CHAPTER 4

What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Try and Except Statement - Catching Exceptions

In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

Syntax

```
try:
You do your operations here;
.....
except:
If there is any exception, then execute this block.
.....
else:
If there is no exception then execute this block.
```

Example

```
a = ["Python", "Exceptions", "try and except"]
try:
   #looping through the elements of the array a, choosing a range that goes beyond the l
ength of the array
    for i in range( 4 ):
      print( "The index and element from the array is", i, a[i] )
#if an error occurs in the try block, then except block will be executed by the Python int
erpreter
except:
   print ("Index out of range")
```

Raise an Exception

If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

```
#Python code to show how to raise an exception in Python
num = [3, 4, 5, 7]
if len(num) > 3:
   raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )
```

Output:

```
 raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )
Exception: Length of the given list must be less than or equal to 3 but is 4
```

The implementation stops and shows our exception in the output, providing indications as to what went incorrect.

Try with Else Clause

Python also supports the else clause, which should come after every except clause, in the try, and except blocks. Only when the try clause fails to throw an exception the Python interpreter goes on to the else block.

Here is an instance of a try clause with an else clause.

Code

```
# Python program to show how to use else clause with try and except clauses
# Defining a function which returns reciprocal of a number
def reciprocal( num1 ):
   try:
      reci = 1 / num1
   except ZeroDivisionError:
      print( "We cannot divide by zero" )
   else:
      print ( reci )
# Calling the function and passing values
reciprocal( 4 )
reciprocal( 0 )
```

Output:

```
0.25
We cannot divide by zero
```

Finally Keyword in Python

The finally keyword is available in Python, and it is always used after the try-except block. The finally code block is always executed after the try block has terminated normally or after the try block has terminated for some other reason.

Here is an example of finally keyword with try-except clauses:

Code

```
# Python code to show the use of finally clause

# Raising an exception in try block

try:

   div = 4 // 0

   print( div )

# this block will handle the exception raised

except ZeroDivisionError:

   print( "Atepting to divide by zero" )

# this will always be executed no matter exception is raised or not

finally:

   print( 'This is code of finally clause' )
```

User-Defined Exceptions

By inheriting classes from the typical built-in exceptions, Python also lets us design our customized exceptions.

Here is an illustration of a RuntimeError. In this case, a class that derives from RuntimeError is produced. Once an exception is detected, we can use this to display additional detailed information.

We raise a user-defined exception in the try block and then handle the exception in the except block. An example of the class EmptyError is created using the variable var.

Code

```
class EmptyError( RuntimeError ):

  def __init__(self, argument):

    self.arguments = argument
```

Once the preceding class has been created, the following is how to raise an exception:

Code

```
var = " "

try:

   raise EmptyError( "The variable is empty" )

except (EmptyError, var):

   print( var.arguments )
```

Output:

```
    2 try:
----> 3     raise EmptyError( "The variable is empty" )
    4 except (EmptyError, var):


EmptyError: The variable is empty
```

Assertions in Python

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.

The simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.

Assertions are made via the assert statement, which was added in Python 1.5 as the latest keyword.

Assertions are commonly used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.

The assert Statement

Python examines the adjacent expression, preferably true when it finds an assert statement. Python throws an AssertionError exception if the result of the expression is false.

The syntax for the assert clause is –

```
assert Expressions[, Argument]
```

Python uses ArgumentException, if the assertion fails, as the argument for the AssertionError. We can use the try-except clause to catch and handle AssertionError exceptions, but if they aren't, the program will stop, and the Python interpreter will generate a traceback.

