https://pythonschoolkvs.wordpress.com/

CLASS-XI

SUBJECT: COMPUTER SCIENCE (083) – PYTHON

INDEX

By: Vikash Kumar Yadav

PGT-Computer Science

K.V. No.-IV ONGC Vadodara

CHAPTER-1

INTRODUTION TO PYTHON

1.1 Introduction:

- General-purpose Object Oriented Programming language.
- High-level language
- Developed in late 1980 by **Guido van Rossum** at National Research Institute for Mathematics and Computer Science in the Netherlands.
- It is derived from programming languages such as ABC, Modula 3, small talk, Algol-68.
- It is Open Source Scripting language.
- It is **Case-sensitive language** (Difference between uppercase and lowercase letters).
- One of the official languages at Google.

1.2 Characteristics of Python:

- Interpreted: Python source code is compiled to byte code as a **.pyc** file, and this byte code can be interpreted by the interpreter.
- Interactive
- Object Oriented Programming Language
- Easy & Simple
- Portable
- Scalable: Provides improved structure for supporting large programs.
- Integrated
- Expressive Language

1.3 Python Interpreter:

Names of some Python interpreters are:

- PyCharm
- Python IDLE
- The Python Bundle
- pyGUI
- Sublime Text etc.

There are **two modes** to use the python interpreter:

i. Interactive Mode
ii. Script Mode

i. **Interactive Mode**: Without passing python script file to the interpreter, directly execute code to Python (Command line).

Example:
>>>6+3

Output: 9

Fig: Interactive Mode

Note: **>>>** is a command the python interpreter uses to indicate that it is ready. The interactive mode is better when a programmer deals with small pieces of code.
To run a python file on command line:
exec(open("C:\Python33\python programs\program1.py").read( ))

ii. **Script Mode:** In this mode source code is stored in a file with the **.py** extension and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file.

Example:
 if you have a file name **Demo.py** , to run the script you have to follow the following steps:

Step-1: Open the text editor i.e. Notepad
Step-2: Write the python code and save the file with .py file extension. (Default directory is *C:\Python33/Demo.py*)
Step-3: Open IDLE (Python GUI) python shell
Step-4: Click on file menu and select the open option
Step-5: Select the existing python file
Step-6: Now a window of python file will be opened
Step-7: Click on Run menu and the option Run Module.

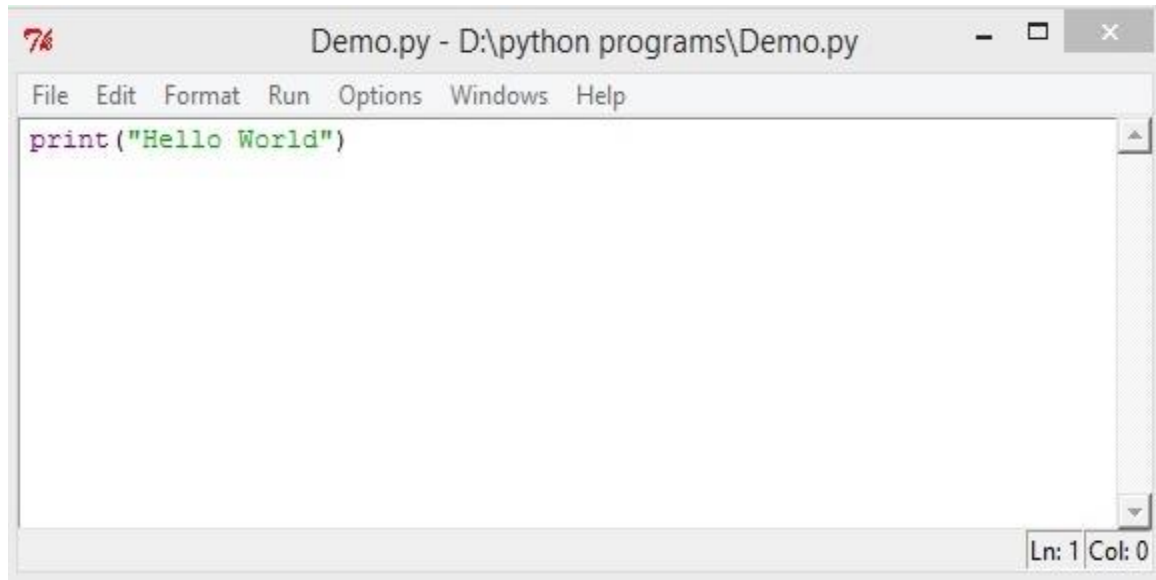**Step-8:** Output will be displayed on python shell window.
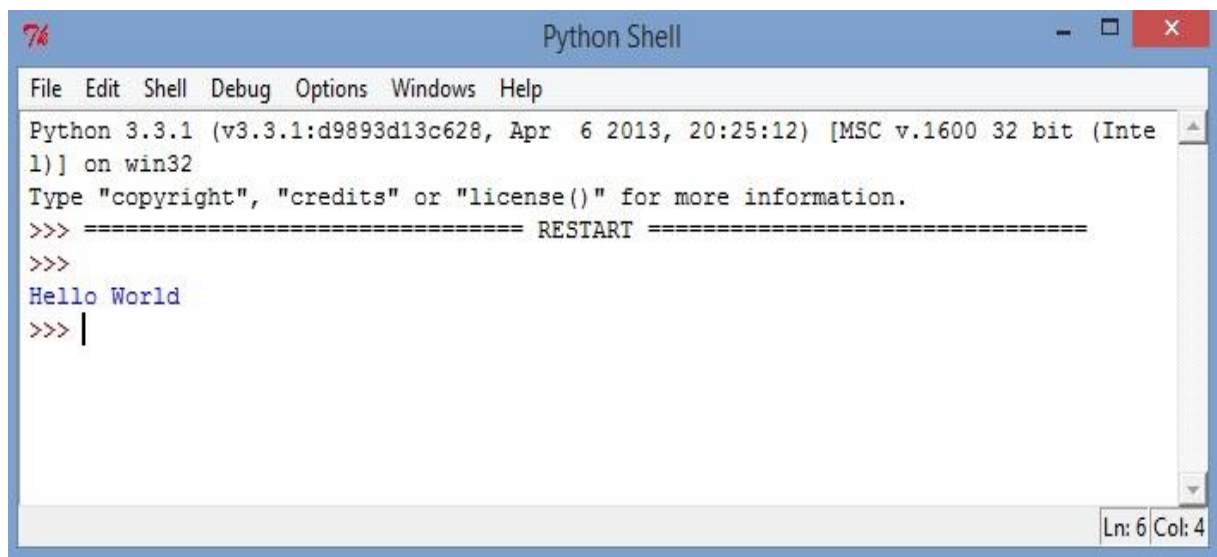

Fig. : IDLE (Python GUI)

Fig: Python Shell

CHAPTER-2
PYTHON FUNDAMENTALS

2.1 Python Character Set :

It is a set of valid characters that a language recognize.

Letters: A-Z, a-z

Digits : 0-9

Special Symbols

Whitespace

2.2 TOKENS

Token: Smallest individual unit in a program is known as token.

There are five types of token in python:

1. Keyword
2. Identifier
3. Literal
4. Operators
5. Punctuators

1. **Keyword**: Reserved words in the library of a language. There are 33 keywords in python.

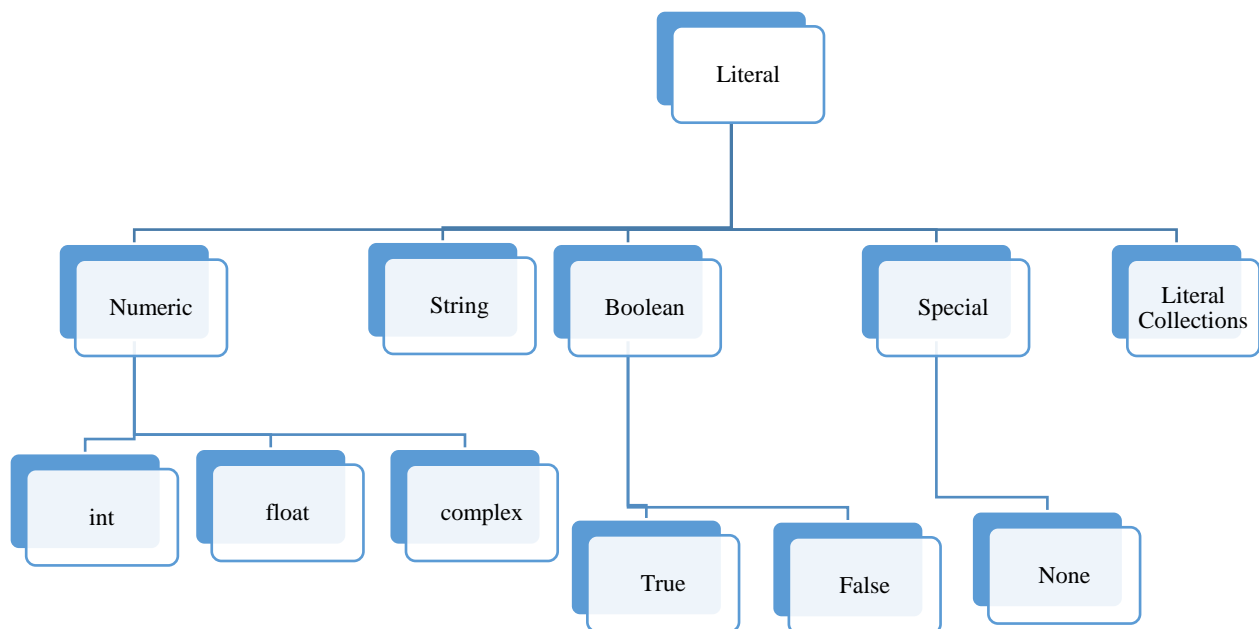| False	class	finally	is	return	break
None	continue	for	lambda	try	except
True	def	from	nonlocal	while	in
and	del	global	not	with	raise
as	elif	if	or	yield	
assert	else	import	pass		

All the keywords are in lowercase except 03 keywords (True, False, None).

2. **Identifier:** The name given by the user to the entities like variable name, class-name, function-name etc.

   ### *Rules for identifiers:*

   - It can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore.
   - It cannot start with a digit.
   - Keywords cannot be used as an identifier.
   - We cannot use special symbols like !, @, #, $, %, + etc. in identifier.
   - _ (underscore) can be used in identifier.
   - Commas or blank spaces are not allowed within an identifier.

3. **Literal:** Literals are the constant value. Literals can be defined as a data that is given in a variable or constant.

```
                          Literal
    ┌──────────┬──────────┬──────────┬──────────┐
 Numeric     String    Boolean    Special   Literal
                                            Collections
 ┌──┬──┬──┐           ┌────┬────┐    │
int float complex    True False    None
```

A. **Numeric literals:** Numeric Literals are immutable.

Eg.

   5,   6.7,   6+9j

B. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes for a String.

Eg:

"Aman" , '12345'

Escape sequence characters:

\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	Backspace
\f	ASCII Formfeed
\n	New line charater
\t	Horizontal tab

C. Boolean literal: A Boolean literal can have any of the two values: **True** or **False**.

D. Special literals: Python contains one special literal i.e. **None**.

None is used to specify to that field that is not created. It is also used for end of lists in Python.

E. Literal Collections: Collections such as tuples, lists and Dictionary are used in Python.

4. Operators: An operator performs the operation on operands. Basically there are two types of operators in python according to number of operands:
   A. Unary Operator
   B. Binary Operator

A. Unary Operator: Performs the operation on one operand.

Example:

+   Unary plus

-     Unary minus

~   Bitwise complement

not   Logical negation

B. Binary Operator: Performs operation on two operands.

5. **Separator or punctuator : , ; , ( ), { }, [ ]**

2.3 Mantissa and Exponent Form:

A real number in exponent form has two parts:

> ➢ mantissa
> ➢ exponent

Mantissa : It must be either an integer or a proper real constant.

Exponent : It must be an integer. Represented by a letter E or e followed by integer value.

Valid Exponent form	Invalid Exponent form
123E05	2.3E (No digit specified for exponent)
1.23E07	0.24E3.2 (Exponent cannot have fractional part)
0.123E08	23,455E03 (No comma allowed)
123.0E08	
123E+8	
1230E04	
-0.123E-3	
163.E4	
.34E-2	
4.E3	

2.4 Basic terms of a Python Programs:

A. Blocks and Indentation

B. Statements

C. Expressions

D. Comments

A. Blocks and Indentation:

- Python provides no braces to indicate blocks of code for class and function definition or flow control.

- Maximum line length should be maximum 79 characters.

- Blocks of code are denoted by line indentation, which is rigidly enforced.

- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

for example –

```
if True:
        print("True")
else:
        print("False")
```

B. Statements

A line which has the instructions or expressions.

C. Expressions:

A legal combination of symbols and values that produce a result. Generally it produces a value.

D. Comments: Comments are not executed. Comments explain a program and make a program understandable and readable. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

There are two types of comments in python:

i. Single line comment

ii. Multi-line comment

i. **Single line comment**: This type of comments start in a line and when a line ends, it is automatically ends. Single line comment starts with # symbol.

Example: if a>b: # Relational operator compare two values

ii. **Multi-Line comment**: Multiline comments can be written in more than one lines. Triple quoted ' ' ' or " " ") multi-line comments may be used in python. It is also known as **docstring**.
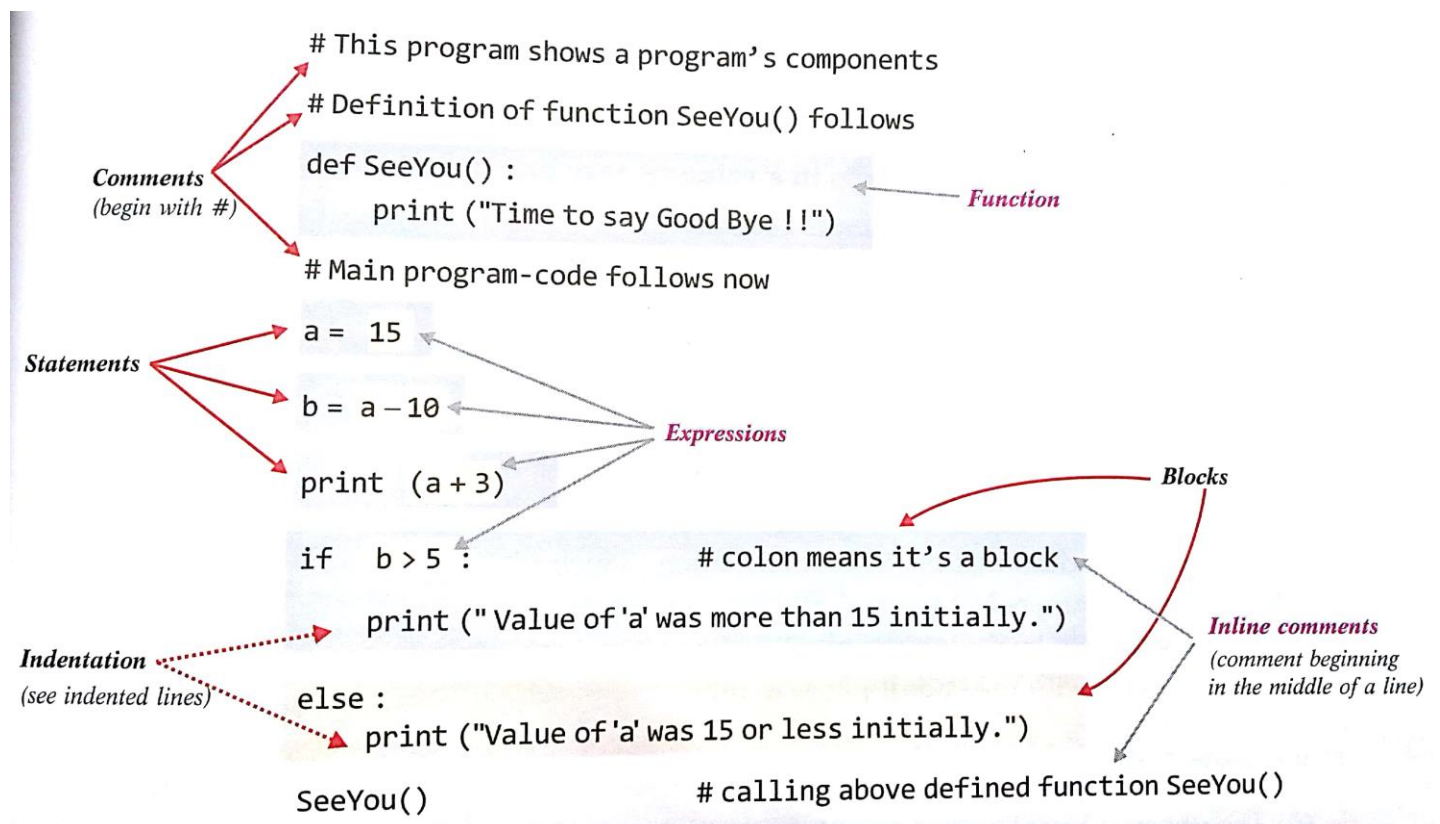
Example:

''' This program will calculate the average of 10 values.

