Unit 06 [Marks: 12]

File I/O Handling & Exception Handling

I/O Operations: Reading Keyborad i/p, Printing to Screen:

1) Input:

1) There are multiple ways to take input. E.g. keyboard, mouse, internet.
2) Keyboard is the one input method that is widely used in order to get input From the user.
3) Python has supported this feature with the input() function. This function has optional argument called prompt string.
4) After this functionhas been invoked, the first message that will get displayed on the screen will be from the prompt string.
5) then the program will stop until the user enter inputs and return the entered input as string.

2) Output:

1) The print function is use to display output on consol screen.
2) syntax:

```
print(“Hello”)
```

Output: Hello

File:

1) File is a storage space for data or information. File is an object that preserves Data or information.
2) The most important task for a file is to store the user data.
3) Whenever needed we can retrive or update the data in files. Files are stored On persistent storage space.
4) It means your data will be saved on the devices even after we cut the power For the device.

5) Using filename we can access or update data or information as & when needed.
6) Every file has a name associated with it called filename.

Python File Handling:

1) Till now, we were taking the input from the console and writing it back to the console to interact with the user.
2) Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.
3) The file handling plays an important role when the data needs to be stored permanently into the file.
4) A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.
5) The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.
6) In Python, files are treated in two modes as **text or binary**. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

1) Opening a file

1) Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed.

2)The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

1. file object = open(<file-name>, <access-mode>, <buffering>)

The following are the details about the access mode to open a file.

SN	Access mode	Description
1	r	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example

```
#opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
    print("file is opened successfully")
```

Output:

```
<class '_io.TextIOWrapper'>
file is opened successfully
```

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement

2) The close() method

1) Once all the operations are done on the file, we must close it through our Python script using the **close()** method.

2) Any unwritten information gets destroyed once the **close()** method is called on a file object.

3) We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

Syntax

1. fileobject.close()

Consider the following example.

```
# opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
   print("file is opened successfully")

#closes the opened file
fileptr.close()
```

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

```
try:
  fileptr = open("file.txt")
  # perform file operations
finally:
  fileptr.close()
```

Writing the file

1) To write some text to a file, we need to open the file using the open method with one of the following access modes.
2) **w:** It will overwrite the file if any file exists. The file pointer is at the beginning of the file.
3) **a:** It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

Example

```
# open the file.txt in append mode. Create a new file if no such file exists.
fileptr = open("file2.txt", "w")

# appending the content to the file
```

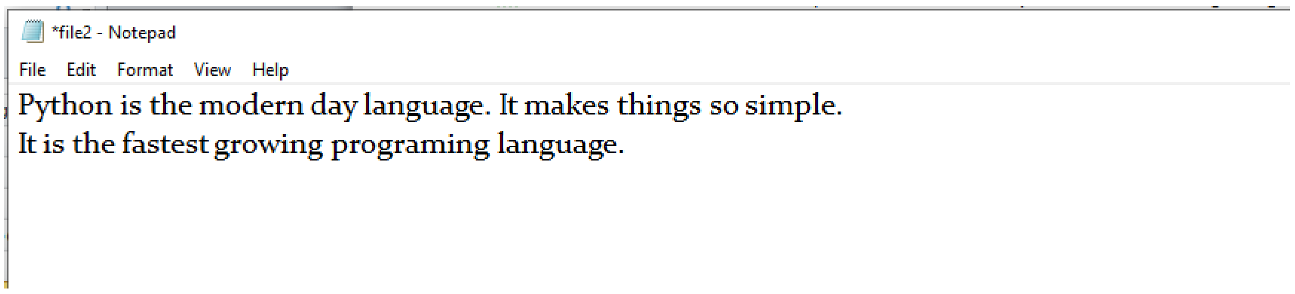
```
fileptr.write("""Python is the modern day language. It makes things so simple.
It is the fastest-growing programing language""")

# closing the opened the file
fileptr.close()
```

Output:

File2.txt

```
Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.
```

Snapshot of the file2.txt



```
Python is the modern day language. It makes things so simple.
It is the fastest growing programing language.
```

We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.