Code

```
#Python program to show how to use assert keyword
# defining a function
def square_root( Number ):
   assert ( Number < 0), "Give a positive integer"
   return Number**(1/2)
 #Calling function and passing the values
print( square_root( 36 ) )
print( square_root( -36 ) )
```

Output:

```
    7 #Calling function and passing the values
----> 8 print( square_root( 36 ) )
    9 print( square_root( -36 ) )
```

```
Input In [23], in square_root(Number)
      3 def square_root( Number ):
----> 4     assert ( Number < 0), "Give a positive integer"
      5     return Number**(1/2)
AssertionError: Give a positive integer
```

Clean Up Actions in Python

There are numerous situation occurs when we want our program to do this specific task, irrespective of whether it runs perfectly or thrown some error. Mostly to catch at any errors or exceptions, we use to try and except block.

The "try" statement provides very useful optional clause which is meant for defining 'clean-up actions' that must be executed under any circumstances. For example –

```
>>> try:
  raise SyntaxError
finally:
  print("Learning Python!")
Learning Python!
Traceback (most recent call last):
  File "<pyshell#11>", line 2, in <module>
    raise SyntaxError
  File "<string>", line None
SyntaxError: <no detail available>
```

The final clause will execute no matter what, however, the else clause executes only if an exception was not raised.

Example1 − Consider below example, where everything looks ok and writing to a file with no exception (the program is working), will output the following –

```
file = open('finally.txt', 'w')
try:
  file.write("Testing1 2 3.")
  print("Writing to file.")
except IOError:
  print("Could not write to file.")
else:
  print("Write successful.")
finally:
  file.close()
```

```
  print("File closed.")
```

On running above program, will get −

Writing to file.

Write successful.

File closed.

Example 2 − Let's try to raise an exception by making a file read-only and try to write onto it, thus causing it to raise an exception.

```
file = open('finally.txt', 'r')
try:
  file.write("Testing1 2 3.")
  print("Writing to file.")
except IOError:
  print("Could not write to file.")
else:
  print("Write successful.")
finally:
  file.close()
  print("File closed.")
```

Above program will give an output, something like −

Could not write to file.

File closed.

In case we have an error, but we have not put any except clause to handle it. In such a case, the clean-up action (finally block) will be executed first and then the error is raised by the compiler. Let's understand the concept with the below example −

Example

```
file = open('finally.txt', 'r')
try:
  file.write(4)
  print("Writing to file.")
except IOError:
  print("Could not write to file.")
else:
  print("Write successful.")
finally:
  file.close()
```

```
print("File closed.")
```

Output

File closed.

Traceback (most recent call last):

File "C:/Python/Python361/finally_try_except1.py", line 4, in <module>

file.write(4)

TypeError: write() argument must be str, not int

So from above, we can see that, finally, clause executes always, irrespective of whether an exception occurs or not.

GUI Programming with Tkinter

Python provides the standard library Tkinter for creating the graphical user interface for desktop based applications.

Developing desktop based applications with python Tkinter is not a complex task. An empty Tkinter top-level window can be created by using the following steps.

1. import the Tkinter module.
2. Create the main application window.
3. Add the widgets like labels, buttons, frames, etc. to the window.
4. Call the main event loop so that the actions can take place on the user's computer screen.

Example:

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

Tkinter widgets

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

SN	Widget	Description
1	Button	The Button is used to add various kinds of buttons to the python application.
2	Canvas	The canvas widget is used to draw the canvas on the window.
3	Checkbutton	The Checkbutton is used to display the CheckButton on the window.
4	Entry	The entry widget is used to display the single-line text field to the user. It is commonly used to accept user values.
5	Frame	It can be defined as a container to which, another widget can be added and organized.

6	Label	A label is a text used to display some message or information about the other widgets.
7	ListBox	The ListBox widget is used to display a list of options to the user.
8	Menubutton	The Menubutton is used to display the menu items to the user.
9	Menu	It is used to add menu items to the user.
10	Message	The Message widget is used to display the message-box to the user.
11	Radiobutton	The Radiobutton is different from a checkbutton. Here, the user is provided with various options and the user can select only one option among them.
12	Scale	It is used to provide the slider to the user.
13	Scrollbar	It provides the scrollbar to the user so that the user can scroll the window up and down.
14	Text	It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it.
14	Toplevel	It is used to create a separate window container.
15	Spinbox	It is an entry widget used to select from options of values.
16	PanedWindow	It is like a container widget that contains horizontal or vertical panes.
17	LabelFrame	A LabelFrame is a container widget that acts as the container
18	MessageBox	This module is used to display the message-box in the desktop based applications.