First find the sum of 10 values

and divide the sum by number of values

'''

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block.

Example:-

   x=5; print("Value =" x)

2.5 Variable/Label in Python:

Definition: Named location that refers to a value and whose value can be used and processed during program execution.

Variables in python do not have fixed locations. The location they refer to changes every time their values change.

Creating a variable:

A variable is created the moment you first assign a value to it.

Example:

x = 5

y = "hello"

Variables do not need to be declared with any particular type and can even change type after they have been set. It is known as dynamic Typing.

```
x = 4 # x is of type int
x = "python" # x is now of type str
print(x)
```


Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscore (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Python allows assign a single value to multiple variables.

Example: x = y = z = 5

You can also assign multiple values to multiple variables. For example −

x , y , z = 4, 5, "python"

4 is assigned to x, 5 is assigned to y and string "python" assigned to variable z respectively.

x=12

y=14

x,y=y,x

print(x,y)

Now the result will be

14   12

Lvalue: the LHS part of the expression

Rvalue: the RHS part of the expression

Python first evaluates the RHS expression and then assigns to LHS.

Example:

p, q, r= 5, 10, 7

q, r, p = p+1, q+2, r-1

print (p,q,r)

Now the result will be:

6   6   12

Note: Expressions separated with commas are evaluated from left to right and assigned in same order.

❖ If you want to know the type of variable, you can use **type( ) function :**

Syntax:

type (variable-name)

Example:

x=6

type(x)

The result will be:

<class 'int'>

- ❖ If you want to know the memory address or location of the object, you can use **id( )** function.
  Example:
  >>>id(5)
  1561184448
  >>>b=5
  >>>id(b)
  1561184448

You can delete single or multiple variables by using del statement. Example:

del x

del y, z

2.6 Input from a user:

input( ) method is used to take input from the user.

Example:

print("Enter your name:")
x = input( )
print("Hello, " + x)

- ➢ input( ) function always returns a value of string type.

2.7 Type Casting:

To convert one data type into another data type.

Casting in python is therefore done using constructor functions:

- **int( )** - constructs an integer number from an integer literal, a float literal or a string literal.

  Example:

  x = int(1)   # x will be 1

  y = int(2.8) # y will be 2

  z = int("3") # z will be 3

- **float( )** - constructs a float number from an integer literal, a float literal or a string literal.

  Example:

  x = float(1)   # x will be 1.0

  y = float(2.8)   # y will be 2.8

  z = float("3")   # z will be 3.0

  w = float("4.2") # w will be 4.2

- **str( )** - constructs a string from a wide variety of data types, including strings, integer literals and float literals.

  Example:

  x = str("s1") # x will be 's1'

  y = str(2)   # y will be '2'

  z = str(3.0) # z will be '3.0'

Reading a number from a user:

x= int (input("Enter an integer number"))

2.8 OUTPUT using print( ) statement:

Syntax:

    print(object, sep=<separator string >, end=<end-string>)

object : It can be one or multiple objects separated by comma.

sep : sep argument specifies the separator character or string. It separate the objects/items. By default sep argument adds space in between the items when printing.

end : It determines the end character that will be printed at the end of print line. By default it has newline character('\n').

Example:

x=10

y=20

z=30

print(x,y,z, sep='@', end= ' ')

Output:

10@20@30

CHAPTER-3
DATA HANDLING

3.1 Data Types in Python:

Python has Two data types –

1. Primitive Data Type (Numbers, String)
2. Collection Data Type (List, Tuple, Set, Dictionary)

```
                          ┌──────────────┐
                          │  Data Types  │
                          └──────────────┘
            ┌───────────────────┴────────────────────┐
     ┌──────────────┐                          ┌──────────────┐
     │  Primitive   │                          │  Collection  │
     │  Data Type   │                          │  Data Type   │
     └──────────────┘                          └──────────────┘
       ┌──────┴──────┐              ┌────────┬──────┴──────┬──────────┐
  ┌──────────┐  ┌──────────┐   ┌──────┐ ┌───────┐  ┌──────┐  ┌────────────┐
  │  Number  │  │  String  │   │ List │ │ Tuple │  │ Set  │  │ Dictionary │
  └──────────┘  └──────────┘   └──────┘ └───────┘  └──────┘  └────────────┘
   ┌────┬──┴──┬─────────┐
┌─────┐ ┌───────┐ ┌─────────┐
│ int │ │ float │ │ complex │
└─────┘ └───────┘ └─────────┘
```
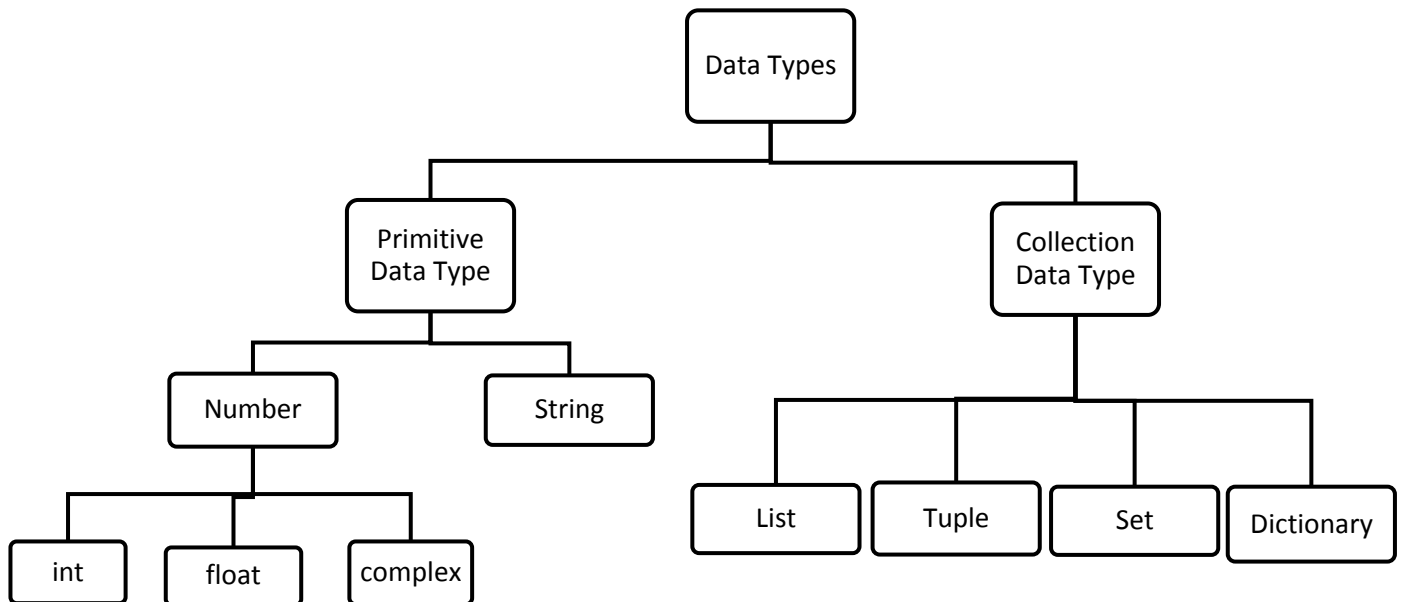
1. Primitive Data Types:

a. Numbers: Number data types store numeric values.

There are three numeric types in Python:

- int
- float
- complex

Example:

```
w = 1          # int
y = 2.8        # float
z = 1j         # complex
```

- **integer** : There are two types of integers in python:
  - ➢ int
  - ➢ Boolean

  - ➢ **int:** int or integer, is a whole number, positive or negative, without decimals.

    Example:

    x = 1
    y = 35656222554887711
    z = -3255522

  - ➢ **Boolean:** It has two values: True and False. **True** has the value **1** and **False** has the value **0**.

    Example:

    >>>bool(0)

    False

    >>>bool(1)

    True

    >>>bool(' ')

    False

    >>>bool(-34)

    True

    >>>bool(34)

    True

- **float** : float or "floating point number" is a number, positive or negative, containing one or more decimals. Float can also be scientific numbers with an "e" to indicate the power of 10.

  Example:

  x = 1.10
  y = 1.0
  z = -35.59

  a = 35e3
  b = 12E4
  c = -87.7e100

- **complex** : Complex numbers are written with a "j" as the imaginary part.

> *Example:*
>
> >>>x = 3+5j
>
> >>>y = 2+4j
>
> >>>z=x+y
>
> >>>print(z)
>
> 5+9j
>
> >>>z.real
>
> 5.0
>
> >>>z.imag
>
> 9.0

Real and imaginary part of a number can be accessed through the attributes **real** and **imag**.

b. **String:** Sequence of characters represented in the quotation marks.

- Python allows for either pairs of single or double quotes. Example: 'hello' is the same as "hello" .

- Python does not have a character data type, a single character is simply a string with a length of **1**.

- The python string store Unicode characters.

- Each character in a string has its own index.

- String is immutable data type means it can never change its value in place.

2. Collection Data Type:

- ➢ List
- ➢ Tuple
- ➢ Set
- ➢ Dictionary

3.2 MUTABLE & IMMUTABLE Data Type:

➢ **Mutable Data Type:**

These are changeable. In the same memory address, new value can be stored.

Example: List, Set, Dictionary

➢ **Immutable Data Type:**

These are unchangeable. In the same memory address new value cannot be stored.

Example: integer, float, Boolean, string and tuple.

3.3 Basic Operators in Python:

i. Arithmetic Operators

ii. Relational Operator

iii. Logical Operators

iv. Bitwise operators

v. Assignment Operators

vi. Other Special Operators

      o Identity Operators

      o Membership operators

i. **Arithmetic Operators**: To perform mathematical operations.

OPERATOR	NAME	SYNTAX	RESULT (X=14, Y=4)
+	Addition	x + y	18
_	Subtraction	x − y	10
*	Multiplication	x * y	56
/	Division (float)	x / y	3.5
//	Division (floor)	x // y	3
%	Modulus	x % y	2
**	Exponent	x**y	38416

Example:
```
>>>x= -5
>>>x**2
>>> -25
```

ii. <mark>**Relational**</mark> <mark>**Operators:**</mark> Relational operators compare the values. It either returns **True** or **False** according to the condition.

OPERATOR	NAME	SYNTAX	RESULT (IF X=16, Y=42)
>	Greater than	x > y	False
<	Less than	x < y	True
==	Equal to	x == y	False
!=	Not equal to	x != y	True
>=	Greater than or equal to	x >= y	False
<=	Less than or equal to	x <= y	True

iii<mark>. Logical operator</mark>s: Logical operators perform **Logical AND**, **Logical OR** and **Logical NOT** operations.

OPERATOR	DESCRIPTION	SYNTAX
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

Examples of Logical Operator:

<mark>The **and** operator:</mark> The and operator works in two ways:

        a. Relational expressions as operands

        b. numbers or strings or lists as operands

a. Relational expressions as operands:

X	Y	X and Y
False	False	False
False	True	False
True	False	False
True	True	True

```
>>> 5>8 and 7>3
False
>>> (4==4) and (7==7)
True
```

b. numbers or strings or lists as operands:

In an expression X and Y, if first operand has **false value**, **then return first operand X** as a result, otherwise returns Y.

```
>>>0 and 0
0
>>>0 and 6
0
>>>'a' and 'n'
'n'
```

X	Y	X and Y
false	false	X
false	true	X
true	false	Y
true	true	Y

```
>>>6>9 and 'c'+9>5   # and operator will test the second operand only if the first operand
False                # is true, otherwise ignores it, even if the second operand is wrong
```

The **or** operator: The or operator works in two ways:
    a. Relational expressions as operands
    b. numbers or strings or lists as operands

a. Relational expressions as operands:

X	Y	X or Y
False	False	False
False	True	True
True	False	True
True	True	True

```
>>> 5>8 or 7>3
True
>>> (4==4) or (7==7)
True
```

b. numbers or strings or lists as operands:

In an expression X or Y, if first operand has **true value**, **then return first operand X** as a result, otherwise returns Y.

X	Y	X or Y
false	false	Y
false	true	Y
true	false	X
true	true	X

```
>>>0 or 0
0
>>>0 or 6
6
>>>'a' or 'n'
'a'
>>>6<9 or 'c'+9>5   # or operator will test the second operand only if the first operand
True                # is false, otherwise ignores it, even if the second operand is wrong
```

The **not** operator:
```
>>>not 6
False
>>>not 0
True
>>>not -7
False
```

Chained Comparison Operators:
```
>>> 4<5>3      is equivalent to    >>> 4<5 and 5>3
True                                True
```

iv. **Bitwise operators:** Bitwise operators acts on bits and performs bit by bit operation.

OPERATOR	DESCRIPTION	SYNTAX
&	Bitwise AND	x & y
|	Bitwise OR	x | y

~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Examples:
Let
a = 10
b = 4

print(a & b)

print(a | b)

print(~a)

print(a ^ b)
print(a >> 2)

print(a << 2)

```
Output:
0
14
-11
14
2
40
```

v. **Assignment operators:** Assignment operators are used to assign values to the variables.

OPERA TOR	DESCRIPTION	SYNTAX
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right side operand with left side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a*=b a=a*b

/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<=b a= a << b

vi. **Other Special operators:** There are some special type of operators like-

 a. **Identity operators- is** and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

is	True if the operands are identical
is not	True if the operands are not identical

Example:
Let
a1 = 3
b1 = 3
a2 = 'PythonProgramming'
b2 = 'PythonProgramming'
a3 = [1,2,3]
b3 = [1,2,3]

print(a1 is not b1)

```
print(a2 is b2)           # Output is False, since lists are mutable.
print(a3 is b3)
```

Output:

False

True

False

Example:

```
>>>str1= "Hello"

>>>str2=input("Enter a String :")

Enter a String : Hello

>>>str1==str2      # compares values of string

True

>>>str1 is str2        # checks if two address refer to the same memory address

False
```

    b. <mark>**Membership operators-**</mark> **in** and **not in** are the membership operators; used to test whether a value or variable is in a sequence.

in	True if value is found in the sequence
not in	True if value is not found in the sequence

Example:
```
Let
x = 'Digital India'
y = {3:'a',4:'b'}


print('D' in x)

print('digital' not in x)

print('Digital' not in x)

print(3 in y)

print('b' in y)
```

Output:

True

True

False

True

False

3.4 Operator Precedence and Associativity:

Operator Precedence: It describes the order in which operations are performed when an expression is evaluated. Operators with higher **precedence** perform the operation first.

Operator Associativity: whenever two or more operators have the same precedence, then associativity defines the order of operations.

