Contents

Files Handling in Python

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).

Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (perform operation)
3. Close the file

Opening Files in Python

Python has a built-in `open()` function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python38/README.txt")  # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read `r`, write `w` or append `a` to the file. We can also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

Mode	**Description**
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.

t	Opens in text mode. (default)
b	Opens in binary mode.
+	Opens a file for updating (reading and writing)

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode
```

Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file. It is done using the close() method available in Python.

Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file

The best way to close a file is by using the with statement. This ensures that the file is closed when the block inside the with statement is exited.

We don't need to explicitly call the close() method. It is done internally.

```
with open("test.txt") as f:
   # perform file operations
```

Writing to Files in Python

In order to write into a file in Python, we need to open it in write w, append a or exclusive creation x mode.

We need to be careful with the w mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using the write() method. This method returns the number of characters written to the file.

```
with open("test.txt",'w') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

This program will create a new file named test.txt in the current directory if it does not exist. If it does exist, it is overwritten.

We must include the newline characters ourselves to distinguish the different lines.

Reading Files in Python

To read a file in Python, we must open the file in reading r mode.

There are various methods available for this purpose. We can use the read(size) method to read in the *size* number of data. If the *size* parameter is not specified, it reads and returns up to the end of the file.

We can read the text.txt file we wrote in the above section in the following way:

```
>>> f = open("test.txt",'r')
>>> f.read(4)    # read the first 4 data
'This'

>>> f.read(4)    # read the next 4 data
' is '

>>> f.read()     # read in the rest till end of file
'my first file\nThis file\ncontains three lines\n'

>>> f.read()  # further reading returns empty sting
''
```

FILE POSITIONING

We can see that the read() method returns a newline as '\n'. Once the end of the file is reached, we get an empty string on further reading.

We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).

```
>>> f.tell()    # get the current file position
56

>>> f.seek(0)   # bring file cursor to initial position
0

>>> print(f.read())  # read the entire file
This is my first file
This file
contains three lines
```

We can read a file line-by-line using a for loop. This is both efficient and fast.

```
>>> for line in f:
...     print(line, end = '')
...
This is my first file
This file
contains three lines
```

In this program, the lines in the file itself include a newline character `\n`. So, we use the end parameter of the `print()` function to avoid two newlines when printing.

Alternatively, we can use the `readline()` method to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
>>> f.readline()
'This is my first file\n'

>>> f.readline()
'This file\n'

>>> f.readline()
'contains three lines\n'

>>> f.readline()
''
```

Lastly, the `readlines()` method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

Here is the complete list of methods in text mode with a brief description:

Method	**Description**
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the TextIOBase and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns True if the file stream is interactive.

read(*n*)	Reads at most *n* characters from the file. Reads till end of file if it is negative or None.
readable()	Returns True if the file stream can be read from.
readline(*n*=-1)	Reads and returns one line from the file. Reads in at most *n* bytes if specified.
readlines(*n*=-1)	Reads and returns a list of lines from the file. Reads in at most *n* bytes/characters if specified.
seek(*offset*,*from*=SEEK_SET)	Changes the file position to *offset* bytes, in reference to *from* (start, current, end).
seekable()	Returns True if the file stream supports random access.
tell()	Returns the current file location.
truncate(*size*=None)	Resizes the file stream to *size* bytes. If *size* is not specified, resizes to current location.
writable()	Returns True if the file stream can be written to.
write(*s*)	Writes the string *s* to the file and returns the number of characters written.
writelines(*lines*)	Writes a list of *lines* to the file.

Python Directory and Files Management

If there are a large number of files to handle in our Python program, we can arrange our code within different directories to make things more manageable.

A directory or folder is a collection of files and subdirectories. Python has the os module that provides us with many useful methods to work with directories (and files as well).

Get Current Directory

We can get the present working directory using the getcwd() method of the os module.

This method returns the current working directory in the form of a string. We can also use the getcwdb() method to get it as bytes object.

