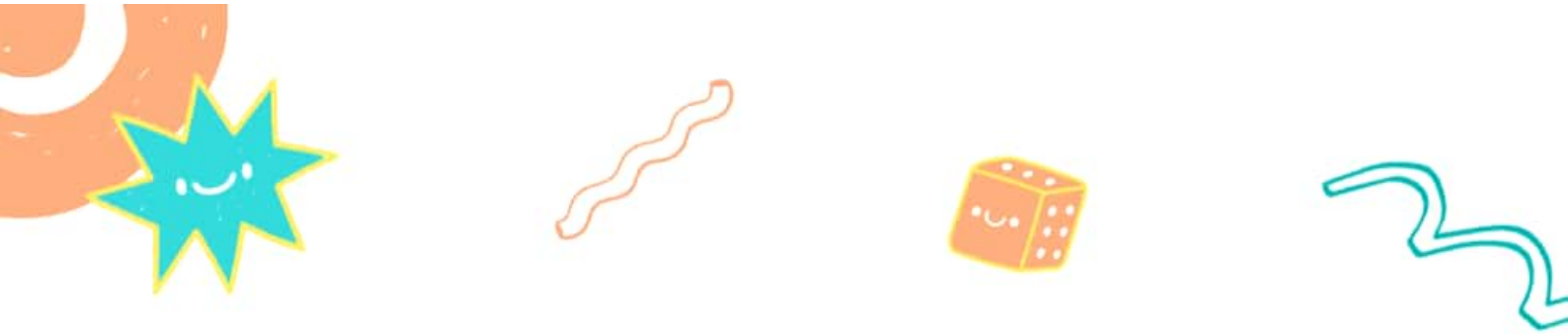


DATA STRUCTURES USING PYTHON

BY
MRCET(CSE)

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

II Year B.Tech. II Sem

L	T/P/D	C
3	- / -/ -	3

OPEN ELECTIVE I
(R18A0553) DATA STRUCTURES USING PYTHON

OBJECTIVES:

1) To read and write simple Python programs.
2) To develop Python programs with conditionals and loops.
3) To define Python functions and call them.
4) To use Python data structures — lists, tuples, dictionaries.
5) To do input/output with files in Python.

UNIT I

Introduction to Python, Installation and Working with Python, Understanding Python variables Python basic Operators, Understanding python blocks, Python Data Types: Declaring and using Numeric data types: int, float,complex,Using string data type and string operations.

UNIT II

Control Flow- if, if-elif-else,loops ,For loop using ranges, string ,Use of while loops in python,Loop manipulation using pass, continue, break and else,Programming using Python conditional and loops block,Python arrays.

UNIT III

Functions -Calling Functions, Passing Arguments, Keyword Arguments,Default Arguments, Variable-length arguments, Anonymous Functions, Fruitful Functions(Function Returning Values), Scope of the Variables in a Function - Global and Local Variables.Powerful Lamda function in python.

UNIT IV

Data Structures-List Operations, Slicing, Methods; Tuples, Sets, Dictionaries,Sequences. Comprehensions,Dictionary manipulation, list and dictionary in build functions

UNIT V

Sorting:BubbleSort,SelectionSort,InsertionSort,Mergesort,Quicksort.LinkedLists,Stacks,Queues

OUTCOMES:

Upon completion of the course, students will be able to
1) Read, write, execute by hand simple Python programs.
2) Structure simple Python programs for solving problems.
3) Decompose a Python program into functions.
4) Represent compound data using Python lists, tuples, dictionaries.
5) Read and write data from/to files in Python Programs

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY

INDEX

NOTE:-List only main topics

UNIT - I

INTRODUCTION TO PYTHON:

Python is a widely used general-purpose, high level programming language. It was initially designed by **Guido van Rossum in 1991** and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code. Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions- **Python 2 and Python 3**.

- On 16 October 2000, Python 2.0 was released with many new features.
- On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

Beginning with Python programming:

1) Finding an Interpreter:

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like **https://ide.geeksforgeeks.org/**, http://ideone.com/ or http://codepad.org/ that can be used to start Python without installing an interpreter.

***Windows*:** There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from **http://python.org/downloads/**

Differences between scripting language and programming language:

SCRIPTING LANGUAGE	PROGRAMMING LANGUAGE
A programming language that supports scripts: programs written for a special run-time environment that automate the execution of tasks	A formal language, which comprises a set of instructions used to produce various kinds of output
Execution speed is slow	Compiler-based languages are executed much faster while interpreter-based languages are executed slower
Can be divided into client-side scripting languages and server-side scripting languages	Can be divided into high-level, low-level languages or compiler-based or interpreter-based languages
Easier to learn	Not as easy to learn
Ex: JavaScript, Perl, PHP, Python and Ruby	Ex: C, C++, and Assembly
Mostly used for web development	Used to develop various applications such as desktop, web, mobile, etc.

2) Writing first program:

```
# Script Begins
        Statement1
        Statement2
        Statement3
# Script Ends
```

Why to use Python:

The following are the primary factors to use python in day-to-day life:

1. **Python is object-oriented**
   Structure supports such concepts as polymorphism, operation overloading and multiple inheritance.
2. **Indentation**
   Indentation is one of the greatest feature in python
3. **It's free (open source)**
   Downloading python and installing python is free and easy
4. **It's Powerful**
   - Dynamic typing
   - Built-in types and tools
   - Library utilities
   - Third party utilities (e.g. Numeric, NumPy, sciPy)
   - Automatic memory management
5. **It's Portable**
   - Python runs virtually every major platform used today
   - As long as you have a compaitable python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.
6. **It's easy to use and learn**
   - No intermediate compile
   - Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
   - This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages.
   - Structure and syntax are pretty intuitive and easy to grasp.
7. **Interpreted Language**
   Python is processed at runtime by python Interpreter
8. **Interactive Programming Language**
   Users can interact with the python interpreter directly for writing the programs
9. **Straight forward syntax**
   The formation of python syntax is simple and straight forward which also makes it popular.

