Sockets

There are two different styles of socket connections. The first type is connection-oriented. This means is that a connection must be established before communication can occur, such as calling a friend using the telephone system. This type of communication is also referred to as a virtual circuit or stream socket. Connection-oriented communication offers sequenced, reliable, and unduplicated delivery of data, without record

boundaries. That basically means that each message may be broken up into multiple pieces, which are all guaranteed to arrive at their destination, put back together and in order, and delivered to the waiting application. The primary protocol that implements such connection types is the Transmission Control Protocol (better known by its acronym, TCP). To create TCP sockets, one must use SOCK_STREAM as the socket type. The SOCK_STREAM name for a TCP socket is based on one of its denotations as stream socket. Because the networked version of these sockets (AF_INET) use the Internet Protocol (IP) to find hosts in the network, the entire system generally goes by the combined names of both protocols (TCP and IP), or TCP/IP.

Next is Connectionless Sockets, it contrast to virtual circuits is the datagram type of socket which is connectionless. This means that no connection is necessary before communication can begin. Here, there are no guarantees of sequencing, reliability, or non duplication in the process of data delivery. Datagrams do preserve record boundaries, however, meaning that entire messages are sent rather than being broken into pieces first, such as with connection-oriented protocols. Message delivery using datagrams can be compared to the postal service. Letters and packages might not arrive in the order they were sent. In fact, they might not arrive at all! To add to the complication, in the land of networking, duplication of messages is even possible. So with all this negativity. Because of the guarantees provided by connection-oriented sockets, a good amount of overhead is required for their setup as well as in maintaining the virtual circuit connection. Datagrams do not have this overhead and thus are "less expensive." They usually provide better performance and might be suitable for some types of applications. The primary protocol that implements such connection types is the User Datagram Protocol (better known by its acronym, UDP). To create UDP sockets, we must use SOCK_DGRAM as the socket type. The SOCK_ DGRAM name for a UDP socket, as you can probably tell, comes from the word "datagram." Because these sockets also use the Internet Protocol to find hosts in the network, this system also has a more general name, going by the combined names of both of these protocols (UDP and IP), or UDP/IP.

10. Networking Programming

Now we know all about client/server architecture, sockets, and networking, let's try to bring these concepts to Python. The primary module we will be using is the socket module. Found within this module is the socket() function, which is used to create socket objects. Sockets also have their own set of methods, which enable socket-based network communication.

10.1 socket() Module Function

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

socket(socket_family, socket_type, protocol=0)

The socket_family is either AF_UNIX or AF_INET, as explained earlier, and the socket_type is either SOCK_STREAM or SOCK_ DGRAM. The protocol is usually left out, defaulting to 0. So to create a TCP/IP socket, you call socket.socket() like this:

tcpSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

Likewise, to create a UDP/IP socket you perform:

udpSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

Because there are numerous socket module attributes, this is one of the exceptions where using from module import * is somewhat acceptable. If we applied from socket import *, we bring the socket attributes into our namespace, but our code is shortened considerably, as demonstrated in the following:

tcpSock = socket(AF_INET, SOCK_STREAM)

Once we have a socket object, all further interaction will occur using that socket object's methods.

10.2 Socket Object (Built-In) Methods

Socket Object (Built-In) Methods

In Table 1, we present a list of the most common socket methods. In the next subsections, we will create both TCP and UDP clients and servers, using some of these methods.

Table 10.1 Common Socket Object Methods and Attributes

Name	Description
Server Socket Methods	
s.bind()	Bind address (hostname, port number pair) to socket
s.listen()	Set up and start TCP listener
s.accept()	Passively accept TCP client connection, waiting until connection arrives (blocking)
Client Socket Methods	
s.connect()	Actively initiate TCP server connection
s.connect_ex()	Extended version of connect(), where problems returned as error codes rather than an exception being thrown
General Socket Methods	
s.recv()	Receive TCP message
s.send()	Transmit TCP message
s.sendall()	Transmit TCP message completely
s.recvfrom()	Receive UDP message
s.sendto()	Transmit UDP message
s.getpeername()	Remote address connected to socket (TCP)
s.getsockname()	Address of current socket
s.getsockopt()	Return value of given socket option
s.setsockopt()	Set value for given socket option

s.close()	Close socket

Creating a TCP Server

We will first present some general pseudocode needed to create a generic TCP server. Keep in mind that this is only one way of designing your server. Once you become comfortable with server design, you will be able to modify the following pseudocode to operate the however want it to:

```
ss = socket()          # create server socket
ss.bind()              # bind socket to address
ss.listen()            # listen for connections
inf_loop:              # server infinite loop
cs = ss.accept()       # accept client connection
comm_loop:             # communication loop
cs.recv()/cs.send()    # dialog (receive/send)
cs.close()             # close client socket
ss.close()             # close server socket # (opt)
```

All sockets are created by using the socket.socket() function. Servers need to "sit on a port" and wait for requests, so they all must bind to a local address. Because TCP is a connection-oriented communication system, some infrastructure must be set up before a TCP server can begin operation. In particular, TCP servers must "listen" for (incoming) connections. Once this setup process is complete, a server can start its infinite loop. A simple (single-threaded) server will then sit on an accept() call, waiting for a connection. By default, accept() is blocking, meaning that execution is suspended until a connection arrives. Sockets do support a nonblocking mode. Once a connection is accepted, a separate client socket is returned (by accept()) for the upcoming message interchange. Using the new client socket is similar to handing off a customer call to a service representative.

When a client eventually does come in, the main switchboard operator takes the incoming call and patches it through, using another line to connect to the appropriate person to handle the client's needs. This frees up the main line

(the original server socket) so that the operator can resume waiting for new calls (client requests) while the customer and the service representative he is connected to carry on their own conversation. Likewise, when an incoming request arrives, a new communication port is created to converse directly with that client, again, leaving the main port free to accept new client connections.

Once the temporary socket is created, communication can commence, and both client and server proceed to engage in a dialog of sending and receiving, using this new socket until the connection is terminated. This usually happens when one of the parties either closes its connection or sends an empty string to its counterpart. In our code, after a client connection is closed, the server goes back to wait for another client connection.

```
#!/usr/bin/env python
from socket import *
from time import ctime
HOST = ''
PORT = 21567
BUFSIZ = 1024
ADDR = (HOST, PORT)
tcpSerSock = socket(AF_INET, SOCK_STREAM)
tcpSerSock.bind(ADDR)
tcpSerSock.listen(5)
while True:
    print 'waiting for connection...'
    tcpCliSock, addr = tcpSerSock.accept()
    print '...connected from:', addr

while True:
    data = tcpCliSock.recv(BUFSIZ)
```

```
        if not data:
        break
      tcpCliSock.send('[%s] %s' % (ctime(), data))
tcpCliSock.close()
tcpSerSock.close()
```

Creating a TCP Client

Creating a client is much simpler than a server. Similar to our description of the TCP server, we will present the pseudocode with explanations first, then show you the real thing.

```
cs = socket()          # create client socket
cs.connect()           # attempt server connection
comm_loop:             # communication loop
cs.send()/cs.recv()    # dialog (send/receive)
cs.close()             # close client socket
```

As we noted earlier, all sockets are created by using socket.socket(). Once a client has a socket, however, it can immediately make a connection to a server by using the socket's connect() method. When the connection has been established, it can participate in a dialog with the server. Once the client has completed its transaction, it can close its socket, terminating the connection.

```
#!/usr/bin/env python
from socket import *
HOST = 'localhost'
 PORT = 21567
 BUFSIZ = 1024
 ADDR = (HOST, PORT)
tcpCliSock = socket(AF_INET, SOCK_STREAM)
tcpCliSock.connect(ADDR)
```

```
while True:
        data = raw_input('> ')
        if not data:
                break
 tcpCliSock.send(data)
data = tcpCliSock.recv(BUFSIZ)
 if not data:
        break
print data
 tcpCliSock.close()
```

Creating a UDP Server

UDP servers do not require as much setup as TCP servers because they are not connection-oriented. There is virtually no work that needs to be done other than just waiting for incoming connections.

```
ss = socket()                   # create server socket
ss.bind()                       # bind server socket
inf_loop:                       # server infinite loop
cs = ss.recvfrom()/ss.sendto()  # dialog (receive/send)
ss.close()                       # close server socket
```

As you can see from the pseudocode, there is nothing extra other than the usual create-the-socket and bind it to the local address (host/port pair). The infinite loop consists of receiving a message from a client, timestamping and returning the message, and then going back to wait for another message. Again, the close() call is optional and will not be reached due to the infinite loop, but it serves as a reminder that it should be part of the graceful or intelligent exit scheme we've been mentioning. One other significant difference between UDP and TCP servers is that because datagram sockets are connectionless, there is no "handing off" of a client connection to a

separate socket for succeeding communication. These servers just accept messages and perhaps reply.

```
#!/usr/bin/env python
from socket import *
from time import ctime
HOST = ''
PORT = 21567
BUFSIZ = 1024
ADDR = (HOST, PORT)
udpSerSock = socket(AF_INET, SOCK_DGRAM)
udpSerSock.bind(ADDR)
while True:
        print 'waiting for message...'
        data, addr = udpSerSock.recvfrom(BUFSIZ)
        udpSerSock.sendto('[%s] %s' % (
        ctime(), data), addr)
        print '...received from and returned to:', addr
       udpSerSock.close()
```

Creating a UDP Client

Of the four clients highlighted here, the UDP client is the shortest bit of code that we will look at. The pseudocode looks like this.

```
cs = socket()  # create client socket comm_loop: # communication loop
cs.sendto()/cs.recvfrom()       # dialog (send/receive)
cs.close()                      # close client socket
```

Once a socket object is created, we enter the dialog loop, wherein we exchange messages with the server. When communication is complete, the socket is closed.

```
#!/usr/bin/env python
from socket import *
HOST = 'localhost'
 PORT = 21567
 BUFSIZ = 1024
 ADDR = (HOST, PORT)
 udpCliSock = socket(AF_INET, SOCK_DGRAM)
 while True:
        data = raw_input('> ')
        if not data:
            break
        udpCliSock.sendto(data, ADDR)
        data, ADDR = udpCliSock.recvfrom(BUFSIZ)
        if not data:
            break
        print data
udpCliSock.close()
```

11. Twisted Framework

Twisted is a complete event-driven networking framework with which you can both use and develop complete asynchronous networked applications and protocols. It is not part of the Python Standard Library as of this writing and must be downloaded and installed separately (you can use the link at the end of the chapter). It provides a significant amount of support for you to build complete systems, including network protocols, threading, security and authentication, chat/IM, DBM and RDBMS database integration, Web/Internet, e-mail, command-line arguments, GUI toolkit integration, etc. Using Twisted to implement our tiny simplistic example is like using a

sledgehammer to pound a thumbtack, but you have to get started somehow, and our application is the equivalent to the "hello world" of networked applications. Like SocketServer, most of the functionality of Twisted lies in its classes. In particular for our examples, we will be using the classes found in the reactor and protocol subpackages of Twisted's Internet component.

Creating a Twisted Reactor TCP Server

You will find the code similar to that of the SocketServer example. Instead of a handler class, however, we create a protocol class and override several methods in the same manner as installing callbacks. Also, this example is asynchronous :

```
#!/usr/bin/env python
from twisted.internet import protocol, reactor
 from time import ctime
PORT = 21567
class TSServProtocol(protocol.Protocol):
def connectionMade(self):
        clnt = self.clnt = self.transport.getPeer().host
        print '...connected from:', clnt
 def dataReceived(self, data):
        self.transport.write('[%s] %s' % ( ctime(), data))
       factory = protocol.Factory()
       factory.protocol = TSServProtocol
       print 'waiting for connection...'
       reactor.listenTCP(PORT, factory)
       reactor.run()
```

Creating a Twisted Reactor TCP Client

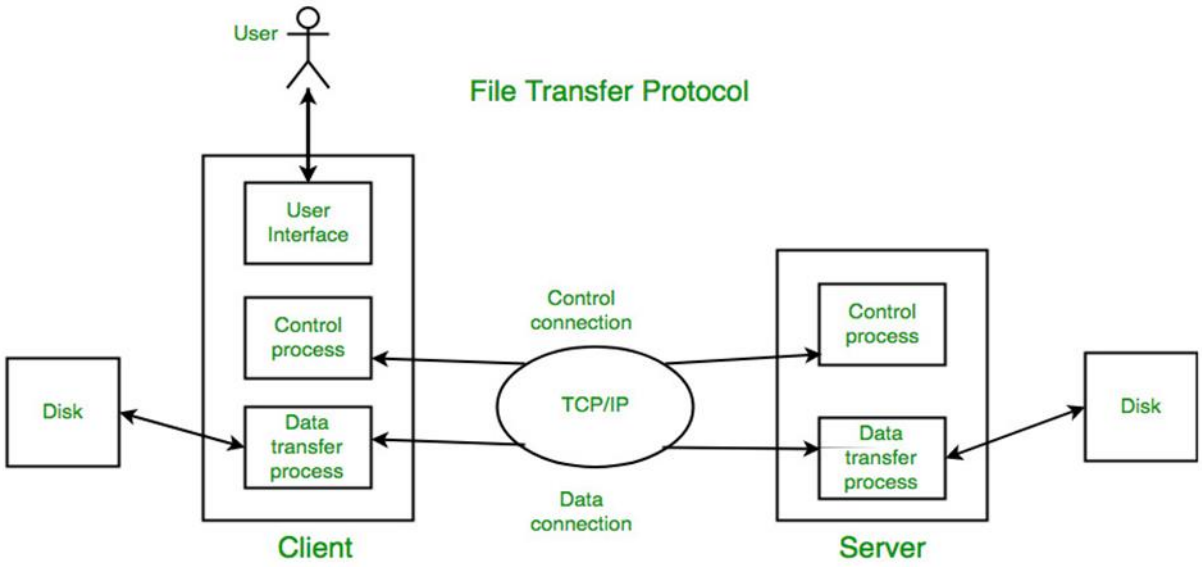
Unlike the SocketServer TCP client, it will not look like all the other clients—this one is distinctly Twisted.

```
#!/usr/bin/env python
from twisted.internet import protocol, reactor
HOST = 'localhost'
PORT = 21567
class TSClntProtocol(protocol.Protocol):
 def sendData(self):
data = raw_input('> ')
if data:
        print '...sending %s...' % data
        self.transport.write(data)
else:
        self.transport.loseConnection()
 def connectionMade(self):
        self.sendData()
def dataReceived(self, data):
        print data
        self.sendData()
class TSClntFactory(protocol.ClientFactory):
        protocol = TSClntProtocol
        clientConnectionLost = clientConnectionFailed = \
                lambda self, connector, reason: reactor.stop()

 reactor.connectTCP(HOST, PORT, TSClntFactory())
reactor.run()
```

12. File Transfer Protocol

The File Transfer Protocol (FTP) was developed by the late Jon Postel and Joyce Reynolds in the Internet Request for Comment (RFC) 959 document and published in October 1985. It is primarily used to download publicly accessible files in an anonymous fashion. It can also be used to transfer files between two computers, especially when you're using a Unix-based system for file storage or archiving and a desktop or laptop PC for work. Before the Web became popular, FTP was one of the primary methods of transferring files on the Internet, and one of the only ways to download software and/or source code.

As mentioned previously, you must have a login/password to access the remote host running the FTP server. The exception is anonymous logins, which are designed for guest downloads. These permit clients who do not have accounts to download files. The server's administrator must set up an FTP server with anonymous logins to enable this. In these cases, the login of an unregistered user is called anonymous, and the password is generally the e-mail address of the client. This is akin to a public login and access to directories that were designed for general consumption as opposed to logging in and transferring files as a particular user. The list of available commands via the FTP protocol is also generally more restrictive than that for real users.



