Python Programming

Unit-II

Functions

A function can be defined as the organized block of reusable code which can be called whenever required.

In other words, Python allows us to divide a large program into the basic building blocks known as function.

Python provide us various inbuilt functions like range() or print(). Although, the user can able to create functions which can be called user-defined functions.

Creating Function:

In python, a function can be created by using **def** keyword.

Syntax:

```
def my_function():
    Statements
    return statement
```

The function block is started with the colon (:) and all the same level block statements remain at the same indentation.

Example:

```
def sample():            #function definition
        print ("Hello world")
sample()        #function calling
```

Output:

```
Hello world
```

The information into the functions can be passed as the parameters. The parameters are specified in the parentheses. We can give any number of parameters, but we have to separate them with a comma.

Example:

```
def welcome(name):
        print(“function with one parameter”)
        print(“welcome : ”,name)
welcome(“Kiran”)
```

Output:

```
function with one parameter
Welcome : Kiran
```

Example:

```
def sum (a,b):
    return a+b;

#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))
#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output:

```
Enter a: 10
Enter b: 15
Sum = 25
```

Built-in Functions in Python:

The Python supports several built-in functions, those are

○ bin()	○ int()	○ max()
○ hex()	○ float()	○ min()
○ oct()	○ str()	○ sum()
○ ord()	○ list()	○ sorted()
○ eval()	○ tuple()	○ input()
○ chr()	○ set()	○ range()
○ abs()	○ len()	○ type()

- **bin():**

  In python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

Syntax:

```
bin (num)
```

Example:

```
x =  14
y =  bin(x)
print (y)
```

Output:

```
0b1110
```

- **hex():**

  In python **hex()** function is used to return the hexadecimal representation of a specified integer. A result always starts with the prefix 0x.

Syntax:

```
hex (num)
```

Example:

```
x =  14
y =  hex(x)
print (y)
```

Output:

```
0xe
```

- **oct():**

  In python **oct()** function is used to return the octal representation of a specified integer. A result always starts with the prefix 0o.

Syntax:

```
oct (num)
```

Example:

```
x =  14
y =  oct(x)
print (y)
```

Output:

```
0o16
```

- **ord():**

  In python **ord()** function is used to return the ASCII(American Standard Code for Information Interchange) value of a specified character.

Syntax:

```
ord (character)
```

Example:

```
print("ASCII value of A :",ord('A'))
print("ASCII value of a :",ord('a'))
print("ASCII value of 1 :",ord('1'))
```

Output:

```
ASCII value of A : 65
ASCII value of a : 97
ASCII value of 1 : 49
```

- **chr():**

  In python **chr()** function is used to return the character of a specified ASCII(American Standard Code for Information Interchange) value.

Syntax:

```
chr (num)
```

Example:

```
print("Character value of 65 :",chr(65))
print("Character value of 97 :",chr(97))
print("Character value of 50 :",chr(50))
```

Output:

```
Character value of 65 : A
Character value of 97 : a
Character value of 50 : 2
```

- **eval():**

  In python **eval()** function is used to evaluate given expression. The expression must in quotes.

Syntax:

```
eval(expression)
```

Example:

```
x=int(input("Enter x :"))
y=int(input("Enter y :"))
print(eval('x+y'))
```

Output:

```
Enter x :13
Enter y :33
46
```

- **abs():**

  In python **abs()** function is used to return absolute value of given number.

Syntax:

```
abs(num)
```

Example:

```
x=-13
y=-22.15
print("Absolute value of x :",abs(x))
print("Absolute value of y :",abs(y))
```

Output:

```
Absolute value of x : 13
Absolute value of y : 22.15
```

Modules

In python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module.

Modules in Python provides us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

Creating Module:

Example:

calc.py

```
def sum(a,b):
        return a+b
def sub(a,b):
        return a-b
def mul(a,b):
        return a*b
def div(a,b):
        return a/b
```

Loading the module in our python code:

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. import statement
2. from-import statement

1. import statement:

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement.

Syntax:

```
        import module1,module2..
```

Example:

Moduledemo1.py

```
import calc                #importing entire Module
a=int(input("Enter a :"))
b=int(input("Enter b :"))
print("Sum is :",calc.sum(a,b))
print("Sub is :",calc.sub(a,b))
print("Mul is :",calc.mul(a,b))
print("Div is :",calc.div(a,b))
```

Output:

```
Enter a :12
Enter b :6
Sum is : 18
Sub is : 6
Mul is : 72
Div is : 2.0
```

2. from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from - import statement.

Syntax:

from module-name **import** function1,function2…

Example:

Moduledemo2.py

```
from calc import sub,mul          #importing specific functionality from Module
a=int(input("Enter a :"))
b=int(input("Enter b :"))
print("Sub is :",sub(a,b))
print("Mul is :",mul(a,b))
```

Output:

