MODULE 2

Functions

A group of related statements to perform a specific task is known as a **function**. Functions help to break the program into smaller units. Functions avoid repetition and enhance reusability. Functions provide better modularity in programming. Python provides two types of functions.

a) Built-in functions

b) User-defined functions

Functions like input(), print() etc. are examples of built-in functions. The code of the functions will be already defined. We can create our own functions to perform a particular task. These functions are called user-defined functions.

1. FUNCTION DEFINITION

The following specifies simple rules for defining a function.

- Function block or Function header begins with the keyword def followed by the function name and parentheses (()).
- Any input parameters or arguments should be placed within these parentheses can also define parameters inside these parentheses.
- The first string after the function header is called the docstring and is short to documentation string. It is used to explain in brief, what a function does. Although optional, documentation is a good programming practice.
- The code block within every function starts with a colon (:) and is indented.
- The return statement [expression] exits a function, optionally passing back expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def function_name (parameters ):
    “function_docstring"
    function suite
    return [expression]
```

Example Program

```
def sum(a,b):                                                    #Function Header
    "This function is used to find the sum of two numbers"       #Docstring
    s=a+b
    return s
```

2. FUNCTION CALLING

After defining a function, we can call the function from another function or directly from the Python prompt. The order of the parameters specified in the function definition should be preserved in function call also.

Example Program

```
#Main Program Code
a=int (input ("Enter the first number:"))
b=int (input ("Enter the second number:"))
Sums=sum(a, b)                              #Function calling
print("Sum of", a, " and ", b, " is ", Sums )
```

Output

```
Enter the first number: 4
Enter the second number:3
Sum of 4 and 3 is 7
```

All the parameters in Python are passed by reference. It means if we change a parameter which refers to within a function, the change also reflects back in the calling function. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope. Lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes. All variables in a program may not be accessible at all locations in that program. This depends on where we have declared a variable. The scope of a variable determines the portion of the program where we can access a particular identifier. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Example Program

```
def value_change (a):   #Function Header
   a=10
   print("Value inside function=", a)  #prints the value of a inside function return

#Main Program Code
a=int (input ("Enter a number:"))
value_change(a)                     #Function call
print("Value outside function=", a)        #print value of a outside function
```

Output

```
Enter a number:
4
```

```
Value inside function= 10
Value outside function= 4
```

Here in the main program a number read and stored in variable a. Even though this value is passed to the function value_change, a is assigned another value inside the function. This value is displayed inside the function. After returning from the function, it will display the value from main function. The variable a declared inside function value_change to that function and hence after returning to the main program, a will display value stored in the main program.

3. FUNCTION ARGUMENTS

We can call the function by any of the following 4 arguments.

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

3.1 Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. Consider the following example,

Example Program

```
def value_change (a): #Function Header
   a=10
   print("value inside function=", a) #prints the value of a inside function
   return
#Main Program Code
a =int(input ("Enter a number:"))
value_change () #Function calling without passing parameter
print("value outside function=", a) #prints value of a outside function
```

Output

```
Enter a number:4
Traceback (most recent call last):
        File "main.py", line 7. in <module>
                value_change () #Function calling without passing parameter
TypeError: value_change() takes exactly 1 argument (0 given)
```

The above example contains a function which requires 1 parameter. In the main program made the function is called without passing the parameter. Hence it resulted in an error.

3.2 Keyword Arguments

When we use Keyword arguments in a function call, the caller identifies the arguments by the name. This allows us to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Example Program

```
#Function definition
def studentinfo (rollno, name, course):
    print("Roll Number:",rollno)
    print("Name:", name)
    print("Course:", course)
#Function calling
#order of parameters in function call is shuffled
studentinfo(course="UG" , rollno=50 , name="Jack" )
```

Output

```
Roll Number: 50
Name: Jack
Course: UG
```

3.3 Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example shows how details arguments are invoked.

Example Program

```
#Function definition
def studentinfo (rollno, name, course="PG"):
    print("Roll Number:",rollno)
    print("Name:", name)
    print("Course:", course)
#Function calling
#order of parameters in function call is shuffled
studentinfo(course="UG" , rollno=50 , name="Jack" )
#The parameter course is omitted
studentinfo (rollno=51, name="Tom")
```

Output

```
Roll Number: 50
Name: Jack
Course: UG
Roll Number: 51
Name: Tom
Course: PG
```

In the above example, the first function call to studentinfo passes the three parameters. In the case of second function call to studentinfo, the parameter course is omitted. Hence it takes the default value "PG" given to course in the function definition.

3.4 Variable-Length Arguments

In some cases we may need to process a function for more arguments than specific while defining the function. These arguments are called variable-length arguments and not named in the function definition, unlike required and default arguments. An asterisk(*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during function call. The following shows the syntax of a function definition with variable-length arguments.

Syntax

```
def function_name ([formal_arguments,] *variable_arguments_tuple ):
    "function_docstring"
    function statements
    return [expression]
```

Example Program

```
# Function definition
def variablelengthfunction(*argument):
    print ("Result:")
    for i in argument:
        print (i)
#Function call with 1 argument
variablelengthfunction (10)
#Function call with 4 arguments
variablelengthfunction (10,30,50,80)
```

Output

```
Result:
10
```

```
Result:
10
30
50
80
```

4. ANONYMOUS FUNCTIONS (LAMBDA FUNCTIONS)

In Python, anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword, in Python anonymous functions are defined using the **lambda** keyword. Hence, anonymous functions are also called lambda function. The following are the characteristics of lambda functions.

1. Lambda functions can take any number of arguments but return only one value in the form of an expression.
2. It cannot contain multiple expressions.
3. It cannot have comments.
4. A lambda function cannot be a direct call to print because lambda requires an expression.
5. Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
6. Lambda functions are not equivalent to inline functions in C or C++.

Syntax

```
Lambda [arg1[, arg2,... ...argn]]:expression
```

Example Program

```
# Lambda Function definition
square=lambda x : x*x
# usage of lambda function
n=int (input ("Enter a number:"))
print("Square of", n,"is", square (n)) #Lambda function call
```

Output

```
Enter a number:
5
Square of 5 is 25
```

In the above example, lambda is the keyword, x shows the argument passed, and x*x is the expression to be evaluated and stored in the variable square. In the case of calculating the square of a number, only one argument is passed. More than one argument is possible for lambda functions. The following example shows a lambda function which takes two arguments and returns the sum.

Example Program

```
# Lambda Function definition
sum=lambda x,y : x+y
# Usage of lambda function
m=int (input ("Enter first number:"))
n=int (input("Enter second number:"))
print("Sum of", m, "and", n, "is", sum(m,n) ) #Lambda function call
```

Output

```
Enter first number:
10
Enter second number:
20
Sum of 10 and 20 is 30
```

Uses of lambda function

We use lambda functions when we require a nameless function for a short period of time. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

Example Program with map()

The map function in Python takes in a function and a list. The function is called with the items in the list and a new list is returned which contains items returned by that function for each item.

