PYTHON PROGRAMMING & DATA SCIENCE

PYTHON PROGRAMMING & DATA SCIENCE

UNIT 2

Strings:

Creating strings and basic operations on strings,string testing methods.

String Data Type

- The most commonly used object in any project and in any programming language is String only.

What is String?

Strings are arrays of bytes representing Unicode characters.

Or

Group of characters

String Data Type

Creating a String

- Any sequence of characters within either single quotes or double quotes or even triple quotes is considered as a String.

Example:

s= ‘Cse ‘

s=“python“

S=“’’cse’’’

N=“456”

String Data Type

Note:

- In most of other languges like C, C++,Java, a single character with in single quotes is treated as char data type value. But in Python there in no char data type. Hence it is treated as String only.

Example:

```
>>> ch='a'
>>> type(ch)
<class 'str'>
```

String Data Type

- We can define multi-line String literals by using triple single or double quotes.

Eg:

```
s= ''' cse branch
                pbr vits
        kavali'''
```

String Data Type

Accessing characters in Python

characters of a string can be accessed in 2 ways . Those are :

1. By using index
2. By using slice operator

1.By using index:

- Python supports both +ve and -ve index.
- +ve index means left to right(Forward direction)
- -ve index means right to left(Backward direction)

String Data Type

Example:

s='python'

p	y	t	h	o	n
0	1	2	3	4	5
-6	**-5**	**-4**	**-3**	**-2**	**-1**

>>> s[0]

'p'

>>> s[4]

'o'

>>> s[-1]

'n'

String Data Type

- While accessing an index out of the range will cause an **IndexError**.
- Only Integers are allowed to be passed as an index, float or other types will cause a **TypeError**.
- **Example:**

s='python'

>>> s[10]

IndexError: string index out of range

String Data Type

2. Accessing characters by using slice operator:

Syntax:

s[bEginindex:endindex:step]

- bEginindex: From where we have to consider slice(substring)
- endindex: We have to terminate the slice(substring) at endindex-1
- step: incremented value

Note:

- If we are not specifying bEgin index then it will consider from bEginning of the string.
- If we are not specifying end index then it will consider up to end of the string
- The default value for step is 1

String Data Type

Accessing characters by using slice operator:

Example:

```
>>> s="Learning Python is very very easy!!!"
>>> s[1:7:1]
'earnin'
>>> s[1:7]
'earnin'
>>> s[1:7:2]
'eri'
>>> s[:7]
'Learnin‘
```

String Data Type

Accessing characters by using slice operator:

Example:

```
>>> s="Learning Python is very very easy!!!“
>>> s[7:]
'g Python is very very easy!!!'
>>> s[::]
'Learning Python is very very easy!!!'
>>> s[:]
'Learning Python is very very easy!!!‘
>>> s[::-1]
'!!!ysae yrev yrev si nohtyP gninraeL'
```

String Data Type

Behaviour of slice operator:

s[bEgin:end:step]

step value can be either +ve or –ve

- **if +ve then it should be forward direction(left to right) and consider from bEgin to end-1.**
- **if -ve then it should be backward direction(right to left) and consider from bEgin to end+1**

String Data Type

Behaviour of slice operator:

In forward direction:

default value for bEgin: 0

default value for end: length of string

default value for step: +1

In backward direction:

default value for bEgin: -1

default value for end: -(length of string+1)

Note: Either forward or backward direction, we can take both +ve and -ve values for bEgin and end index.

String Data Type

Mathematical Operators for String:

The following mathematical operators can be applied on Strings.

1. + operator for concatenation
2. * operator for repetition

Example:

```
print(“cse"+“branch")
csebranch
print("durga"*2)
durgadurga
```

Note:

1. To use + operator for Strings, compulsory both arguments should be str type
2. To use * operator for Strings, compulsory one argument should be str and other argument should be int

String Data Type

Built-in String methods for Strings:

SNO	Method Name	Description
1	capitalize()	Capitalizes first letter of string.
2	center(width, fillchar)	Returns a space-padded string with the original string centered to a total of width columns.
3	count(str, beg= 0,end=len(string))	Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	decode(encoding='UTF-8',errors='strict')	Decodes the string using the codec registered for encoding. Encoding defaults to the default string encoding.
5	encode(encoding='UTF-8',errors='strict')	Returns encoded string version of string; on error, default is to raise a Value Error unless errors is given with 'ignore' or 'replace'.
6	endswith(suffix, beg=0, end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	expandtabs(tabsize=8)	Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.