```
Enter a :12
Enter b :6
Sub is : 6
Mul is : 72
```

Renaming a module:

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

Syntax:

import module-name **as** specific-name

Example:

Moduledemo2.py

```
import calc as c
a=int(input("Enter a :"))
b=int(input("Enter b :"))
print("Sum is :",c.sum(a,b))
print("Sub is :",c.sub(a,b))
```

Output:

```
Enter a :25
Enter b :12
Sum is : 37
Sub is : 13
```

Scope of variables:

In Python, variables are associated with two types of scopes. All the variables defined in a module contain the global scope unless or until it is defined within a function.

All the variables defined inside a function contain a local scope that is limited to this function itself. We cannot access a local variable globally.

If two variables are defined with the same name with the two different scopes, i.e., local and global, then the priority will always be given to the local variable.

Example:

```
name = "Madhu"
def disp_name(name):
   print("Hi",name) #prints the name that is local to this function only.
name = input("Enter the name :")
disp_name(name)
```

Output:

```
Enter the name: Naveen
Hi Naveen
```

Namespaces in Python:

A namespace is basically a system to make sure that all the names in a program are unique and can be used without any conflict.

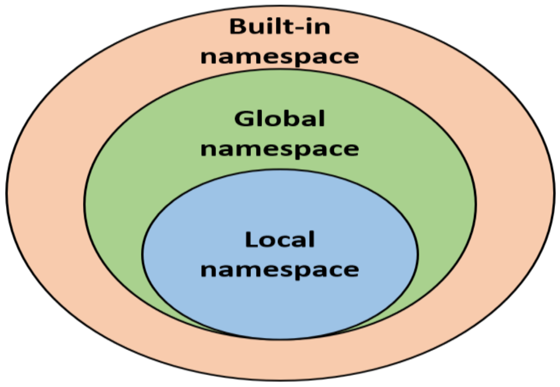
A namespace is a system to have a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary.

Let’s go through an example, a directory-file system structure in computers. Needless to say, that one can have multiple directories having a file with the same name inside of every directory.

Real-time example, the role of a namespace is like a surname. One might not find a single “Kumar” in the class there might be multiple “Kumar” but when you particularly ask for “N Kumar” or “S Kumar” (with a surname), there will be only one (time being don’t think of both first name and surname are same for multiple students).

Types of namespaces:

- **Local Namespace:**
  This namespace includes local names inside a function. This namespace is created when a function is called, and it only lasts until the function returns.
- **Global Namespace:**
  This namespace includes names from various imported modules that you are using in a project. It is created when the module is included in the project, and it lasts until the script ends.
- **Built-in Namespace:** This namespace includes built-in functions and built-in exception names. Like print (), input (), list () and etc.



Type of Namespaces

Example:

```
print("Namespace Example")          #built-in namespace
a=10            #global namespace
def func1():
        b=20    #local namespace
        print(a+b)
func1()
```

Output:

```
30
```

In above code, print () is built-in namespace, 'a' is in the global namespace in python and 'b' is in the local namespace of func1.

Python supports "global" keyword to update global namespaces in local.

Example:

```
count = 5
def func1():
   global count #To update global namespace
   count = count + 1
   print(count)
func1()
```

Output:

```
6
```

Example:

```
a=10 #global namespace
def func1():
        b=20    #non-local namespace
        def func2():
                nonlocal b
                c=30    #local namesapce
                global a
                a=a+c
                b=b+c
        func2()
        print(a,b)
func1()
```

Output:

```
40 50
```

To func2, ‘c’ is local, ‘b’ is nonlocal, and ‘a’ is global. By nonlocal, we mean it isn’t global, but isn’t local either. Of course, here, you can write ‘c’, and read both ‘b’ and ‘a’.

To update ‘b’ (non-local namespace), we need to use “nonlocal” keyword and to update ‘a’(global namespace), we need to “global” keyword.

Module Built-in Functions:

The Python interpreter has a number of built-in functions. They are loaded automatically as the interpreter starts and are always available. For example, print() and input() for I/O, number conversion functions int(), float(), complex(), data type conversions list(), tuple(), set(), etc.

A module is a file containing definitions of functions, classes, variables, constants or any other Python objects. Some functions are pre-defined in module; those are called as built-in functions in module.

Python supports following Module built-in functions, those are

- **dir():**

The dir() function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

Example:

module1.py

```
xy="Madhu"
def sum(a,b):
        return a+b
def sub(a,b):
        return a-b
def mul(a,b):
        return a*b
def div(a,b):
        return a/b
```

dirfun.py

```
import module1
ls=dir(module1)
print(ls)
```

Output:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'div', 'mul', 'sub', 'sum', 'xy']
```

- **globals() and locals() :**

The globals() and locals() functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

Example:

```
xy="Madhu"
def sum(a,b):
        c=0
        c=a+b
        print(c)
        print(globals())
        print(locals())
sum(2,3)
```

Output:

```
5
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_f
rozen_importlib_external.SourceFileLoader object at 0x006BDBD0>, '__spec__': Non
e, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__fil
e__': 'module1.py', '__cached__': None, 'xy': 'Madhu', 'sum': <function sum at 0
x00690858>}
{'a': 2, 'b': 3, 'c': 5}
```

- **reload():**

For reasons of efficiency, a module is only loaded once per interpreter session. That is fine for function and class definitions, which typically make up the bulk of a module's contents. But a module can contain executable statements as well, usually for initialization. Be aware that these statements will only be executed the *first time* a module is imported.