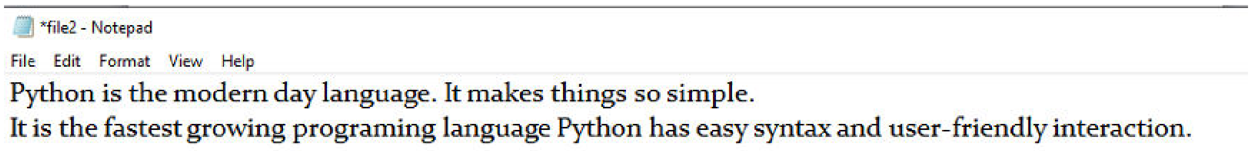
Example 2

```
#open the file.txt in write mode.
fileptr = open("file2.txt","a")

#overwriting the content of the file
fileptr.write(" Python has an easy syntax and user-friendly interaction.")

#closing the opened file
fileptr.close()
```

Output:

```
Python is the modern day language. It makes things so simple.
It is the fastest growing programing language Python has an easy syntax and user-friendly interaction.
```

Snapshot of the file2.txt



```
Python is the modern day language. It makes things so simple.
It is the fastest growing programing language Python has easy syntax and user-friendly interaction.
```

We can see that the content of the file is modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

Syntax:

1. fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

Example

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r")
#stores all the data of the file into the variable content
content = fileptr.read(10)
# prints the type of the data stored in the file
print(type(content))
#prints the content of the file
print(content)
#closes the opened file
fileptr.close()
```

Output:

```
<class 'str'>
Python is
```

In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as ten which means it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

```
content = fileptr.read()
print(content)
```

Output:

```
Python is the modern-day language. It makes things so simple.
It is the fastest-growing programing language Python has easy an syntax and user-friendly interaction.
```

Read file through for loop

We can read the file using for loop. Consider the following example.

```
#open the file.txt in read mode. causes an error if no such file exists.
fileptr = open("file2.txt","r");
#running a for loop
for i in fileptr:
    print(i) # i contains each line of the file
```

Output:

```
Python is the modern day language.
It makes things so simple.
Python has easy syntax and user-friendly interaction.
```

Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

Example 1: Reading lines using readline() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");
#stores all the data of the file into the variable content
content = fileptr.readline()
content1 = fileptr.readline()
#prints the content of the file
print(content)
print(content1)
#closes the opened file
fileptr.close()
```

Output:

```
Python is the modern day language.
It makes things so simple.
```

We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

Example 2: Reading Lines Using readlines() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content
content = fileptr.readlines()

#prints the content of the file
print(content)
```

```
#closes the opened file
fileptr.close()
```

Output:

```
['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']
```

Creating a new file

The new file can be created by using one of the following access modes with the function open().

1) **x:** it creates a new file with the specified name. It causes an error a file exists with the same name.
2) **a:** It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.
3) **w:** It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Example 1

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","x")
print(fileptr)
if fileptr:
    print("File created successfully")
```

Output:

```
<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>
File created successfully
```

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#reading the content of the file
content = fileptr.read();

#after the read operation file pointer modifies. tell() returns the location of the fi
leptr.

print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

Modifying file pointer position

1) In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.
2) For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.
3) The syntax to use the seek() method is given below.

Syntax:

```
<file-ptr>.seek(offset[, from)
```

The seek() method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

Example

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#changing the file pointer location to 10.
fileptr.seek(10);

#tell() returns the location of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 10
```

Renaming the file

1) The Python **os** module enables interaction with the operating system.
2) The os module provides the functions that are involved in file processing operations like renaming, deleting, etc.
3) 3)It provides us the rename() method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

Syntax:

```
rename(current-name, new-name)
```

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

Example 1:

```
import os

#rename file2.txt to file3.txt
os.rename("file2.txt","file3.txt")
```

Output:

The above code renamed current **file2.txt** to **file3.txt**

Removing the file

The os module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