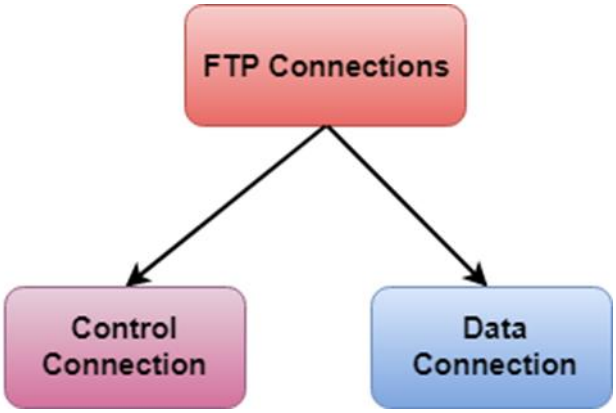
The protocol is diagrammed in Figure shown above and works as follows:

1. Client contacts the FTP server on the remote host
2. Client logs in with username and password (or anonymous and e-mail address)
3. Client performs various file transfers or information requests
4. Client completes the transaction by logging out of the remote host and FTP server

Of course, this is generally how it works. Sometimes there are circumstances whereby the entire transaction is terminated before it's completed. These include being disconnected from the network if one of the two hosts crash or because of some other network connectivity issue. For inactive clients, FTP connections will generally time out after 15 minutes (900 seconds) of inactivity. Under the hood, it is good to know that FTP uses only TCP it does not use UDP in any way. Also, FTP can be seen as a more unusual example of client/server programming because both the clients and the servers use a pair of sockets for communication: one is the control or command port, and the other is the data port.

There are two FTP modes: Active and Passive, and the server's data port is only 20 for Active mode. After the server sets up 20 as its data port, it "actively" initiates the connection to the client's data port. For Passive mode, the server is only responsible for letting the client know where its random data port is; the client must initiate the data connection. As you can see in this mode, the FTP server is taking a more passive role in setting up the data connection. Finally, there is now support for a new Extended Passive Mode to support version 6 Internet Protocol (IPv6) addresses—see RFC 2428. Python supports most Internet protocols, including FTP. Now let's take a look at just how easy it is to create an Internet client with Python.

There are two types of connections in FTP:



Control Connection: The control connection uses very simple rules for communication. Through control connection, we can transfer a line of command or line of response at a time. The control connection is made between the control processes. The control connection remains connected during the entire interactive FTP session.

Data Connection: The Data Connection uses very complex rules as data types may vary. The data connection is made between data transfer processes. The data connection opens when a command comes for transferring the files and closes when the file is transferred.

FTP Clients

- FTP client is a program that implements a file transfer protocol which allows you to transfer files between two hosts on the internet.
- It allows a user to connect to a remote host and upload or download the files.
- It has a set of commands that we can use to connect to a host, transfer the files between you and your host and close the connection.
- The FTP program is also available as a built-in component in a Web browser. This GUI based FTP client makes the file transfer very easy and also does not require to remember the FTP commands.

Advantages of FTP:

- Speed: One of the biggest advantages of FTP is speed. The FTP is one of the fastest way to transfer the files from one computer to another computer.

- Efficient: It is more efficient as we do not need to complete all the operations to get the entire file.
- Security: To access the FTP server, we need to login with the username and password. Therefore, we can say that FTP is more secure.
- Back & forth movement: FTP allows us to transfer the files back and forth. Suppose you are a manager of the company, you send some information to all the employees, and they all send information back on the same server.

Disadvantages of FTP:

- The standard requirement of the industry is that all the FTP transmissions should be encrypted. However, not all the FTP providers are equal and not all the providers offer encryption. So, we will have to look out for the FTP providers that provides encryption.
- FTP serves two operations, i.e., to send and receive large files on a network. However, the size limit of the file is 2GB that can be sent. It also doesn't allow you to run simultaneous transfers to multiple receivers.
- Passwords and file contents are sent in clear text that allows unwanted eavesdropping. So, it is quite possible that attackers can carry out the brute force attack by trying to guess the FTP password.
- It is not compatible with every system.

Python and FTP

Now we write an FTP client by using Python. The only additional work required is to import the appropriate Python module and make the appropriate calls in Python. So let’s review the protocol briefly:

1. Connect to server
2. Log in
3. Make service request(s) (and hopefully get response[s])
4. Quit

When using Python's FTP support, all you do is import the ftplib module and instantiate the ftplib.FTP class. All FTP activity—logging in, transferring files, and logging out—will be accomplished using your object.

Here is some Python pseudocode:

```
from ftplib import FTP
f = FTP('some.ftp.server')
f.login('anonymous', 'your@email.address')
:
f.quit()
```

Soon we will look at a real example, but for now, let's familiarize ourselves with methods from the ftplib.FTP class, which you will likely use in your code.

Methods for FTP Objects

Method	Description
login(*user*='anonymous', *passwd*='', acct='')	Log in to FTP server; all arguments are optional
pwd()	Current working directory
cwd(*path*)	Change current working directory to *path*
dir([*path*[,...[,cb]])	Displays directory listing of *path*; optional callback *cb* passed to retrlines()
rename(*old*, *new*)	Rename remote file from *old* to *new*
delete(*path*)	Delete remote file located at *path*
mkd(*directory*)	Create remote *directory*
rmd(*directory*)	Remove remote *directory*
quit()	Close connection and quit

The methods you will most likely use in a normal FTP transaction include login(), cwd(), dir(), pwd(), stor*(), retr*(), and quit().

A Client Program FTP Example

As mentioned previously that an example script is not even necessary because you can run one interactively and not get lost in any code.

```
#!/usr/bin/env python
import ftplib
import os
import socket
HOST = 'ftp.mozilla.org'
DIRN = 'pub/mozilla.org/webtools'
FILE = 'bugzilla-LATEST.tar.gz'
def main():
 try:
        f = ftplib.FTP(HOST)
 except (socket.error, socket.gaierror) as e:
         print 'ERROR: cannot reach "%s"' % HOST
        return
 print '*** Connected to host "%s"' % HOST
try:
         f.login()
except ftplib.error_perm:
        print 'ERROR: cannot login anonymously'
 f.quit()
return
print '*** Logged in as "anonymous"'

try:
 f.cwd(DIRN)
```

```
        except ftplib.error_perm:
                print 'ERROR: cannot CD to "%s"' % DIRN
                f.quit()
                return
         print '*** Changed to "%s" folder' % DIRN
         try:
                 f.retrbinary('RETR %s' % FILE,
                 open(FILE, 'wb').write)
        except ftplib.error_perm:
                print 'ERROR: cannot read file "%s"' % FILE
                os.unlink(FILE)
         else:
                print '*** Downloaded "%s" to CWD' % FILE
                  f.quit()
                  return
        if __name__ == '__main__':
        main()
```

If no errors occur when we run our script, we get the following output:

```
$ getLatestFTP.py
*** Connected to host "ftp.mozilla.org"
*** Logged in as "anonymous"
*** Changed to "pub/mozilla.org/webtools" folder
*** Downloaded "bugzilla-LATEST.tar.gz" to CWD
$
```

13. Usenets and Newsgroup

Like mailing lists Usenet is also a way of sharing information. It was started by Tom Truscott and Jim Ellis in 1979. Initially it was limited to two sites but today there are thousands of Usenet sites involving millions of people.

Usenet is a kind of discussion group where people can share views on topic of their interest. The article posted to a newsgroup becomes available to all readers of the newsgroup.

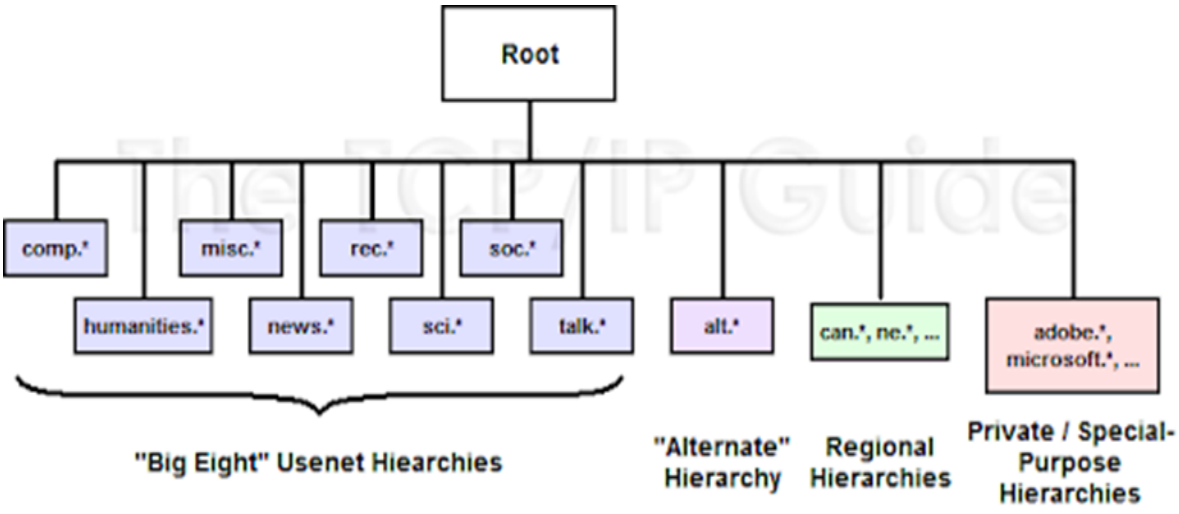
The Usenet News System is a global archival bulletin board. There are newsgroups for just about any topic, from poems to politics, linguistics to computer languages, software to hardware, planting to cooking, finding or announcing employment opportunities, music and magic, breaking up or finding love. Newsgroups can be general and worldwide or targeted toward a specific geographic region.

The entire system is a large global network of computers that participate in sharing Usenet postings. Once a user uploads a message to his local Usenet computer, it will then be propagated to other adjoining Usenet computers, and then to the neighbors of those systems, until it's gone around the world and everyone has received the posting. Postings will live on Usenet for a finite period of time, either dictated by a Usenet system administrator or the posting itself via an expiration date/time. Each system has a list of newsgroups that it subscribes to and only accepts postings of interest—not all newsgroups may be archived on a server. Usenet news service is dependent on which provider you use. Many are open to the public; others only allow access to specific users,

such as paying subscribers, or students of a particular university.

A login and password are optional, configurable by the Usenet system administrator. The ability to post only download is another parameter configurable by the administrator. Usenet has lost its place as the global bulletin board, superseded in large part by online forums. Still it's worthwhile looking at Usenet here specifically for its network protocol.

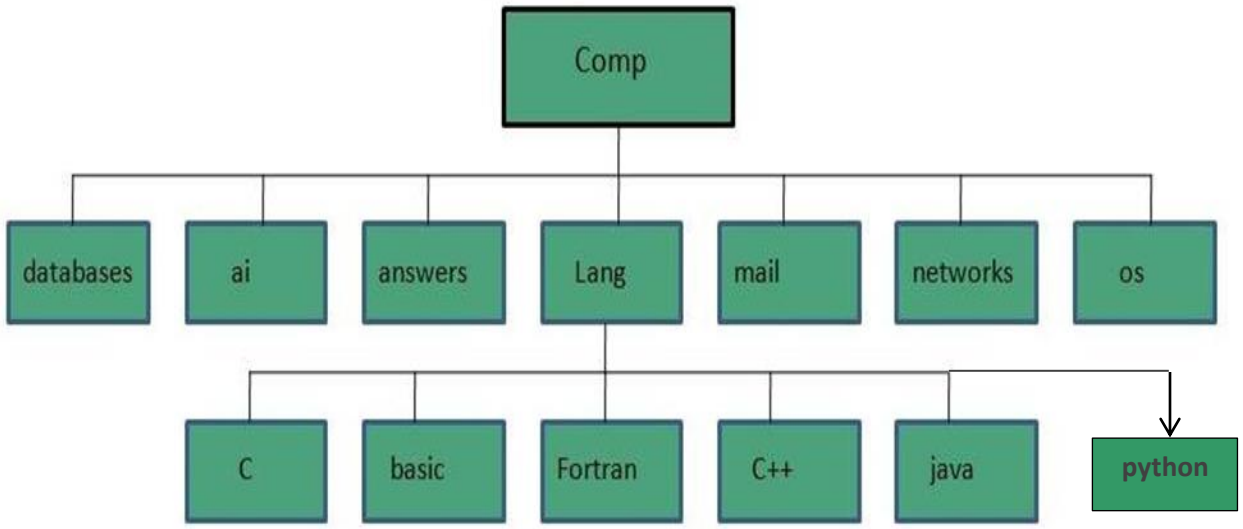
While older incarnations of the Usenet used UUCP as its network transport mechanism, another protocol arose in the mid-1980s when most network traffic began to migrate to TCP/IP.



Usenet messages are not addressed to individual users; rather, they are posted to newsgroups. Each newsgroup represents a topic; those with an interest in the subject of a group can read messages in it, and reply to them as well. Usenet newsgroups are arranged into tree-like hierarchies that are similar in structure to DNS domains. Many of the most widely-used newsgroups are found in a collection of general-interest hierarchies called the Big Eight. There are also many regional and special-purpose hierarchies.

Newsgroup Classification

There exist a number of newsgroups distributed all around the world. These are identified using a hierarchical naming system in which each newsgroup is assigned a unique name that consists of alphabetic strings separated by periods.



Newsgroup hierarchy of 'Comp'

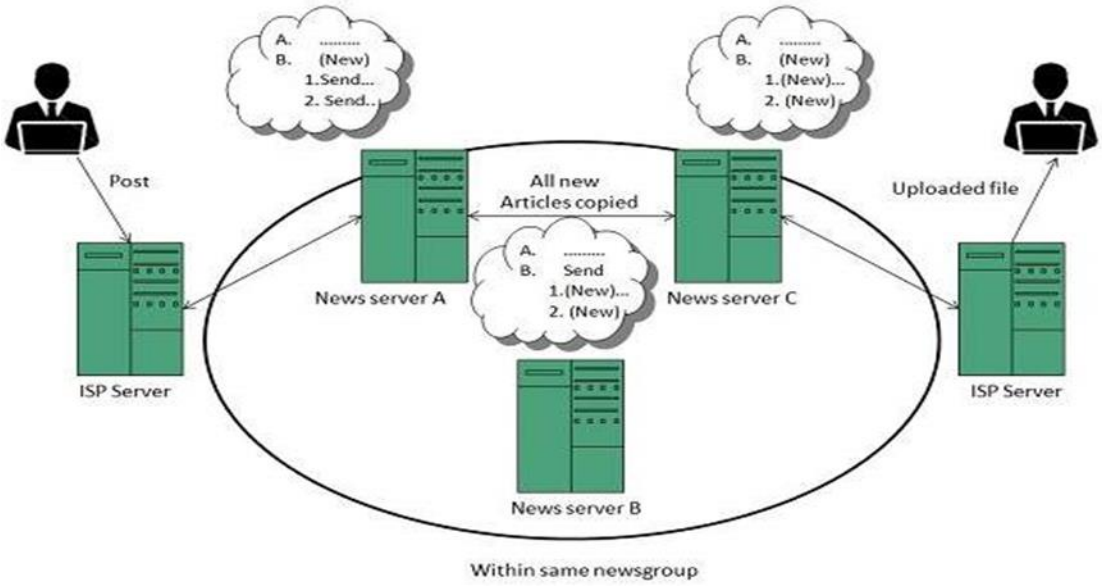
The leftmost portion of the name represents the top-level category of the newsgroup followed by subtopic. The subtopic can further be subdivided and subdivided even further (if needed).For example, the newsgroup comp.lang.python contains discussion on python language. The leftmost part comp classifies the newsgroup as one that contains discussion of computer related topics. The second part identifies one of the subtopic lang that related to computer languages. The third part identifirs one of the computer languages, in this case python.

The following table shows the top-level hierarchies of Usenet Newsgroup:

Comp.*	Computer related topics including computer hardware, software, languages etc.	Comp.lang.java.beans Comp.database.oracle
News.*	Newsgroup and Usenet topics	News.software.nntp
Rec.*	Artistic activities, hobbies, or recreational activities	Rec.arts.animation

	such as books, movies etc.	
Sci.*	Scientific topics	Sci.bio.botany
Soc.*	Social issues and various culture	Soc.culture.india
Talk.*	Conventional subjects such as religion, politics etc.	Soc.politics.india
Humanities.*	Art, literature, philosophy and culture	Humanities.classics
Misc.*	Miscellaneous topics i.e. issues tat may not fit into other categories	Misc.answers Misc.books.technical

When a newsreader such as outlook express connects to a news server, it downloads all the new messages posted in the subscribed newsgroup. We can either reply a message after reading or post a news article to the news server. The article posted to a news server is appended to the file maintained for that newsgroup Then the news server shares article with other news servers that are connected to it. Then each news server compares if both carry the same newsgroup. If yes, then by comparing the files it checks that if there are any new articles in the file, if so they are appended to the file. The updated file of the news servers is then sent to other news servers connected to it. This process is continues until all of the news servers have updated information.



Reading Articles

If user wants to read article, user has to connect to the news server using the newsreader. The newsreader will then display a list of newsgroups available on the news server where user can subscribe to any of the news group. After subscription the newsreader will automatically download articles from the newsgroup. After reading the article user can either post a reply to newsgroup or reply to sender by email. The newwsreader saves information about the subscribed newsgroups and articles read by the user in each group.

