UNIT-1

- Introduction
- Basics

Introduction to Python:

Python is high level, interpreted, interactive and object-oriented programming language.

It was created by Guido van Rossum in 1991 and further developed by the Python software Foundation.

Python syntax allows programmers to express their concept in fewer line of codes and lets us work quickly and integrate systems more efficiently

Python is high-level:

According to basic principle of coding, the factor that makes a language high level is its distance from machine binary code. Hence being an interpreted language, which is not subject to processor, makes Python a high-level language.

Python is Interpreted:

Python is an interpreted language which uses interpreter. An interpreter executes the statement of code "line-by-line". whereas the compiler executes the code entirely and lists all possible errors at a time.

Python is interactive:

When a python statement is entered and is followed by the return key, the result will be printed on the screen. immediately- in the

Python is object-oriented:

Python is object-oriented because it is designed with OOP (object-oriented-Programming) approach and offers following advantages:

- Provides a clear program structure
- Facilitates easy maintenance and modification of existing code.

Characteristics of Python:

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

Python Syntax comparison to other Programming language:-

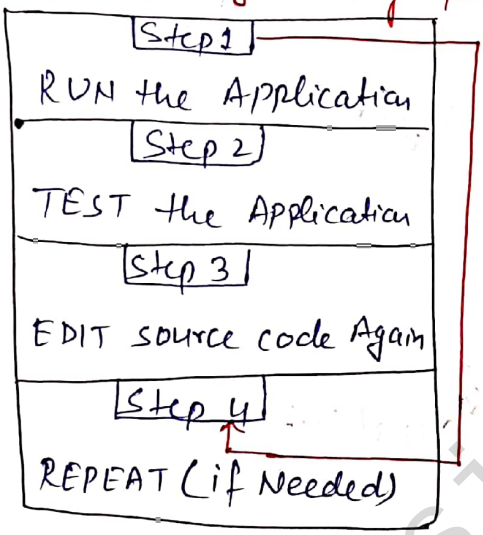
- Python has some similarity with english language with influence from mathematics.
- Python uses newlines to complete a command. Other languages generally uses parentheses or semicolon
- Python relies on indentation using whitespace to define scopes, such as scope of loops, functions & classes. Other languages uses curly braces for this.

ex: print ("Hello, world!")

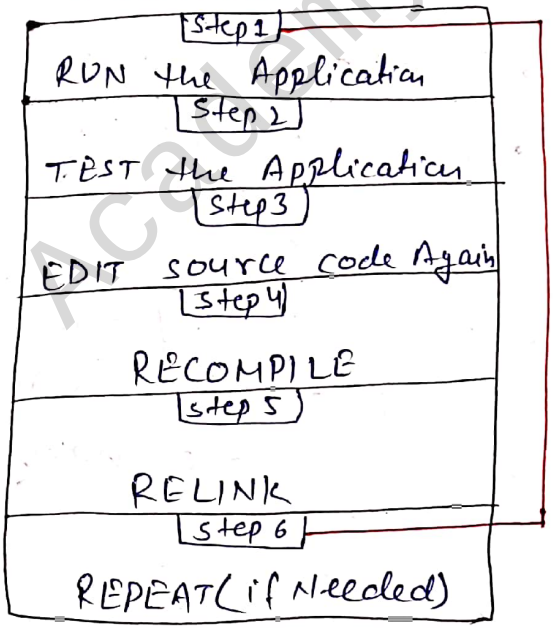
Programming Cycle for Python:

Python programming cycle is a little bit different from the traditional programming cycle. Python DO NOT have compile or link steps because of this behaviour Programs runs immediately after changes are made. The programming cycle of python is in rapid prototyping.

Python Programming Cycle



Others (Traditional)



Python IDE:

Following are the list of IDE available on Python for windows.

- PyCharm
- Jupyter Notebook
- Wing IDE
- Komodo IDE
- Sublime text
- Atom

Pycharm:

- Interactive Python console
- Support for web framework
- Faster refracting time.
- Lesser development

Jupyter Notebook:

- Compitability with almost every python module.
- Lesser space and hardware requirements.
- Inbuilt terminal and kernel features.
- A wide variety of widget can be applied.

Wing ide:

- Inbuilt debugging tools.
- Support for unit testing.
- Easy code navigation capability.

Komodo ide:

- Third party library support
- XML autocompletion.
- Inbuilt refracting capacity.

Sublime text:

- Cross - platform
- Multitasking
- Better customization.

Atom:

- Better customization
- Better user interface.
- Cross platform editor

Interacting with Python Programs:

To run a python program we can have various ways described below:

Run on IDLE (Integrated Development environment for Python):

- To run python program on IDLE, follow the given steps:
  - Write the python code and save it.
  - To run the program, go to Run > Run Module or simply click F5.

Run on Command line:

The python script file is saved with ".py" extension. After saving the file we can run it from the command line. On the cmd, type keyword 'Python' followed by the name of the file with which you saved the python script.

ex: Suppose we have a python script saved with "hello.py". To run it on command line, type the following.

```
Python hello.py
```

Run on IDE (PyCharm):

To run on PyCharm, we need to follow the steps as below:

Create a new python file and save it with some name, say "hello.py".

- After writing the required code in python file, we need to run it.
- To run, click on the "Green Play Button" at the top right corner of the IDE. The second way is, Right click and select "Run file in Python console" option.
- This will open a console box at the bottom and output will be shown there.

Run on Text Editor:

Steps are as follows:

Create a file with a name, let "hello.py". Write some python code in the file.

To run the code, Righ click > select Run code. Else press 'Ctrl + Alt + N' to run the code.

Scripting Vs interactive Modes in Python:

Interactive mode also known as REPL (read evaluate Print loop) provides us with a quick way of running blocks or single line of python codes. The code executes via python shell which comes with "Python" installation.
This mode is very suitable for beginners as it helps them to evaluate their code line by line and understand the execution as well.

How to run the code in interactive mode:

To run the code type "python" in command prompt. Then simply type the Python statement on >>> prompt. As we type and press enter we get the output in very next line.

Scripting Mode:

If you need to write long piece of python code or your Python script spans multiple files, interactive mode is not recommended. In such cases we require scripting mode.

In script mode we write our code in a text file then save it with a '.py' extension which stands for python.

Steps for running python code in script mode.

- Make a file using a text editor. You can use any text editor like notepad.
- After writing the code save the file using .py extension.
- Now open the command prompt directory to the one where your file is stored.
- Type python "filename.py" and press enter.
- Now we can see the o/p on the ~~current~~ command prompt.

Elements of Python :-

- Identifiers
- Keywords or reserve words.
- Lines and indentation.
- Comments in Python
- Quotation in Python.
- Python variables, constants and literals.

Identifiers in Python

Python keywords:

- Keywords are the reserve words in Python. We cannot use a keyword as a variable name, function name or any other identifier.
- They are used to define the syntax and structure of the Python language.
- In Python keywords are case sensitive. The list is given below.

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

Python Identifiers:

- A python identifier is a name used to identify a variable, function, class, module or other object.
- It helps to differentiate one entity from another.

Rules for writing identifiers:

1- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0-9) or an underscore '_'. - ex: myClass, var_1 etc.

2- An identifier cannot start with a digit. '1variable' is invalid, but variable1 is a valid name.

3- Keyword cannot be used as identifiers.

```
global = 1
```

```
O/P : File "<interactive input>", line 1
      global = 1
              ^
SyntaxError : invalid syntax.
```

4- We cannot use special symbols like !, @, #, $, % etc. in our identifier.

```
a@ = 0
```

```
O/P :- File "<interactive input>", line 1
       a@ = 0
        ^
SyntaxError : invalid syntax.
```

5- An identifier can be of any length.

Lines and Indentation :-

Python does not use braces ({}) to indicate blocks of code for class and function definition or flow control. Blocks of code are denoted by line indentation.

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

Quotation in Python :-

Python accepts single (' '), double (" ") and triple (''' ''') quotes to denote string literals, as long as same type of quotes starts and ends the string.

ex:- word = 'word'

Sentence = "This is a sentence"

paragraph = """This is a paragraph. It is made up of multiple paragraphs."""

Comments in Python:-

In python, we use the hash(#) symbol to start writing a comment.

Python interpreter ignores comments.

eg:-
```
# This is a comment
# print out hello
print('Hello')
```

Python variables, constants and literals:-

Python Variables:-

A variable is a named location used to store data in the memory. It is like containers that holds data that can be changed later in the program.

eg:- number = 10

number ↓ variable name

10 ↓ value assigned to variable.

we can change the values of variables accordingly.

eg:-
```
number = 10
number = 10.5
```

Assigning values to variable:-

eg:- Declaring and assigning values.
```
website = "apple.com"
print(website)
```
O/P= apple.com

eg-2 changing the value of variable.

```
website = "apple.com"
print(website)
#assigning new value to variable (website)
website = "programmer.com"
print(website)
```

O/P: apple.com
programmer.com.

Constants:

A constant is a type of variable whose value cannot be changed.

Declaring and assigning values to constant.

Create a "constant.py"

```
PI = 3.14
GRAVITY = 9.8
```

Create a "main.py"

```
import constant
print(constant.PI)
print(constant.GRAVITY)
```

O/P = 3.14
9.8

Literals:

Literals are the raw data given in a variable or constant. It can be either numeric, string, boolean or some special type of literals.

eg:

```
a = 0b1010   #Binary literals
b = 100      # Decimal "
c = 0o310    # Octal "
d = 0x12C    # Hexadecimal literals.
```

```
# float literals
float_1 = 10.5
float_2 = 1.5e2

print(a, b, c, d)
print(float_1, float_2)
```

O/P:- 10, 100, 200, 300
10.5, 150.0

eg:- String literals

```
strings = "This is a string"
char = "C"
print(strings)
print(char)
```

O/P:- This is a string
C

Python Input & Output:

- Python provide various built-in functions that are readily available to us at the Python prompt.
- input() and print() are widely used for standard input and output operations respectively

Python Output using print() function:

We use the print() function to get the output.

eg:- print('This is an output')
or
print("This is an output")

Note:- Both the statements are correct. In some version(older) single code is used while in newer one double quotes is being used.

eg2:-
```
a = 5
print ('The value of a is ', a)
```
o/p:- The value of a is 5

Formatting Output:

1- Using formatted string literals :-

We can use formatted string literals, by starting a string with 'f' or 'F' before opening quotation marks or triple quotation marks.

eg:-
```
# Declaring a variable
name = "abc"

# output
print(f'Hello {name}! How are you?')
```
o/p: Hello abc ! How are you ?

2- Using format()

We can also use format() function to format our output to make it look presentable. The curly braces works as placeholders. We can specify the order in which variables occur in the output.

eg:-
```
# initializing variables
a = 20
b = 10
# addition
sum = a+b
# subtraction
sub = a-b
# output
print('The value of a is {} and b is {}'.
      format(a,b))
```

print('{2} is the sum of {0} and {1}'.format(a, b, sum))

print('{sub_value} is the subtraction of {value_a} and {value_b}'.format(value_a = a, value_b = b, sub_value = sub))

O/P:- The value of a is 20 and b is 10
30 is the sum of 20 and 10
10 is the subtraction of 20 and 10

Using % Operator:-

We can use % operator. % values are replaced by 0 or more values of element. The formatting is similar to that of printf in C programming language.

- %d = integer
- %f = float
- %s = string
- %x = hexadecimal
- %o = octal.

eg:-
```
# Taking input from the user.
num = int(input("enter a value"))
add = num + 5

# output
print("The sum is %d" %add)
```

O/P:- enter a value 50
The sum is 55

Taking input in Python:

Python provides two input functions to read the input from the users.

- input (prompt)
- raw_input (prompt)

input()

This function first takes the input from the user and then evaluates the expression, which means python automatically identifies whether user entered a string or a number or list.

eg1: #Python Program showing use of input()

```
value=input(" enter a value:")
print (value)
```

O/P: Enter your value : 123
123
>>>

eg2

```
num = input(" enter number")
print(num)
name1 = input (" enter name")
print (name1)
```

O/P: Enter number : 123
123
Enter name : ArtiRanjan
ArtiRanjan

raw_input (): This function works in older versions

```
#Python program showing raw_input
g= raw_input("enter your name:")
print g
```

output: Enter your name : Arti
Arti

Python Input( ) more examples:

If we want to take input from the user then we have the input() function.

Syntax: input('prompt')

where prompt is the optional string that is displayed.

eg: Python user input

```
# taking input from the user
name = input("Enter your name:")

# output
print("Hello, "+name)
```

→ concatenation of two strings.

Output:

```
Enter your name: Arti Ranjan
Hello, Arti Ranjan
```

Note: Python takes all the inputs as a string input by default. To conver it to any other data type it requires typecasting. For this we can use int(), float() etc.

eg: integer input

```
# Taking input from the user as integer
num = int(input("Enter a number:"))

add = num + 1

# output
print(add)
```

output:

```
Enter a number: 25
26
```

Type Conversion:

The process of converting the value of one data type to another data type is called type conversion. Python has two types of type conversion.

1- Implicit Type Conversion.
2- Explicit Type Conversion.

Implicit Type Conversion:

In implicit type conversion, Python automatically converts one data type to another data type. This process doesnot need any user involvement.

eg: Converting integer to float.

```
x = 10
print("x is of type:", type(x))

y = 10.6
print("y is of type:", type(y))

x = x+y
print(x)
print("x is of type:", type(x))
```

o/p:
```
x is of type: <class 'int'>
y is of type: <class 'float'>
20.6
x is of type: <class 'float'>
```

Here type of x is automatically converted to the "float" type from "integer" type. This is a simple case of implicit type conversion.

Explicit Type Conversion:

In explicit type conversion in Python, the data is manually changed by the user as per their requirement. Various forms of explicit type conversion is given below.

1- **int(a, base):**

This function converts any data type to integer. 'Base' specifies the "base in which string is" if the data is of type string.

2- **float():**

This function is used to convert any data type to a floating-point number.

ex:-
```
# python program to demonstrate Type conversion
# using int(), float()

# initializing a string
s = "10010"

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

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

O/P: After converting to integer base 2 : 18
After converting to float : 10010.0

3- **ord():**

This function is used to convert a character to string.

4- **hex():**

used for converting hexadecimal to string.

5- oct():

This function converts octal integer to octal string

ex:
```
# Type conversion using ord(), hex() oct()
# initializing integer.
S = '4'
# printing character converting to string.
c = ord(S)
print("After converting character to integer:", end=" ")
print(c)

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

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

O/P:- After converting character to integer : 52
After " 56 to hexadecimal string : 0x38
After converting 56 to octal string : 0070

6- tuple :- This function is used to convert to a tuple

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

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

9- chr(number):- This function is used to convert number to its corresponding ASCII character.

ex:
```
# convert to ASCII value of number.
a = chr(76)
b = chr(77)
```

```
print(a)
print(b)
```

o/p:- L
M

Basics:- Expressions and operators- precedence.

Expressions with operators in Python:-

An expression is a combination of operators and operands that is interpreted to produce some other value.

In any programming language, an expression is evaluated as per the precedance of its operators. So that if their is more than one operator in an expression, their precedance decided which operation will be performed first.

The various types of expressions are as follows.

1 Constant expression.
2 Arithmatic "
3 Integer expression.
4 Floating expressions
5 Relational expressions
6 Logical expressions
7 Combinational Expressions

The various operators used in various expressions are as follows:-

1- Arithmatic operators
2- Assignment operators
3- Comparison Operators
4- Logical Operators.
5- Identity Operators
6- Membership Operators
7- Bitwise operators.

Python Arithmatic Operators:

Arithmatic operater are ~~performed~~ used to perform mathematical operations.