String Data Type

Built-in String methods for Strings:

SNO	Method Name	Description
8	find(str, beg=0 end=len(string))	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	index(str, beg=0, end=len(string))	Same as find(), but raises an exception if str not found.
10	isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	isdigit()	Returns true if string contains only digits and false otherwise.
13	islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.

String Data Type

Built-in String methods for Strings:

SNO	Method Name	Description
14	isnumeric()	Returns true if a unicode string contains only numeric characters and false otherwise.
15	isspace()	Returns true if string contains only whitespace characters and false otherwise.
16	istitle()	Returns true if string is properly "titlecased" and false otherwise.
17	isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	join(seq)	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	len(string)	Returns the length of the string.
20	ljust(width[, fillchar])	Returns a space-padded string with the original string left-justified to a total of width columns.
21	lower()	Converts all uppercase letters in string to lowercase.
22	lstrip()	Removes all leading whitespace in string.

String Data Type

Built-in String methods for Strings:

SNO	Method Name	Description
23	maketrans()	Returns a translation table to be used in translates function.
24	max(str)	Returns the max alphabetical character from the string str.
25	min(str)	Returns min alphabetical character from the string str.
26	replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	rfind(str, beg=0,end=len(string))	Same as find(), but search backwards in string.
28	rindex(str, beg=0, end=len(string))	Same as index(), but search backwards in string.
29	rjust(width,[, fillchar])	Returns a space-padded string with the original string right-justified to a total of width columns.
30	rstrip()	Removes all trailing whitespace of string.

String Data Type

Built-in String methods for Strings:

SNO	Method Name	Description
31	split(str="", num=string.count(str))	Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	startswith(str, beg=0,end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
33	upper()	Converts lowercase letters in string to uppercase.
34	zfill (width)	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
35	isdecimal()	Returns true if a unicode string contains only decimal characters and false otherwise.

String Data Type

Built-in String methods for Strings:

Examples:

```
>>>str1="welcome"
>>>print "Capitalize function---",str1.capitalize()
Capitalize function--- Welcome

>>>print str1.center(15,"*")
****welcome****
>>> print "length is",len(str1)
length is 7
```

String Data Type

Built-in String methods for Strings:

Examples:

```
>>> print "count function---",str1.count('e',0,len(str1))
count function--- 2

>>> print "endswith function---
",str1.endswith('me',0,len(str1))

endswith function--- True

>>> print "startswith function---
",str1.startswith('me',0,len(str1))
startswith function--- False
```

String Data Type

Built-in String methods for Strings:

Examples:

```
>>> print "find function---",str1.find('e',0,len(str1))
find function--- 1

str2="welcome2021"
>>>print "isalnum function---",str2.isalnum()
isalnum function--- True

>>> print "isalpha function---",str2.isalpha()
isalpha function--- False
```

String Data Type

Built-in String methods for Strings:

Examples:

```
>>> print "islower function---",str2.islower()
islower function--- True

>>> print "isupper function---",str2.isupper()
isupper function--- False

>>> str3="    welcome"
print "lstrip function---",str3.lstrip()
lstrip function--- welcome
```

String Data Type

Built-in String methods for Strings:

Examples:

```
str4="77777777cse777777";
>>> print "lstrip function---",str4.lstrip('7')
lstrip function--- cse777777
>>> print "rstrip function---",str4.rstrip('7')
rstrip function--- 77777777cse
>>> print "strip function---",str4.strip('7')
strip function--- cse
str5="welcome to java"
print "replace function---",str5.replace("java","python")
replace function--- welcome to python
```

Strings formating

- **Example 1: Formatting string using % operator**

```
x = 'language'
print(“Python %s powerful %s around"%(‘very',x))
```

- **‘%s’** is used to inject strings similarly **‘%d’** for integers, **‘%f’** for floating-point values, **‘%b’** for binary format. For all formats, conversion methods visit the official documentation.