Posting an Article

In order to send new article to a newsgroup, user first need to compose an article and specify the names of the newsgroup to whom he/she wants to send. An article can be sent to one or more newsgroup at a time provided all the newsgroups are on same news server. It is also possible to cancel the article that you have posted but if someone has downloaded an article before cancellation then that person will be able to read the article.

Replying an Article

After reading the article user can either post a reply to newsgroup or reply to sender by email. There are two options available Reply and Reply group. Using Reply, the reply mail will be sent to the autor of the article while Reply group will send a reply to whole of the newsgroup.

Cancelling an Article

To cancle an article after it is sent, select the message and click Message > Cancel message. It will cancle the message from the news server. But if someone has downloaded an article before cancellation then that person will be able to read the article.

Usenet netiquette

While posting an article on a newsgroup, one should follow some rules of netiquette as listed below:

- Spend some time in understanding a newsgroup when you join it for first time.
- Article posted by you should be easy to read, concise and grammatically correct.
- Information should be relevant to the article title.
- Don't post same article to multiple newsgroups.
- Avoid providing your business email address while subscribing to a newsgroup as may be used by spammers.
- Avoid using capital letters as someone may interpret as shouting.
- Prefer to use plain text wherever possible in your article

Mailing list vs. Newsgroup

S.N.	Mailing List	Newsgroup
1.	Messages are delivered to individual mailboxes of subscribed member of group.	Messages are not posted to individual mailboxes but can be viewed by anyone who has subscribed to that newsgroup.
2.	Working with mailing list is easier than newsgroup. It is easy to compose and receive emails.	Working with a particular newsgroup requires proper knowledge of that newsgroup.
3.	In order to send or receive	It requires a newsgroup reader.

		mails, you required an email program.	
4.		Messages are delivered to certain group of people.	Messages are available to public.
5.		Mailing list does not support threaded discussion.	Newsgroup supports threaded discussion.
6.		Messages delivered to listed subscribers can not be cancelled.	Article posted on a newsgroup can be cancelled.

14. Email

E-mail, is both archaic and modern at the same time. For those of us who have been using the Internet since the early days, e-mail seems so "old," especially compared to newer and more immediate communication mechanisms, such as Web-based online chat, instant messaging (IM), and digital telephony such as Voice over Internet Protocol (VoIP) applications. If you are already familiar with this and just want to move on to developing e-mail related clients in Python. Before we take a look at the e-mail infrastructure, have you know what is the exact definition of an e-mail message? It is message consists of header fields (collectively called 'the header of the message') followed, optionally, by a body." When we think of e-mail as users, we immediately think of its contents, whether it be a real message or an unsolicited commercial advertisement.

E-Mail System Components and Protocols

E-mail actually existed before the modern Internet came around. It actually started as a simple message exchange between mainframe users; there wasn't even any networking involved as senders and receivers all used the same computer. Then when networking became a reality, it was possible for users on different hosts to exchange messages. This, of course, was a complicated concept because people used different computers, which more than likely also used different networking protocols. It was not until the

early 1980s that message exchange settled on a single standard for moving e-mail around the Internet. Before we get into the details, let's first know how does e-mail work. How does a message get from sender to recipient across the vastness of all the computers accessible on the Internet. To put it simply, there is the originating computer (the sender's message departs from here) and the destination computer (recipient's mail server). The optimal solution is if the sending computer knows exactly how to reach the receiving host, because then it can make a direct connection to deliver the message.

The sending computer queries to find another intermediate host that can pass the message along its way to the final recipient host. Then that host searches for the next host who is another step closer to the destination. So in between the originating and final destination hosts are any number of computers. These are called hops. If you look carefully at the

full e-mail headers of any message you receive, you will see a "passport" stamped with all the places your message bounced to before it finally reached you. To get a clearer picture, let's take a look at the components of the e-mail system. The foremost component is the message transport agent (MTA). This is a server process running on a mail exchange host that is responsible for the routing, queuing, and sending of e-mail. These represent all the hosts that an e-mail message bounces from, beginning at the source host all the way to the final destination host and all hops in between. It is agents of message transport. For all this to work, MTAs need to know two things:

1) How determine the next MTA to forward a message to,

2) how to talk to another MTA.

The first it is solved by using a domain name service (DNS) lookup to find the MX (Mail eXchange) of the destination domain. This is not necessarily the final recipient; it might simply be the next recipient who can eventually get the message to its final destination.

Sending E-Mail

To send e-mail, our mail client must connect to an MTA, and the only language they understand is a communication protocol. The way MTAs communicate with one another is by using a message transport system (MTS). This protocol must be recognized by a pair of MTAs before they can communicate with one another. As we described earlier, such communication was dicey and unpredictable in the early days because there were so many different types of computer systems, each running different networking software. In addition, computers were using both networked transmission as well as dial-up modem, so delivery times were unpredictable. The Simple Mail Transfer Protocol (SMTP), one of the foundations of modern e-mail.

Receiving E-Mail

Communicating by e-mail on the Internet was relegated to university Students, Researchers, and employees of private industry and commercial Corporations. Desktop Computers were predominantly still Unix-based workstations. Home users focused mainly on dial-up Web access on PCs and really didn't use e-mail. When the Internet began to explode in the mid-1990s, e-mail came home to everyone. Because it was not feasible for home users to have workstations in their dens running SMTP, a new type of system had to be devised to leave e-mail on an incoming mail host while periodically downloading mail for offline reading. Such a system had to consist of both a new application and a new protocol to communicate with the mail server. The application, which runs on a home computer, is called a mail user agent (MUA). An MUA will download mail from a server, perhaps automatically deleting it from the server in the process (or leaving the mail on the server to be deleted manually by the user). However, an MUA must also be able to send mail; in other words, it should also be able to speak SMTP to communicate directly to an MTA when sending mail

15. Simple Mail Transfer Protocol (SMTP)

Email is emerging as one of the most valuable services on the internet today. Most of the internet systems use SMTP as a method to transfer mail from one user to another. SMTP is a push protocol and is used to send the mail whereas POP (post office protocol) or IMAP (internet message access protocol) are used to retrieve those mails at the receiver's side.

SMTP Fundamentals

SMTP is an application layer protocol. The client who wants to send the mail opens a TCP connection to the SMTP server and then sends the mail across the connection. The SMTP server is always on listening mode. As soon as it listens for a TCP connection from any client, the SMTP process initiates a connection on that port (25). After successfully establishing the TCP connection the client process sends the mail instantly.

SMTP Protocol

The SMTP model is of two type :

- End-to- end method
- Store-and- forward method

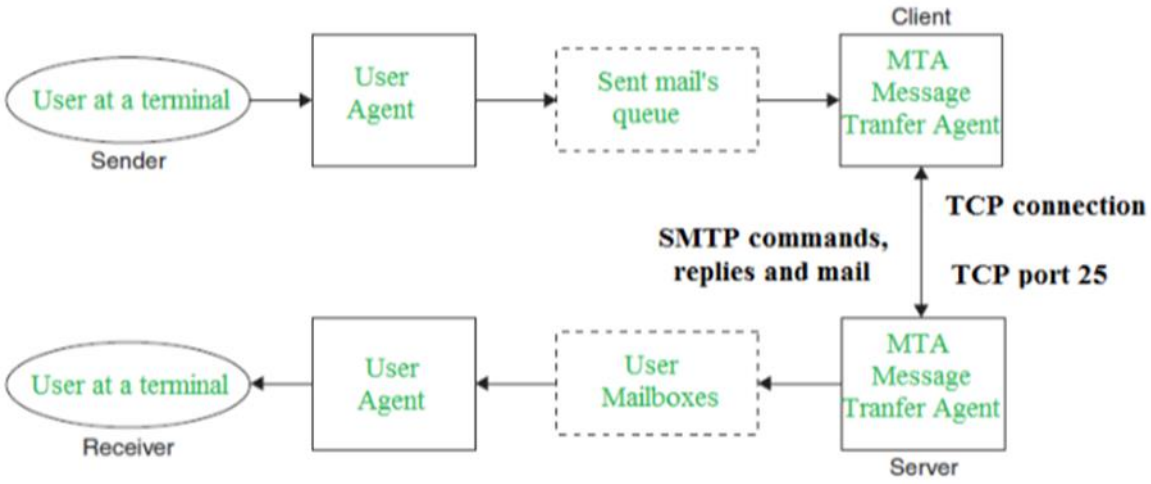
The end to end model is used to communicate between different organizations whereas the store and forward method are used within an organization. A SMTP client who wants to send the mail will contact the destination's host SMTP directly in order to send the mail to the destination. The SMTP server will keep the mail to itself until it is successfully copied to the receiver's SMTP.

The client SMTP is the one which initiates the session let us call it as the client- SMTP and the server SMTP is the one which responds to the session request and let us call it as receiver-SMTP. The client- SMTP will start the session and the receiver-SMTP will respond to the request.

Model of SMTP system

In the SMTP model user deals with the user agent (UA) for example Microsoft Outlook, Netscape, Mozilla, etc. In order to exchange the mail using TCP, MTA is used. The users sending the mail do not have to deal

with the MTA it is the responsibility of the system admin to set up the local MTA. The MTA maintains a small queue of mails so that it can schedule repeat delivery of mail in case the receiver is not available. The MTA delivers the mail to the mailboxes and the information can later be downloaded by the user agents.



Both the SMTP-client and MSTP-server should have 2 components:

- User agent (UA)
- Local MTA

Communication between sender and the receiver :

The senders, user agent prepare the message and send it to the MTA. The MTA functioning is to transfer the mail across the network to the receivers MTA. To send mail, a system must have the client MTA, and to receive mail, a system must have a server MTA.

SENDING EMAIL:

Mail is sent by a series of request and response messages between the client and a server. The message which is sent across consists of a header and the body. A null line is used to terminate the mail header. Everything which is after the null line is considered as the body of the message which is a sequence of ASCII characters. The message body contains the actual information read by the receipt.

RECEIVING EMAIL:

The user agent at the server side checks the mailboxes at a particular time of intervals. If any information is received it informs the user about the mail. When the user tries to read the mail it displays a list of mails with a short description of each mail in the mailbox. By selecting any of the mail user can view its contents on the terminal.

Some SMTP Commands:

- HELO – Identifies the client to the server, fully qualified domain name, only sent once per session
- MAIL – Initiate a message transfer, fully qualified domain of originator
- RCPT – Follows MAIL, identifies an addressee, typically the fully qualified name of the addressee and for multiple addressees use one RCPT for each addressee
- DATA – send data line by line

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

```
#!/usr/bin/python
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. Unless you are using a webmail service (such as Hotmail or Yahoo! Mail), your e-mail

provider must have provided you with outgoing mail server details that you can supply them, as follows −

```
smtplib.SMTP('mail.your-domain.com', 25)
```

16. Post Office Protocol (POP3)

POP stands for Post Office Protocol. It is generally used to support a single client. There are several versions of POP but the POP 3 is the current standard. POP3 or Post Office Protocol Version 3 is an application layer protocol used by email clients to retrieve email messages from mail servers over TCP/IP network. POP was designed to move the messages from server to local disk but version 3 has the option of leaving a copy on the server.POP3 is a very simple protocol to implement but that limits its usage. For example, POP3 supports only one mail server for each mailbox. It has now has been made obsolete by modern protocols like IMAP.

Key Points

- POP is an application layer internet standard protocol.
- Since POP supports offline access to the messages, thus requires less internet usage time.
- POP does not allow search facility.
- In order to access the messaged, it is necessary to download them.
- It allows only one mailbox to be created on server.
- It is not suitable for accessing non mail data.
- POP3 commands are generally abbreviated into codes of three or four letters. Eg. STAT.

POP3 Commands

The following table describes some of the POP commands:

Sl.No.	Command	Description
1	LOGIN	This command opens the connection.
2	STAT	It is used to display number of messages currently in the mailbox.
3	LIST	It is used to get the summary of messages where each message summary is shown.
4	RETR	This command helps to select a mailbox to access the messages.

5	DELE	It is used to delete a message.
6	RSET	It is used to reset the session to its initial state.
7	QUIT	It is used to log off the session.

The poolib module from Python's standard library defines POP3 and POP3_SSL classes. POP3 class encapsulates a connection to a POP3 server and implements the protocol as defined in RFC 1939. POP3_SSL classsupports POP3 servers that use SSL as an underlying protocol layer. POP3 protocol is obsolescent as its implementation quality of POP3 servers is quite poor. If your mailserver supports IMAP, it is recommended to use the imaplib.IMAP4 class.

Both classes have following methods defined

- getwelcome()
  Returns the greeting string sent by the POP3 server.
- user(username)
  Send user command, response should indicate that a password is required.
- pass_(password)
  Send password.
- Stat()
  Get mailbox status. The result contains 2 integers: (message count, mailbox size).
- list()
  Request message list, result is in the form (response, ['mesg_num octets', ...], octets).
- retr()
  Retrieve message of specified index, and set its seen flag.
- Dele()
  Flag message number which for deletion.
- Top()
  Retrieves the message header plus number of lines of the message after the header of message
- quit(): Signoff
  commit changes, unlock mailbox, drop connection.

Example

Following code retrieves all unread messages from gmail's POP server.

```
import poplib
box = poplib.POP3_SSL('pop.googlemail.com', '995')
box.user("YourGmailUserName")
box.pass_('YourPassword')
N = len(box.list()[1])
for i in range(N):
   for msg in box.retr(i+1)[1]:
      print (msg)
box.quit()
```

UNIT – V

DataBase and GUI Programming

1. DBM Database

The dbm module provides an interface to the Unix "(n)dbm" library. Dbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a dbm object doesn't print the keys and values, and the items() and values() methods are not supported.

This module can be used with the "classic" ndbm interface, the BSD DB compatibility interface, or the GNU GDBM compatibility interface. On Unix, the configure script will attempt to locate the appropriate header file to simplify building this module.

The module defines the following:

exception dbm.error

Raised on dbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

dbm.library

Name of the ndbm implementation library used.

dbm.open(filename[, flag[, mode]])

Open a dbm database and return a dbm object. The filename argument is the name of the database file (without the .dir or .pag extensions; note that the BSD DB implementation of the interface will append the extension .db and only create one file).

The optional flag argument must be one of these values:

Value	Meaning
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing

Value	Meaning
'c'	Open database for reading and writing, creating it if it doesn’t exist
'n'	Always create a new, empty database, open for reading and writing

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

In addition to the dictionary-like methods, dbm objects provide the ollowing method:

dbm.close()

Close the dbm database.

The dbm package in Python's built-in library provides a dictionary like an interface DBM style databases. The dbm library is a simple database engine, written by Ken Thompson. DBM stands for DataBase Manager, used by UNIX operating system, the library stores arbitrary data by use of a single key (a primary key) in fixed-size buckets and uses hashing techniques to enable fast retrieval of the data by key.

There are following modules in dbm package −

- The dbm.ndbm module provides an interface to the Unix “(n)dbm” library. Dbm objects behave like dictionaries, with keys and values should be stored as bytes. The module doesn't support and the items() and values() methods.
- The dbm.dumb module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as dbm.gnu no external library is required. As with other persistent mappings, the keys and values are always stored as bytes.

- These modules are internally used by Python's shelve module. As in the case of shelve database, user-specified database name carries '.dir' postfix. The dbm object's whichdb() function tells which implementation of dbm is available on current Python installation.

```
>>> dbm.whichdb('mydbm.db')
dbm.dumb'
>>> db = dbm.open('mydbm.db','n')
>>> db['name'] = Rajani Deshmukh'
>>> db['address'] = 'Shivajinagar Pune'
>>> db['PIN'] = '431001'
>>> db.close()
```

A dbm object is a dictionary like an object, just as a shelf object. Hence all dictionary operations can be performed. The dbm object can invoke get(),pop(), append(0 and update() methods. Following code opens 'mydbm.db' with 'r' flag and iterates over the collection of key-value pairs.