```
>>> import os

>>> os.getcwd()
'C:\\Program Files\\PyScripter'

>>> os.getcwdb()
b'C:\\Program Files\\PyScripter'
```

The extra backslash implies an escape sequence. The print() function will render this properly.

```
>>> print(os.getcwd())
C:\Program Files\PyScripter
```

Changing Directory

We can change the current working directory by using the chdir() method.

The new path that we want to change into must be supplied as a string to this method. We can use both the forward-slash / or the backward-slash \ to separate the path elements.

It is safer to use an escape sequence when using the backward slash.

```
>>> os.chdir('C:\\Python33')

>>> print(os.getcwd())
C:\Python33
```

List Directories and Files

All files and sub-directories inside a directory can be retrieved using the listdir() method.

This method takes in a path and returns a list of subdirectories and files in that path. If no path is specified, it returns the list of subdirectories and files from the current working directory.

```
>>> print(os.getcwd())
```

Graphical User Interface Programming USING Tkinter

Tkinter is the inbuilt python module that is used to create GUI applications. It is one of the most commonly used modules for creating GUI applications in Python as it is simple and easy to work with. You don't need to worry about the installation of the Tkinter module separately as it comes with Python already. It gives an object-oriented interface to the Tk GUI toolkit.

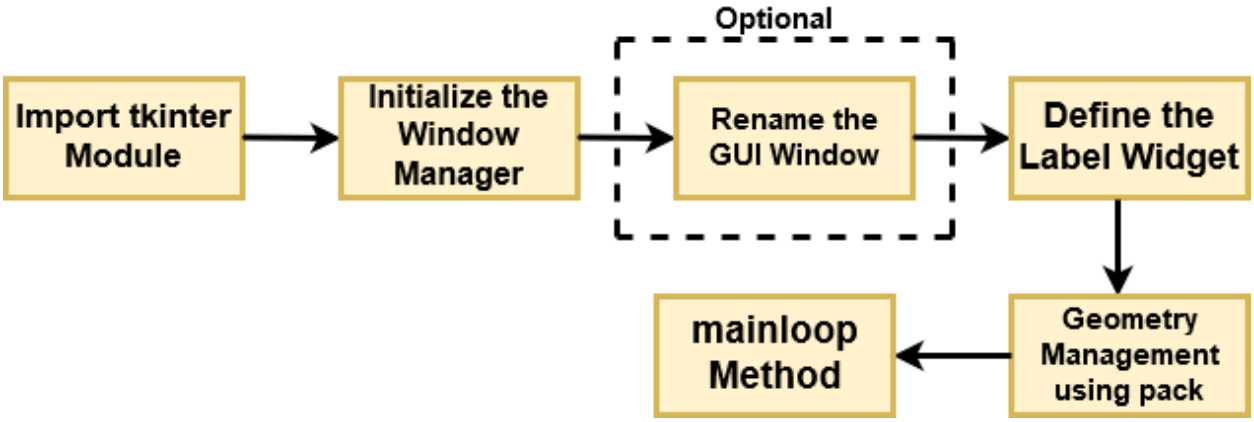
Some other Python Libraries available for creating our own GUI applications are

- Kivy
- Python Qt
- wxPython

What are Widgets?

Widgets in Tkinter are the elements of GUI application which provides various controls (such as Labels, Buttons, ComboBoxes, CheckBoxes, MenuBars, RadioButtons and many more) to users to interact with the application.

Fundamental structure of tkinter program



Flow Diagram for Rendering a Basic GUI

Basic Tkinter Widgets:

Widgets	Description
Label	It is used to display text or image on the screen
Button	It is used to add buttons to your application

Canvas	It is used to draw pictures and others layouts like texts, graphics etc.
ComboBox	It contains a down arrow to select from list of available options
CheckButton	It displays a number of options to the user as toggle buttons from which user can select any number of options.
RadiButton	It is used to implement one-of-many selection as it allows only one option to be selected
Entry	It is used to input single line text entry from user
Frame	It is used as container to hold and organize the widgets
Message	It works same as that of label and refers to multi-line and non-editable text
Scale	It is used to provide a graphical slider which allows to select any value from that scale
Scrollbar	It is used to scroll down the contents. It provides a slide controller.
SpinBox	It is allows user to select from given set of values
Text	It allows user to edit multiline text and format the way it has to be displayed
Menu	It is used to create all kinds of menu used by an application

FIRST GUI PROGRAM

