PYTHON PROGRAMMING NOTES

I B.Tech-I Sem 2022-23

→ INTRODUCTION TO PYTHON:

* Python is a high-level, interpreted, interactive and object-Oriented scripting language.

Python is High-Level:
↳ Programing Language

Python is designed to be highly readable. It uses English Keywords frequently where as other languages use punctuation and it has fewer syntactical constructions than other languages.

Python is Interpreted:

Python is processed at runtime by the interpreter. You don't need to compile your program before executing it.

Python is Interactive:

You can actually sit at a Python prompt and interact with the interpreter directly to write your programs..

Python is Object-Oriented:

Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

HISTORY OF PYTHON :

* Python was developed by Guido van Rossum in the late 1980's and early 1990's at the National Research Institute for Mathematics and Computer Science in the Netherlands.

* Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix Shell and other scripting languages.

NEED OF PYTHON PROGRAMMING :

1. Software Quality :

   * Python focus on readability, coherence and software quality in general sets it apart from other tools in the scripting world.

   * Python has deep support for more advanced software reuse mechanisms, such as object-oriented programming (oop).

2. Developer Productivity

* Python code is typically one-third to one-fifth the size of equivalent C++ or Java code. that means there is less to type, less to debug and less to maintain

* Python program also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

3. Support Libraries:

* Python comes with a large collection of prebuilt and portable functionality, known as the standard library. this library supports an array of application-level programming tasks, from text pattern matching to network scripting.

* Python can be extended with both homegrown libraries and a vast collection of third-party application support software. Python's third-party domain offers tools for website construction, numeric programming, serial port access, game development and much more.

4. Easy to Understand:

* Being a very high-level language, Python reads like English, which takes a lot of syntax-learning stress off coding beginners.

* Python handles a lot of complexity for you, so it is very beginner-friendly in that it allows beginners to focus on learning programming concepts and not have to worry about too much details.

5. Very Flexible:

* As a dynamically typed language, Python is really flexible. this means there are no hard rules on how to build features and you'll have more flexibility solving problems using different methods.

* Python is also more forgiving of errors, so you'll still be able to compile and run your program until you hit the problematic part.

→ APPLICATIONS OF PYTHON :

* the following are the applications of Python in a wide range of areas :

1. Web Applications
2. Desktop Applications
3. Database Applications
4. Web Scraping
5. Web Mapping
6. Data Analysis
7. Interactive Web visualization
8. Computer vision for image and video processing
9. Object Oriented Programming

→ PYTHON IDENTIFIERS :

* A Python identifier is a name used to identify a variable, function, class, module or other objects.

* An identifier starts with a letter A-Z or a-z or an underscore (_) followed by zero or more letters, underscore and digits (0 to 9).

* the following are naming conventions for Python identifiers:

i) Class names start with an uppercase letter. All other identifiers start with a lowercase letter.

ii) Starting an identifier with a single leading underscore indicates that the identifier is private.

iii) Starting an identifier with two leading underscores indicating a strongly private identifier.

iv) If the identifier also ends with two trailing underscores the identifier is a language-defined special name.

→ PYTHON KEYWORDS :

* Keywords are reserved words and you cannot use them as Constant or Variable or any other identifiers names.

* All the python keywords contain lowercase letters only.

and	def	exec	if	not	return
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	Print	with
Continue	except	global	lambda	raise	Yield

VARIABLES :-

* Variables are nothing but reserved memory locations to store values. This means when you create a variable, you reserve some space in memory.
* Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory.

Assigning Values to Variables :

* Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

Eg:
```
a = 15          # An integer assignment
b = 3.12        # A float
c = "RISE"      # A string
```

Eg:
```
a = b = c = 2   # Assign single value to several variables
```

Eg:
```
a, b, c = 15, 3.12, "RISE"
# Assign multiple objects to multiple variables
```

→ INPUT - OUTPUT :

PYTHON INPUT :

* In Python, we have the input() function is allow to take the input from the user.

SYNTAX :

input ([prompt])

where prompt is the string we wish to display on the screen. It is optional.

Eg: >>> num = input ("Enter a number:")
Enter a number : 15
>>> num
15

PYTHON OUTPUT :

* In Python, we use the print() function to output data to the standard output device.

SYNTAX :

print (*objects , sep=' ' , end = '\n' ,
file = sys.stdout , flush = False)

Here, objects is the values to be printed.

→ the sep separator is used between the values. It defaults into a space character.

→ After all values are printed, end is printed. It defaults into a new line.

→ the file is the object where the values are printed and its default value is sys.stdout (screen).

UNIT-2

→ DATA TYPES

* the data stored in memory can be of many types.
* Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.
* Python has 5 standard data types :

1. Numbers
2. String
3. List
4. Tuple
5. Dictionary

1. Number :-

→ Number data types store numeric values.

→ Number objects are created when you assign a value to them.

Eg: num1 = 5
name = "RISE"

→ You can also delete the reference to a number object by using the del statement.

Syntax:

del var1 [, var2 [, var3 [..., varN]]]

→ Python supports 4 different numerical types :

i) int (signed integers)

ii) long (long integers, they can also be represented in octal and hexadecimal)

iii) float (floating point real values)

iv) Complex (Complex numbers)

2. Strings :-

→ Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

→ Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

→ the plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

Eg :-
```
str = 'Hello RISE'
Print str              # Prints Complete string
Print str[0]           # Prints first character of the string
Print str[2:5]         # prints characters from 2nd position to 5th - 1 position
```

```
Print str[2:]        # Prints string starting from 2nd index position
Print str * 2        # prints string two times
Print str + "TEST"   # Prints concatenated string
```

output :

```
Hello RISE
H
llo
llo RISE
Hello RISE Hello RISE
Hello RISE TEST
```

3. Lists :-

→ Lists are the most versatile of Python's compound data types.

→ A list contains items separated by commas and enclosed within square brackets ([]).

→ Lists are similar to array, one difference between them is that all the items belonging to a list can be of different data type.

Eg:-
```
a_list = [1, 2.5, "RISE"]
b_list = [5, "CSE"]
print a_list          # Prints Complete list
print a_list[0]       # prints first element of the list
print a_list[1:3]     # prints elements from 1st index
                        position to 3-1 index position
print a_list[1:]      # prints elements starting from 1st
                        index position
print b_list * 2      # prints list two times
print a_list + b_list # prints concatenated lists
```

Output:-

```
[1, 2.5, "RISE"]
1
[2.5, "RISE"]
[2.5, "RISE"]
[5, "CSE", 5, "CSE"]
[1, 2.5, "RISE", 5, "CSE"]
```

4. Tuples :-

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

→ the main difference between lists and tuples are:

List	TUPLE
1. Lists are enclosed in brackets [ ].	1. Tuples are enclosed in parenthesis ( ).
2. Lists can be updated.	2. Tuples cannot be updated.

→ Tuples can be thought of as read-only lists.

Eg:—

```
a_tuple = (1, 2.5, "RISE")
b_tuple = (5, "CSE")
print a_tuple          # Prints Complete list
print a_tuple[0]       # Prints first element of the list
print a_tuple[0:2]     # Prints elements from 0th index position to
                                    2-1 index position
print a_tuple[1:]      # prints elements starting from 1st
                                    index position
print b_tuple *2       # prints lists two times
print a_tuple + b_tuple # prints Concatenated lists.
```

Output :

```
(1, 2.5, "RISE")
1
(1, 2.5)
(2.5, "RISE")
(5, "CSE", 5, "CSE")
(1, 2.5, "RISE", 5, "CSE")
```

Eg:- tuple = (1 , 2.5 , "RISE")

list = [1 , 2.5 , "RISE"]

```
tuple[1] = 3.14      # Invalid Syntax with tuple
list[1] = 3.14       # valid Syntax with list
```

5. Dictionary :-

→ Python's dictionaries are like associative arrays or hashes found in Perl and consist of key-value pairs.