```
>>> db = dbm.open('mydbm.db','r')
>>> for k,v in db.items():
print (k,v)
b'name' : Rajani Deshmukh'
b'address' : b'Shivajinagar Pune'
b'PIN' : b'431001'
```

dbm objects also provide the following methods −

sync(): Synchronize the on-disk directory and data files. This method is called by the Shelve.sync() method.

close(): Close the dbm database.

gnu dbm objects have the following methods –

firstkey()

It's possible to loop over every key in the database using this method and the nextkey() method. This method returns the starting key.

gdbm.nextkey(key): Returns the key that follows key in the traversal.

gdbm.reorganize(): this function will reorganize the database. gnu dbm objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

dbm — Unix Key-Value Databases

Purpose: dbm provides a generic dictionary-like interface to DBM-style, string-keyed databases.dbm is a front-end for DBM-style databases that use simple string values as keys to access records containing strings. It uses whichdb() to identify databases, then opens them with the appropriate module. It is used as a back-end for shelve, which stores objects in a DBM database using pickle.

Database Types

Python comes with several modules for accessing DBM-style databases. The default implementation selected depends on the libraries available on the current system and the options used when Python was compiled. Separate interfaces to the specific implementations allow Python programs to exchange data with programs in other languages that do not automatically switch between available formats, or to write portable data files that will work on multiple platforms.

dbm.gnu

dbm.gnu is an interface to the version of the dbm library from the GNU project. It works the same as the other DBM implementations described here, with a few changes to the flags supported by open().

Besides the standard 'r', 'w', 'c', and 'n' flags, dbm.gnu.open() supports:

'f' to open the database in fast mode. In fast mode, writes to the database are not synchronized.

's' to open the database in synchronized mode. Changes to the database are written to the file as they are made, rather than being delayed until the database is closed or synced explicitly.

'u' to open the database unlocked.

dbm.ndbm

dbm.ndbm module provides an interface to the Unix ndbm implementations of the dbm format, depending on how the module was configured during compilation. The module attribute library identifies the name of the library configure was able to find when the extension module was compiled.

dbm.dumb

The dbm.dumb module is a portable fallback implementation of the DBM API when no other implementations are available. No external dependencies are required to use dbm.dumb, but it is slower than most other implementations.

Creating a New Database

The storage format for new databases is selected by looking for usable versions of each of the sub-modules in order.

dbm.gnu
dbm.ndbm
dbm.dumb

The open() function takes flags to control how the database file is managed. To create a new database when necessary, use 'c'. Using 'n' always creates a new database, overwriting an existing file.

```
dbm_new.py
import dbm
with dbm.open('/tmp/example.db', 'n') as db:
    db['key'] = 'value'
    db['today'] = 'Sunday'
    db['author'] = 'Doug'
```

In this example, the file is always re-initialized.

```
$ python3 dbm_new.py
```

whichdb() reports the type of database that was created.

```
dbm_whichdb.py
import dbm
print(dbm.whichdb('/tmp/example.db'))
```

Output from the example program will vary, depending on which modules are installed on the system.

```
$ python3 dbm_whichdb.py
```

dbm.ndbm

Opening an Existing Database

To open an existing database, use flags of either 'r' (for read-only) or 'w' (for read-write). Existing databases are automatically given to whichdb() to identify, so it as long as a file can be identified, the appropriate module is used to open it.

```
dbm_existing.py
import dbm
with dbm.open('/tmp/example.db', 'r') as db:
    print('keys():', db.keys())
    for k in db.keys():
        print('iterating:', k, db[k])
    print('db["author"] =', db['author'])
```

Once open, db is a dictionary-like object. New keys are always converted to byte strings when added to the database, and returned as byte strings.

```
$ python3 dbm_existing.py

keys(): [b'key', b'today', b'author']
iterating: b'key' b'value'
iterating: b'today' b'Sunday'
iterating: b'author' b'Doug'
db["author"] = b'Doug'
```

Error Cases

The keys of the database need to be strings.

```
dbm_intkeys.py
import dbm
with dbm.open('/tmp/example.db', 'w') as db:
    try:
        db[1] = 'one'
    except TypeError as err:
        print(err)
```

Passing another type results in a TypeError.

```
$ python3 dbm_intkeys.py
dbm mappings have bytes or string keys only
```

Values must be strings or None.

```
dbm_intvalue.py
import dbm
with dbm.open('/tmp/example.db', 'w') as db:
    try:
        db['one'] = 1
    except TypeError as err:
        print(err)
```

A similar TypeError is raised if a value is not a string.

```
$ python3 dbm_intvalue.py
dbm mappings have byte or string elements only
```

2, SQL Database

All software applications interact with data, most commonly through a database management system (DBMS). Some programming languages come with modules that you can use to interact with a DBMS, while others require the use of third-party packages. In this tutorial, you'll explore the different Python SQL libraries that you can use. You'll develop a straightforward application to interact with SQLite, MySQL, and PostgreSQL databases.

In this tutorial, you'll learn how to:

- Connect to different database management systems with Python SQL libraries
- Interact with SQLite, MySQL, and PostgreSQL databases
- Perform common database queries using a Python application
- Develop applications across different databases using a Python script

Python can be used in database applications. One of the most popular databases is MySQL.

MySQL Database

To be able to experiment with the code examples in this tutorial, you should have MySQL installed on your computer. You can download a free MySQL database

Install MySQL Driver

Python needs a MySQL driver to access the MySQL database.

- In this tutorial we will use the driver "MySQL Connector".
- We recommend that you use PIP to install "MySQL Connector".
- PIP is most likely already installed in your Python environment.

Test MySQL Connector

To test if the installation was successful, or if you already have "MySQL Connector" installed, create a Python page with the following content:

Demo_mysql_test.py:

```
import mysql.connector
```

Create Connection

Start by creating a connection to the database. Use the username and password from your MySQL database:

demo_mysql_connection.py:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)
print(mydb)
```

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

create a database named "mydatabase":

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE DATABASE mydatabase")
```

Check if Database Exists

You can check if a database exist by listing all databases in your system by using the "SHOW DATABASES" statement:

Return a list of your system's databases:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)
mycursor = mydb.cursor()
mycursor.execute("SHOW DATABASES")

for x in mycursor:
  print(x)
```

Creating a Table

To create a table in MySQL, use the "CREATE TABLE" statement.Make sure you define the name of the database when you create the connection
Create a table named "customers":

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))")
```

Check if Table Exists

You can check if a table exist by listing all tables in your database with the "SHOW TABLES" statement:
Return a list of your system's databases:

```
import mysql.connector
mydb = mysql.connector.connect(
```

```
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SHOW TABLES")

for x in mycursor:
  print(x)
```

Primary Key

When creating a table, you should also create a column with a unique key for each record. This can be done by defining a PRIMARY KEY. We use the statement "INT AUTO_INCREMENT PRIMARY KEY" which will insert a unique number for each record. Starting at 1, and increased by one for each record.

Create primary key when creating the table:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255), address VARCHAR(255))")
```

If the table already exists, use the ALTER TABLE keyword:

Create primary key on an existing table:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
```

```
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("ALTER TABLE customers ADD COLUMN id
INT AUTO_INCREMENT PRIMARY KEY")
```

Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

Insert a record in the "customers" table:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)

mydb.commit()
print(mycursor.rowcount, "record inserted.")
```

Insert Multiple Rows

To insert multiple rows into a table, use the executemany() method.The second parameter of the executemany() method is a list of tuples, containing the data you want to insert:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
```

```
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = [
  ('Peter', 'Lowstreet 4'),
  ('Amy', 'Apple st 652'),
  ('Hannah', 'Mountain 21'),
  ('Michael', 'Valley 345'),
  ('Sandy', 'Ocean blvd 2'),
  ('Betty', 'Green Grass 1'),
  ('Richard', 'Sky st 331'),
  ('Susan', 'One way 98'),
  ('Vicky', 'Yellow Garden 2'),
  ('Ben', 'Park Lane 38'),
  ('William', 'Central st 954'),
  ('Chuck', 'Main Road 989'),
  ('Viola', 'Sideway 1633')
]
mycursor.executemany(sql, val)

mydb.commit()
        print(mycursor.rowcount, "was inserted.")
```

Select From a Table

To select from a table in MySQL, use the "SELECT" statement:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
```

```
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

Selecting Columns

To select only some of the columns in a table, use the "SELECT" statement followed by the column name(s):

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT name, address FROM customers")
myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

Delete Record

You can delete records from an existing table by using the "DELETE FROM" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
```

```
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "DELETE FROM customers WHERE address = 'Mountain 21'"
mycursor.execute(sql)
mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

Delete a Table

You can delete an existing table by using the "DROP TABLE" statement.

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "DROP TABLE customers"

mycursor.execute(sql)
```

Update Table

You can update existing records in a table by using the "UPDATE" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
```

```
mycursor = mydb.cursor()
sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = Valley 345'"
mycursor.execute(sql)

mydb.commit()
print(mycursor.rowcount, "record(s) affected")
```

Join Two or More Tables

You can combine rows from two or more tables, based on a related column between them, by using a JOIN statement.

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()

sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  INNER JOIN products ON users.fav = products.id"

mycursor.execute(sql)
myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties −

- Atomicity − Either a transaction completes or nothing happens at all.
- Consistency − A transaction must start in a consistent state and leave the system in a consistent state.
- Isolation − Intermediate results of a transaction are not visible outside the current transaction.
- Durability − Once a transaction was committed, the effects are persistent, even after a system failure.

Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

Sl.No.	Exception & Description
1	**Warning** Used for non-fatal issues. Must subclass StandardError.
2	**Error** Base class for errors. Must subclass StandardError.
3	**InterfaceError** Used for errors in the database module, not the database itself. Must subclass Error.
4	**DatabaseError** Used for errors in the database. Must subclass Error.
6	**OperationalError** Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally

	outside of the control of the Python scripter**.**
7	**IntegrityError** Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys**.**
8	**InternalError** Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active**.**
9	**ProgrammingError** Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you**.**
10	**NotSupportedError** Subclass of DatabaseError that refers to trying to call unsupported functionality.

GUI PROGRAMMING USING TKinder

1. **Introduction**

   There are many GUI modules available for developing GUI programs in Python. We have used the turtle module for drawing geometric shapes. Turtle is easy to use and is an
   effective pedagogical tool for introducing the fundamentals of programming to beginners.

   However, we cannot use turtle to create graphical user interfaces. Here we introduces Tkinter, which will enable you to develop GUI projects. Tkinter is not only a useful tool for developing GUI projects, but it is also a valuable pedagogical tool for learning object oriented programming.

2. **Getting Started with TKinder**

   The tkinter module contains the classes for creating GUIs. The Tk class creates a window for holding GUI widgets (i.e., visual components**).**

Here is simple example for TKinder

```
from tkinter import * # Import all definitions from tkinter
window = Tk()       # Create a window
label = Label(window, text = "Welcome to Python") # Create a label
button = Button(window, text = "Click Me")          # Create a button
label.pack()          # Place the label in the window
button.pack()         # Place the button in the window
window.mainloop() # Create an event loop
```

When you run the program, a label and a button appear in the Tkinter window, will be as shown.

Whenever you create a GUI-based program in Tkinter, you need to import the tkinter module and create a window by using the Tk class. The asterisk (*) 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 :

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

constructs a label with the text **Welcome to Python** that is contained in the window. The statement :

```
label.pack()
```

Label in the container using a pack manager. 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

```
window.mainloop()
```

The statement creates an event loop. The event loop processes events continuously until we close the main window, as shown in Figure 2.1.

.

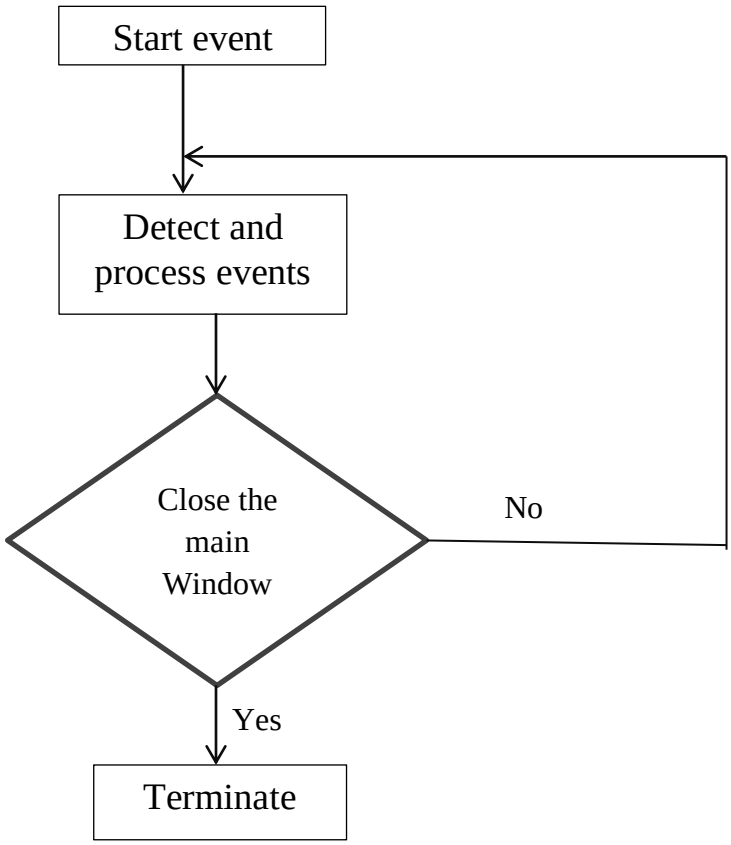


FIG 2.1 A Tkinter GUI program listens and processes events in a continuous loop.

3. Processing Events

A Tkinter widget can be bound to a function, which is called when an event occurs. The Button widget is a good way to demonstrate the basics of event-driven programming. Now we see simple example on processing events.

When the user clicks a button, the program should process this event. It should enable this action by defining a processing function and binding the function to the button, as shown.

ProcessButtonEvent.py

```
from tkinter import *  # Import all definitions from tkinter
def processOK():
    print("OK button is clicked")
def processCancel():
    print("Cancel button is clicked")
window = Tk() # Create a window
```

```
btOK = Button(window, text = "OK", fg = "red", command =
                    processOK)
 btCancel = Button(window, text = "Cancel", bg = "yellow",
 command = 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
```

When you run the program, two buttons appear, as shown in Figure 3.1a. we can watch the events being processed and see their associated messages in the command window in Figure 3.1b.

Fig 3.1(a) displays two buttons in a window.

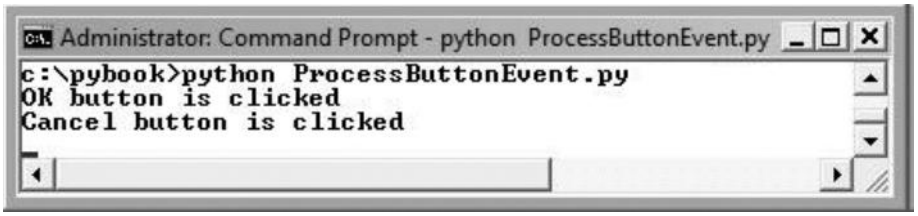


Fig 3.1(b) Watching events being processed in the command window.

The program defines the functions processOK and processCancel. These functions are bound to the buttons when the buttons are constructed. These functions are known as callback functions, or handlers.
The following statement

```
btOK = Button(window, text = "OK", fg = "red", command = processOK)
```

binds the OK button to the processOK function, which will be called when the button is clicked. The fg option specifies the button's foreground color

and the bg option specifies its background color. By default, fg is black and bg is gray for all widgets.

Now we can also write this program by placing all the functions in one class, as shown.

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

class ProcessButtonEvent:
def _ _init_ _(self):
window = Tk()     # Create a window
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() # Create an object to invoke _ _init_ _ method
```

The program defines a class for creating the GUI in the _ _init_ _ method. The functions processOK and processCancel are now instance methods in the class, so they are called by self.processOK and self.processCancel .
There are two advantages of defining a class for creating a GUI and processing GUI events.

- We can reuse the class in the future.
- Defining all the functions as methods enables them to access instance data fields in the class.

4. The Widget Classes

Tkinter's GUI classes define common GUI widgets such as buttons, labels, radio buttons, check buttons, entries, canvases, and others.

Table 4.1 describes the core widget classes Tkinter provides :

TABLE 4.1 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. weu can specify a foreground color, background color, font, and cursor style when constructing a widget

Color

To specify a color, use either a color name (such as red, yellow, green, blue, white, black, purple) or explicitly specify the red, green, and blue (RGB)

color components by using a string #RRGGBB, where RR, GG, and BB are hexadecimal representations of the red, green, and blue values, respectively.

Font

You can specify a font in a string that includes the font name, size, and style. Here are some examples:

Times 10 bold
Helvetica 10 bold italic
CourierNew 20 bold italic
Courier 20 bold italic overstrike underline

Text Formatting

By default, the text in a label or a button is centered. You can change its alignment by using the justify option with the named constants LEFT, CENTER, or RIGHT. We can also display the text in multiple lines by inserting the newline character \n to separate lines of text.