Operator	Description	Associativity	Precedence
( ), { }	Parentheses (grouping)	Left to Right	
f(args…)	Function call	Left to Right	
x[index:index]	Slicing	Left to Right	
x[index]	Subscription	Left to Right	
**	Exponent	**Right to Left**	
~x	Bitwise not	Left to Right	
+x, -x	Positive, negative	Left to Right	
*, /, %	Product, division, remainder	Left to Right	
+, −	Addition, subtraction	Left to Right	
<<, >>	Shifts left/right	Left to Right	
&	Bitwise AND	Left to Right	
^	Bitwise XOR	Left to Right	
|	Bitwise OR	Left to Right	
<=, <, >, >=	Comparisons	Left to Right	
=, %=, /=, +=	Assignment		
is, is not	Identity		
in, not in	Membership		
not	Boolean NOT	Left to Right	
and	Boolean AND	Left to Right	
or	Boolean OR	Left to Right	
lambda	Lambda expression	Left to Right	

CHAPTER-4

CHAPTER-4
FLOW OF CONTROL

**1. Decision Making and branching (Conditional Statement)**

2. Looping or Iteration

3. Jumping statements

4.1 DECISION MAKING & BRANCHING

Decision making is about deciding the order of execution of statements based on certain conditions. Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome.

if else Statement- Flowchart

Entry

[False]   Test_Condition ?   [True]

False Statement-block;   True Statement-block;

Statement-X;
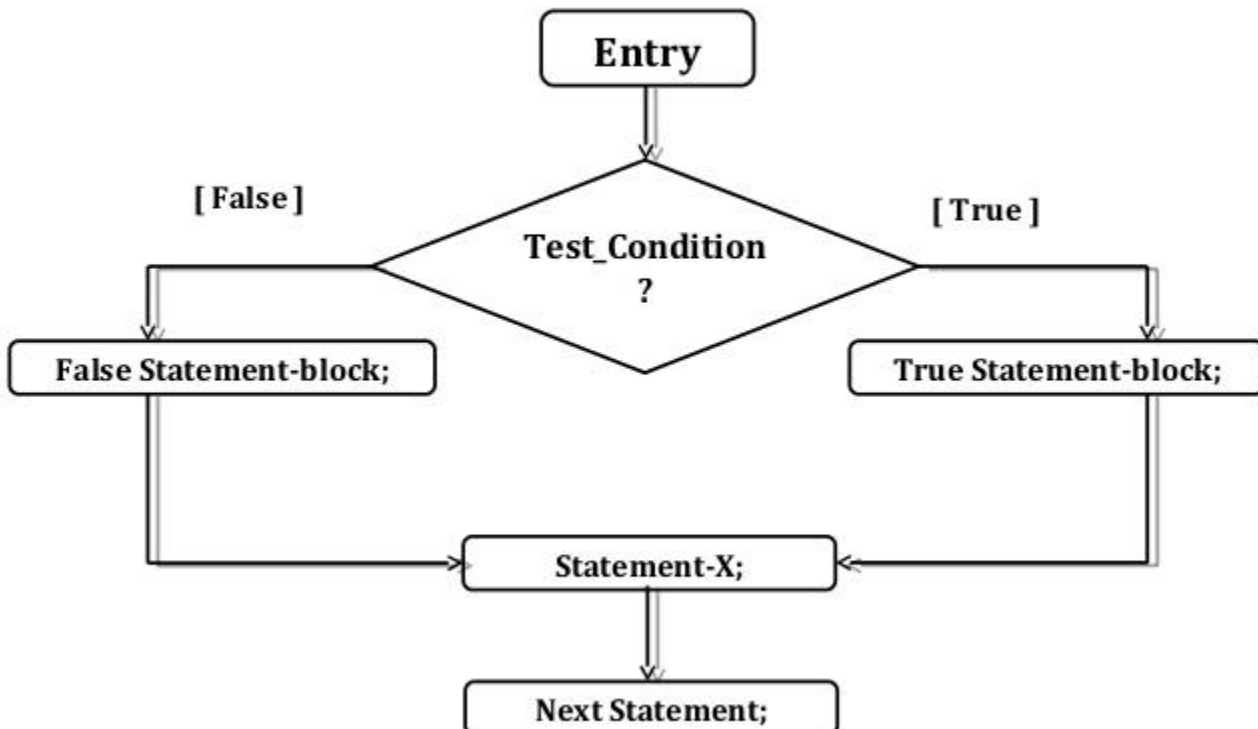
Next Statement;

There are three types of conditions in python:

1. if statement

2. if-else statement

3. elif statement

1. **if statement:** It is a simple if statement. When condition is true, then code which is associated with if statement will execute.

Example:

a=40

b=20

if a>b:

    print("a is greater than b")

2. **if-else statement:** When the condition is true, then code associated with if statement will execute, otherwise code associated with else statement will execute.

Example:

a=10

b=20

if a>b:

    print("a is greater")

else:

    print("b is greater")

3. **elif statement:** It is short form of else-if statement. If the previous conditions were not true, then do this condition". It is also known as nested if statement.

Example:

a=input("Enter first number")

b=input("Enter Second Number:")

if a>b:

```
    print("a is greater")
elif a==b:
    print("both numbers are equal")
else:
    print("b is greater")
```
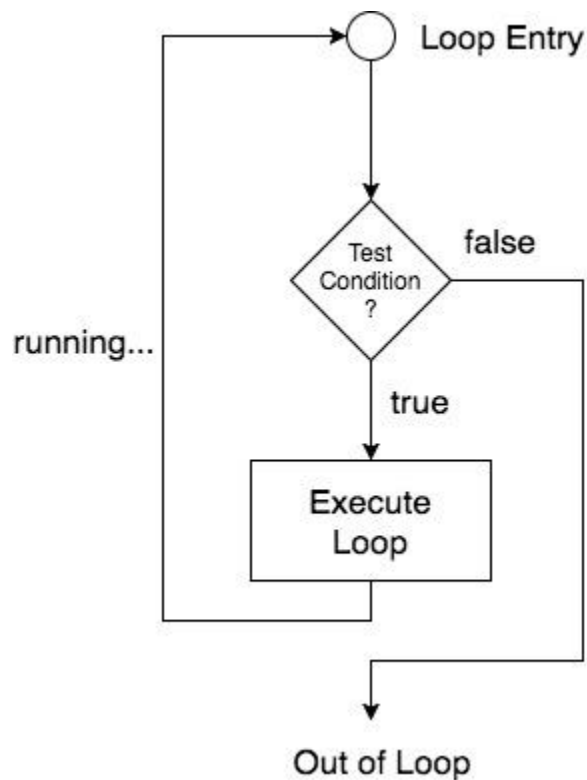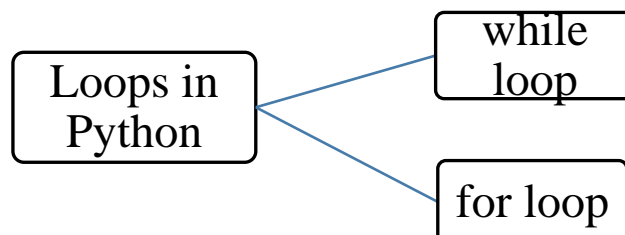
4.2 LOOPS in PYTHON

Loop: Execute a set of statements repeatedly until a particular condition is satisfied.

There are two types of loops in python:

1. while loop

2. for loop

1. **while loop:** With the while loop we can execute a set of statements as long as a condition is true. It requires to define an indexing variable.

Example: To print table of number 2

i=2

while i<=20:

    print(i)

    i+=2

2. **for loop** : The for loop iterate over a given sequence (it may be list, tuple or string).

Note: The for loop does not require an indexing variable to set beforehand, as the for command itself allows for this.

primes = [2, 3, 5, 7]

for x in primes:

   print(x)

The range( ) function:

it generates a list of numbers, which is generally used to iterate over with for loop. range( ) function uses three types of parameters, which are:

- start: Starting number of the sequence.
- stop: Generate numbers up to, but not including last number.
- step: Difference between each number in the sequence.

Python use range( ) function in three ways:

a. range(stop)

b. range(start, stop)

c. range(start, stop, step)

Note:

- ➢ All parameters must be integers.
- ➢ All parameters can be positive or negative.

a. range(stop): By default, It starts from 0 and increments by 1 and ends up to stop, but not including **stop** value.

Example:

for x in range(4):
  print(x)

Output:

0
1
2
3

b. range(start, stop): It starts from the **start** value and up to stop, but not including stop value.

Example:

for x in range(2, 6):
  print(x)

Output:

2
3
4
5

c. range(start, stop, step): Third parameter specifies to increment or decrement the value by adding or subtracting the value.

Example:

for x in range(3, 8, 2):

  print(x)

Output:

3
5
7

Explanation of output: 3 is starting value, 8 is stop value and 2 is step value. First print 3 and increase it by 2, that is 5, again increase is by 2, that is 7. The output can't exceed stop-1 value that is 8 here. So, the output is 3, 5, 8.

Difference between range( ) and xrange( ):

S. No.	range( )	xrange( )
1	returns the list of numbers	returns the generator object that can be used to display numbers only by looping
2	The variable storing the range takes more memory	variable storing the range takes less memory
3	all the operations that can be applied on the list can be used on it	operations associated to list cannot be applied on it
4	slow implementation	faster implementation

4.3 JUMP STATEMENTS:

There are two jump statements in python:

1. break

2. continue

1. **break statement** : With the break statement we can stop the loop even if it is true.

Example:

in while loop	in for loop
| ```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
``` | ```
languages = ["java", "python", "c++"]
for x in languages:
  if x == "python":
    break
  print(x)
``` |
| **Output:**
1
2
3 | **Output:**
java |

Note: If the **break** statement appears in a nested loop, then it will terminate the very loop it is in i.e. if the **break** statement is inside the inner loop then it will terminate the inner loop only and the outer loop will continue as it is.

2. continue statement : With the continue statement we can stop the current iteration, and continue with the next iteration.

Example:

in while loop	in for loop

``python
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
``
	``python
	languages = ["java", "python", "c++"]
	for x in languages:
	  if x == "python":
	    continue
	  print(x)
	``

Output: 1 2 4 5 6	**Output:** java c++

4.4 Loop else statement:

The **else** statement of a python loop executes when the loop terminates normally. The else statement of the loop will not execute when the **break** statement terminates the loop.

The else clause of a loop appears at the same indentation as that of the loop keyword **while** or **for**.

Syntax:

for loop	while loop

``
for <variable> in <sequence>:
    statement-1
    statement-2
    .
    .
else:
    statement(s)
``
	``
	while <test condition>:
	    statement-1
	    statement-2
	    .
	    .
	else:
	    statement(s)
	``

4.5 Nested Loop :

A loop inside another loop is known as nested loop.

Syntax:

for <variable-name> in <sequence>:

      for <variable-name> in <sequence>:

            statement(s)

      statement(s)

Example:

for i in range(1,4):

      for j in range(1,i):

            print("*", end=" ")

      print(" ")

==**Programs related to Conditional, looping and jumping statements**==

1. Write a program to check a number whether it is even or odd.

```
num=int(input("Enter the number: "))
if num%2==0:
   print(num, " is even number")
else:
   print(num, " is odd number")
```


2. Write a program in python to check a number whether it is prime or not.

```
num=int(input("Enter the number: "))
for i in range(2,num):
   if num%i==0:
      print(num, "is not prime number")
      break;
else:
   print(num,"is prime number")
```

3. Write a program to check a year whether it is leap year or not.

```python
year=int(input("Enter the year: "))
if year%100==0 and year%400==0:
    print("It is a leap year")
elif year%4==0:
    print("It is a leap year")
else:
    print("It is not leap year")
```

4. Write a program in python to convert °C to °F and vice versa.

```python
a=int(input("Press 1 for C to F \n Press 2 for F to C \n"))
if a==1:
        c=float(input("Enter the temperature in degree celcius:  "))
        f= (9/5)*c+32
        print(c, "Celcius = ",f," Fahrenheit")
elif a==2:
        f=float(input("Enter the temperature in Fahrenheit:  "))
        c= (f-32)*5/9
        print(f, "Fahrenheit = ",c," Celcius")
else:
        print("You entered wrong choice")
```

5. Write a program to check a number whether it is palindrome or not.

```python
num=int(input("Enter a number : "))
n=num
res=0
while num>0:
    rem=num%10
```

```
        res=rem+res*10
        num=num//10
    if res==n:
        print("Number is Palindrome")
    else:
        print("Number is not Palindrome")
```

6. A number is Armstrong number or not.

```
    num=input("Enter a number : ")
    length=len(num)
    n=int(num)
    num=n
    sum=0
    while n>0:
        rem=n%10
        sum=sum+rem**length
        n=n//10
    if num==sum:
        print(num, "is armstrong number")
    else:
        print(num, "is not armstrong number")
```

7. To check whether the number is perfect number or not

```
    num=int(input("Enter a number : "))
    sum=0
    for i in range(1,num):
        if(num%i==0):
            sum=sum+i
```

```
if num==sum:
    print(num, "is perfect number")
else:
    print(num, "is not perfect number")
```

8. Write a program to print Fibonacci series.

```
n=int(input("How many numbers : "))
first=0
second=1
i=3
print(first, second, end=" ")
while i<=n:
    third=first+second
    print(third, end=" ")
    first=second
    second=third
    i=i+1
```

9. To print a pattern using nested loops

```
for i in range(1,5):                    1
    for j in range(1,i+1):              1  2
        print(j," ", end=" ")          1  2  3
    print('\n')                         1  2  3  4
```
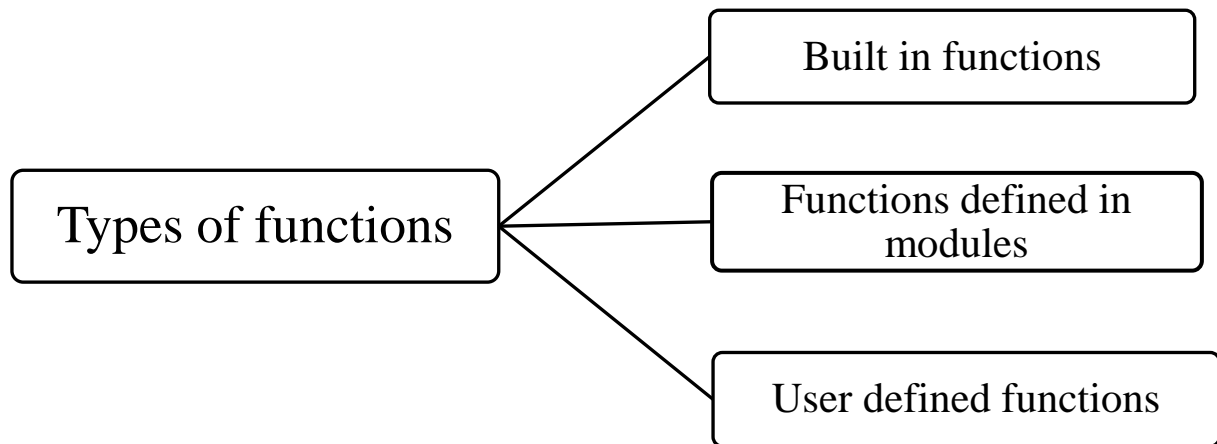
CHAPTER-5
FUNCTIONS IN PYTHON

5.1 Definition: Functions are the subprograms that perform specific task. Functions are the small modules.

5.2 Types of Functions:

There are **two** types of functions in python:

1. Library Functions (Built in functions)
2. Functions defined in modules
3. User Defined Functions

```
                                          ┌─────────────────────────┐
                                          │    Built in functions   │
                                          └─────────────────────────┘

┌─────────────────────┐                   ┌─────────────────────────┐
│  Types of functions │───────────────────│  Functions defined in   │
└─────────────────────┘                   │        modules          │
                                          └─────────────────────────┘

                                          ┌─────────────────────────┐
                                          │  User defined functions │
                                          └─────────────────────────┘
```

1. Library Functions: These functions are already built in the python library.

2. Functions defined in modules: These functions defined in particular modules. When you want to use these functions in program, you have to import the corresponding module of that function.

3. User Defined Functions: The functions those are defined by the user are called user defined functions.

1. Library Functions in Python:

These functions are already built in the library of python.

For example: type( ), len( ), input( ) etc.