→ A dictionary key can be almost any Python type, but are usually numbers or strings. Values can be any arbitrary Python object.

Eg:-
```
dict = {'name': "RISE", 'code': 8, 'dept': "CSE"}
print dict['name']      # prints value for 'name' key
print dict              # prints Complete dictionary
print dict.keys()       # prints all the keys
Print dict.values()     # prints all the values
```

Output :-
```
RISE
{'name': 'RISE', 'code': 8, 'dept': 'CSE'}
['name', 'code', 'dept']
['RISE', 8, 'CSE']
```

OPERATORS :-

* Python language supports the following types of operators:

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

1. Arithmetic Operators :- Assume a = 10, b = 20

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
− Subtraction	Subtracts right hand operand from left hand operand.	a − b = −10
* Multiplication	Multiplies values on either side of the operator.	a * b = 200
/ Division	Divides left hand operand by right hand operand.	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder.	b % a = 0
** Exponent	Performs exponential calculation on operators	a ** b = 10 to the power 20
// Floor Division	the division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

2. Comparison Operators :

→ these operators compare the values on either sides of them and decide the relation among them. they are also called Relational operators.

→ Assume a=10 , b=20

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

3. Assignment Operators :

→ An assignment operator is the operator used to assign a new value to a variable.

→ Assume a = 10 and b = 20

Operator	Description	Example
=	Assigns values from right side operands to left side operand.	c = a+b assigns value of a+b into c.
+=	It adds right operand to the left operand and assign the result to left operand.	a += b which is equivalent to a = a+b
-=	It subtracts right operand from the left operand and assign the result to left operand.	a -= b which is equivalent to a = a-b
*=	It multiplies right operand with the left operand and assign the result to left operand.	a *= b which is equivalent to a = a*b
/=	It divides left operand with the right operand and assign the result to left operand	a /= b which is equivalent to a = a/b
%=	It takes modulus using two operands and assign the result to left operand.	a %= b which is equivalent to a = a%b
**=	Performs exponential (Power) calculation on operators and assign value to the left operand	a **= b which is equivalent to a = a**b
//=	It performs floor division on operators and assign value to the left operand.	a //= b which is equivalent to a = a//b

4. Logical Operators :

→ Logical operators are typically used with Boolean values.

→ Assume variable a holds True and variable b holds False:

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

5. Bit wise Operators :

→ Bitwise operator works on bits and performs bit-by-bit operation.

→ Assume if a = 60 and b = 13

Operator	Description	Example
& Binary AND	Operator Copies a bit to the result, if it exists in both operands.	128 64 32 16 8 4 2 1 a = 0011 1100 b = 0000 1101 a&b = 0000 1100 (a&b) = 12
| Binary OR	It copies a bit, if it exists in either operand.	a = 0011 1100 b = 0000 1101 a|b = 0011 1101 (a|b) = 61

^ Binary XOR	It copies the bit, if it is set in one operand but not both.	a = 0011 1100 b = 0000 1101 a^b = 0011 0001 (a^b) = 49
~ Binary one's Complement	It is unary and has the effect of 'flipping' bits.	a = 0011 1100 ~a = 1100 0011 ~a = -61
<< Binary Left shift	the left operand's value is moved left by the number of bits specified by the right operand.	a<<2 = 240 means 1111 0000
>> Binary right shift	the right operand's value is moved right by the number of bits specified by the right operand.	a>>2 = 15 means 0000 1111

6. Membership Operators :

→ Membership operators test for membership in a sequence such as strings, lists or tuples.

Operator	Description	Example
in	Evaluates to true, if it finds a variable in the specified sequence and false otherwise.	(x in y) is true when x is a member of sequence y.
not in	Evaluates to true, if it does not find a variable in the specified sequence and false otherwise.	(x not in y) is true, when x is not a member of sequence y.

7. Identity Operators :

→ Identity operators compare the memory locations of two objects.

Operator	Description	Example
is	Evaluates to true, if the variables on either side of the operator point to the same objects and false otherwise	(x is y) is true, when id(x) equals id(y)
is not	Evaluates to false, if the variables on either side of the operator point to the same object and true otherwise.	(x is not y) is true, when id(x) is not equals to id(y)

Operators Precedence :

* the following table lists all the operators from highest precedence to the lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus
* / % //	Multiply, divide, Modulo and floor division
+ -	Addition and Subtraction
>> <<	Right and left bitwise shift
&	Bitwise AND
^ |	Bitwise exclusive 'OR' and OR
<= < > >=	Comparison operators

<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

→ DECISION MAKING:

* Decision making is the anticipation of conditions occuring during the execution of a program and specified actions taken according to the conditions.

* Python programming language provides the following types of decision-making statements:

1. if statement
2. if...else statements
3. elif statements

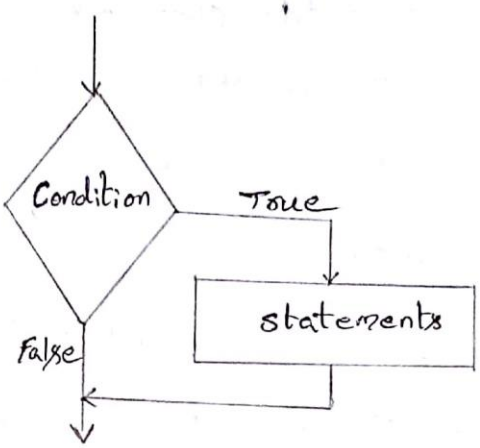
1. if Statement

⇒ the if statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

Syntax :-

```
if expression:
    statement(s)
```

Note : In Python, statements in a block are uniformly indented after the : symbol.



Example :

```
a = int (input ("Enter a value:"))
b = int (input ("Enter b value:"))
if (a > b):
    Print ("Largest element is:", a)
Print ("Good Bye.")
```

Output :

```
Enter a value : 5
Enter b value : 3
Largest element is : 5
Good Bye.
```

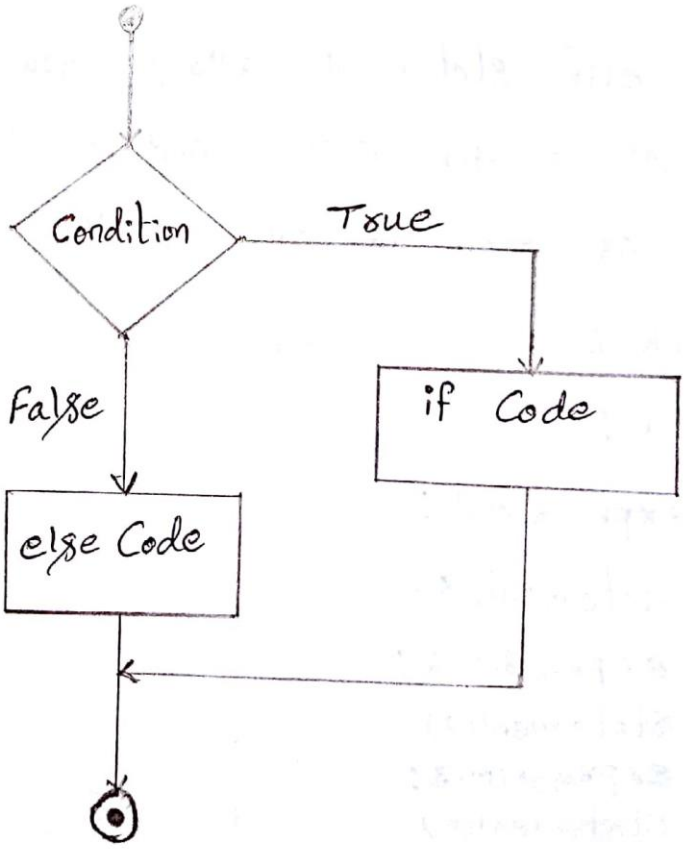
```
Enter a value : 5
Enter b value : 6
Good Bye.
```

2. if ... else Statements :

→ An else statement can be combined with an if statement. An else statement contains a block of code that executes if the conditional expression in the if statement resolves to 0 or a False value.