Mouse Cursor

We can specify a particular style of mouse cursor by using the cursor option with string values such as arrow (the default), circle, cross, plus, or some other shape.

Change Properties

When we construct a widget, we can specify its properties such as fg, bg, font, cursor, text, and command in the constructor. We can change the widget's properties by using the following syntax:

widgetName["propertyName"] = newPropertyValue

For example, the following code creates a button and its text property is changed to Hide, bg property to red, and fg to #AB84F9. #AB84F9 is a color specified in the form of changed to Hide, bg property to red, and fg to #AB84F9. #AB84F9 is a color specified in the form of RRGGBB.

```
btShowOrHide = Button(window, text = "Show", bg = "white")
btShowOrHide["text"] = "Hide"
btShowOrHide["bg"] = "red"
```

```
btShowOrHide["fg"] = "#AB84F9" # Change fg color to #AB84F9
btShowOrHide["cursor"] = "plus" # Change mouse cursor to plus
btShowOrHide["justify"] = LEFT # Set justify to LEFT
```

Each class comes with a substantial number of methods.

Now we see a program that uses the widgets Frame, Button, Checkbutton, Radiobutton, Label, Entry (also known as a text field), Message, and Text (also known as a text area).

Example 1: WidgetsDemo.py

```
from tkinter import * # Import all definitions from tkinter
  class WidgetsDemo:
        def _ _init_ _(self):
         window = Tk() # Create a window
         window.title("Widgets Demo") # Set a title

    # Add a check button, and a radio button to frame1
    frame1 = Frame(window) # Create and add a frame to window
    frame1.pack()
     self.v1 = IntVar()
    cbtBold = Checkbutton(frame1, text = "Bold",
         variable = self.v1 , command = self.processCheckbutton )
    self.v2 = IntVar()
     rbRed = Radiobutton(frame1, text = "Red", bg = "red",
              variable = self.v2, value = 1,
             command = self.processRadiobutton )
    rbYellow = Radiobutton(frame1, text = "Yellow",
          bg = "yellow", variable = self.v2, value = 2,
          command = self.processRadiobutton )
    cbtBold.grid(row = 1, column = 1)
    rbRed.grid(row = 1, column = 2)
    rbYellow.grid(row = 1, column = 3)

    # Add a label, an entry, a button, and a message to frame1
    frame2 = Frame(window) # Create and add a frame to window
    frame2.pack()
    label = Label(frame2, text = "Enter your name: ")
```

```
        self.name = StringVar()
        entryName = Entry(frame2, textvariable = self.name )
        btGetName = Button(frame2, text = "Get Name",
            command = self.processButton)
        message = Message(frame2, text = "It is a widgets demo")
        label.grid(row = 1, column = 1)
        entryName.grid(row = 1, column = 2)
        btGetName.grid(row = 1, column = 3)
        message.grid(row = 1, column = 4)

        # Add text
         text = Text(window) # Create and add text to the window
         text.pack()
        text.insert(END,
        "Tip\nThe best way to learn Tkinter is to read ")
        text.insert(END,
        "these carefully designed examples and use them ")
         text.insert(END, "to create your applications.")

         window.mainloop() # Create an event loop

    def processCheckbutton(self):
            print("check button is "
             + ("checked " if self.v1.get() == 1 else "unchecked"))
    def processRadiobutton(self):
            print(("Red" if self.v2.get() == 1 else "Yellow")
             + " is selected " )
    def processButton(self):
        print("Your name is " + self.name.get())

WidgetsDemo() # Create GUI
```

When you run the program, the widgets are displayed as shown in Figure 4.1a. As we click the Bold button, select the Yellow radio button, and type

in “Johnson,” we can watch the events being processed and see their associated messages in the command window in Figure 4.1b.

The program creates the window and invokes its title method to set a title. The Frame class is used to create a frame named frame1 and the parent container for the frame is the window . This frame is used as the parent container for a check button and two radio buttons

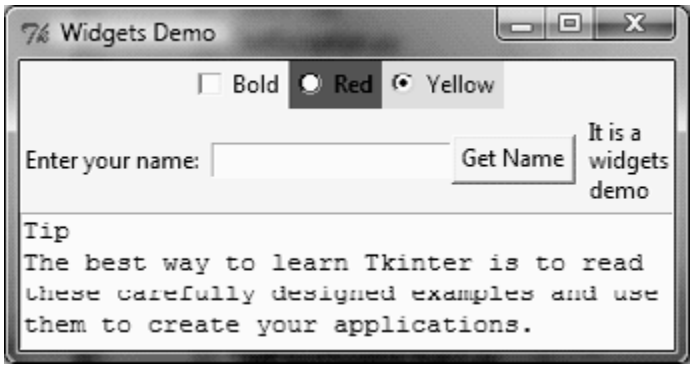


Fig 4.1(a) The widgets are displayed in the user interface.

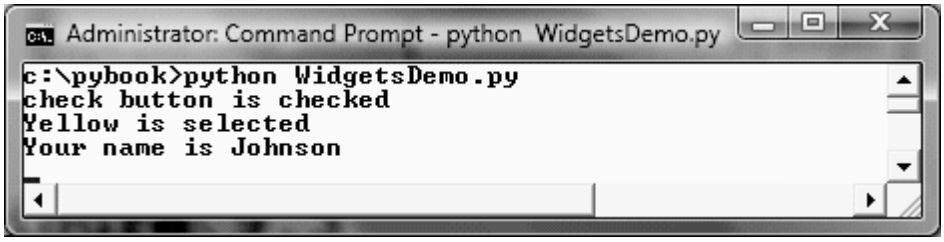


Fig 4.1(b) Watching events being processed

We use an entry (text field) for entering a value. The value must be an object of IntVar, DoubleVar, or StringVar representing an integer, a float, or a string, respectively. IntVar, DoubleVar, and StringVar are defined in the tkinter module.

Explanation

The program creates a check button and associates it with the variable v1. v1 is an instance of IntVar. v1 is set to 1 if the check button is checked, or 0 if it isn’t checked. When the check button is clicked, Python invokes the processCheckbutton method .

The program then creates a radio button and associates it with an IntVar variable, v2. v2 is set to 1 if the Red radio button is selected, or 2 if the Yellow radio button is checked. We can define any integer or string values when constructing a radio button. When either of the two buttons is clicked,

the processRadiobutton method is invoked. The grid geometry manager is used to place the check button and radio buttons into frame1. These three widgets are placed in the same row and in columns 1, 2, and 3, respectively.
The program creates another frame, frame2 , for holding a label, an entry, a button, and a message widget. Like frame1, frame2 is placed inside the window. An entry is created and associated with the variable name of the StringVar type for storing the value in the entry. When you click the Get Name button, the processButton method displays the value in the entry . The Message widget is like a label except that it automatically wraps the words and displays them in multiple lines. The grid geometry manager is used to place the widget in frame2. These widgets are placed in the same row and in columns 1, 2, 3, and 4, respectively .
The program creates a Text widget for displaying and editing text. It is placed inside the window. We can use the insert method to insert text into this widget. The END option specifies that the text is inserted into the end of the current content.
Next we a program that lets the user change the color, font, and text of a label as shown in figure 4.2.

Example 2 ChangeLabelDemo.py

```
from tkinter import * # Import all definitions from tkinter
class ChangeLabelDemo:
    def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Change Label Demo") # Set a title

        # Add a label to frame1
        frame1 = Frame(window) # Create and add a frame to window
        frame1.pack()
        self.lbl = Label(frame1, text = "Programming is fun")
        self.lbl.pack()
        # Add a label, entry, button, two radio buttons to frame2
        frame2 = Frame(window) # Create and add a frame to window
        frame2.pack()
        label = Label(frame2, text = "Enter text: ")
```

```
        self.msg = StringVar()
        entry = Entry(frame2, textvariable = self.msg)
        btChangeText = Button(frame2, text = "Change Text",
            command = self.processButton)
        self.v1 = StringVar()
        rbRed = Radiobutton(frame2, text = "Red", bg = "red",
            variable = self.v1, value = 'R',
        command = self.processRadiobutton)
        rbYellow = Radiobutton(frame2, text = "Yellow",
            bg = "yellow", variable = self.v1, value = 'Y',
            command = self.processRadiobutton)
 label.grid(row = 1, column = 1)
 entry.grid(row = 1, column = 2)
 btChangeText.grid(row = 1, column = 3)
 rbRed.grid(row = 1, column = 4)
 rbYellow.grid(row = 1, column = 5)
window.mainloop() # Create an event loop

def processRadiobutton(self):
    if  self.v1.get() == 'R':
        self.lbl["fg"] = "red"
    elif self.v1.get() == 'Y':
        self.lbl["fg"] = "yellow"
def processButton(self) :
    self.lbl["text"] = self.msg.get() # New text for the label

 ChangeLabelDemo() # Create GUI
```

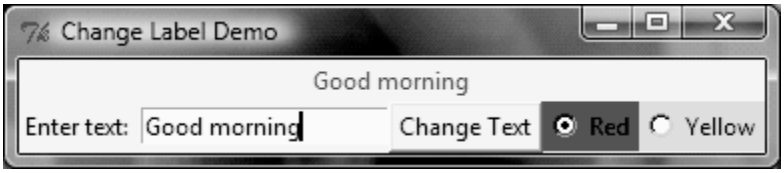


Fig 4.2 The program changes the label's text and fg properties dynamically.

When you select a radio button, the label's foreground color changes. If you enter new text in the entry field and click the Change Text button, the new text appears in the label.

Explanation

The program creates the window and invokes its title method to set a title.

The Frame class is used to create a frame named frame1 and the parent container for the frame is the window. This frame is used as the parent container for a label created . Because the label is a data field in the class, it can be referenced in a callback function.

The program creates another frame, frame2, for holding a label, an entry, a button, and two radio buttons. Like frame1, frame2 is placed inside the window. An entry is created and associated with the variable msg of the StringVar type for storing the value in the entry. When you click the Change Text button, the processButton method sets a new text entry for the label in frame1, using the text in the entry. Two radio buttons are created and associated with a StringVar variable, v2. v2 is set to R if the Red radio button is selected, or to Y if the Yellow radio button is clicked. When the user clicks either of the two buttons, Python invokes the processRadiobutton method to change the label's foreground color in frame1.

5. Canvas

We use the Canvas widget for displaying shapes. We 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.

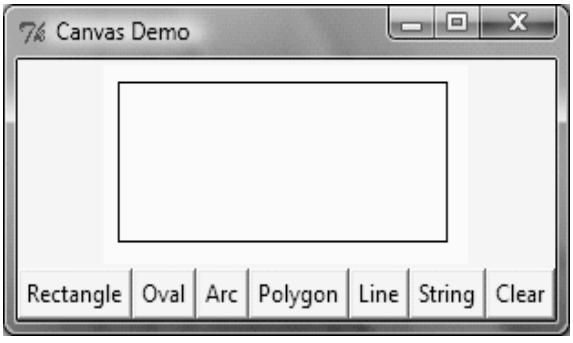
Program shows how to use the Canvas widget. The program displays a rectangle, an oval, an arc, a polygon, a line, and a text string. The objects are all controlled by buttons as shown in figure 5.1.

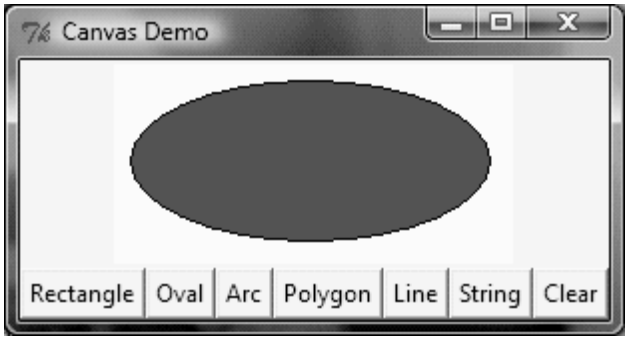
Program 1 CanvasDemo.py

```
from tkinter import * # Import all definitions from tkinter
class CanvasDemo:
    def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Canvas Demo") # Set title

    # Place canvas in the window
```

```
self.canvas = Canvas(window, width = 200, height = 100,
        bg = "white")
self.canvas.pack()
 # Place buttons in frame
 frame = Frame(window)
 frame.pack()
 btRectangle = Button(frame, text = "Rectangle",
        command = self.displayRect)
btOval = Button(frame, text = "Oval",
        command = self.displayOval)
 btArc = Button(frame, text = "Arc",
        command = self.displayArc)
btPolygon = Button(frame, text = "Polygon",
        command = self.displayPolygon)
 btLine = Button(frame, text = "Line",
       command = self.displayLine)
btString = Button(frame, text = "String",
        command = self.displayString)
btClear = Button(frame, text = "Clear",
        command = self.clearCanvas)
btRectangle.grid(row = 1, column = 1)
btOval.grid(row = 1, column = 2)
btArc.grid(row = 1, column = 3)
 btPolygon.grid(row = 1, column = 4)
btLine.grid(row = 1, column = 5)
btString.grid(row = 1, column = 6)
 btClear.grid(row = 1, column = 7)

window.mainloop() # Create an event loop

# Display a rectangle
 def displayRect(self):
        self.canvas.create_rectangle(10, 10, 190, 90,
        tags = "rect")
# Display an oval
```

```
 def displayOval(self):
        self.canvas.create_oval(10, 10, 190, 90, fill = "red",
        tags = "oval")
# Display an arc
def displayArc(self):
        self.canvas.create_arc(10, 10, 190, 90, start = 0,
        extent = 90, width = 8, fill = "red", tags = "arc")
# Display a polygon
def displayPolygon(self):
        self.canvas.create_polygon(10, 10, 190, 90, 30, 50,
        tags = "polygon")
# Display a line
def displayLine(self):
        self.canvas.create_line(10, 10, 190, 90, fill = "red",
        tags = "line")
         self.canvas.create_line(10, 90, 190, 10, width = 9,
                 arrow = "last", activefill = "blue", tags = "line")
# Display a string
 def displayString(self):
                self.canvas.create_text(60, 40, text = "Hi, I am a
                string", font = "Times 10 bold underline", tags =
                "string")
# Clear drawings
def clearCanvas(self):
        self.canvas.delete("rect", "oval", "arc", "polygon",
                "line", "string")
 CanvasDemo() # Create GUI
```





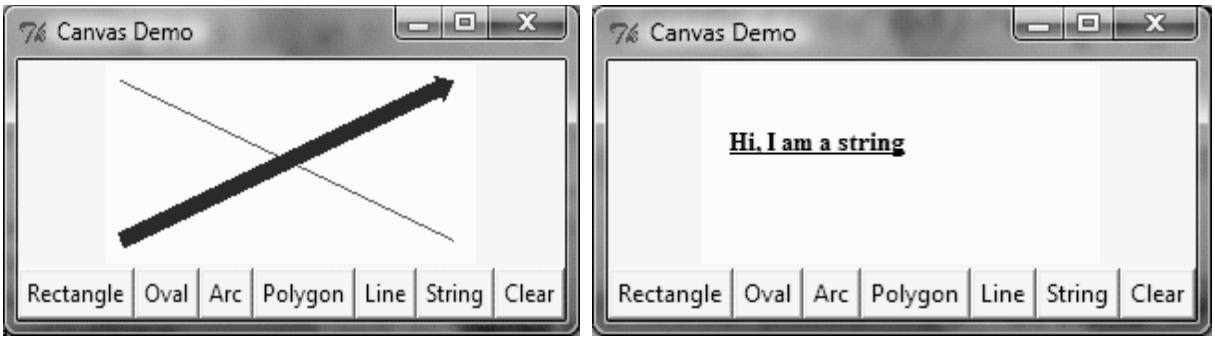


Fig 5.1 The geometrical shapes and strings are drawn on the canvas

Explanation

The program creates a window and sets its title. A Canvas widget is created within the window with a width of 200 pixels, a height of 100 pixels, and a background color of white. Seven buttons—labeled with the text Rectangle, Oval, Arc, Polygon, Line, String, and Clear—are created . The grid manager places the buttons in one row in a frame .

To draw graphics, you need to tell the widget where to draw. Each widget has its own coordinate system with the origin (0, 0) at the upper-left corner. The x-coordinate increases to the right, and the y-coordinate increases downward. Note that the Tkinter coordinate system differs from the conventional coordinate system, as shown Figure 5.2.