- print('Joe stood up and %s to the crowd.' %'spoke')
- print('There are %d dogs.' %4)

```
    variable = 12
string = "Variable as integer = %d \n\
Variable as float = %f" %(variable, variable)
print (string)
```

Output:

```
ariable as integer = 12
Variable as float = 12.000000
```

Formatting string using format() method

- Format() method was introduced with Python3 for handling complex string formatting more efficiently. Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces { } into a string and calling the str.format(). The value we wish to put into the placeholders and concatenate with the string passed as parameters into the format function.
- **Syntax:** 'String here {} then also {}'.format('something1','something2')

- print(‘We all are {}.'.format('equal'))
- print('{2} {1} {0}'.format('directions',
- 'the', 'Read'))

Functions

Function:

- A function is a self contained program segment (or) a sub program which carries out some specific, well defined task.

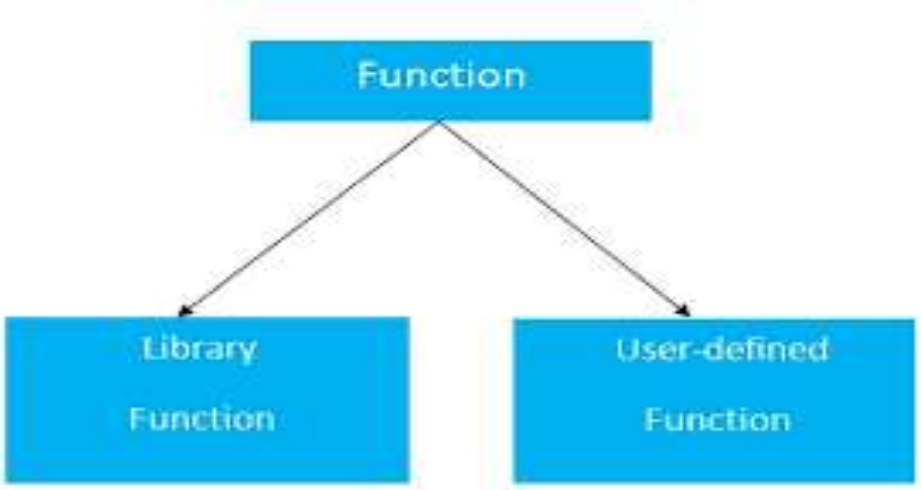
Advantages:

1. Avoid repetition of codes.
2. Increases program readability.
3. Divide a complex problem into simpler ones.
4. Reduces chances of error.
5. Create, Modify and debugging a program becomes easier by using function.
6. It can be executed any number of times.

Functions

Types Of Function:

- Python functions can be classified into two categories,
  1. Library functions
  2. User-defined functions



Functions

Library functions :

- Functions that are defined by the compiler are known as library functions.

Examples:

1. input()
2. print()
3. len()
4. max()
5. Count() , etc

Functions

User defined functions:

- User-defined functions are those functions which are defined by the user at the time of writing program.
- These functions are made for code reusability, saving time & space.
- For creating user defined function user can follow the two things.
    1. Defining a Function.
    2. Calling a Function.

Defining functions

- We can define functions to provide the required functionality.
- simple rules to define a function in Python are:

1. Function blocks begin with the keyword **def** followed by the function name and parentheses ( ).
2. Any input parameters or arguments should be placed within these parentheses.
3. The statements that form the body of the function start at the next line, and must be indented.
4. The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*.

Defining functions

5. The code block within every function starts with a colon (:) and is indented.
6. The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

➢By default, parameters have a positional behavior and need to inform them in the same order that they were defined.

Calling a Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, we can execute it by calling it from another function or directly from the Python prompt.