Syntax :-

```
if expression :
    Statement(s)
else :
    statement(s)
```



Example :

```
a = int(input("Enter a value:"))
b = int(input("Enter b value:"))
if (a > b):
    print("Largest number is :", a)
else:
    print("Largest number is :", b)
```

Output :

```
Enter a value : 5
Enter b value : 7
Largest number is : 7
```

3. elif Statement

→ The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Syntax :-

```
if expression1:
    Statement(s)
elif expression2:
    Statement(s)
elif expression3:
    Statement(s)
else:
    Statement(s)
```

Example:-

```
hours = int(input("Enter time in hours: "))
if (hours >= 6 and hours < 12):
    Print ("GOOD MORNING")
elif (hours >= 12 and hours < 16):
    Print ("GOOD AFTERNOON")
elif (hours >= 16 and hours < 19):
    Print ("GOOD EVENING")
else:
    Print ("GOOD NIGHT")
```

Output:

```
Enter time in hours: 9
GOOD MORNING
```

SINGLE STATEMENT SUITES

* If the suite of an if clause consists only of a single line, it may go on the same line as the header statement.

Example :-

```
num1 = int(input("Enter first number:"))
num2 = int(input("Enter second number:"))
if num1 > num2: print("num1," is greatest number")
else: print(num2," is greatest number.")
```

Output :

```
Enter first number: 5
Enter second number: 7
```

→ LOOPS :

* A loop statement allows us to execute a statement or group of statements multiple times.
* Python programming language provides the following types of loops :

1. while loop
2. for loop
3. Nested loops

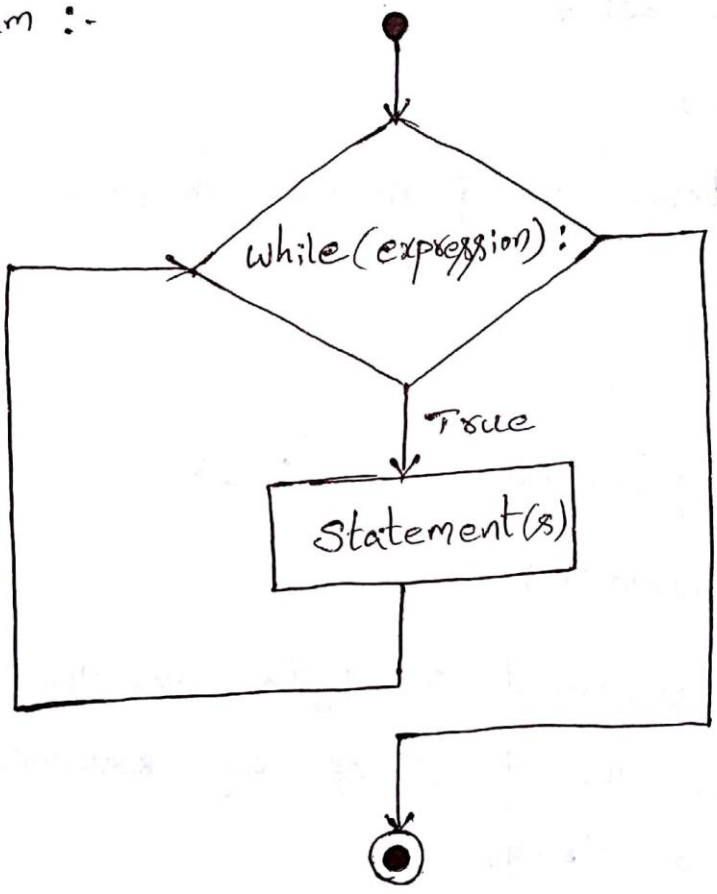
1. While Loop Statement :

→ A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax :-

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

Flow Diagram :-



→ Here, statement(s) may be a single statement or a block of statements with uniform indent. the condition may be any expression, and true is any non-zero value. the loop iterates while the condition is true.

→ When the condition becomes false, program control passes to the line immediately following the loop.

Example :

```
n = int(input("Enter number:"))
fact = 1
i = 1
while (i<=n):
    fact = fact * i
    i = i+1
Print("Factorial of given number is :", fact)
```

Output :

```
Enter number : 5
Factorial of given number is : 120
```

2. for Loop Statements :-

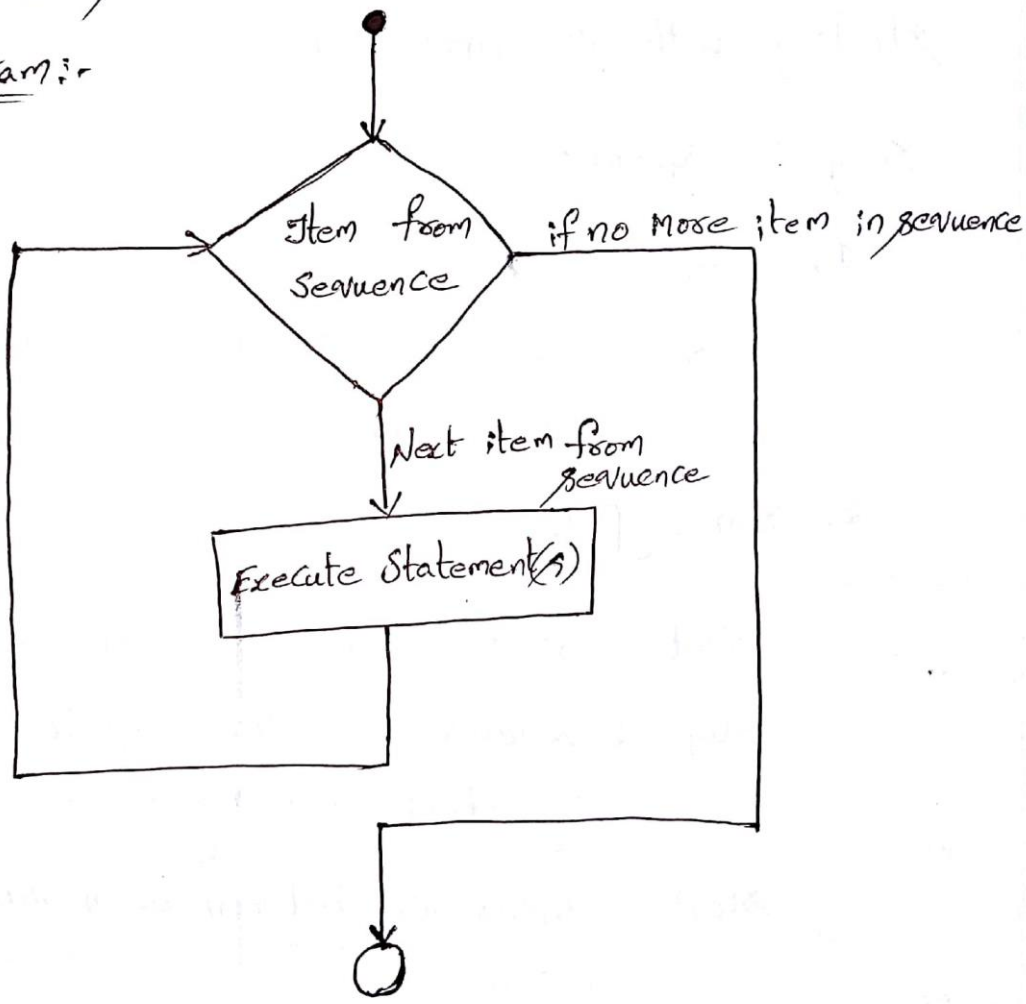
→ the for statement in Python has the ability to iterative over the times of any sequence, such as a list or a string.