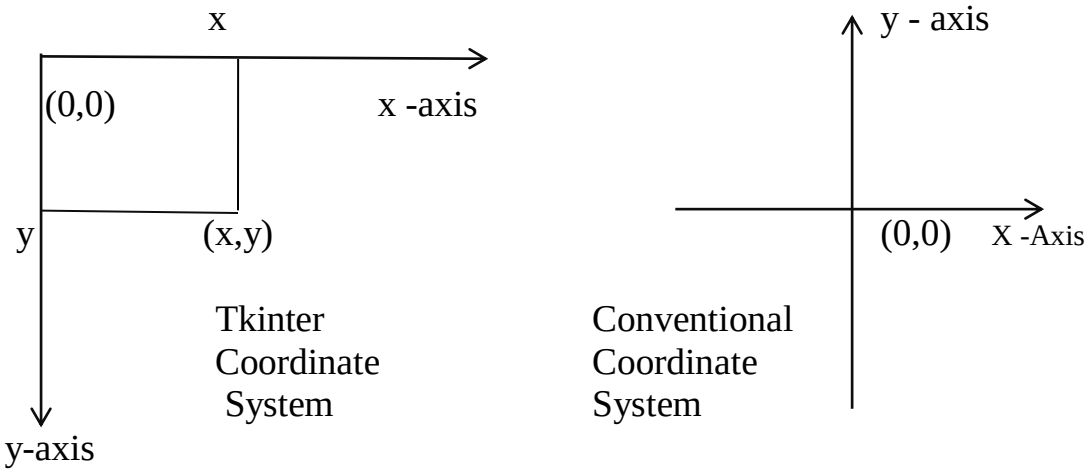


Fig. 5.2 The Tkinter coordinate system is measured in pixels, with (0, 0) at its upperleft corner.

The methods create_rectangle, create_oval, create_arc, create_polygon, and create_line are used to draw rectangles, ovals, arcs, polygons, and lines, as illustrated in Figure 5. The create_text method is used to draw a text

string. Note that the horizontal and vertical center of the text is displayed at (x, y) for create_text(x, y, text) as shown in Figure 5.3. All the drawing methods use the tags argument to identify the drawing. These tags are used in the delete method for clearing the drawing from the canvas.

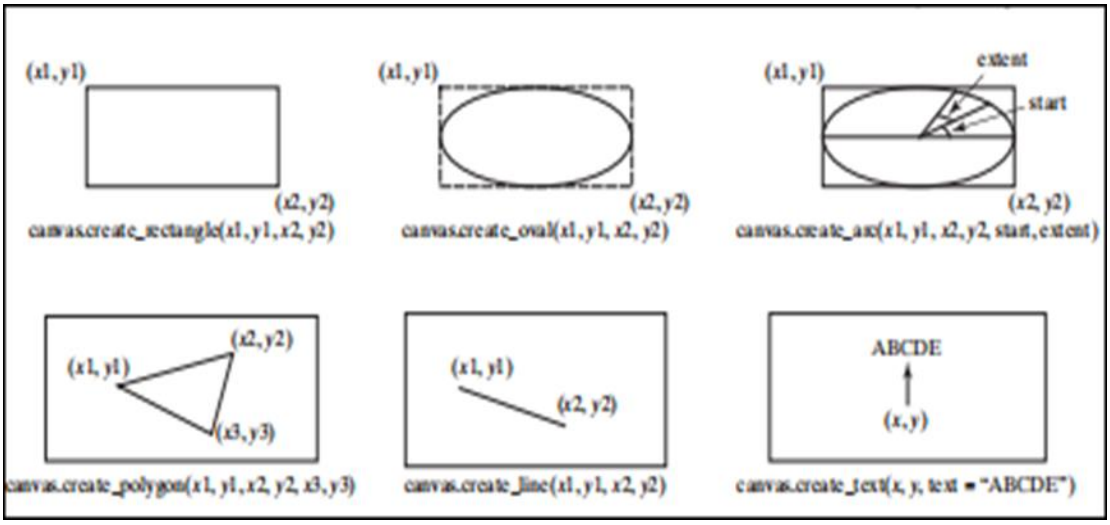


Fig. 5.3 The Canvas class contains the methods for drawing graphics.

The width argument can be used to specify the pen size in pixels for drawing the shapes. The arrow argument can be used with create_line to draw a line with an arrowhead. The arrowhead can appear at the start, end, or both ends of the line with the argument value first, end, or both. The activefill argument makes the shape change color when you move the mouse over it.

6. The Geometry Managers

Tkinter uses a geometry manager to place widgets inside a container. Tkinter supports three geometry managers: the grid manager, the pack manager, and the place manager.

6.1The Grid Manager

The grid manager places widgets into the cells of an invisible grid in a container. We can place a widget in a specified row and column. We can also use the rowspan and columnspan parameters to place a widget in multiple rows and columns. Program below uses the grid manager to lay out a group of widgets, and output is shown in Figure 6.1.

PROGRAM :GridManagerDemo.py

```
from tkinter import * # Import all definitions from tkinter
class GridManagerDemo:
    window = Tk() # Create a window
    window.title("Grid Manager Demo") # Set title
    message = Message(window, text =
    "This Message widget occupies three rows and two Columns")
    message.grid(row = 1, column = 1, , columnspan = 2)
Label(window, text = "First Name:").grid(row = 1, column = 3)
    Entry(window).grid(row = 1, column = 4, , pady = 5)
Label(window, text = "Last Name:").grid(row = 2, column = 3)
    Entry(window).grid(row = 2, column = 4)
    Button(window, text = "Get Name").grid(row = 3,
    padx = 5, pady = 5, column = 4, )
window.mainloop() # Create an event loop

GridManagerDemo() # Create GUI
```

Fig 6.1 The grid manager was used to place these widgets.

Explanation

The Message widget is placed in row 1 and column 1 and it expands to three rows and two columns. The Get Name button uses the sticky = E option to stick to the east in the cell so that it is right aligned with the Entry widgets in the same column. The sticky option defines how to expand the widget if the resulting cell is larger than the widget itself. The sticky option can be any combination of the named constants S, N, E, and W, or NW, NE, SW, and

SE. The padx and pady options pad the optional horizontal and vertical space in a cell. We can also use the ipadx and ipady options to pad the optional horizontal and vertical space inside the widget borders.

6.2 The Pack Manager

The pack manager can place widgets on top of each other or place them side by side. You can also use the fill option to make a widget fill its entire container. Program below displays three labels, as shown in Figure 6.2 (a). These three labels are packed on top of each other. The red label uses the option fill with value BOTH and expand with value 1. The fill option uses named constants X, Y, or BOTH to fill horizontally, vertically, or both ways. The expand option tells the pack manager to assign additional space to the widget box. If the parent widget is larger than necessary to hold all the packed widgets, any extra space is distributed among the widgets whose expand option is set to a nonzero value.

Program :PackManagerDemo.py

```
from tkinter import * # Import all definitions from tkinter
class PackManagerDemo:
  def _ _init_ _(self):
      window = Tk() # Create a window
      window.title("Pack Manager Demo 1") # Set title

      Label(window, text = "Blue", bg = "blue").pack()
      Label(window, text = "Red", bg = "red").pack(
     fill = BOTH, expand = 1)
     Label(window, text = "Green", bg = "green").pack(
     fill = BOTH)
window.mainloop() # Create an event loop

PackManagerDemo() # Create GUI
```

Fig. 6.2 (a) The pack manager uses the fill option to fill the container. (b) You can place widgets side by side.

Program below displays the three labels shown in Figure 6.2b. These three labels are packed side by side using the side option. The side option can be LEFT, RIGHT, TOP, or BOTTOM. By default, it is set to TOP.

```
PackManagerDemoWithSide.py
from tkinter import * # Import all definitions from tkinter
class PackManagerDemoWithSide:
        window = Tk() # Create a window
        window.title("Pack Manager Demo 2") # Set title

        Label(window, text = "Blue", bg = "blue").pack( )
        Label(window, text = "Red", bg = "red").pack(
       side = LEFT , fill = BOTH, expand = 1)
        Label(window, text = "Green", bg = "green").pack(
       side = LEFT , fill = BOTH)

 window.mainloop() # Create an event loop

PackManagerDemoWithSide() # Create GUI
```

6.3 The Place Manager

The place manager places widgets in absolute positions. Program below displays the three labels shown in Figure 6.3.

Program :PlaceManagerDemo.py

```
from tkinter import * # Import all definitions from tkinter
class PlaceManagerDemo:
def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Place Manager Demo") # Set title

        Label(window, text = "Blue", bg = "blue"). place(
       x = 20, y = 20)
        Label(window, text = "Red", bg = "red").place(
        x = 50, y = 50)
        Label(window, text = "Green", bg = "green").place(
        x = 80, y = 80)

window.mainloop() # Create an event loop

PlaceManagerDemo() # Create GUI
```

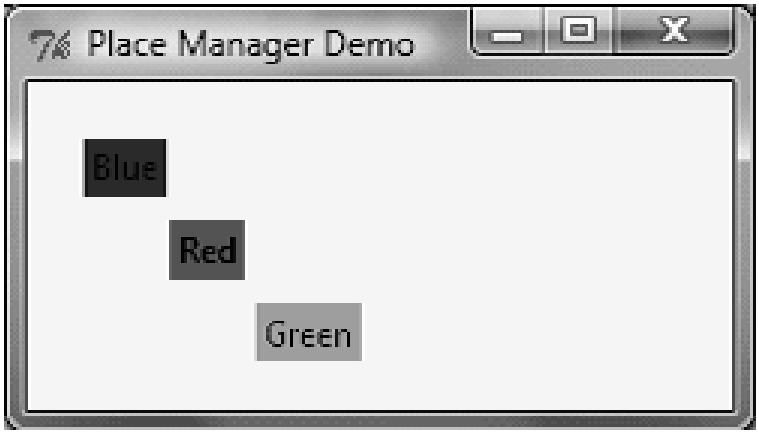


Fig 6.3 The place manager places widgets in absolute positions.

The upper-left corner of the blue label is at (20, 20). All three labels are placed using the place manager.

7. Displaying Images

You can add an image to a label, button, check button, or radio button.

To create an image, use the PhotoImage class as follows:

```
photo = PhotoImage(file = imagefilename)
```

The image file must be in GIF format. You can use a conversion utility to convert image files in other formats into GIF format. Program below shows you how to add images to labels, buttons, check buttons, and radio buttons. You can also use the create_image method to display an image in a canvas, as shown in Figure 7.1.

Program :ImageDemo.py

```
from tkinter import * # Import all definitions from tkinter
 class ImageDemo:
  def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Image Demo") # Set title
# Create PhotoImage objects
 chinaImage = PhotoImage(file = "image/china.gif")
 leftImage = PhotoImage(file = "image/left.gif")
 rightImage = PhotoImage(file = "image/right.gif")
 usImage = PhotoImage(file = "image/usIcon.gif")
 ukImage = PhotoImage(file = "image/ukIcon.gif")
 crossImage = PhotoImage(file = "image/x.gif")
 circleImage = PhotoImage(file = "image/o.gif")

 # frame1 to contain label and canvas
        frame1 = Frame(window)
        frame1.pack()
        Label(frame1, image = caImage ).pack(side = LEFT)
        canvas = Canvas(frame1)
       canvas.create_image(90, 50, image = chinaImage)
       canvas["width"] = 200
        canvas["height"] = 100
        canvas.pack(side = LEFT)

 # frame2 contains buttons, check buttons, and radio buttons
        frame2 = Frame(window)
        frame2.pack()
       Button(frame2, image = leftImage).pack(side = LEFT)
       Button(frame2, image = rightImage).pack(side = LEFT)
```

```
Checkbutton(frame2, image = usImage).pack(side = LEFT)
Checkbutton(frame2, image = ukImage).pack(side = LEFT)
Radiobutton(frame2, image = crossImage).pack(side = LEFT)
Radiobutton(frame2, image = circleImage).pack(side = LEFT)

 window.mainloop() # Create an event loop

ImageDemo() # Create GUI
```

Fig 7.1 The program displays widgets with Images**.**

Explanation

The program places image files in the image folder in the current program directory, then creates PhotoImage objects for several images. These objects are used in widgets. The image is a property in Label, Button, Checkbutton, and RadioButton . Image is not a property for Canvas, but you can use the create_image method to display an image on the canvas . In fact, you can display multiple images in one canvas.

8. Menus

We can use Tkinter to create menus, popup menus, and toolbars. Tkinter provides a comprehensive solution for building graphical user interfaces. In this section we are going to see menus, popup menus, and toolbars. Menus make selection easier and are widely used in windows. We 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. Program below shows you how to create the menus shown in Figure 8.1.

Program :MenuDemo.py

```
from tkinter import *
```

```
class MenuDemo:
    def _ _init_ _(self):
        window = Tk()
        window.title("Menu Demo")
# Create a menu bar
    menubar = Menu(window)
    window.config(menu = menubar) # Display the menu bar
# Create a pull-down menu, and add it to the menu bar
    operationMenu = Menu(menubar, tearoff = 0)
    menubar.add_cascade(label="Operation",
                        menu = operationMenu)
    operationMenu.add_command(label = "Add",
    command = self.add)
     operationMenu.add_command(label = "Subtract",
     command = self.subtract)
     operationMenu.add_separator()
     operationMenu.add_command(label = "Multiply",
     command = self.multiply)
     operationMenu.add_command(label = "Divide",
     command = self.divide)
# Create more pull-down menus
    exitmenu = Menu(menubar, tearoff = 0)
    menubar.add_cascade(label = "Exit", menu = exitmenu)
    exitmenu.add_command(label = "Quit", command =
                                    window.quit)
# Add a tool bar frame
     frame0 = Frame(window) # Create and add a frame to window
      frame0.grid(row = 1, column = 1, sticky = W)
# Create images
     plusImage = PhotoImage(file = "image/plus.gif")
     minusImage = PhotoImage(file = "image/minus.gif")
     timesImage = PhotoImage(file = "image/times.gif")
     divideImage = PhotoImage(file = "image/divide.gif")

    Button(frame0, image = plusImage, command =
```

```
        self.add).grid(row = 1, column = 1, sticky = W)
        Button(frame0, image = minusImage,
        command = self.subtract).grid(row = 1, column = 2)
        Button(frame0, image = timesImage,
        command = self.multiply).grid(row = 1, column = 3)
         Button(frame0, image = divideImage,
        command = self.divide).grid(row = 1, column = 4)

    # Add labels and entries to frame1
         frame1 = Frame(window)
         frame1.grid(row = 2, column = 1, pady = 10)
         Label(frame1, text = "Number 1:").pack(side = LEFT)
         self.v1 = StringVar()
         Entry(frame1, width = 5, textvariable = self.v1,
         justify = RIGHT).pack(side = LEFT)
        Label(frame1, text = "Number 2:").pack(side = LEFT)
        self.v2 = StringVar()
        Entry(frame1, width = 5, textvariable = self.v2,
         justify = RIGHT).pack(side = LEFT)
         Label(frame1, text = "Result:").pack(side = LEFT)
         self.v3 = StringVar()
        Entry(frame1, width = 5, textvariable = self.v3,
         justify = RIGHT).pack(side = LEFT)

    # Add buttons to frame2
         frame2 = Frame(window) # Create and add a frame to window
        frame2.grid(row = 3, column = 1, pady = 10, sticky = E)
        Button(frame2, text = "Add", command = self.add).pack(
        side = LEFT)
        Button(frame2, text = "Subtract",
         command = self.subtract).pack(side = LEFT)
         Button(frame2, text = "Multiply",
         command = self.multiply).pack(side = LEFT)
         Button(frame2, text = "Divide",
         command = self.divide).pack(side = LEFT)
```

```
mainloop()

def add(self):
        self.v3.set(eval(self.v1.get()) + eval(self.v2.get()))

 def subtract(self):
        self.v3.set(eval(self.v1.get()) - eval(self.v2.get()))

 def multiply(self):
 self.v3.set(eval(self.v1.get()) * eval(self.v2.get()))

 def divide(self):
        self.v3.set(eval(self.v1.get()) / eval(self.v2.get()))
```

MenuDemo() # Create GUI

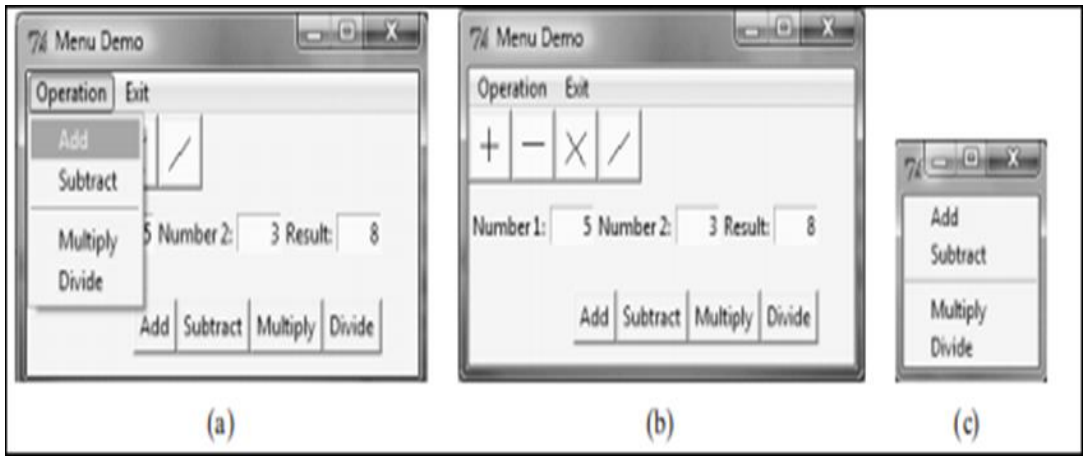


Fig 8.1 The program performs arithmetic operations using menu commands, toolbar buttons, and buttons.

