MUSLIM ASSOCIATION COLLEGE OF ARTS AND SCIENCE

Panavoor,Thiruvananthapuram,Kerala
(Affiliated to the University of Kerala)

CS1543: PYTHON PROGRAMMING

S5 B.Sc Computer Science

Name : ……

Candidate Code: ……

CS1543: PYTHON PROGRAMMING:-SYLLABUS

Module I: Introduction to Python- Features of Python - Identifiers - Reserved Keywords - Variables Comments in Python – Input , Output and Import Functions - Operators – Data Types and Operations – int, float, complex, Strings, List, Tuple, Set,Dictionary - Mutable and Immutable Objects – Data Type Conversion - Illustrative programs: selection sort,insertion sort, bubble sort

Module II: Decision Making -conditional (if), alternative (if-else), if..elif..else -nested if - Loops for,range() while, break, continue, pass; Functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Lists as arrays. Illustrative programs: square root, gcd, exponentiation, sum of an array of numbers, linear search, binary search, bubble sort, insertion sort, selection sort

Module III: Built-in Modules - Creating Modules - Import statement - Locating modules - Namespaces and Scope - The dir() function - The reload function - Packages in PythonFiles and exception: text files, reading and writing files Renaming and Deleting files Exception handling exceptions, Exception with arguments, Raising an Exception - User defined Exceptions - Assertions in python

Module IV: GUI Programming- Introduction – Tkinter Widgets – Label – Message Widget – Entry Widget – Text Widget – tk Message Box – Button Widget – Radio Button- Check Button – Listbox- Frames _ Toplevel Widgets – Menu Widget

MODULE 1

Module I: Introduction to Python- Features of Python - Identifiers - Reserved Keywords - Variables Comments in Python – Input , Output and Import Functions - Operators – Data Types and Operations – int, float, complex, Strings, List, Tuple, Set,Dictionary - Mutable and Immutable Objects – Data Type Conversion - Illustrative programs: selection sort,insertion sort, bubble sort

Introduction to Python

- Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990.
- Python is easy to learn yet powerful and versatile scripting language, which makes it attractive for Application Development.
- Python's syntax and dynamic typing with its interpreted nature make it an ideal language for scripting and rapid application development.

Characteristics of Python

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Advantages of Python

1. **Python is Interpreted –** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
2. **Python is Interactive –** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
3. **Python is Object-Oriented –** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
4. **Python is a Beginner's Language –** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Features in Python

There are many features in Python, some of which are discussed below –

1.Easy to code:

Python is high level programming language.Python is very easy to learn language as compared to other language like c, c#, java script, java etc.It is very easy to code in python language and anybody can learn python basic in few hours or days.It is also developer-friendly language.

2. Free and Open Source:

Python language is freely available at official website.Since, it is open-source, this means that source code is also available to the public.So you can download it as, use it as well as share it.

3.Object-Oriented Language:

One of the key features of python is Object-Oriented programming.Python supports object oriented language and concepts of classes, objects encapsulation etc.

4. GUI Programming Support:

Graphical Users interfaces can be made using a module such as PyQt5, PyQt4, wxPython or Tk in python.PyQt5 is the most popular option for creating graphical apps with Python.

5. High-Level Language:

Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory.

6.Extensible feature:

Python is a Extensible language. we can write our some python code into c or c++ language and also we can compile that code in c/c++ language.

7. Python is Portable language:

Python language is also a portable language.for example, if we have python code for windows and if we want to run this code on other platform such as Linux, Unix and Mac then we do not need to change it, we can run this code on any platform.

8. Python is Integrated language:

Python is also an Integrated language because we can easily integrated python with other language like c, c++ etc.

9. Interpreted Language:

Python is an Interpreted Language. because python code is executed line by line at a time. The source code of python is converted into an immediate form called bytecode.

10. Large Standard Library

Python has a large standard library which provides rich set of module and functions so you do not have to write your own code for every single thing.There are many libraries present in python for such as regular expressions, unit-testing, web browsers etc.

11. Dynamically Typed Language:

Python is dynamically-typed language. That means the type (for example- int, double, long etc) for a variable is decided at run time not in advance.because of this feature we don't need to specify the type of variable.

Python Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Example: a=10

Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore _. Names like myClass, var_1 and print_this_to_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is a valid name.
3. Keywords cannot be used as identifiers.
4. We cannot use special symbols like !, @, #, $, % etc. in our identifier.
5. An identifier can be of any length.

Keywords in Python

- A python keyword is a reserved word which you can't use as a name of your variable, class, function etc.
- These keywords have a special meaning and they are used for special purposes in Python programming language.
- For example – Python keyword "while" is used for while loop thus you can't name a variable with the name "while" else it may cause compilation error.
- There are total 33 keywords in Python

```
False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass
```

Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.
- Comments starts with a #, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Multi Line Comments

Python does not really have a syntax for multi line comments.To add a multiline comment you could insert a # for each line or make the comments inside three quotes.

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Or, not quite as intended, you can use a multiline string.

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Python Output Function

- In python, if you want to show something on screen as an output, there we use the *print()* function. Using the print() function you can show the output data on your standard screen.
- Normally print() function is used to show the output data of variables and strings.

Example:

```
print("Welcome to Python")
print(a)
```

String Formatting:

- There is a special function called format() which is used with a string, that gives you more flexibility over the print() function. format() function is used with a string and you can insert variables in specific places of that particular string.
- We use {} curly brackets to hold the places of variables or string that pass in the format() function.
- Example

```
s= "Kiran"
age=25
print("My name is {} and I am {} years old ". format(s,age))
```

Output: My name is Kiran and I am 25 years old

Python Input Function

- Here we use the input() function which allows the user to input the values in the programme.
- Whenever you input a value in the programme with the help of input() function the value is treated as a string and what if you want to enter an integer or float for that you need to convert it into corresponding data type

Syntax1:

```
variable = input()
Example:
        a=input()
```

Syntax2: variable = input("Prompt")

```
Example:
        a=input("Enter the value")
```

The prompt is a string which is used as a message that displays on the screen

Import in Python:

- Import is a keyword which is used to import the definitions of the module (modules are the python files which contain the prewritten code or function) in the current file. Import keyword is used along with another keyword from which is used to import the module
- Suppose you want a programme which accepts an integer and give the square root of the integer. For this, you can make a program or you can use a predefined function sqrt() which is a part of module math. So to use the sqrt() function you need to import it on your current file by using the import keyword.

Let's understand it with an example

```
from math import sqrt
var = int(input("Enter a Number: "))
sqt = sqrt(var)
print(sqt)
```

Output:
Enter a Number:25
5

Operators in Python

- Operators are the constructs which can manipulate the value of operands.
- Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Types of Operator

Python language supports the following types of operators.

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Membership Operators
6. Identity Operators

1.Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2

% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

2.Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.

>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

3.Assignment Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a

//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

4.Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

5.Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

6.Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below −

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

1.Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example −

```
var1 = 1
var2 = 10
```

Python supports four different numerical types −

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

2.Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example –

```
str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

3.Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example –

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

4.Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example –

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

5.Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example –

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result –

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Mutable vs Immutable Objects in Python

- Every variable in python holds an instance of an object. There are two types of objects in python i.e. **Mutable** and **Immutable objects**.
- Whenever an object is instantiated, it is assigned a unique object id. The type of the object is defined at the runtime and it can't be changed afterwards. However, it's state can be changed if it is a mutable object.
- To summarise the difference, mutable objects can change their state or contents and immutable objects can't change their state or content.
- **Immutable Objects :** These are of in-built types like *int, float, bool, string, unicode, tuple*. In simple words, an immutable object can't be changed after it is created.
- **Mutable Objects :** These are of type *list, dict, set* . Custom classes are generally mutable.

Type Conversion in Python

Python defines type conversion functions to directly convert one data type to another which is useful in day to day and competitive programming. The following are type conversion functions in python.

1.int(a,base) : This function converts any data type to integer**.** 'Base' specifies the base in which string is if data type is string.

2. float() : This function is used to convert **any data type to a floating point number**

```
Example

s = "10010"