Syntax :

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

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

Flow Diagram:-



Example:-

```
alist = [ 10, 20, 30]
for ele in alist:
    print(ele)
```

output :-

10
20
30

→ the built-in function range() is the right function to iterate over a sequence of numbers. range() generates an iterator to progress integers starting with 0 upto n-1

range() syntax :-

1. range (stop)

   stop : number of integers to generate, starting from zero

2. range([start], stop [, step])

   start : starting number of the sequence

   stop : Generate numbers up to, but not including this number.

   step : Difference between each number in the sequence.

Example:

```
alist = [10, 20, 30, 40]
for i in range(0, len(alist)):
    Print(alist[i])
```

Output:

```
10
20
30
40
```

Using else statement with Loops:

* Python supports having an else statement associated with a loop statement.
  - → If the else statement is used with a for loop, the else block is executed only if for loops terminates normally (and not by encountering break statement).
  - → If the else statement is used with a while loop, the else statement is executed when the condition becomes false

3. Nested Loops

→ Python programming language allows the use of one loop inside another loop.

Nested for loop Syntax:

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

Nested while loop Syntax:

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

Example:

```
for i in range(0,3):
    for j in range(0,3):
        Print((i, j))
```

Output:-

(0,0)
(0,1)
(0,2)
(1,0)
(1,1)
(1,2)
(2,0)
(2,1)
(2,2)

→ LOOP CONTROL STATEMENTS :-

* the loop control statements change the execution from its normal sequence

* Python supports the following control statements :

1. break statement
2. continue statement
3. pass statement

1. break Statement :

→ the break statement terminates the loop statement and transfers execution to the statement immediately following the loop.

Example :

```
for letter in 'Python':
    if (letter == 'h'):
        break
    Print ("Current letter : ", letter)
```

Output :

Current letter : P
Current letter : y
current letter : t

2. Continue Statement

→ The continue statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Example :

```
for letter in 'Python':
    if (letter == 'h'):
        Continue.
    Print ("Current letter :", letter)
```

Output :

Current letter : P
Current letter : y
Current letter : t
Current letter : O
Current letter : n

3. pass statement :

→ The pass statement is a null operation ; nothing happens when it executes.

→ The pass statement is also useful in places where your code will eventually go, but has not been written yet.

Example :

```
for letter in 'Python':
    if (letter =='h'):
        Pass
        print ("This is pass block")
    Print ("Current letter :", letter)
```

output :

```
Current letter : P
Current letter : y
Current letter : t
This is Pass block
Current letter : h
Current letter : o
Current letter : n
```

LISTS :

* the list is the most versatile datatype available in Python, which can be written as a list of comma separated values between square brackets.
* Important thing about a list is that the items in a list need not be of the same type.

Creating a List :

```
list1 = [ 10 , "RISE" , 8.5 , "CSE"]
```

Similar to string indices, list indices start at 0, and can be sliced, concatenated and so on.

Accessing Values in Lists

```
list1 = [ 10, "RISE" , 8.5 , "CSE" ]
Print("list1[2] : " , list1[2])
Print ( "list1[1:3] :" , list1[1:3])
```

output:-

```
list1[2]: 8.5
list1[1:3] : [ "RISE" , 8.5]
```

Updating Lists :

→ You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method.

Example :-

```
list1 = [10, "RISE", 8.5, "CSE"]
Print ("list1[1]:", list1[1])
list1[1] = "RGAN"
print ("list1[1]:", list1[1])
```

Output :

```
list1[1] : 'RISE'
list1[1] : 'RGAN'
```

Delete List Elements :

→ To remove a list element, you can use either the del statement if you know exactly which element you are deleting.

Example :

```
list1 = [ 10, "RISE", 8.5, "CSE"]
Print ("Before deleting list elements are :", list1)
del list1[2]
Print ("After deleting list elements are :", list1)
```

Output :

```
Before deleting list elements are :
        [ 10, "RISE", 8.5, "CSE"]
After deleting list elements are :
        [ 10, "RISE", "CSE"]
```

Basic List Operations :

1. len() – Find the length of the list.

Eg:- alist = [10, 20, 30]

```
Print ("Number of elements in list :", len(alist))
```

Output :-

Number of elements in list : 3

2. + - Concatenation of lists.

Eg:-

```
alist = [10, 20]
blist = [30, 40]
print (alist + blist)
```

Output:

```
[10, 20, 30, 40]
```

3. * - Repetition of list elements

Eg:-

```
alist = [10, 20, 30]
print ( alist * 2)
```

Output:-

```
[10, 20, 30, 10, 20, 30]
```

4. in - Membership operator which returns True when element present in list. False otherwise.

Eg:-

```
alist = [10, 20, 30, 40]
print ( 20 in alist)
print ( 50 in alist)
```

Output:-

```
True
False
```

Built-in Functions and Methods :

1. len() : this method returns the number of elements in the list.

Syntax :

```
len(list)
```

2. max() : this method returns the element from the list with maximum value.

Syntax :

```
max(list)
```

3. min() : this method returns the element from the list with minimum value.

Syntax :

```
min(list)
```

4. list() : this method takes sequence types and converts them to lists. This is used to convert a given tuple into list.

Syntax :

```
list(seq)
```

5. append() : this method appends a passed obj into the existing list.

Syntax :

```
list.append(obj)
```

6. Count() : this method returns count of how many times obj occurs in list.

Syntax :

```
list.count(obj)
```

7. extend() : this method appends the contents of seq to list.

Syntax :

```
list.extend(seq)
```

8. index() : this method returns the lowest index in list that obj appears.

Syntax :

```
list.index(obj)
```

9. insert() : this method inserts obj into list at offset index.

Syntax :

```
list.insert(index, obj)
```

10. pop() : this method removes and returns last obj from the list.

Syntax :

list.pop([index])

11. remove() : this method does not return any value but removes the given object from the list.

Syntax :

list.remove(obj)

12. reverse() : this method does not return any value but reverse the given object from the list.

Syntax :

list.reverse()

13. sort() : this method sorts objects of list, use compare function if given.

Syntax :

list.sort([func])

→ TUPLES :

* A tuple is a sequence of immutable Python objects.
* Tuples are sequences, just like lists.
* the main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. Tuples use parentheses, whereas lists use square brackets.