```
# Lambda Function to increment the items in list by 2
oldlist=[2,31,42,11,6,5,23,44]
print (oldlist)
#Usage of lambda function
newlist=list(map(lambda x: x+2,oldlist))
print("List after incrementation by 2")
print (newlist)
```

Output

```
[2, 31, 42, 11, 6, 5, 23, 44]
List after incrementation by 2
[4, 33, 44, 13, 8, 7, 25, 46]
```

4.1 filter() Function

The function filter(function, list) offers an elegant way to filter out all the elements of a list for which the function returns True. The function filter(func, lis) needs a function func as its first argument. func returns a Boolean value, i.e. either True or False. This function will be applied to every element of the list lis. Only if func returns True will the element of the list be included in the result list.

Example Program with filter()

The filter() function in Python takes in a function and a list as arguments. The function called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

```
# Lambda Function to filter out only the odd numbers from a list
oldlist=[2,31,42,11,6,5,23,44]
# Usage of lambda function
newlist=list(filter(lambda x:(x%2),oldlist))
print(oldlist)
print(newlist)
```

Output

```
[2, 31, 42, 11, 6, 5, 23, 44]
[31, 11, 5, 23]
```

4.2 reduce() Function

The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.

If seq=[s_1,s_2,s_3,…,s_n], calling reduce(func, seq) works like this:

At first the first two elements of seq will be applied to func, i.e. func(s_1,s_2) The list on which reduce() works looks now like this: [func(s_1,s_2), s_3,…,s_n]

In the next step func will be applied on the previous result and the third element of the list, i.e.

[func(func(s_1,s_2), s_3),…. ,s_n]

Continue like this until just one element is left and return this element as the result of reduce().

This is a really useful function for performing some computation on a list and returning the result. The following example illustrates the use of reduce() function, that computes the product of a list of integers.

Example Program with reduce()

```
import functools
list=[1,2,3,4]
product=functools.reduce (lambda x, y: x* y,list)
print (list)
print("Product=", product)
```

Output

```
[1, 2, 3, 4]
Product = 24
```

5. RECURSIVE FUNCTIONS

Recursion is the process of defining something in terms of itself. A function can call other functions. It is possible for a function to call itself. This is known as recursion. The following example shows a recursive function to find the factorial of a number.

Example Program

```
def recursion_fact (x):
   "This is a recursive function to find the factorial of an integer"
   if x == 1:
      return 1
   else:
      return (x * recursion_fact (x-1)) #Recursive calling

num = int(input ("Enter a number: "))
if num >= 1:
   print("The factorial of", num, "is", recursion_fact (num))
```

Output

```
Enter a number:
5
The factorial of 5 is 120
```

In the above example recursion_fact() is a recursive function as it calls itself. When we call this function with a positive integer, it will recursively call itself by decreasing the number. Each function call multiplies the number with the factorial of number-1 until the number is equal to one. This recursive call can be explained in the following steps. Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely. We must avoid infinite recursion.

```
recursion_fact (5)                  # 1st call with 5
5* recursion_fact (4)               #2nd call with 4
5*4*recursion_fact (3)              # 3rd call with 3
5* 4 * 3 * recursion_fact (2)       #4th call with 2
5* 4* 3* 2* recursion_fact (1)      # 5th call with 1
```

```
5* 4*3* 2*1                    # return from 5th call as number=1
5 *4*3* 2                      # return from 4th call
5 * 4 * 6                      # return from 3rd call
5*24                           # return from 2nd call
120                            #return from 1st call
```

6. FUNCTION WITH MORE THAN ONE RETURN VALUE

Python has a strong mechanism of returning more than one value at a time. This is very flexible when the function needs to return more than one value. Instead of writing separate functions for returning individual values, we can return all the values within same function. The following shows an example for function returning more than one value.

Example Program

```
def calc(a,b):
    "Python Function Returning more than one value "
    sum=a+b
    diff=a-b
    prod=a*b
    quotient=a/b
    return sum, diff, prod, quotient

a=int (input("Enter first number:"))
b=int (input("Enter second number:"))
s, d, p, q=calc(a,b)
print("sum=",s)
print("Difference=",d)
print("Product=",p)
print("Quotient=",q)
```

Output

```
Enter first number:
10
Enter second number:
5
sum= 15
Difference= 5
```

```
Product= 50
Quotient= 2.0
```

Modules

Modules refer to **a file containing Python statements and definitions**. A module is a Python object with arbitrarily named attributes that we can bind and reference. **We use modules to break down large programs into small manageable and organized files**. Further, modules **provide reusability of code**. A module can **define functions, classes and variables**. A module can also include **runnable code**. Python has a lot of standard modules (built-in modules) available. A full list of Python standard modules is available in the **Lib directory** inside the location where you installed Python. 5.1

1. BUILT-IN MODULES

Examples:

- math
- string
- statistics
- array
- email
- html

2. CREATING MODULES

We can define our most used functions in a module and import it, instead of copying their definitions into different programs. A file containing Python code, for eg: **prime.py** is called a module and its module name would be prime. The Python code for a module named **test** normally resides in a file named test.py.

Let us create a module for finding the sum of two numbers. The following code creates a module in Python. Type the code and save it as **test.py**.

Example

```
# Python Module example
“ This program adds two numbers and return the result"
def sum (a, b) :
    result = a + b
    return result
```

Here we have a function sum() inside a module named test. The function takes in two numbers and returns their sum.

3. IMPORT STATEMENT

We can use any Python source file as a module by executing an import statement in some other Python source file. User-defined modules can be imported the same way as we import built-in modules.

We use the **import** keyword to do this. The import has the following syntax.

import modulel1 [, module2, ... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module test.py, you need to put the following command at the top of the script.

Example

```
    import test
    print (test.sum (2,3))
```

Output

```
    5
```

When the above code in the example is executed, we get the result 5.

A module is loaded only once, regardless of the number of times it is imported prevents the module execution from happening over and over again if multiple imports occur.

3.1 import with renaming

We can import built-in or user-defined modules with an alias name. The following code shows how the built-in module math can be imported using an alias name.

Example Program

```
    import math as m
    print("The value of pi is", m.pi)
```

Output

```
    The value of pi is 3.141592653589793
```

We have renamed the math module as m. This can save typing time in some cases. It is important to note that the name math is not recognized in our scope. Hence, math.pi is invalid, m.pi is the correct implementation.

3.2 from...import statement

We can import specific names from a module without importing the module as a whole. The module math contains a lot of built-in functions. But if we want to import only the pi function, the **from...import** statement can be used. The following example illustrates importing function, pi from math module.

Example Program