Python Tkinter Geometry

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

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

Let's discuss each one of them in detail.

Python Tkinter pack() method

The pack() widget is used to organize widget in the block. The positions widgets added to the python application using the pack() method can be controlled by using the various options specified in the method call.

However, the controls are less and widgets are generally added in the less organized manner.

The syntax to use the pack() is given below.

syntax

1. widget.pack(options)

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

- **expand:** If the expand is set to true, the widget expands to fill any space.
- **Fill:** By default, the fill is set to NONE. However, we can set it to X or Y to determine whether the widget contains any extra space.
- **size:** it represents the side of the parent to which the widget is to be placed on the window.

Python Tkinter grid() method

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

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

Syntax

1. widget.grid(options)

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

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

Python Tkinter place() method

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

Syntax

1. widget.place(options)

A list of possible options is given below.

- **Anchor:** It represents the exact position of the widget within the container. The default value (direction) is NW (the upper left corner)

- **bordermode:** The default value of the border type is INSIDE that refers to ignore the parent's inside the border. The other option is OUTSIDE.
- **height, width:** It refers to the height and width in pixels.
- **relheight, relwidth:** It is represented as the float between 0.0 and 1.0 indicating the fraction of the parent's height and width.
- **relx, rely:** It is represented as the float between 0.0 and 1.0 that is the offset in the horizontal and vertical direction.
- **x, y:** It refers to the horizontal and vertical offset in the pixels.

Python - Tkinter Button

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

Syntax

Here is the simple syntax to create this widget –

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

Parameters

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

SN	Option	Description
1	activebackground	It represents the background of the button when the mouse hover the button.
2	activeforeground	It represents the font color of the button when the mouse hover the button.
3	Bd	It represents the border width in pixels.
4	Bg	It represents the background color of the button.
5	Command	It is set to the function call which is scheduled when the function is called.
6	Fg	Foreground color of the button.
7	Font	The font of the button text.
8	Height	The height of the button. The height is represented in the number of text lines for the textual lines or the number of pixels for the images.
10	Highlightcolor	The color of the highlight when the button has the focus.
11	Image	It is set to the image displayed on the button.
12	justify	It illustrates the way by which the multiple text lines are represented. It is set to LEFT for left justification, RIGHT for the right justification, and CENTER for the center.

13	Padx	Additional padding to the button in the horizontal direction.
14	pady	Additional padding to the button in the vertical direction.
15	Relief	It represents the type of the border. It can be SUNKEN, RAISED, GROOVE, and RIDGE.
17	State	This option is set to DISABLED to make the button unresponsive. The ACTIVE represents the active state of the button.
18	Underline	Set this option to make the button text underlined.
19	Width	The width of the button. It exists as a number of letters for textual buttons or pixels for image buttons.
20	Wraplength	If the value is set to a positive number, the text lines will be wrapped to fit within this length.

Example

```
from tkinter import *
top = Tk()
top.geometry("200x100")
  def fun():
    messagebox.showinfo("Hello", "Red Button clicked")
  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 = 10)
b1.pack(side = LEFT)
b2.pack(side = RIGHT)
b3.pack(side = TOP)
b4.pack(side = BOTTOM)
top.mainloop()
```

Python Tkinter Entry

The Entry widget is used to provde the single line text-box to the user to accept a value from the user. We can use the Entry widget to accept the text strings from the user. It can only be used for one line of text from the user. For multiple lines of text, we must use the text widget.