Example 1:

```
def my_function():
  print("Hello from a function")
my_function()
```

Output:

Hello from a function

Function

Example 2:

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
   print str
   return;
# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same  function")
```

Output:

```
I'm first call to user
defined function!
Again second call to the
same function
```

Defining functions

- The *execution* of a function introduces a new symbol table used for the local variables of the function.

- Thus, global variables cannot be directly assigned a value within a function, although they may be referenced.

- The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value*.

Defining functions

➢When a function calls another function, a new local symbol table is created for that call.

```
def fib(n):# write Fibonacci series up to n
...    """Print a Fibonacci series up to n."""
...    a, b = 0, 1
...    while a < n:
...        print a,
...        a, b = b, a+b
...
# Now call the function we just defined:
... fib(2000)
```

Output:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

Defining functions

Example:

fib(0)
print fib(0)

Output:

None

Types of function arguments

- It is also possible to define functions with a variable number of arguments.
- There are 4 types.

1. **Default Argument Values**
2. **Keyword Arguments**
3. **Arbitrary Argument Lists(**Variable-length arguments)
4. **Required arguments**

1. Default Argument Values:

➢A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

Example:

```
def  fun1(a,b=50):
  print(a,b)

fun1(20)
fun1(20,100)
```

Output:

```
20   50
20   100
```

2.Keyword Arguments:

- Keyword arguments are related to the function calls.
- When we use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows 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:

```
def  fun1(a, b, c):
    print(a,b,c)
fun1(c = 20, b = 15, a = 10)
```

Output:

```
10   15   20
```

3. Arbitrary Argument Lists

- If we do not know how many arguments that will be passed in function, add a * before the parameter name in the function definition.
- the function will receive a tuple of arguments, and can access the items accordingly.

Example:

```
def my_function(*kids):
  print("The youngest child is " + kids[1])

my_function("bhaskar", "vijay", "Raju","Pinky")
```

Output:

The youngest child is vijay

4. 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.

Example:

```
def  fun1( a,b ):
       print(a,b)
a=5
b=10
fun1(a,b)
fun1(a)
```

Output:

```
5 10
error
```

```
def  fun1( a,b ):
      print(a,b)
a=5
b=10
fun1(a,b)
# Calling function and passing only one argument
print( "Passing only one argument" )
try:
   fun1( b )
except:
    print( "Function needs two positional arguments" )
```

More on Defining Function

The Anonymous Functions

- These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. We can use the *lambda* keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression

More on Defining Function

- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Syntax

lambda [arg1 [,arg2,.....argn]]:expression

More on Defining Function

Example:

```
#!/usr/bin/python
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

Output:

```
Value of total :  30
Value of total :  40
```

Functions

Functions are first class objects

- **First-class** objects in a language are handled uniformly throughout. They may be stored in data structures, passed as arguments, or used in control structures.
- A programming language is said to support first-class functions if it treats functions as first-class objects. Python supports the concept of First Class functions.

Properties of first-class functions

1. It is an instance of an Object type
2. Functions can be stored as variable
3. Pass First Class Function as argument of some other functions
4. Return Functions from other function
5. Store Functions in lists, sets or some other data structures.

Functions

- Python functions are first-class objects.
- Functions as First Class Objects means that functions can be passed around into lists and used as arguments for other functions.
- Functions in Python can be used as an object. In Python, a function can be assigned as variable. To assign it as variable, the function will not be called. So parentheses '()' will not be there.

Functions

Example 1:

```
# program to illustrate functions can be treated as objects
def display(text):
    return text.upper()
print display('hello')
x = display
print x('hello')
```

Output:

```
HELLO
HELLO
```

Functions

Example 2:

```
def cube(x):
    return x*x*x
res = cube(5)
print(res)
my_cube = cube #The my_cube is same as the cube method
res = my_cube(5)
print(res)
```

Output:

125
125

Functions

Example 3: # create a 2 functions

```
def square(x):
   return x*x
 def cube(x):
   return x*x*x
# create a dictionary of functions
funcs = {
    'square': square,
    'cube': cube,
}
x = 2
for func in sorted(funcs):
    print func, funcs[func](x)
```