In python the **reload ()** reloads a previously imported module.

Syntax:

```
import importlib
importlib.reload(module)
```

Example:

mod.py

```
a = [100, 200, 300]
print('a =', a)
```

reldemo.py

```
import mod
import importlib
importlib.reload(mod)
```

Output:

```
a = [100, 200, 300]
```

❖ Python Packages

Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality.

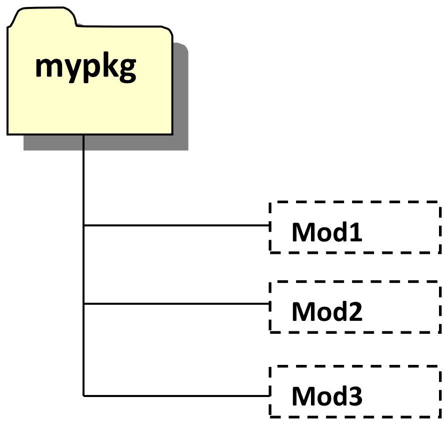
In python a Package contains collection of modules and sub-packages.

Packages are a way of structuring many packages and modules which help in a well-organized hierarchy of data set, making the packages and modules easy to access. Just like there are different drives and folders in an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.

To create a package in Python, we need to follow these three simple steps:

1. First, we create a directory and give it a package name, preferably related to its operation.
2. Then we put the all modules in it.
3. Finally we create an __init__.py file inside the directory(folder), to let Python know that the directory is a package.(This is optional from python 3.3)

Example:



Here, there is a directory named **"mypkg"** that contains three modules,Mod1.py,Mod2.py and Mod3.py. The contents of the modules are:

Mod1.py

```
def models():
        ls=["Galaxy J6","Galaxy M20","Galaxy A10"]
        print('These are the available models for SAMSUNG')
        print(ls)
```

Mod2.py

```
def models():
        ls=["RealMe 1","RealMe 2","RealMe 3","RealMe 3"]
        print('These are the available models for REALME')
        print(ls)
```

Mod3.py

```
def models():
	ls=["5","5s","6","6s","X"]
	print('These are the available models for IPHONE')
	print(ls)
```

packdemo.py

```
import mypack.Mod1 as m1
import mypack.Mod2 as m2
import mypack.Mod3 as m3
m1.models()
m2.models()
m3.models()
```

Output:

```
These are the available models for SAMSUNG
['Galaxy J6', 'Galaxy M20', 'Galaxy A10']
These are the available models for REALME
['RealMe 1', 'RealMe 2', 'RealMe 3', 'RealMe 3']
These are the available models for IPHONE
['5', '5s', '6', '6s', 'X']
```

If a file named __init__.py is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data, this data can able to use any modules under package.

Example:

__init__.py

```
print("List of Mobile Brands :")
mob=["SAMSUMG","IPHON","REALME","NOKIA"]
```

packdemo.py

```
import mypack as m
import mypack.Mod1 as m1
import mypack.Mod2 as m2
import mypack.Mod3 as m3
print(m.mob)
m1.models()
m2.models()
m3.models()
```

Output:

```
List of Mobile Brands :
['SAMSUMG', 'IPHON', 'REALME', 'NOKIA']
These are the available models for SAMSUNG
['Galaxy J6', 'Galaxy M20', 'Galaxy A10']
These are the available models for REALME
['RealMe 1', 'RealMe 2', 'RealMe 3', 'RealMe 3']
These are the available models for IPHONE
['5', '5s', '6', '6s', 'X']
```

❖ **Other Features of Modules**

1. Auto-Loaded Modules:

When the Python interpreter starts up in standard mode, some modules are loaded by the interpreter for system use.

The **sys.modules** variable consists of a dictionary of modules that the interpreter has currently loaded (in full and successfully) into the interpreter. The module names are the keys, and the location from which they were imported are the values.