```
remove(file-name)
```

Example 1

```
import os;
#deleting the file named file3.txt
os.remove("file3.txt")
```

Creating the new directory

The **mkdir()** method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

Syntax:

```
mkdir(directory name)
```

Example 1

```
import os

#creating a new directory with the name new
os.mkdir("new")
```

The getcwd() method

This method returns the current working directory.

The syntax to use the getcwd() method is given below.

Syntax

1. os.getcwd()

Example

```
import os
os.getcwd()
```

Output:

```
'C:\\Users\\Neer'
```

Changing the current working directory

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is given below.

Syntax

```
chdir("new-directory")
```

Example

```
import os
# Changing current directory with the new directiory
os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")
#It will display the current working directory
os.getcwd()
```

Output:

```
'C:\\Users\\Neer\\Documents'
```

Deleting directory

The rmdir() method is used to delete the specified directory.
The syntax to use the rmdir() method is given below.

Syntax

```
os.rmdir(directory name)
```

Example 1

```
import os
#removing the new directory
os.rmdir("directory_name")
```

It will remove the specified directory.

Writing Python output to the files

1) In Python, there are the requirements to write the output of a Python script to a file.

2) The **check_call()** method of module **subprocess** is used to execute a Python script and write the output of that script to a file.

3) The following example contains two python scripts. The script file1.py executes the script file.py and writes its output to the text file **output.txt.**

Example

file.py

```
temperatures=[10,-20,-289,100]
def c_to_f(c):
    if c< -273.15:
        return "That temperature doesn't make sense!"
    else:
        f=c*9/5+32
        return f
for t in temperatures:
        print(c_to_f(t))
```

file.py

```
import subprocess

with open("output.txt", "wb") as f:
    subprocess.check_call(["python", "file.py"], stdout=f)
```

SN	Method	Description
1	file.close()	It closes the opened file. The file once closed, it can't be read or write anymore.
2	File.fush()	It flushes the internal buffer.
3	File.fileno()	It returns the file descriptor used by the underlying implementation to request I/O from the OS.
4	File.isatty()	It returns true if the file is connected to a TTY device, otherwise returns false.
5	File.next()	It returns the next line from the file.
6	File.read([size])	It reads the file for the specified size.
7	File.readline([size])	It reads one line from the file and places the file pointer to the beginning of the new line.
8	File.readlines([sizehint])	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function.
9	File.seek(offset[,from)	It modifies the position of the file pointer to a specified offset with the specified reference.
10	File.tell()	It returns the current position of the file pointer within the file.
11	File.truncate([size])	It truncates the file to the optional specified size.
12	File.write(str)	It writes the specified string to a file
13	File.writelines(seq)	It writes a sequence of the strings to a file.

Python Exception

1) An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

2) Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script.

3) An exception is a Python object that represents an error.

4) Python provides a way to handle the exception so that the code can be executed without any interruption.

5) If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

6) Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

The problem without handling exceptions

1) As we have already discussed, the exception is an abnormal condition that halts the execution of the program.

2) Suppose we have two variables **a** and **b**, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator?

3) It will interrupt the program execution and through a **ZeroDivision** exception. Let's see the following example.

Example

```
a = int(input("Enter a:"))
```

```
b = int(input("Enter b:"))
c = a/b
print("a/b = %d" %c)

#other code:
print("Hi I am other part of the program")
```

Output:

```
Enter a:10
Enter b:0
Traceback (most recent call last):
  File "exception-test.py", line 3, in <module>
    c = a/b;
ZeroDivisionError: division by zero
```

4) The above program is syntactically correct, but it through the error because of unusual input.

5) That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution.

6) That's why an exception-handling plays an essential role in handling these unexpected exceptions.

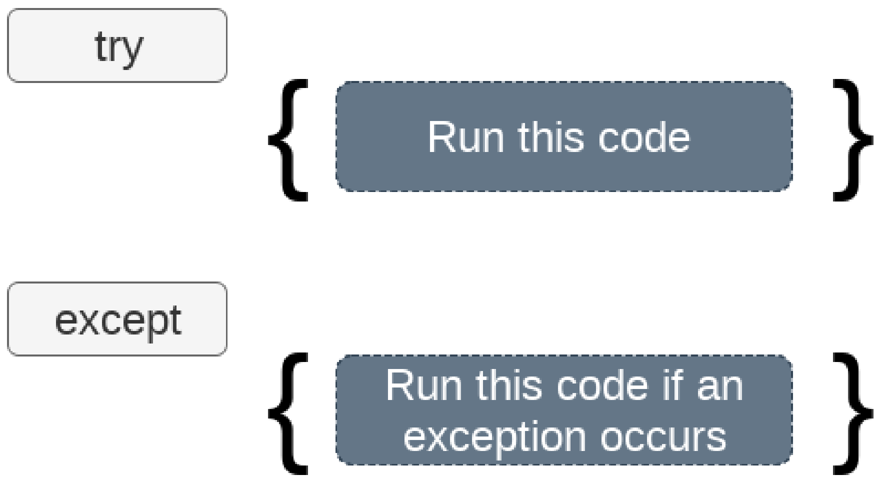
We can handle these exceptions in the following way.

Exception handling in python

The try-expect statement

1) If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block.

2) The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.



Syntax

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```

Consider the following example.

Example 1

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```

Output:

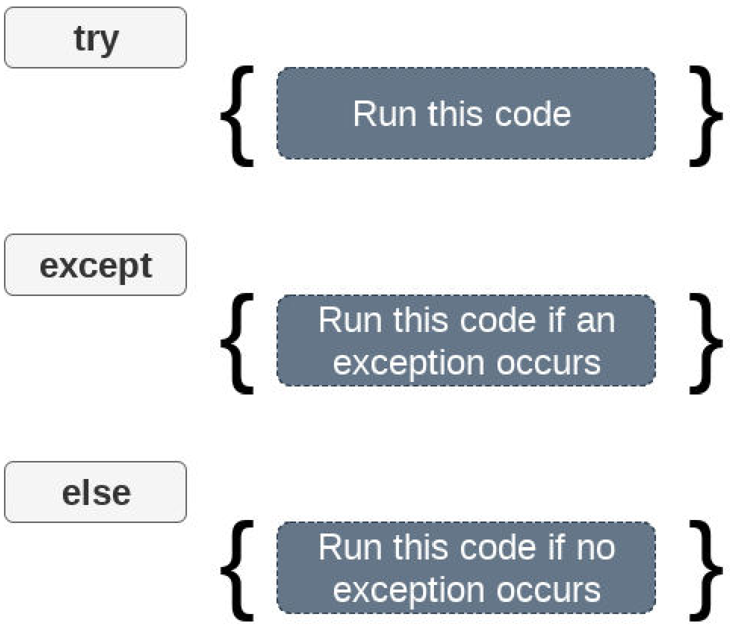
```
Enter a:10
Enter b:0
```

```
Can't divide with zero
```

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

The syntax to use the else statement with the try-except statement is given below.

```
try:
    #block of code

except Exception1:
    #block of code
else:
    #this code executes if no except block is executed
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs

Consider the following program.

Example 2

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
# Using Exception with except statement. If we print(Exception) it will return e
xception class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
<class 'Exception'>
```

The except statement with no exception

Python provides the flexibility not to specify the name of exception with the exception statement.

Consider the following example.

Example

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except:
    print("can't divide by zero")
else:
    print("Hi I am else block")
```

The except statement using with exception variable

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

```
try:
      a = int(input("Enter a:"))
      b = int(input("Enter b:"))
      c = a/b
      print("a/b = %d"%c)
      # Using exception object with the except statement
   except Exception as e:
      print("can't divide by zero")
      print(e)
   else:
      print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
division by zero
```

Points to remember

1. Python facilitates us to not specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.

Example

```
    try:
       #this will throw an exception if the file doesn't exist.
       fileptr = open("file.txt","r")
    except IOError:
       print("File not found")
    else:
       print("The file opened successfully")
       fileptr.close()
```

Output:

```
File not found
```

Declaring Multiple Exceptions

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. The syntax is given below.