Output:

```
cube 8
square 4
```

Functions

Example 4:

```
# functions can be passed as argument of another functions.
def cube(x):
  return x*x*x
def my_map(method, argument_list):
  result = list()
  for item in argument_list:
    result.append(method(item))
  return result
my_list = my_map(cube, [1, 2, 3, 4, 5, 6, 7, 8])
#Pass the function as argument
print(my_list)
```

Output:

[1, 8, 27, 64, 125, 216, 343, 512]

Functions

Example 5:

```
# return one function from another function.
def create_logger(message):
  def log():
    print('Log Message: ' + message)
  return log #Return a function
my_logger = create_logger('Hello World')
my_logger()
```

Output:

Log Message: Hello World

Functions

Returning multiple values from a function

- Python functions can return multiple values.
- These values can be stored in variables directly.
- A function is not restricted to return a variable, it can return zero, one, two or more values.

Example:

```
def getPerson():
    name = "bhaskar"
    age = 35
    country = "India"
    return name,age,country
name,age,country = getPerson()
print(name), print(age), print(country)
```

Output:

```
bhaskar
35
India
```

Functions

Returning multiple values from a function

➢For returning multiple values from a function, we can return tuple, list or dictionary object as per our requirement.

Method 1: Using tuple

```
def func(x):
   y0 = x+ 1
   y1 = x * 3
   y2 = y0 ** 3
   return (y0, y1, y2)
a,b,c = func(3)
print "a=",a
print "b=",b
print "c=",c
```

Output:

```
a= 4
b= 9
c= 64
```

Functions

Returning multiple values from a function

Method 2: Using a dictionary

Example 1:

```
def fun():
    d = dict();
    d['str'] = "bhaskar"
    d['x']   = 20
    return d
d = fun()
print(d)
```

Output:

```
{'x': 20, 'str': 'bhaskar'}
```

Functions

Returning multiple values from a function

Method 2: Using a dictionary

Example 2:

```
def func(x):
   y0 = x+ 1
   y1 = x * 3
   y2 = y0 ** 3
   return {'y0': y0, 'y1': y1, 'y2': y2}
a=func(3)
print a
```

Output:

```
{'y1': 9, 'y0': 4, 'y2': 64}
```

Functions

Returning multiple values from a function

Method 3: Using a list

```
def func(x):
   result = [x + 1]
   result.append(x * 3)
   result.append(x ** 3)
   return result
  a=func(3)
print a
```

Output:

[4, 9, 27]

Functions

formal and actual arguments

Actual Parameter	Formal Parameter
The arguments that are passed in a function call are called actual arguments.	The formal arguments are the parameters/arguments in a function declaration.
Data type not required. But data type should match with corresponding data type of formal parameters.	Data types needs to be mentioned.
Actual parameters are the parameters which specify when to call the Functions or Procedures.	A formal parameter is a parameter which specify when we define the function.
The actual parameters are passed by the calling function.	The formal parameters are in the called function.

Ex:
Void main()
{ Int a,b;
a= sum(4,5); //function call
}
// 4,5 are actual parameter
	Ex:
	Int sum(int a, int b)
	{ Int s;
	s=a+b;
	return(s);
	}
	//a&b are formal parameter

Functions

Recursive functions

➢A function calls itself is known as recursion and the function is called as recursive function.

Syntax:

```
def func():
            <--
        |
        | (recursive call)
        |
    →
  func()
```

Functions

Recursive functions

Advantages of using recursion

1. A complicated function can be split down into smaller sub-problems utilizing recursion.
2. Sequence creation is simpler through recursion than utilizing any nested iteration.
3. Recursive functions render the code look simple and effective.

Disadvantages of using recursion

1. A lot of memory and time is taken through recursive calls which makes it expensive for use.
2. Recursive functions are challenging to debug.