Creating a tuple :

```
tuple1 = (10, "RISE", 8.5, "CSE")
tuple2 = ();
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated and so on.

Accessing Values in Tuples :

```
tuple1 = (10, "RISE", 8.5, "CSE")
print("tuple1[2] : ", tuple1[2])
print("tuple1[1:3] :", tuple1[1:3])
```

output :-

```
tuple1[2] : 8.5
tuple1[1:3] : ("RISE", 8.5)
```

Updating Tuples :

→ Tuples are immutable, which means you cannot update or change the values of tuple elements.

Delete Tuple Elements :

→ Removing individual tuple elements is not possible.

→ To explicitly remove an entire tuple, just use the del statement.

Basic Tuples operations :

1. len() – Finds the length of the tuple.

Eg:-

```
atuple = (10, "RISE", 8.5, "CSE")
Print(len(tuple))
```

Output :

```
4
```

2. + – Concatenation

Eg :-

```
atuple = (10, 20, 30, 40)
btuple = (50, 60, 70)
Print (atuple + btuple)
```

Output :

```
(10, 20, 30, 40, 50, 60, 70)
```

3. * - Repetition

Eg:-

```
atuple = (10, 20, 30)
Print (atuple * 2)
```

Output:-

(10, 20, 30, 10, 20, 30)

4. in - Membership operator which returns True when an element is in tuple. False otherwise.

Eg:

```
atuple = (10, 20, 30)
Print (20 in atuple)
Print (60 in atuple)
```

Output:

True
False

Built-in Functions:

1. len() - this method returns the number of elements in the tuple.

Syntax:

```
len (tuple)
```

2. max() - this method returns the elements from the tuple with maximum value.

Syntax:

```
max (tuple)
```

3. min() - This method returns the element from the tuple with minimum value.

Syntax :

min(tuple)

4. tuple() - this method converts a list of items into tuples.

Syntax :

tuple(seq)

→ ① DICTIONARY :

* Dictionary is an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary).
* Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type ; strings and numbers can always be keys.

*Creating Dictionary :

```
adict = { 'rollno' : 8 , 'name' : "RISE" , 'branch' : "CSE"}
adict = { }    # Empty dictionary
```

Accessing Values in Dictionary :

```
adict = {'rollno' : 8 , 'name' : "RISE" , 'branch' : "CSE" }
print ("adict['branch']: " , adict['branch'])
```

output:-

```
adict['branch'] : 'CSE'
```

Updating Dictionary :

* You can update a dictionary by adding a new entry or a key-value pair.

Example:-

```
adict = {'rollno': 8 , 'name' : "RISE" , 'branch' : "CSE" }
print (adict)
adict['name'] = "RGAN"
adict['address'] = "Ongole"
Print (adict)
```

Output :

```
{'rollno' : 8 , 'name' : "RISE" , 'branch' : "CSE"}
{'rollno' : 8 , 'name' : "RGAN" , 'branch' : "CSE", 'address' : "ongole"}
```

Delete Dictionary Element :

* You can remove individual dictionary elements use del statement.

Example:-

```
adict = {'rollno' : 8 , 'name' : "RISE" , 'branch' : "CSE"}
Print (adict)
del adict ['rollno']
print (adict)
del adict
print (adict)
```

Note:- An exception is raised because after del dict, the dictionary does not exist anymore.

Output:-

```
{'rollno' : 8 , 'name' : "RISE" , 'branch' : "CSE"}
{'name' : "RISE" , 'branch' : "CSE"}
Error: adict is not defined
```

Properties of Dictionary Keys :

* these are two important points to remember about dictionary keys :

i) More than one entry per key is not allowed. this means no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment is to be considered.

ii) Keys must be immutable. this means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Built-in Dictionary Functions and Methods :

1. len() – this method returns length of the dictionary.

Syntax :

```
len(dict)
```

2. str() – this method produces a printable string representation of a dictionary.

Syntax :-

```
str(dict)
```

3. type() – this method returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.

Syntax :

```
type(dict)
```

4. clear() – this method removes all items from the dictionary.

Syntax :

```
dict.clear()
```

5. copy() – this method returns a copy of the dictionary.

Syntax :

```
dict.copy()
```

6. fromkeys() – this method creates a new dictionary with keys from seq and values set to value.

Syntax :

```
dict.fromkeys(seq [,value])
```

7. get() - this method returns a value from the given key. If the key is not available then returns default value None.

Syntax :

dict.get (key, default = None)

8. items() - this method returns a list of dictionary (key, value) tuple pairs

Syntax :

dict.items()

9. keys() - this method returns a list of all the available keys in the dictionary.

Syntax:

dict.keys()

10. setdefault() - this method is similar to get(), but will set dict[key] = default if the key is not already in dict.

Syntax :.

dict.setdefault(key, default = None)

11. update() - this method adds dictionary dict2's key-values pairs in to dict1. This function does not return anything.

Syntax :

dict1.update(dict2)

12. values() - This method returns a list of all values available in a given dictionary.

Syntax :

dict.values()

→ SETS :-

* A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed).

Create a set :

* A set is created by placing all the items (elements) inside curly braces { }, separated by comma or by using the built-in function set().

* It can have any numbers of items and they may be different types (integer, float, tuple, string etc.). But a set cannot have a mutable element like list or dictionary, as its elements.

```
aset = { 8.5, "RISE", (1,2,3) }
```

(or)

```
aset = set([8.5, "RISE", (1,2,3)])
```

* To create empty set:

```
aset = set()
```

Set Operations:

* Sets can be used to carry out mathematical set operations like union, intersection, difference and Symmetric difference.

→ Let us consider the following two sets:

```
aset = { 1, 2, 3, 4, 5}
bset = { 4, 5, 6, 7, 8}
```

i) Set Union:

→ Union of A and B is a set of all elements from both sets.

→ Union is performed using | operator

```
print ( aset | bset)
```

output:-

```
{ 1, 2, 3, 4, 5, 6, 7, 8 }
```

ii) Set Intersection:

→ Intersection of A and B is a set of elements that are common in both sets.

→ Intersection is performed using & operator

```
print ( aset & bset)
```

output:- { 4, 5}

iii) Set Difference :

→ Difference of A and B (A-B) is a set of elements that are only in A but not in B.

→ Similarly, B-A is a set of elements that are only in B but not in A.

→ Difference is performed using - operator.

```
print (aset - bset)
print (bset - aset)
```

Output

```
{1, 2, 3}
{ 6, 7, 8}
```

iv) Set Symmetric Difference :

→ Symmetric difference of A and B is a set of elements in both A and B except those that are common in both.

→ Symmetric difference is performed using ^ operator.

```
print (aset ^ bset)
```

Output

```
{1, 2, 3, 6, 7, 8}
```

Built-in Functions with set:

1. all() - Return True if all elements of the set are true (or if the set is empty).

Syntax:

```
all(set)
```

2. any() - Return True if any elements of the set is true. If the set is empty, return false.

Syntax:-

```
any(set)
```

3. len() - Return the length of the set.

Syntax:-

```
len(set)
```

4. max() - Return the largest item in the set.

Syntax:-

```
max(set)
```

5. min() - Return the smallest item in the set.

Syntax:-

```
min(set)
```

6. sorted() - Return a new sorted list from elements in the set.

Syntax:-

```
sorted(set)
```

7. sum() - Return the sum of all elements in the set.

Syntax:-

```
sum(set)
```

Set Methods :-

1. add() — Add an element to a set.

Syntax :-

```
set.add(element)
```

2. clear() - Remove all elements from a set.

Syntax :

```
set.clear()
```

3. copy() - Return a shallow copy of a set.

Syntax :

```
set.copy()
```

4. difference() - Return the difference of two or more sets as a new set.

Syntax :

```
set1.difference(set2)
```

5. difference_update() - Remove all elements of another set from this set.

Syntax :

```
set1.difference_update(set2)
```

6. intersection() - Returns the intersection of two sets as a new set.

Syntax :