2. Functions defined in modules:

a. Functions of math module:

To work with the functions of math module, we must import math module in program.

import math

S. No.	Function	Description	Example
1	sqrt( )	Returns the square root of a number	>>>math.sqrt(49) 7.0
2	ceil( )	Returns the upper integer	>>>math.ceil(81.3) 82
3	floor( )	Returns the lower integer	>>>math.floor(81.3) 81
4	pow( )	Calculate the power of a number	>>>math.pow(2,3) 8.0
5	fabs( )	Returns the absolute value of a number	>>>math.fabs(-5.6) 5.6
6	exp( )	Returns the e raised to the power i.e. e^3	>>>math.exp(3) 20.085536923187668

b. Function in random module:

random module has a function randint( ).

- ➤ randint( ) function generates the random integer values including start and end values.
- ➤ Syntax: randint(start, end)
- ➤ It has two parameters. Both parameters must have integer values.

Example:

import random

n=random.randint(3,7)

*The value of n will be 3 to 7.

3. USER DEFINED FUNCTIONS:

The syntax to define a function is:

```
def   function-name ( parameters) :

       #statement(s)
```

Where:

> ➢ Keyword **def** marks the start of function header.
> ➢ A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
> ➢ Parameters (arguments) through which we pass values to a function. They are optional.
> ➢ A colon (**:**) to mark the end of function header.
> ➢ One or more valid python statements that make up the function body. Statements must have same indentation level.
> ➢ An optional return statement to return a value from the function.

Example:

```
def display(name):

    print("Hello " +  name   + "  How are you?")
```

5.3 Function Parameters:

A functions has two types of parameters:

1. **Formal Parameter**: Formal parameters are written in the function prototype and function header of the definition. Formal parameters are local variables which are assigned values from the arguments when the function is called.

2. **Actual Parameter**: When a function is *called*, the values that are passed in the call are called *actual parameters*. At the time of the call each actual parameter is assigned to the corresponding formal parameter in the function definition.

Example :

```
def ADD(x, y):                 #Defining a function and x and y are formal parameters

    z=x+y

    print("Sum = ", z)

a=float(input("Enter first number: " ))

b=float(input("Enter second number: " ))

ADD(a,b)        #Calling the function by passing actual parameters
```

In the above example, **x** and **y** are formal parameters. **a** and **b** are actual parameters.

5.4 <mark>Calling the function:</mark>

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.
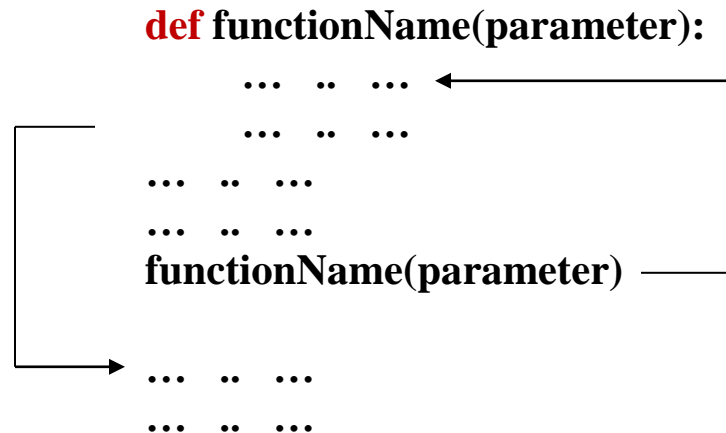
Syntax:

function-name(parameter)

Example:

ADD(10,20)

OUTPUT:

Sum = 30.0

How function works?

def **functionName(parameter):**

        … .. … ⬅

        … .. …

   … .. …

   … .. …

   functionName(parameter) ⎯

   … .. …

   … .. …

<mark>**The return statement:**</mark>

<mark>The **return** statement is used to exit a function and go back to the place from where it was called.</mark>

There are two types of functions according to return statement:

    a. Function returning some value (non-void function)

    b. Function not returning any value (void function)

a. Function returning some value (non-void function) :

> **Syntax:**
>
> > **return expression/value**
>
> **Example-1: Function returning one value**
>
> > def my_function(x):
> > > **return 5 * x**
>
> **Example-2 Function returning multiple values:**
>
> > def sum(a,b,c):
> >
> > > return a+5, b+4, c+7
> >
> > S=sum(2,3,4)   # S will store the returned values as a tuple
> >
> > print(S)
> >
> > OUTPUT:
> >
> > (7, 7, 11)
>
> **Example-3: Storing the returned values separately:**
>
> > def sum(a,b,c):
> >
> > > return a+5, b+4, c+7
> >
> > s1, s2, s3=sum(2, 3, 4)   # storing the values separately
> >
> > print(s1, s2, s3)
> >
> > OUTPUT:
> >
> > 7 7 11

b. Function not returning any value (void function) : The function that performs some operationsbut does not return any value, called void function.

> def message():
>   print("Hello")
>
> m=message()
> print(m)

OUTPUT:

Hello

None

5.5 <mark>Scope and Lifetime of variables:</mark>

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

There are two types of scope for variables:

1. Local Scope

2. Global Scope

1. <mark>**Local Scope:**</mark> Variable used inside the function. It can not be accessed outside the function. In this scope, The lifetime of variables inside a function is as long as the function executes. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

2. <mark>**Global Scope:**</mark> Variable can be accessed outside the function. In this scope, Lifetime of a variable is the period throughout which the variable exits in the memory.

Example:

```
def my_func():
        x = 10
        print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```

OUTPUT:

Value inside function: 10

Value outside function: 20

Here, we can see that the value of **x** is **20** initially. Even though the function my_func()changed the value of **x** to **10**, it did not affect the value outside the function.

This is because the variable **x** inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a **global** scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword **global**.

5.6 RECURSION:

Definition: A function calls itself, is called recursion.

5.6.1Python program to find the **factorial of a number** using recursion:

Program:

```
def factorial(n):
    if n == 1:
        return n
    else:
        return n*factorial(n-1)


num=int(input("enter the number: "))
if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num = = 0:
    print("The factorial of 0 is 1")
else:
    print("The factorial of  ",num," is  ", factorial(num))
```

OUTPUT:

enter the number: 5

The factorial of   5   is   120

5.6.2 Python program to print the Fibonacci series using recursion:

Program:

```python
def fibonacci(n):
    if n<=1:
        return n
    else:
        return(fibonacci(n-1)+fibonacci(n-2))


num=int(input("How many terms you want to display: "))
for i in range(num):
    print(fibonacci(i)," ", end=" ")
```
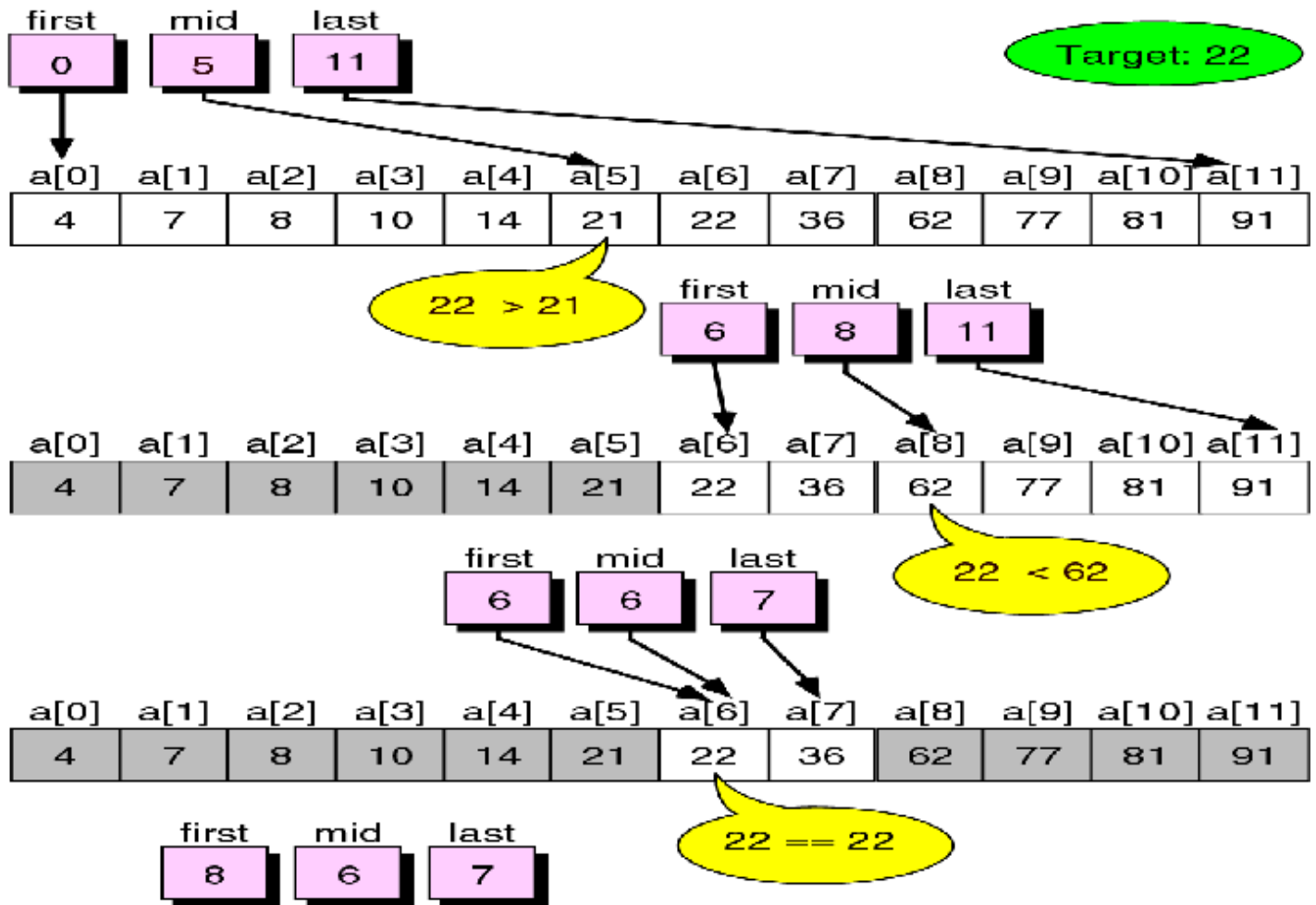
OUTPUT:

How many terms you want to display: 8

0 1 1 2 3 5 8 13

5.6.3 Binary Search using recursion:

Note: The given array or sequence must be sorted to perform binary search.

Program:

```
def Binary_Search(sequence, item, LB, UB):
    if LB>UB:
        return -5              # return any negative value
    mid=int((LB+UB)/2)
    if item==sequence[mid]:
        return mid
    elif item<sequence[mid]:
        UB=mid-1
        return Binary_Search(sequence, item, LB, UB)
    else:
        LB=mid+1
```

```
                return Binary_Search(sequence, item, LB, UB)

        L=eval(input("Enter the elements in sorted order: "))

        n=len(L)

        element=int(input("Enter the element that you want to search :"))

        found=Binary_Search(L,element,0,n-1)

        if found>=0:

            print(element, "Found at the index : ",found)

        else:

            print("Element not present in the list")
```

5.7 lambda Function:

lambda keyword, is used to create anonymous function which doesn't have any name.

While normal functions are defined using the **def** keyword, in Python anonymous functions are defined using the **lambda** keyword.

Syntax:

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example:

value = lambda x: x * 4

print(value(6))

Output:

24

In the above program, lambda x: x * 4 is the lambda function. Here x is the argument and x * 4 is the expression that gets evaluated and returned.

Programs related to Functions in Python topic:

1. Write a python program to sum the sequence given below. Take the input **n** from the user.

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

Solution:

```
def fact(x):
    j=1
    res=1
    while j<=x:
        res=res*j
        j=j+1
        return res
n=int(input("enter the number : "))
i=1
sum=1
while i<=n:
 f=fact(i)
 sum=sum+1/f
 i+=1
print(sum)
```

2. Write a program to compute GCD and LCM of two numbers

```
def gcd(x,y):
    while(y):
        x, y = y, x % y
    return x
```

```python
def lcm(x, y):

    lcm = (x*y)//gcd(x,y)

    return lcm


num1 = int(input("Enter first number: "))

num2 = int(input("Enter second number: "))


print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))

print("The G.C.D. of", num1,"and", num2,"is", gcd(num1, num2))
```

CHAPTER-6
STRING IN PYTHON

6.1 Introduction:

Definition: Sequence of characters enclosed in single, double or triple quotation marks.

Basics of String:

➢ Strings are immutable in python. It means it is unchangeable. At the same memory address, the new value cannot be stored.

➢ Each character has its index or can be accessed using its index.

➢ String in python has two-way index for each location. (0, 1, 2, ……. In the forward direction and -1, -2, -3, …….. in the backward direction.)

Example:

0	1	2	3	4	5	6	7
k	e	n	d	r	i	y	a
-8	-7	-6	-5	-4	-3	-2	-1

➢ The index of string in forward direction starts from 0 and in backward direction starts from -1.
➢ The size of string is total number of characters present in the string. (If there are n characters in the string, then last index in forward direction would be n-1 and last index in backward direction would be –n.)

➢ String are stored each character in contiguous location.
➢ The character assignment is not supported in string because strings are immutable.
   Example :
   str = "kendriya"
   str[2] = 'y'   # it is invalid. Individual letter assignment not allowed in python

6.2 Traversing a String:

Access the elements of string, one character at a time.

   str = "kendriya"

   for ch in str :

      print(ch, end= ' ')

Output:

kendriya

6.3 String Operators:

a. Basic Operators (+, *)

b. Membership Operators (in, not in)

c. Comparison Operators (==, !=, <, <=, >, >=)

a. Basic Operators: There are two basic operators of strings:

    i.   String concatenation Operator (+)

    ii.   String repetition Operator (*)

i. **String concatenation Operator**: The + operator creates a new string by joining the two operand strings.

Example:

>>>"Hello"+"Python"

'HelloPython'

>>>'2'+'7'

'27'

>>>"Python"+"3.0"

'Python3.0'

Note: You cannot concate numbers and strings as operands with + operator.

Example:

>>>7+'4'   # unsupported operand type(s) for +: 'int' and 'str'

It is invalid and generates an error.

ii. **String repetition Operator:** It is also known as <mark>String replication operator</mark>. It requires <mark>two types of operands- a string and an integer number.</mark>

Example:

>>>"you" * 3

'youyouyou'

>>>3*"you"

'youyouyou'

Note:You cannot have strings as n=both the operands with * operator.

Example:

>>>"you" * "you"   # can't multiply sequence by non-int of type 'str'

It is invalid and generates an error.

<mark>b. Membership Operators:</mark>

<mark>in –</mark> Returns **True** if a character or a substring exists in the given string; otherwise **False**

<mark>not in -</mark> Returns **True** if a character or a substring does not exist in the given string; otherwise **False**

Example:

>>> "ken" in "Kendriya Vidyalaya"

False

>>> "Ken" in "Kendriya Vidyalaya"

True

>>>"ya V" in "Kendriya Vidyalaya"

True

>>>"8765" not in "9876543"

False

c. <mark>**Comparison Operators:**</mark> These operators compare two strings character by character according to their ASCII value.

Characters	ASCII (Ordinal) Value
'0' to '9'	48 to 57
'A' to 'Z'	65 to 90
'a' to 'z'	97 to 122

Example:

>>> 'abc'>'abcD'

False

>>> 'ABC'<'abc'

True

>>> 'abcd'>'aBcD'

True

>>> 'aBcD'<='abCd'

True

6.4 Finding the Ordinal or Unicode value of a character:

Function	Description
ord(<character>)	Returns ordinal value of a character
chr(<value>)	Returns the corresponding character

Example:

>>> ord('b')

98

>>> chr(65)

'A'

Program: Write a program to display ASCII code of a character and vice versa.