```
    from math import pi
    print ("The value of pi is :", pi)
```

Output

```
    The value of pi is : 3.14159265359
```

We have imported only the pi function from the math module hence no need to use the dot operator as given in example 3.1. We can also import multiple attributes from the module using from...import statement. The following example illustrates the import of pi as well as sqrt() from the math module using from... import statement.

Example Program

```
        from math import pi, sqrt
        print("The value of pi is :", pi)
        print("The square root of 4 is:", sqrt(4))
```

Output

```
        The value of pi is: 3.14159265359
        The square root of 4 is: 2.0
```

3.3 import all names

We can import all names(definitions) from a module using the following construct. The following example shows how all the definitions from the math module can be imported. This makes all names except those beginning with an underscore, visible in our scope.

Example Program

```
        from math import *
        print("The value of pi is :", pi)
        print("The square root of 4 is:",sqrt (4) )
```

Output

```
        The value of pi is: 3.14159265359
        The square root of 4 is: 2.0
```

Here all functions in the module math are imported. But we have used only pi and sqrt. Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also reduces the readability of our code.

4. LOCATING MODULES

When we import a module, the Python interpreter searches for the module in the following sequence.

1. The current directory.
2. If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
3. If the above two mentioned fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/. The module search path is stored in the system module **sys** as the sys.path variable contains the current directory, PYTHONPATH, and the installation dependent default.

4.1 PYTHONPATH variable

The PYTHONPATH is an **environment variable**, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

The following shows a typical PYTHONPATH from a Windows operating system.

```
set PYTHONPATH=C:\python37-32\Lib
```

And here is a typical PYTHONPATH from a UNIX system.

```
set PYTHONPATH=/usr/local/lib/python
```

5. NAMESPACES AND SCOPE

Variables are names (identifiers) that map to objects. A namespace is **a dictionary of variable names (keys) and their corresponding objects (values)**. A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, **the local variable shadows the global variable**.

Each function has its own local namespace. Python makes certain guesses on whether variables are local or global. It assumes that any variable assigned a value in a function local. Therefore, in order to assign a value to a global variable within a function, you must first use the **global** statement. The statement global variable_name tells Python variable_name is a global variable. Once a variable is declared global, Python stops searching the local namespace for that variable. Consider the following example.

Example Program

```
#Program to illustrate local and global namespace
a=10
def Add ():
    a=a+1
    print (a)
 Add()
print(a)
```

Output

```
Traceback (most recent call last):
    File "main.py", line 5, in <module>
        Add()
    File "main.py", line 3, in Add
        a=a+1
UnboundLocalError:  local variable 'a' referenced before assignment
```

When we run the above program, the output is listed above with errors. The reason is that we are trying to add 1 to a before a is assigned a value. This is because even though a is assigned a value 10, its scope is outside the function Add(). Further Python always searches for variables in the local namespace. Since a is not assigned value inside the function, it resulted in an error. The below program is re-written using global variable concept.

Example Program

```
        #Program to illustrate local and global namespace
        a=10
        print( a)
        def Add():
                global a
                a=20
                a=a+1
                print(a)
        Add()
        print(a)
```

Output

```
        10
        21
        21
```

In the above example, a is assigned a value 10 and it is printed. Inside the function, a is declared global and it is incremented by one. The result is 21. Even after, exiting the function, when we try to print the value of a, it is printing it as 21 since a is declared a global variable.

6. THE dir() FUNCTION

The dir( ) built-in function returns **a sorted list of strings containing the names defined by a module**. The list contains the names of **all the modules, variables and functions that are defined in a module**. The following example illustrates the use of dir() function for the built-in module called math.

Example Program

```
        import math
        c=dir (math)
        print(c)
```

Output

```
['__doc__ , '__file__' , '__name__', '__package__', 'acos', 'acosh', asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expml', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', '1dexp', 'lgamma', 'log', 'log10', 'loglp', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh, trunc']
```

In the above output,

doc is present in every namespace. Initially it will be None, we can store documentation here.

file is the folder name from which the module was loaded.

name is the name of the current module (minus the .py extension).

_package__ is the name of package.(if the module is a package). When the module is not a package, _package_ should be set to the empty string.

The rest shows the built-in functions available in the math module.

Another example illustrates the use of dir() function along with user-defined modules.

Example Program

```
import test
c=dir (test)
print (c)
```

Output

```
['_builtins_'. '_doc_', '_file_', '_name_', '_package__'. 'sum')
```

The output contains built-in strings and we have defined only one function called sum in that module. builtins contain Python's built-in error handling functions.

7. THE reload() FUNCTION

When the module is imported into a script, the code in the top-level portion of a module is executed only once. Therefore, if you want to reexecute the top-level code in a module, you can use the reload() function. The reload() function imports a previously imported module again. The syntax of the reload() function is as follows.

reload (module_name)

Here, module_name is the name of the module we want to reload. For example, to reload test module, do the following

```
reload (test)
```

8. PACKAGES IN PYTHON

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and so on. As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modulus in different packages. This makes a project easy to manage and organize. Similar, as a directory can contain subdirectories and files, a Python package can have subpackages and modulus. A directory must contain a file named **__init__.py** in order for python to consider it as a package. This file can be left empty but we generally place the initialization code for the package in this file.

8.1 Important module from a package

Consider a file **Asample.py** available in **pack** directory. This file has following line of source code.

```
def a:
    print(“This is from A”)
```

Similarly we have two more files **Bsample.py** and **Csample.py** available in pack directory with the following codes.

Bsample.py

```
        def b:
                print(“This is from B”)
```

Csample.py

```
        def c:
                print(“This is from C”)
```

Now we have three files in the same directory **pack.** To make all of our functions available when we are importing **pack,** we need to put explicit import statements in __init__.py as follows.

```
        from Asample import a
        from Bsample import b
        from Csample import c
```

After adding these lines to __init__.py, we have all of these functions available when we import the **pack** package. The following codes shows how the package **pack** can be imported.

```
        import pack
        pack.a()
        pack.b()
        pack.c()
```

When the above code is executed, it gives the following result.