Operater	Name	Example.
+	Addition	x+y
-	Subtraction	x-y
*	Multiplication	x*y
/	Division	x/y
%	Modulus	x%y
**	Exponentiation	x**y
//	Floor Division	x//y.

Python Assignment Operators:

Assignment operater are used to assign values to variables.

Operater	Example.
=	x=5
+=	x+=3
-=	x-=3
*=	x*=3
/=	x/=3
%=	x%=3
//=	x//=3
**=	x**=3
&=	x&=3
|=	x|=3
^=	x^=3
>>=	x>>=3
<<=	x<<=3

Python Comparison Operators:

Comparison operators are used to compare two values.

Operator	Name	Example.
==	Equal	x == y
!=	not equal	x != y
>	greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python Logical Operators:

Logical operators are used to combine conditional statements.

Operator	Description	Example.
and	Returns true if both statements true	x < 5 and x < 10
or	Returns true if one of the statement true	x < 5 or x < 4
not	Reverse the result, returns false if the result is true.	not(x < 5 and x < 10)

Python Identity operators:

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with same memory location.

Operator	Description	Example.
is	Returns true if both variables are the same object.	x is y
is not	Returns true if both variables are not the same object.	x is not y

Python Membership Operators:-

Membership operators are used to test if a sequence is present in an object.

Operator	Description	Example.
in	Returns true if a sequence with the specified value is present in the object.	x in y
not in	Returns true if a sequence is not present in the object (with specified sequence)	x not in y.

Python bitwise Operators:-

Bitwise operators are used to compare (binary) numbers).

Operator	Name	Description.
&	AND	Sets each bit 1 if both bits are 1
|	OR	Sets each bit 1 if one of two bit is 1
^	XOR	Sets each bit to 1 if only one of two bit is 1.
~	NOT	Inverts all bits.
<<	left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off.
>>	right shift.	Shifts right by pushing copies of the leftmost bit in from the left.

Python Booleans:-

Booleans represents one of two values true or false.

Boolean Values:-

When two variables are compared, the expression is evaluated and python returns the boolean answer. True or false.

example:-

```
print(10>9)
print(10==9)
print(10<9)
```

o/p: True
False
False.

eg: Print a message based on whether the condition is True or False.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

o/p: b is not greater than a.

Precedence and Associativity of Operators in Python.

There can be more than 1 operator in an expression. To evaluate these type of expressions there is a rule of precedence in Python. It guides the order in which these operations are carried out.

ex:
```
# Multiplication has higher Precedence.
# than subtraction.
>>> 10-4*2
2
```

But we can change this order by using parantheses [()] as it has higher precedance than multiplication.

eg:
```
# Parentheses has higher precedence.
>>> (10-4)*2
12
```

Associativity of Python Operators:

When two or more operators have same precedance, associativity helps us to determine the order of operation.

Almost all the operators have left-to-right associativity.

eg: Multiplication and floor division has same precedence. Hence if both of them are present in an expression, the

left one will be evaluated first.

Left to right associativity.

output 3:

Print(5*2 //3)

shows left-right associativity

Print(5*(2//3)) # output 0

Output

3

0

Note: Exponent operator has right-to-left associativity. in Python

Eg: print(2**3**2) # output 512, since

~~2**(3**2)~~

2**(3**2) = 2**9

= 512

if 2 needs to be exponated first, need to

use ()

output 64

print((2**3)**2)

Non-Associative Operators:

Some operators like assignment operator and comparison operator do not have associativity in Python. There are separate rules for sequences of this kind of operators, and cannot be expressed as associativity.

UNIT-2 → Conditionals
→ Loops.

Conditionals in Python:

Decision making in a programming language is automated using conditional statements, in which Python evaluates the code to see if it meets the specified conditions. The conditions are evaluated and processed as true or false.

Conditional statement in Python :-

- if-else
- Nested if
- Elif

i- if-statement:

A Python if statement evaluates whether a condition is equal to true or false. The statement will execute a block of code if a specified condition is equal to true. Otherwise the block of code within the if statement is not executed.

Syntax:

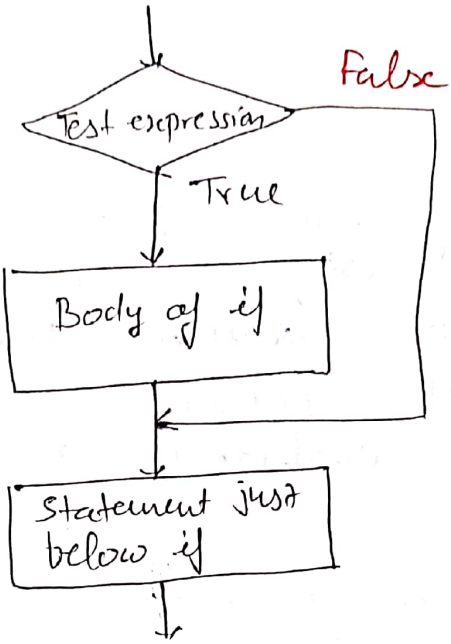
```
if condition :
    # statement to execute if
    # condition is true.
```

ex:

```
# if statement example.
if 10>5:
    print("10 is greater than 5")
print("Program ended")
```

O/P: 10 is greater than 5
program ended

Flowchart of if statement:



Note: Indentation (white space) is used to delimit the block of code. As shown in previous example it is mandatory to use indentation in Python3 coding.

if-else statement:

An if-else Python statement evaluates whether an expression is true or false. If a condition is true the "if" statement executes. Otherwise the "else" statement executes.

Syntax:

```
if (condition):
    # execute the block of if
    # condition is true

else:
    # execute the block of if
    # condition is false.
```

ex:-1

```
# if-else statement example
x = 3
if x == 4
else: print("yes")
      print ("NO")
```

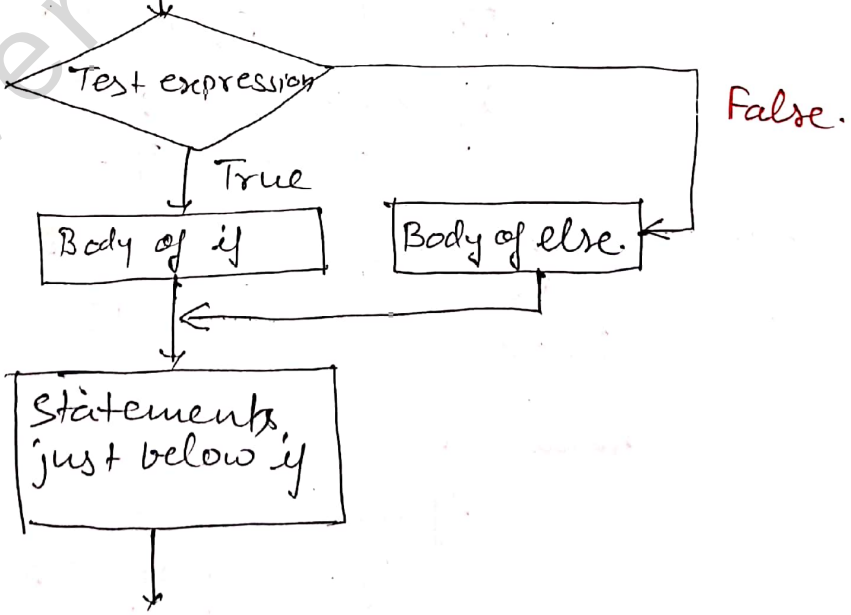
o/p:- NO

Ex:- 2 we can also chain if-else statement with 2-3 more than one condition.

```
#if -- else chain statement
letter = "A"
if letter == "B":
    print(letter is B")
else
    if letter == "C":
        print(letter is C")
else:
    if letter == "A":
        print("letter is A")
    else:
        print(" letter is not A, B, and C")
```

O/P: letter is A

Flow chart of if-else:



Nested if Statements:

If statement can also be checked inside other if statement. This conditional statement is called a nested if statement. This means that inner if condition will be checked only if outer if condition is true and we can see multiple conditions to be satisfied.

Syntax:-

```
if condition 1:
    # Execute when condition 1 is true
    if condition 2:
        # Executes when condition 2 is true
        # if block is end here.
    # if Block is end here
```

Eg:-

```
#Nested if statement example
num = 10
if num > 5:
    print("Bigger than 5")
    if num <= 15:
        print("Between 5 and 15")
```

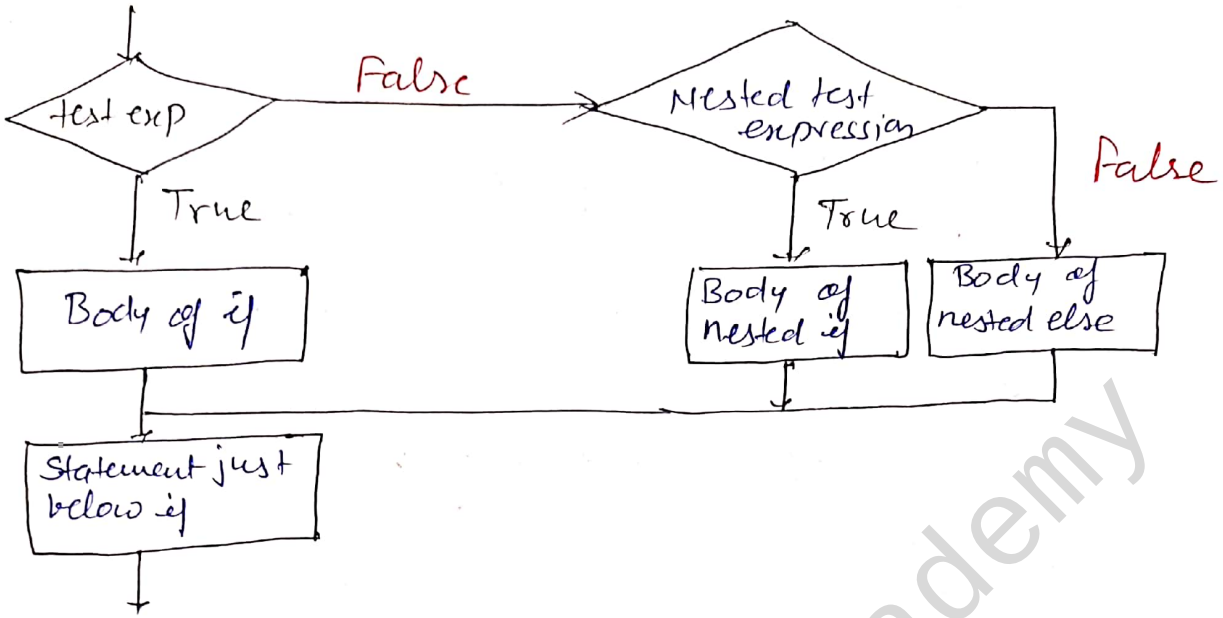
O/P:- Bigger than 5
Between 5 and 15

Eg:-2

```
num = 15
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive Number")
else:
    print("Negative Number")
```

o/p : Positive Number

Flowchart for Nested if :



if - elif Statement :

The if-elif statement is shortcut of if-else chains. While using if-elif statement at the end else block is added which is performed if none of the above if-elif statement is true.

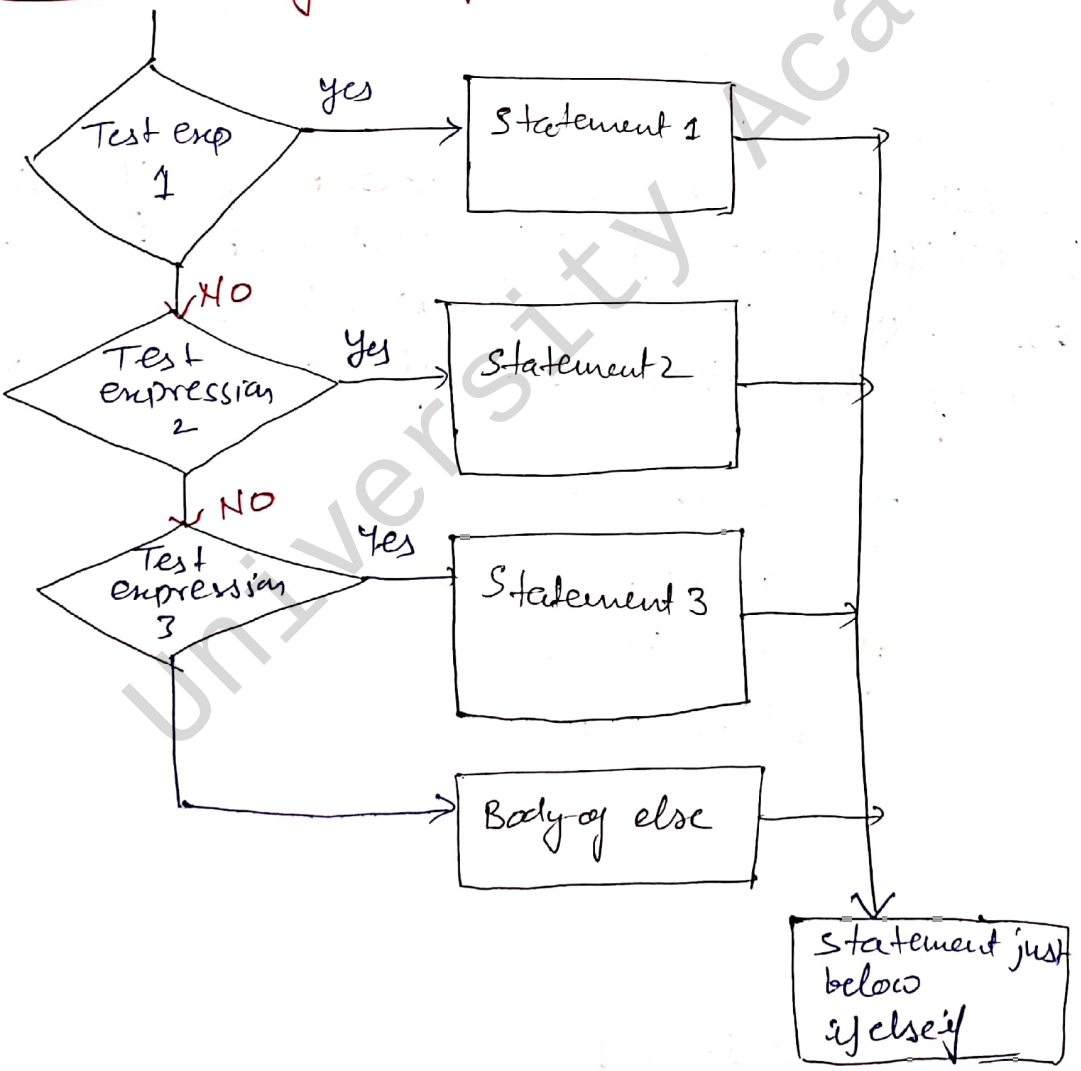
Syntax :

```
if condition :
    statement
elif condition :
    statement
.
.
.
else:
    Statement
```

ex:

```
# if-elif example
x = 5
y = 10
z = 22
if x > y:
    print("x is greater than y")
elif x < z:
    print("x is less than z")
else:
    print("if and elif never ran")
```

Flowchart of if-elif:-



if-elif-else ladder:

Here a user can decide among multiple options. The if statements are executed from the top to bottom. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and rest of the ladder is bypassed. If none of the condition is true, then the final else statement will be executed.

Syntax:

```
if condition:
    statement
elif condition:
    statement
.
.
.
else:
    statement
```

eg:

```
# Python Program to illustrate if-elif-else ladder.
i = 20
if i == 10:
    print("i is 10")
elif i==15:
    print("i is 15")
elif i==20:
    print("i is 20")
else:
    print("i is not present")
```

O/P:

```
i is 20
```

Short hand if :

Whenever there is only one statement to be executed inside the if block then shorthand if can be used.