Explanation

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, as shown in Figure 10c.

The program creates another menu named Exit and adds the Quit menu item to it. The program creates a frame named frame0 and uses it to hold toolbar buttons. The toolbar buttons are buttons with images, which are created by using the PhotoImage class.The command for each button specifies a callback function to be invoked when a toolbar button is clicked .The program creates a frame named frame1 and uses it to hold labels and entries for numbers. Variables v1, v2, and v3 bind the entries.The program creates a frame named frame2 and uses it to hold four buttons for performing Add, Subtract, Multiply, and Divide. The Add button, Add menu item, and Add tool bar button have the same callback function add, which is invoked when any one of them—the button, menu item, or tool bar button—is clicked.

9. Popup Menus

A popup menu, also known as a context menu, is like a regular menu, but it does not have a menu bar and it can float anywhere on the screen. Creating a popup menu is similar to creating a regular menu. First, you create an instance of Menu, and then you can add items to it. Finally, you bind a widget with an event to pop up the menu. The example below Show program uses popup menu commands to select a shape to be displayed in a canvas, as shown in Figure 9.1.

Program :PopupMenuDemo.py

```
from tkinter import * # Import all definitions from tkinter
class PopupMenuDemo:
    def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Popup Menu Demo") # Set title.
    # Create a popup menu
        self.menu = Menu(window, tearoff = 0)
        self.menu.add_command(label = "Draw a line",
        command = self.displayLine)
        self.menu.add_command(label = "Draw an oval",
        command = self.displayOval)
        self.menu.add_command(label = "Draw a rectangle",
```

```
            command = self.displayRect)
            self.menu.add_command(label = "Clear",
            command = self.clearCanvas)

        # Place canvas in window
            self.canvas = Canvas(window, width = 200,
            height = 100, bg = "white")
            self.canvas.pack()

        # Bind popup to canvas
            self.canvas.bind("<Button-3>", self.popup)
            window.mainloop() # Create an event loop

        # Display a rectangle
            def displayRect(self):
                self.canvas.create_rectangle(10, 10, 190, 90, tags =
                    "rect")

        # Display an oval
            def displayOval(self):
                self.canvas.create_oval(10, 10, 190, 90, tags =
                "oval")
        # Display two lines
            def displayLine(self):
                self.canvas.create_line(10, 10, 190, 90, tags = "line")
                self.canvas.create_line(10, 90, 190, 10, tags = "line")

        # Clear drawings
            def clearCanvas(self):
                    self.canvas.delete("rect", "oval", "line")
            def popup(self, event):
                    self.menu.post(event.x_root, event.y_root)

PopupMenuDemo() # Create GUI
```

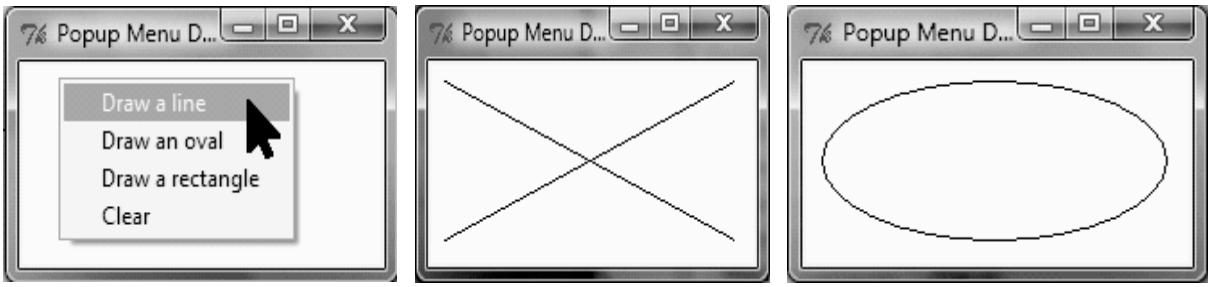


Fig 9.1 The program displays a popup menu when the canvas is clicked

Explanation

The program creates a menu to hold menu items. A canvas is created to display the shapes. The menu items use callback functions to instruct the canvas to draw shapes. Customarily, you display a popup menu by pointing to a widget and clicking the right mouse button. The program binds the right mouse button click with the popup callback function on the canvas . When you click the right mouse button, the popup callback function is invoked, which displays the menu at the location where the mouse is clicked.

10. Mouse , Key Events, and Bindings

You can use the bind method to bind mouse and key events to a widget.

The preceding example used the widget's bind method to bind a mouse event with a callback handler by using the syntax:

```
widget.bind(event, handler)
```

If a matching event occurs, the handler is invoked. In the preceding example, the event is <Button-3> and the handler function is popup. The event is a standard Tkinter object, which is automatically created when an event occurs. Every handler has an event as its argument. The following example defines the handler using the event as the argument:

```
def popup(event):
menu.post(event.x_root, event.y_root)
```

The event object has a number of properties describing the event pertaining to the event. For example, for a mouse event, the event object uses the x, y properties to capture the current mouse location in pixels.

Table below lists some commonly used events

TABLE Events

Event	Description
<Bi-Motion>	An event occurs when a mouse button is moved while being held down on the widget.
<Button-i>	Button-1, Button-2, and Button-3 identify the left, middle, and right buttons. When a mouse button is pressed over the widget, Tkinter automatically grabs the mouse pointer's location. ButtonPressed-*i* is synonymous with Button-*i*.
<ButtonReleased-i>	An event occurs when a mouse button is released.
<Double-Button-i>	An event occurs when a mouse button is double-clicked.
<Enter>	An event occurs when a mouse pointer enters the widget.
<Key>	An event occurs when a key is pressed.
<Leave>	An event occurs when a mouse pointer leaves the widget.
<Return>	An event occurs when the *Enter* key is pressed. You can bind any key such as *A*, *B*, *Up*, *Down*, *Left*, *Right* in the keyboard with an event.
<Shift+A>	An event occurs when the *Shift+A* keys are pressed. You can combine *Alt*, *Shift*, and *Control* with other keys.
<Triple-Button-i>	An event occurs when a mouse button is triple-clicked.

Table below lists some event properties.

Table :Event Properties

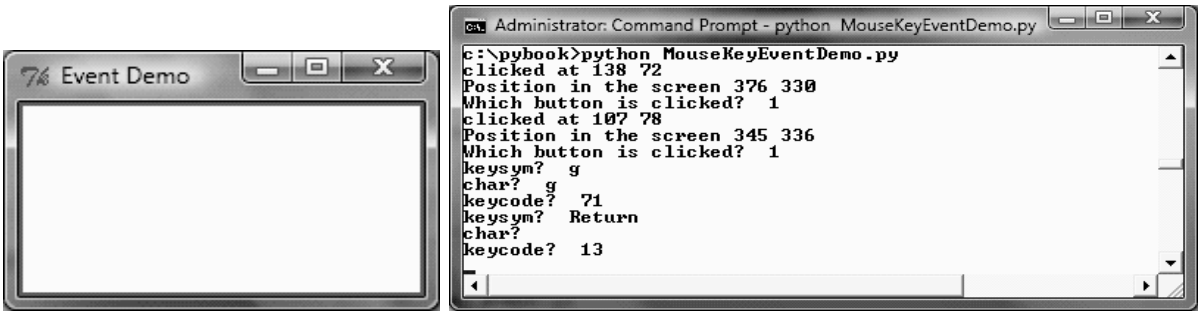
Event Property	Description
char	The character entered from the keyboard for key events.
keycode	The key code (i.e., Unicode) for the key entered from the keyboard for key events.
keysym	The key symbol (i.e., character) for the key entered from the keyboard for key events.
num	The button number (1, 2, 3) indicates which mouse button was clicked.
widget	The widget object that fires this event.
x and **y**	The current mouse location in the widget in pixels.
x_ _root and **y_root**	The current mouse position relative to the upper-left corner of the screen, in pixels

The program below processes mouse and key events. It displays the window as shown in Figure 10.1a. The mouse and key events are processed and the processing information is displayed in the command window, as shown in Figure10.1b.

Program :MouseKeyEventDemo.py

```
from tkinter import * # Import all definitions from tkinter
class MouseKeyEventDemo:
    def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Event Demo") # Set a title
        canvas = Canvas(window, bg = "white", width = 200, height =
                        100)
        canvas.pack()
# Bind with <Button-1> event
        canvas.bind("<Button-1>", self.processMouseEvent)
```

```
# Bind with <Key> event
        canvas.bind("<Key>", self.processKeyEvent)
        canvas.focus_set()

    window.mainloop() # Create an event loop

def processMouseEvent(self, event):
    print("clicked at", event.x, event.y)
     print("Position in the screen", event.x_root, event.y_root)
     print("Which button is clicked? ", event.num)

def processKeyEvent(self, event):
     print("keysym? ", event.keysym)
     print("char? ", event.char)
     print("keycode? ", event.keycode)

 MouseKeyEventDemo() # Create GUI
```



(a) **(b)**

Fig. 10.1 The program processes mouse and key events

The program creates a canvas and binds a mouse event <Button-1> with the callback function processMouseEvent on the canvas. Nothing is drawn on the canvas. So it is blank as shown in Figure 12a. When the left mouse utton is clicked on the canvas, an event is created. The processMouseEvent is invoked to process an event that displays the mouse pointer's location on the canvas, on the screen , and which mouse button is clicked. The Canvas widget is also the source for the key event. The program binds a key event

with the callback function processKeyEvent on the canvas (line 14) and sets the focus on the canvas so that the canvas will receive input from the keyboard .

Program below displays a circle on the canvas. The circle radius is increased with a left mouse click and decreased with a right mouse click, as shown in Figure 10.2.

Program :EnlargeShrinkCircle.py

```
from tkinter import * # Import all definitions from tkinter
class EnlargeShrinkCircle:


    def _ _init_ _(self):
        self.radius = 50

        window = Tk() # Create a window
        window.title("Control Circle Demo") # Set a title
        self.canvas = Canvas(window, bg = "white",
                             width = 200, height = 200)
        self.canvas.pack()
        self.canvas.create_oval(
        100 - self.radius, 100 - self.radius,
        100 + self.radius, 100 + self.radius, tags = "oval")

 # Bind canvas with mouse events
    self.canvas.bind("<Button-1>", self.increaseCircle)
    self.canvas.bind("<Button-3>", self.decreaseCircle)

     window.mainloop() # Create an event loop

def increaseCircle(self, event):
    self.canvas.delete("oval")
     if self.radius < 100:
        self.radius += 2
     self.canvas.create_oval(
```

```
                    100 - self.radius, 100 - self.radius,
                    100 + self.radius, 100 + self.radius, tags = "oval")
    def decreaseCircle(self, event):
        self.canvas.delete("oval")
        if self.radius > 2:
            self.radius -= 2
        self.canvas.create_oval(
            100 - self.radius, 100 - self.radius,
            100 + self.radius, 100 + self.radius, tags = "oval")

    EnlargeShrinkCircle() # Create GUI
```

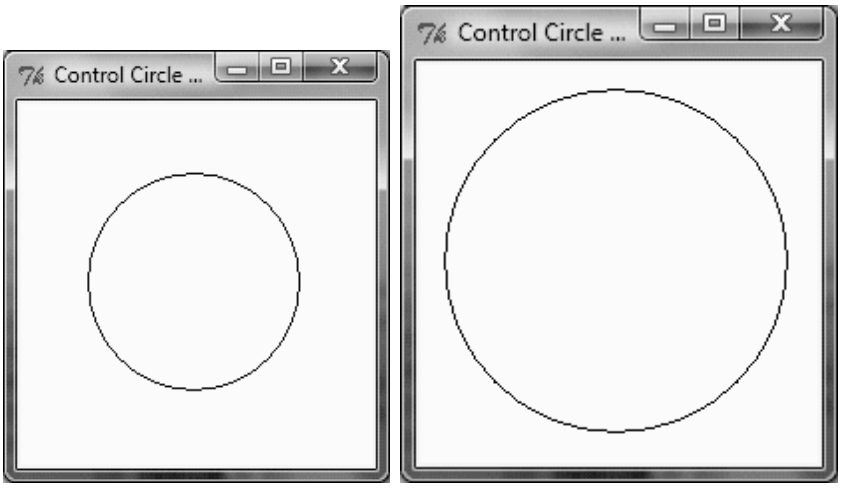


Fig. 10.2 The program uses mouse events to control the circle's size

The program creates a canvas and displays a circle on the canvas with an initial radius of 50. The canvas is bound to a mouse event <Button-1> with the handler increaseCircle and to a mouse event <Button-3> with the handler decreaseCircle. When the left mouse button is pressed, the increaseCircle function is invoked to increase the radius and redisplay the circle.When the right mouse button is pressed, the decreaseCircle function is invoked to decrease the radius and redisplay the circle.

11. List boxes

The Listbox widget is used to display the list items to the user We can place only text items in the Listbox and all text items contain the same font and

color. The user can choose one or more items from the list depending upon the configuration. The syntax to use the Listbox is given below**.**

w = Listbox(parent, options)

A list of possible options is given below. After that program explain how list box work and output is shown figure 11.1.

SN	Option	Description
1	bg	The background color of the widget.
2	bd	It represents the size of the border. Default value is 2 pixel.
3	cursor	The mouse pointer will look like the cursor type like dot, arrow, etc.
4	font	The font type of the Listbox items.
5	fg	The color of the text.
6	height	It represents the count of the lines shown in the Listbox. The default value is 10.
7	highlightcolor	The color of the Listbox items when the widget is under focus.
8	highlightthickness	The thickness of the highlight.
9	relief	The type of the border. The default is SUNKEN.
10	selectbackground	The background color that is used to display the selected text.
11	selectmode	It is used to determine the number of items that can be selected from the list. It can set to BROWSE, SINGLE, MULTIPLE, EXTENDED.
12	width	It represents the width of the widget in characters.
13	xscrollcommand	It is used to let the user scroll the Listbox horizontally.
14	yscrollcommand	It is used to let the user scroll the Listbox vertically.

```
from tkinter import *
top = Tk()
top.geometry("200x250")
lbl = Label(top,text = "A list of favourite countries...")
listbox = Listbox(top)
listbox.insert(1,"India")
listbox.insert(2, "USA")
listbox.insert(3, "Japan")
listbox.insert(4, "Austrelia")
lbl.pack()
listbox.pack()
top.mainloop()
```

Fig. 11.1 list box

12. Animations

Animations can be created by displaying a sequence of drawings. The Canvas class can be used to develop animations. You can display graphics and text on the canvas and use the move(tags, dx, dy) method to move the graphic with the specified tags dx pixels to the right, if dx is positive and dy pixels down if dy is positive. If dx or dy is negative, the graphic is moved

left or up. The program belowdisplays a moving message repeatedly from left to right, as shown in Figure 12.1

```
Program :AnimationDemo.py
from tkinter import * # Import all definitions from tkinter
class AnimationDemo:
        def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Animation Demo") # Set a title

        width = 250 # Width of the canvas
        canvas = Canvas(window, bg = "white",
            width = 250, height = 50)
        canvas.pack()

x = 0 # Starting x position
canvas.create_text(x, 30,
        text = "Message moving?", tags = "text")
dx = 3
while True:
        canvas.move("text", dx, 0)  # Move text dx unit
        canvas.after(100)           # Sleep for 100 milliseconds
        canvas.update()             # Update canvas
if x < width:
        x += dx # Get the current position for string
else:
        x = 0 # Reset string position to the beginning
        canvas.delete("text")
# Redraw text at the beginning
canvas.create_text(x, 30, text = "Message moving?",
                tags = "text")
window.mainloop() # Create an event loop

AnimationDemo() # Create GUI
```

Fig.12.1 The program animates a moving message

The program creates a canvas and displays text on the canvas at the specified initial location . The animation is done essentially in the following three statements in a loop :

```
canvas.move("text", dx, 0) # Move text dx unit
canvas.after(100) # Sleep for 100 milliseconds
canvas.update() # Update canvas
```

The x-coordinate of the location is moved to the right dx units by invoking canvas.move . Invoking canvas.after(100) puts the program to sleep for 100 milliseconds . Invoking canvas.update() redisplays the canvas .