```
        This is from A
        This is from B
        This is from C
```

9 DATE AND TIME MODULES

Python provides various modules like time, calendar, datetime

9.1 The time module

The time module contains a lot of functions to process time. Time intervals are represented as floating-point numbers in seconds. The time since January 1, 1970, 12:00 a.m is available and is termed as an epoch(era). The following example shows how many seconds have elapsed since the epoch.

Example Program

```
        import time
        seconds = time.time()
        print ("Number of seconds since 12:00am, January 1, 1970:", seconds)
```

Output

```
Number of seconds since 12:00am, January 1, 1970: 1631202130.8693042
```

The float representation is useful when storing or comparing dates, but not as useful for producing human readable representations. For printing the current time ctime() may be more useful. The following shows an example for ctime() method.

Example Program

```
import time print("The time is:", time.ctime())
later = time.time() + 30
print ("30 secs from now:", time.ctime (later))
```

Output

```
The time is: Sat Nov 21 10:10:08 2015
30 secs from now: Sat Nov 21 10:10:38 2015
```

9.1.1 struct_time Structure

The time module defines struct_time for holding date and time values with components broken are easy to access. The struct_time has the following attributes listed in Table 5.2.

Attribute	Values
tm_year	Current year
tm_mon	1 to 12
tm_mday	1 to 31
tm_hour	0 to 23
tm_min	0 to 59
tm_sec	0 to 61(60 and 61 are leap seconds)
tm_yday	0 to 6(0 is Monday)
tm_wday	1 to 366(Julian Day)
tm_isdst	-1,0,1 (-1 if DST)

Daylight saving time (DST) or summer time is the practice of advancing clocks during summer months by one hour so that in the evening, daylight is experienced an hour longer, while sacrificing normal sunrise times.

Example Program

```
import time
print ("gmtime: ", time.gmtime())
print("localtime:", time.localtime())
t = time.localtime ()
print("Current Year:", t.tm_year)
```

```
print ("Current Month:",t.tm_mon)
print ("Current Day:", t.tm_mday)
print ("Day of month:", t.tm_mday)
print ("Day of week:", t.tm_wday)
print ("Day of year:", t.tm_yday)
print ("DST:", t.tm_isdst)
```

Output

```
gmtime: time.struct_time(tm_year=2021, tm_mon=9, tm_mday=10, tm_hour=2, tm_min=30,
tm_sec=9, tm_wday=4, tm_yday=253, tm_isdst=0)
localtime: time.struct_time(tm_year=2021, tm_mon=9, tm_mday=10, tm_hour=2, tm_min=30,
tm_sec=9, tm_wday=4, tm_yday=253, tm_isdst=0)
Current Year: 2021
Current Month: 9
Current Day: 10
Day of month: 10
Day of week: 4
Day of year: 253
DST: 0
```

gmtime(**) returns the current time in **Universal Time Clock. localtime() returns current time with the **current time zone** applied. mktime() takes a struct_time and convert it to the floating point representation.

9.1.2 Parsing and Formatting Time

The two functions **strptime()** and **strftime()** convert between struct_time and String representations of time values.

The following example converts the current time from a string, to a struct_time instance, and back to a string.

Example Program

```
import time
now = time.ctime()
print (now)
parsed = time.strptime (now)
print (parsed)
print (time.strftime ("%a %b %d %H:%M:%S %Y", parsed))
```

Output

```
Fri Sep 10 02:39:23 2021
time.struct_time(tm_year=2021, tm_mon=9, tm_mday=10, tm_hour=2, tm_min=39, tm_sec=23, tm_wday=4, tm_yday=253, tm_isdst=-1)
Fri Sep 10 02:39:23 2021
```

- %a - abbreviated weekday name
- %b - abbreviated month name
- %d - day of the month (01 to 31)
- %H - hour, using a 24-hour clock (00 to 23)
- %M - minute
- %S - second
- %Y - year including the century

9.2 The calendar Module

The calendar module has many built-in functions for printing and formatting the output. By default, calendar takes Monday as the first day of the week and Sunday as the last one. **calendar.TextCalendar:** It can be used to generate plain text calendars.

The following example shows formatted text calendar.

Example Program

```
import calendar
c = calendar.TextCalendar ()
c.prmonth(2021, 9)
```

Output

```
            September 2021
Mo    Tu    We    Th    Fr    Sa    Su
             1    2     3     4     5
 6    7     8     9     10    11    12
13    14    15    16    17    18    19
20    21    22    23    24    25    26
27    28    29    30
```

Example Program

```
import calendar
print(calendar.TextCalendar () .formatyear(2021))
```

Output

```
(calendar 2021)
```

9.2.1 Built-in functions

1. calendar.firstweekday( ) - Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, first week day is 0(Monday).
2. calendar.setfirstweekday(weekday) - Sets the first day of each week to weekday with the parameter passed. Weekday codes are 0 (Monday) to 6 (Sunday).
3. calendar.isleap(year)- returns True if year is a leap year, False otherwise.
4. calendar.leapdays(y1,y2)- Returns the total number of leap days in the years within range(y1,y2).

Example Program

```
import calendar
print("Default first weekday:", calendar. firstweekday ( ))
calendar.setfirstweekday(calendar.SUNDAY)
print("New first weekday:", calendar. firstweekday())
print ("Leap Year:", calendar.isleap (2014))
print("Leap days between 2010 and 2015:", calendar.leapdays (2010,2015))
```

Output

```
Default first weekday: 0
New first weekday: 6
Leap Year: False
Leap days between 2010 and 2015: 1
```

5. calendar.calendar(year,w=2,1=1,c=6) - Returns a multiline string with a calendar for year formatted into three columns separated by c spaces. w is the width in characters of each date. 1 is the number of lines for each week.

Example Program

```
import calendar
print (calendar.calendar (2021,2,1,6))
```

Output(2021 Calender)

9.3 The datetime Module

The datetime module includes functions and classes for doing date and time parsing formatting and arithmetic. It contains functions and classes for working with dates and times, separately and together.

9.3.1 The time Class

Time values are represented with the time class. Times have attributes for hour, minute second and microsecond. They can also include time zone information. A time instance only holds values of time and not a date associated with the time.

Example Program

```
import datetime
```

```
t = datetime.time (1, 2, 3)
print (t)
print('hour :', t.hour)
print('minute:', t.minute)
print('second:', t.second)
print('microsecond:', t.microsecond)
print('tzinfo:', t.tzinfo)
```

Output

```
01:02:03
hour : 1
minute: 2
second: 3
microsecond: 0
tzinfo: None
```

9.3.2 The date Class

Calendar date values are represented with the date class. Instances have attributes for month and day. It is easy to create a date representing today's date using the today()

Example Program

```
import datetime
today = datetime.date.today()
print (today)
print('ctime:', today.ctime())
print ('Year:', today.year)
print ( 'Mon :', today.month)
print('Day :', today.day)
```

Output

```
2021-09-10
ctime: Fri Sep 10 00:00:00 2021
Year: 2021
Mon : 9
Day : 10
```

9.3.3 timedeltas

We can use datetime to perform basic **arithmetic** on date values via the **timedelta** class.

Example Program

```
import datetime
print ( "microseconds:", datetime.timedelta(microseconds=1))
print("milliseconds:", datetime.timedelta (milliseconds=1))
print("seconds:", datetime.timedelta (seconds=1))
print ("minutes: ", datetime.timedelta (minutes=1))
print("hours: ", datetime.timedelta (hours=1))
print("days:", datetime.timedelta (days=1))
print("weeks:", datetime.timedelta (weeks=1))
```

Output