Syntax:

```
if condition:
    statement
```

eg:

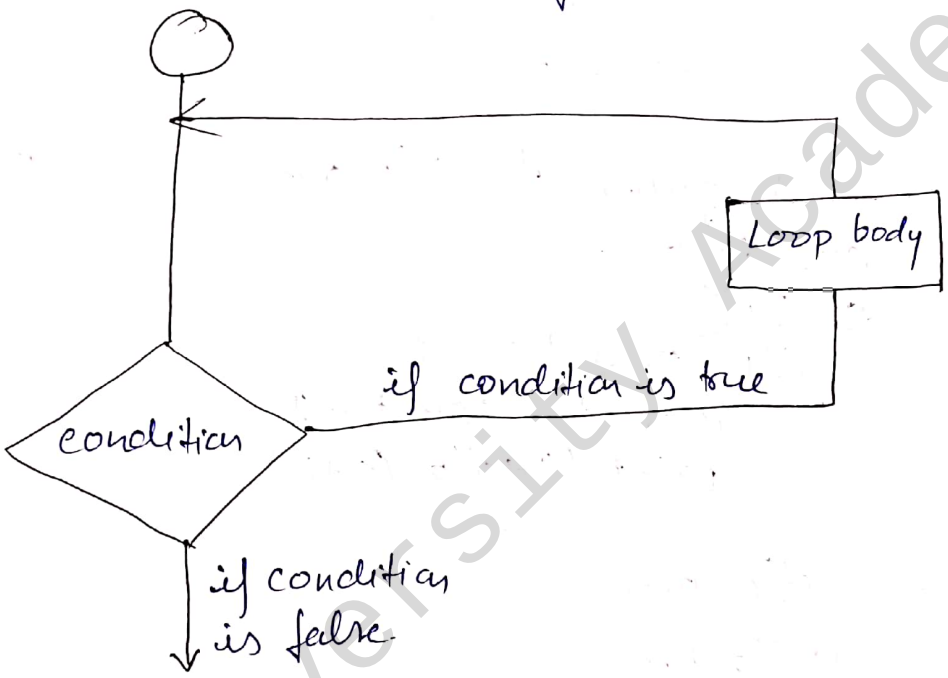
```
# python program to illustrate short hand if
i = 10
if i < 15:
    print("i is less than 15")
```

O/P: i is less than 15.

Loops:-

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of any specific code may need to be repeated several no. of times.

For this purpose loops are being provided which are capable of repeating some specific code several no. of times.



Advantages of loops:

Following are the advantages of loops.

- It provides code-re-usability.
- Using loops, we do not need to write the same code again and again.
- Using loops, we can traverse over the elements of data structure (array or linked lists)

Types of loops:

- while loop
- for loop
- nested loops

while loop:

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

syntax:

```
while expression:
    statements
```

ex:

```
#python Program to illustrate while loop.
count = 0
while count < 3:
    count = count + 1
    print ("Hello Arti")
```

O/P:

```
Hello Arti
Hello Arti
Hello Arti
```

ex: WAP in Python to add natural number upto n.

Soln:

```
#To take user input
n = int(input ("Enter n : "))
#initialize sum and counter
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i + 1    #update counter.
```

```
#print the sum
print ("the sum is ", sum)
```

O/P: Enter n
The sum is 55

While with else:

While-else is similar to if-else.

Syntax:

```
while condition:
    #execute these statements.
else:
    # execute these statements.
```

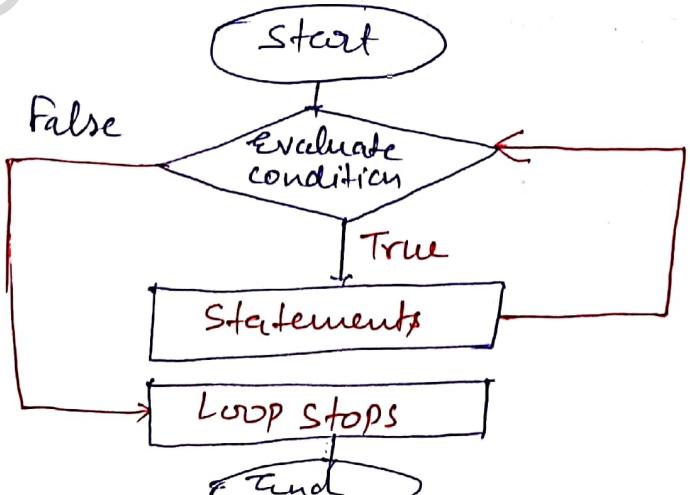
eg:

```
# Python program to illustrate else with wile.
count = 0
while & count < 3:
    count = count + 1
    print ("Hello world")
else:
    print ("In else Block")
```

O/P:

```
Hello world
Hello world
Hello world
In else block
```

Flow-chart for while loop:



For loop in Python:

The for loop in Python is used to iterate over a sequence. (list, tuple, string) or other iterable object.

Syntax:

```
for val in sequence:
    loop body.
```

Here val is the variable that takes the value of the item inside the sequence. on each iteration.

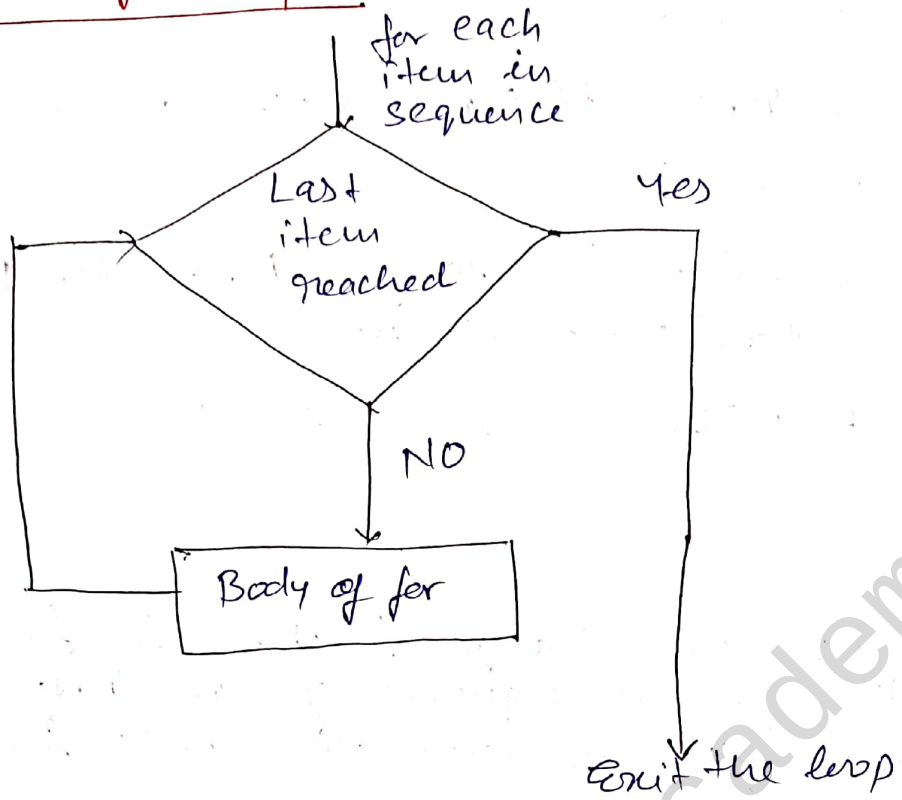
Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

ex:

```
# program to find the sum of all numbers stored in a list.

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum + val
print("The sum is", sum)
```

O/P: The sum is 48

Flowchart - "for loop" :



ex:- # program to print the table of given number.

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n= 5
for i in list:
    c= n * i
    print(c)
```

o/p:-

```
5
10
15
20
25
30
35
40
45
50
```

eg: #Python Program to find the sum of $N^{2.14}$ natural numbers

```
number = int( input (" Please Enter any no.")
total= 0
for value in range (1 , number+1):
    total = total+ value
print(" The sum is " , total)
```

The range() function:

We can use the range function in for loops to iterate through a sequence of numbers. It can be combined ~~through~~ with the len() function to iterate through a sequence of numbers using indexing. The above example explains the working of range() function with for loop.

Ex: Python program to iterate through index

```
list = [ "cat", "bat", "rat"]
for index in range ( len( list)):
    print list[index]
```

O/P:
```
cat
bat
rat
```

Ex: #Python Program to combine for with else.

```
list= [ "cat", "bat", "rat"]
for index in range ( len ( list)):
    print list[index]
else:
    print " Inside Else block"
```

O/P:
cat
bat
rat
Inside Else block.

Nested for loop:

A nested loop is loop inside loop.
The inner loop will be executed one time for each iteration of the "outer loop":

ex1:

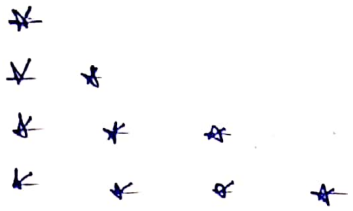
```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

O/P:
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry

ex2:

```
rows = int(input("Enter no. of rows:"))
for i in range(0, rows+1):
    for j in range(i):
        print("*", end=" ")
    print()
```

O/P: Enter the rows : 4



Loop Control Statements :-

1- The break statement
2- The continue statement.

1- The Break Statement :-

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

Syntax: break

Ex:
```
#use of Python break
for val in "string":
    if val == "i":
        break
    print(val)
print("The end")
```

O/P:
```
s
t
r
The end
```

For loop	for var in sequence:
	  if condition:
	  break
		While loop	while test expression:
			  if condition:
			  break

Python Continue Statement:

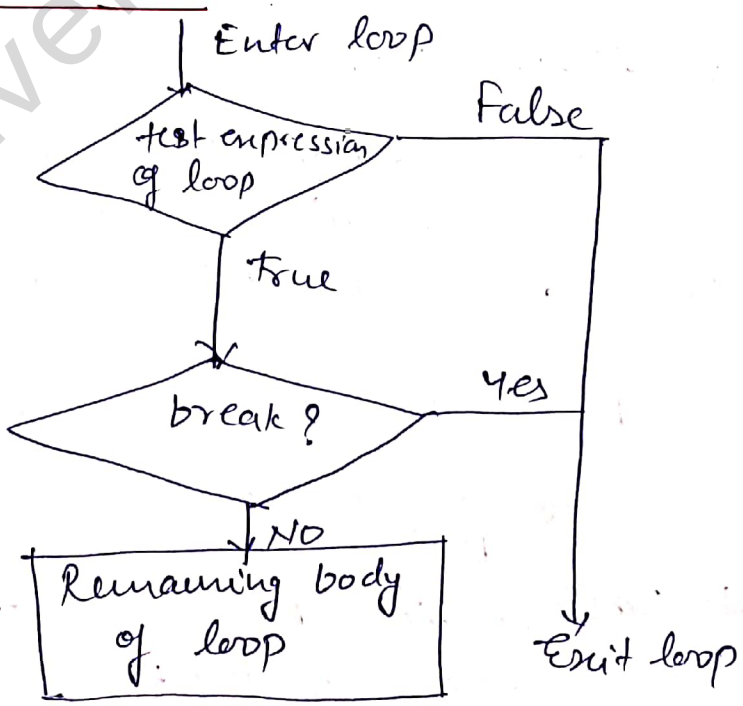
The continue statement is used to skip the rest of the code inside a loop for the current iteration only.

Syntax: continue

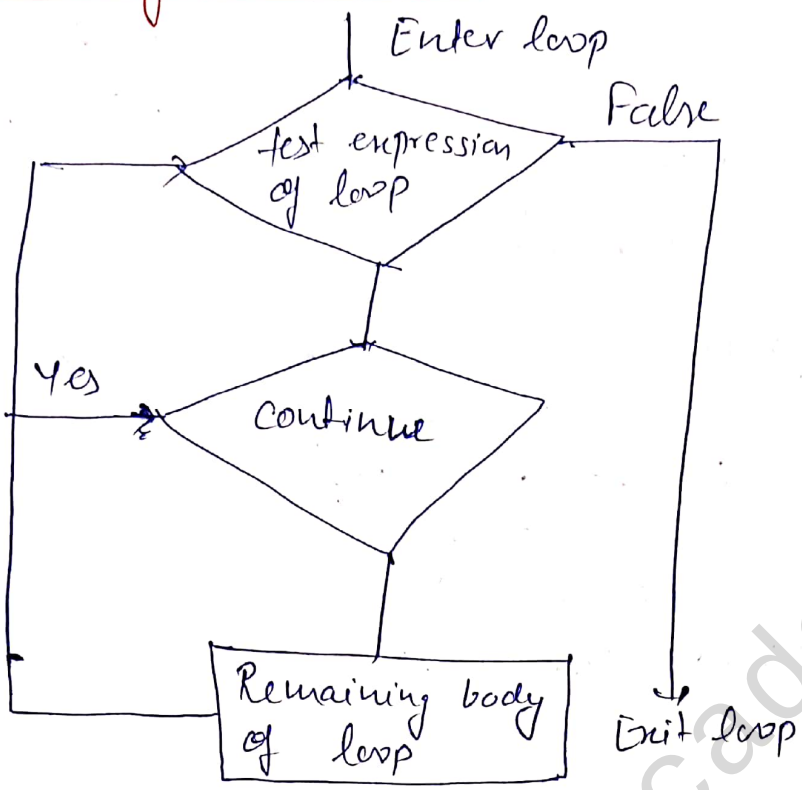
ex:
```
# program to show continue statement.
for val in "string":
    if val == "i":
        continue
    print(val)
print("The end")
```

O/P:
```
s
t
r
n
g
The end.
```

Flowchart of Break:-



Flowchart of Continue:



Python Pass statement:

The "pass" statement is a null statement

The pass statement is generally used as a Placeholder i.e. when the user does not know what code to write. So user simply places pass at that ~~time~~ line.

So user can simply place "pass" where empty code is not allowed like in loops, function definitions, class definitions, or in if statement.

ex:- # pass statement in empty functions.

```
def Function:
    pass
```

ex:- # pass statement in empty class

```
class Myclass:
    pass
```

UNIT-3
- Functions
- Strings
- Python Data structure
- Higher Order Functions.
- Sieve of Eratosthenes
- File I/O, Exceptions & Assertions.

Functions:

A function can be defined as the organized block of reusable code, which can be called whenever required.

A function can be called multiple times to provide reusability and modularity to the Python program.

The function helps the programmer to break the program into smaller part.

Types of functions:

There are mainly two types of functions.

1) User-defined functions
2) Built-in functions.

Built-in Functions:

Built-in functions are those functions that are pre-defined in Python.

ex: abs() returns the absolute value of a number

all()
any()
bin()
dict()
dir()

} All these functions are predefined and can be used whenever required.

Creating a function:

Python provides the def keyword to define the function.

Syntax:

```
def my-function( parameters):
    function_block
    return expression
```

Explanation:

- The "def" keyword alongwith function name is used to define the function.
- The identifier rule must follow the function name.
- A function accepts the parameter (arguments), and they can be optional.
- The function block is started with the colon(:) and block statements must be at the same indentation.
- The "return" statement is used to return the value. A function can have only one return.

ex:

```
# A simple Python Function
def fun():
    print("welcome to world")
```

Calling a function:

After creating a function we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

ex:- # A simple Python Function.

```
def fun():
    print("welcome to world")

# calling a function
fun()
```

O/P:- welcome to world.

The return statement :-

The return statement is used at the end of the function and returns the result of the function.

It terminates the function execution and transfers the result where the function is called. The return function (statement) cannot be used outside of the function.