The **sys.modules** variable contains a large number of loaded modules, so we will shorten the list by requesting only the module names. This is accomplished by using the dictionary's keys() method:

Example

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'readline', 'exceptions','__main__', 'posix', 'sys', '__builtin__', 'site','signal', 'UserDict',
'posixpath', 'stat']
```

2. Preventing Attribute Import:

If you do not want module attributes imported when a module is imported with "from module import *", prepend an underscore (_) to those attribute names (you do not want imported). This minimal level of data hiding does not apply if the entire module is imported or if you explicitly import a "hidden" attribute, **e.g.,** import mypack._mod4.

3. Case-Insensitive Import:

There are various operating systems with case-insensitive file systems e.g. windows, MacOS.

To import case-insensitive module, an environment variable named PYTHONCASEOK must be defined. Python will then import the first module name that is found (in a case-insensitive manner) that matches. Otherwise Python will perform its native case-sensitive module name matching and import the first matching one it finds.

4. Source Code Encoding:

Starting in Python 2.3, it is now possible to create your Python module file in a native encoding other than 7-bit ASCII(UTF-8). Of course before python 2.3 ASCII is the default, but with an additional encoding directive at the top of your Python modules, it will enable the importer to parse your modules using the specified encoding.

5. Module Execution:

There are many ways to execute a Python module: script invocation via the command-line or shell, execfile(), module import, interpreter -m option, etc.

Exceptions

❖ Exceptions in Python:

An exception can be defined as an abnormal condition in a program. It interrupts the flow of the program.

Whenever an exception occurs, the program halts the execution, and thus the further code is not executed.

In general an exception is an error that happens during execution of a program. When that error occurs, it terminates program execution.

In Python, an error can be a syntax error or an exception. Now we will see what an exception is and how it differs from a syntax error.

Syntax errors occur when the parser detects an incorrect statement.

Example: **expdemo.py**

```
a=5
b=0
print(a/b))
```

Output:

```
  File "expdemo.py", line 3
    print(a/b))
               ^
SyntaxError: invalid syntax
```

The arrow indicates where the parser ran into the syntax error. In this example, there was one more bracket. Remove it and run your code again:

Example: **expdemo.py**

```
a=5
b=0
print(a/b)
```

Output:

```
Traceback (most recent call last):

  File "expdemo.py", line 3, in <module>

    print(a/b)

ZeroDivisionError: division by zero
```

This time, the parser ran into an exception error. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.

Instead of showing the message exception error, Python details what type of exception error was encountered. In this case, it was a ZeroDivisionError.

❖ **Standard Exceptions in Python:**

Python supports various built-in exceptions, the commonly used exceptions are

- **NameError:** It occurs when a name is not found.i.e attempt to access an undeclared variable

Example:

```
a=5
c=a+b
print("Sum =",c)
```

Output:

```
Traceback (most recent call last):
  File "expdemo.py", line 2, in <module>
    c=a+b
NameError: name 'b' is not defined
```

- **ZeroDivisionError:** Occurs when a number is divided by zero.

Example:

```
a=5
b=0
print(a/b)
```

Output:

```
Traceback (most recent call last):
  File "expdemo.py", line 3, in <module>
    print(a/b)
ZeroDivisionError: division by zero
```

- **ValueError:** Occurs when an inappropriate value assigned to variable.

Example:

```
a=int(input("Enter a number : "))
b=int(input("Enter a number : "))
print("Sum =",a+b)
```

Output:

```
Enter a number : 23
Enter a number : abc
Traceback (most recent call last):
  File "expdemo.py", line 2, in <module>
    b=int(input("Enter a number : "))
ValueError: invalid literal for int() with base 10: 'abc'
```

- **IndexError:** Occurs when we request for an out-of-range index for sequence

Example:

```
ls=['c','java','python']
print("list item is :",ls[5])
```

Output:

```
Traceback (most recent call last):
  File "expdemo.py", line 2, in <module>
    print("list item is :",ls[5])
IndexError: list index out of range.
```

- **KeyError:** Occurs when we request for a non-existent dictionary key

Example:

```
dic={"name":"Madhu","location":"Hyd"}
print("The age is :",dic["age"])
```

Output:

```
Traceback (most recent call last):
  File "expdemo.py", line 2, in <module>
    print("The age is :",dic["age"])
KeyError: 'age'
```

- **IOError:** Occurs when we request for a non-existent input/output file.

Example:

```
fn=open("exam.py")
print(fn)
```

Output:

```
Traceback (most recent call last):
  File "expdemo.py", line 1, in <module>
    fn=open("exam.py")
FileNotFoundError: [IOError] No such file or directory: 'exam.py'
```

❖ Exception handling in python:

Python provides us with the way to handle the Exception so that the other part of the code can be executed without any interrupt.

For Exception handling, python uses following keywords or statements

- try
- except
- else
- finally
- raise
- assert

- **try – except statement:**

If the python program contains suspicious code that may throw the exception, we must place that code in the **try** block. 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
```

Example:

```
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
   c = a/b;
   print("a/b = %d"%c)
except Exception:
   print("can't divide by zero")
#other code:
print("other part of the program")
```