```
microseconds: 0:00:00.000001
milliseconds: 0:00:00.001000
seconds : 0:00:01
minutes : 0:01:00
hours: 1:00:00
days: 1 day, 0:00:00
weeks: 7 days, 0:00:00
```

Date math uses the standard arithmetic operators. This example with date objects illustrates using timedelta objects to compute new dates, and subtracting date instances to produce timedeltas (including a negative delta value).

Example Program

```
import datetime
today = datetime.date.today()
print('Today :', today)
one_day = datetime.timedelta (days=1)
print('One day :', one_day)
yesterday = today - one_day
print("Yesterday:", yesterday)
tomorrow = today+ one_day
print('Tomorrow :',tomorrow)
print ('tomorrow - yesterday:', tomorrow - yesterday)
print ( 'yesterday - tomorrow:', yesterday - tomorrow)
```

Output

```
Today : 2021-09-13
One day : 1 day, 0:00:00
```

```
Yesterday: 2021-09-12
Tomorrow : 2021-09-14
tomorrow - yesterday: 2 days, 0:00:00
yesterday - tomorrow: -2 days, 0:00:00
```

Both date and time values can be compared using the standard operators to determine which is earlier or later.

Example Program

```
import time
print('Times:')
t1=datetime.time(12,55,0)
print('\tt1:',t1)
t2=datetime.time(13,5,0)
print('\tt2:', t2)
print ('\ttl < t2:', t1 < t2)
print('Dates:')
d1=datetime.date.today()
print('\td1:',d1)
d2=datetime.date.today()+datetime.timedelta(days=1)
print('\td2:',d2)
print('\td1>d2:',d1>d2)
```

Output

```
Times:
        t1: 12:55:00
        t2: 13:05:00
        t1 < t2: True
Dates:
        d1: 2015-11-23
        d2: 2015-11-24
        d1 > d2: False
```

In addition, we can use combine() a date instance and time instance to create a datetime instance

Example Program

```
import datetime
t = datetime.time (1, 2, 3)
print('t :', t)
```

```
d = datetime.date.today()
print('d :', d)
dt = datetime.datetime.combine(d,t)
print ('dt:', dt)
```

Output

```
t : 01:02:03
d : 2021-09-13
dt: 2021-09-13 01:02:03
```

File Handling

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). Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

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.**

a. Open a file

b. Read or write (perform operation)

c. Close the file

1. OPENING A FILE

Before we can read or write a file, we have to open it using Python's built-in **open()** function. This function returns **a file object**, also called a **handle**, as it is **used to read or modify the file accordingly**.

We can specify the mode while read 'r', write 'w' or append ‘a’ to the file. We also specify if we want to open the file in text mode or binary mode. The **default is reading in text mode**. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files. The following shows the syntax for opening a file.

Syntax

file_object = open(filename [, accessmode] [,buffering])

The following gives the explanation of the parameters for opening a ne

f**ilename**: The filename argument is a string value that contains the **name of file** that we want to access.

accessmode: The accessmode determines the mode in which the file has to be opened, i.e., read, write, append, etc. This is optional parameter and the **default file access mode is read (r).**

buffering: If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. This is also an optional parameter.

Example

```
    f = open("abc.txt", 'r')                    # open file in current directory
    f = open("C:/Python33/sample.txt", 'r')     # specifying full path
```

In the above example, the file **abc.txt** is opened in the current working directory read mode and **sample.txt** is opened in read mode inside C:\Python33 folder.

1.1 Modes for Opening a File

1. **'r'** - Opens a file for reading only. The **file pointer** is placed **at the beginning of the file**. This is the default mode.

2. **'rb**'- Opens a file for **reading** only in **binary** format. The **file pointer** is placed at the **beginning of the file.**

3. **'r+'-** Opens a file for **both reading and writing**. The **file pointer** is placed at the **beginning of the file**.

4. **'rb+'** - Opens a file for both **reading and writing in binary format**. The **file pointer** is placed at the **beginning of the file**.

5. **'w'-** Opens a file for **writing only**. **Overwrites the file if the file exists**. If the file do**es not exist, creates a new file for writing.**

6. **'wb**' - Opens a file for **writing only in binary format**. **Overwrites the file if the file exists. If the file does not exist, creates a new file for writing,**

7. '**w+**' - Opens a file for **both writing and reading**. **Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.**

8. **"wb+'** - Opens a file for both **writing and reading in binary format**. **Overwrites existing file if the file exists. If the file does not exist, creates a new file for read and writing.**

9. **'a'** - Opens a file for **appending**. The **file pointer** is at the **end of the file if the exists**. That is, the file is in the append mode. If the file **does not exist, it create new file for writing**.

10 '**ab**'- Opens a file for **appending in binary format**. The **file pointer is at the end of the file if the file exists** that is, the file is in the append mode. If the file **does not exist, it creates a new file for writing**.

11. **'a+'** -Opens ante for **both appending and reading**. The **file pointer** is at the **end of the** file if the file exists. The file opens in the append mode. If the file **does not exist creates a new file for reading and writing.**

12. **‘ab+’** - Opens a file for **both appending and reading in binary** format. The **file pointer** at the **end of the file if the file exist**s. The file opens in the append mode. If the file s not **exist, it creates a new** file for reading and writing.

13. **‘x’** - Open a file for **exclusive creation**. If the file **already exists, the operation fails**.(If you open a file in mode `x`, the file is created and opened for writing – but *only* if it doesn’t already exist. Otherwise you get a FileExistsError.)

Since the version 3.x, Python has made a clear distinction between str (text) and bytes Unlike other languages, the character 'a' does not imply the number 97 until it is using ASCII (or other equivalent encodings). Hence, when working with files in text mode it is recommended to specify the encoding type. Files are stored in bytes in the disk, we need to decode them into str when we read into Python. Similarly, encoding is performed while writing texts to the file.

The default **encoding is platform dependent**. In **windows,** it is '**cp1252**' but **'utf-8'** in **Linux.** Hence, we must not rely on the default encoding otherwise, our code will behave differently in different platforms. Thus, this is the preferred way to open a file for reading in text mode.

f = open("abc.txt", mode = 'r', encoding = 'utf-8')

1.2 Attributes of file object

Once a file is opened, we have one file object, we can get various information related to that file. The following shows **various attributes for the file object**.

1. **file.closed** - Returns True if file is closed, False otherwise.
2. **file. mode** - Returns access mode with which file was opened.
3. **file.name -** Returns name of the file.
4. **file.softspace** - Returns 0 if space is explicitly required with print, or 1 otherwise. (`softspace` is a read-write attribute that is used **internally by the `print` statement to keep track of its own state**

The following example shows the working of the above described attributes of file object.

Example Program