Syntax:-

```
return [expression_list]
```

example 1:-

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a + b
    return c
# calling sum() function in print statement.
print("The sum is", sum())
```

Output:-

```
The sum is : 30
```

Explanation:

In the above example, we have defined the function name sum, and it has statement c = a+b, which computes the given values, and the result is returned by the return statement to the caller function.

Note:- If we create the same example without return statement it will return "None" object to the caller function.

Arguments in function:

The arguments are type of information which can be passed into the function. The arguments are specified in the parenthesis. We can pass any no. of arguments, but they must be separated by comma.

ex: # Python function to calculate the sum of two #variable

```
# defining function
def sum(a, b):
    return a+b
# taking values from the user
a = int(input("enter a:"))
b = int(input("enter b:"))
# printing the sum of a and b
print("Sum = ", sum(a,b))
```

O/P:-

```
Enter a: 10
Enter b: 20
Sum = 30
```

Call by reference in Python:

In Python, call by reference means passing the actual value as an argument in the function. All the functions are called by reference, i.e. all the changes made to the reference inside the function revert back to the original value referred by the reference.

ex: Passing Immutable Object (list)

```
# Python code to demonstrate call by reference
def add_more(list):
    list.append(50)
    print("Inside Function", list)

# Drivers Code
mylist = [10, 20, 30, 40]

add_more(mylist)
print("Outside Function", mylist)
```

Output:

```
Inside Function [10, 20, 30, 40, 50]
Outside Function [10, 20, 30, 40, 50]
```

Types of Arguments:

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

1- Required Arguments:

Required arguments are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition.

If either of the arguments is not provided in the call, or the position of the argument is changed, the interpreter will show error.

Ex:-1

```
def func (name):
    message = "Hi," + name
    return message
name = input ("Enter name:")
print(func(name))
```

O/P:-

```
Enter the name: John
Hi John
```

Ex-2

```
def calculate (a, b):
    return a+b
calculate (10) # this will cause an error, as we are missing a required argument b.
```

O/P:

```
TypeError: calculate() missing 1 required positional argument: 'b'
```

2- Default Arguments:

A default argument is a parameter that assumes a default value if a value is not provided in the function call of that argument.

ex:- # Python Program to demonstrate default argument.

```
def myFun(x, y=50):
    print("x:", x)
    print("y:", y)

# calling
myFun()
```

o/p:-

```
x: 10
y: 50
```

Variable length arguments (*args) :-

In Python, we can pass a variable number of arguments to a function using special symbols.

There are two special symbols.

- *args (Non-keyword arguments)
- **kwargs (keyword Arguments)

*args :-

At the function definition, we define the variable length argument using the *args (star) as

*<variable-name>

ex:-

```
def MyFunc(*names):
    print("type of passed argument is", type(names))
    print("Printing the passed arguments")
    for name in names:
        print(name)
MyFunc("John", "David", "Smith", "nick")
```

op:-

```
type of passed argument is <class 'tuple'>
printing the passed arguments...
John
David
Smith
nick
```

Keyword Argument (** kwargs)

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

```
# the function simple_intrest(p,t,r) is called with the
# keyword argument, the order of arguments doesnot
# matter.

def simple_interest(p, t, r):
    return (p*t*r)/100
print("simple_interest:", simple_interest(t=10,r=10, p=19000))
```

O/P: simple_interest: 19000.0

If we provide different name of arguments at the time of function call, an error will occur-

```
def simple_interest(p, t, r):
    return (p*t*r)/100
print("Simple_interest:", simple_interest(time=10, rate=10,
                                          principle=19000))
```

O/P: TypeError: simple_interest() got an unexpected keyword argument 'time'

Python Scope :-

A variable is only available from inside the region is created. This is called scope.

Local Scope :-

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

ex :-
```
def myfunc():
    x = 300
    print(x)
myfunc
```

O/P : 300

Global Scope :-

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

ex :-
```
x = 300
def myfunc():
    print(x)
myfunc()
print(x)
```

O/P : 300
300

Naming Variable :-

If you operate with the same variable name inside and outside of a function, Python will

Treat them as two separate variable, one available in the global space (scope) and one available in the local scope.

ex:

```
x = 300
def my-func():
    x = 200
    print(x)
my-func()
print(x)
```

O/P:-

```
200
300
```

String in Python:

In Python strings are array of bytes representing Unicode characters. Python doesnot have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

Creating a String:

Strings in python are surrounded by either single quotation marks, or double quetation marks. 'hello' is the same as "hello".

Ex:
```
# defining Strings in Python
my_string = 'Hello'
print (my_string)
my_string = "Hello"
print (my_string)
my_string = '''Hello'''
# tripple quote string can extend multiple lines.
my_string = """Hello, welcome to the
world of Python"""
print(my_string)
```

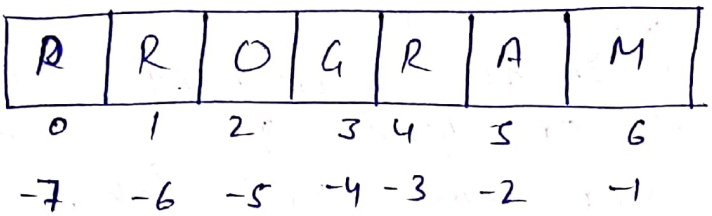
O/P:
```
Hello
Hello
Hello
Hello, welcome to the world of Python.
```

Accessing Characters in Python:

In Python, individual characters of string can be accessed by using the method of indexing.

Indexing allows negative address references to access characters from the back of string,

eg: -1 refers to the last character, -2 refers the second last character and so on.



Ex:-

```
#Python Program to Access characters of string.

String = "Program"
Print ("intial String:")
print (string)

# printing first character.
print("first character of string:")
print(string[0])

# printing last character.
print("Last character of string")
print(string[-1])
```

O/P:-

```
Initial string:
program

First character of string :
P

Last character of string
m
```

Python String Operations:

1- Concatenation of two strings
2- Iteration through a string.
3- String length

1) Concatenation of two strings:

Joining two or more strings into a single one is called concatenation.

The '+' operator does this in Python.

The '*' operater can be used to repeat the string for a given number of times.

ex:-
```
#Python string Operations
str1 = "Hello"
str2 = "world"

# concatenation using '+'
print("concatenation =", str1+str2)

# repetition using *
print("repetition=", str1*3)
```

O/P:
```
concatenation = Helloworld
repetition = HelloHelloHello
```

2) Iteration through a string:

We can iterate through a string using a for loop.

```
count = 0
for letter in "Hello world":
    count = count +1
print(count)
```

O/P: 10

ex-2:

```
count = 0
for letter in "Hello world":
    if(letter == "l"):
        count = count + 1
print(count)
```

o/p :- 3

3- String length:

To get the length of a string, use the len() function.

```
a = "Hello world"
print(len(a))
```

o/p : 11

Deletion / Updation of string:

In Python, updation or deletion of characters from a string is not possible. This will cause an error because item assignment or item deletion from a string is not supported.

Only deletion of entire string is possible with the use of built-in "del" keyword.

Note: Because strings are immutable, hence elements of a string cannot be changed once it has been assigned. Only new string can be reassigned to the same name.

Updating entire string:

```
string1 = "Hello, I am a doll"
print(string1)
# updating string.
string1 = "Now, I am a ball"
print(string1)
```

O/P:- Hello, I am a doll
Now, I am a ball

Deleting entire string:

```
# program to delete entire string
str1 = "Hello, I am a clock"
print (str1)

# deleting entire string
del str1
print(str1)
```

O/P:- NameError: name 'str1' is not defined.

String slicing:

To access a range of characters in the string, the method of slicing is used. Slicing in a string is done by using a slicing operator (:)

Ex:-

```
# creating a string
str1 = 'Hellotoall'
print("initial string")
print(str1)

# printing 3rd to 8th character
print(str1[3:8])

# slicing from the start
print(str1[:5]) # Get the characters from the start to position 5

# slice to the end
print(str1[2:] # Get the character from position 2 to end.
```

O/P:-

initial string
Hellotoall
lotoall # 3-8
Hellot # :5
lotoall # 2:

ex:1 # Python Program to check whether a string 3.1C
is symmetrical or Palindrome.

```
def palindrome(a):
    mid = (len(a)-1)//2
    start = 0
    last = len(a)-1
    flag = 0
    while start <= mid:
        if a[start] == a[last]:
            start = start+1
            last = last-1
        else:
            flag = 1
            break
    if flag == 0:
        print("The entered string is palindrome")
    else:
        print("The entered string is not
                palindrome")
```

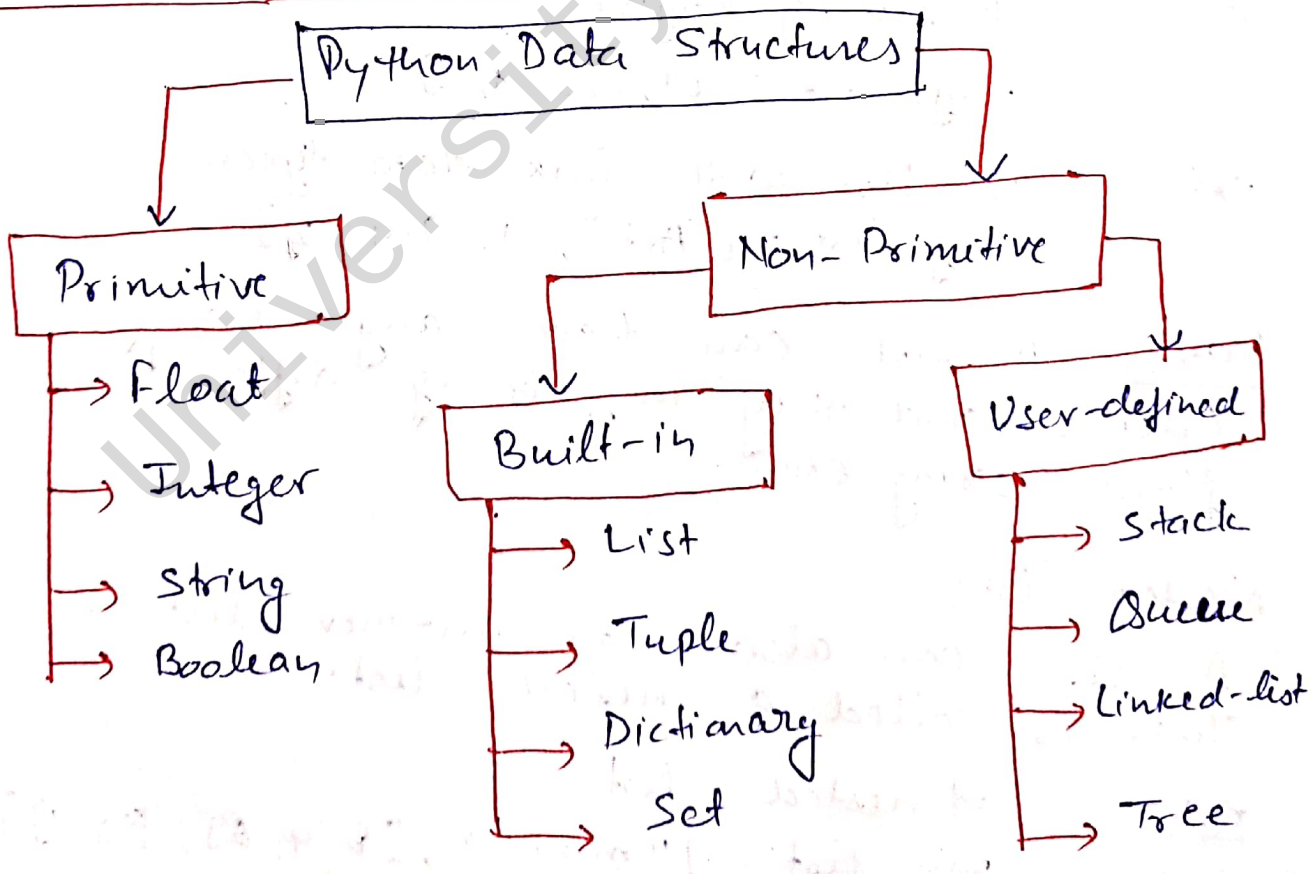
Function to check whether symmetrical or not.

```
def symmetry(a):
    n = len(a)
    flag = 0
    if n%2:
        mid = n//2+1
    else:
        mid = n//2
    start1 = 0
    start2 = mid
    while start1 < mid and start2 < n:
        if a[start1] == a[start2]:
            start1 = start1+1
            start2 = start2+1
```

```
else flag:
    flag = 1
    break
if flag == 0:
    print("The entered string is symmetrical")
else:
    print("The entered string is not symmetrical")

string = "ammaama"
palindrome(string)
symmetry(string)
```

O/P:
The entered string is palindrome
The entered string is symmetrical.

Python Data Structure:



Lists in Python:

A list can be defined as a collection of values or items of different types. Python lists are mutable types it means we can modify its elements after it created.

The items in the list are separated with the comma (,) and enclosed with the square brackets [].

ex: my_list = [1, "Hello", 3.4]

Creating a list:-

A list is created by placing elements inside square brackets [], separated by commas.

ex: my_list = [] #empty list.

ex: # list of integers
my_list = [1, 2, 3, 4, 5]

ex: # list with mix data types.
my_list = ["Arti", 1, 9.4, 2]

Note: A list can have any no. of arguments (items) and they may be of different types (int, float, string etc.

Nested list:-

A list can also have another list as an item. This is called a nested list.

ex: # nested list
my_list = ["mouse", [8, 4, 6], ['a']]
print (type (my_list))

o/p: <class 'list'>

Accessing items of list:

We can use the index operator [ ] to access an item in a list. In python indices starts at 0.

The index must be an integer. We can't use float or other types, this will result in TypeError.

Nested lists are ~~used~~ accessed using nested indexing.

ex:- my_list = ["p", "r", "o", "b", "e"]

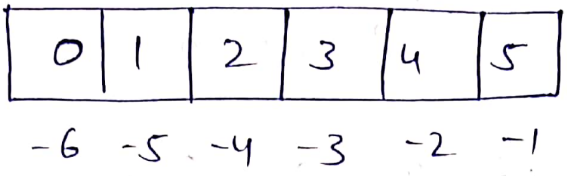
```
# first item
print(my_list[0])      # p

# third item
print(my_list[2])      # o

# Fifth item
print(my_list[4])      # e


# nested list
n_list = ["Happy", [2, 0, 1, 5]]
```

(Annotations: "Happy" and [2, 0, 1, 5] marked 0, 1 — indexing; elements 2, 0, 1, 5 marked 0 1 2 3 — indexing)

```
# nested indexing
print(n_list[0][1])      # a
print(n_list[1][3])      # 5
```

Output:-
p
o
e
a
5

Negative indexing in Python lists:



ex:

```
list = [0, 1, 2, 3, 4, 5]
print (list[-1])
print (list[-3])
print (list[-3:])
print (list[:-1])
print (list[3:-1])
```

This is also called list slicing. as already discussed in string chapter.

Output:

```
5
3
[3, 4, 5]
[0, 1, 2, 3, 4]
[3, 4]
```

Updating List values:

List values can be updated using slice and assignment operator.

Python also provide append() and insert() methods, which can be used to add values to the list.

ex:

```
list = [1, 2, 3, 4, 5, 6]
print (list)
list[2] = 10
print (list)
list[1:3] = [89, 78]
print (list)
list[-1] = 25
print (list)
```

Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]
```

Deleting List Elements:

1) del [a:b] → This method deletes all the elements in range starting from index 'a' till 'b'. mentioned in the arguments.