```
var=True
while var:
    choice=int(input("Press-1 to find the ordinal value \n Press-2 to find a character of a value\n"))
    if choice==1:
        ch=input("Enter a character : ")
        print(ord(ch))
    elif choice==2:
        val=int(input("Enter an integer value: "))
        print(chr(val))
    else:
        print("You entered wrong choice")

    print("Do you want to continue? Y/N")
    option=input()
    if option=='y' or option=='Y':
        var=True
    else:
        var=False
```

6.5 Slice operator with Strings:

The slice operator slices a string using a range of indices.

Syntax:

string-name[start**:**end]

where **start** and **end** are integer indices. It returns a string from the index **start** to **end-1**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
d	a	t	a		s	t	r	u	c	t	u	r	e
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Example:

>>> str="data structure"

>>> str[0:14]

'data structure'

>>> str[0:6]

'data s'

>>> str[2:7]

'ta st'

>>> str[-13:-6]

'ata str'

>>> str[-5:-11]

' '   #returns empty string

>>> str[:14]   # Missing index before colon is considered as 0.

'data structure'

>>> str[0:]   # Missing index after colon is considered as 14. (length of string)

'data structure'

>>> str[7:]

'ructure'

>>> str[4:]+str[:4]

' structuredata'

>>> str[:4]+str[4:]   #for any index **str[:n]+str[n:]** returns original string

'data structure'

>>> str[8:]+str[:8]

'ucturedata str'

>>> str[8:], str[:8]

('ucture', 'data str')

Slice operator with step index:

Slice operator with strings may have third index. Which is known as step. It is optional.

Syntax:

string-name[start:end:step]

Example:

>>> str="data structure"

>>> str[2:9:2]

't tu'

>>> str[-11:-3:3]

'atc'

>>> str[: : -1]   # reverses a string

'erutcurts atad'

Interesting Fact: Index out of bounds causes error with strings but slicing a string outside the index does not cause an error.

Example:

>>>str[14]

IndexError: string index out of range

>>> str[14:20]   # both indices are outside the bounds

' '   # returns empty string

>>> str[10:16]

'ture'

Reason: When you use an index, you are accessing a particular character of a string, thus the index must be valid and out of bounds index causes an error as there is no character to return from the given index.

But slicing always returns a substring or empty string, which is valid sequence.

6.6 Built-in functions of string:

Example:

str="data structure"

s1= "hello365"

s2= "python"

s3 = '4567'

s4 = ' '

s5= 'comp34%@'

S. No.	Function	Description	Example
1	len( )	Returns the length of a string	>>>print(len(str)) 14
2	capitalize( )	Returns a string with its **first character** capitalized.	>>> str.capitalize() 'Data structure'

3	find(sub,start,end)	Returns the **lowest index** in the string where the substring **sub** is found within the slice range.
		Returns -1 if **sub** is not found.
			>>> str.find("ruct",5,13)
			7
			>>> str.find("ruct",8,13)
			-1

4	isalnum( )	Returns **True** if the characters in the string are alphabets or numbers. **False** otherwise	>>>s1.isalnum( ) True >>>s2.isalnum( ) True >>>s3.isalnum( ) True >>>s4.isalnum( ) False >>>s5.isalnum( ) False
5	isalpha( )	Returns **True** if all characters in the string are alphabetic. **False** otherwise.	>>>s1.isalpha( ) False >>>s2.isalpha( ) True >>>s3.isalpha( ) False >>>s4.isalpha( ) False >>>s5.isalpha( ) False
6	isdigit( )	Returns **True** if all the characters in the string are digits. **False** otherwise.	>>>s1.isdigit( ) False >>>s2.isdigit( ) False >>>s3.isdigit( ) True >>>s4.isdigit( ) False >>>s5.isdigit( ) False

7	islower( )	Returns **True** if all the characters in the string are lowercase. **False** otherwise.	>>> s1.islower()
			True
			>>> s2.islower()

			True
			>>> s3.islower()
			False
			>>> s4.islower()
			False
			>>> s5.islower()
			True

8	isupper( )	Returns **True** if all the characters in the string are uppercase. **False** otherwise.	>>> s1.isupper()
			False
			>>> s2.isupper()
			False
			>>> s3.isupper()
			False
			>>> s4.isupper()
			False
			>>> s5.isupper()
			False

9	isspace( )	Returns **True** if there are only whitespace characters in the string. **False** otherwise.	>>> " ".isspace()
			True
			>>> "".isspace()
			False

10	lower( )	Converts a string in lowercase characters.	>>> "HeLlo".lower() 'hello'
11	upper( )	Converts a string in uppercase characters.	>>> "hello".upper() 'HELLO'

12	lstrip( )	Returns a string after removing the leading characters. (Left side).
		if used without any argument, it removes the leading whitespaces.
			>>> str="data structure"
			>>> str.lstrip('dat')
			' structure'
			>>> str.lstrip('data')
			' structure'
			>>> str.lstrip('at')
			'data structure'
			>>> str.lstrip('adt')
			' structure'
			>>> str.lstrip('tad')
			' structure'

13	rstrip( )	Returns a string after removing the trailing characters. (Right side).
		if used without any argument, it removes the trailing whitespaces.
			>>> str.rstrip('eur')
			'data struct'
			>>> str.rstrip('rut')
			'data structure'
			>>> str.rstrip('tucers')
			'data '

14	split( )	breaks a string into words and creates a list out of it	>>> str="Data Structure"
			>>> str.split( )
			['Data', 'Structure']

1. Write a program that takes a string with multiple words and then capitalize the first letter of each word and forms a new string out of it.

Solution:

```
s1=input("Enter a string : ")

length=len(s1)

a=0

end=length

s2=""    #empty string

while a<length:

   if a==0:

      s2=s2+s1[0].upper()

      a+=1

   elif (s1[a]==' 'and s1[a+1]!=''):

      s2=s2+s1[a]

      s2=s2+s1[a+1].upper()

      a+=2

   else:

      s2=s2+s1[a]

      a+=1

print("Original string : ", s1)

print("Capitalized wrds string: ", s2)
```


2. Write a program that reads a string and checks whether it is a palindrome string or not.

```
str=input("Enter a string : ")

n=len(str)

mid=n//2

rev=-1
```

```python
for i in range(mid):
    if str[i]==str[rev]:
        i=i+1
        rev=rev-1
    else:
        print("String is not palindrome")
        break
else:
    print("String is palindrome")
```

3. Write a program to convert lowercase alphabet into uppercase and vice versa.

```python
choice=int(input("Press-1 to convert in lowercase\n Press-2 to convert in uppercase\n"))
str=input("Enter a string: ")
if choice==1:
    s1=str.lower()
    print(s1)
elif choice==2:
    s1=str.upper()
    print(s1)
else:
    print("Invalid choice entered")
```

<h1 style="text-align:center">CHAPTER-7</h1>
<h2 style="text-align:center">LIST IN PYTHON</h2>

7.1 Introduction:

- List is a collection of elements which is ordered and changeable (mutable).
- Allows duplicate values.
- A list contains items separated by commas and enclosed within square brackets ([ ]).
- All 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.

Difference between list and string:

List	String
Mutable	Immutable
Element can be assigned at specified index	Element/character cannot be assigned at specified index.
Example: >>>L=[7,4,8,9] >>>L[2]=6 #valid	**Example:** >>>str= "python" >>>str[2]= 'p' #error

7.2 Creating a list:

To create a list enclose the elements of the list within square brackets and separate the elements by commas.

Syntax:

list-name= [item-1, item-2, …….., item-n]

Example:

mylist = ["apple", "banana", "cherry"]   # a list with three items

L = [ ]   # an empty list

7.2.1 Creating a list using list( ) Constructor:

o It is also possible to use the list( ) constructor to make a list.

```
mylist = list(("apple", "banana", "cherry"))        #note the double round-brackets
print(mylist)

L=list( )          # creating empty list
```

7.2.2 Nested Lists:

```
>>> L=[23,'w',78.2, [2,4,7],[8,16]]

>>> L

[23, 'w', 78.2, [2, 4, 7], [8, 16]]
```

7.2.3 Creating a list by taking input from the user:

```
>>> List=list(input("enter the elements: "))

enter the elements: hello python

>>> List

['h', 'e', 'l', 'l', 'o', ' ', 'p', 'y', 't', 'h', 'o', 'n']


>>> L1=list(input("enter the elements: "))

enter the elements: 678546

>>> L1

['6', '7', '8', '5', '4', '6']          # it treats elements as the characters though we entered digits
```

To overcome the above problem, we can use eval( ) method, which identifies the data type and evaluate them automatically.

```
>>> L1=eval(input("enter the elements: "))

enter the elements: 654786

>>> L1

654786          # it is an integer,  not a list

>>> L2=eval(input("enter the elements: "))
```

enter the elements: [6,7,8,5,4,3]   # for list, you must enter the [ ] bracket

>>> L2

[6, 7, 8, 5, 4, 3]

Note: With eval( ) method, If you enter elements without square bracket[ ], it will be considered as a tuple.

>>> L1=eval(input("enter the elements: "))
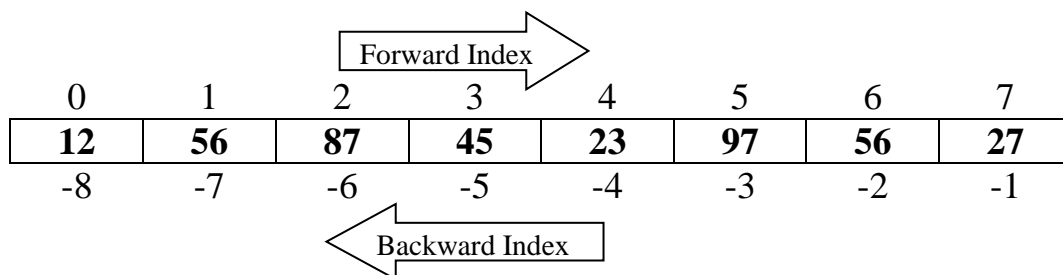
enter the elements: 7,65,89,6,3,4

>>> L1

(7, 65, 89, 6, 3, 4)   #tuple

7.3 Accessing lists:

➢ The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes.
➢ **List-name[start:end]** will give you elements between indices **start** to **end-1**.
➢ The first item in the list has the index zero (0).

Example:
>>> number=[12,56,87,45,23,97,56,27]

			Forward Index				
0	1	2	3	4	5	6	7
12	**56**	**87**	**45**	**23**	**97**	**56**	**27**
-8	-7	-6	-5	-4	-3	-2	-1
			Backward Index				

>>> number[2]
87
>>> number[-1]
27
>>> number[-8]
12
>>> number[8]
IndexError: list index out of range
>>> number[5]=55   #Assigning a value at the specified index

```
>>> number
[12, 56, 87, 45, 23, 55, 56, 27]
```

7.4 Traversing a LIST:

 Traversing means accessing and processing each element.

Method-1:

```
>>> day=list(input("Enter elements :"))
Enter elements : sunday
>>> for d in day:
       print(d)
```

Output:
```
s
u
n
d
a
y
```

Method-2
```
>>> day=list(input("Enter elements :"))
Enter elements : wednesday
>>> for i in range(len(day)):
       print(day[i])
```

Output:
```
w
e
d
n
e
s
d
a
y
```

7.5 List Operators:

- ➢ Joining operator   +
- ➢ Repetition operator   *
- ➢ Slice operator   [:]
- ➢ Comparison Operator <, <=, >, >=, ==, !=

- **Joining Operator:** It joins two or more lists**.**

  Example:

  >>> L1=['a',56,7.8]

  >>> L2=['b','&',6]

  >>> L3=[67,'f','p']

  >>> L1+L2+L3

  ['a', 56, 7.8, 'b', '&', 6, 67, 'f', 'p']

- **Repetition Operator:** It replicates a list specified number of times**.**

  Example:

  >>> L1*3

  ['a', 56, 7.8, 'a', 56, 7.8, 'a', 56, 7.8]

  >>> 3*L1

  ['a', 56, 7.8, 'a', 56, 7.8, 'a', 56, 7.8]

- **Slice Operator:**

  List-name[start:end] will give you elements between indices **start** to **end-1**.

  >>> number=[12,56,87,45,23,97,56,27]

  >>> number[2:-2]

  [87, 45, 23, 97]

  >>> number[4:20]

  [23, 97, 56, 27]

  >>> number[-1:-6]

  [ ]

  >>> number[-6:-1]

  [87, 45, 23, 97, 56]

>>> number[0:len(number)]

[12, 56, 87, 45, 23, 97, 56, 27]

List-name[start:end:step] will give you elements between indices **start** to **end-1** with skipping elements as per the value of **step**.

>>> number[1:6:2]

[56, 45, 97]

>>> number[: : -1]

[27, 56, 97, 23, 45, 87, 56, 12]   #reverses the list

List modification using slice operator:

>>> number=[12,56,87,45,23,97,56,27]

>>> number[2:4]=["hello","python"]

>>> number

[12, 56, 'hello', 'python', 23, 97, 56, 27]

>>> number[2:4]=["computer"]

>>> number

[12, 56, 'computer', 23, 97, 56, 27]

Note: The values being assigned must be a sequence (list, tuple or string)

Example:

>>> number=[12,56,87,45,23,97,56,27]

>>> number=[12,56,87,45,23,97,56,27]

>>> number[2:3]=78   # 78 is a number, not a sequence

TypeError: can only assign an iterable

- **Comparison Operators:**
    - Compares two lists
    - Python internally compares individual elements of lists in lexicographical order.
    - It compares the each corresponding element must compare equal and two sequences must be of the same type.
    - For **non-equal comparison** as soon as it gets a result in terms of True/False, from corresponding elements' comparison. If Corresponding elements are equal, it goes to the next element and so on, until it finds elements that differ.

Example:

>>>L1, L2 = [7, 6, 9], [7, 6, 9]

>>>L3 = [7, [6, 9]]

For Equal Comparison:

Comparison	Result	Reason
>>>L1==L2	True	Corresponding elements have same value and same type
>>>L1==L3	False	Corresponding values are not same

For Non-equal comparison:

Comparison	Result	Reason
>>> L1>L2	False	All elements are equal
>>> L2>L3	TypeError: '>' not supported between instances of 'int' and 'list'	in L2, element at the index 1 is int type and in L3 element at the index 1 is list type
>>>[3,4,7,8]<[5,1]	True	3<5 is True
>>>[3,4,7,8]<[3,4,9,2]	True	First two elements are same so move to next element and 7<9 is True
>>>[3,4,7,8]<[3,4,9,11]	True	7<9 is True
>>>[3,4,7,8]<[3,4,7,5]	False	8<5 is False

List Methods:

Consider a list:

company=["IBM","HCL","Wipro"]

S. No.	Function Name	Description	Example

1	append( )	To add element to the list at the end.
		Syntax:
		list-name.append (element)
			>>> company.append("Google")
			>>> company
			['IBM', 'HCL', 'Wipro', 'Google']

Error: >>>company.append("infosys","microsoft") # takes exactly one element TypeError: append() takes exactly one argument (2 given)			

2	extend( )	Add a list, to the end of the current list.
		Syntax:
		list-name.extend(list)
			>>>company=["IBM","HCL","Wipro"]
			>>> desktop=["dell","HP"]
			>>> company.extend(desktop)
			>>> company
			['IBM', 'HCL', 'Wipro', 'dell', 'HP']

Error: >>>company.extend("dell","HP") #takes only a list as argument TypeError: extend() takes exactly one argument (2 given)			

3.	len( )	Find the length of the list.
		Syntax:
		len(list-name)
			>>>company=["IBM","HCL","Wipro"]
			>>> len(company)
			3
			>>> L=[3,6,[5,4]]
			>>> len(L)
			3

4	index( )	Returns the index of the first element with the specified value.
		Syntax:
		list-name.index(element)
			>>> company = ["IBM", "HCL", "Wipro", "HCL","Wipro"]
			>>> company.index("Wipro")
			2

Error: >>> company.index("WIPRO") # Python is case-sensitive language ValueError: 'WIPRO' is not in list			

			>>> company.index(2)　　　# Write the element, not index ValueError: 2 is not in list

5	insert( )	Adds an element at the specified position.
		
		Syntax:
		list.insert(index, element)
			>>>company=["IBM","HCL","Wipro"]
			>>> company.insert(2,"Apple")
			>>> company
			['IBM', 'HCL', 'Apple', 'Wipro']
			
			>>> company.insert(16,"Microsoft")
			>>> company
			['IBM',　'HCL',　'Apple',　'Wipro',
			'Microsoft']
			
			>>> company.insert(-16,"TCS")
			>>> company
			['TCS',　'IBM',　'HCL',　'Apple',　'Wipro',
			'Microsoft']

6	count( )	Return the number of times the value appears.
		Syntax:
		list-name.count(element)
			>>>　company　=　["IBM",　"HCL",
			"Wipro", "HCL","Wipro"]
			>>> company.count("HCL")
			2
			>>> company.count("TCS")
			0

7	remove( )	To remove an element from the list.
		Syntax:
		list-name.remove(element)
			>>>　company　=　["IBM",　"HCL",
			"Wipro", "HCL","Wipro"]
			>>> company.remove("Wipro")
			>>> company
			['IBM', 'HCL', 'HCL', 'Wipro']

Error: >>> company.remove("Yahoo") ValueError: list.remove(x): x not in list			

8	clear( )	Removes all the elements from list.
		Syntax:
		list-name.clear( )
			>>> company=["IBM","HCL", "Wipro"]
			
			>>> company.clear( )
			
			>>> company
			
			[ ]

9	pop( )	Removes the element at the specified position and returns the deleted element.
		
		Syntax:
		list-name.pop(index)
		
		The index argument is optional. If no index is specified, pop( ) removes and returns the last item in the list.
			>>>company=["IBM","HCL", "Wipro"]
			>>> company.pop(1)
			'HCL'
			>>> company
			['IBM', 'Wipro']
			
			>>> company.pop( )
			'Wipro'

Error: >>>L=[ ] >>>L.pop( ) IndexError: pop from empty list

10	copy( )	Returns a copy of the list.
		
		Syntax:
		list-name.copy( )
			>>>company=["IBM","HCL", "Wipro"]
			>>> L=company.copy( )
			>>> L
			['IBM', 'HCL', 'Wipro']

11	reverse( )	Reverses the order of the list.
		Syntax:
		list-name.reverse( )
		
		Takes no argument, returns no list.
			>>>company=["IBM","HCL", "Wipro"]
			>>> company.reverse()
			>>> company
			['Wipro', 'HCL', 'IBM']

12.	sort( )	Sorts the list. By default in ascending order.
		
		Syntax:
		list-name.sort( )
			>>>company=["IBM","HCL", "Wipro"]
			>>>company.sort( )
			>>> company
			['HCL', 'IBM', 'Wipro']
			
			To sort a list in descending order:
			
			>>>company=["IBM","HCL", "Wipro"]
			>>> company.sort(reverse=True)
			>>> company
			['Wipro', 'IBM', 'HCL']

Deleting the elements from the list using *del* statement:

Syntax:

del list-name[index]   # to remove element at specified index

del list-name[start:end]   # to remove elements in list slice

Example:

>>> L=[10,20,30,40,50]

>>> del L[2]   # delete the element at the index 2

>>> L

[10, 20, 40, 50]

>>> L= [10,20,30,40,50]

>>> del L[1:3]   # deletes elements of list from index 1 to 2.

>>> L

[10, 40, 50]

>>> del L   # deletes all elements and the list object too.

>>> L

NameError: name 'L' is not defined

Difference between del, remove( ), pop( ), clear( ):

S. No.	del	remove( )	pop( )	clear( )
1	Statement	Function	Function	Function
2	Deletes a single element or a list slice or complete list.	Removes the first matching item from the list.	Removes an individual item and returns it.	Removes all the elements from list.
3	Removes all elements and deletes list object too.			Removes all elements but list object still exists.

Difference between append( ), extend( ) and insert( ) :

S. No.	append( )	extend( )	insert( )
1	Adds single element in the end of the list.	Add a list in the end of the another list	Adds an element at the specified position. (Anywhere in the list)
2	Takes one element as argument	Takes one list as argument	Takes two arguments, position and element.
3	The length of the list will increase by 1.	The length of the list will increase by the length of inserted list.	The length of the list will increase by 1.

ACCESSING ELEMENTS OF NESTED LISTS:

Example:

>>> L=["Python", "is", "a", ["modern", "programming"], "language", "that", "we", "use"]

>>> L[0][0]

'P'

>>> L[3][0][2]

'd'