```
    #Demo of file object attribute in Python
    fo = open("abc.txt", "w")
    print("Name of the file:", fo.name)
    print("Closed or not.", fo.closed)
    print("Opening mode :", fo.mode)
    print("Softspace flag:", fo.softspace)
```

Output

```
    Name of the file: abc.txt
    Closed or not: False
```

```
Opening mode: W
Softspace flag: 0
```

2 CLOSING A FILE

When we have finished the operations to a file, we need to properly close it. Python has a **garbage collector** to clean up unreferenced objects. But we must not rely on it to file. Closing a file will free up the resources that were tied with the file and is done, **close()** method. The following shows the syntax for closing a file.

Syntax

fileObject.close()

Example Program

```
# Open a file for writing in binary format
fo = open("bin.txt", "wb")
print ("Name of the file: ", fo.name)
 fo.close()                    # Close opened file
print("File Closed")
```

Output

```
Name of the file: bin.txt
File Closed
```

3 WRITING TO A FILE

In order to write into a file we need to open it in write **'w', append 'a'** or **exclusive creation 'x'** mode. We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data will be erased.

Writing a string or sequence of bytes (for binary files) is done using **write()** method. This method **returns** the **number of characters written to the file**. The write() method does **not add a newline character ('\n') to the end of the string**. The following shows the syntax of write() method.

Syntax

fileObject.write (string)

Here, passed parameter string is the content to be written into the opened file.

Example Program

```
# Open a file in writing mode
fo = open("test.txt", "w")
fo.write("Programming with Python is Fun. \nLet's try Python!\n");
fo.close()                    # Close opened file
print("File", fo.name, "closed.")
```

Output

```
File test.txt closed.
```

In the above program, we have opened a file test.txt in writing mode. The string to be written is passed to the write() method.

3.1 with Statement

Python's **with** statement is handy when we **have two related operations** which we would like **to execute as a pair**, with a block of code in between. The classic example is **opening a file, manipulating the** file, then closing it.

Example Program

```
with open('output.txt', 'w') as f:
    f.write('Hello Python!')
```

The above with statement will **automatically close the file after the nested block of code**. The advantage of using a with statement is that it is **guaranteed to close the file** no matter the nested block exits. If an exception occurs before the end of the block, it will close the before the exception is caught by an outer exception handler. If the nested block were to contain a return statement, or a continue or break statement, the with statement would automatically close the file in those cases, also.

4 READING FROM A FILE

To read the content of a file, we must open the file in **reading mode**. The **read(**) method reads a string from an open file. It is important to note that Python strings can have binary data, apart from text data. The following gives the syntax for reading from a file.

Syntax

```
fileObject.read([size])
```

Here, the passed parameter size is the **number of bytes to be read from the opened file.** This method starts reading from the **beginning of the file** and if size is missing, then it tries to read as much as possible, maybe until the **end of file**. The following example shows how to read from the file test.txt which we have already created.

Example Program

```
fo = open("test.txt", "r")                # Open a file in read mode
str = fo.read (11)
print("String Read is : ", str)
fo.close()                                # Close opened
```

```
print("File", fo.nam, "is closed.")
```

Output

```
String Read is: Programming
File test.txt is closed.
```

5 FILE METHODS

1. **file.close**()
2. **file.fileno**() - Returns an **integer number** (file descriptor) **of the file**.

Example Program

```
# Example for fileno () in File Handling
fo=open ("abc.txt","w")
str="Python Programming"
fo.write(str)
print ("The file number is:",fo.fileno ( )
fo.close()
```

Output

```
The file number is: 3
```

3. file.seek(offset,from=SEEK_SET)- Change the file position to offset bytes, in reference to from (start=0, current=1, end=2).

4. file.tell( ) - Returns the **current file pointer location**.

Example Program

```
#Demo program for seek and tell
# Creating a file
str="Python Programming is fun"
fo=open("test.txt", "w")
fo.write(str)
fo.close()
fo = open("test.txt", "r+")
s = fo.read(10)
print ("Read String is : ", s)
pos = fo.tell()                    # Check current position
print ("Current file position : ", pos)
pos = fo.seek (0, 0)               # Reposition pointer at the beginning once again
```

```
s = fo.read(10)
 print ("Again read String is : ", s)
fo.close()                                # Close opened file
```

Output

```
Read String is: Python Pro
Current file position :10
Again read String is : Python Pro
```

5. file.write()

6. file.read()

7. file.readline()-The method readline **reads one entire line from the file**. A trailing newline character is kept in the string. An empty string is returned only when EOF is encountered immediately.

8. **file.truncate([size])** - The method truncate() truncates the file's size. If the optional size argument is present, the file is truncated to (atmost) that size.

Example Program

```
fo = open("test.txt", "r")                # Open a file
line = fo.readline ()
print("Read Line:", line)
fo.truncate()                             #Now truncate remaining file.
line = fo.readline ()                     # Try to read file now
print("Read Line: ", line)
fo.close()                                # close opened file
```

Output

```
Read Line: Python Programming is fun
Read Line:
```

9. **file.readlines([sizeint])** - The method readlines() **reads until EOF using** r**eadline**() and turns a list containing the lines. sizeint is an optional argument and if it is present, instead of reading up to EOF, whole lines approximate to sizeint bytes are read. An empty string is returned only when EOF is encountered immediately.

10. **file.writelines(sequence)** - The method writelines() **writes a sequence of strings to the file**. The sequence can be any iterable object containing strings, typically a list of strings. There is **no return value**. Example Program

```
fo = open("Demo.txt", "w")                # Open a file in write mode
seq=["First Line\n", "Second Line\n", "Third Line\n", "Fourth Line\n". "Fifth Line\n"]
```

```
fo.writelines (seq)
fo.close()                              #close the file
fo = open("Demo.txt", "r")              #Open the file in read mode
line = fo.readlines ()
print("readlines ():", line)
line= fo.readlines (2)
print("readlines (2):", line)
fo.close()                              # Close opened file
```

Output

```
readlines (): [ First Line\n', 'Second Line\n', 'Third Line\n', 'Fourth Line\n', 'Fifth Line\n']
readlines (2): [ ]
```

6 RENAMING A FILE

The **os module** Python provides methods that help to perform file-processing such as **renaming and deleting files**. To use this os module, we need to import it first and then we can call any related functions. To rename an existing file, we can use the **rename( )** method. This method takes two arguments, current filename and new filename. The shows the syntax for rename () method.

Syntax

os.rename (current_file_name, new_file_name)

Example Program

```
import os
os.rename ( "test.txt", "Newtest.txt" )     # Rename a file from test.txt to Newtest.txt
print("File renamed.")
```

Output

```
File renamed.
```

7 DELETING A FILE

We can delete a file by using the **remove()** method available in **os** module. This method receives one argument which is the filename to be deleted. The following gives the syntax for remove() method.

Syntax

os.remove (filename)

Example Program