2) pop() :- This method deletes the element at the position mentioned in its arguments.

ex:
```
list = [2, 1, 3, 5, 4, 3, 8]
del list[2:5]
print ("List elements after deleting are:", end=" ")
for i in range (0, len(list)):
    print (list[i], end = " ")
    print ("\r")

# using pop()
list.pop(2)
print ("List elements after popping are:", end=" ")
for i in range (0, len(list)):
    print (list[i], end = " ")
```

Output: List elements after deleting are: 2 1 3 8
List elements after popping are: 2 1 8 8

Deleting entire list:

```
list = [1, 2, 3, 4, 5]
del list[:]
print (list)
```

O/P: [ ]

Add/change list:-

We can add one item to a list using append() method or add several items using the extend() method.

Ex:-
```
odd = [1, 3, 5]
odd.append(7)
print(7)
print(odd)
odd.extend[9, 11, 13]
print(odd)
```

O/P:-
```
[1, 3, 5, 7]
[1, 3, 5, 7, 9, 11, 13]
```

We can also use '+' to combine two list.

```
odd = [1, 3, 5]
print(odd + [9, 7, 5])
print("re" * 3)
```

O/P:-
```
[1, 3, 5, 9, 7, 5]
['re', 're', 're']
```

insert()

```
odd = [1, 9]
odd.insert(1,3)
print(odd)
odd[2:2] = [5, 7]
print(odd)
```

Output:-
```
[1, 3, 9]
[1, 3, 5, 7, 9]
```

Tuples in Python:

Python tuple is used to store the sequence of immutable Python objects. It is quite similar to list in accessing and but the only difference is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

Creating a Tuple:

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas.

The parentheses are optional.

A tuple can have any number of arguments (elements) and they may be of different types (integer, float, list, string etc.)

ex:

```
# Different types of tuple.

#empty Tuple
my_tuple = ()
print(my_tuple)

#Tuple having integers
my_tuple = (1, 2, 3)
print (my_tuple)

#tuple with mix datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

#nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print (my_tuple)
```

o/p's

```
()
(1,2,3)
```

(1, 'Hello', 3-4)

('mouse', [8, 4, 6], (1, 2, 3))

Creating a tuple with one element:

It is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is a tuple.

ex:-

```
my_tuple = ("hello")
print (type (my_tuple))   # <class 'str'>

# Creating a tuple having one element
my_tuple = ("hello",)
print(type (my_tuple))   # <class 'tuple'>

# Parentheses is optional.
my_tuple = "hell",
print (type(my_tuple))   # class 'tuple'>
```

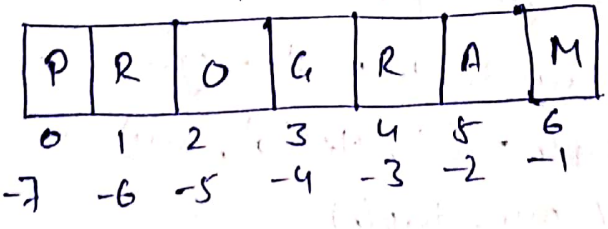
O/P:-

```
< class 'str'>
< class 'tuple'>
< class 'tuple'>
```

Accessing Tuple Elements:-

1- By using index number:-

We can use the index operator to access an item in a tuple, where the index starts from 0.



ex:-

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print (my_tuple[0])    # p
print (my_tuple[5])    # t
```

```
#nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

#nested index
Print(n_tuple[0][3])      #'s'
Print(n_tuple[1][1])      #'4
```

O/P:
```
p
t
s
4
```

Negative Indexing:

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[-1])
Print(my_tuple[-6])
```

O/P:
```
t
p
```

Slicing:

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[1:4])
print(my_tuple[:-2])
print(my_tuple[-2:])
print(my_tuple[:])
```

O/P:

	output
('r', 'o', 'g')	('e', 'r', 'm')
('p', 'r')	('p', 'e', 'r', 'm')
('i', 'z')	('r', 'm', 'i', 't')
('p',	('p', 'e', 'r', 'm', 'i', 't')

Changing a Tuple:

Unlike lists, tuples are immutable.
This means that elements of a tuple cannot be

changed once they have been assigned. But if the tuple is itself a mutable datatype like a list, its nested items can be changed.

ex:
```
my_tuple = (4, 2, 3, [6, 5])
# try changing tuple element will lead to error.
#my_tuple[1] = 9  # O/P: TypeError: tuple object does not
                  # support item assignment.
# item of mutable elements can be changed.
my_tuple[3][0] = 9      # output: (4, 2, 3, [9, 5])
print(my_tuple)

# Tuple can be reassigned.
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a' 'm')
print(my_tuple)
```

O/P:
```
(4, 2, 3, [9, 5])
('p', 'r', 'o', 'g', 'r', 'a' 'm')
```

Concatenation of Tuples:

- We can use '+' operator to combine two tuples. This is called concatenation.
- We can also repeat the elements in a tuple for a given number of times. using the '*' operator.

ex:
```
#concatenation.
print((1,2,3) + (4, 5, 6))

#repeat
print(("repeat",) * 3)
```

O/P:
```
(1, 2, 3, 4, 5, 6)
('repeat', 'repeat', 'repeat')
```

Deleting a tuple:-

We can not delete a single element in tuple but we can delete the entire tuple entirely, by using the del keyword.

Ex:-
```
my-tuple = ('p','r','o','g','r','a','m')
del my-tuple
print (my-tuple)
```

O/P:- NameError : name 'my-tuple' is not defined.

Python Dictionary:-

A dictionary is an unordered and mutable Python container that stores mapping of unique key to values.

Creating a Dictionary:-

Dictionaries are written with curly brackets ({}), including key-value pairs separated by commas(,). A colon(:) separates each key from its value.

Ex:-
```
# dictionary containing population of 5 cities.
population = {'Berlin':3748148, 'Hamburg':1822445,
              'Munich': 1471508, 'Cologne':1085664,
              'Frankfurt': 753056}

# empty dictionary
my_dict = { }
```

Accessing Elements from Dictionary:-

- To access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method.
- If we use square bracket([]) KeyError is raised if a key not found in the dictionary. get() method returns 'None' if key is not found.

ex: # [] vs get

```
my_dict = {'name': 'Jack', 'age': 26}  #create #dictionary
print(my_dict['name'])   #output: Jack
print(my_dict['age'])    #output: 26
```

```
# get
my_dict = {'name': 'Jack', 'age': 26}
print(my_dict.get('name'))   #O/P: John
print(my_dict.get('age'))    #o/p: 26
```

```
# Accessing key which is not available
print(my_dict.get('address'))  #O/P: None
print(my_dict['address'])      #o/p: KeyError: 'address'
```

Changing and adding dictionary elements:

If the key is already present, then the existing value gets updated. In case the key is not present, a new (key:value) pair is added to the dictionary.

ex:

```
my_dict = {'name': 'Jack', 'age': 26}
#print(my_dict)
#update values
my_dict['age'] = 27
print(my_dict)

#add item
my_dict['address'] = 'Downtown'
print(my_dict)
```

O/P:

```
{'name': 'Jack', 'age': 26}
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

Removing elements from dictionary :

- We can use a particular item in a dictionary by using the pop() method. This method remove an item with the provided key and returns the value.
- The popitem() method can be used to remove and return an arbitrary (key, value) item pairs from the dictionary.
- All the items can be removed at once using the clear() method.
- We can also use the del keyword to remove the entire dictionary.

Ex:

```
# creating a dictionary.
squares = {1:1, 2:4, 3:9, 4:16, 5:25}

# removing a particular item returns its value.
print(squares.pop(4))   #output : 16
print(squares)  #O/P: {1:1, 2:4, 3:9, 5:25}

# removing arbitrary item
print(squares.popitem())   #O/P: (5, 25)
print(squares)  #output: {1:1, 2:4, 3:9}

# remove all items
squares.clear()   #O/P: { }
print(squares)
# Delete the dictionary itself
del squares
print(squares)  #O/P : Throws Error.
```

O/P:

```
16
{1:1, 2:4, 3:9, 5:25}
(5, 25)
{1:1, 2:4, 3:9}
{}
```

Python Dictionary Comprehension:

Dictionary comprehension is an elegant way to create a new dictionary from an iterable in python.

Dictionary comprehension consists of an expression pair (key: value) followed by a "for" statement inside curly braces. {}.

ex:-
```
#Dictionary Comprehension
squares = {x: x*x for x in range(6)}
print(squares)
```

o/p:- {0:0, 1:1, 2:4, 3:9, 4:16, 5:25}

The above code is equivalent to

```
squares = {}
for x in range(6):
    squares[x] = x*x
print(squares)
```

o/p:- {0:0, 1:1, 2:4, 3:9, 4:16, 5:25}

Dictionary Comprehension with if:

```
# for "if" condition.
odd_squares = {x: x*x for x in range(11) if x%2==1}
print(odd_squares)
```

o/p:- {1:1, 3:9, 5:25, 7:49, 9:81}

Python Sets:

A set is an unordered, collection of datatype that is iterable, mutable and has no duplicate value.

Creating a Python Set:

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

A set can have different types of items (int, float, tuple string etc.) But a set cannot have mutable elements like lists, sets or dictionaries as its element.

ex:-
```
my_set = {1,2,3}
print(my_set)

#set of mixed datatypes
my_set = {1.0, "Hello", (1,2,3)}
print(my_set)
```

o/p:-
```
{1,2,3}
{1.0, 'Hello', (1,2,3)}
```

Creating an empty set:

Empty curly braces { } will make an empty dictionary, but it will not create empty set. To create a set without elements, we use the set() function without any argument.

```
#initialize a with {}
a = { }
print(type(a))
#initialize a with set()
a = set()
# check data type of a
print(type(a))
```

o/p

o/p: <class 'dict'>
<class 'set'>

Modifying a set in Python:

We can add a single element using the add() method and multiple element using the update() method. The update() method can take tuples, lists, strings or other sets as its arguments. In all cases duplicates are avoided.

ex:

```
my_set = {1, 3}
print(my_set)

#add an element
my_set.add(2)
print(my_set)        #o/p: {1, 2, 3}

#add multiple elements
my_set.update([2, 3, 4])
print(my_set)        #o/p: {1, 2, 3, 4}

# add list and set
my_set.update([4, 5, 6], {1, 6, 8})
print(my_set)        #o/p: {1, 2, 3, 4, 5, 6, 7, 8}
```

o/p:

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

Removing Element From set:

A particular element can be removed from a set using the discard() and remove() method. The difference b/w the two is that discard() leaves a set unchanged on the other hand remove will raise an error if the element not present in the set

ex:-
```
# initializing a set.
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
my_set.discard(4)
print(my_set)   #o/p: {1, 3, 5, 6}

# remove an element
my_set.remove(6)
print(my_set)   #o/p: {1, 3, 5}

# discard an element not present in set.
my_set.discard(2)
print(my_set)   #o/p: {1, 3, 5}

# removing an element
# not present in set
my_set.remove(2)
print(my_set)   #o/p: KeyError:2
```

i.e. list will remain unchanged

set items will remains same.

o/p:-
```
{1, 3, 4, 5, 6}
{1, 3, 5, 6}
{1, 3, 5}
{1, 3, 5}
Traceback (most recent call last):
  File "<string>", line 26, in <module>
KeyError:2
```

pop() vs clear():-

Since set is an unordered datatype, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all the items from a set using the clear() method.

```
# initialize set
my_set = set ("Helloworld")
print (my_set)

# popping an element
print(my_set.pop())      #o/p : random element

# pop another element
my_set.pop()
print (my_set)

# clear my set
my_set.clear()
print(my_set)      #O/P:- set()
```

O/P:- {'H', 'l', 'r', 'W', 'o', 'd', 'e'}

H

{'r', 'W', 'o', 'd', 'e'}

set()

Python Set operations:

1) Set - Union

```
#initialize A and B
A = {1, 2, 3, 4, 5, 6}
B = {4, 5, 6, 7, 8}

# use '|' operator (Set union)
point (A | B)   #O/P: {1, 2, 3, 4, 5, 6, 7, 8}
```

2) Set intersection:

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
# use & operator
print (A & B)
```

O/P:- {4, 5}

Higher Order Functions:

A function is called higher order function if it contains other functions as a parameter or returns a function as an output.

Properties of higher order functions:

- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash-tables, lists etc.

Function as objects:

In python, a function can be assigned to a variable. The assignment does not call the function, instead a reference to that function is created.

Ex:-
```
#Python program to illustrate functions treated as
# objects

def shout(text):
    ~~return text.upper~~
    return text.upper()
print(shout("Hello"))

#Assigning function to variable
yell = shout
print(yell('Hello'))
```

O/P:
```
HELLO
HELLO
```

Note: In the above example a function object referenced by shout and creates a second name pointing to it yell.

Passing Function as an argument

Functions are like objects in Python, therefore they can be passed as an argument to other functions.

ex:-
```
# Python g program to explain that functions can
# be passed as argument to another functions.
def shout(text):
    return text.upper()
def whisper(text):
    return text.lower()
def greet(func):
    # storing function in variable.
    greeting = func("Hi, I am created by a function passed as an argument")
    print(greeting)
greet(shout)
greet(whisper)
```

output: "HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.

hi, i am created by a function passed as an argument.

Returning Functions

We can also return a function from another function.

ex:-
```
def create_adder(x):
    def adder(y):
        return x+y
    return adder
```

```
add_15 = create_adder(15)
print(add_15(10))
```

O/P:- 0.25

Decorators

- Decorators are the most common use of higher order functions. It allows programmers to modify the behaviour of function or class.
- Decorators allows us to wrap another function in order to extend the behaviour of wrapped function, without permanently modifying it.
- In decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Syntax:

```
@gfg_decorater
def hello_decorater():
    .
    .
    .
```

The above code is equivalent to-

```
def hello_decorater():
    .
    .
    .
hello_decorater = gfg_decorater(hello_decorater)
```

ex:

```
#defining a decorater
def hello_decorater(func):
#inner1 is wrapper function in which argument is called.
# inner function can access the outer local functions (func)
    def inner1():
```

```
        print("Hello, this is before function execution")
        # calling the actual function now
        # inside the wrapper function.
        func()
        print("This is after function execution")
    return inner1

# defining a function, to be called inside wrapper
def function_to_be_used():
    print("This is inside the function")

# passing 'function_to_be_used inside the decorater
# to control its behaviour
function_to_be_used = hell_decorator(function_to_be_used)
# calling the function
function_to_be_used()
```

3.38

Outputs:

```
Hello, this is before function execution
This is inside the function.
This is after function execution.
```

Example-2

```
import time
import math

def calculate_time(func):
    def inner1(*args, **kwargs):
        begin = time.time()
        func(*args, **kwargs)
        end = time.time()
        print("total time taken in: ", func.__name__,
              end - begin)
    return inner1

@ calculate_time
def factorial(num):
    time.sleep(2)
    print(math.factorial(num))
factorial(10)
```

O/P: 3628800

Total time taken in: factorial 2.00

Lambda Expression in Python:

Python lambda functions are anonymous functions means that the function is without a name.

Syntax:

lambda arguments : expression

Note: Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned.

We can use lambda functions wherever function objects are required.

ex:

```
double = lambda x : x*2
print(double(5))
```

O/P: 10

where lambda x : x*2 = lambda function

x = argument

x*2 = expression that gets evaluated and returned.

The above function has no name. It returns a function object which is assigned to the identifier double.

double = lambda x : x*2

is nearly same as

```
def double(x):
    return x*2
```

Use of lambda functions in Python:-

Lambda functions are used alongwith built-in functions like filter(), map() etc.

The filter() function in Python takes in a function and a list as arguments. The map() also takes in a function and a list.

ex-1 Example of filter() function.