# printing string converting to int base 2
c = int(s,2)
print ("After converting to integer base 2 : ")
print (c)

# printing string converting to float
e = float(s)
print ("After converting to float : ")
print (e)
```

Output:

```
After converting to integer base 2 : 18
After converting to float : 10010.0
```

3. ord() : This function is used to convert a character to integer.

4. hex() : This function is to convert integer to hexadecimal string.

5. oct() : This function is to convert integer to octal string.

Example:

```
# initializing integer
s = '4'

c = ord(s)
print ("After converting character to integer : ")
print (c)

# printing integer converting to hexadecimal string
c = hex(56)
print ("After converting 56 to hexadecimal string : ")
print (c)

# printing integer converting to octal string
c = oct(56)
print ("After converting 56 to octal string : ")
print (c)
```

Output:

```
After converting character to integer : 52
After converting 56 to hexadecimal string : 0x38
After converting 56 to octal string : 0o70
```

6. tuple() : This function is used to convert to a tuple.

7. set() : This function returns the type after converting to set.

8. list() : This function is used to convert any data type to a list type.

Example:

```
# initializing string
s = 'geeks'

# printing string converting to tuple
c = tuple(s)
print ("After converting string to tuple : ")
print (c)

# printing string converting to set
c = set(s)
print ("After converting string to set : ")
print (c)
```

```
# printing string converting to list
c = list(s)
print ("After converting string to list : ")
print (c)
```

Output:

```
After converting string to tuple : ('g', 'e', 'e', 'k', 's')
After converting string to set : {'k', 'e', 's', 'g'}
After converting string to list : ['g', 'e', 'e', 'k', 's']
```

9. dict() : This function is used to convert a tuple of order (key,value) into a dictionary.
10. str() : Used to convert integer into a string.
11. complex(real,imag) : : This function converts real numbers to complex(real,imag) number.

Example

```
# initializing integers
a = 1
b = 2

# initializing tuple
tup = (('a', 1) ,('f', 2), ('g', 3))

# printing integer converting to complex number
c = complex(1,2)
print ("After converting integer to complex number : ")
print (c)

# printing integer converting to string
c = str(a)
print ("After converting integer to string : ")
print (c)

# printing tuple converting to expression dictionary
c = dict(tup)
print ("After converting tuple to dictionary : ")
print (c)
```

Output:

```
After converting integer to complex number : (1+2j)
After converting integer to string : 1
After converting tuple to dictionary : {'a': 1, 'f': 2, 'g': 3}
```

12. chr(number) : : This function converts number to its corresponding ASCII character.

Example:

```
# Convert ASCII value to characters
a = chr(76)
b = chr(77)

print(a)
print(b)
```

Output:

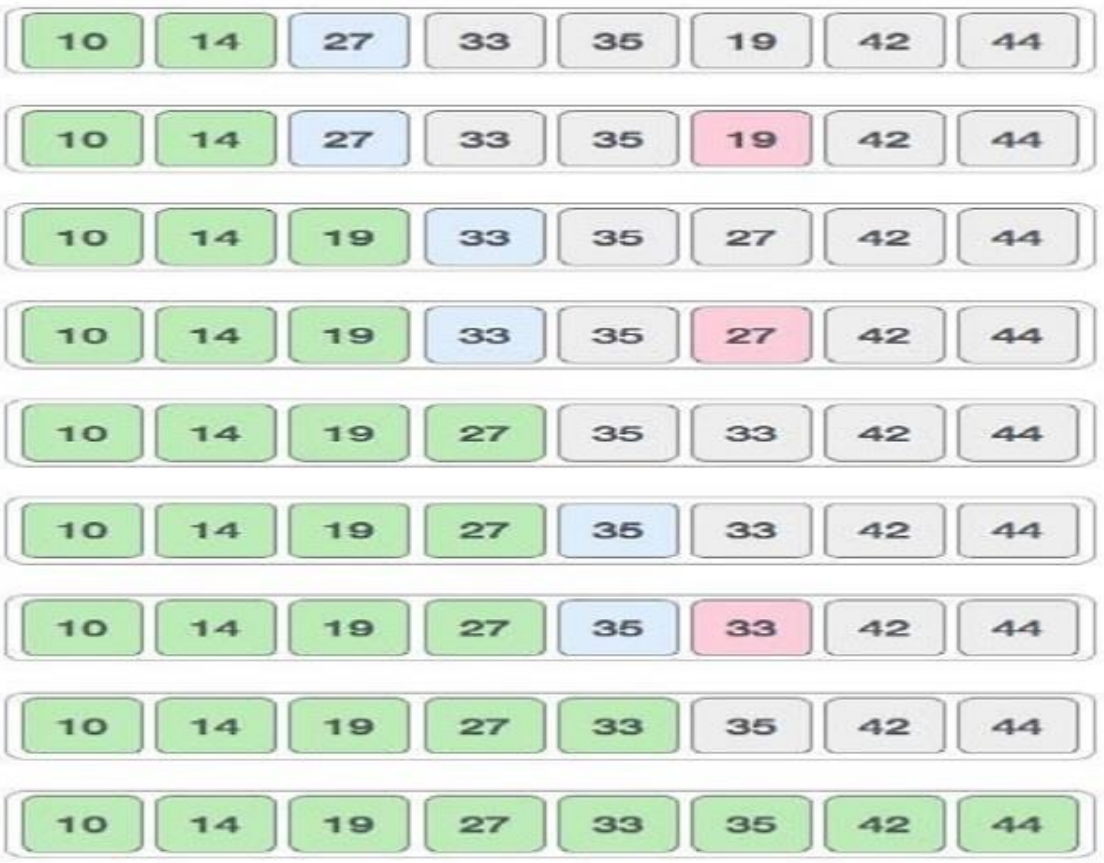
```
L

M
```

Selection sort in Python

In the **selection sort** algorithm, an array is sorted by recursively finding the minimum element from the unsorted part and inserting it at the beginning. Two subarrays are formed during the execution of Selection sort on a given array.

- The subarray, which is already sorted
- The subarray, which is unsorted.

During every iteration of selection sort, the minimum element from the unsorted subarray is popped and inserted into the sorted subarray.



```
A = ['t','u','t','o','r','i','a','l']

for i in range(len(A)):

   min_= i

   for j in range(i+1, len(A)):

      if A[min_] > A[j]:

         min_ = j

   #swap

A[i], A[min_] = A[min_], A[i]

# main

for i in range(len(A)):

   print(A[i])
```

Output

```
a
i
l
o
r
t
t
u
```

Insertion Sort in Python

- Iterate over the input elements by growing the sorted array at each iteration.
- Compare the current element with the largest value available in the sorted array.
- If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position in the array.
- This is achieved by shifting all the elements towards the right, which are larger than the current element, in the sorted array to one position ahead.

Now let's see the visual representation of the algorithm


```
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        # Move elements of arr[0..i-1], that are greater
        than key,
        # to one position ahead of their current position
        j = i-1
        while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

```
# main

arr = ['t','u','t','o','r','i','a','l']

insertionSort(arr)

print ("The sorted array is:")

for i in range(len(arr)):

   print (arr[i])
```