```
from tkinter import *
from tkinter.ttk import *

# writing code needs to create the main window of  the application creating
# main window object named root
root = Tk()

# giving title to the main window
root.title("First GUI Program")
# Label is what output will be shown  on the window
label = Label(root, text ="Hello World !").pack()

# calling mainloop method which is used when your application is ready to run
# and it tells the code to keep displaying
root.mainloop()
```

CREATING WIDGETS

Each separate widget is a Python object. When creating a widget, you must pass its parent as a parameter to the widget creation function. The only exception is the “root” window, which is the top-level window that will contain everything else and it does not have a parent.

Geometry Management

Creating a new widget doesn't mean that it will appear on the screen. To display it, we need to call a special method: either **grid**, **pack**(example above), or **place**.

Method	Description
pack()	The **Pack** geometry manager packs widgets in rows or columns.
grid()	The **Grid** geometry manager puts the widgets in a 2-dimensional table. The master widget is split into a number of rows and columns, and each "cell" in the resulting table can hold a widget.
place()	The **Place** geometry manager is the simplest of the three general geometry managers provided in Tkinter. It allows you explicitly set the position and size of a window, either in absolute terms, or relative to another window.

place() method in Tkinter

The **Place** geometry manager is the simplest of the three general geometry managers provided in Tkinter. It allows you explicitly set the position and size of a window, either in absolute terms, or relative to another window. You can access the **place** manager through the **place()** method which is available for all standard widgets.

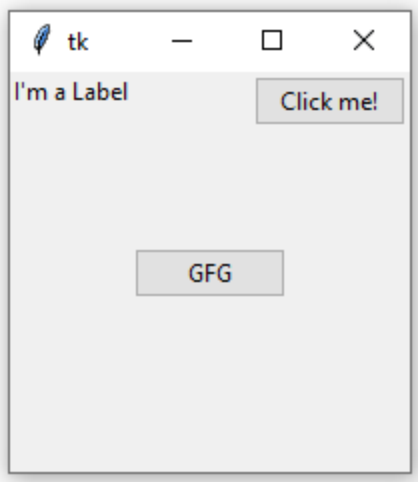
It is usually not a good idea to use **place()** for ordinary window and dialog layouts; its simply to much work to get things working as they should. Use the **pack()** or **grid()** managers for such purposes.

widget.place(relx = 0.5, rely = 0.5, anchor = CENTER)
Note : place() method can be used with **grid()** method as well as with **pack()** method.

```
from tkinter import * from tkinter.ttk import *

# creating Tk window
master = Tk()

# setting geometry of tk window
master.geometry("200x200")

# button widget
b1 = Button(master, text = "Click me !")
b1.place(relx = 1, x =-2, y = 2, anchor = NE)

# label widget
l = Label(master, text = "I'm a Label")
l.place(anchor = NW)

# button widget
```

```
b2 = Button(master, text = "GFG")
b2.place(relx = 0.5, rely = 0.5, anchor = CENTER)