```
#Program to find (filter out) only the even items
#from a list.
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x : (x%2==0), my_list))
print(new_list)
```

O/P: [4, 6, 8, 12]

ex-2

```
#program to double each item in a list
#using map()
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x : x*2, my_list))
print(new_list)
```

Output:

[2, 10, 8, 12, 16, 22, 6, 24]

Sieve of Eratosthenes:

Sieve of eratosthenes is a simple and ancient algom used to find the prime number upto any given limit. It is one of the most efficient ways to find the small prime numbers.

Working:

For a given upper limit n the algom works by iteratively marking the multiples of primes as composite, starting from 2. Once all multiples of 2 have been marked composite, the multiples of next prime, ie. 3 are marked composite. This process continues until $P \leq \sqrt{n}$ where P is prime number.

Algorithm: In following algorithm number 0 represents a composite number.

1- To find out all primes under n, generate a list of all integers from 2 to n. (1 is not prime)

2- Start with a smallest prime number, ie p = 2

3- Mark all the ~~prime number~~ multiples of p which are less than n as composite. To do this, mark the value of the number (multiples of p) in the generated list as 0. Do not mark p itself as composite.

4- Assign the value of p to the next prime. The next prime is the next non-zero number in the list which is greater than p.

5- Repeat the process until $p \leq \sqrt{n}$.

Example: Generate all prime numbers less than 11.

1- Create a list of integer from 2 to 10.
list = [2, 3, 4, 5, 6, 7, 8, 9, 10]

2- Start with p = 2

3- Since $2^2 \leq 10$, continue.

4- Mark all multiples of 2 as composite by setting their value as 0 in the list.
list = [2, 3, 0, 5, 0, 7, 0, 9, 0]

5- Assign the value of p to the next prime ie 3

6- Since $3^2 \leq 10$, continue.

7- Mark all multiples of 3 as composite by setting their value as 0 in the list. list = [2, 3, 0, 5, 0, 7, 0, 0, 0]

8- Assign the value of p to 5.

9- Since $5^2 \nleq 10$, stop.

So, the final list is —

[2, 3, 0, 5, 0, 7, 0, 0, 0]

Here all non-zero numbers are prime numbers.
Hence the prime numbers less than 11 are—

2, 3, 5, 7

File I/O in Python:

- A file is a named location on disk to store related information. We can access the stored info after the program termination.
- File handling plays an important role when the data needs to be stored permanently in a file.
- In python files are treated in two modes as text or binary.

File operation can be done in following order.

1- Open a file
2- Read or write - Performing Operation
3- Close the file.

Opening a file:

Python provides a built-in function open() for opening a file. It accepts two arguments, file name and access mode.

Syntax: file object = open(<file name>, <access mode>)

example:

```
#opening file in read mode.
fileptr = open("file.txt","r")
if fileptr:
    print("file is opened successfully")
```

Closing the File:

Once all the operations are done, we must close the file using the close() method.

Syntax:

```
fileobject.close()
```

Example:

```
fileptr = open("file.txt", "r")  #opening
if fileptr:
    print("file is opened successfully")
fileptr.close()  #closing the file.
```

Working of read() mode:

If we need to extract a string that contains all characters in the file then we can use "file.read()".

Example:

```
file = open("file.txt", "r")
print(file.read())
```

Working of write() mode:

write() mode is used for the manipulation in a file.

Ex:

```
file = open("file.txt", "w")
file.write("This is a write command")
file.write("It allows us to write in a file")
file.close()
```

Note: The close() command terminates all the resources in use and frees the system of this particular program. (resources are freed).

Working of append():

```
file = open("file.txt", "a")
file.write("This will add this line")
file.close()
```

File Access Modes

1- r: open an existing file for a read operation

2- w: open an existing file for a write operation. If the file contains some data then it will be overridden.

a: open an existing file for append operation. It won't override existing data.

r+: To read and write data into the file. The previous data in the file will not be deleted.

w+: To write and read data. It will override existing data.

a+: To append and read data from the file. It won't override existing data.

UNIT - 4 → Modules
→ Abstract Data Types (ADT)
→ Classes.

Modules: A python module is a file containing Python definitions and statements.

Python modules are simply files with the ".py" extension containing Python code that can be imported inside another Python Program.

A module can define functions, classes, and variables.

Example: Creating a simple module.

Save this code in a file named module.py

```
def greeting(name):
    print("Hello, "+name)
```

Using a Module:

Now we can use the module by using the "import" statement:

```
import module
module.greeting("Ram")
```

module name — function name

O/P:- Hello, Ram

Variables in Module:

The modules also contains variables of all types (arrays, dictionaries, objects etc.)

Example:

Save this code in the file "module.py"

```
person1 = {
    "name": "John",
    "age": 38,
    "country": "India"
}
```

} dictionary

Accessing: Import the module named module, & access the person1 dictionary:

```
import module
a = module.person1["age"]
print(a)
```

O/P:- 38

Renaming a Module:

You can create an alias by using as keyword

Example:

Create an alias for module called mx:

```
import module as mx
a = mx.person1["age"]
print(a)
```

O/P:- 38

Built in Modules:

There are so many built-in modules in Python which can be imported whenever required.

Example:- Import and use platform module:

```
import platform
x = platform.system()
print(x)
```

O/P :- windows

Import From Module:

You can select only parts of module to import from a module by using the from keyword.

Module created:- module.py (file name)

```
def greeting(name):
    print("Hello, " + name)

person1 = {
    "name" = "John",
    "age" = "38"
    "country" = "India"
}
```

Now importing only person1 dictionary from the module! (module name)

```
from module import person1
print(person1["age"])
```

O/P: 38

Python Math:-

Python has a set of built-in math functions;

Some examples of built-in math functions.

1) min() and max()

```
x = min(5, 10, 25)
y = max(5, 10, 25)
print(x)
print(y)
```

o/p: 5
25

2) abs(): It returns the absolute (positive) value of the specified number.

```
x = abs(-7.25)
print(x)
```

o/p: 7.25

3) pow(x,y): returns the value of x to the power of y (x^y).

```
x = pow(4,3)
print(x)
```

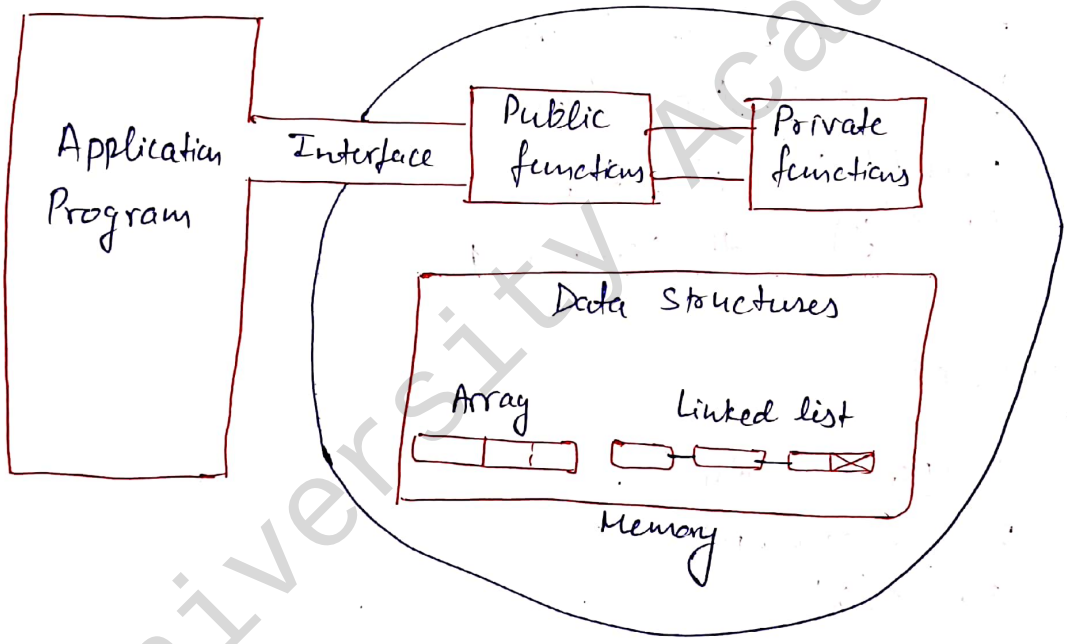
o/p: 64

The Math Module:-

Python has also a built-in module called math, which is used for extending the list of mathematical functions.

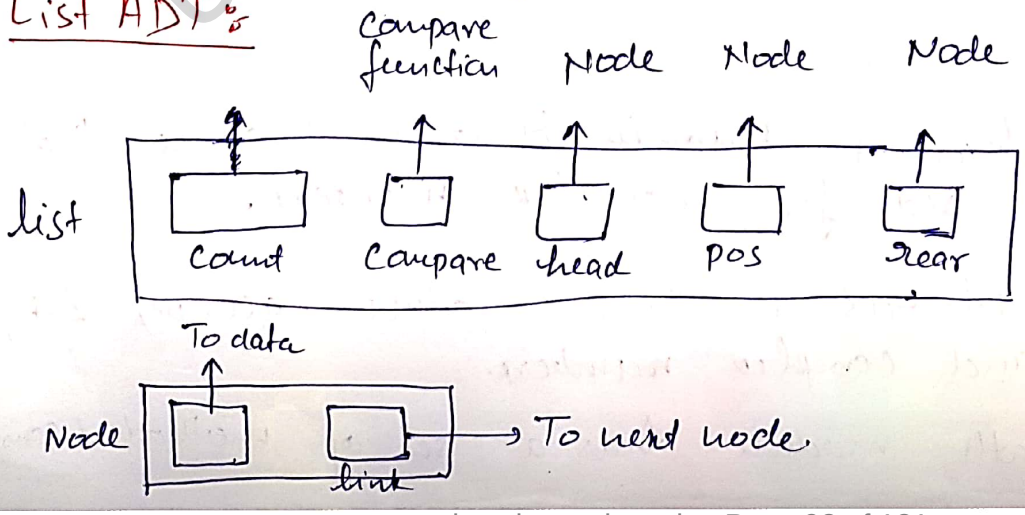
Abstract Data Types (ADT) in Python:

- Abstract Data Type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations.
- The keyword "Abstract" is used as we can use these data types for performing different operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.



Some examples of ADTs are - Queue, Stack, list etc.

List ADT:



To use it, we need to import the math module.

Syntax: import math

Examples

```
import math
x = math.sqrt(64)
print(x)
```
(method name)

O/P: = 8.0

other methods can also be used like this)

ceil() and floor() methods :-

```
import math
x = math.ceil(1.4)
y = math.floor(1.4)
print(x)  # returns 2
print(y)  # returns 1
```

math.pi

```
import math
x = math.pi
print(x)
```

O/P: 3.141592653589793

Python cmath Module :-

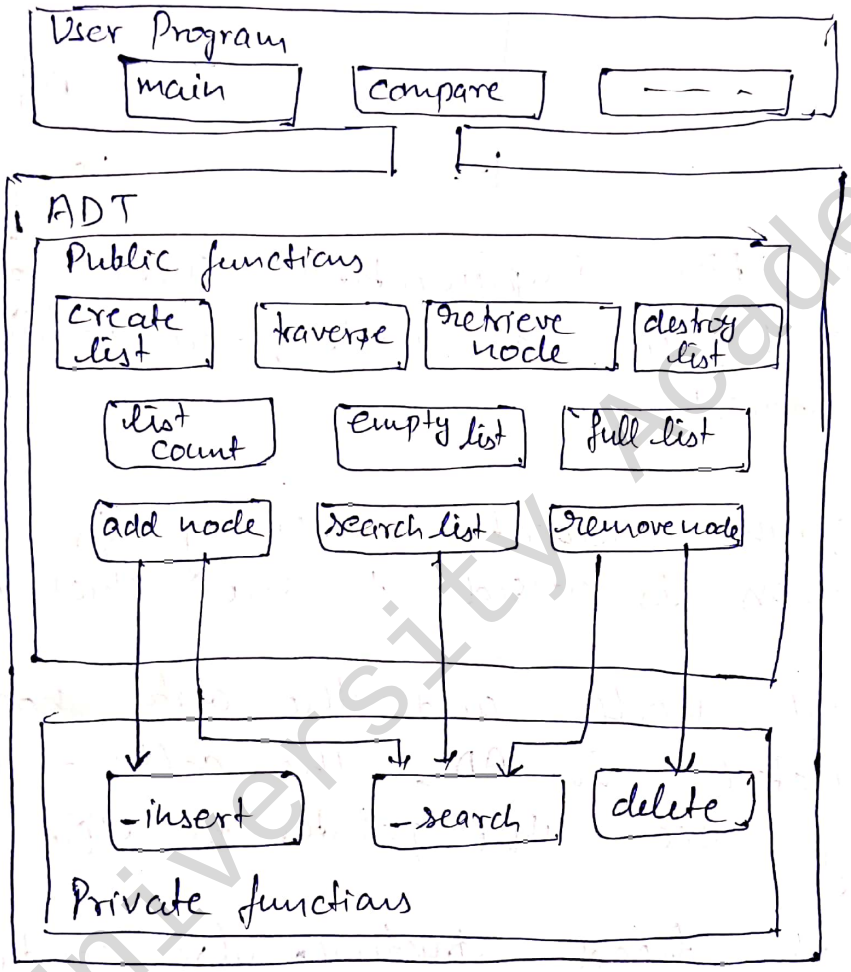
Python has a ~~build-in~~ built-in module that can be used for complex numbers.

These methods in this module accepts int, float, and complex numbers.

The cmath module has a set of methods & const.

The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.

List ADT functions:-

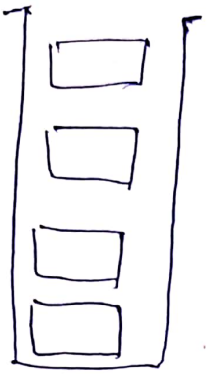


A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

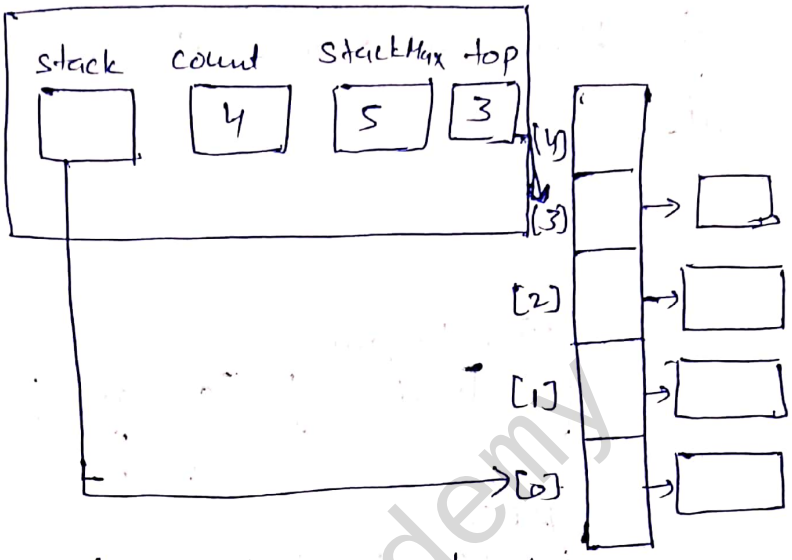
- get()
- insert()
- remove()
- removeAt()
- replace()
- size()
- isEmpty()
- isFull()

Stack ADT :-

a) Conceptual

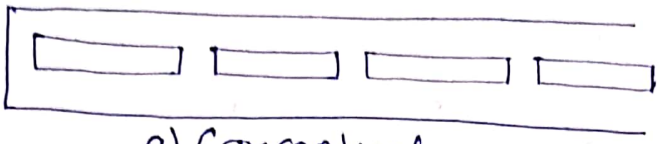


b) Physical Structure

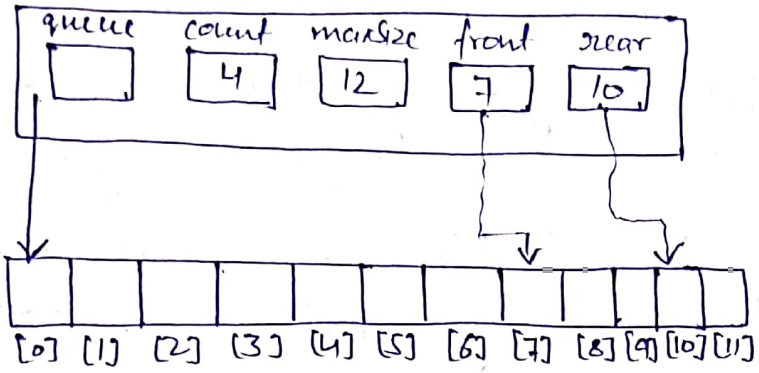


- In stack ADT implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the data and address is passed to the stack ADT.
- The head node and the data nodes are encapsulated in ADT. The calling function can only see the pointer to the stack.
- The stack head structures also contains a pointer to top and count of number of entries currently in stack
- The operations that are performed in stack implementation are as follows.
  - push()
  - pop()
  - peek()
  - size()
  - isEmpty()
  - isFull()

Queue ADT :



a) Conceptual



b) Physical structures.

- The queue abstract data type (ADT) follows the basic design of stack abstract data type.
- Each node contains a void pointer to data and link pointer to the next element in the queue.
- Queue contains elements of the same type arranged in sequential order.

Following operations can be performed.

- enqueue()
- dequeue()
- peek()
- size()
- isEmpty()
- isFull()

Python object oriented programming:

- Python is a multi-paradigm programming language. It supports different programming approaches.
- An python object-orrented programming (OOPs) is a programming paradigm that uses objects and classes in programming.
- It aims to implement real-world entities like inheritance, polymorphism, encapsulation etc. in the programming.
- The main concept of OOPs is to bind the data and functions that work on that together as a single unit so that no other part of the code can access this data.

Concepts in OOPs

1. Class
2. objects
3. polymorphism
4. encapsulation
5. Inheritance.

1- Class in Python:

A class can be defined as a collection of objects. It contains the blueprints or the prototype from which the objects are being created. It also contains some attributes and methods.

Basic Points of Python Class:

Classes are created by the keyword class.
Attributes are the variables that belong to a class.
Attributes are always public and can be accessed using the dot(.) operator.

Class definition Syntax:

```
Class ClassName
    # statement 1
    |
    |
    #Statement N
```

ex: empty class in Python.

```
class Dog:
    pass
```

2. Objects:

The object is an entity that has a state and behaviour associated with it. It may be any real world object like mouse, keyword, chair, table pen etc.

An object consists of:

- State: It is represented by the attributes of an object. It also reflects the properties of an object.
- Behaviour:-

It is represented by the methods of an object. It also reflects the response of an object to the other objects.

- Identity:

It gives a unique name to an object and enables one object to interact with other objects.

ex: For class Dog state, behavior and identity can be given as follows.

- The identity can be considered as the name of the dog.
- State or attributes can be considered as the breed, age or color of the dog.
- The behaviour can be considered as to whether the dog is eating or sleeping.

Example:

```
obj = Dog()
```

This will create an object named obj of the class Dog defined above.

The self :

- Class method must have an extra first parameter in the method definition. We do not give a value for this parameter at the time of calling but python will provide it.
- If we have a method that takes no arguments, then we still have to have one arguments.

The __init__ method :

The __init__ method is similar to constructors in C++ and java. It is run as soon as an object of a class is instantiated.

This method is useful to do any initialization you want to do with your object.

Ex: Creating a class and object with class and instance attributes.

```
class Dog:
    #class attribute
    attr1 = "mammal"
    #instance attribute
    def __init__(self, name):
        self.name = name

#Driver code
# object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")
#Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))
#Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Output:

```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy.
```

Ex: Creating Class and objects with methods.

```
class Dog:
    #class attribute
    attr1 = "mammal"
    #instance attribute
    def __init__(self, name):
```

```
        self.name = name
    def speak(self):
        print("My name is {}".format(self.name))

#Driver code
# object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")
# Accessing a class method
Rodger.speak
Tommy.speak
```

O/P:

```
My name is Rodger
My name is Tommy
```

3- Polymorphism:-

Polymorphism simply means having many forms. It refers to the use of a single type entity (methods, operator or object) to represent different types in different scenarios.

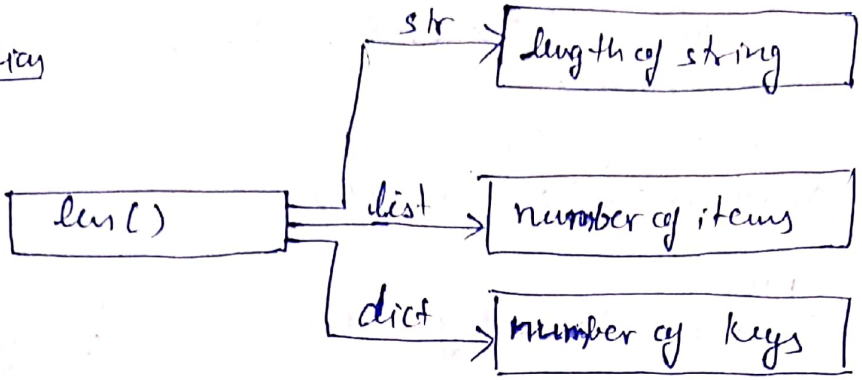
ex:- '+' operator can be used for addition of two numbers, as well as for performing concatenation (str1+str2) of two strings.

Polymorphic len() function():-

ex:
```
print(len("program"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "India"}))
```

Outputs:
```
7
3
2
```

Explanation



Here, we can see that many data types such as string, list, tuple, set and dictionary can work with the same len() function. However they returns different information in different cases.

Polymorphism in Class Method :-

We can use the concept of polymorphism while creating class methods as python allows different classes to have methods with the same name.

Ex:

```
Class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f"I am a cat. My name is {self.name}. I am
                {self.age} years old")
    def make_sound(self):
        print("Meow")
Class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f"I am a dog. My name is {self.name}. I am
                {self.age} years old")
    def make_sound(self):
        print("Bark")
cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)
```

```
for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

Output:

```
Meoco
I am a Cat. My name is kitty. I am 2.5 years old.
Meow
Bark
I am dog. My name is Fluffy. I am 4 years old.
Bark
```

Explanation: Here we have created two classes Cat and Dog. They share a similar structure and have the same method name info() and make_sound()

4- Inheritance:

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class and the class from which the properties are being derived is called the base class or parent class.

Benifits of Inheritance:

- It represents real-world relationships well.
- It provides reusability of code. we dont need to write the same code again and again. Also it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means if a class B inherits from another class A, then all the subclass of B would automatically inherit from class A.

ex:- Inheritance in Python:

```
# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person( object):
    # __init__ is known as the constructor
    def __init__ (self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display (self):
        Print(self.name)
        Print(self.idnumber)

    def details (self):
        Print("My name is {}".format(self.name))
        Print("Id number: {}".format(self.idnumber))

#child class
class Employee (Person):
    def __init__ (self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post
        # invoking the __init__ of the parent class
        person.__init__(self, name, idnumber)

    def details (self):
        Print ("My name is {}".format(self.name))
        print ("Idnumber: {}".format(self.idnumber))
        print ("Post: {}".format(self.post))
#creation of an object variable or an instance
a = Employee ('Rahul', 886012, 200000, "Intern")
# calling a function of the class Person using its instance
    a.display()
    a.details()
```

Output:

```
Rahul
BB6012
My name is Rahul
IdNumber = BB6012
Post : Intern
```

Explanation:

In the above example, we have created two classes. i.e. Person (parent class) and employee (child class).

The employee class inherits from the person class. We can use the methods of the person class through employee class as seen in the display func in the above code.

A child class can also modify the behaviour of the parent class as seen through the details method.

5- Encapsulation:

- Encapsulation is one of the fundamental concept in OOP. It describes the idea of wrapping data and the methods that work on the data within one unit.
- A class is an example of encapsulation as it encapsulate all the data that is member function, variables etc.

Ex: Encapsulation in Python:

```
#creating a base class
class Base():
    def __init__(self):
        self.a = "HumanValues"
        self._c = "HumanValues"
#creating a derived class
class Derived(Base):
    def __init__(self):
```

```
#calling constructor of base class.
Base.__init__(self)
print("Calling private members of base class")
print(self.__c)

# Driver code
obj1 = Base()
print(obj1.a)
```

O/P:- Human Values

UNIT-5
- Iterators and Recursion
- search
- Sorting and Merging.

Iterators and Recursion:

While writing a computer program, some instructions may be repetitive with a common pattern. Recursion or iteration helps us to write lines of codes for repetitive tasks.

ex: Suppose a python list with five elements. This operation needs five lines of codes.

```
flowers = ['lily', 'tulip', 'rose', 'lavender', 'lotus']
print (flower[0])
print (flower[1])
print (flower[2])
print (flower[3])
print (flower[4])
print (flower[5])
```

output:-
```
lily
tulip
rose
lavender
lotus
```

Now using iteration —

```
flowers = ['lily', 'tulip', 'rose', 'lavender', 'lotus']
for flower in flowers:
    print (flower)
```

o/p:-
```
lily
tulip
rose
lavender
lotus
```

Note:- Iteration can be performed through 'for' and while loops.

Recursion in Python:-

Recursion is a functional approach of breaking down a problem into a set of simple subproblems, with an identical pattern and solving them by calling one subproblem inside another in sequential order.

Recursion is executed by defining a function that can solve one subproblem at a time. Inside that function, it calls itself but solving another subproblem. Thus the call to itself continues until some limiting criteria is met.

Eg:-
```
#factorial using for loop
n=10
result=1
for i in range(1,n+1):
    result*=i
print(result)
```

O/P:- 3628800

```
#factorial using recursion.
def factorial(n):
#declare a base case (limiting criteria)
    if n==1
        return 1
    else:
        return n*factorial(n-1)
print(factorial(10))
```

O/P:- 3628800

Explanation of recursion version:-

Here problem is broken into subproblems. First of all it attempts to find factorial(10). Factorial 10 is broken down into 10 * Factorial(9). Further Factorial(9) is broken into 9 * Factorial(8) and so on.
When Factorial(1) is called, it stops the recursion.

Fibonacci series using recursion in Python:

A fibonacci sequence is the integer sequence of

0 1 1 2 3 5, 8 - - - -

or

0+1 → 1
1+1 → 2
1+2 → 3
2+3 → 5
3+5 → 8

#Python Program to display the fibonacci sequence.

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

nterms = 10

if nterms <=0:
    print("Please enter a positive integer")
else:
    print("Fibonacci sequence")
    for i in range(nterms):
        print(recur_fibo(i))
```

O/P:

```
Fibonacci Sequence
0
1
1
2
3
5
8
13
21
34
```

Tower of Hanoi

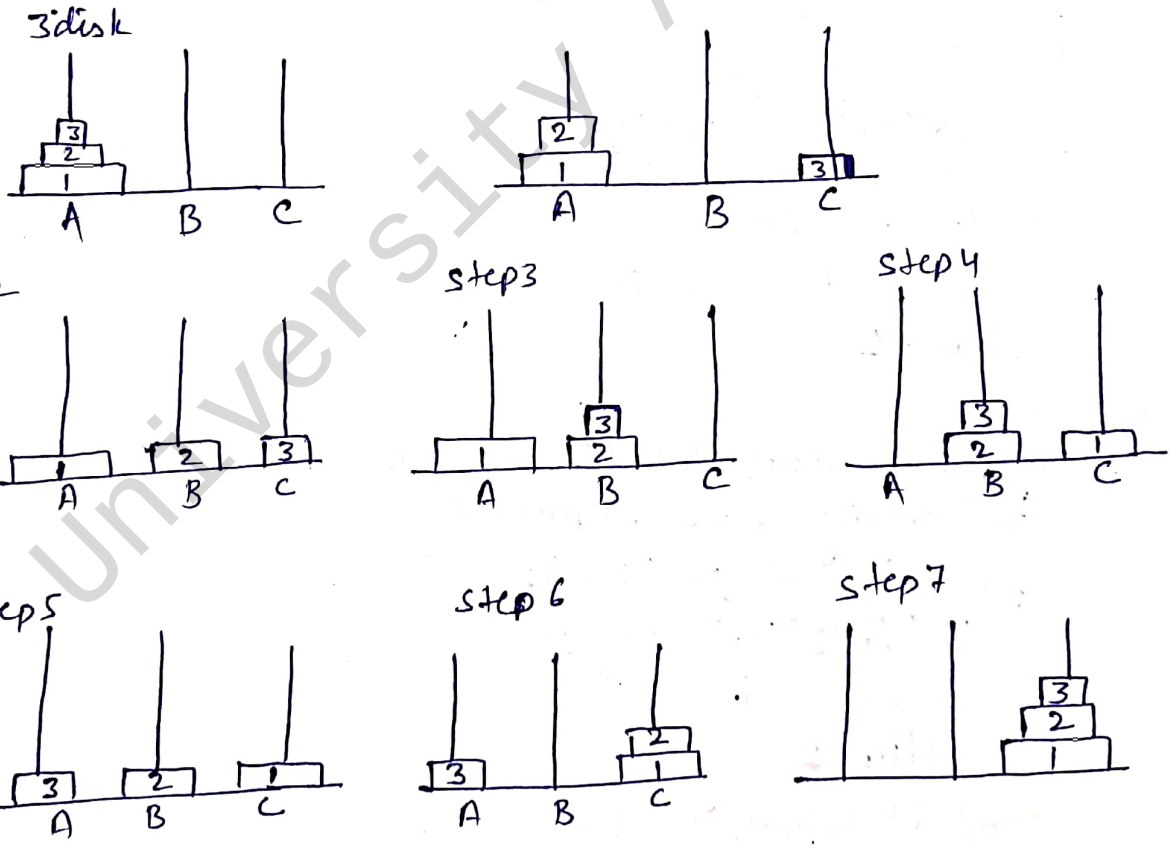
Tower of Hanoi is a mathematical puzzle where we have 3 rods and n disks. The objectives of the puzzle is to move the entire stack to another rod, obeying the following simple rules.

1- Only one disk can be moved at a time.

2- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

3. No disk may be placed on top of a smaller disk.

Ex:



Searching in Python:

Searching is a technique to find the particular element is present or not in the given list.

Type of Searching Algorithms:

- Linear search
- Binary search

Linear Search:

Linear search is a method of finding elements within a list. It is also called a sequential search, because it searches the desired element in a sequential manner.

It compares each and every element with the value that we are searching for. If both are matched, the element is found.