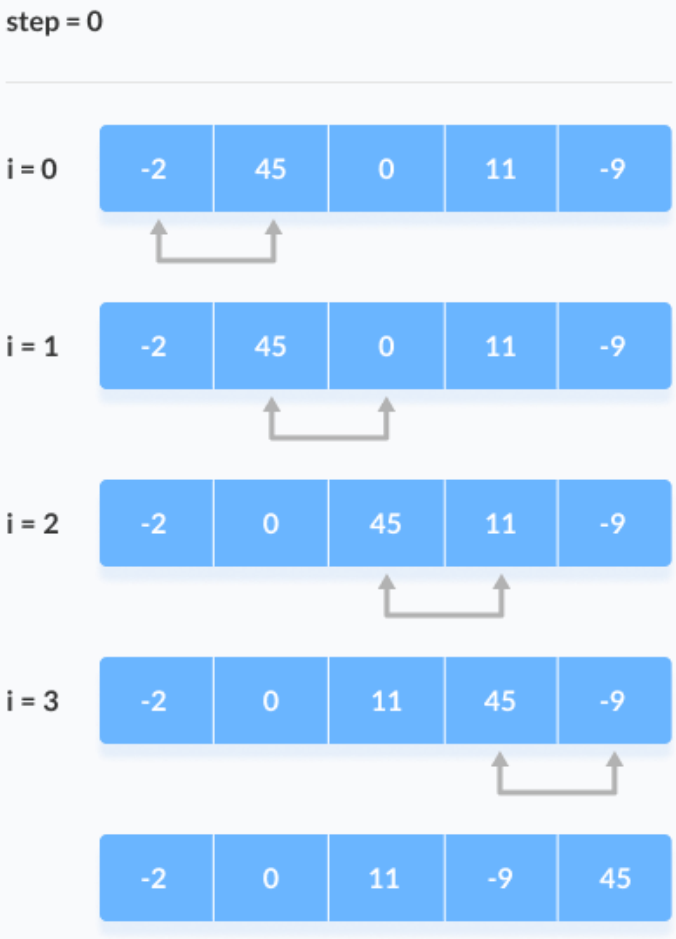
Output

```
The sorted array is:
a
i
l
o
r
t
t
u
```

Bubble Sort in Python

Step0: Starting from the first index, compare the first and the second elements.If the first element is greater than the second element, they are swapped.
Now, compare the second and the third elements. Swap them if they are not in order.

The above process goes on until the last element.

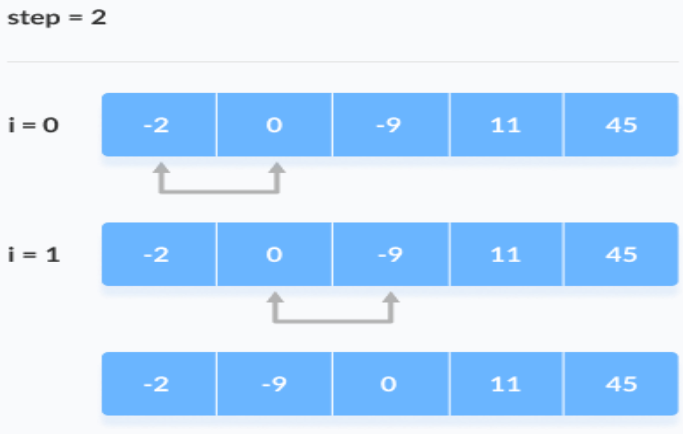


Compare the adjacent elements

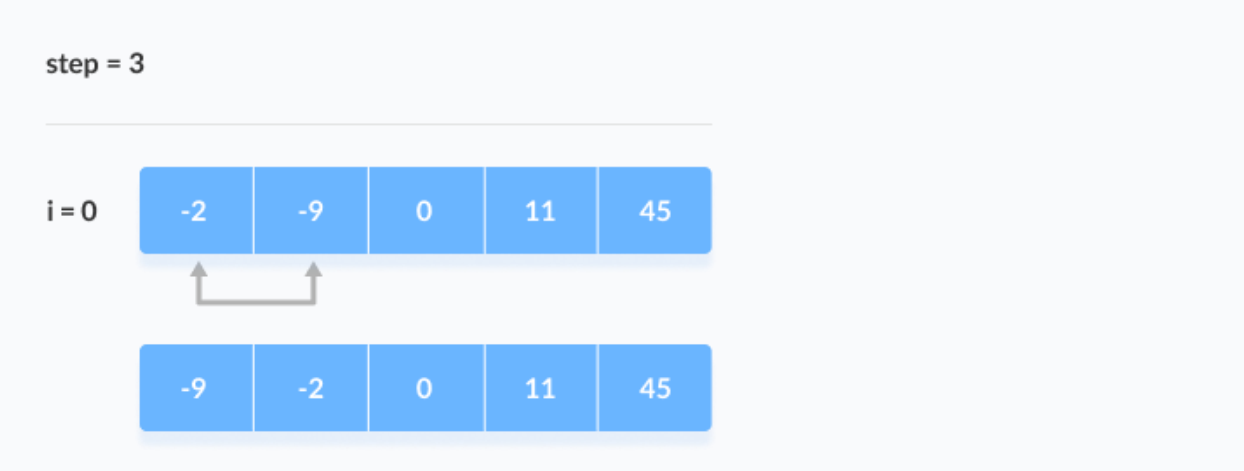
Step1:
The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.
In each iteration, the comparison takes place up to the last unsorted element.

The array is sorted when all the unsorted elements are placed at their correct positions.

Compare the adjacent elements



Compare the adjacent Elements



Compare the adjacent elements

```
def bubbleSort(array):

    # run loops two times: one for walking throught the array
    # and the other for comparison
    for i in range(len(array)):
        for j in range(0, len(array) - i - 1):

            # To sort in descending order, change > to < in this line.
            if array[j] > array[j + 1]:

                # swap if greater is at the rear position
                (array[j], array[j + 1]) = (array[j + 1], array[j])


data = [-2, 45, 0, 11, -9]
bubbleSort(data)
print('Sorted Array in Asc ending Order:')
print(data)
```

MODULE 2

Module II: Decision Making -conditional (if), alternative (if-else), if..elif..else -nested if - Loops for,range() while, break, continue, pass; Functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Lists as arrays. Illustrative programs: square root, gcd, exponentiation, sum of an array of numbers, linear search, binary search, bubble sort, insertion sort, selection sort

Decision Making in Python

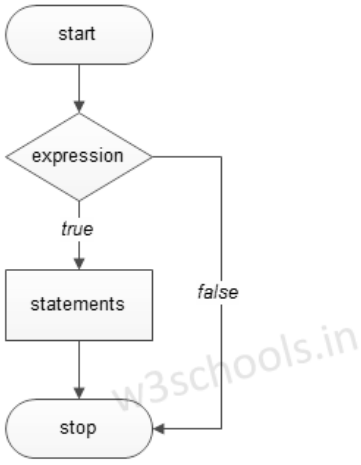
Decisions in a program are used when the program has conditional choices to execute a code block. It is the prediction of conditions that occur while executing a program to specify actions. Multiple expressions get evaluated with an outcome of either TRUE or FALSE.

Python provides various types of conditional statements:

1. if Statement
2. if_else Statement
3. elif Statement

1.if Statements

It consists of a Boolean expression which results are either TRUE or FALSE, followed by one or more statements.



Synatx:

```
if expression:
        #execute your code
```

Example:

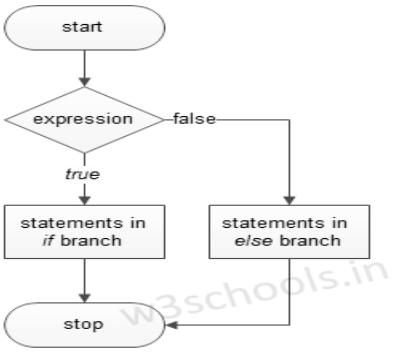
```
a = 15
if a > 10:
print("a is greater")
```

Output:

```
a is greater
```

2.if else Statements

It also contains a Boolean expression. The if the statement is followed by an optional else statement & if the expression results in FALSE, then else statement gets executed. It is also called alternative execution in which there are two possibilities of the condition determined in which any one of them will get executed.