# infinite loop which is required to
# run tkinter program infinitely
# until an interrupt occurs
mainloop()
```

Output:



When we use **pack()** or **grid()** managers, then it is very easy to put two different widgets separate to each other but putting one of them inside other is a bit difficult. But this can easily be achieved by **place()** method

pack() method in Tkinter

The Pack geometry manager packs widgets in rows or columns. We can use options like **fill**, **expand**, and **side** to control this geometry manager.

Compared to the **grid** manager, the **pack** manager is somewhat limited, but it's much easier to use in a few, but quite common situations:

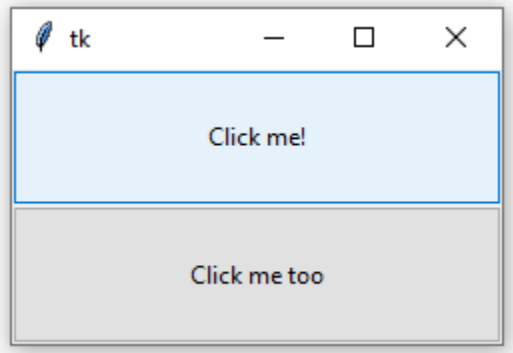
- Put a widget inside a frame (or any other container widget), and have it fill the entire frame
- Place a number of widgets on top of each other
- Place a number of widgets side by side

Code #1: Putting a widget inside frame and filling entire frame. We can do this with the help of **expand** and **fill** options.

```
from tkinter import * from tkinter.ttk import *

# creating Tk window
```

```
master = Tk()

# cretaing a Fra, e which can expand according
# to the size of the window
pane = Frame(master)
pane.pack(fill = BOTH, expand = True)

# button widgets which can also expand and fill
# in the parent widget entirely
b1 = Button(pane, text = "Click me !")
b1.pack(fill = BOTH, expand = True)

b2 = Button(pane, text = "Click me too")
b2.pack(fill = BOTH, expand = True)

mainloop()
```

Output:



- **Button**: Button widget has a property for switching on/off. When a user clicks the button, an event is triggered in the Tkinter.

  Syntax: **button_widget = tk.Button(widget, option=placeholder)** where widget is the argument for the parent window/frame while option is a placeholder that can have various values like foreground & background color, font, command (for function call), image, height, and width of button.

- **Canvas**: Canvas is used to draw shapes in your GUI and supports various drawing methods.

  Syntax: **canvas_widget = tk.Canvas(widget, option=placeholder)** where widget is the parameter for the parent window/frame while option is a placeholder that can have various values like border-width, background color, height and width of widget.

- **Checkbutton**: Checkbutton records on-off or true-false state. It lets you can select more than one option at a time and even leave it unchecked.

  Syntax: **checkbutton_widget = tk.CheckButton(widget, option=placeholder)** where widget is the parameter for the parent window/frame while option is a placeholder that can have various values like title, text, background & foreground color while widget is under the cursor, font, image, etc.

- **Entry**: Entry widget is used to create input fields or to get input text from the user within the GUI.

  Syntax: **entry_widget = tk.Entry(widget, option=placeholder)** where widget is the parameter for the parent window/frame while option is a placeholder that can have various values like border-width, background color, width & height of button etc.

- **Frame**: Frame is used as containers in the Tkinter for grouping and adequately organizing the widgets.

  Syntax: **frame_widget = tk.Frame(widget, option=placeholder)** where widget is the parameter for the parent window/frame while option is a placeholder that can have various values like border-width, height & width of widget, highlightcolor (color when widget has to be focused).

- **Label**: Label is used to create a single line widgets like text, images, etc.

  Syntax: **label_widget = tk.Label(widget, option=placeholder)** where widget is the parameter for the parent window/frame while option is a placeholder that can have various values like the font of a button, background color, image, width, and height of button.

Python Tkinter Button

The button widget is used to add various types of buttons to the python application. Python allows us to configure the look of the button according to our requirements. Various options can be set or reset depending upon the requirements.

We can also associate a method or function with a button which is called when the button is pressed.

The syntax to use the button widget is given below.