The syntax to use the Entry widget is given below.

Syntax

1. w = Entry (parent, options)

Example

```
1. # !/usr/bin/python3
2. from tkinter import *
3. top = Tk()
4. top.geometry("400x250")
5. name = Label(top, text = "Name").place(x = 30,y = 50)
6. email = Label(top, text = "Email").place(x = 30, y = 90)
7. password = Label(top, text = "Password").place(x = 30, y = 130)
8. sbmitbtn = Button(top, text = "Submit",activebackground = "pink", activeforeground = "
   blue").place(x = 30, y = 170)
9. e1 = Entry(top).place(x = 80, y = 50)
10. e2 = Entry(top).place(x = 80, y = 90)
11. e3 = Entry(top).place(x = 95, y = 130)
12. top.mainloop()
```

Python Tkinter Label

The Label is used to specify the container box where we can place the text or images. This widget is used to provide the message to the user about other widgets used in the python application.

There are the various options which can be specified to configure the text or the part of the text shown in the Label.

The syntax to use the Label is given below.

Syntax

1. **w = Label (master, options)**

Example 1

```
1. # !/usr/bin/python3
2. from tkinter import *
3. top = Tk()
4. top.geometry("400x250")
5. #creating label
6. uname = Label(top, text = "Username").place(x = 30,y = 50)
7. #creating label
8. password = Label(top, text = "Password").place(x = 30, y = 90)
9. sbmitbtn = Button(top, text = "Submit",activebackground = "pink", activeforeground = "
   blue").place(x = 30, y = 120)
10. e1 = Entry(top,width = 20).place(x = 100, y = 50)
11. e2 = Entry(top, width = 20).place(x = 100, y = 90)
12. top.mainloop()
```

Python Tkinter Listbox

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)
```

Example 1

```
# !/usr/bin/python3
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()
```

Python Tkinter Checkbutton

The Checkbutton is used to track the user's choices provided to the application. In other words, we can say that Checkbutton is used to implement the on/off selections.

The Checkbutton can contain the text or images. The Checkbutton is mostly used to provide many choices to the user among which, the user needs to choose the one. It generally implements many of many selections.

The syntax to use the checkbutton is given below.

Syntax

```
w = checkbutton(master, options)
```

Example

```
from tkinter import *
top = Tk()
top.geometry("200x200")
checkvar1 = IntVar()
checkvar2 = IntVar()
checkvar3 = IntVar()
```

```
7. chkbtn1 = Checkbutton(top, text = "C", variable = checkvar1, onvalue = 1, offvalue = 0, h
   eight = 2, width = 10)
8. chkbtn2 = Checkbutton(top, text = "C++", variable = checkvar2, onvalue = 1, offvalue =
   0, height = 2, width = 10)
9. chkbtn3 = Checkbutton(top, text = "Java", variable = checkvar3, onvalue = 1, offvalue =
   0, height = 2, width = 10)
10. chkbtn1.pack()
11. chkbtn2.pack()
12. chkbtn3.pack()
13. top.mainloop()
```

Python Tkinter Radiobutton

The Radiobutton widget is used to implement one-of-many selection in the python application. It shows multiple choices to the user out of which, the user can select only one out of them. We can associate different methods with each of the radiobutton.

We can display the multiple line text or images on the radiobuttons. To keep track the user's selection the radiobutton, it is associated with a single variable. Each button displays a single value for that particular variable.

The syntax to use the Radiobutton is given below.

Syntax