Output:

Case 1:

```
Enter a:23
Enter b:3
a/b = 7
other part of the program
```

Case 2:

```
Enter a:3
Enter b:0
can't divide by zero
other part of the program
```

Case 3:

```
Enter a:34
Enter b:abc
can't divide by zero
other part of the program
```

- **try – except-else statement:**

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

Syntax:

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

Example:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except Exception:
    print("can't divide by zero")
else:
            print("code for else block")
#other code:
print("other part of the program")
```

Output:

Case1:

```
Enter a:21
Enter b:3
a/b = 7
code for else block
other part of the program
```

Case2:

```
Enter a:3
Enter b:0
can't divide by zero
other part of the program
```

Case3:

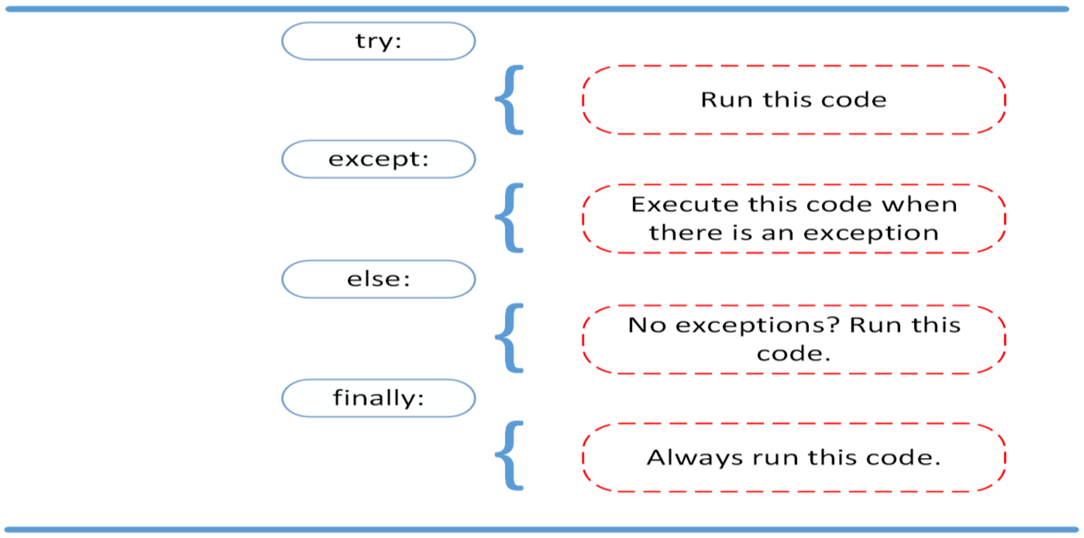
```
Enter a:3
Enter b:df
can't divide by zero
other part of the program
```

- **finally statement:**

We can use the finally block with the try block in which, we can place the important code which must be executed either try throws an exception or not.

Syntax:

```
try:
    #block of code
 except Exception1:
    #block of code
 else:
    #this code executes if no except block is executed
finally:
     # this code  will always be executed
```


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

Example:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except ZeroDivisionError:
    print("can't divide by zero")
except ValueError:
            print("Enter Numbers only")
else:
            print("code for else block")
finally:
            print("Imp code-always executes")
```

Output:

Case1:

```
Enter a:24
Enter b:4
a/b = 6
code for else block
Imp code-always executes
```

Case2:

```
Enter a:3
Enter b:0
can't divide by zero
Imp code-always executes
```

Case3:

```
Enter a:36
Enter b:abc
Enter Numbers only
Imp code-always executes
```

- **raise statement (or) Raising exceptions:**

We can use **raise** to throw an exception if a condition occurs. i.e. If you want to throw an error when a certain condition occurs we can use **raise** keyword.

Syntax:

raise Exception_class

Example:

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

Output:

Case1:

```
Enter the age : 34
the age is valid
```

Case2:

```
Enter the age : 12
The age is not valid
```

Example:

```
try:
    a = int(input("Enter a : "))
    b = int(input("Enter b : "))
    if b is 0:
        raise ArithmeticError;
    else:
        print("a/b = ",a/b)
except ValueError:
            print("The values must be numbers")
except ArithmeticError:
    print("The value of b can't be 0")
```

Output:

Case1:

```
python expdemo.py
Enter a : 34
Enter b : 12
a/b =  2.8333333333333335
```

Case2:

```
python expdemo.py
Enter a : 22
Enter b : 0
The value of b can't be 0
```

Case3:

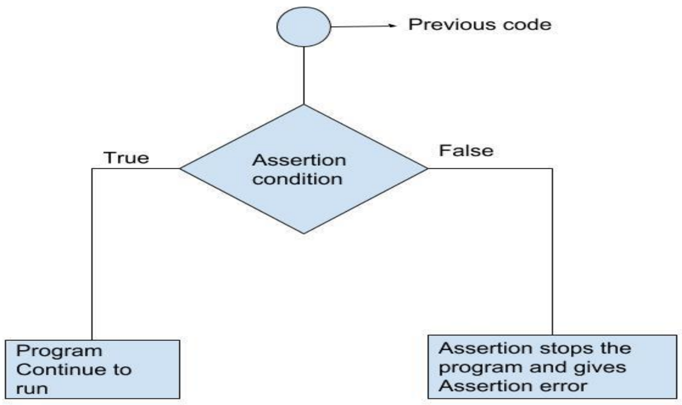
```
python expdemo.py
Enter a : 22
Enter b : sdf
The values must be numbers
```

- **assert statement (or) Assertions:**

Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.

Assertions are simply Boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and moves to the next line of code. However, if it's false, the program stops and throws an error.

It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.



assert Statement

Python has built-in **assert** statement to use assertion condition in the program. **assert** statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an **AssertionError.**

Syntax:

assert condition [, error_message]

In Python we can use assert statement in two ways as mentioned above.

1. **assert** statement has a condition and if the condition is not satisfied the program will stop and give **AssertionError.**
2. **assert** statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives **AssertionError** along with the **error message**.

Example:

```
def avg(marks):
   assert len(marks) != 0,"The List is empty."
   return sum(marks)/len(marks)
marks1 = [67,59,86,75,92]
print("The Average of Marks1:",avg(marks1))
marks2 = []
print("The Average of Marks2:",avg(marks2))
```

Output:

```
The Average of Marks1: 75.8
AssertionError: The List is empty.
```

In the above example, we have passed a non-empty list marks1 and an empty list marks2 to the avg() function. We received an output for marks1 list successfully, but after that, we got an error **AssertionError: List is empty.**

The assert condition is satisfied by the marks1 list and lets the program to continue to run. However, marks2 doesn't satisfy the condition and gives an AssertionError.

Example:

```
x = int(input("Enter x :"))
y = int(input("Enter y :"))
# It uses assert to check for 0
print ("x / y value is : ")
assert y != 0, "Divide by 0 error"
print (x / y)
```

Output:

Case1:

```
Enter x :33
Enter y :11
x / y value is :
3.0
```

Case2:

```
Enter x :33
Enter y :0
x / y value is :
AssertionError: Divide by 0 error
```

❖ **Exceptions as Strings:**

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have relationships to each other. As of python 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

Python 2.5 begins the process of deprecating string exceptions from Python forever. In 2.5, raise of string exceptions generates a warning. In 2.6, the catching of string exceptions results in a warning. Since they are rarely used and are being deprecated, we will no longer consider string exceptions.

❖ **Creating Exceptions:**

Python allow programmers to create their own exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly.

In the following example, we create custom exception class **UnderAge** that is derived from the base class **Exception**.

Using **assert** statement we can initially create our own exception. Basically **assert** statement check for a condition. If the condition is not met then it will throw AssertionError.

Syntax:

```
class User_Excep_Name(Exception):
  #code
```

Example:

```
class UnderAge(Exception):
  pass
def verify_age(age):
  if int(age) < 18:
     raise UnderAge
  else:
     print('Age: '+str(age))
# main program
try:
        verify_age(23)  # won't raise exception
        verify_age(17)  # will raise exception
except UnderAge:
        print("UnderAgeException: Less Age")
```

Output:

```
Age: 23
UnderAgeException: InEligible
```

Files

Till now, we were taking the input from the console and writing it back to the console to interact with the user. Instead of that we can able use files as input or output.

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

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 resources that are tied with the file are freed.

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

- Open a file
- Read or write (perform operation)
- Close the file

❖ File Built-in Function [open ()]:

Python has a built-in function **open()** to open a file. Which accepts two arguments, **file name** and **access mode** in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

Fileobject = **open** (file-name, access-mode)

file-name: It specifies the name of the file to be opened.

access-mode: There are following access modes for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

"r+" - Open a file for updating (reading and writing), doesn't overwrite if the file exists

"w+" - Open a file for updating (reading and writing), overwrite if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Example:
```
fileptr = open("myfile.txt","r")
if fileptr:
    print("file is opened successfully with read mode only")
```
Output:
```
file is opened successfully with read mode only
```

Example:
```
fileptr = open("myfile1.txt","x")
if fileptr:
    print("new file was created successfully")
```
Output:
```
new file was created successfully
```

❖ File Built-in Methods:

Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. For this, python provides following built–in functions, those are

- read()
- readline()
- write()
- writelines()
- tell()
- seek()
- close()

- **read():**

The **read ()** method is used to read the content from file. To read a file in Python, we must open the file in reading mode.

Syntax:
```
    Fileobject.read([size])
```

Where 'size' specifies number of bytes to be read.