```
from tkinter import *
```

```
top = Tk()
top.geometry("200x100")
b = Button(top,text = "Simple")
b.pack()
top.mainaloop()
```

OR

```
b1 = Button(top,text = "Red",command = fun,activeforeground = "red",activebackground = "pink
",pady=10)
b2 = Button(top, text = "Blue",activeforeground = "blue",activebackground = "pink",pady=10)
b3 = Button(top, text = "Green",activeforeground = "green",activebackground = "pink",pady = 10)
b4 = Button(top, text = "Yellow",activeforeground = "yellow",activebackground = "pink",pady = 1
0)
b1.pack(side = LEFT)
b2.pack(side = RIGHT)
b3.pack(side = TOP)
b4.pack(side = BOTTOM)
```

EVENT HANDLING (Binding or Command Functions)

Binding or Command functions are those who are called whenever an event occurs or is triggered.

Let's take an example to understand binding functions.

You will define a button which, when clicked, calls a function called showresult. Further, the function showresult will create a new label with the text GUI with Tkinter!.

```
import tkinter
window = tkinter.Tk()
window.title("GUI")

def showresult():
    tkinter.Label(window, text = "GUI with Tkinter!").pack()

tkinter.Button(window, text = "Click Me!", command = showresult).pack()
window.mainloop()
```

Apart from invoking binding functions with a mouse click, the events can be invoked with a mouse-move, mouse-over, clicking, scrolling, etc.

Let's now look at the bind function, which provides you the same functionality as above.

Mouse Clicking Event via the Bind Method

The Bind method provides you with a very simplistic approach to implementing the mouse clicking events. Let's look at the three pre-defined functions which can be used out of the box with the bind method.

Clicking events are of three types leftClick, middleClick, and rightClick.

<Button-1> parameter of the bind method is the left-clicking event, i.e., when you click the left button, the bind method will call the function specified as a second parameter to it.

<Button-2> for middle-click

<Button-3> for right-click

```
import tkinter
window = tkinter.Tk()
window.title("GUI")

def left_click(event):
    tkinter.Label(window, text = "Left Click!").pack()

def middle_click(event):
    tkinter.Label(window, text = "Middle Click!").pack()

def right_click(event):
    tkinter.Label(window, text = "Right Click!").pack()

window.bind("<Button-1>", left_click)
window.bind("<Button-2>", middle_click)
window.bind("<Button-3>", right_click)

window.mainloop()
```

Alert Boxes

You can create alert boxes in the Tkinter using the messagebox method. You can also create questions using the messasgebox method.

Here you will create a simple alert-box and also create a question. For generating an alert, you will use messagebox function showinfo. For creating a question, you will use the askquestion method, and based on the response to the question, you will output a Label on the GUI.

```
import tkinter
import tkinter.messagebox
```

```
window = tkinter.Tk()
window.title("GUI")

tkinter.messagebox.showinfo("Alert Message", "This is just a alert message!")

response = tkinter.messagebox.askquestion("Tricky Question", "Do you love Deep Learning?")

# A basic 'if/else' block where if user clicks on 'Yes' then it returns 1 # else it returns 0. For each response you will display a message with the #help of 'Label' method.
if response == 1:
    tkinter.Label(window, text = "Yes, of course I love Deep Learning!").pack()
else:
    tkinter.Label(window, text = "No, I don't love Deep Learning!").pack()

window.mainloop()
```

Rendering Images

If you have been able to follow along until here, then, adding images and icons to the GUI should be a piece of cake. All you need to do is use the PhotoImage method of Tkinter and pass the file_path as the parameter to it.

In order to display the image in a GUI, you will use the 'PhotoImage' #method of Tkinter. It will an image from the directory (specified path) and # store the image in a variable. Finally, to display the image you will make use of the 'Label' method and pass the 'image' variable as a parameter and use the pack() method to display inside the GUI.

```
import tkinter

window = tkinter.Tk()
window.title("GUI")

icon = tkinter.PhotoImage(file = "CNN.png")

label = tkinter.Label(window, image = icon)
label.pack()

window.mainloop()
```