Linear Search Algorithm:

```
LinearSearch(list, key)
  for each item in the list
    if item == value
      return its index position.
  return -1
```

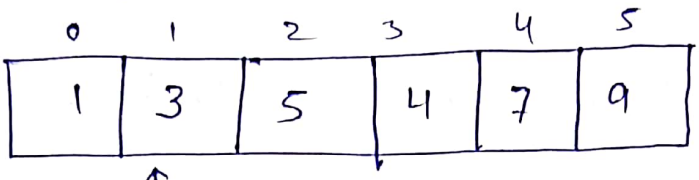
Program:

```
def linear_search(list1, n, key):
    #Searching list1 sequentially.
    for i in range(0, n):
        if (list1[i] == key):
            return i
    return -1
list = [1,3,5,4,7,9]
key = 7
n = len(list1)
res = linear_search(list1, n, key)
if (res == 1)
```

```
    print("Element not found")
else:
    print("Element found at index:", res)
```

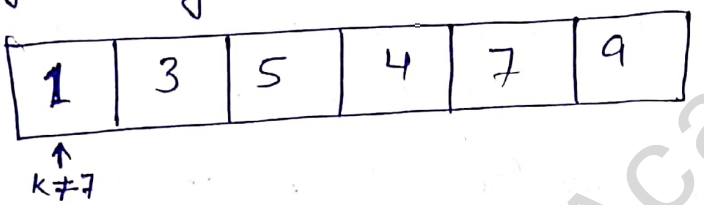
O/P:- Element found at index: 4

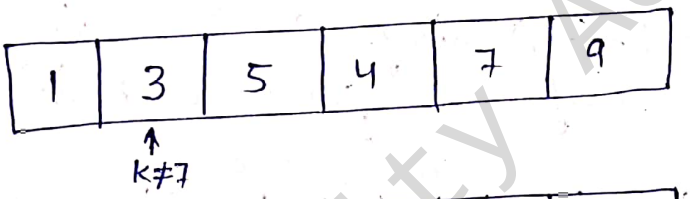
Explanation:-

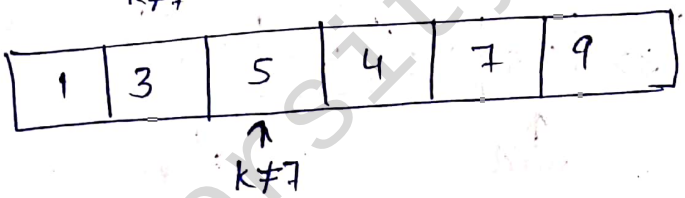


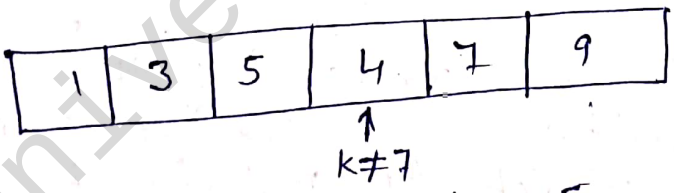
Step1:- Start the search from the ~~list~~ first element and check key = 7 with each ~~list~~ element of list x.

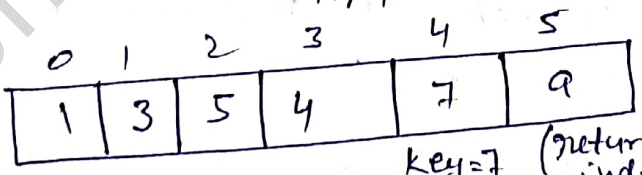
Step2: If element is found, return the index position of the key.











Step-3 If element is not found, return element is not present.

Linear Search Complexity:-

Best case - O(1)
Average case - O(n)
worst case - O(n)

Binary Search :-

Binary search is an efficient algorithm for finding an item from a sorted list of items.

Algorithm :-

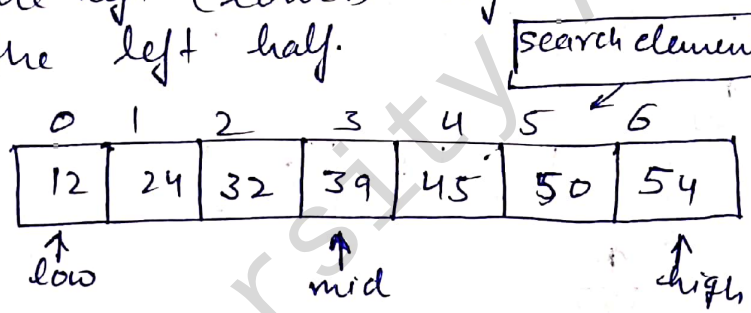
1- Compare x (element to be search) with the middle element.

2- If x matches with the middle element, we return the mid index.

3- Else if x is greater than the mid element, then x can lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.

4- Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.

ex :-



$mid = \frac{low + high}{2}$

$= \frac{0+6}{2} = 3$

Step 1 → we have a sorted list of element and we are looking for the index position of 45. So we are setting two pointers in our list. (low & high). Now we will calculate the mid element by using the formula $\left(mid = \frac{low + high}{2}\right)$.

Step-2- Now we compare 45 with the mid element i.e 39. $(39 \neq 45)$. So we need to do the further comparison.

Step-3- The number to be search is greater than the middle number we compare x with the middle element of the the elements on the right side of mid and set low to low = mid+1

Step-4 otherwise, compare x with the middle element of the elements on the left side of mid and set high to high = mid - 1.

Python Program using recursion.

```
def binary_search (arr, low, high, x):
    # check base case
    if high >= low:
        mid = (high+low)//2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, low, mid-1, x)
        else:
            return binary_search (arr, mid+1, high, x)
    else:
        return -1
arr = [2, 3, 4, 10, 40]
x = 10
result = binary_search (arr, 0, len(arr)-1, x)
if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
```

O/P:- Element is present at index 3.

Python Program using iteration:

```
def binary_search (arr, x):
    low = 0
    high = len(arr)-1
    mid = 0
    while low <= high:
        mid = (high+low) //2
        if arr[mid] < x:
            low = mid+1
```

```
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1

arr = [2, 3, 4, 10, 40]
x = 10
result = binary_search(arr, x)
if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
```

O/P: Element is present at index 3.

Complexities of Binary search:

1- Best Case:

Best case occurs when the element to be search is in the middle of the list.

In this case element is found in the first step itself and this involves 1 comparison.

So best case complexity = O(1).

2) Average Case: ~~O(log n)~~ O(log n)

3) worst case - ~~O(n)~~ O(log n)

Sorting :

Sorting refers to arranging data in a particular format. Sorting algorithms specify the way to arrange data in a particular order.

Applications of sorting :

1) Searching
2) Selecting
3) Duplicates finding
4) Distribution of elements.

Types of Sorting Algorithms :

1- Selection Sort
2- Merge Sort
3- Bubble Sort
4- Insertion Sort etc.

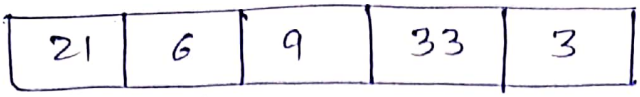
Selection Sort :

Selection sort is a sorting algo^m, that selects the smallest element from an unsorted ~~list~~ list in each iteration and places that element at the beginning of the unsorted list.

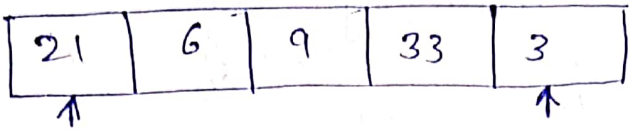
Selection Sort Algorithm :

1- Get the value of n which is the total size of the array.
2- Partition the list into sorted and unsorted sections. The sorted section is initially empty while the unsorted section contains the entire list.
3- Pick the minimum value from the unpartitioned section and place it into the sorted section.
4- Repeat the process (n-1) times untill all of the elements in the list have been sorted.

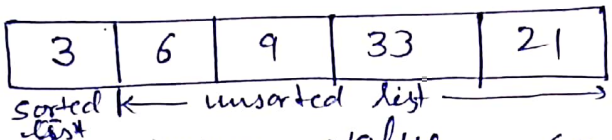
ex:-



Step 1:-

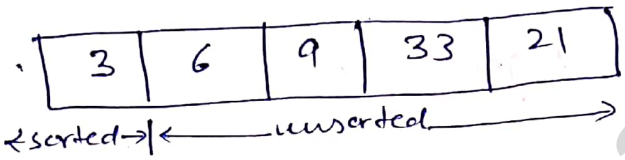


The first value 21 is compared with the rest of the values to check if it is the minimum value.



3 is the minimum value, so swap 21 with 3. Now the list is divided into two parts i.e. sorted and unsorted list.

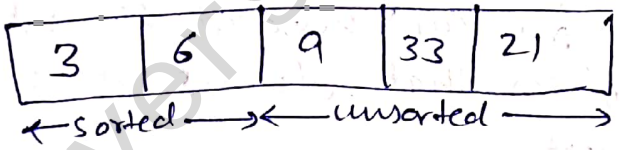
Step-2



Now value 6 is the minimum value, ~~and first~~ is the unsorted list. So compare 6 with the rest of the value in unsorted list to find out the lower value.

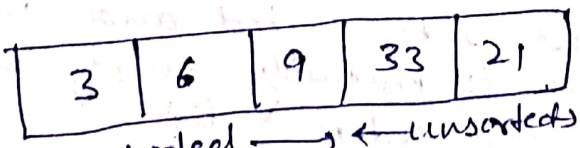
Since 6 itself is lower so it maintains its position.

Step-3



Now first element of unsorted list i.e. 9 will be compared to the rest of the unsorted list element. The value 9 is itself minimum so it maintains its position.

Step 4



Now value 33 will be compared with rest elements in unsorted.

21 is less than 33 so position of 21 and 33 will be swapped and produce the new list as —

Mergesort:

- Merge sort is based on divide and conquer approach. It divided the given list into two equal halves, calls itself and for the two halves and then merges the two sorted halves. We have to define merge() function to perform the merging.
- The sublists are divided again and again into halves until the list cannot be divided further
- Then we combine the pair of one element list into two element lists, sorting them in the process.
- The sorted two element ~~list~~ pairs is merged into the four element list and so on until ~~the~~ we get the sorted list.

Algo^m:

```
MERGE_SORT(arr, beg, end)
if beg < end
Set mid = (beg + end)/2
MERGE_SORT (arr, beg, mid)
MERGE_SORT (arr, mid+1, end)
MERGE(arr, beg, mid, end)
end of if
END MERGE_SORT
```

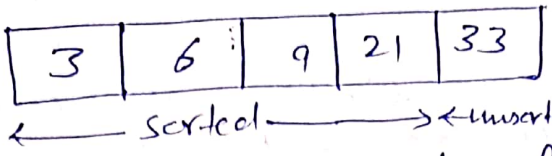
Program of Merge Sort in Python:

~~void merge (int a[], int beg, int mid, int end)~~

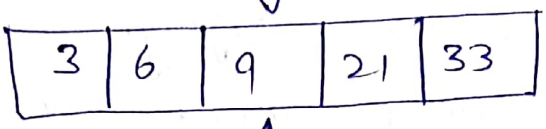
~~{~~

~~int i, j, K;~~

```
def mergeSort(array):
    if len(array) > 1:
        r = len(array) //2
        L = array[:r]
        M = array[r:]
```



Now we have only one value left in unsorted list. Therefore it is already sorted.



This is the final list which is sorted.

Python Program:

```
def selectionsort (itemList):
    n = len(itemList)
    for i in range (n-1):
        minValueIndex = i
        for j in range (i+1, n):
            if itemList[j] < itemList[minValueIndex]:
                minValueIndex = j
        if minValueIndex != i:
            temp = itemList[i]
            itemList[i] = itemList[minValueIndex]
            itemList[minValueIndex] = temp
    return itemList

el = [21, 6, 9, 33, 3]
print (selection sort (el))
```

O/P: 3, 6, 9, 21, 33

Time Complexities of Selection Sort:

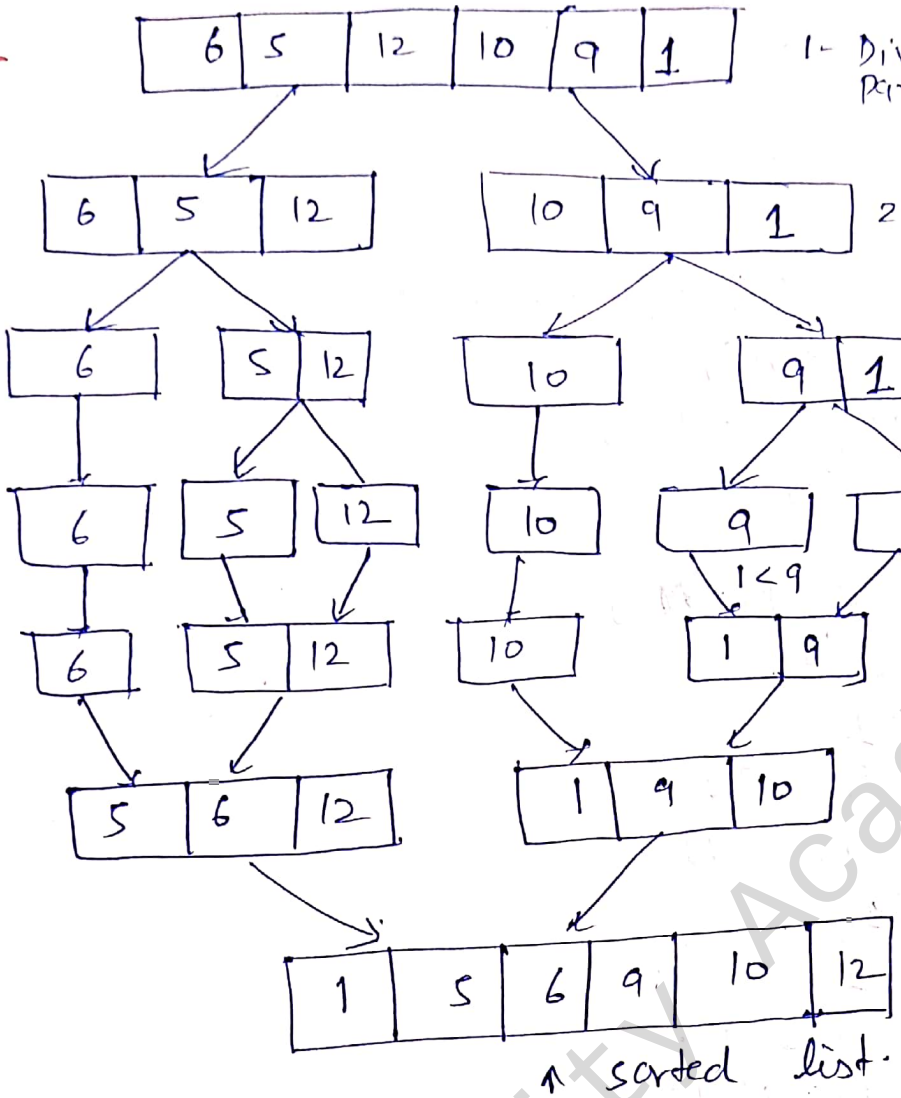
1- worst case - $O(n^2)$.

2- Best case - (n^2)

3- Average Case - (n^2)

Space Complexity: $\rightarrow O(1)$

Ex:-



1- Divide the list in two parts.

2- Further dividing the list

3) Now Merging the elements while comparing.)

4) Merging while comparing.

Merge Sort Complexities:

Best case	$O(n * \log n)$
worst case	$O(n * \log n)$
Average Case	$O(n * \log n)$
Space Complexity	$O(n)$
Stability	yes.

```
#Sort the two halves
mergeSort(L)
mergeSort(M)
i=j=k=0
while i<len(L) and j<len(M):
    if L[i]<M[j]:
        array[k]=L[i]
        i=i+1
    else:
        array[k]=M[j]
        j=j+1
    k=k+1
while i<len(L):
    array[k]=L[i]
    i=i+1
    k=k+1
while j<len(M):
    array[k]=M[j]
    j=j+1
    k=k+1
#Print the array
def printList(array):
    for i in range(len(array)):
        print(array[i],end=" ")
    print()
# Driver Program
if __name__=='__main__':
    array=[6,5,12,10,9,1]
    mergeSort(array)
    print("Sorted list is:")
    printList(array)
```