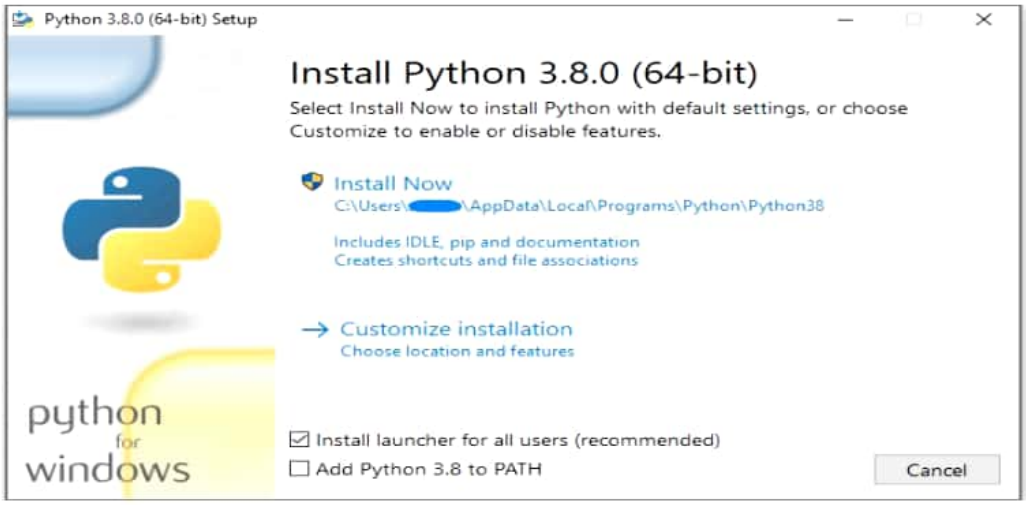
INSTALLATION AND WORKING WITH PYTHON:

Installation:

There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from http://python.org/downloads/

Steps to be followed and remembered:

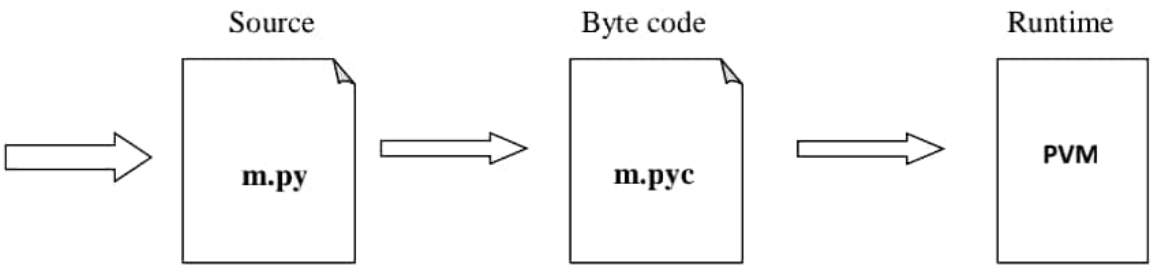
Step 1: Select Version of Python to Install.
Step 2: Download Python Executable Installer.
Step 3: Run Executable Installer.
Step 4: Verify Python Was Installed On Windows.
Step 5: Verify Pip Was Installed.
Step 6: Add Python Path to Environment Variables (Optional)



Working with Python

Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.



Source code extension is .py
Byte code extension is .pyc (Compiled python code)

There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

Running Python in interactive mode:

Without passing python script file to the interpreter, directly execute code to Python prompt. Once you're inside the python interpreter, then you can start.

>>> print("hello world")

hello world

Relevant output is displayed on subsequent lines without the >>> symbol

>>> x=[0,1,2]

Quantities stored in memory are not displayed by default.

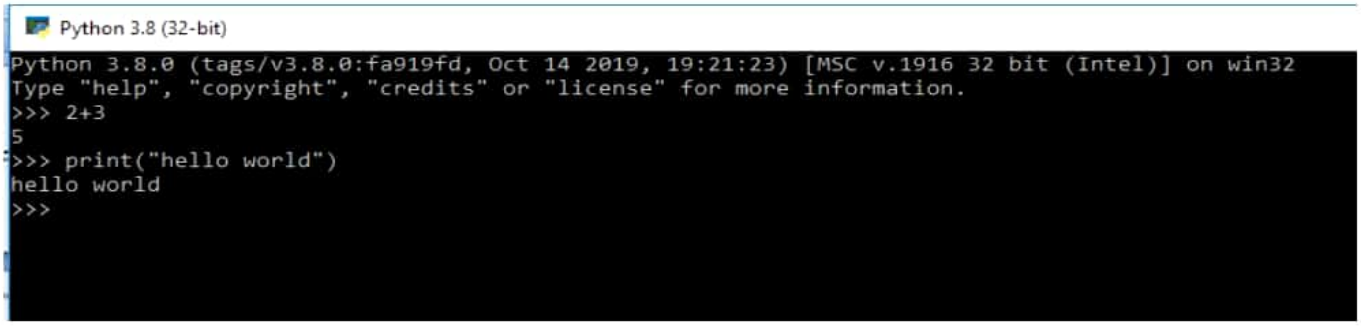
>>> x

#If a quantity is stored in memory, typing its name will display it.

[0, 1, 2]

>>> 2+3

5



The chevron at the beginning of the 1st line, i.e., the symbol >>> is a prompt the python interpreter uses to indicate that it is ready. If the programmer types 2+6, the interpreter replies 8.

Running Python in script mode:

Alternatively, programmers can store Python script source code in a file with the .py extension, and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file. For example, if you have a script name MyFile.py and you're working on Unix, to run the script you have to type:**python MyFile.py**

Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute them immediately, but when the code is more than 2-4 lines, using the script for coding can help to modify and use the code in future.

Example:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy>python e1.py
resource open
the no cant be divisible zero division by zero
resource close
finished
```

UNDERSTANDING PYTHON VARIABLES:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Rules for Python variables:

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

Assigning Values to Variables:

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

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example −

```
a= 100          # An integer assignment
b = 1000.0          # A floating point
c = "John"          # A string
print (a)
print (b)
```

print (c)

This produces the following result −

100

1000.0

John

Multiple Assignment:

Python allows you to assign a single value to several variables simultaneously.

For example :

a = b = c = 1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example −

a,b,c = 1,2,"mrcet“

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Output Variables:

The Python print statement is often used to output variables.

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5              # x is of type int
x = "mrcet "       # x is now of type str
print(x)
```

Output: mrcet

To combine both text and a variable, Python uses the **“+”** character:

Example

```
x = "awesome"
print("Python is " + x)
```

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z = x + y
print(z)
```

Output:

Python is awesome

PYTHON BASIC OPERATORS:

Operators are used to perform operations on variables and values. Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y

Assignment operators

Operator	Example	Same As
=	x = 5	x = 5

+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3

Comparison operators

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

Logical operators

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

Identity operators

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

Membership operators

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 with the specified value is not present in the object	x not in y

Bitwise operators

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

UNDERSTANDING PYTHON BLOCKS:

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
>>> for i in range(1,11):
        print(i)
        if i == 5:
            break
```

output:

1

2

3

4

5

The enforcement of indentation in Python makes the code look neat and clean. This results into Python programs that look similar and consistent.

Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
>>> if True:
        print('Hello')
        a = 5
```

Output: Hello

```
>>> if True: print('Hello'); a = 5
```

Output: Hello

A ***code block*** is a piece of Python program text that can be executed as a unit, such as a module, a class definition or a function body. Some code blocks (**like modules**) are normally executed only once, others (**like function bodies**) may be executed many times. Code blocks may textually contain other code blocks. Code blocks may invoke other code blocks (that may or may not be textually contained in them) as part of their execution, e.g., by invoking (**calling**) a function.

The following are code blocks: A module is a code block. A function body is a code block. A class definition is a code block. Each command typed interactively is a separate code block; a script file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block; a script command (a command specified on the interpreter command line with the `**-c**' option) is a code block. The file read by the built-in function execfile() is a code block. The string argument passed to the built-in function eval() and to the exec statement is a code block. And finally, the expression read and evaluated by the built-in function input() is a code block.

Some examples:

1. **if-statement**

```
pwd=input("enter string")

if pwd == 'mrcet':

    print('Logging on ...')

else:

    print('Incorrect password.')

print('All done!')
```

Output:

```
================================================
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/iff.py
================================================

enter string mrcet

Logging on ...

All done!
```

2. **if/elif-statements**

```
age = int(input('How old are you? '))

if age <= 2:

    print(' free')
```

```
elif 2 < age < 13:
    print(' child fare')
else:
    print('adult fare')
```

Output:

```
================================================
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/if1.py
================================================
How old are you? 5
 child fare
```

3. Functions

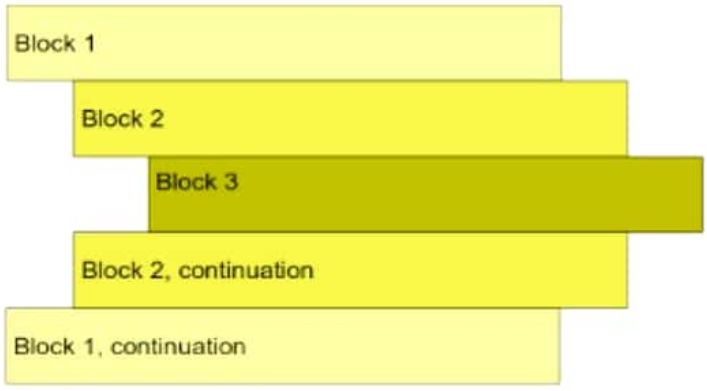
```
def my_college():
    print("Hello mrcet")
my_college()
```

Output:

```
================================================
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/if2.py
================================================
```

Hello mrcet

Sample structure of block:



PYTHON DATA TYPES:

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

Python has five standard data types −

Numbers

String

List

Tuple

Dictionary

DECLARING AND USING NUMERIC DATA TYPES:

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

For example:

var1 = 1

var2 = 10

You can delete a single object or multiple objects by using the del statement.

For example:

del var

del var_a, var_b

Python supports four different numerical types −

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

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Float can also be scientific numbers with an "e" to indicate the power of 10.

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

Examples: Here are some examples of numbers –

Example: 1

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

# To verify the type of any object in Python, use the type() function:

print(type(x))
print(type(y))
print(type(z))
```

Output:

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

Example: 2

```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
print(type(y))
print(type(z))
```

Output:

```
<class 'float'>
<class 'float'>
<class 'float'>
```

Python Casting:

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types. Casting in python is therefore done using constructor functions:

int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)

float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)

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

Examples:

Integers:

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3

Print(x)
Print(y)
Print(z)
```

Output:

```
1
2
3
```

Floats:

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2

Print(x)
Print(y)
Print(z)
Print(w)
```

Output:

```
1.0
2.8
3.0
4.2
```

Strings:

```
x = str("s1")  # x will be 's1'
y = str(2)     # y will be '2'
z = str(3.0)  # z will be '3.0'

Print(x)
Print(y)
Print(z)
```

Output:

s1
2
3.0

USING STRING DATA TYPE AND STRING OPERATIONS:

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

- 'hello' is the same as "hello".
- Strings can be output to screen using the print function. **For example: print("hello").**

2. Subsets of strings can be taken using the slice operator (**[ ] and [:]**) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
3. The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.
4. Like many other popular programming languages, strings in Python are arrays of bytes representing Unicode characters. However, Python does not 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.

Examples:

Get the character at position 1 (remember that the first character has the position 0):

```
mrcet = "Hello, World!"
print(mrcet[1])
```

Output:

e

- **Substring.** Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
print(b[2:5])
```

Output:

llo

- **The strip()** method removes any whitespace from the beginning or the end:

```
a = 'Hello,World!'

print(a.strip('He'))

string = 'android is awesome'
```

```
print(string.strip('an'))

b = 'Hello,World! Hello'

print(b.strip('Hello'))
```

Output:

```
llo,World!

droid is awesome

,World!
```

- **The len()** method returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

Output:

```
13
```

- **The lower()** method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

Output:

```
hello, world!
```

- **The upper()** method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

Output:

```
HELLO, WORLD!
```

- **The replace()** method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Output:

```
Jello, World!
```

- **The split()** method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"

b = a.split(",")

print(b)
```

Output:

```
['Hello', 'World!']
```

For example −

```
str = 'Hello World!'

print str    # Prints complete string

print str[0] # Prints first character of the string

print str[2:5] # Prints characters starting from 3rd to 5th

print str[2:] # Prints string starting from 3rd character print

str * 2 # Prints string two times

print str + "TEST" # Prints concatenated string
```

Output:

```
Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST
```

UNIT – II

CONTROL FLOWS:

if

if Statement Syntax:

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

if Statement Flowchart:

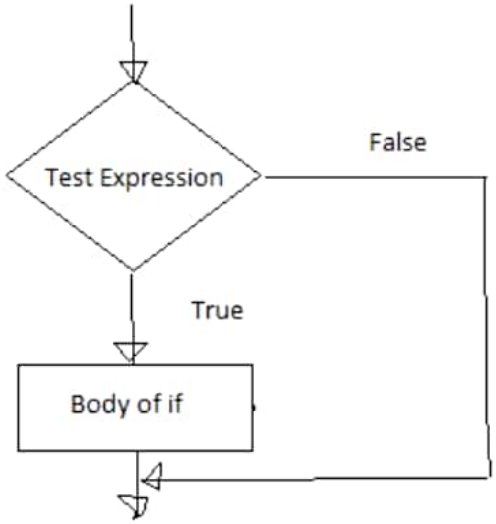


Fig: Operation of if statement

Example: Python if Statement

```
a = 3
if a > 2:
   print(a, "is greater")
print("done")

a = -1
if a < 0:
    print(a, "a is smaller")
print("Finish")
```

output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/if1.py
3 is greater
done
-1 a is smaller
```

Finish

Syntax of if - else :

```
if test expression:
        Body of if stmts
else:
        Body of else stmts
```

If - else Flowchart :

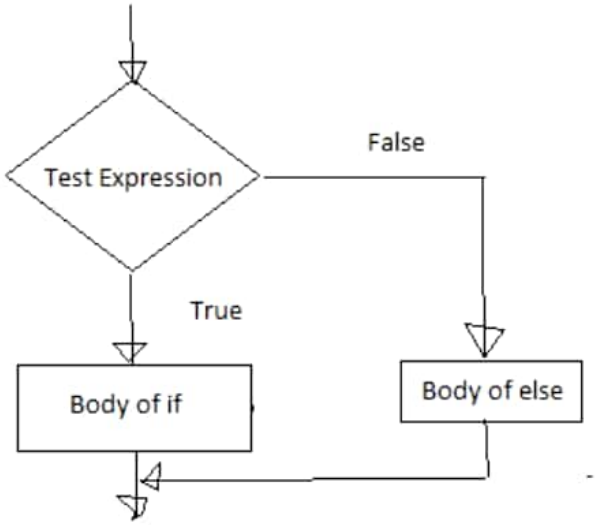


Fig: Operation of if – else statement

Example of if - else:

```
a=int(input('enter the number'))
if a>5:
   print("a is greater")
else:
   print("a is smaller than the input given")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number 2
a is smaller than the input given
```

If – elif - else:

Syntax of if – elif - else :

```
If test expression:
        Body of if stmts
elif test expression:
        Body of elif stmts
else:
        Body of else stmts
```

Flowchart of if – elif - else:

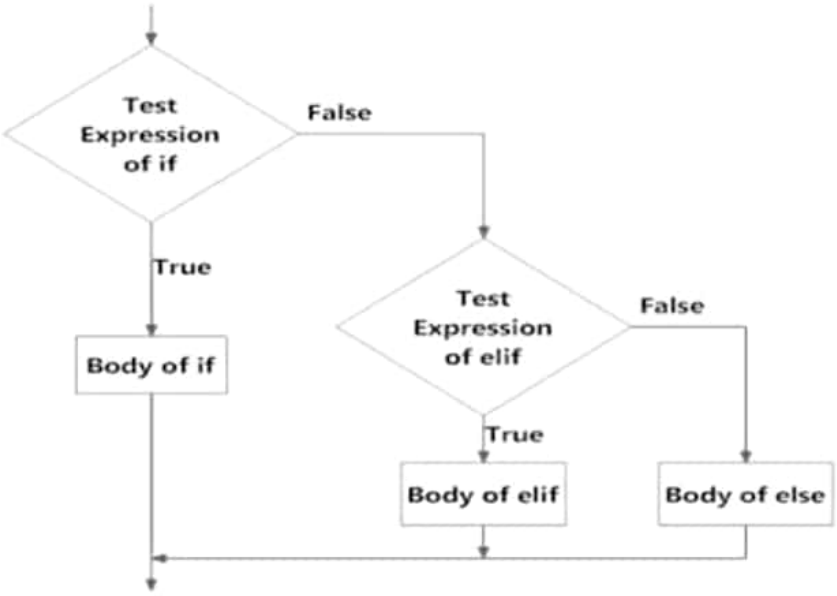


Fig: Operation of if – elif - else statement

Example of if - elif – else:

```
a=int(input('enter the number'))
b=int(input('enter the number'))
c=int(input('enter the number'))
if a>b:
   print("a is greater")
elif b>c:
   print("b is greater")
else:
   print("c is greater")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number5
enter the number2
enter the number9
a is greater
>>>
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
```

```
enter the number2
enter the number5
enter the number9
c is greater
```

PYTHON NESTED IF STATEMENTS

Syntax of nested if – elif - else :

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   elif expression4:
      statement(s)
   else:
      statement(s)
else:
   statement(s)
```

Example of Nested if:

```
a = int(input("Enter a number: "))
if a >= 0:
    if a == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Output:

```
 C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/nestedif.py

Enter a number: -1

Negative number

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/nestedif.py

Enter a number: 5

Positive number

>>>
```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/nestedif.py

Enter a number: 0

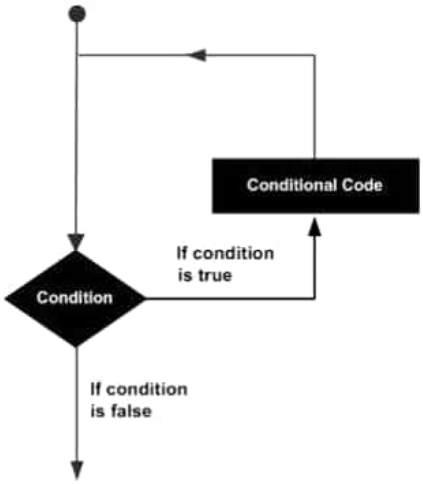
Zero

LOOPS:

Statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −

Flow chart:



There are different types of loops to handle looping requirements:

1. while loop

2. for loop

3. Nested loops

Loop control statements:

These control statements change execution from its normal sequence. Python supports the following:

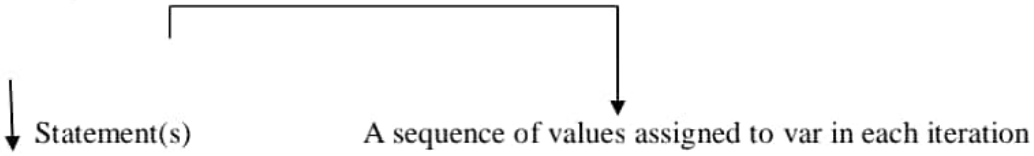
- Break statement
- Continue statement
- Pass statement

FOR LOOP USING RANGES:

For loop:

Python **for loop** is used for repeated execution of a group of statements for the desired number of times. It iterates over the items of lists, tuples, strings, the dictionaries and other iterable objects

Syntax: for var in sequence:



Holds the value of item
in sequence in each iteration

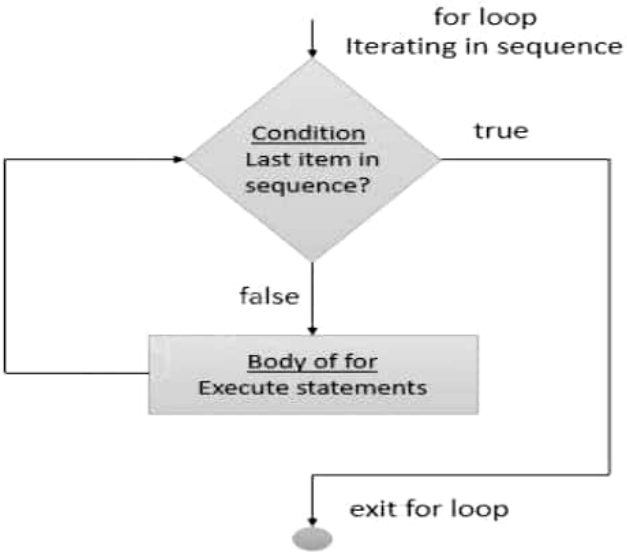
Sample Program:

```
numbers = [1, 2, 4, 6, 11, 20]
seq=0
for val in numbers:
    seq=val*val
    print(seq)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/fr.py
1
4
16
36
121
400
```

Flowchart:



Iterating over a list:

```
#list of items
list = ['M','R','C','E','T']
i = 1

#Iterating over the list
for item in list:
  print ('college ',i,' is ',item)
  i = i+1
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/lis.py
college  1  is  M
college  2  is  R
college  3  is  C
college  4  is  E
college  5  is  T
```

Iterating over a Tuple:

```
tuple = (2,3,5,7)
print ('These are the first four prime numbers ')
#Iterating over the tuple
for a in tuple:
  print (a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fr3.py
These are the first four prime numbers
```

2
3
5
7

Iterating over a dictionary:

```
#creating a dictionary
college = {"ces":"block1","it":"block2","ece":"block3"}

#Iterating over the dictionary to print keys
print ('Keys are:')
for keys in college:
  print (keys)

#Iterating over the dictionary to print values
print ('Values are:')
for blocks in college.values():
  print(blocks)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/dic.py
Keys are:
ces
it
ece
Values are:
block1
block2
block3

Iterating over a String:

```
#declare a string to iterate over
college = 'MRCET'

#Iterating over the string
for name in college:
  print (name)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strr.py
M
R
C
E
T

Range ():

range() function in for loop to iterate over numbers defined by range().

How to use range():

- range(n) : will generate numbers from 0 to (n-1)

  For example: range(8) is equivalent to [0, 1, 2, 3, 4, 5, 6, 7]

- range(x, y) : will generate numbers from x to (y-1)

  For example: range(5, 9) is equivalent to [5, 6, 7, 8]

- range(start, end, step_size) : will generate numbers from start to end with step_size as incremental factor in each iteration. step_size is default if not explicitly mentioned.

  For example: range(1, 10, 2) is equivalent to [1, 3, 5, 7, 9]

Examples:

```
x=10

for i in range(x):
    print(i)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/fr2.py
0
1
2
3
4
5
6
7
8
9
----------------------------
```

```
x=10
for i in range(6,x):
    print(i)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/fr2.py
6
7
```

```
8
9
------------------------------
x=10

for i in range(2,13,2):

    print(i)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/fr2.py
2
4
6
8
10
12
```

STRING:

Iterating over a String:

```
#declare a string to iterate over
college = 'MRCET'

#Iterating over the string
for name in college:
  print (name)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strr.py
M
R
C
E
T
```

Using range():

```
-----------------------------------

#declare a string to iterate over
college = 'MRCET'
print("the college name is")
#Iterating over the string
for i in range(len(college)):
  print (college[i])
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/rn.py =
the college name is
```

```
M
R
C
E
T
---------------------------------
#declare a string to iterate over
college = 'MRCET'
print("To print the portion of string")
#Iterating over the string
for i in college[0:3:1]:
  print (i)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strr1.py
To print the portion of string
M
R
C
---------------------------------------
#declare a string to iterate over
college = 'MRCET'
print("To print the string in reverse")
#Iterating over the string
for i in college[ : :-1]:
  print (i)
```

Output:

```
 C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strr2.py
To print the string in reverse
T
E
C
R
M
-------------------------------------
#declare a string to iterate over
college = 'MRCET'
print("To print the string in reverse using index")
#Iterating over the string
i=len(college) - 1
while i > 0:
    print(college[i])
    i=i-1
#for i in college[ : :-1]:
  #print (i)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strr3.py
```

To print the string in reverse using index
T
E
C
R

#declare a string to iterate over
i=1
college = 'MRCET'
print("To print the string in reverse using negative index")
#Iterating over the string
while i<=len(college):

    print(college[-i])
    i=i+1

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strr4.py
To print the string in reverse using index
T
E
C
R
M

USE OF WHILE LOOPS IN PYTHON:

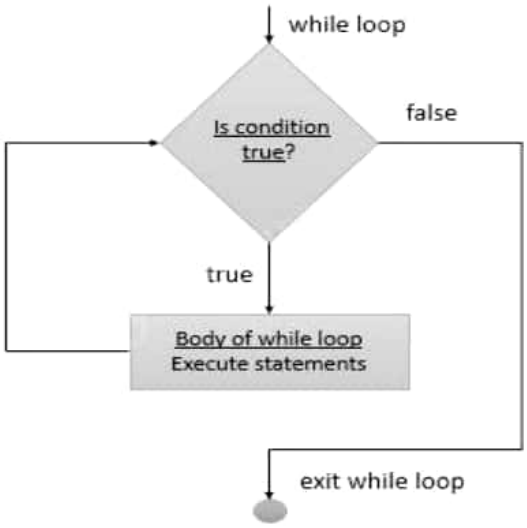
While loop:

- Loops are either infinite or conditional. Python while loop keeps reiterating a block of code defined inside it until the desired condition is met.
- The while loop contains a boolean expression and the code inside the loop is repeatedly executed as long as the boolean expression is true.
- The statements that are executed inside while can be a single line of code or a block of multiple statements.

Syntax:

```
while(expression):
        Statement(s)
```

Flowchart:



Example Programs:

1. ```
   ------------------------------------
   i=1
   while i<=6:
           print("Mrcet college")
           i=i+1
   ```

output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh1.py
Mrcet college
Mrcet college
Mrcet college
Mrcet college
Mrcet college
Mrcet college
```

2. ```
   ------------------------------------------------
   i=1

   while i<=3:
      print("MRCET",end="  ")
      j=1
      while j<=1:
         print("CSE DEPT",end="")
         j=j+1
      i=i+1
      print()
   ```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh2.py

MRCET CSE DEPT
MRCET CSE DEPT
MRCET CSE DEPT

3. ----------

```
i=1
j=1
while i<=3:
   print("MRCET",end="  ")

   while j<=1:
      print("CSE DEPT",end="")
      j=j+1
   i=i+1
  print()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh3.py

MRCET CSE DEPT
MRCET
MRCET

4. ----------

```
i = 1
while (i < 10):
      print (i)
      i = i+1
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh4.py
1
2
3
4
5
6
7
8
9

5. ----------

```
a = 1
b = 1
while (a<10):
```

```
        print ('Iteration',a)
        a = a + 1
        b = b + 1
        if (b == 4):
            break
print ('While loop terminated')
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh5.py
Iteration 1
Iteration 2
Iteration 3
While loop terminated
```

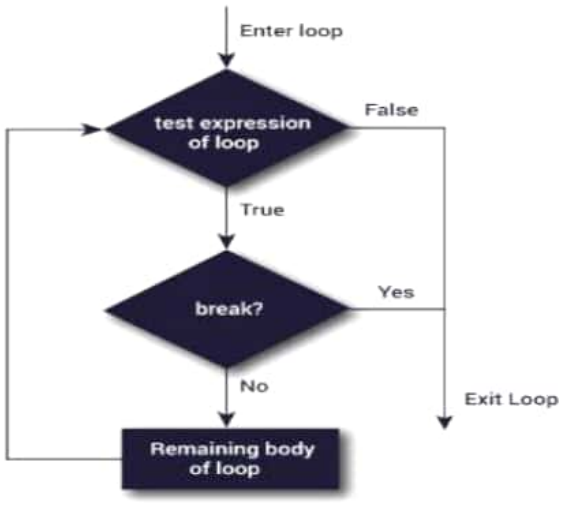
LOOP MANIPULATION USING PASS, CONTINUE, BREAK AND ELSE:

In Python, break and continue statements can alter the flow of a normal loop. Sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The break and continue statements are used in these cases.

Break:

The break statement terminates the loop containing it and control of the program flows to the statement immediately after the body of the loop. If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Flowchart:



The following shows the working of break statement in for and while loop:

```
for var in sequence:
        # code inside for loop
```

```
        If condition:
                break (if break condition satisfies it jumps to outside loop)
        # code inside for loop
# code outside for loop


while test expression

        # code inside while loop
        If condition:
                break (if break condition satisfies it jumps to outside loop)
        # code inside while loop
# code outside while loop
```

Example:

```
for val in "MRCET COLLEGE":
    if val == " ":
        break
    print(val)
print("The end")
```

Output:

```
M
R
C
E
T
The end
```

 # Program to display all the elements before number 88

```
for num in [11, 9, 88, 10, 90, 3, 19]:
   print(num)
   if(num==88):
            print("The number 88 is found")
            print("Terminating the loop")
            break
```

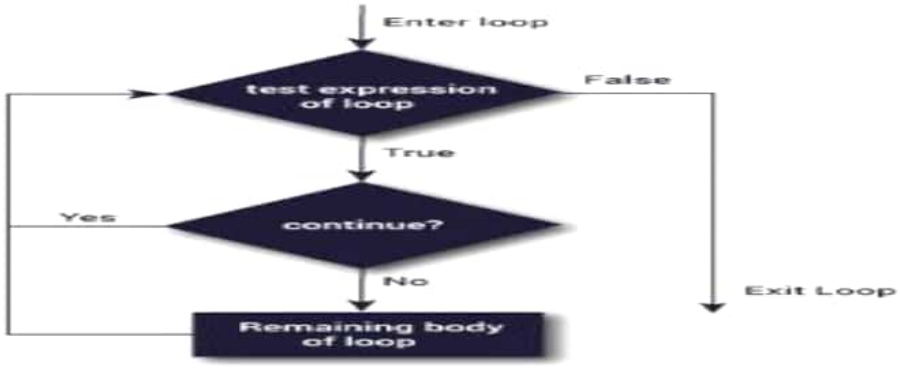
Output:

```
11
9
88
The number 88 is found
Terminating the loop
```

Continue:

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Flowchart:



The following shows the working of break statement in for and while loop:

```
for var in sequence:
        # code inside for loop
        If condition:
                continue (if break condition satisfies it jumps to outside loop)
        # code inside for loop
# code outside for loop


while test expression

        # code inside while loop
        If condition:
                continue(if break condition satisfies it jumps to outside loop)
        # code inside while loop
# code outside while loop
```

Example:

```
# Program to show the use of continue statement inside loops

for val in "string":
   if val == "i":
      continue
   print(val)
```

```
print("The end")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/cont.py
s
t
r
n
g
The end
```

```
# program to display only odd numbers

for num in [20, 11, 9, 66, 4, 89, 44]:
    # Skipping the iteration when number is even
    if num%2 == 0:
        continue
    # This statement will be skipped for all even numbers
    print(num)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/cont2.py
11
9
89
```

Pass:

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.
pass is just a placeholder for functionality to be added later.