Functions

Example 1:

```
# Program to print factorial of a number  recursively.
 def recursive_factorial(n):
   if n == 1:
      return n
   else:
      return n * recursive_factorial(n-1)
 num =input("enter a number")
  if num < 0:
   print("Invalid input ! Please enter a positive number.")
elif num == 0:
   print("Factorial of number 0 is 1")
else:
   print("Factorial of number", num, "=", recursive_factorial(num))
```

Result:

Input:

Enter a number:

6

Output:

Factorial of number

6 = 720

Functions

Example 2:

```
# Program to print the fibonacci series upto n_terms
 def recursive_fibonacci(n):
   if n <= 1:
      return n
   else:
      return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))
 n_terms = 10
if n_terms <= 0:
   print("Invalid input ! Please input a positive value")
else:
   print("Fibonacci series:")
 for i in range(n_terms):
      print(recursive_fibonacci(i))
```

Output:

Fibonacci series:

0 1 1 2 3 5 8

13 21 34

ERRORS AND EXCEPTIONS

- There are Two types of Errors in python.

1. Syntax errors
2. Logical errors (Exceptions)

- Errors are the problems in a program due to which the program will stop the execution.
- exceptions are raised when the some internal events occur which changes the normal flow of the program.

ERRORS AND EXCEPTIONS

Syntax Errors:

- Syntax errors, also known as parsing errors.
- When the proper syntax of the language is not followed then syntax error is thrown.

Example:

```
a = 8
b = 10
c = a b
```

Output:

```
File "<ipython-input-8-3b3ffcedf995>", line 3
c = a b
      ^
SyntaxError: invalid syntax
```

ERRORS AND EXCEPTIONS

SyntaxError: invalid syntax

- The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected.

- The error is caused by (or at least detected at) the token *preceding* the arrow

- File name and line number are printed so we know where to look in case the input came from a script.

ERRORS AND EXCEPTIONS

SyntaxError: invalid syntax

Example:

```
amount = 10000
if(amount>2999)
        print("You are eligible to purchase ")
```

Output:

```
  File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

➢It returns a syntax error message because after if statement a colon : is missing. We can fix this by writing the correct syntax.

ERRORS AND EXCEPTIONS

Exceptions:

➢Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

➢Errors detected during execution are called *exceptions.*

➢Python provides two very important features to handle any unexpected error in Python programs and to add debugging capabilities in them.

Those are

1. **Exception Handling**
2. **Assertions**

➢ Most exceptions are not handled by programs, however, and result in error messages as shown here:

ERRORS AND EXCEPTIONS

Examples:

1. 10 * (1/0)

   Traceback (most recent call last):
   File "<stdin>", line 1, in ?
   ZeroDivisionError: integer division or modulo by zero

2. 4 + spam*3

   Traceback (most recent call last):
   File "<stdin>", line 1, in ?
   NameError: name 'spam' is not defined

3. '2' + 2

   Traceback (most recent call last):
   File "<stdin>", line 1, in ?
   TypeError: cannot concatenate 'str' and 'int' objects

➢The last line of the error message indicates what happened.

ERRORS AND EXCEPTIONS

- Exceptions come in different types, and the type is printed as part of the message:

- The exception types in the example are :

ZeroDivisionError,
NameError
TypeError.
IOError

ERRORS AND EXCEPTIONS

Handling Exceptions:

- It is possible to write programs that handle selected exceptions.
- We can handle error by Try/Except/Finally method.
- we write unsafe code in the try, fall back code in except and final code in else block.

Handling Exceptions:

➢Syntax

```
try:
    do our operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
finally:
   The finally block is a place to put any code that must execute
```

ERRORS AND EXCEPTIONS

Handling Exceptions:

Here are few important points about the above-mentioned syntax −

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- we can also provide a generic except clause, which handles any exception.
- After the except clause(s) include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The finally block is a place to put any code that must execute

ERRORS AND EXCEPTIONS

Handling Exceptions:

Example 1:

```
#!/usr/bin/python
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully“
```

Output:

```
Error: can't find file or read data
```

Handling Exceptions:

Example 2:

```
while True:
   try:
   x = int(raw_input("Please enter a number: "))
   break
   except ValueError:
            print "Oops!       That was no valid number. Try again..."
```

➢which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program that a user-generated interruption is signalled by raising the KeyboardInterrupt exception.

ERRORS AND EXCEPTIONS

Handling Exceptions:

Argument of an Exception

- An exception can have an *argument*, which is a value that gives additional information about the problem.

Syntax:

try: You do your operations here; .....

except *ExceptionType, Argument*:

You can print value of Argument here...

Handling Exceptions:

Example:

```
def temp_convert(var):
  try:
       return int(var)
  except ValueError as Argument:
    print  ("The argument does not contain numbers\n", Argument)

temp_convert("xyz")
```

The argument does not contain numbers

invalid literal for int() with base 10: 'xyz‘

ERRORS AND EXCEPTIONS

Raising Exception:

- The raise statement allows the programmer to force a specified exception to occur.
- When we want to code for limitation of certain condition then we can raise an exception.

Raising Exception:

For example:

```
amount = 1500
try:
        if amount > 1000:
          raise ValueError("please add money in your account")
        else:
          print("You are eligible to purchase .")
        # if false then raise the value error
except ValueError as e:
                print(e)
```

```
def division(a, b):
        try:
            return a / b
        except ZeroDivisionError as ex:
         raise ValueError('b must not zero')
print(division(5,2))
Print(division(5,0))
```

```
def division(a, b):
    try:
        return a / b
    except ZeroDivisionError as ex:
        print('Logging exception:', str(ex))
        raise
print(division(5,2))
Print(division(5,0))
```

```
def division(a, b):
    try:
        return a / b
    except:   raise

print(division(5,2))
Print(division(5,0))
```

Example 1: division by zero

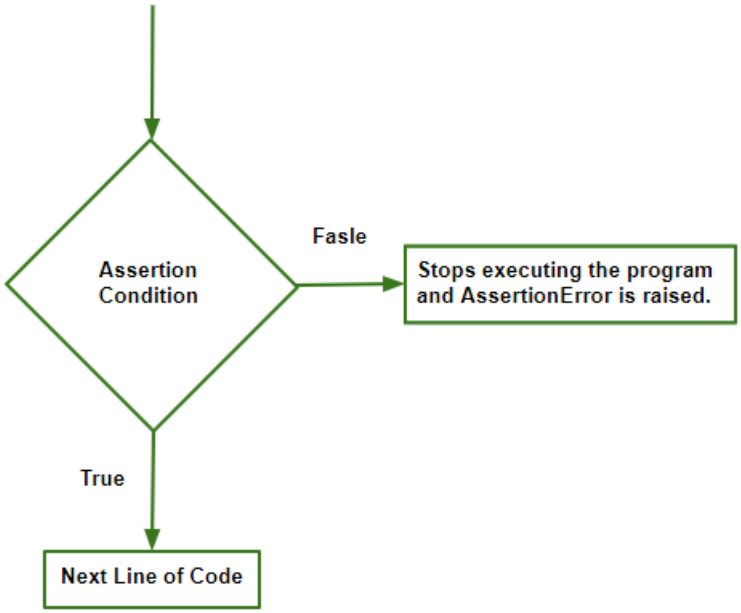
```
a=5
b=0
try:
    r1=a/b
except: print("division by zero")
else : print  (r1)
r2=a*b
print(r2)
```

Example 2: index out of range

```
a = ["Python", “Java", “C++"]
try:
    for i in range( 4 ):
        print( "The index and element from the array is", i, a[i] )
 except:
        print ("Index out of range")
```

Assertions in Python

- Python examines the adjacent expression, preferably true when it finds an assert statement. Python throws an AssertionError exception if the result of the expression is false.
- **The syntax for the assert clause is –**
- **assert** Expressions[, Argument]
- Assertion is a programming concept used while writing a code where the user declares a condition to be true using *assert* statement prior to running the module. If the condition is *True*, the control simply moves to the next line of code.
- In case if it is *False* the program stops running and returns *AssertionError* Exception.


Assertion
Condition
Fasle
Stops executing the program
and AssertionError is raised.
True
Next Line of Code

- # AssertionError with error_message.