```
>>> L[3:4][0]
['modern', 'programming']
>>> L[3:4][0][1]
'programming'
>>> L[3:4][0][1][3]
'g'
>>> L[0:9][0]
'Python'
>>> L[0:9][0][3]
'h'
>>> L[3:4][1]
IndexError: list index out of range
```

<mark>**Programs related to lists in python:**</mark>

Program-1 Write a program to find the minimum and maximum number in a list.

```python
L=eval(input("Enter the elements: "))
n=len(L)
min=L[0]
max=L[0]
for i in range(n):
    if min>L[i]:
        min=L[i]
    if max<L[i]:
        max=L[i]

print("The minimum number in the list is : ", min)
print("The maximum number in the list is : ", max)
```

Program-2 Find the second largest number in a list.

```
L=eval(input("Enter the elements: "))
n=len(L)
max=second=L[0]
for i in range(n):
   if max<L[i]>second:
     max=L[i]
     seond=max


print("The second largest number in the list is : ", second)
```

Program-3: Program to search an element in a list. (Linear Search).

```
L=eval(input("Enter the elements: "))
n=len(L)
item=eval(input("Enter the element that you want to search : "))
for i in range(n):
   if L[i]==item:
     print("Element found at the position :", i+1)
     break
else:
   print("Element not Found")
```

Output:

Enter the elements: 56,78,98,23,11,77,44,23,65

Enter the element that you want to search : 23

Element found at the position : 4

TUPLE IN PYTHON

8.1 INTRODUCTION:

- Tuple is a collection of elements which is **ordered and unchangeable (Immutable)**. Immutable means you cannot change elements of a tuple in place.
- Allows duplicate members.
- Consists the values of any type, separated by comma.
- Tuples are enclosed within parentheses ( ).
- Cannot remove the element from a tuple.

8.2 Creating Tuple:

Syntax:

tuple-name = ( )   # empty tuple

 tuple-name = (value-1, value-2, …….. , value-n)

Example:

>>> T=(23, 7.8, 64.6, 'h', 'say')

>>> T

(23, 7.8, 64.6, 'h', 'say')

8.2.1 Creating a tuple with single element:

>>> T=(3)   #With a single element without comma, it is a value only, not a tuple

>>> T

3

>>> T= (3,)   # to construct a tuple, add a comma after the single element

>>> T

(3,)

>>> T1=3,   # It also creates a tuple with single element

>>> T1

(3,)

8.2.2 Creating a tuple using tuple( ) constructor:

o It is also possible to use the **tuple( )** constructor to create a tuple.

```
>>>T=tuple( )                # empty tuple

>>> T=tuple((45,3.9, 'k',22))        #note the double round-brackets
>>> T
(45, 3.9, 'k', 22)

>>> T2=tuple('hello')  # for single round-bracket, the argument must be of sequence type
>>> T2
('h', 'e', 'l', 'l', 'o')

>>> T3=('hello','python')
>>> T3
('hello', 'python')
```

8.2.3 Nested Tuples:

>>> T=(5,10,(4,8))

>>> T

(5, 10, (4, 8))

8.2.4 Creating a tuple by taking input from the user:

```
>>> T=tuple(input("enter the elements: "))

enter the elements: hello python

>>> T
```

('h', 'e', 'l', 'l', 'o', ' ', 'p', 'y', 't', 'h', 'o', 'n')

>>> T1=tuple(input("enter the elements: "))

enter the elements: 45678

>>> T1

('4', '5', '6', '7', '8')   # it treats elements as the characters though we entered digits

To overcome the above problem, we can use eval( ) method, which identifies the data type and evaluate them automatically.

>>> T1=eval(input("enter the elements: "))

enter the elements: 56789

>>> T1

56789   # it is not a list, it is an integer value

>>> type(T1)

<class 'int'>

>>> T2=eval(input("enter the elements: "))

enter the elements: (1,2,3,4,5)   # Parenthesis is optional

>>> T2

(1, 2, 3, 4, 5)

>>> T3=eval(input("enter the elements: "))

enter the elements: 6, 7, 3, 23, [45,11]   # list as an element of tuple

>>> T3

(6, 7, 3, 23, [45, 11])

8.3 Accessing Tuples:

Tuples are very much similar to lists. Like lists, tuple elements are also indexed. Forward indexing as 0,1,2,3,4……… and backward indexing as -1,-2,-3,-4,………

➤ The values stored in a tuple can be accessed using the slice operator ([ ] and [:]) with indexes.
➤ **tuple-name[start:end]** will give you elements between indices **start** to **end-1**.
➤ The first item in the tuple has the index zero (0).

Example:

>>> alpha=('q','w','e','r','t','y')
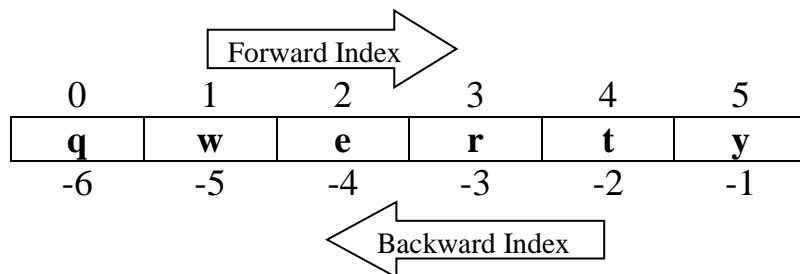

```
>>> alpha[5]
'y'
>>> alpha[-4]
'e'
>>> alpha[46]
```

IndexError: tuple index out of range

>>> alpha[2]='b'   #can't change value in tuple, the value will remain unchanged

TypeError: 'tuple' object does not support item assignment

8.3.1 Difference between List and Tuple:

S. No.	List	Tuple
1	Ordered and changeable (Mutable)	Ordered but unchangeable (Immutable)
2	Lists are enclosed in brackets. [ ]	Tuples are enclosed in parentheses. ( )
3	Element can be removed.	Element can't be removed.

8.4 Traversing a Tuple:

Syntax:

for <variable> in tuple-name:

       statement

Example:

Method-1

>>> alpha=('q','w','e','r','t','y')
>>> for i in alpha**:**
      print(i)

Output:
q
w
e
r
t
y

Method-2
>>> for i in range(0, len(alpha))**:**
      print(alpha[i])

Output:
q
w
e
r
t
y

8.5 Tuple Operations:

- ➢ Joining operator   +
- ➢ Repetition operator   *
- ➢ Slice operator   [:]
- ➢ Comparison Operator <, <=, >, >=, ==, !=

- **Joining Operator:** It joins two or more tuples**.**

  Example:

  ```
  >>> T1 = (25,50,75)
  >>> T2 = (5,10,15)
  >>> T1+T2
  (25, 50, 75, 5, 10, 15)
  ```

  >>> T1 + (34)
  TypeError: can only concatenate tuple (not "int") to tuple

  >>> T1 + (34,)

  (25, 50, 75, 34)

- **Repetition Operator:** It replicates a tuple, specified number of times**.**

  Example:

  >>> T1*2

  (25, 50, 75, 25, 50, 75)

  >>> T2=(10,20,30,40)

  >>> T2[2:4]*3

  (30, 40, 30, 40, 30, 40)

- **Slice Operator:**

  tuple-name[start:end] will give you elements between indices **start** to **end-1**.

  ```
  >>>alpha=('q','w','e','r','t','y')

  >>> alpha[1:-3]

  ('w', 'e')

  >>> alpha[3:65]

  ('r', 't', 'y')

  >>> alpha[-1:-5]
  ```

( )

>>> alpha[-5:-1]

('w', 'e', 'r', 't')

List-name[start:end:step] will give you elements between indices **start** to **end-1** with skipping elements as per the value of **step**.

>>> alpha[1:5:2]

('w', 'r')

>>> alpha[**: :** -1]

('y', 't', 'r', 'e', 'w', 'q')   #reverses the tuple

- **Comparison Operators:**

  o Compare two tuples
  o Python internally compares individual elements of tuples in lexicographical order.
  o It compares the each corresponding element must compare equal and two sequences must be of the same type.
  o For **non-equal comparison** as soon as it gets a result in terms of True/False, from corresponding elements' comparison. If Corresponding elements are equal, it goes to the next element and so on, until it finds elements that differ.

Example:

```
>>> T1 = (9, 16, 7)
>>> T2 = (9, 16, 7)
>>> T3 = ('9','16','7')
>>> T1 = = T2
True
>>> T1==T3
False
>>> T4 = (9.0, 16.0, 7.0)
>>> T1==T4
True
>>> T1<T2
```

False
>>> T1<=T2
True

8.6 Tuple Methods:

Consider a tuple:

subject=("Hindi","English","Maths","Physics")

S. No.	Function Name	Description	Example
1	len( )	Find the length of a tuple. **Syntax:** len (tuple-name)	>>>subject=("Hindi","English","Maths","Physics”) >>> len(subject) 4
2	max( )	Returns the largest value from a tuple. **Syntax:** max(tuple-name)	>>> max(subject) 'Physics'

Error: If the tuple contains values of different data types, then it will give an error because mixed data type comparison is not possible.
 	
>>> subject = (15, "English", "Maths", "Physics", 48.2)
>>> max(subject)
TypeError: '>' not supported between instances of 'str' and 'int'			

3.	min( )	Returns the smallest value from a tuple. **Syntax:** min(tuple-name)	>>>subject=("Hindi","English","Maths","Physics") >>> min(subject) 'English'

Error: If the tuple contains values of different data types, then it will give an error because mixed data type comparison is not possible.
 	
>>> subject = (15, "English", "Maths", "Physics", 48.2)
>>> min(subject)
TypeError: '>' not supported between instances of 'str' and 'int'			

4	index( )	Returns the index of the first element with the specified value. **Syntax:**	>>>subject=("Hindi","English","Maths","Physics")

		tuple-name.index(element)	>>> subject.index("Maths") 2
5	<mark>count( )</mark>	Return the number of times the value appears. **Syntax:** tuple-name.count(element)	>>> subject.count("English") 1

8.7 <mark>Tuple Packing and Unpacking:</mark>

<mark>Tuple Packing:</mark> Creating a tuple from set of values.

Example:

```
>>> T=(45,78,22)
>>> T
(45, 78, 22)
```

<mark>Tuple Unpacking :</mark> Creating individual values from the elements of tuple.

Example:

```
>>> a, b, c=T
>>> a
45
>>> b
78
>>> c
22
```

Note: Tuple **unpacking** requires that the number of variable on the left side must be equal to the length of the tuple.

8.8 <mark>Delete a tuple:</mark>

<mark>The *del* statement</mark> is used to delete elements and objects but as you know that tuples are immutable, which also means that individual element of a tuple cannot be deleted.

Example:

>> T=(2,4,6,8,10,12,14)

>>> del T[3]

TypeError: 'tuple' object doesn't support item deletion

But you can delete a complete tuple with *del* statement as:

Example:

>>> T=(2,4,6,8,10,12,14)

>>> del T

>>> T

NameError: name 'T' is not defined

DICTIONARY IN PYTHON

9.1 INTRODUCTION:

- Dictionary is a collection of elements which is **unordered, changeable and indexed**.
- Dictionary has **keys** and **values**.
- Doesn't have index for values. Keys work as indexes.
- Dictionary doesn't have duplicate member means no duplicate key.
- Dictionaries are enclosed by curly braces { }
- The key-value pairs are separated by commas (,)
- A dictionary key can be almost any Python type, but are usually numbers or strings.
- Values can be assigned and accessed using square brackets [ ].

9.2 CREATING A DICTIONARY:

Syntax:

dictionary-name = {key1:value, key2:value, key3:value, keyn:value}

Example:

>>> marks = { "physics" : 75, "Chemistry" : 78, "Maths" : 81, "CS":78 }

>>> marks

{'physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 78}

>>> D = { }   #Empty dictionary

>>> D

{ }

>>> marks = { "physics" : 75, "Chemistry" : 78, "Maths" : 81, "CS":78 }

{'Maths': 81, 'Chemistry': 78, 'Physics': 75, 'CS': 78}   # there is no guarantee that

                     # elements in dictionary can be accessed as per specific order.

Note: Keys of a dictionary must be of immutable types, such as **string, number, tuple**.

Example:

>>> D1={[2,3]:"hello"}

TypeError: unhashable type: 'list'

Creating a dictionary using dict( ) Constructor:

A. use the dict( ) constructor with single parentheses:

>>> marks=dict(Physics=75,Chemistry=78,Maths=81,CS=78)

>>> marks

{'Physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 78}

> ➤ **In the above case the keys are not enclosed in quotes and equal sign is used for assignment rather than colon**.

B. dict ( ) constructor using parentheses and curly braces:

>>> marks=dict({"Physics":75,"Chemistry":78,"Maths":81, "CS":78})

>>> marks

{'Physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 78}

C. dict( ) constructor using keys and values separately:

>>> marks=dict(**zip**(("Physics","Chemistry","Maths","CS"),(75,78,81,78)))

>>> marks

{'Physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 78}

In this case the keys and values are enclosed separately in parentheses and are given as argument to the zip( ) function. zip( ) function clubs first key with first value and so on.

D. dict( ) constructor using key-value pairs separately:

Example-a

```
>>> marks=dict([['Physics',75],['Chemistry',78],['Maths',81],['CS',78]])
# list as argument passed to dict( ) constructor contains list type elements.