```
import os
os.remove ("Newtest.txt")               # Delete a file
```

```
    print("File Deleted.")
```

Output

```
    File Deleted.
```

8 DIRECTORIES IN PYTHON

All files will be saved in various directories, and Python has efficient methods for handling directories and files. The **os module** has several methods that help to create, remove, change directories.

8.1 mkdir() method

The mkdir() method of the os module is used to **create directories** in the current directory. We need to supply an argument to this method which contains the name of the directory to be created. The following shows the syntax of mkdir() method.

Syntax

```
    os.mkdir("dirname")
```

Example

```
    import os
    os.mkdir("Fruits")      #create a directory "Fruits"
```

The above example creates a directory named Fruits.

8.2 chdir() method

To **change the current directory**, we can use the chdir() method. The chdir() method takes an argument, which is the name of the directory that we want to make the current directory. The following shows the syntax of chdir() method.

Syntax

```
    os.chdir("dirname")
```

Example

```
    import os
    os.chdir("/home/abc")                    # Change a directory
```

The above example goes to the directory /home/abc.

8.3 getcwd() method

The getcwd() method **displays the current working directory**. The following shows the syntax of getcwd() method.

getcwd-get current working directory

Syntax

```
    os.getcwd()
```

Example

```
    import os
```

```
os.getcwd()
                                    #Displays the location of current directory
```

8.4 rmdir() method

The rmdir ( ) method **deletes the directory**, which is passed as an argument in the method.

rmdir-remove directory

Syntax

```
os.rmdir("dirname")
```

Example

```
import os
os. rmdir ("Fruits")                    # Removes the directory
```

It is intended to give the exact path of a directory. Otherwise it will search working directory.

6.9 SOLVED LAB EXERCISES

1. **Write a program to copy a text file to another file**

Program

```
file1=input("Enter the source file to be copied:")
file2=input("Enter the destination file name:")
fr=open(file1, "r")                     # Open the file to be copied in read mode
fw=open (file2, "w")                    # Open a file to be copied in write mode
for line in fr.readlines ():
        fw.write(line)
fr.close()                  #close the file
fw.close()                  #close the file
print("1 File Copied")
```

Output

```
1 File Copied
```

2. **Program to count the number of lines in a file.**

Program

```
fr=open("Demo.txt", "r")
 countlines=0
for line in fr.readlines () :
        countlines=countlines+1
print("Number of Lines:", countlines)
fr.close()
```

Output

```
Number of Lines: 2
```

3. Write a program to count the frequencies of each word from a file.

Program

```
fr=open("Demo.txt","r")
wordcount={}
for word in  fr.read().split():
    if word not in wordcount:
        wordcount [word] = 1
    else:
        wordcount [word] += 1
for k, v in wordcount.items ():
    print (k, v)
fr.close()
```

Output

```
Handling 1
easy 1
Python 2
is 2
Programming 1
File 1
fun 1
```

4. **Write a program to append a file with the contents of another file.**

Program

```
filel=input("Enter the file to be opened for appending:")
file2=input("Enter the file name to be appended :")
fa=open (file1, "a")
fr=open(file2, "r")
for line in fr.readlines ():
    fa.write(line)
fr.close()
fa.close()
print("1 File Appended")
```

Output

```
Enter the file to be opened for appending: TruncateDemocopy.py
 Enter the file name to be appended: TruncateDemo.py
1 File Appended
```

Exception Handling

An exception is an abnormal condition that is caused by a **runtime error** in the program. It disturbs the normal flow of the program. An example for an exception is **division by 0**. When a Python script encounters a situation that it cannot deal with, it raises an Exception. **An exception is a Python object that represents an error.**

If the exception object is not caught and handled properly, the interpreter will display an error message. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display appropriate messages for taking corrective actions. This task is known as **exception handling**.

1 BUILT-IN EXCEPTIONS

Python has a collection of built-in exception classes. If the runtime error belongs to any of the pre-defined built-in exception, it will throw the object of the appropriate exception. The following Table 1 gives a detailed explanation of built-in exceptions and the situation in which they are invoked.

Table .1 Built-in Exceptions

Exception Name	**When it is raised**
AttributeError	Raised in case of failure of attribute reference or assignment.
EnvironmentError	Base class for all Exceptions that occur outside the Python environment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
Exception	Base class for all Exceptions
ImportError	**Raised when an import statement fails.**
IndentationError	**Raised when indentation is not specified properly.**
IndexError	Raised when an index is not found in a sequence.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist
NameError	Raised when an identifier is not found in the local or global namespace.
OSError	Raised for operating system-related errors.
Overflow Error	Raised when a calculation exceeds maximum limit for a numeric type.
RuntimeError	**Raised when a generated error does not fall into any category.**

SyntaxError	Raised when there is an error in Python syntax.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
ZeroDivison Error	**Raised when division or modulo by zero takes place for all numeric types**

2 HANDLING EXCEPTIONS

Python uses a keyword **try** to prepare a block of code that is **likely to cause an error and throw an exception**. An **except** block is defined **which catches the exception throw try block and handles it**. The try block can have one or more statements that could generate an exception. If anyone statement generates an exception, then the remaining statements in the block are skipped and execution jumps to the except block that is placed next to the try block.

The except block can have more than one statement and if the except parameter matches with the type of the exception object, then the exception is caught and statements in the block will be executed. Every try block should be followed by atleast one except statement.

2.1 try….. except

The following gives the **syntax of try....except** statement.

Syntax

```
try:
    suite
except Exception1:
    Exceptions_suite        #Executes when Exception1 occurs.
except Exception2:
    Exception2_suite        #Executes when Exception2 occurs.
else:
    else_suite              #Executes if there is no Exception in the try block.
```

When an exception occurs inside a try block, it will go to the corresponding except block. If no exception is raised inside a try block then after the try block, the statements in the else block is executed. The else block can also be considered a place to put the code that does not raise any exception.

Example Program 1

```
#Exception Handling
try:
    a=int (input("First Number :"))
    b=int (input (Second Number:"))
```

```
        result=a/b
        print("Result=", result)
    except ZeroDivisionError:
        print("Division by zero")
    else:
        print("Successful Division")
```

Output

```
    First Number:10
    Second Number: 0
    Division by zero
```

In the above example, the second number is 0. Since division by zero is not possible, an exception is thrown and the execution goes to the except block. All the rest of the statements are bypassed in the try block. Hence the output is displayed as Division by zero.

Example Program 2

```
    #Exception Handling
    try:
        a=int (input("First Number:”))
        b=int (input("Second Number:”))
        result=a/b
        print("Result=", result)
    except ZeroDivisionError:
        print("Division by zero”)
    else:
        print("Successful Division")
```

Output