```
set1.intersection(set2)
```

7. intersection_update() - Update the set with the intersection of itself and another.

Syntax:-

set1.intersection_update(set2)

8. isdisjoint() - Return True if two sets have a null intersection.

Syntax :-

set1.isdisjoint(set2)

9. issubset() - Return True if another set contains this set

Syntax :-

set1.issubset(set2)

10. issuperset() - Return True if this set contains another set.

Syntax:-

set1.issuperset(set2)

11. pop() - Remove and return an arbitary set element. Raise Keyerror if the set is empty.

Syntax:-

set.pop()

12. remove() - Remove an element from a set. If the element is not a member, raise a Keyerror.

Syntax:-

set.remove(element)

13. Symmetric_difference() - Return the symmetric difference of two sets as a new set.

Syntax:-

set1.symmetric_difference(set2)

14. Symmetric_difference_update() - Update a set with the symmetric difference of itself and another.

Syntax:

set1.symmetric_difference_update(set2)

15. Union() - Return the union of sets in a new set.

Syntax:-

set1.union(set2)

16. update() - Update a set with the union of itself and other.

Syntax:-

set1.update(list[,set2])

17. discard() - Remove an element from set if it is a member. Do nothing if the element is not in set.

Syntax:-

set.discard(element)

→ SEQUENCE :

* In Python, sequence is the generic term for an ordered set.
* there are several types of sequences in python, the following three are the most important:
  1. Lists
  2. Tuples
  3. Strings

→ LIST COMPREHENSIONS :

* Python supports computed lists called list comprehensions.

Syntax:-

```
list1 = [expression for variable in sequence]
```

where,
the expression is evaluated once, for every item in the sequence.

* List comprehensions help programmers to create lists in a a concise way. this is mainly beneficial to make new lists where each element is the obtained by applying some operations to each member of another sequence or iterable.

* List comprehension is also used to create a subsequence of those elements that satisfy a certain condition.

Example1:

```
squares = [(i**2) for i in range(11)]
Print(squares)
```

Output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

* You can also use the list comprehension to combine the elements of two lists.

Example:-

```
alist = [10, 20, 30]
blist = [10, 40, 50]
Print([(x,y) for x in alist for y in blist if(x!=y)])
```

Output:-

```
[(10,40), (10,50), (20,10), (20,40), (20,50), (30,10),
 (30,40), (30,50)]
```

UNIT-4

① DEFINING FUNCTIONS:

* A function is a block of organized reusable code that is used to perform a single, related action.

* Functions provide better modularity for your application and a high degree of code reusing.

* Python gives you many built-in functions like input(), print() etc., but you can also create your own functions. these functions are called user-defined functions.

* Here are simple rules to define a function in Python:

  i) Function blocks begin with the keyword def followed by the function name and parentheses(()).

  ii) Any input parameters or arguments should be placed within these parentheses.

  iii) the first statement of a function can be an optional statement - the documentation string of the function or docstring.

  iv) the code block within every function starts with a colon(:) and is indented.

  v) the statement return [expression] exits a function, optionally passing back an expression to the caller.

A return statement with no arguments is the same as return None.

Syntax :

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

→ CALLING A FUNCTION:

* Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

Example:

```
def display (str1):
    "this displays a passed string into this function"
    Print (str1)
    return
display ("Hello World")
```

output :.

Hello World

PASS BY REFERENCE VS VALUE

* All parameters in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Example:-

```
# Pass By Reference
def changelist(alist):
    "this changes a passed list into this function"
    Print("List elements inside the function before
          change : ", alist)
    alist[2] = 25
    Print("List elements inside the function after
          change : ", alist)
    return
alist = [10, 20, 30, 40, 50]
Print("Before calling function, list elements are :", alist)
changelist(alist)
Print("After calling function, list elements are : ", alist)
```

output:-

```
Before calling function, list elements are:
[10, 20, 30, 40, 50]
List elements inside the function before change:
[10, 20, 30, 40, 50]
List elements inside the function after change:
[10, 20, 25, 40, 50]
After calling function, list elements are:
[10, 20, 25, 40, 50]
```

Pass By Value

```
def changevalue(a):
    "this changes a passed value into this function"
    Print("Value of a inside the function before
                change:", a)
    a = a+1
    Print("Value of a inside the function after
                change:", a)
    return
```

```
a = 5
Print ("Before calling function, value of a is : ", a)
Changevalue(a)
Print ("After calling function, value of a is :", a)
```

Output :

```
Before calling function, value of a is : 5
Value of a inside the function before change : 5
Value of a inside the function after change : 6
After calling function, value of a is : 5
```

→ FUNCTION ARGUMENTS :

* You can call a function by using the following types of formal arguments :

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

1. Required Arguments :

→ Required arguments are the arguments passed to a function in correct positional order.

→ The number of arguments in the function call should match exactly with the function definition.

2. Keyword Arguments :

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

→ this allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

3. Default Arguments :

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

4. Variable-length Arguments :

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

Syntax :-

```
def functionname([formal_args,] * var_args_tuple):
    "function_docstring"
    "function_suite
    return [expression]
```

→ An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments. this tuple remains empty if no additional arguments are specified during the function call.

→ ANONYMOUS FUNCTIONS :

* Anonymous functions are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.
* Lambda forms can take any number of arguments but return just one value in the form of an expression. they cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression.
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Syntax :-

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

Example:

```
add = lambda arg1, arg2 : arg1 + arg2
Print (" Addition of two numbers is : ", add(10,20))
Print ("Addition of two numbers is : ", add(9, 7))
```

Output:

```
Addition of two numbers is : 30
Addition of two numbers is : 16
```

→ FUNCTION RETURNING VALUES (Fruitful Function):

* the statement return [expression] exits a function, optionally passing back an expression to the caller.

A return statement with no arguments is the same as return None.

Example:

```
def add ( x, y):
    z = x+y
    return z
a = int(input ("Enter a value: "))
b = int (input ("Enter b value: "))
C = add (a, b)
Print (" Sum of two numbers is : ", c)
```

Output:.

```
Enter a value : 9
Enter b value : 7
Sum of two numbers is : 16
```

SCOPE OF VARIABLES:

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

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

1. Global Variables

   The variables that are defined outside a function body have a global scope. This means that global variables can be accessed throughout the program body by all functions.

2. Local Variables

   The variables that are defined inside a function body have a local scope. This means that local variables can be accessed only inside the function in which they are declared.

Example :

```
a = 5                    # this is global variable
def display():
    a = a + 3            # Here a is local variable
    Print("Inside the function, value of a is :", a)
    return
display()
Print("Outside the function, value of a is :", a)
```

Output :

```
Inside the function, value of a is : 8
Outside the function, value of a is : 5
```

→ MODULE :

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

import Statement :-

→ You can use any Python source file as a module by executing an import statement in some other Python source file.

Syntax :

import module1 [, module2 [, ... moduleN]]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

Example :- Creating module welcome.py

```
# Creating a module
def greet(name):
    print("Welcome ", name)
    return
```

Program input_module.py that imports welcome.py

```
# Import module welcome
import welcome
name = input("Enter your name : ")
welcome.greet(name)
```

Output

```
Enter your name : RISE
Welcome RISE
```

→ from ... import STATEMENT :

* Python's from statement lets you import specific attributes from a module into the current name-space.

Syntax:

from modname import name1 [, name2 [, ... nameN]]

Example:-

→ To import the function greet from the module welcome.py, use the following statement:

```
>>> from welcome import greet
>>> greet("RISE")
Welcome RISE
```

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

```
from modname import *
```

NAMESPACES :

* A namespace is a container that provides a named context for identifiers. Two identifiers with the same name in the same scope will lead to a name clash.
* Namespaces enable programs to avoid potential name clashes by associating each identifier with the namespace from which it orginates.
* For Example,