Example:

```
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/f1.y.py
>>>
```

Similarily we can also write,

```
def f(arg): pass    # a function that does nothing (yet)
```

class C: pass # a class with no methods (yet)

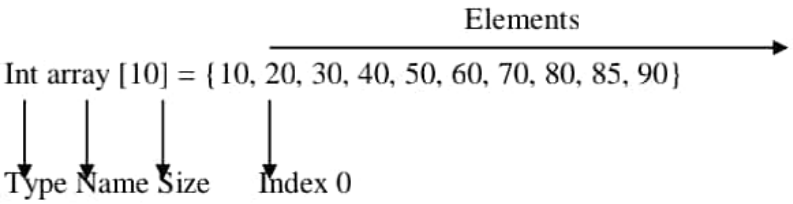
PYTHON ARRAYS:

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element**− Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 70

Basic Operations

Following are the basic operations supported by an array.

- Traverse − print all the array elements one by one.
- Insertion − Adds an element at the given index.
- Deletion − Deletes an element at the given index.
- Search − Searches an element using the given index or by the value.
- Update − Updates an element at the given index.

Array is created in Python by importing array module to the python program. Then the array is declared as shown below.

```
from array import *
arrayName=array(typecode, [initializers])
```

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are:

Typecode	Value
b	Represents signed integer of size 1 byte/td>
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Creating an array:

```
from array import *
array1 = array('i', [10,20,30,40,50])
for x in array1:
 print(x)
```

Output:

```
>>>

 RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/arr.py

10
20
30
40
50
```

Accessing Array Element

We can access each element of an array using the index of the element.

```
from array import *
array1 = array('i', [10,20,30,40,50])
```

```
print (array1[0])
print (array1[2])
```

Output:

```
RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/arr2.py
10
30
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.insert(1,60)
for x in array1:
 print(x)
```

Output:

```
==============================================
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/arr3.py
==============================================

10
60
20
30
40
50
>>>
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.remove(40)
for x in array1:
 print(x)
```

Output:

```
==========================================
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/arr4.py
==========================================
10
20
30
50
```

Search Operation

You can perform a search for an array element based on its value or its index.

Here, we search a data element using the python in-built index() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
print (array1.index(40))
```

Output:

```
==========================================
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/arr5.py
==========================================
3
>>>
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Here, we simply reassign a new value to the desired index we want to update.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1[2] = 80
for x in array1:
 print(x)
```

Output:

```
==========================================
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/arr6.py
==========================================
10
20
80
40
50
```

UNIT – III

FUNCTIONS:

Function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. It avoids repetition and makes code reusable.

Basically, we can divide functions into the following two types:

1. **Built-in functions** - Functions that are built into Python.

   Ex: abs(),all().ascii(),bool()…… so on….

   integer = -20

   print('Absolute value of -20 is:', abs(integer))

   Output:

   Absolute value of -20 is: 20

2. **User-defined functions** - Functions defined by the users themselves.

```
def add_numbers(x,y):
  sum = x + y
  return sum

print("The sum is", add_numbers(5, 20))
```

   Output:

   The sum is 25

There are three types of Python function arguments using which we can call a function.

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

Syntax:

```
def functionname():
    statements
      .
      .
      .
functionname()
```

Function definition consists of following components:

1. Keyword **def** indicates the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A **colon (:)** to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Example:

```
def hf():
    hello world
hf()
```

In the above example we are just trying to execute the program by calling the function. So it will not display any error and no output on to the screen but gets executed.

To get the statements of function need to be use print().

#calling function in python:

```
def hf():
    print("hello world")
hf()
```

Output:

```
hello world
```

```
def hf():
    print("hw")
    print("gh kfjg 66666")
hf()
hf()
hf()
```

Output:

```
hw
gh kfjg 66666
hw
gh kfjg 66666
hw
gh kfjg 66666
```

```
def add(x,y):
    c=x+y
    print(c)
add(5,4)
```

Output:

```
9
```

```
def add(x,y):
    c=x+y
    return c
print(add(5,4))
```

Output:

```
9
```

```
def add_sub(x,y):
    c=x+y
    d=x-y
    return c,d
print(add_sub(10,5))
```

Output:

```
(15, 5)
```

The **return** statement is used to exit a function and go back to the place from where it was called. This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the **None** object.

```
def hf():
    return "hw"
print(hf())
```

Output:

```
hw
```

```
def hf():
    return "hw"
hf()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu.py
>>>
```

```
def hello_f():
    return "hellocollege"
print(hello_f().upper())
```

Output:

```
HELLOCOLLEGE
```

 # Passing Arguments

```
def hello(wish):
    return '{}'.format(wish)
print(hello("mrcet"))
```

Output:

mrcet

Here, the function wish() has two parameters. Since, we have called this function with two arguments, it runs smoothly and we do not get any error. If we call it with different number of arguments, the interpreter will give errors.

```
def wish(name,msg):
  """This function greets to
  the person with the provided message"""
     print("Hello",name + ' ' + msg)
wish("MRCET","Good morning!")
```

Output:

Hello MRCET Good morning!

Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> wish("MRCET")    # only one argument
TypeError: wish() missing 1 required positional argument: 'msg'
>>> wish()    # no arguments
TypeError: wish() missing 2 required positional arguments: 'name' and 'msg'
```

```
def hello(wish,hello):
   return  “hi”  '{},{}'.format(wish,hello)
print(hello("mrcet","college"))
```

Output:

himrcet,college

#Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed.