Synatx:

```
if expression:
        #execute your code
else:
       #execute your code
```

Example:

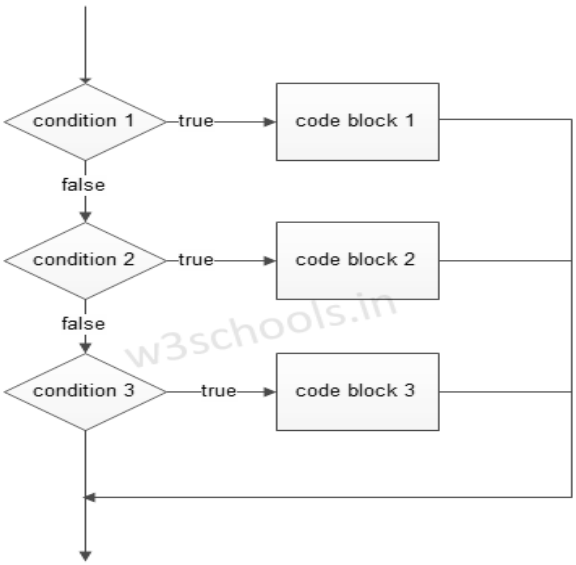
```
a = 15
b = 20
if a > b:
    print("a is greater")
else:
    print("b is greater")
```

Output:

```
b is greater
```

3.elif Statements

- We can implement if statement and or if-else statement inside another if or if - else statement. Here more than one if conditions are applied & there can be more than one if within elif.
- elif - is a keyword used in Python replacement of else if to place another condition in the program. This is called chained conditional.



Example:

```
if expression:
        #execute your code
elif expression:
        #execute your code
else:
        #execute your code
```

Example:

```
a = 15
b = 15
if a > b:
    print("a is greater")
elif a == b:
   print("both are equal")
 else:
    print("b is greater")
```

Output:

```
both are equal
```

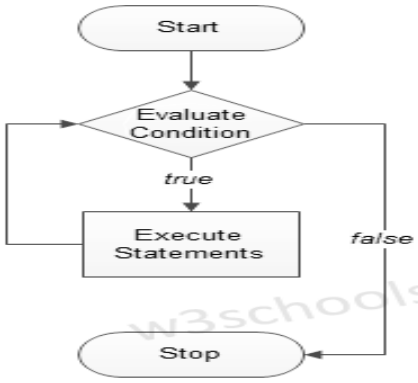
looping in Python

- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.
- Programming languages provide various control structures that allow for more complicated execution paths.A loop statement allows us to execute a statement or group of statements multiple times.
- Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop : Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	for loop :-Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	nested loops You can use one or more loop inside any another while, for or do..while loop.

1.While Loop

- A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
- Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.



Synatx:

```
while expression:
       #execute your code
```

Example:

```
count =1
while count < 6 :
   print (count)
   count+=1
```

Output:

1
2
3
4
5

2.for loop

It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

```
for iterating_var in sequence:
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Example :

```
for x in range (0,3) :
        print (x)
```

Output:

0
1
2

3.Nested Loop

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax of nested for loop

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

The syntax for a nested while loop

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

Example:

```
for g in range(1, 6):
    for k in range(1, 3):
        print ("%d * %d = %d" % ( g, k, g*k))
```

Output:

1 * 1 = 1
1 * 2 = 2
2 * 1 = 2
2 * 2 = 4
3 * 1 = 3
3 * 2 = 6
4 * 1 = 4
4 * 2 = 8
5 * 1 = 5
5 * 2 = 10

Loop control Statements in Python

- These statements are used to change execution from its normal sequence.
- Python supports three types of loop control statements:

1.Break statement:-It is used to exit a while loop or a for a loop. It terminates the looping & transfers execution to the statement next to the loop.

Sytanx:

```
break
```

Example:

```
i = 0
while i <= 10:
        print (i)
        i++
if i == 3:
        break
```

Output:

```
0
1
2
```

2.Continue statement:- It causes the looping to skip the rest part of its body & start re-testing its condition. The continue statement in Python returns the control to the beginning of the current loop. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.

Sytanx:

```
continue
```

Example:

```
i = 0
while i <= 10:
        print (i)
        i++
if i == 3:
        continue
```

Output:

```
0
1
2
4
5
6
7
8
9
10
```

3.Pass statement:-

It is used in Python to when a statement is required syntactically, and the programmer does not want to execute any code block or command.

It is used when a statement is required syntactically but you do not want any command or code to execute.

The pass statement is a *null* operation; nothing happens when it executes. The pass statement is also useful in places where your code will eventually go, but has not been written yet i.e. in stubs.

```
Syntax:
       pass
```

Example

```
for i in 'Python':
   if i == 'h':
      pass
       print ('This is pass block')
   print ('Current Letter :', i)

print ("Good bye!")
```

Output

When the above code is executed, it produces the following result –

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Python functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses.
- Any input parameters or arguments should be placed within these parentheses.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function
- . A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
          "function body"
   return [expression]
```

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

Example

The following function takes a string as input parameter and prints it on standard screen.

```
   def printme( str ):
          print str
   return
```

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, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
def printme(a):
        print a
   return;

# Now you can call printme function
printme(21)
printme('x')
printme("hai"
```

Output: 21 x hai

The *return* Statement

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.

All the above examples are not returning any value. You can return a value from a function as follows –

```
# Function definition is here
def sum( a, b):
   # Add both the parameters and return them."
   total = a + b
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result –

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular variable. There are two basic scopes of variables in Python –

- Global variables
- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example –

```
total = 0; # This is global variable.
# Function definition is here
def sum( a, b ):
   # Add both the parameters and return them."
   total = a + b; # Here total is local variable.
   print "Inside the function local total : ", total
   return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result –

```
Inside the function local total :  30
Outside the function global total :  0
```

Parameters in Python Functions

Paremeters are also known as arguments,they are values passed to the fuctions.You can call a function by using the following types of formal arguments –

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

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.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows –

```
# Function definition is here
def printme( a ):
      print(a)
   return;

# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result –

```
Traceback (most recent call last):
   File "test.py", line 11, in <module>
      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

2.Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you 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. You can also make keyword calls to the *printinfo()* function in the following ways −

```
# Function definition is here
def printinfo( name, age ):
   print "Name: ", name
   print "Age: ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="lal" )
```

Output:

```
Name:  lal
Age:  50
```

2.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 gives an idea on default arguments, it prints default age if it is not passed −

```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="lal" )
printinfo( name="lal" )
```

When the above code is executed, it produces the following result −

```
Name:  lal
Age  50
Name:  lal
Age  35
```

4.Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
example:
def printinfo(a,*var_argument)
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
def printinfo( arg1, *vartuple ):
      print "Output is: "
      print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result −

```
Output is:
10
Output is:
70
60
50
```

Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You 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
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows –

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

Following is the example to show how *lambda* form of function works –

```
# 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 )
```

When the above code is executed, it produces the following result –

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

Recursion in Python

- Python also accepts function recursion, which means a defined function can call itself.
- Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
- In this example, *fact()* is a function that we have defined to call itself ("recurse"). The factorial of a number is the product of all the integers from 1 to that number.
- For example, the factorial of 6 is `1*2*3*4*5*6 = 720`. Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1.

```
def fact (x):

    if x == 1:
        return 1
    else:
        return (x * fact (x-1))


num = 3
print("The factorial of", num, "is", fact (num))
```

Output

The factorial of 3 is 6

String in Python

- A string is a sequence of characters.
- A character is simply a symbol. For example, the English language has 26 characters.
- Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable.
- For example –

  var1 = 'Hello World!'

  var2 = "Python Programming"

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

For example –

```
var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

When the above code is executed, it produces the following result –

```
var1[0]:  H
var2[1:5]:  ytho
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

String Slicing in Python

Python slicing is about obtaining a sub-string from the given string by slicing it respectively from start to end.Python slicing can be done in two ways.

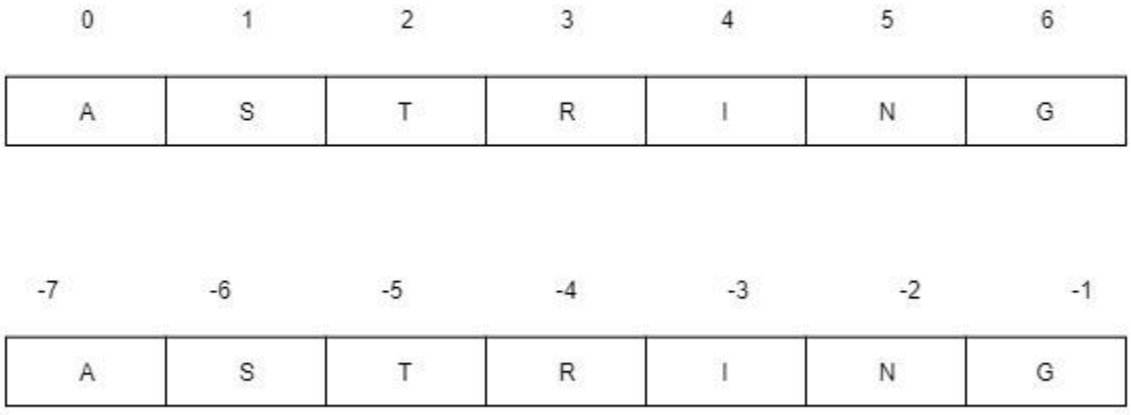
1. slice() Constructor
2. Extending Indexing

1.slice() Constructor : The slice() constructor creates a slice object representing the set of indices specified by range(start, stop, step).

Syntax:

- *slice(stop)*
- *slice(start, stop, step)*

Parameters:
start: *Starting index where the slicing of object starts.*
stop: *Ending index where the slicing of object stops.*
step: *It is an optional argument that determines the increment between each index for slicing.*



Example:

```
String ='ASTRING'
s1 = slice(3)
s2 = slice(1, 5)
s3 = slice(-1, -7, -3)

print("String slicing")
print(String[s1])
print(String[s2])
print(String[s3])
```

Output:

String slicing
AST
STRI
GR

2.Extending indexing

In Python, indexing syntax can be used as a substitute for the slice object. This is an easy and convenient way to slice a string both syntax wise and execution wise.

Syntax

string[start:end:step]
start, end and step have the same mechanism as slice() constructor.

Example

```
String ='ASTRING'

# Using indexing sequence

print(String[:3])

print(String[1:5:2])

print(String[-1:-12:-2])

# Prints string in reverse

print("\nReverse String")

print(String[::-1])
```

Output:

AST
SR
GITA
Reverse String
GNIRTSA

String Immutability

In python, the string data types are immutable. Which means a string value cannot be updated. We can verify this by trying to update a part of the string which will led us to an error.

Example

```
t= "Tutorialspoint"
print type(t)
t[0] = "M"
```

When we run the above program, we get the following output –

```
t[0] = "M"
TypeError: 'str' object does not support item assignment
```

String Functions in Python

Python provides lots of built-in methods which we can use on strings.Below are the list of string methods available in Python 3.

Method	Description	Examples

capitalize()	Returns a copy of the string with its first character capitalized and the rest lowercased.	>>> mystring = "hello python"
		>>> print(mystring.capitalize())
		Hello python

Casefold()	Returns a casefolded copy of the string. Casefolded strings may be used for caseless matching.	>>> mystring = "hello PYTHON"
		>>> print(mystring.casefold())
		hello python

Center(width, [fillchar])	Returns the string centered in a string of length *width*.	>>> mystring = "Hello"
		>>> x = mystring.center(12,"-")
		>>> print(x)
		---Hello----

Count(sub, [start], [end])	Returns the number of non-overlapping occurrences of substring (*sub*) in the range [*start*, *end*]. Optional arguments *start*and *end* are	>>> mystr = "Hello Python"
		>>> print(mystr.count("o"))
		2
		>>> print(mystr.count("th"))
		1

	interpreted as in slice notation.	>>> print(mystr.count("l"))
		2
		>>> print(mystr.count("h"))
		1
		>>> print(mystr.count("H"))
		1
		>>> print(mystr.count("hH"))
		0

endswith(suffix, [start], [end])	Returns True if the string ends with the specified suffix, otherwise it returns False.	>>> mystr = "Python" >>>print(mystr.endswith("y")) False >>>print(mystr.endswith("hon")) True
Find(sub, [start], [end])	Returns the lowest index in the string where substring *sub* is found within the slice s[start:end].	>>> mystring = "Python" >>>print(mystring.find("P")) 0 >>>print(mystring.find("on")) 4

Index(sub, [start], [end])	Searches the string for a specified value and returns the position of where it was found	>>> mystr = "HelloPython"
		>>> print(mystr.index("P"))
		5
		>>>print(mystr.index("hon"))
		8
		>>> print(mystr.index("o"))
		4

isalnum	Returns True if all characters in the string are alphanumeric	>>> mystr = "HelloPython"
		>>> print(mystr.isalnum())
		True
		>>> a = "123"
		>>> print(a.isalnum())
		True
		>>> a= "$*%!!!"
		>>> print(a.isalnum())
		False

Isalpha()	Returns True if all characters in the string are in the alphabet	>>> mystr = "HelloPython"
		>>> print(mystr.isalpha())
		True
		>>> a = "123"

		>>> print(a.isalpha())
		False
		>>> a= "$*%!!!"
		>>> print(a.isalpha())
		False

Isdecimal()	Returns True if all characters in the string are decimals	>>> mystr = "HelloPython"
		>>> print(mystr.isdecimal())
		False
		>>> a="1.23"
		>>> print(a.isdecimal())
		False
		>>> c = u"\u00B2"
		>>> print(c.isdecimal())
		False
		>>> c="133"
		>>> print(c.isdecimal())
		True

Isdigit()	Returns True if all characters in the string are digits	>>> c="133"
		>>> print(c.isdigit())
		True
		>>> c = u"\u00B2"
		>>> print(c.isdigit())
		True
		>>> a="1.23"
		>>> print(a.isdigit())
		False

Islower()	Returns True if all characters in the string are lower case	>>> c="Python"
		>>> print(c.islower())
		False
		>>> c="_user_123"
		>>> print(c.islower())
		True
		>>> print(c.islower())
		False

Isnumeric()	Returns True if all characters in the string are numeric	>>> c="133"
		>>> print(c.isnumeric())
		True
		>>> c="_user_123"
		>>> print(c.isnumeric())
		False

		>>> c="Python"
		>>> print(c.isnumeric())
		False

isprintable()	Returns True if all characters in the string are printable	>>> c="133"
		>>> print(c.isprintable())
		True
		>>> c="_user_123"
		>>> print(c.isprintable())
		True
		>>> c="\t"
		>>> print(c.isprintable())
		False

isspace()	Returns True if all characters in the string are whitespaces	>>> c="133"
		>>> print(c.isspace())
		False
		>>> c="Hello Python"
		>>> print(c.isspace())
		False
		73
		>>> c="Hello"
		>>> print(c.isspace())
		False
		>>> c="\t"
		>>> print(c.isspace())
		True

isupper()	Returns True if all characters in the string are upper case	>>> c="Python"
		>>> print(c.isupper())
		False
		>>> c="PYHTON"
		>>> print(c.isupper())
		True
		>>> c="\t"
		>>> print(c.isupper())
		False

join(iterable)	Joins the elements of an iterable to the end of the string	>>> a ="-"
		>>> print(a.join("123"))
		1-2-3
		>>> a="Hello Python"
		>>> a="**"
		>>> print(a.join("Hello

		Python")) H**e**l**l**o** **P**y**t**h**o**n

lower()	Converts a string into lower case	>>> a = "Python"
		>>> print(a.lower())
		Python

replace(*old*, *new*[,*count*])	Returns a string where a specified value is replaced with a specified value	>>> mystr = "Hello Python.Hello Java. Hello C++." >>>print(mystr.replace("Hello","Bye")) Bye Python. Bye Java. ByeC++.

swapcase()	Swaps cases, lower case becomes upper case and vice versa	>>> mystr = "Hello PYthon"
		>>> print(mystr.swapcase())
		hELLO python

title()	Converts the first character of each word to upper case	>>> mystr = "Hello PYthon"
		>>> print(mystr.title())
		Hello Python
		>>> mystr = "HELLO JAVA"
		>>> print(mystr.title())
		Hello Java

upper()	Converts a string into upper case	>>> mystr = "hello Python"
		>>> print(mystr.upper())
		HELLO PYTHON

String Module

- The string module provides additional tools to manipulate strings.
- It's a built-in module and we have to import it before using any of its constants and classes.
- Python String Module Classes-Python string module contains two classes –
    1. Formatter
    2. Template.

1.Formatter

It behaves exactly same as str.format() function. This class become useful if you want to subclass it and define your own format string syntax

Example:

```
from string import Formatter

    formatter = Formatter()
    print(formatter.format('{website}', website='JournalDev'))
    print(formatter.format('{} {website}', 'Welcome to', website='JournalDev'))

    # format() behaves in similar manner
    print('{} {website}'.format('Welcome to', website='JournalDev'))
```

Output:

Welcome to JournalDev
Welcome to JournalDev

2.Template

This class is used to create a string template for simpler string substitutions as described in PEP 292. It's useful in implementing internationalization (i18n) in an application where we don't need complex formatting rules.

Example:

```
from string import Template
        t = Template('$name is the $title of $company')
        s = t.substitute(name='Pankaj', title='Founder', company='JournalDev.')
        print(s)
```

Output: Pankaj is the Founder of JournalDev.

List as Arrays

- During programming, there will be instances when you will need to convert existing lists to arrays in order to perform certain operations on them (arrays enable mathematical operations to be performed on them in ways that lists do not).
- In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.
- It can have any number of items and they may be of different types (integer, float, string etc.).
- Lists can be converted to arrays using the built-in functions in the Python numpy library.
- numpy provides us with two functions to use when converting a list into an array:

1. numpy.array()
2. numpy.asarray()

1. Using numpy.array()

This function of the numpy library takes a list as an argument and returns an array that contains all the elements of the list. See the example below:

Example:

```
import numpy as np
    my_list = [2,4,6,8,10]
    my_array = np.array(my_list)
    # printing my_array
    print my_array
    # printing the type of my_array
    print type(my_array)
```

Output:

```
    [ 2  4  6  8 10]
    <type 'numpy.ndarray'>
```

2. Using numpy.asarray()

This function calls the numpy.array() function inside itself. See the definition below

```
import numpy as np
    my_list = [2,4,6,8,10]
    my_array = np.asarray(my_list)
    # printing my_array
    print my_array
    # printing the type of my_array
    print type(my_array)
```

Output:

```
[ 2  4  6  8 10]
<type 'numpy.ndarray'>
```

The main difference between np.array() and np.asarray() is that the copy flag is false in the case of np.asarray(), and true (by default) in the case of np.array().

Illustrative Programs:

1.Python program to find square root of a number

Python number method **sqrt()** returns the square root of x for x > 0.

```
import math   # This will import math module
a=100
b=7
c=81
print "Square root of a is : ", math.sqrt(a)
print " Square root of b is ", math.sqrt(b)
print " Square root of c is ", math.sqrt(c)
```

Output

```
Square root of a is 10
Square root of b is 2.645
Square root of c is 9
Python program to find gsd of two numbers
```

2.Python program to find gcd of two numbers

Greatest common divisor or gcd is a mathematical expression to find the highest number which can divide both the numbers whose gcd has to be found with the resulting remainder as zero. It has many mathematical applications. Python has a inbuilt gcd function in the math module which can be used for this purpose.

gcd():-It accepts two integers as parameter and returns the integer which is the gcd value.

Syntax

```
Syntax: gcd(x,y)

Where x and y are positive integers.
```

Example of gcd()

In the below example we print the result of gcd of a pair of integers.

```
import math

print ("GCD of 75 and 30 is ",math.gcd(75, 30))

print ("GCD of 0 and 12 is ",math.gcd(0, 12))

print ("GCD of 0 and 0 is ",math.gcd(0, 0))

print ("GCD of -24 and -18 is ",math.gcd(-24, -18))
```

Output

Running the above code gives us the following result –

```
GCD of 75 and 30 is 15
GCD of 0 and 12 is 12
GCD of 0 and 0 is 0
GCD of -24 and -18 is 6
```

3.Python program for exponentiation

a= 3

b= 4

print "Exponential Value is: ", pow(a, b)

output:

Exponential value is 81

4.Python program to find sum of elements in an array

```
arr = [1, 2, 3, 4, 5];
sum = 0;
for i in range(0, len(arr)):
sum = sum + arr[i];
print("Sum of all the elements of an array: " + str(sum));
```

output

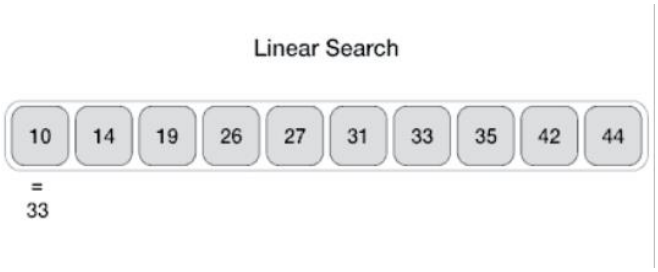
```
Sum of all the elements of an array: 15
```

5.Python program to perform linear search

Algorithm

- Start from the leftmost element of given arr[] and one by one compare element x with each element of arr[]
- If x matches with any of the element, return the index value.
- If x doesn't match with any of elements in arr[] , return -1 or element not found.

Now let's see the visual representation of the given approach −




```
def linearsearch(arr, x):
   for i in range(len(arr)):
      if arr[i] == x:
         return i
   return -1
arr = ['t','u','t','o','r','i','a','l']
x = 'a'
print("element found at index "+str(linearsearch(arr,x)))
```

Here we linearly scan the list with the help of for loop.

Output

```
element found at index 6
```

6.Python program to perform binary search

Problem statement − We will be given a sorted list and we need to find an element with the help of a binary search.

Algorithm

- Compare x with the middle element.
- If x matches with the middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for the right half.
- Else (x is smaller) recur for the left half

Recursive Algorithm

Example

```
def binarySearchAppr (arr, start, end, x):
# check condition
if end >= start:
   mid = start + (end- start)//2
   # If element is present at the middle
   if arr[mid] == x:
      return mid
   # If element is smaller than mid
   elif arr[mid] > x:
      return binarySearchAppr(arr, start, mid-1, x)
   # Else the element greator than mid
   else:
      return binarySearchAppr(arr, mid+1, end, x)
   else:
   # Element is not found in the array
      return -1
arr = sorted(['t','u','t','o','r','i','a','l'])
x ='r'
result = binarySearchAppr(arr, 0, len(arr)-1, x)
if result != -1:
   print ("Element is present at index "+str(result))
else:
print ("Element is not present in array")
```

```
Output: Element is present at index 4
```

MODULE 3

Module III: Built-in Modules - Creating Modules - Import statement - Locating modules - Namespaces and Scope - The dir() function - The reload function - Packages in PythonFiles and exception: text files, reading and writing files Renaming and Deleting files Exception handling exceptions, Exception with arguments, Raising an Exception - User defined Exceptions - Assertions in python

Built-in Modules in Python

- The Python interpreter has a number of built-in functions. They are loaded automatically as the interpreter starts and are always available.
- For example, print() and input() for I/O, number conversion functions int(), float(), complex(), data type conversions list(), tuple(), set(), etc.
- In addition to built-in functions, a large number of pre-defined functions are also available as a part of libraries bundled with Python distributions. These functions are defined in modules. A module is a file containing definitions of functions, classes, variables, constants or any other Python objects. Contents of this file can be made available to any other program.

Creating Modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example: Here's an example of a simple module, **support.py**

```
def print_func( par ):
   print "Hello : ", par
   return
```

import Statement

- We can use any Python source file as a module by executing an import statement in some other Python source file.
- The *import* has the following syntax –

```
import module_name
```

- 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 **support.py**, you need to put the following command at the top of the script –

```
import support
support.print_func("Zara")
```

When the above code is executed, it produces the following result –

```
Hello : Zara
```

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

from...import Statement

- Python's *from* statement lets you import specific attributes from a module into the current namespace.
- The *from...import* has the following syntax –

```
from module_name import name1
```

For example, to import the function fibonacci from the module fib, use the following statement –

```
from fib import fibonacci
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

from...import * Statement

- It is also possible to import all names from a module into the current namespace by using the following import statement –

```
from modname import *
```

- This provides an easy way to import all the items from a module into the current namespace.

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences

1. The current directory.
2. If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
3. If all else 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. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The 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.