```
# module1
def display (name):
    Print ("Hello ", name)
    return

# module2
def display (name):
    Print ("Welcome ", name)
    return

# main module
import module1
import module2
name= input ("Enter your name: ")
display (name)    # Ambiguous reference for identifier display
```

* In the example, module1 and module2 are imported into the same program. Each module has a function display(), which produces different results. When we call the display() from the main module, there will be a name clash as it will be difficult to determine which of these two functions should be called. Namespaces provide a means for resolving such problems.

* In Python, each module has its own space. This namespace includes the names of all items (functions and variables) defined in the module. Therefore, two instances of display(), each defined in their own module, are distinguished by being fully qualified with the name of the module in which each is defined as module1.display and module2.display. This is illustrated as follows:

```
import module1
import module2
name = input("Enter your name:")
module1.display(name)   #refers to display in module1
module2.display(name)   #refers to display in module2
```

Local, Global and Built-in Namespaces :

* During a program's execution, there are three main namespaces :

i) Local namespace - the local namespace has identifiers defined in the currently executing function.

ii) Global namespace - the global namespace contains identifiers of the currently executing module. and the I

iii) Built-in namespace - the built-in namespace contains names of all the built-in functions, constants, etc. that are already defined in python.

Example:-

```
def largest(numbers):                    # Global namespace
    large = 0                            # local namespace
    for i in numbers:
        if (i > large):
            large = i
    return large
numbers = [5, 4, 9, 3, 2]
Print("Largest number is : ", largest(numbers))
print("Sum of these numbers : ", Sum(numbers))
                                         #Built-in Namespace
```

output:-

Largest number is : 9

Sum of these numbers : 23

→ PACKAGES IN PYTHON

* A package is a hierarchical file directory structure that has modules and other packages within it. Like modules, you can very easily create packages in Python.

* Every package in Python is a directory which must have a special file called __init__.py. This file may not even have a single line of code. It is simply added to indicate that this directory is not an ordinary directory and contains a Python package.

In your programs, you can import a package in the same way as you import any module.

Syntax:-

```
import MyPackage.MyModule
        or
from MyPackage import MyModule
```

* the __init__.py is a very important file that also determines which modules the package exports as the API, while keeping other modules internal, by overriding the __all__ variable as shown below:

```
__init__.py
__all__ = ["MyModule"]
```

Key Points to Remember:

i) Packages are searched for in the path specified by sys.path.

ii) __init__.py file can be an empty file and may also be used to execute initialization code for the package or set the __all__ variable.

iii) the import statement first checks if the item is defined in the package. If it is unable to find it, an ImportError exception is raised.

iv) When importing an item using syntax like import item.subitem.subitem, each item except the last must be a package. That is, the last item should either be a module or a package. In no case it can be a class or function or variable defined in the previous item.

v) Packages have an attribute __path__ which is initialized with a list having the name of the directory holding the __init__.py file. The __path__ attribute can be modified to change the future searches for modules and sub-packages contained in the package.

CLASSES AND OBJECTS :

* Classes and objects are the two main aspects of object oriented programming. In fact, a class is the basic building block in Python.
* A class creates a new type and object is an instance of the class.
* Classes provides a blueprint or a template using which objects are created. In fact, in Python everything is an object or an instance of some class.

Defining Classes :

```
class  class_name:
        <statement-1>
        <statement-2>
          .
          .
          .
        <statement-N>
```

* Class definition starts with a keyword class followed by the class_name and a colon(:). The statement in the definition can be any of these - sequential instructions, decision control statements, loop statements and can even include function definitions.

* Variables defined in a class are called class variables and functions defined inside a class are called class methods. Class variables and class methods are together known as class members. the class members can be accessed through class objects.

Creating Objects :

* Once a class is defined, the next job is to create an object or instance of that class.
* the object can then access class variables and class methods using the dot operator(.).

Syntax :

```
object_name = class_name()
```

* the syntax for accessing a class member through the class object is

```
object_name.class_member_name
```

Example :

```
class A:
    rollno = 501
Aobj = A()
Print("Roll Number is", Aobj.rollno)
```

output:

Roll Number is 501

→ CLASS METHOD AND SELF ARGUMENT:

* Class methods are exactly same as ordinary functions that we have been defining so far with just one small difference. Class methods must have the first argument named as self. This is the first argument that is added to the beginning of the parameter list.

* the self argument refers to the object itself. That is, the object that has called the method. This means that even if a method that takes no arguments, it should be defined to accept the self.

Example:

```
class Addition:
    a = 10
    b = 20
    def result(self):
        c = self.a + self.b
        print("Sum of two numbers is", c)

obj = Addition()
print("Numbers are :", obj.a, obj.b)
obj.result()
```

Output:

Numbers are : 10 20

Sum of two numbers is 30

→ CLASS CONSTRUCTOR :

* the __init__() method has a special significance in Python classes.
* the __init__() method is automatically executed when an object of a class is created.
* the __init__() method can be defined as

```
def __init__(self, [args ....])
```

* Note the __init__() is prefixed as well as suffixed by double underscores.

Example:-

```
class ABC():
    def __init__(self, val):
        print("In class method")
        self.val = val
        print("the value is :", val)

obj = ABC(10)
```

output:

In class method

The value is : 10

→ INHERITANCE:

* the technique of creating a new class from an existing class is called inheritance. the old or existing class is called the base class and the new class is known as the derived class or subclass.
* In this process of inheritance, the base class remains unchanged. the concept of inheritance is therefore, frequently used to implement the 'is-a' relationship.

Syntax:

```
class DerivedClass(BaseClass):
    body_of_derived_class
```

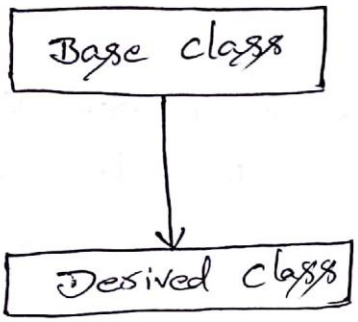
Types of Inheritance:

* Python supports 4 types of inheritance such as

1. Single inheritance
2. Multiple inheritance
3. Multi-level inheritance
4. Multi-path inheritance

1. Single Inheritance:

* In single inheritance, a class can be derived from a single base class.

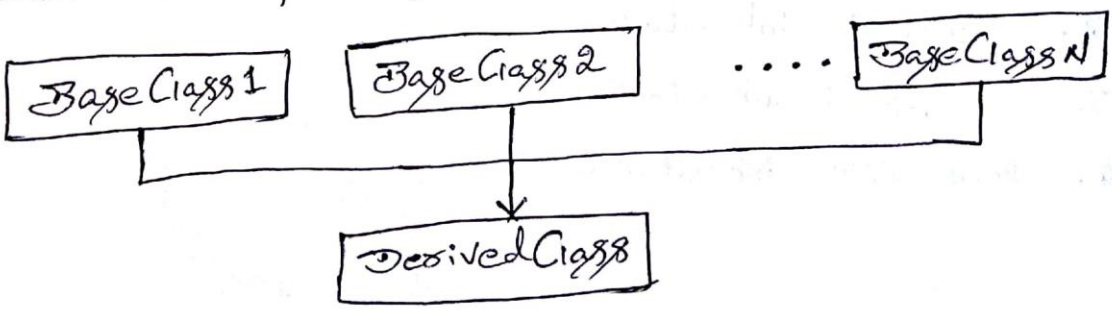


Syntax:-

```
class BaseClass:
    statement block
class DerivedClass(BaseClass):
    statement block
```

2. Multiple Inheritance:

* When a derived class inherits features from more than one base class it is called multiple inheritance. the derived class has all the features of both the base classes and in addition to them, can have additional new features.

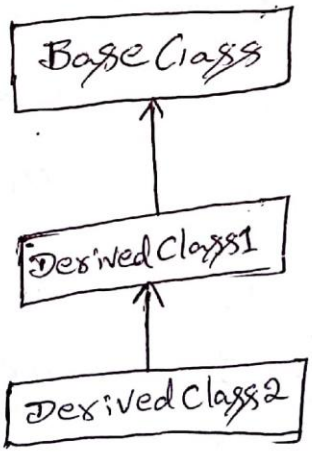


Syntax:-