Example:
```
# Opening a file with read mode
fileptr = open("myfile.txt","r")
if fileptr:
        print("file is opened successfully")
        content = fileptr.read(5)                #read 5 characters
        print(content)
        content = fileptr.read()         #read all characters
        print(content)
else:
        print("file not opened ")
```

"myfile.txt"

function open() to open a file. method read() to read a file.

Output:
```
file is opened successfully
funct
ion open() to open a file.
method read() to read a file.
```

- **readline():**

Python facilitates us to read the file line by line by using a function readline(). 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.

Syntax:

```
Fileobject.readline()
```

Example:

```
fileptr = open("myfile.txt","r")
if fileptr:
        print("file is opened successfully")
        content=fileptr.readline()
        print(content)
        content=fileptr.readline()
        print(content)
```

"myfile.txt"

```
function open() to open a file.
method read() to read a file.
```

Output:

```
function open() to open a file.
method read() to read a file.
```

- **write():**

The write () method is used to write the content into file. To write some text to a file, we need to open the file using the open method with one of the following access modes.

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.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

Syntax:

```
Fileobject.write(content)
```

Example:

```
fileptr = open("myfile.txt","a");
#appending the content to the file
fileptr.write ("Python is the modern day language.")
#closing the opened file
fileptr.close();
```

Now, we can see that the content of the file is modified.

myfile.txt:

```
function open() to open a file.
method read() to read a file.
Python is the modern day language.
```

Example:

```
fileptr = open("myfile.txt","w");
#appending the content to the file
fileptr.write("Python is the modern day language.")
#closing the opened file
fileptr.close();
```

Now, we can see that the content of the file is modified.

myfile.txt:

Python is the modern day language.

- **writelines():**

The writelines () method is used to write multiple lines of content into file. To write some lines to a file, we need to open the file using the open method with one of the following access modes.

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.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

Syntax:

```
Fileobject.writelines(list)
```

Example:

```
f = open("myfile.txt", "a")
f.writelines(["Python supports Files \n", "python supports Strings."])
f.close()
#open and read the file after the appending:
f = open("myfile.txt", "r")
print(f.read())
```

"myfile.txt"

```
function open() to open a file.
method read() to read a file.
Python is the modern day language.
```

Output:

```
function open() to open a file.
method read() to read a file.
Python is the modern day language.
Python supports Files
python supports Strings.
```

- **tell():**

The tell() method returns the current file position in a file stream. You can change the current file position with the seek() method.

Syntax:

```
Fileobject.tell()
```

Example:

```
f = open("myfile.txt", "r")
print(f.readline())
print(f.tell())
```

Output:

```
33
```

- **seek():**

The seek() method sets and returns the current file position in a file stream.

Syntax:

```
Fileobject.seek(offset)
```

"myfile.txt"

```
function open() to open a file.
method read() to read a file.
Python is the modern day language.
```

Example:

```
f = open("myfile.txt", "r")
print(f.seek(9))
print(f.read())
```

Output:

```
9
open() to open a file.
method read() to read a file.
Python is the modern day language
```

- **close():**

The close() method used to close the currently opened file.

Syntax:

```
Fileobject.close()
```

Example:

```
f = open("myfile.txt", "r")
f.close()
```

❖ **File Built-in Attributes:**

Python Supports following built-in attributes, those are

- **file.name -** returns the name of the file which is already opened.
- **file.mode -** returns the access mode of opened file.
- **file.closed -** returns true, if the file closed, otherwise false.

Example:

```
f = open ("myfile.txt", "r")
print(f.name)
print(f.mode)
print(f.closed)
f.close()
print(f.closed)
```

Output:

```
myfile.txt
r
False
True
```

❖ Standard Files:

There are generally three standard files that are made available to you when your program starts. These are **standard input** (usually the keyboard), **standard output** (the monitor or display), and **standard error** (unbuffered output to the screen).

These files are named **stdin, stdout,** and **stderr** and take their names from the C language. When we say these files are "available to you when your program starts," that means that these files are pre-opened for you, and access to these files may commence once you have their file handles.

Python makes these file handles available to you from the **sys** module. Once you **import sys**, you have access to these files as **sys.stdin, sys.stdout,** and **sys.stderr**. The print statement normally outputs to sys.stdout while the input() built-in function receives its input from sys.stdin.

❖ Command-Line Arguments

Till now, we have taken input in python using raw_input() or input() . There is another method that uses command line arguments. The command line arguments must be given whenever we want to give the input before the start of the script, while on the other hand, input() is used to get the input while the python program / script is running.

The **sys** module also provides access to any command-line arguments via **sys.argv**. Command-line arguments are those arguments given to the program in addition to the script name on invocation.

- **sys.argv** is the list of command-line arguments
- **len(sys.argv)** is the number of command-line arguments.