Here is a typical PYTHONPATH from a Windows system –

```
set PYTHONPATH = c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system –

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

Namespaces and Scoping

- 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*
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is 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 VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.
- For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

```
Money = 2000
       def AddMoney():
              Money = Money + 1
              Print("Value of money in function"+ Money)
AddMoney()
Print("value of money outside the function"+Money)
```

Output

Value of money in function 2001

Value of money outside the function 2000

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. Following is a simple example −

```
import math

content = dir(math)
print content
```

When the above code is executed, it produces the following result −

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

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 this –

```
reload(module_name)
```

Here, *module_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following –

```
reload(hello)
```

Packages in Python

- A package is basically a directory with Python files and a file with the name **init**.py. This means that every directory inside of the Python path, which contains a file named **init**.py, will be treated as a package by Python. It's possible to put several modules into a Package.
- A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.
- Packages are a way of structuring Python's module namespace by using "dotted module names". A.B stands for a submodule named B in a package named A. Two different packages like P1 and P2 can both have modules with the same name, let's say A, for example. The submodule A of the package P1 and the submodule A of the package P2 can be totally different. A package is imported like a "normal" module.

Here we are creating two packages a.py and b.py with two functions bar() and foo() respectively.

The content of a.py:

```
def bar():
    print("Hello, function 'bar' from module 'a' calling")
```

The content of b.py:

```
def foo():
    print("Hello, function 'foo' from module 'b' calling")
```

Now we are importing a.py and b.py into our program from the python code simple_package and calling functions bar() and foo() using package name

```
from simple_package import a, b
a.bar()
b.foo()
```

Output:

```
Hello, function 'bar' from module 'a' calling
Hello, function 'foo' from module 'b' calling
```

Text Files in Python

- Text files are normal files that contain the English alphabets. We call the content present in the files as text.
- Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.

File Accessing Modes

Whenever we are working with the files in *Python*, we have to mention the accessing mode of the file. For example, if you want to open a file to write something in it, then it's a type of mode. Like the same way, we have different accessing modes.

Read Only - r

In this mode, we can only read the contents of the file. If the file doesn't exist, then we will get an error.

Read and Write - r+

In this mode, we can read the contents of the file, and we can also write data to the file. If the file doesn't exist, then we will get an error.

Write Only - w

In this mode, we can write content to the file. Data present in the file will be overridden. If the file doesn't exist, then it will create a new file.

Append Only - a

In this mode, we can append data to the file at the end. If the file doesn't exist, then it will create a new file.

Append and Write - a+

In this mode, we can append and write data to the file. If the file doesn't exist, then it will create a new file.

Writing to a File

Let's see how to write data to a file.

- Open a file using the **open()** in **w** mode. If you have to read and write data using a file, then open it in an **r+** mode.
- Write the data to the file using **write()** or **writelines()** method
- Close the file.

```
file = open('sample.txt', 'w')
rator as an argument
file.write("I am a Python programmer.\nI am happy.")
file.close()
```

Reading from a File

We have seen a method to write data to a file. Let's examine how to read the data which we have written to the file.

- Open a file using the **open()** in **r** mode. If you have to read and write data using a file, then open it in an **r+** mode.
- Read data from the file using **read()** or **readline()** or **readlines()** methods. Store the data in a variable.
- Display the data.
- Close the file.

We have the following code to achieve our goal.

Example

```
file = open('sample.txt', 'r')
is number of lines
data = file.read()
print(data)
file.close()
```

Output

If you run the above program, you will get the following results.

```
I am a Python programmer.
I am happy.
```

Renaming and Deleting Files in Python

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

The rename() Method

The rename() method takes two arguments, the current filename and the new filename.

Syntax

```
os.rename(current_file_name, new_file_name)
```

Example

Following is the example to rename an existing file test1.txt −

```
import os

os.rename( "test1.txt", "test2.txt" )
```

The remove() Method

You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

```
os.remove(file_name)
```

Example : Following is the example to delete an existing file test2.txt −

```
import os
os.remove("text2.txt")
```

Exception Handling in Python

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

The keywords used to handle exception

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The else keyword to define a block of code to be executed if no errors were raised:
- The finally block lets you execute code, regardless of the result of the try- and except blocks

List of Standard Exceptions −

Sr.No.	Exception Name & Description
1	**Exception** : Base class for all exceptions
2	**StopIteration** :Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** : Raised by the sys.exit() function.
4	**StandardError** :Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError**:Base class for all errors that occur for numeric calculation.
6	**OverflowError**:Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** .Base class for all lookup errors.

15	**IndexError** Raised when an index is not found in a sequence.
16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
21	**IOError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.

Syntax

Here is simple syntax of *try....except...else* blocks –

```
try:
   You do your 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.
```

Example

```
try:
  a=int(input()
  b=int(input())
  div=a/b

except ArithmeticError:
   print("Divistion with zero is not possible")
else:
   print(div)
```

Output without exception:

10
2
5

Output With Exception

10
0
Division with zero is not possible

Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```
try:
   You do your operations here;
   ......................
except ExceptionType, Argument:
   You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement.

Example

Following is an example for a single exception –

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

# Call above function here.
temp_convert("xyz");
```

This produces the following result –

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows –

```
def functionName( level ):
   if level < 1:
      raise "Invalid level!", level
      # The code below to this would not be executed
      # if we raise the exception
```

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class *Networkerror*.

```
class Networkerror(RuntimeError):
   def __init__(self, arg):
      self.args = arg
```

So once you defined above class, you can raise the exception as follows –

```
try:
   raise Networkerror("Bad hostname")
except Networkerror,e:
   print e.args
```

Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The **syntax** for assert is −

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. 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

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature −

```
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32
print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

When the above code is executed, it produces the following result −

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

MODULE 4

Module IV: GUI Programming- Introduction – Tkinter Widgets – Label – Message Widget – Entry Widget – Text Widget – tk Message Box – Button Widget – Radio Button- Check Button – Listbox- Frames _ Toplevel Widgets – Menu Widget

Python GUI

- Python offers multiple options for developing GUI (Graphical User Interface). Out of all the GUI methods, tkinter is the most commonly used method. It is a standard Python interface to the Tk GUI toolkit shipped with Python.
- Python with tkinter is the fastest and easiest way to create the GUI applications. Creating a GUI using tkinter is an easy task.

To create a tkinter app:

1. Importing the module – tkinter
2. Create the main window (container)
3. Add any number of widgets to the main window
4. Apply the event Trigger on the widgets.

- Importing tkinter is same as importing any other module in the Python code

import tkinter

- There are two main methods used which the user needs to remember while creating the Python application with GUI.

1. **Tk(screenName=None, baseName=None, className='Tk', useTk=1):**

- To create a main window, tkinter offers a method *'Tk(screenName=None, baseName=None, className='Tk', useTk=1)'*.
- To change the name of the window, you can change the className to the desired one. The basic code used to create the main window of the application is:

m=tkinter.Tk() where m is the name of the main window object

2. **mainloop():**
   There is a method known by the name mainloop() is used when your application is ready to run. mainloop() is an infinite loop used to run the application, wait for an event to occur and process the event as long as the window is not closed.

m.mainloop()

Example

```
#!/usr/bin/python

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

This would create a following window −

Geometry Management

All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- The *pack()* Method − This geometry manager organizes widgets in blocks before placing them in the parent widget.
- The *grid()* Method − This geometry manager organizes widgets in a table-like structure in the parent widget.
- The *place()* Method − This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

Tkinter is implemented as a Python wrapper for the Tcl Interpreter embedded within the interpreter of Python. Tk provides the following widgets:

- button
- canvas
- combo-box
- frame
- level
- check-button
- entry
- level-frame
- menu
- list - box
- menu button
- message
- tk_optoinMenu
- progress-bar
- radio button
- scroll bar
- separator
- tree-view, and many more.

Tkinter Label

This widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time you want.