```
    First Number:20
    Second Number:10
    Result= 2
    Successful Division
```

In the above example, exception is not thrown and hence all the statements in the try block is executed. After executing the try block, the control passes on to the else block. Hence the print statement in the else block is executed.

2.2 except clause with no Exception

We can use except statement with no exceptions. This kind of try-except statement **catches all the Exceptions**. Since it catches all the exceptions, it will not help the programmer exactly identify what is the cause for the error occurred. Hence it is not considered as a good programming practice. The following shows the syntax for except clause with no exception

Syntax

```
try:
        suite
except:
        Exception_suite        #Executes when whatever Exception occurs.
else:
        else suite             #Executes if there is no Exception in the try block.
```

Example Program

```
#Exception Handling
try:
        a=int (input ("First Number:”))
        b=int(input ("Second Number:”))
        result=a/b
        print("Result=", result)
except:
        print("Error Occured")
else:
        print("Successful Division”)
```

Output

```
First Number: 2
Second Number: 0
Error Occurred
```

2.3 except clause with multiple Exceptions

This is used when **we want to give multiple exceptions in one except statement**. The following shows the syntax of the except clause with multiple exceptions.

Syntax

```
try:
        suite
```

```
except (Exception1[, Exception2 [,... ExceptionN]]]):
        Exception_suite                  #Executes when whatever Exception specified occurs.
else:
        else_suite                       #Executes if there is no Exception in the try block.
```

Example Program

```
#Exception Handling
try:
        a=int (input ("First Number:”))
        b=int(input ("Second Number:”))
        result=a/b
        print("Result=", result)
except(ZeroivisionError, TypeError):
        print("Error Occurred")
else:
        print("Successful Division”)
```

Output

```
First Number: 10
Second Number: 0
Error Occurred
```

2.4 try...finally

A finally block can be used with a try block. **The code placed in the finally executed no matter exception is caused or caught**. **We cannot use except clause and finally clause together with a try block. It is also not possible to use else clause and finally together with a try block**. The following gives the syntax for a try...finally block.

Syntax

```
try:
        suite
finally:
        finally suite           #Executed always after the try block.
```

Example Program 1

```
#Exception Handling
try:
        a=int (input("First Number:"))
        b=int (input ("Second Number:"))
```

```
                result=a/b
                print("Result=", result)
        finally:
                print("Executed Always")
```

Output

```
        First Number:20
        Second Number: 0
        Executed Always
        Traceback (most recent call last):
                File "main.py", line 5, in <module>
                        result=a/b
        ZeroDivisionError: integer division or modulo by zero
```

In the above example, an error occurred in the try block. It is caught by the Python's error handling mechanism. But the statement in the finally block is executed even though an error has occurred.

Example Program 2

```
        #Exception Handling
        try:
                a=int (input("First Number:"))
                b=int (input ("Second Number:"))
                result=a/b
                print("Result=", result)
        finally:
                print("Executed Always")
```

Output

```
        First Number: 10
        Second Number: 5
        Result= 2
        Executed Always
```

In the above example there was no exception thrown. Hence after the try block, the finally block is executed.

3 EXCEPTIONS WITH ARGUMENTS

It is possible to have exception with arguments. **An argument is a value that gives additional information about the problem**. This variable receives the value of the exception mostly **containing the**

cause of the exception. The variable can receive a single value or multiple values in the form of a **tuple**. This tuple usually **contains the error string, the error and an error location**. The syntax of exception with arguments is as follows.

```
    try:
        suite
    except ExceptionType,  argument:
        except_suite           #Executed when exception occurs.
```

Example Program

```
    def display(a):
        try:
            return int (a)
        except ValueError, argument1:
            print ("Argument does not contain numbers", argument1)
    display("a")                                          #Function Call
```

Output

```
    Argument does not contain numbers invalid literal for int() with base 10: “a”
```

In the above example, the function display is **called with string arguments**. But the actual function is defined with **integer parameter**. Hence an exception is thrown and it is caught by the except clause.

4 RAISING AN EXCEPTION

We have discussed about raising built-in exceptions. It is also possible to **define a new exception and raise it if need. This is done using the raise statement**. The following shows the syntax of raise statement. An exception can be a string, a class or an object.

Synatx

```
    raise [Exception [, argument [, traceback]]]
```

Here, exception is the type of exception (for example, IOError) and argument is a value for the exception argument. This argument is optional. The exception argument is None does not supply any argument. The argument, traceback, is also optional. If this is used, then the traceback object is used for the exception.

In order to catch the exception defined using the raise statement, we need to us except clause which is already discussed. The following code shows how an exception defined using the raise statement can be used.

need to use the

Example Program

```
    # Raising an Exception
```

```
a=int (input ("Enter the parameter value:"))
try:
   if a<=0:
      raise ValueError("Not a Positive Integer")
except ValueError:
   print (ValueError)
else:
   print("Positive Integer=", a)
```

Output

```
Enter the parameter value:-9
Not a Positive Integer
```

5 USER-DEFINED EXCEPTION

Python also allows us to create our own exceptions by deriving classes from the standard built-in exceptions. In the try block, the user-defined exception is raised and caught in the except block. This is useful when we need to provide more specific information when an exception is caught. The following shows an example for user-defined exception.

Example Program

```
# 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:
  num = int(input("Enter a number: "))
  if num ==0:
   raise Dividebyzero
except Dividebyzero:
  print("Input value is zero, try again!")
  print()
```

Output

```
Enter a number: 0
Input value is zero, try again!
```

[The **pass statement** is used as a placeholder for future code. When the **pass statement** is executed, nothing happens.]

6. ASSERTIONS IN PYTHON

An assertion is **a checking in Python that can be turned on or off while testing a program**. **In assertion, an expression is tested and if the result is False, an exception is raised**. Assertions are done using the **assert** statement. The application of assertion is **to check for a valid input or for a valid output**. The following shows the syntax for assert statement.

When Python interpreter encounters an assertion statement, it evaluates the accompanying expression. If the expression is evaluated to False, Python raises an AssertionError exception. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Example Program

```
#Python Assertions
def sum(a,b):
   sum=a+b
   assert(sum>0),"Too low value"
   return (sum)
a=int(input("Enter first no: "))
b=int(input("Enter second no: "))
print(sum(a,b))
```

Output

```
First Number:-8
Second Number:-2
Traceback (most recent call last):
        File "main.py", line 8, in <module>
                print sum(a,b)
        File "main.py", line 4, in sum
                assert (sum>0), "Too low value"
AssertionError: Too low value
```

7. REVIEW QUESTIONS

1. Briefly explain exception handling in Python.

2. What are built-in exceptions?
3. Explain try... except statement in Python.
4. Explain except clause with no exception.
5. Explain except clause with multiple exceptions.
6. What is try... finally in Python?
7. Explain exception with arguments.
8. How to raise an exception in Python using the raise statement?
9. Write short notes on user-defined exceptions.
10. What are assertions in Python?