To use **argv**, you will first have to import it (**import sys**) The first argument, **sys.argv[0]**, is always the name of the program as it was invoked, and **sys.argv[1]** is the first argument you pass to the program. It's common that you slice the list to access the actual command line arguments.

<u>Example:</u>   **"cmdarg.py"**

```
import sys
program_name = sys.argv[0]
arguments = sys.argv[1:]
count = len(arguments)
print(program_name)
print(arguments)
print("Number of arguments ",count)
```

<u>Output:</u>

```
python cmdarg.py 45 56
cmdarg.py
['45', '56']
Number of arguments  2
```

❖ File System:

In python, the file system contains the files and directories. To handle these files and directories python supports "**os**" module. Python has the "**os"** module, which provides us with many useful methods to work with directories (and files as well).

The **os** module provides us the methods that are involved in file processing operations and directory processing like renaming, deleting, get current directory, changing directory etc.

- **Renaming the file - rename():**

The **os** module provides us the rename() method which is used to rename the specified file to a new name.

Syntax:

```
os.rename ("current-name", "new-name")
```

Example

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

- **Removing the file – remove():**

The **os** module provides us the remove() method which is used to remove the specified file.

Syntax:

```
os.remove("file-name")
```

Example

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

- **Creating the new directory – mkdir():**

The mkdir() method is used to create the directories in the current working directory.

Syntax:

```
os.mkdir("directory-name")
```

Example

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

- **Changing the current working directory – chdir():**

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

Syntax:

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

Example

```
import os;
#changing the current working directory to new
os.chdir("dir2")
```

- **Get current working directory – getpwd():**

This method returns the current working directory.

Syntax:

```
os.getcwd()
```

Example

```
import os;
#printing the current working directory
print(os.getcwd())
```

- **Deleting directory - rmdir():**

The rmdir() method is used to delete the specified directory.

Syntax:

```
os.rmdir(“directory name”)
```

Example

```
import os;
#removing the new directory
os.rmdir("dir2")
```

- **List Directories and Files – listdir():**

All files and sub directories inside a directory can be known using the listdir() method. This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

Syntax:

```
os.listdir([“path”])
```

Example:

```
import os;
#list of files and directories in current working directory
print(os.listdir())
#list of files and directories in specified path
print(os.listdir(“D:\\”))
```

Output:

```
['Animals', 'assertdemo.py', 'assertdemo.py.bak', 'cmdarg.py', 'cmdarg.py.bak',
'ex2.py', 'ex3.py', 'ex4.py', 'ex4.py.bak', 'ex5.py']

['$RECYCLE.BIN', 'Attendance', 'Calculator.class', 'Calculator.java', 'demo.html',’wtobjsam.xlsx',
'wtobjsam1.xlsx']
```

Imp Exercises

1. Python program to print number of lines, words and characters in given file.

Source Code:

```
fname = input("Enter file name: ")
num_lines = 0
num_words = 0
num_chars = 0
try:
 fp=open(fname,"r")
 for i in fp:
        # i contains each line of the file
        words = i.split()
        num_lines += 1
        num_words += len(words)
        num_chars += len(i)
 print("Lines = ",num_lines)
 print("Words = ",num_words)
 print("Characters = ",num_chars)
 fp.close()
except Exception:
 print("Enter valid filename")
```

myfile.txt

```
function open() to open a file.
method read() to read a file.
Python is the modern day language.
Python supports Files
python supports Strings.
```

Output:

```
I)      Enter file name: myfile.txt
        Lines =  5
        Words =  24
        Characters =  144
II)     Enter file name: gh
        Enter valid filename
```

2. Python Program to merge two files using command line argument.

Source Code:

```
from sys import argv
if len(argv)==4:
 try:
          fp1=open(argv[1],"r")
          fp2=open(argv[2],"r")
          fp3=open(argv[3],"w+") #w+ mode create a file if file doesn’t exist.
          for i in fp1:
                   fp3.write(i) # write content from first file to third file
          for i in fp2:
                   fp3.write(i) # write content from second file to third file
          print("Two files merged successfully")
          print("Content in ",argv[3])
          fp3.seek(0,0) # to move file point cursor to starting of file
          for i in fp3:
                   print(i,end=" ")
          fp1.close()
          fp2.close()
          fp3.close()
 except Exception:
          print("Enter valid filenames")
```

myfile.txt

```
function open() to open a file.
method read() to read a file.
Python is the modern day language.
```

myfile1.txt

```
Python supports Files
python supports Strings.
```

Output:

```
>>>python fileprg2.py myfile.txt myfile1.txt myfile2.txt
Two files merged successfully
Content in  myfile2.txt
function open() to open a file.
 method read() to read a file.
 Python is the modern day language
 Python supports Files
 python supports Strings.

>>> python fileprg2.py abc.txt myfile1.txt myfile2.txt
Enter valid filenames
```