It is also possible to underline part of the text (like to identify a keyboard shortcut) and span the text across multiple lines

Syntax

Here is the simple syntax to create this widget −

```
w = Label ( master, option, ... )
```

Parameters

- **master** – This represents the parent window.
- **options** – Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.Commanly used options are height,width,font,bgcolor etc

Example

Try the following example yourself –

```
from Tkinter import *

root = Tk()
var = StringVar()
label = Label( root, textvariable=var)

var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

When the above code is executed, it produces the following result –

Python - Tkinter Message

This widget provides a multiline and noneditable object that displays texts, automatically breaking lines and justifying their contents.

Its functionality is very similar to the one provided by the Label widget, except that it can also automatically wrap the text, maintaining a given width or aspect ratio.

Syntax

Here is the simple syntax to create this widget –

w = Message (master, option, ...)

Example

Try the following example yourself –

```
from Tkinter import *

root = Tk()
var = StringVar()
label = Message( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

When the above code is executed, it produces the following result –

Tkinter Entry

The Entry widget is used to accept single-line text strings from a user.

- If you want to display multiple lines of text that can be edited, then you should use the *Text* widget.
- If you want to display one or more lines of text that cannot be modified by the user, then you should use the *Label* widget.

Syntax

Here is the simple syntax to create this widget –

```
w = Entry( master, option, ... )
```

Example

Try the following example yourself –

```
from Tkinter import *

top = Tk()
L1 = Label(top, text="User Name")
L1.pack( side = LEFT)
E1 = Entry(top, bd =5)
E1.pack(side = RIGHT)

top.mainloop()
```

When the above code is executed, it produces the following result −

Python - Tkinter Text

Text widgets provide advanced capabilities that allow you to edit a multiline text and format the way it has to be displayed, such as changing its color and font.

You can also use elegant structures like tabs and marks to locate specific sections of the text, and apply changes to those areas. Moreover, you can embed windows and images in the text because this widget was designed to handle both plain and formatted text.

Syntax

Here is the simple syntax to create this widget −

```
w = Text ( master, option, ... )
```

Example

Try the following example yourself −

```
from Tkinter import *

def onclick():
   pass
root = Tk()
text = Text(root)
text.insert(INSERT, "Hello.....")
text.insert(END, "Bye Bye.....")
text.pack()

text.tag_add("here", "1.0", "1.4")
text.tag_add("start", "1.8", "1.13")
text.tag_config("here", background="yellow", foreground="blue")
text.tag_config("start", background="black", foreground="green")
root.mainloop()
```

Tkinter tkMessageBox

The tkMessageBox module is used to display message boxes in your applications. This module provides a number of functions that you can use to display an appropriate message.

Some of these functions are showinfo, showwarning, showerror, askquestion, askokcancel, askyesno, and askretryignore.

Syntax

Here is the simple syntax to create this widget −

```
tkMessageBox.FunctionName(title, message [, options])
```

Parameters

- FunctionName − This is the name of the appropriate message box function.
- title − This is the text to be displayed in the title bar of a message box.
- message − This is the text to be displayed as a message.
- options − options are alternative choices that you may use to tailor a standard message box. You could use one of the following functions with dialogue box −

- showinfo()
- showwarning()
- showerror ()
- askquestion()
- askokcancel()
- askyesno ()
- askretrycancel ()

Example

Try the following example yourself −

```
import Tkinter
import tkMessageBox

top = Tkinter.Tk()
def hello():
   tkMessageBox.showinfo("Say Hello", "Hello World")

B1 = Tkinter.Button(top, text = "Say Hello", command = hello)
B1.pack()

top.mainloop()
```

When the above code is executed, it produces the following result –

Python - Tkinter Button

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

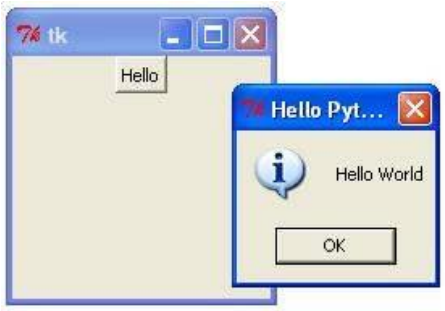
Syntax : Here is the simple syntax to create this widget –

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

Example

```
import Tkinter
import tkMessageBox

top = Tkinter.Tk()

def helloCallBack():
   tkMessageBox.showinfo( "Hello Python", "Hello World")

B = Tkinter.Button(top, text ="Hello", command = helloCallBack)

B.pack()
top.mainloop()
```

When the above code is executed, it produces the following result –



Python - Tkinter Radiobutton

This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.

In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another.

Syntax

Here is the simple syntax to create this widget –

```
w = Radiobutton ( master, option, ...  )
```

Example

```
from Tkinter import *

def sel():
   selection = "You selected the option " + str(var.get())
   label.config(text = selection)

root = Tk()
var = IntVar()
R1 = Radiobutton(root, text="Option 1", variable=var, value=1,
                 command=sel)
R1.pack( anchor = W )

R2 = Radiobutton(root, text="Option 2", variable=var, value=2,
                 command=sel)
R2.pack( anchor = W )

R3 = Radiobutton(root, text="Option 3", variable=var, value=3,
                 command=sel)
R3.pack( anchor = W)

label = Label(root)
label.pack()
root.mainloop()
```

When the above code is executed, it produces the following result –

Python - Tkinter Checkbutton

The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option.

You can also display images in place of text.

Syntax

```
w = Checkbutton ( master, option, ... )
```

Example

```
from Tkinter import *
import tkMessageBox
import Tkinter

top = Tkinter.Tk()
CheckVar1 = IntVar()
CheckVar2 = IntVar()
C1 = Checkbutton(top, text = "Music", variable = CheckVar1, \
                 onvalue = 1, offvalue = 0, height=5, \
                 width = 20)
C2 = Checkbutton(top, text = "Video", variable = CheckVar2, \
                 onvalue = 1, offvalue = 0, height=5, \
                 width = 20)
C1.pack()
C2.pack()
top.mainloop()
```

When the above code is executed, it produces the following result −

Tkinter Listbox

The Listbox widget is used to display a list of items from which a user can select a number of items.

Syntax

```
w = Listbox ( master, option, ... )
```

Example

```
from Tkinter import *
import tkMessageBox
import Tkinter

top = Tk()

Lb1 = Listbox(top)
Lb1.insert(1, "Python")
Lb1.insert(2, "Perl")
Lb1.insert(3, "C")
Lb1.insert(4, "PHP")
Lb1.insert(5, "JSP")
Lb1.insert(6, "Ruby")

Lb1.pack()
top.mainloop()
```

When the above code is executed, it produces the following result −

Tkinter Frame

The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets.

It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets.

Syntax

```
w = Frame ( master, option, ... )
```

Example

```
from Tkinter import *

root = Tk()
frame = Frame(root)
frame.pack()

bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )

redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)

greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )

bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )

blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)

root.mainloop()
```

When the above code is executed, it produces the following result −

Toplevel Widget:-Tkinter Menu

The goal of this widget is to allow us to create all kinds of menus that can be used by our applications. The core functionality provides ways to create three menu types: pop-up, toplevel and pull-down.

It is also possible to use other extended widgets to implement new types of menus, such as the *OptionMenu* widget, which implements a special type that generates a pop-up list of items within a selection.

Syntax

Here is the simple syntax to create this widget −

```
w = Menu ( master, option, ... )
```

Example

Try the following example yourself −

```
from Tkinter import *

def donothing():
   filewin = Toplevel(root)
   button = Button(filewin, text="Do nothing button")
   button.pack()

root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...", command=donothing)
filemenu.add_command(label="Close", command=donothing)

filemenu.add_separator()

filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=donothing)

editmenu.add_separator()

editmenu.add_command(label="Cut", command=donothing)
editmenu.add_command(label="Copy", command=donothing)
editmenu.add_command(label="Paste", command=donothing)
editmenu.add_command(label="Delete", command=donothing)
editmenu.add_command(label="Select All", command=donothing)

menubar.add_cascade(label="Edit", menu=editmenu)
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Help Index", command=donothing)
helpmenu.add_command(label="About...", command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)

root.config(menu=menubar)
root.mainloop()
```