>>> marks

{'Physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 78}
```

Example-b

```
>>> marks=dict((['Physics',75],['Chemistry',78],['Maths',81],['CS',78]))

# tuple as argument passed to dict( ) constructor contains list type elements

>>> marks

{'Physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 78}
```

Example-c

```
>>> marks=dict((('Physics',75),('Chemistry',78),('Maths',81),('CS',78)))

# tuple as argument to dict( ) constructor and contains tuple type elements

>>> marks

{'Physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 78}
```

9.3 ACCESSING ELEMENTS OF A DICTIONARY:

Syntax:

dictionary-name[key]

Example:

>>> marks = { "physics" : 75, "Chemistry" : 78, "Maths" : 81, "CS":78 }

>>> marks["Maths"]

81

>>> marks["English"]   #Access a key that doesn't exist causes an error

KeyError: 'English'

>>> marks.keys( )   #To access all keys in one go

dict_keys(['physics', 'Chemistry', 'Maths', 'CS'])

>>> marks.values( )   # To access all values in one go

dict_values([75, 78, 81, 78])

Lookup : A dictionary operation that takes a key and finds the corresponding value, is

         called lookup.

9.4 TRAVERSING A DICTIONARY:

Syntax:

for <variable-name> in <dictionary-name>:

         statement

Example:

```
>>> for i in marks:
        print(i, ": ", marks[i])
```

OUTPUT:

physics : 75

Chemistry : 78

Maths : 81

CS : 78

9.5 CHANGE AND ADD THE VALUE IN A DICTIONARY:

Syntax:

dictionary-name[key]=value

Example:

```
>>> marks = { "physics" : 75, "Chemistry" : 78, "Maths" : 81, "CS":78 }
>>> marks
{'physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 78}
>>> marks['CS']=84      #Changing a value
>>> marks
{'physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 84}
>>> marks['English']=89            # Adding a value
>>> marks
{'physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 84, 'English': 89}
```

9.6 DELETE ELEMENTS FROM A DICTIONARY:

There are **two** methods to delete elements from a dictionary:

(i) using **del** statement

(ii) using **pop( )** method

(i) Using **del** statement:

Syntax:

del dictionary-name[key]

Example:

>>> marks

{'physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 84, 'English': 89}

>>> del marks['English']

>>> marks

{'physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 84}

(ii) Using **pop( )** method: It deletes the key-value pair and returns the value of deleted

element.

Syntax:

dictionary-name.pop( )

Example:

>>> marks

{'physics': 75, 'Chemistry': 78, 'Maths': 81, 'CS': 84}

>>> marks.pop('Maths')

81

9.7 CHECK THE EXISTANCE OF A KEY IN A DICTIONARY:

To check the existence of a key in dictionary, two operators are used:

(i) in : it returns **True** if the given key is present in the dictionary, otherwise **False**.

(ii) not in : it returns **True** if the given key is not present in the dictionary, otherwise

False.

Example:

>>> marks = { "physics" : 75, "Chemistry" : 78, "Maths" : 81, "CS":78 }

>>> 'Chemistry' in marks

True

>>> 'CS' not in marks

False

>>> 78 in marks   # in and not in only checks the existence of keys not values

False

However, if you need to search for a value in dictionary, then you can use in operator with the following syntax:

Syntax:

value in dictionary-name. values( )

Example:

>>> marks = { "physics" : 75, "Chemistry" : 78, "Maths" : 81, "CS":78 }

>>> 78 in marks**.**values( )

True

9.8 PRETTY PRINTING A DICTIONARY:

What is Pretty Printing?

To print a dictionary in more readable and presentable form.

For pretty printing a dictionary you need to import <mark>**json** module</mark> and then you can use <mark>**dumps( )** function from json module.</mark>

Example:

>>> print(json.dumps(marks, indent=2))

OUTPUT:

```
{
  "physics": 75,
  "Chemistry": 78,
  "Maths": 81,
  "CS": 78
}
```

dumps( ) function prints key:value pair in separate lines with the number of spaces which is the value of indent argument.

9.9 COUNTING FREQUENCY OF ELEMENTS IN A LIST USING DICTIONARY

Steps:

1. import the json module for pretty printing
2. Take a string from user
3. Create a list using split( ) function
4. Create an empty dictionary to hold words and frequency
5. Now make the word as a key one by one from the list
6. If the key not present in the dictionary then add the key in dictionary and count
7. Print the dictionary with frequency of elements using dumps( ) function.

Program:

```
import json

sentence=input("Enter a string: ")

L = sentence.split()

d={ }

for word in L:

    key=word

    if key not in d:

        count=L.count(key)

        d[key]=count


print("The frequqency of elements in the list is as follows: ")

print(json.dumps(d,indent=2))
```

9.10 DICTIONARY FUNCTIONS:

Consider a dictionary **marks** as follows:

>>> **marks = { "physics" : 75, "Chemistry" : 78, "Maths" : 81, "CS":78 }**

S. No.	Function Name	Description	Example
1	len( )	Find the length of a dictionary. **Syntax:** len (dictionary-name)	>>> len(marks) 4

2	clear( )	removes all elements from the dictionary
		Syntax:
		dictionary-name**.**clear( )
			>>> marks.clear( )
			
			>>> marks
			
			{ }

3.	get( )	Returns value of a key. **Syntax:** dictionary-name.get(key)	>>> marks.get("physics") 75

Note: When key does not exist it returns no value without any error.

>>> marks.get('Hindi')

>>>

4	items( )	returns all elements as a sequence of (key,value) tuples in any order. **Syntax:** dictionary-name.items( )	>>> marks.items() dict_items([('physics', 75), ('Chemistry', 78), ('Maths', 81), ('CS', 78)])

Note: You can write a loop having two variables to access key: value pairs.
>>> seq=marks.items()
>>> for i, j in seq:
        print(j, i)

OUTPUT:
75 physics
78 Chemistry
81 Maths
78 CS

5	keys( )	Returns all keys in the form of a list. **Syntax:** dictionary-name.keys( )	>>> marks.keys() dict_keys (['physics', 'Chemistry', 'Maths', 'CS'])
6	values( )	Returns all values in the form of a list. **Syntax:** dictionary-name.values( )	>>> marks.values() dict_values([75, 78, 81, 78])
7	update( )	Merges two dictionaries. Already present elements are override. **Syntax:** dictionary1.update(dictionary2)	

Example:
>>> marks1 = { "physics" : 75, "Chemistry" : 78, "Maths" : 81, "CS":78 }
>>> marks2 = { "Hindi" : 80, "Chemistry" : 88, "English" : 92 }
>>> marks1.update(marks2)
>>> marks1
{'physics': 75, 'Chemistry': 88, 'Maths': 81, 'CS': 78, 'Hindi': 80, 'English': 92}
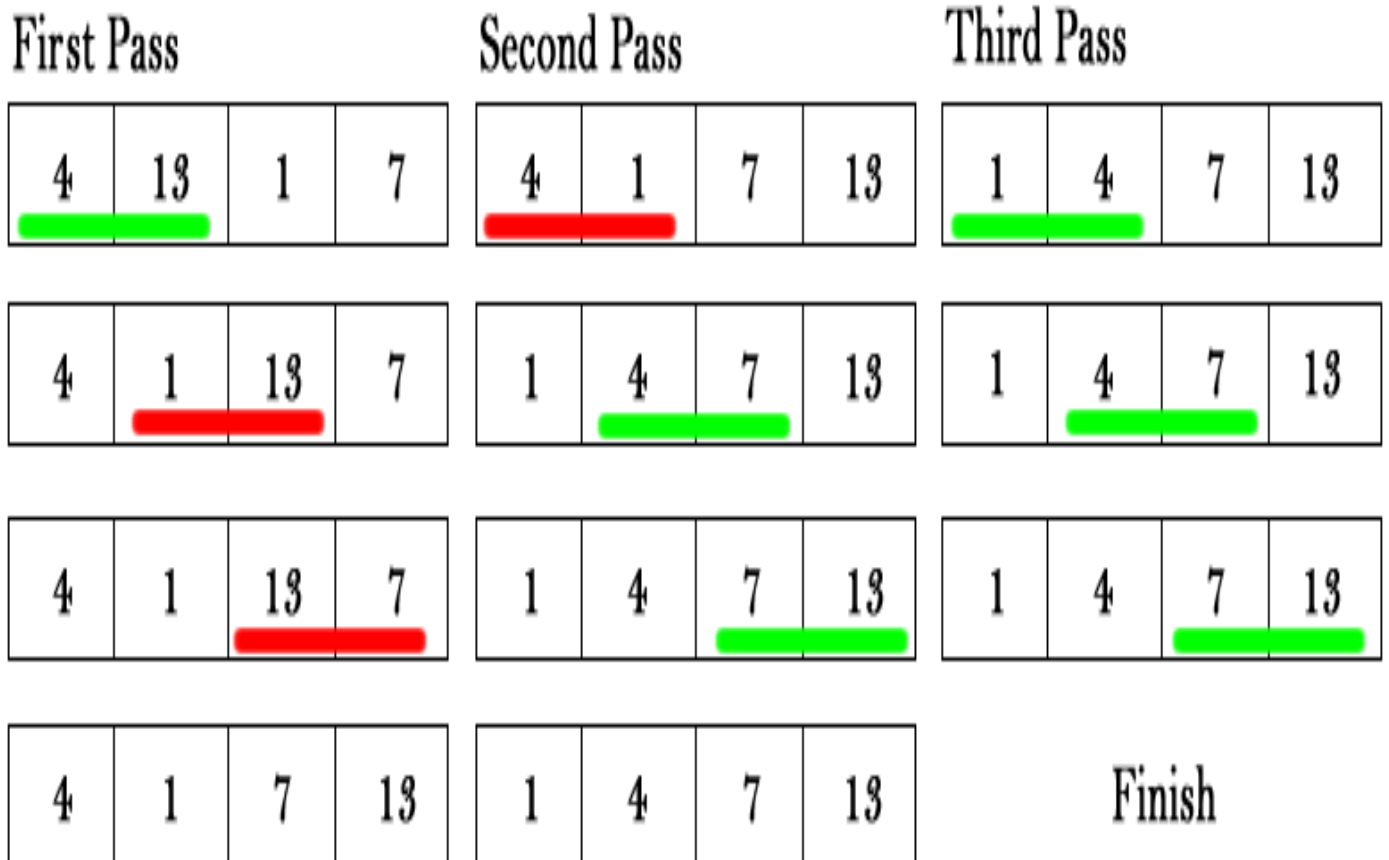
CHAPTER-10
SORTING

10.1 DEFINITION:

To arrange the elements in ascending or descending order.

In this chapter we shall discuss two sorting techniques:

1. Bubble Sort

2. Insertion Sort

1. BUBBLE SORT: Bubble sort is a simple sorting algorithm. It is based on comparisons, in which each element is compared to its adjacent element and the elements are swapped if they are not in proper order.

First Pass

4	13	1	7

4	1	13	7

4	1	13	7

4	1	7	13

Second Pass

4	1	7	13

1	4	7	13

1	4	7	13

1	4	7	13

Third Pass

1	4	7	13

1	4	7	13

1	4	7	13

Finish

PROGRAM:

```
L=eval(input("Enter the elements:"))

n=len(L)

for p in range(0,n-1):

    for i in range(0,n-1):

        if L[i]>L[i+1]:

            L[i], L[i+1] = L[i+1],L[i]

print("The sorted list is : ", L)
```
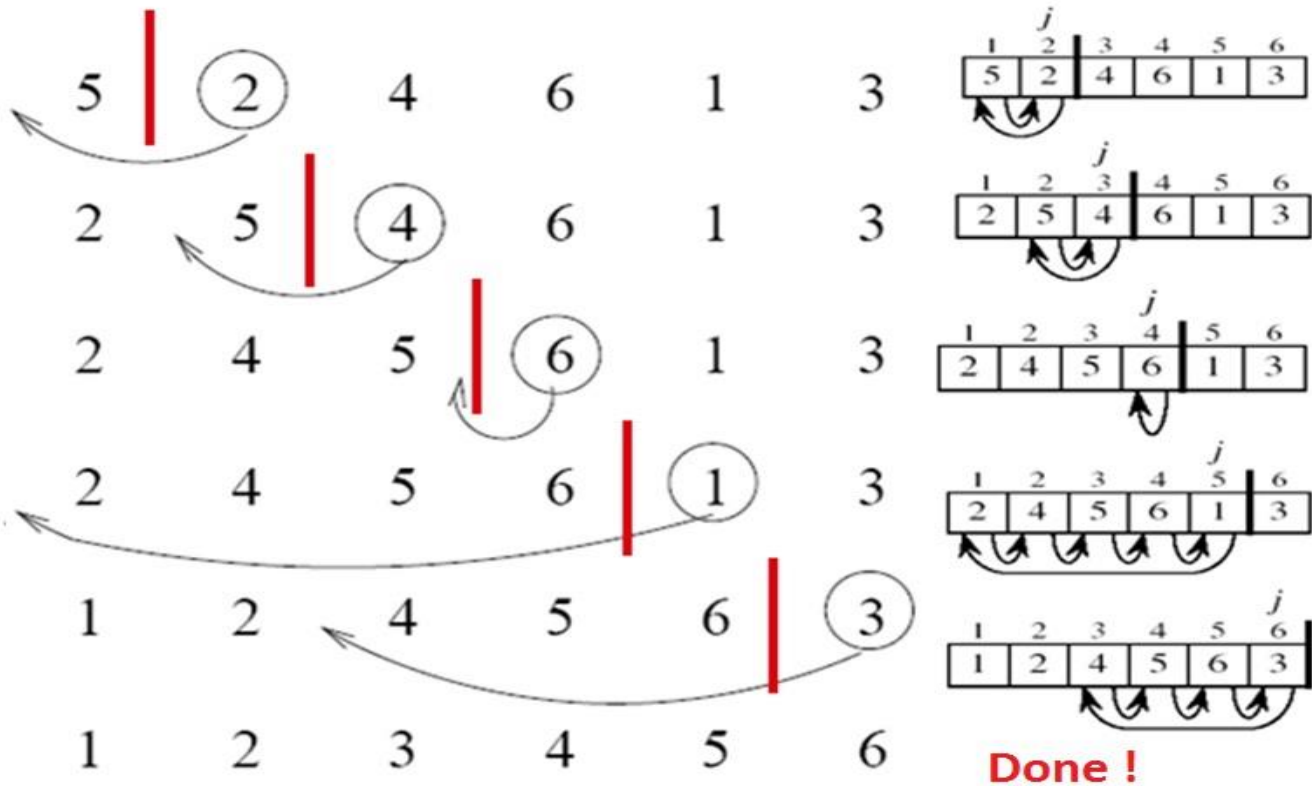
OUTPUT:

Enter the elements:[60, 24, 8, 90, 45, 87, 12, 77]

The sorted list is : [8, 12, 24, 45, 60, 77, 87, 90]

Calculating Number of Operations (Bubble sort):

Step	CODING	No. of Operations
1	L=eval(input("Enter the elements:"))	**1**
2	n=len(L) #for example n=7	**1**
3	for p in range(0,n-1):	one operation for each pass (executes **6** times, 0 to 5)
4	for i in range(0,n-1):	executes 6 times for elements, same will repeat 6 times under each pass (outer loop) so 6x6=**36** operations
5	if L[i]>L[i+1]:	executes 6 times for comparisons in each pass, same will repeat 6 times under each pass (outer loop) so 6x6=**36** operations
6	L[i], L[i+1] = L[i+1],L[i]	statement related to step-5, so 6x6=**36** operations
7	print("The sorted list is : ", L)	**1**
		TOTAL: 1+1+6+36+36+36+1=117 operations

2. INSERTION SORT: Sorts the elements by shifting them one by one and inserting the element at right position.

PROGRAM:

```python
L=eval(input("Enter the elements: "))

n=len(L)

for j in range(1,n):

    temp=L[j]

    prev=j-1

    while prev>=0 and L[prev]>temp:        # comparison the elements

        L[prev+1]=L[prev]              # shift the element forward

        prev=prev-1

    L[prev+1]=temp                  #inserting the element at proper position

print("The sorted list is :",L)
```

OUTPUT:

Enter the elements: [45, 11, 78, 2, 56, 34, 90, 19]

The sorted list is : [2, 11, 19, 34, 45, 56, 78, 90]

Calculating Number of Operations (Insertion Sort):

Step	CODING	No. of Operations
1	L=eval(input("Enter the elements:"))	**1**
2	n=len(L)　　　　　　#for example n=7	**1**
3	**for** j in range(1,n):	Executes **6** times
4	temp=L[j]	one operation at a time, executes 6 times so 1x6 = **6** operations
5	prev=j-1	one operation at a time, executes 6 times so 1x6= **6** operations
6	**while** prev>=0 and L[prev]>temp:	first time 1 comparison, second time 2, and so on. In this case 1+2+3+4+5+6=**21** operations
	L[prev+1]=L[prev]	first time 1 element shifted, second time 2 and so on. In this case 1+2+3+4+5+6= **21** operations
	prev=prev-1	**21** operations
	L[prev+1]=temp	element insertion at right place, **6** operations
7	print("The sorted list is : ", L)	**1**
	TOTAL: 1+1+6+6+6+21+21+21+6+1 = 90 operations	

10.2 How insertion sort is better than bubble sort?

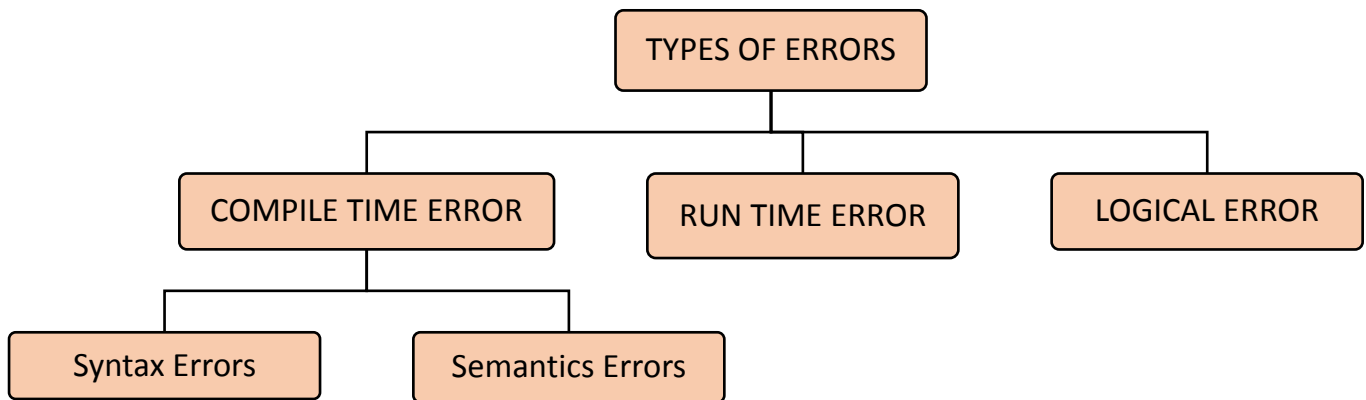| Bubble Sort	Insertion Sort
Inefficient	Efficient
Require more memory	Does not require additional memory
Slow	fast for small sequences
More operations	Less operations

DEBUGGING PROGRAMS

11.1 INTRODUCTION:

Errors in a program are known as 'bugs'.

To remove the errors from a program, known as debugging.

11.2 TYPES OF ERRORS:

```
                    ┌─────────────────┐
                    │ TYPES OF ERRORS │
                    └─────────────────┘
          ┌──────────────┼───────────────────┐
┌──────────────────┐ ┌────────────────┐ ┌────────────────┐
│COMPILE TIME ERROR│ │ RUN TIME ERROR │ │ LOGICAL ERROR  │
└──────────────────┘ └────────────────┘ └────────────────┘
      ┌──────┴───────┐
┌──────────────┐ ┌──────────────────┐
│ Syntax Errors│ │ Semantics Errors │
└──────────────┘ └──────────────────┘
```

1. Compile Time Error

       a. Syntax Error

       b. Semantics Error

2. Run Time Error

3. Logical Error

1. Compile Time Error: Compile time errors are those errors that occur at the time of compilation of the program.

There are **two types** of compile time errors:

a. Syntax Error: These errors occur due to *violation of grammatical rules* of a programming language.

for example:

```
a=10

b=20

if a>b                    #Syntax error, : is missing

        Print("a is greater than b")        # Syntax error, P is capital in Print statement
```

b. Semantics Error: Semantic errors occur when the statements written in the program are not meaningful.

Example:

```
a+b  = c     # expression cannot come on the left side of the assignment operator.
```

2. Run Time Error: Errors that occur during the execution or running the program are known as run time errors. When a program "crashed" or "abnormally terminated" during the execution, then this type errors are run time errors.

Example: When a loop executes infinite time.

```
a=1
while a<5:                 # value of a is always less than 5, infinite loop
        print(a)
        a=a-1
```

3. Logical Error: These types of errors occur due to wrong logic. When a program successfully executes but giving wrong output, then it is known as logical error.

Example:

```
a=10
b=20
c=30
average=a+b+c/3          #wrong logic to find the average
print(average)
```

OUTPUT:
40.0

11.3 Exception handling in python:

Exception: The unusual and unexpected condition other than syntax or logical errors, encountered by a program at the time of execution, is called exception.

The purpose of exception handling mechanism is to provide means to detect and report an exceptional circumstance, so that appropriate action can be taken.

Exception handling in python can be done using **try** and **except** blocks.

Syntax:

```
try:
    # Code that may generate an exception
except:
    # Code for handling the exception
```

Example:

```
num1=int(input("Enter first number :"))
num2=int(input("Enter second number: "))
try:
    r=num1/num2
    print("Result is :", r)
except:
    print("Divided by zero")
```

OUTPUT:

```
Enter first number :30
Enter second number: 0
Divided by zero
```

11.4 Steps to debug a program:

a. Carefully follow the syntax of a language

b. Spot the origin of error

c. Read the type of error and understand it well

d. Use intermediate print statements to know the value and type of a variable

e. Trace the code carefully

a. True and False

True and **False** are the results of comparison operations or logical (Boolean) operations in Python. For example:

>>> 3 = = 3

True

>>> 5 < = 4

False

>>> 7 > 2

True

>>> True or False

True

>>> True and False

False

Note:- True and False in python is same as 1 and 0.

Example:

>>> True = = 1

True

>>> False = = 0

True

>>> True + True

2

b. None

None is a special constant in Python that represents the absence of a value or a null value.

None does mean False, 0 or any empty list.

Example:

>>> None = = 0

False

>>> None = = False

False

>>> None = = [ ]

False

>>> x = None

>>> y = None

>>> x = = y

True

void functions that do not return anything will return a **None** object automatically. **None** is also returned by functions in which the program flow does not encounter a return statement. For example:

def   My_Function( ) :

    x = 5

    y = 7

    z = x + y

sum = My_Function( )

print(sum)

OUTPUT :

None

Another Example:

def ODD_EVEN(x) :

    if(x % 2) = = 0:

    return True

r = ODD_EVEN(7)

print(r)

OUTPUT :

None

Although this function has a return statement, it is not reached in every case. The function will return **True** only when the input is even.

c. *as*

    as is used to create an alias while importing a module.

Example:

>>> import math as mymath

>>> mymath.sqrt(4)

2.0

d. **assert**

assert is used for debugging purposes.

assert helps us to find bugs more conveniently.

If the condition is true, nothing happens. But if the condition is false, AssertionError is raised.

Syntax:

assert condition, message

example:

>>> x = 7

>>> assert x > 9, "The value is smaller"

Traceback (most recent call last):

File "<string>", line 201, in runcode

File "<interactive input>", line 1, in <module>

AssertionError: The value is smaller

e. **def**

    def is used to define a user-defined function.

Syntax:

def   function-name(parameters) :

f. del

del is used to delete the reference to an object. Everything is object in Python. We can delete a variable reference using **del**

Syntax: del variable-name

```
>>> a = b = 9

>>>del  a

>>> a

Traceback (most recent call last):

File "<string>", line 201, in tuncode

File "<interactive input>", line 1, in <module>

NameError : name 'a' is not defined

>>>b

9
```

.

del is also used to delete items from a list or a dictionary:

```
>>> x = [ 'p', 'q', 'r' ]

>>> del  x[1]

>>> x
```

['q', 'r']

g. except, raise, try

except, raise, try are used with exceptions in Python.

Exceptions are basically errors that suggests something went wrong while executing our program

try...except blocks are used to catch exceptions in Python.

We can raise an exception explicitly with the **raise** keyword.

Syntax:

```
try:
    Try-block
except exception1:
    Exception1-block
except exception2:
    Exception2-block
else:
    Else-block
finally:
    Finally-block
```

example:

```
def reciprocal(num):
    try:
        r = 1/num
    except:
        print('Exception caught')
        return
    return r

print(reciprocal(10))
print(reciprocal(0))
```

Output
0.1

Exception caught
None

h. finally

finally is used with try…except block to close up resources or file streams.

i. from, import

import keyword is used to import modules into the current namespace. from…import is used to import specific attributes or functions into the current namespace.

For example:
import math

will import the math module.
Now we can use the sqrt( ) function inside it as math.sqrt( ). But if we wanted to import just the sqrt( ) function, this can done using from as

Example :
from math import sqrt

now we can use the function simply as sqrt( ), no need to write math.sqrt( ).

j. global

global is used to declare that a variable inside the function is global (outside the function).
If we need to read the value of a global variable, it is not necessary to define it as global. This is understood.
If we need to modify the value of a global variable inside a function, then we must declare it with global. Otherwise a local variable with that name is created.

Example:
```
globvar = 10
def read1( ):
    print(globvar)
def write1( ):
    global globvar
    globvar = 5
def write2( ):
    globvar = 15

read1( )
```

```
        write1( )
        read1( )
        write2( )
        read1( )
```

Output
10
5
5

    k. in

in is used to test if a sequence (list, tuple, string etc.) contains a value. It returns **True** if the value is present, else it returns **False**. For example:

>>> a = [1, 2, 3, 4, 5]
>>> 5 in a
True
>>> 10 in a
False

The secondary use of **in** is to traverse through a sequence in for loop.
for i in 'hello':
   print(i)

Output
h
e
l
l
o

    l. is
                is keyword is used in Python for testing object identity.

While the = = operator is used to test if two variables are equal or not, **is** is used to test if the two variables refer to the same object.
It returns **True** if the objects are identical and **False** if not.

>>> True is True
True

```
>>> False is False
True
>>> None is None
True
```

We know that there is only one instance of **True**, **False** and **None** in Python, so they are identical.

```
>>> [ ] == [ ]
True
>>> [ ] is [ ]
False
>>> { } == { }
True
>>> { } is { }
False
```

An empty list or dictionary is equal to another empty one. But they are not identical objects as they are located separately in memory. This is because list and dictionary are mutable (value can be changed).

```
>>> " == "
True
>>> " is "
True
>>> ( ) == ( )
True
>>> ( ) is ( )
True
```

Unlike list and dictionary, string and tuple are immutable (value cannot be altered once defined). Hence, two equal string or tuple are identical as well. They refer to the same memory location.

    m. lambda

        lambda is used to create an anonymous function (function with no name). It is an inline function that does not contain a return statement. It consists of an expression that is evaluated and returned.

example:

```
a = lambda x: x*2
for i in range(1,6):
    print(a(i))
```

Output
2
4
6
8
10

Note: range (1,6) includes the value from 1 to 5.

n. nonlocal
The use of nonlocal keyword is very much similar to the global keyword. nonlocal is used to declare a variable inside a nested function (function inside a function) is not local to it.
If we need to modify the value of a non-local variable inside a nested function, then we must declare it with nonlocal. Otherwise a local variable with that name is created inside the nested function.

Example:
```python
def outer_function( ):
    a = 5
    def inner_function( ):
        nonlocal a
        a = 10
        print("Inner function: ",a)
    inner_function ( )
    print("Outer function: ",a)

outer_function( )
```

Output
Inner function: 10
Outer function: 10

Here, the inner_function( ) is nested within the outer_function( ).
The variable **a** is in the outer_function( ). So, if we want to modify it in the inner_function( ), we must declare it as **nonlocal**. Notice that a is not a global variable.
Hence, we see from the output that the variable was successfully modified inside the nested inner_function( ).

The result of not using the **nonlocal** keyword is as follows:

```python
def outer_function ():
    a = 5
```

```
def inner_function( ):
    a = 10
    print("Inner function: ",a)
inner_function( )
print("Outer function: ",a)
```

```
outer_function( )
```

Output
Inner function: 10
Outer function: 5

Here, we do not declare that the variable a inside the nested function is **nonlocal**. Hence, a new local variable with the same name is created, but the non-local a is not modified as seen in our output.

o. pass

pass is a null statement in Python. Nothing happens when it is executed. It is used as a placeholder.

Suppose we have a function that is not implemented yet, but we want to implement it in the future. Simply writing,

```
def function(args):
```
in the middle of a program will give us IndentationError. Instead of this, we construct a blank body with the pass statement.

```
def function(args):
    pass
```

p. while

while is used for looping.

```
i = 5
while(i):
    print(i)
    i = i − 1
```

Output
5
4
3

2
1

q. with

with statement is used to wrap the execution of a block of code within methods defined by the context manager.

Context manager is a class that implements **enter** and **exit** methods. Use of **with** statement ensures that the **exit** method is called at the end of the nested block.

Example

```
with open('Book.txt', 'w') as my_book:
    my_file.write(' Computer Science ')
```

This example writes the text Computer Science to the file Book.txt. File objects have **enter** and **exit** method defined within them, so they act as their own context manager.

First the **enter** method is called, then the code within with statement is executed and finally the **exit** method is called. **exit** method is called even if there is an error. It basically closes the file stream.

r. yield

yield is used inside a function like a return statement. But yield returns a generator.

Generator is an iterator that generates one item at a time. A large list of value will take up a lot of memory. Generators are useful in this situation as it generates only one value at a time instead of storing all the values in memory.

Example:

```
>>> g = (2**x for x in range(100))
```

will create a generator g which generates the values 2^0 to 2^{99}. We can generate the numbers using the next( ) function as shown below:

```
>>> next(g)
1
>>> next(g)
2
>>> next(g)
4
>>> next(g)
8
```

>>> next(g)

16

And so on…

This type of generator is returned by the **yield** statement from a function.

Example:

	Output

def generator( ):
    for i in range(6):
        yield i*i
 	
g = generator( )
for i in g:
    print(i)
	Output
	0
	1
	4
	9
	16
	25