Syntax

```
    try:
       #block of code

    except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
       #block of code

    else:
       #block of code
```

Consider the following example.

```
try:
   a=10/0;
except(ArithmeticError, IOError):
   print("Arithmetic Exception")
else:
   print("Successfully Done")
```

Output:

```
Arithmetic Exception
```

The try...finally block

1) Python provides the optional **finally** statement, which is used with the **try** statement.

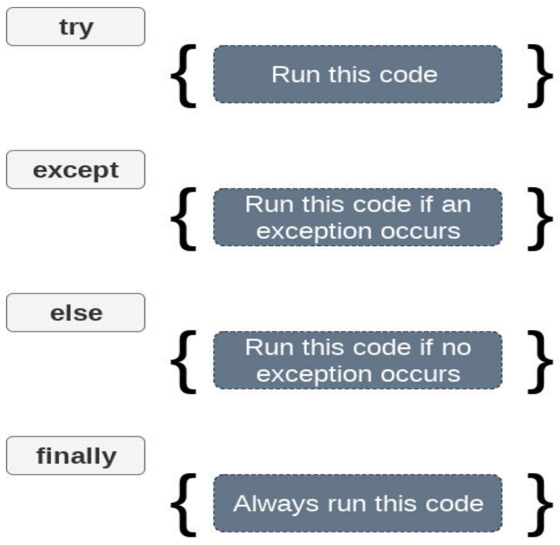
2) It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

3) We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

Syntax:

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


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs
finally
Always run this code

Example:

```
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

Output:

```
file closed
Error
```

Raising exceptions

1) An exception can be raised forcefully by using the **raise** clause in Python.

2) It is useful in in that scenario where we need to raise an exception to stop the execution of the program.

3) For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

Syntax

```
raise Exception_class,<value>
```

Points to remember

1. To raise an exception, the raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.
3. To access the value "**as**" keyword is used. "**e**" is used as a reference variable which stores the value of the exception.

4. We can pass the value to an exception to specify the exception type.

Example

```
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Output:

```
Enter the age:17
The age is not valid
```

Example 2 Raise the exception with message

```
try:
     num = int(input("Enter a positive integer: "))
     if(num <= 0):
# we can pass the message in the raise statement
          raise ValueError("That is  a negative number!")
except ValueError as e:
     print(e)
```

Output:

```
Enter a positive integer: -5
That is a negative number!
```

User Defined Exception:

1) Python throws errors and exceptions whenever code behaves abnormally & its execution stop abruptly.

2) Python provides us tools to handle such scenarios by the help of exception handling method using try-except statements.

3) Some standard exceptions are ArithmeticError, AssertionError, AttributeError, ImportError, etc.

Creating a User-defined Exception class

Here we created a new exception class i.e. User_Error. Exceptions need to be derived from the built-in Exception class, either directly or indirectly. Let’s look at the given example which contains a constructor and display method within the given class.

Example

```
# class MyError is extended from super class Exception
class User_Error(Exception):
  # Constructor method
  def __init__(self, value):
    self.value = value
  # __str__ display function
  def __str__(self):
    return(repr(self.value))
try:
  raise(User_Error("User defined error"))
  # Value of Exception is stored in error
except User_Error as error:
  print('A New Exception occured:',error.value)
```

Output

```
A New Exception occured: User defined error
```

Creating a User-defined Exception class (Multiple Inheritance)

Derived class Exceptions are created when a single module handles multiple several distinct errors. Here we created a base class for exceptions defined by that module. This base class is inherited by various user-defined class to handle different types of errors.

Example

```
# define Python user-defined exceptions
class Error(Exception):
  """Base class for other exceptions"""
  pass
class Dividebyzero(Error):
  """Raised when the input value is zero"""
  pass
try:
  i_num = int(input("Enter a number: "))
  if i_num ==0:
    raise Dividebyzero
except Dividebyzero:
  print("Input value is zero, try again!")
  print()
```

Output

```
Enter a number: Input value is zero, try again!
```