```
1. w = Radiobutton(top, options)
```

Example

```
1. from tkinter import *
2. def selection():
3.    selection = "You selected the option " + str(radio.get())
4.    label.config(text = selection)
5.  top = Tk()
6. top.geometry("300x150")
7. radio = IntVar()
8. lbl = Label(text = "Favourite programming language:")
9. lbl.pack()
10. R1 = Radiobutton(top, text="C", variable=radio, value=1,  command=selection)
11. R1.pack( anchor = W )
12.  R2 = Radiobutton(top, text="C++", variable=radio, value=2, command=selection)
13. R2.pack( anchor = W )
14. R3 = Radiobutton(top, text="Java", variable=radio, value=3,  command=selection)
15. R3.pack( anchor = W)
16.  label = Label(top)
```

17. label.pack()
18. top.mainloop()

Python Tkinter Menu

The Menu widget is used to create various types of menus (top level, pull down, and pop up) in the python application.

The top-level menus are the one which is displayed just under the title bar of the parent window. We need to create a new instance of the Menu widget and add various commands to it by using the add() method.

The syntax to use the Menu widget is given below.

Syntax

1. w = Menu(top, options)

Creating a top level menu

A top-level menu can be created by instantiating the Menu widget and adding the menu items to the menu.

Example 1

```
1. # !/usr/bin/python3
2. from tkinter import *
3. top = Tk()
4. def hello():
5.     print("hello!")
6. # create a toplevel menu
7. menubar = Menu(root)
8. menubar.add_command(label="Hello!", command=hello)
9. menubar.add_command(label="Quit!", command=top.quit)
10.  # display the menu
11. top.config(menu=menubar)
12. top.mainloop()
```

Example 2

```
1. from tkinter import Toplevel, Button, Tk, Menu
2. top = Tk()
3. menubar = Menu(top)
4. file = Menu(menubar, tearoff=0)
5. file.add_command(label="New")
6. file.add_command(label="Open")
7. file.add_command(label="Save")
```

8. file.add_command(label="Save as...")
9. file.add_command(label="Close")
10. file.add_separator()
11. file.add_command(label="Exit", command=top.quit)
12. menubar.add_cascade(label="File", menu=file)
13. edit = Menu(menubar, tearoff=0)
14. edit.add_command(label="Undo")
15. edit.add_separator()
16. edit.add_command(label="Cut")
17. edit.add_command(label="Copy")
18. edit.add_command(label="Paste")
19. edit.add_command(label="Delete")
20. edit.add_command(label="Select All")
21. menubar.add_cascade(label="Edit", menu=edit)
22. help = Menu(menubar, tearoff=0)
23. help.add_command(label="About")
24. menubar.add_cascade(label="Help", menu=help)
25. top.config(menu=menubar)
26. top.mainloop()

Python - Tkinter Canvas

The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets or frames on a Canvas.

Syntax

Here is the simple syntax to create this widget –

```
w = Canvas ( master, option=value, ... )
```

Parameters

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

The Canvas widget can support the following standard items –

- arc − Creates an arc item, which can be a chord, a pieslice or a simple arc.
- coord = 10, 50, 240, 210
- arc = canvas.create_arc(coord, start=0, extent=150, fill="blue")
- image − Creates an image item, which can be an instance of either the BitmapImage or the PhotoImage classes.
- filename = PhotoImage(file = "sunshine.gif")

- image = canvas.create_image(50, 50, anchor=NE, image=filename)
- line − Creates a line item.
- line = canvas.create_line(x0, y0, x1, y1, ..., xn, yn, options)
- oval − Creates a circle or an ellipse at the given coordinates. It takes two pairs of coordinates; the top left and bottom right corners of the bounding rectangle for the oval.
- oval = canvas.create_oval(x0, y0, x1, y1, options)
- polygon − Creates a polygon item that must have at least three vertices.
- oval = canvas.create_polygon(x0, y0, x1, y1,...xn, yn, options)

Example: Creating an arc

```
1. from tkinter import *
2. top = Tk()
3. top.geometry("200x200")
4. #creating a simple canvas
5. c = Canvas(top,bg = "pink",height = "200",width = 200)
6. arc = c.create_arc((5,10,150,200),start = 0,extent = 150, fill= "white")
7. c.pack()
8. top.mainloop()
```

*****