We can add tools to control the animation's speed, stop the animation, and resume the animation. by adding four buttons to control the animation, as shown in Figure 12.2.

Program :ControlAnimation.py

```
from tkinter import * # Import all definitions from tkinter
class ControlAnimation:
    def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Control Animation Demo") # Set a title

        self.width = 250 # Width of self.canvas
        self.canvas = Canvas(window, bg = "white",
                width = self.width, height = 50)
        self.canvas.pack()

        frame = Frame(window)
        frame.pack()
        btStop = Button(frame, text = "Stop", command = self.stop)
        btStop.pack(side = LEFT)
        btResume = Button(frame, text = "Resume",
```

```
                command = self.resume)
            btResume.pack(side = LEFT)
            btFaster = Button(frame, text = "Faster",
                    command = self.faster)
            btFaster.pack(side = LEFT)
            btSlower = Button(frame, text = "Slower",
                    command = self.slower)
            btSlower.pack(side = LEFT)

            self.x = 0  # Starting x position
            self.sleepTime = 100 # Set a sleep time
            self.canvas.create_text(self.x, 30,
                    text = "Message moving?", tags = "text")

            self.dx = 3
            self.isStopped = False
            self.animate()

 window.mainloop() # Create an event loop
def stop(self): # Stop animation
        self.isStopped = True

def resume(self):# Resume animation
        self.isStopped = False
        self.animate()

def faster(self): # Speed up the animation
        if self.sleepTime > 5:
                self.sleepTime -= 20

def slower(self): # Slow down the animation
        self.sleepTime += 20

def animate(self): # Move the message
        while not self.isStopped:
```

```
            self.canvas.move("text", self.dx, 0) # Move text
            self.canvas.after(self.sleepTime) # Sleep
            self.canvas.update() # Update canvas
            if self.x < self.width:
                self.x += self.dx # Set new position
            else:
                self.x = 0 # Reset string position to beginning
                self.canvas.delete("text")
                # Redraw text at the beginning
                self.canvas.create_text(self.x, 30,
                    text = "Message moving?", tags = "text")

ControlAnimation() # Create GUI
```

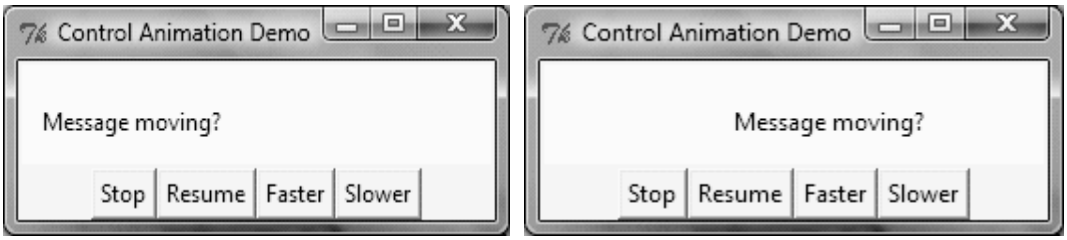


Fig. 12.2 The program uses buttons to control the animation

The program starts the animation by invoking animate(). The isStopped variable determines whether the animation continues to move. It is set to False initially. When it is false, the loop in the animate method executes continuously. Clicking the buttons Stop, Resume, Faster, or Slower stops, resumes, speeds up, or slows down the animation. When the Stop button is clicked, the stop function is invoked to set isStopped to True. This causes the animation loop to terminate (line 53). When the Resume button is clicked, the resume function is invoked to set isStopped to False and resume animation . The speed of the animation is controlled by the variable sleepTime, which is set to 100 milliseconds initially. When the Faster button is clicked, the faster method is invoked to reduce sleepTime by 20. When the Slower button is clicked, the slower function is invoked to increase sleepTime by 20.

13.Scrollbars

A Scrollbar widget can be used to scroll the contents in a Text, Canvas, or Listbox widget vertically or horizontally. Program below gives an example of scrolling in a Text widget, as shown in Figure 13.1.

Program :ScrollText.py

```
from tkinter import * # Import all definitions from tkinter
 class ScrollText:
        def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Scroll Text Demo") # Set title

        frame1 = Frame(window)
        frame1.pack()
        scrollbar = Scrollbar(frame1)
        scrollbar.pack(side = RIGHT, fill = Y)
        text = Text(frame1, width = 40, height = 10, wrap = WORD,
                yscrollcommand = scrollbar.set)
        text.pack()
       scrollbar.config(command = text.yview)

window.mainloop() # Create an event loop

ScrollText() # Create GUI
```

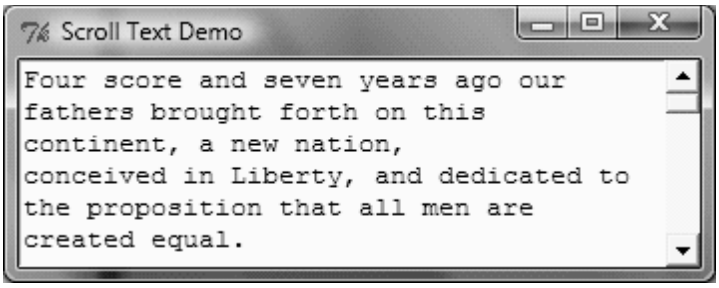


Fig 13.1 You can use the scrollbar (on the far right) to scroll to see text not currently visible in the Text widget.

The program creates a Scrollbar (line 10) and places it to the right of the text .The scrollbar is tied to the Text widget so that the contents in the Text widget can be scrolled through.

14. Standard Dialog Boxes

You can use standard dialog boxes to display message boxes or to prompt the user to enter numbers and strings. Finally, let's look at Tkinter's standard dialog boxes (often referred to simply as dialogs). Program below gives an example of using these dialogs. A sample run of the program is shown in figure 14.1.

Program :DialogDemo.py

```
import tkinter.messagebox
import tkinter.simpledialog
import tkinter.colorchooser
tkinter.messagebox.showinfo("showinfo", "This is an info msg")

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)
```

The program invokes the showinfo, showwarning, and 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.

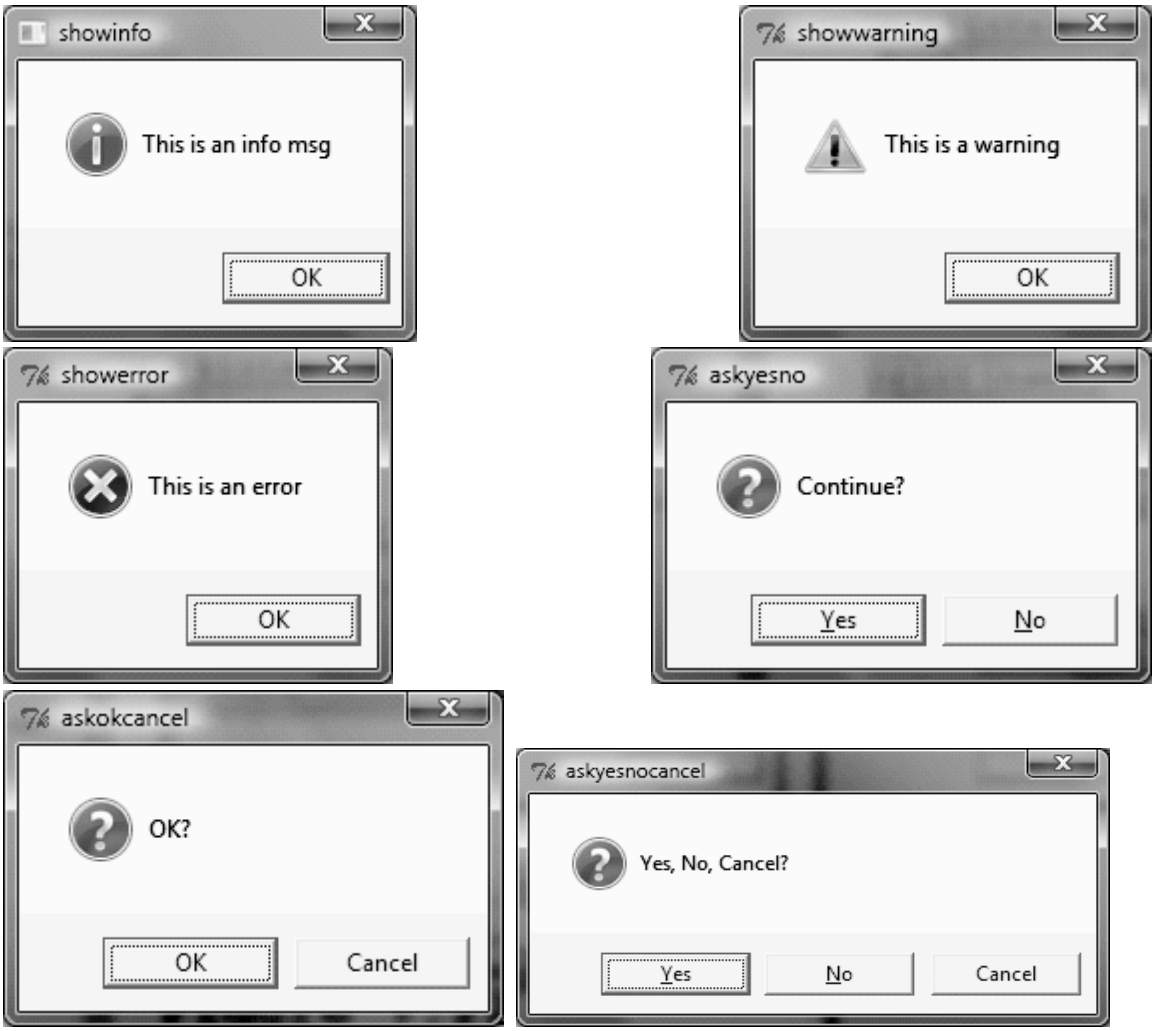


Fig 14.1 You can use the standard dialogs to display message boxes and accept input.

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 Cancel buttons 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 askfloat function 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.

15. Grids

This geometry manager organizes widgets in a table-like structure in the parent widget**.** The master widget is split into rows and columns, and each part of the table can hold a widget. It uses column, columnspan, ipadx, ipady, padx, pady, row, rowspan and sticky.

Syntax

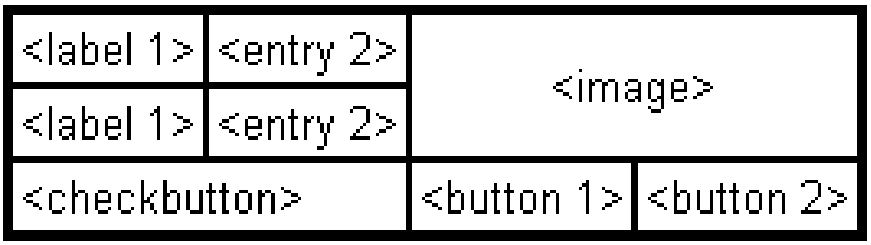
widget.grid(grid_options)

Here is the list of possible options −

- **column** − The column to put widget in; default 0 (leftmost column).
- **columnspan** − How many columns widgetoccupies; default 1.
- **ipadx, ipady** − How many pixels to pad widget, horizontally and vertically, inside widget's borders.
- **padx, pady** − How many pixels to pad widget, horizontally and vertically, outside v's borders.
- **row** − The row to put widget in; default the first row that is still empty.
- **rowspan** − How many rowswidget occupies; default 1.
- **sticky** − What to do if the cell is larger than widget. By default, with sticky='', widget is centered in its cell. sticky may be the string concatenation of zero or more of N, E, S, W, NE, NW, SE, and SW, compass directions indicating the sides and corners of the cell to which widget sticks.

The grid manager is the most flexible of the geometry managers in Tkinter. If you don’t want to learn how and when to use all three managers, you should at least make sure to learn this one.

Consider the following example –

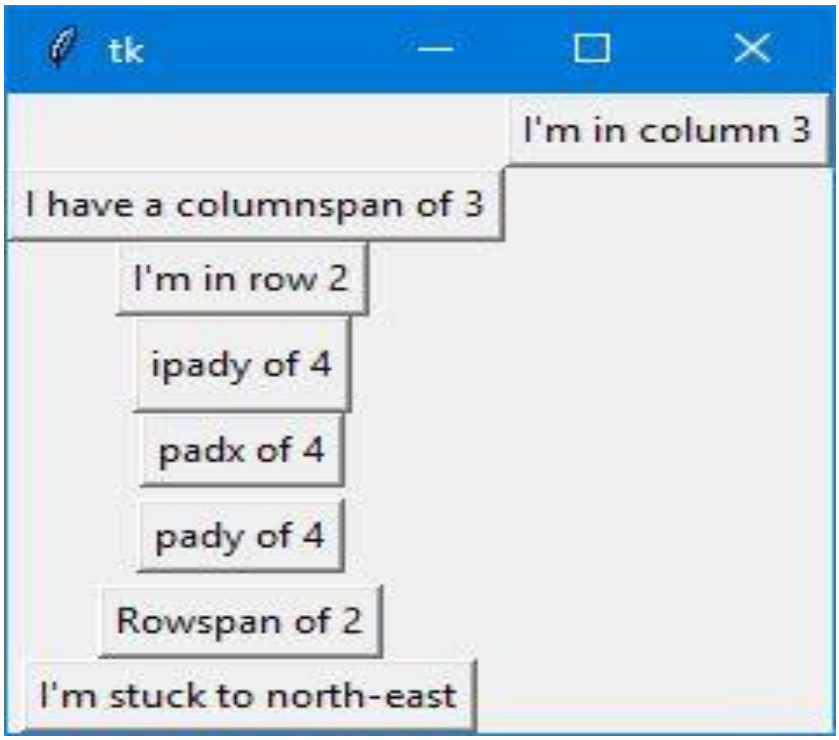


Creating this layout using the pack manager is possible, but it takes a number of extra frame widgets, and a lot of work to make things look good. If you use the grid manager instead, you only need one call per widget to get everything laid out properly. But using the grid manager is easy. Just create the widgets, and use the grid method to tell the manager in which row and column to place them. You don't have to specify the size of the grid beforehand; the manager automatically determines that from the widgets in it.

Program: Grids.py

```
from tkinter import *
root = Tk()
btn_column = Button(root, text="I'm in column 3")
btn_column.grid(column=3)

btn_columnspan = Button(root, text="I have a columnspan of 3")
btn_columnspan.grid(columnspan=3)

btn_ipadx = Button(root, text="ipadx of 4")
btn_ipadx.grid(ipadx=4)

btn_ipady = Button(root, text="ipady of 4")
btn_ipady.grid(ipady=4)

btn_padx = Button(root, text="padx of 4")
btn_padx.grid(padx=4)
```

```
btn_pady = Button(root, text="pady of 4")
btn_pady.grid(pady=4)

btn_row = Button(root, text="I'm in row 2")
btn_row.grid(row=2)

btn_rowspan = Button(root, text="Rowspan of 2")
btn_rowspan.grid(rowspan=2)

btn_sticky = Button(root, text="I'm stuck to north-east")
btn_sticky.grid(sticky=NE)

root.mainloop()
```



Program :Grid1.py

```
# import tkinter module
from tkinter import * from tkinter.ttk import *

# creating main tkinter window/toplevel
```

```
master = Tk()

# this wil create a label widget
l1 = Label(master, text = "First:")
l2 = Label(master, text = "Second:")

# grid method to arrange labels in respective
# rows and columns as specified
l1.grid(row = 0, column = 0, sticky = W, pady = 2)
l2.grid(row = 1, column = 0, sticky = W, pady = 2)

# entry widgets, used to take entry from user
e1 = Entry(master)
e2 = Entry(master)

# this will arrange entry widgets
e1.grid(row = 0, column = 1, pady = 2)
e2.grid(row = 1, column = 1, pady = 2)

# infinite loop which can be terminated by keyboard
# or mouse interrupt
mainloop()
```