```
x = 1
y = 0
assert y != 0, "Invalid Operation"          # denominator can't be 0
print(x / y)
```

```
# Handling it manually
try:
        x = 1
        y = 0
        assert y != 0, "Invalid Operation“
         print(x / y)
                # the errror_message provided by the
user gets printed
  except AssertionError as msg:
    print(msg)
```

Example for assertion

```
def square_root( Number ):
    try:
      assert ( Number > 0), "Give a positive integer"
      return Number**(1/2)
  except AssertionError as e:  print(e)

  #Calling function and passing the values
   print( square_root( 36 ) )
   print( square_root( -36 ) )
   s=square_root(4)
   print(s)
```

Roots of a quadratic equation

```
import math
def qroots(a, b, c):
    try:
        assert a != 0, "Not a quadratic equation as coefficient of x ^ 2 can't be 0"
        D = (b * b - 4 * a*c)
        assert D>= 0, "Roots are imaginary"
        r1 = (-b + math.sqrt(D))/(2 * a)
        r2 = (-b - math.sqrt(D))/(2 * a)
        print("Roots of the quadratic equation are :", r1, "", r2)
    except AssertionError as msg:
        print(msg)

qroots(-1, 5, -6)
qroots(1, 1, 6)
qroots(2, 12, 18)
```

- **Output :**

Roots of the quadratic equation are : 2.0 3.0
Roots are imaginary
Roots of the quadratic equation are : -3.0 -3.0

Python Exceptions List

Sr.No.	Name of the Exception	Description of the Exception
1	**Exception**	All exceptions of Python have a base class.
2	**StopIteration**	If the next() method returns null for an iterator, this exception is raised.
3	**SystemExit**	The sys.exit() procedure raises this value.
4	**StandardError**	Excluding the StopIteration and SystemExit, this is the base class for all Python built-in exceptions.
5	**ArithmeticError**	All mathematical computation errors belong to this base class.
6	**OverflowError**	This exception is raised when a computation surpasses the numeric data type's maximum limit.
7	**FloatingPointError**	If a floating-point operation fails, this exception is raised.
8	**ZeroDivisionError**	For all numeric data types, its value is raised whenever a number is attempted to be divided by zero.
9	**AssertionError**	If the Assert statement fails, this exception is raised.
10	**AttributeError**	This exception is raised if a variable reference or assigning a value fails.

11	**EOFError**	When the endpoint of the file is approached, and the interpreter didn't get any input value by raw_input() or input() functions, this exception is raised.
12	**ImportError**	This exception is raised if using the import keyword to import a module fails.
13	**KeyboardInterrupt**	If the user interrupts the execution of a program, generally by hitting Ctrl+C, this exception is raised.
14	**LookupError**	LookupErrorBase is the base class for all search errors.
15	**IndexError**	This exception is raised when the index attempted to be accessed is not found.
16	**KeyError**	When the given key is not found in the dictionary to be found in, this exception is raised.
17	**NameError**	This exception is raised when a variable isn't located in either local or global namespace.
18	**UnboundLocalError**	This exception is raised when we try to access a local variable inside a function, and the variable has not been assigned any value.
19	**EnvironmentError**	All exceptions that arise beyond the Python environment have this base class.
20	**IOError**	If an input or output action fails, like when using the print command or the open() function to access a file that does not exist, this exception is raised.

21	**SyntaxError**	This exception is raised whenever a syntax error occurs in our program.
22	**IndentationError**	This exception was raised when we made an improper indentation.
23	**SystemExit**	This exception is raised when the sys.exit() method is used to terminate the Python interpreter. The parser exits if the situation is not addressed within the code.
24	**TypeError**	This exception is raised whenever a data type-incompatible action or function is tried to be executed.
25	**ValueError**	This exception is raised if the parameters for a built-in method for a particular data type are of the correct type but have been given the wrong values.
26	**RuntimeError**	This exception is raised when an error that occurred during the program's execution cannot be classified.
27	**NotImplementedError**	If an abstract function that the user must define in an inherited class is not defined, this exception is raised.