```
class BaseClass1:
    statement block
class BaseClass2:
    statement block
class DerivedClass(BaseClass1, BaseClass2):
    statement block
```

3. Multi-level Inheritance:

* the technique of deriving a class from an already derived class is called multi-level inheritance.

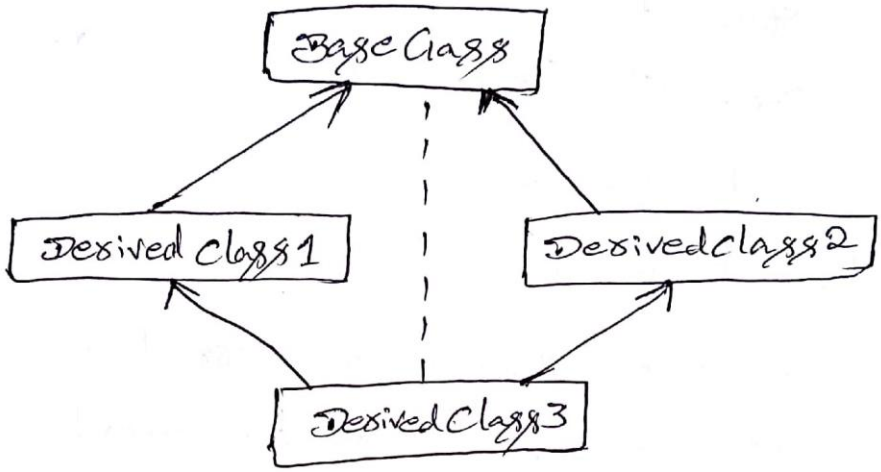


Syntax:-

```
class BaseClass:
    statement block
class DerivedClass1(BaseClass):
    statement block
class DerivedClass2(DerivedClass2):
    statement block
```

4. Multi-path Inheritance:

* Deriving a class from other derived classes that are in turn derived from the same base class is called multi-path inheritance.



→ POLYMORPHISM AND METHOD OVERRIDING:

* Polymorphism refers to having several different forms. It is one of the key features of OOP.

* Polymorphism enables the programmers to assign a different meaning or usage to a variable, function or an object in different contexts.

* In Python, method overriding is one way of Implementing Polymorphism.

* Method overriding is the ability of a class to change the implementation of a method provided by one of its ancestors.

Example:-

```
class BaseClass1(object):
    def __init__(self):
        Print("Base Class 1")
class BaseClass2(object):
    def __init__(self):
        Print("Base Class 2")
class DerivedClass(BaseClass1, BaseClass2):
    Pass
ObjDC = DerivedClass()
```

Output:-

```
Base Class 1
```

* In the above method, an object of derived class is made. Since there is no __init__() method in the derived class, the __init__() method of the first base class gets executed.
* A super() call can be made inside any method. This means that all methods can be modified via overriding and calls to super. The call to super can also be made at any point in the method; we don't have to make the call as the line in the method.

Example:-

```
class BaseClass1 (object):
    def __init__(self):
        Print ("Base class1")
        super (BaseClass1, self).__init__()
class BaseClass2 (object):
    def __init__(self):
        Print ("Base class2")
class DerivedClass (BaseClass1, Baseclass2):
    Pass
ObjDC = DerivedClass()
```

Output:-

```
Base class 1
Base class 2.
```

→ DATA HIDING :

* private instance variables that cannot be accessed except from inside an object doesn't exist in Python.
* There is a convention that is followed by most of the Python code: a name prefixed with an underscore should be treated as a non-public part of the API.

Example:-

```
class A:
    __value = 5
    def display(self):
        Print("In class value is:", self.__value)
objA = A()
objA.display()
Print("value is:", objA.__value)
```

* the above code shows error when it is executed because name attributes with a double underscore Prefix will not be directly visible to outsiders.
* You can access such attributes as object._classname__attributename.

Example:-

```
class A:
    __value = 5
    def display(self):
        Print("In class value is:", self.__value)
objA = A()
objA.display()
Print("value is:", objA._A__value)
```

Output:-

```
In class value is: 5
value is: 5
```

ERROR AND EXCEPTION:

* the programs that we write may behave abnormally or unexpectedly because of some errors and/or expections.
* the two common types of errors that we very often encounter are syntax errors and logic errors.
* Logic errors occur due to poor understanding of problem and its solution, Syntax errors arises due to poor understanding of the language.
* Exceptions are run-time anomalies or unusual conditions (such as divide by zero, accessing arrays out of its bounds etc.,) that a program may encounter during execution.
* Like errors, exceptions can also be categorized as Synchronous or asynchronous exceptions.
* Synchronous exceptions (like divide by zero, array index out of bound etc.,) can be controlled by the program.

Asynchronous exceptions (like an interrupt from the keyboard, hardware malfunction or disk failure) are caused by events that are beyond the control of the program.

Example:-

```
a = 5
b = 0
c = a/b
Print ("Result is :", c)
```

* In the above program, at line 3 we get an exception

i.e., ZeroDivisionError: integer division or modulo by zero

⇒ HANDLING EXCEPTIONS:

* We can handle exceptions in our program by using try block and except block. A critical operation which can raise exception is placed inside the try block and the code that handles exception is written in except block.

Syntax:-

```
try:
    statements
except ExceptionName:
    statements
```

* the try statement works as follows:

Step 1: First, the try block is executed.

Step 2a : If no exception occurs, the except block is skipped.

Step 2b : If an exception occurs, during execution of any statement in the try block, then,

i) Rest of the statements in the try block are skipped.

ii) If the exception type matches the exception named after the except keyword, the except block is executed and then execution continues after the try statement.

iii) If an exception occurs which does not match the exception named in the except block, then it is passed on to outer try block (in case of nested try blocks). If no exception handler is found in the program, then it is an unhandled exception and the program is terminated with an error message.

Example:-

```
num = int(input("Enter numerator :"))
deno = int(input("Enter denominator :"))
try :
    quo = num/deno
    print("Quotient :", quo)
except ZeroDivisionError:
    print("Denominator cannot be zero.")
```

Output:

Enter numerator : 5

Enter denominator : 0

Denominator cannot be zero

RAISING EXCEPTIONS:

* You can slowly raise an exception using raise keyword.

Syntax:-

raise [Exception [, args [, trackback]]]

Here, Exception is the name of exception to be raised. args is optional and specifies a value for the exception argument. If args is not specified, then the exception argument is None. the final argument, trackback is also optional and if present, is the trackback object is used for the exception.

Example 1:-

```
try:
    num = 10
    print (num)
    raise ValueError
except:
    print ("Exception occurred... Program Terminating...")
```

Output:

```
10
Exception occurred... Program Terminating...
```

Example 2:-

```
a = int(input("Enter numerator value:"))
b = int(input("Enter denominator value:"))
try:
    if (b==0):
        raise Exception(b)
    c= a/b
    print("c=",c)
except Exception as e:
    print("Denominator can not be ",e)
```

output:

```
Enter numerator value: 5
Enter denominator value: 0
Denominator can not be 0
```

→ USER-DEFINED EXCEPTION :-

* Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class, either directly or indirectly.

Example:-

```
class MyException(Exception):
    def __init__(self):
        self.msg = "Exception occurred"

try:
    raise(MyException())
except MyException as e:
    print(e.msg)
```