(Or)

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called **keyword arguments** - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two *advantages* - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

```
def func(a, b=5, c=10):
        print 'a is', a, 'and b is', b, 'and c is', c

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

Note:

The function named func has one parameter without default argument values, followed by two parameters with default argument values.

In the first usage, func(3, 7), the parameter a gets the value 3, the parameter b gets the value 5 and c gets the default value of 10.

In the second usage func(25, c=24), the variable a gets the value of 25 due to the position of the argument. Then, the parameter c gets the value of 24 due to naming i.e. keyword arguments. The variable b gets the default value of 5.

In the third usage func(c=50, a=100), we use keyword arguments completely to specify the values. Notice, that we are specifying value for parameter c before that for a even though a is defined before c in the function definition.

For example: if you define the function like below

```
def func(b=5, c=10,a):  # shows error : non-default argument follows default argument
```

```
---------------------------------------------------

def print_name(name1, name2):

  """ This function prints the name """

  print (name1 + " and " + name2 + " are friends")

#calling the function

print_name(name2 = 'A',name1 = 'B')
```

Output:

B and A are friends

#Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=)

```
def hello(wish,name='you'):

    return '{},{}'.format(wish,name)

print(hello("good morning"))
```

Output:

good morning,you

```
---------------------------------------------

def hello(wish,name='you'):

    return '{},{}'.format(wish,name)     //print(wish  +  ‘  ‘ +  name)

print(hello("good morning","nirosha"))   // hello("good morning","nirosha")
```

Output:

good morning,nirosha // good morning nirosha

Note: Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

def hello(name='you', wish):

Syntax Error: non-default argument follows default argument

---

def sum(a=4, b=2): #2 is supplied as default argument

""" This function will print sum of two numbers

if the arguments are not supplied

it will add the default value """

print (a+b)

sum(1,2) #calling with arguments

sum( ) #calling without arguments

Output:

3

6

#Variable-length arguments

Sometimes you may need more arguments to process function then you mentioned in the definition. If we don't know in advance about the arguments needed in function, we can use variable-length arguments also called arbitrary arguments.

For this an asterisk (*) is placed before a parameter in function definition which can hold non-keyworded variable-length arguments and a double asterisk (**) is placed before a parameter in function which can hold keyworded variable-length arguments.

If we use one asterisk (*) like *var, then all the positional arguments from that point till the end are collected as a tuple called 'var' and if we use two asterisks (**) before a variable like **var, then all the positional arguments from that point till the end are collected as a dictionary called 'var'.

```
def wish(*names):
  """This function greets all
  the person in the names tuple."""

  # names is a tuple with arguments
  for name in names:
     print("Hello",name)
```

```
wish("MRCET","CSE","SIR","MADAM")
```

Output:

Hello MRCET

Hello CSE

Hello SIR

Hello MADAM

Some examples on functions:

 # To display vandemataram by using function use no args no return type

```
#function defination
def display():
    print("vandemataram")
print("i am in main")

#function call
display()
print("i am in main")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
i am in main
vandemataram
i am in main
```

#Type1 : No parameters and no return type

```
def Fun1() :
    print("function 1")
Fun1()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

function 1

#Type 2: with param with out return type

```
def fun2(a) :
    print(a)
```

```
fun2("hello")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Hello
```

#Type 3: without param with return type

```
def fun3():
    return "welcome to python"
print(fun3())
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

welcome to python
```

#Type 4: with param with return type

```
def fun4(a):
    return a
print(fun4("python is better then c"))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

python is better then c
```

#Program to find area of a circle using function use single return value function with argument.

```
pi=3.14
def areaOfCircle(r):

    return pi*r*r
r=int(input("Enter radius of circle"))

print(areaOfCircle(r))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
Enter radius of circle 3
28.259999999999998
```

#Program to write sum different product and using arguments with return value function.

```
def calculete(a,b):
    total=a+b
    diff=a-b
    prod=a*b
    div=a/b
    mod=a%b
    return total,diff,prod,div,mod
a=int(input("Enter a value"))
b=int(input("Enter b value"))
 #function call
s,d,p,q,m = calculete(a,b)
print("Sum= ",s,"diff= ",d,"mul= ",p,"div= ",q,"mod= ",m)
#print("diff= ",d)
#print("mul= ",p)
#print("div= ",q)
#print("mod= ",m)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
Enter a value 5
Enter b value 6
Sum=  11 diff=  -1 mul=  30 div=  0.8333333333333334 mod=  5
```

#program to find biggest of two numbers using functions.

```
def biggest(a,b):
    if a>b :
        return a
```

```
    else :
        return b


a=int(input("Enter a value"))
b=int(input("Enter b value"))

 #function call
big= biggest(a,b)
print("big number= ",big)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
Enter a value 5
Enter b value-2
big number=  5
```

#program to find biggest of two numbers using functions. (nested if)

```
def biggest(a,b,c):
    if a>b :
        if a>c :
            return a
        else :
            return c
    else :
        if b>c :
            return b
        else :
            return c


a=int(input("Enter a value"))
b=int(input("Enter b value"))
c=int(input("Enter c value"))
 #function call
big= biggest(a,b,c)
print("big number= ",big)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Enter a value 5
Enter b value -6
Enter c value 7
big number=  7
```

#Writer a program to read one subject mark and print pass or fail use single return values function with argument.

```
def result(a):
    if a>40:
       return "pass"
    else:
        return "fail"
a=int(input("Enter one subject marks"))

print(result(a))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
Enter one subject marks 35
fail
```

#Write a program to display mrecet cse dept 10 times on the screen. (while loop)

```
def usingFunctions():
    count =0
    while count<10:
        print("mrcet cse dept",count)
        count=count+1

usingFunctions()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
mrcet cse dept 0
mrcet cse dept 1
mrcet cse dept 2
mrcet cse dept 3
mrcet cse dept 4
mrcet cse dept 5
mrcet cse dept 6
mrcet cse dept 7
mrcet cse dept 8
mrcet cse dept 9
```

ANONYMOUS FUNCTIONS:

Anonymous function is a function i.e. defined without name.

While normal functions are defined using the **def keyword**.
Anonymous functions are defined using **lambda keyword** hence anonymous functions are also called **lambda functions**.

Syntax: lambda arguments: expression

- Lambda function can have any no. of arguments for any one expression.
- The expression is evaluated and returns.

Use of Lambda functions:

- Lambda functions are used as nameless functions for a short period of time.
- In python lambda functions are an argument to higher order functions.
- Lambda functions are used along with built-in functions like filter(),map() and reduce() etc….

 # Write a program to double a given number

```
double = lambda x:2*x

print(double(5))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
10
```

#Write a program to sum of two numbers

```
add = lambda x,y:x+y

print(add(5,4))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
9
```

#Write a program to find biggest of two numbers

```
biggest = lambda x,y: a if x>y else y

print(biggest(20,30))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

30
```

FRUITFUL FUNCTIONS (FUNCTION RETURNING VALUES):

We write functions that return values, which we will call **fruitful functions**. We have seen the return statement before, but in a fruitful function the return statement includes a **return value**. This

statement means: "Return immediately from this function and use the following expression as a return value."

 # returns the area of a circle with the given radius:

```
def area(radius):
  temp = 3.14 * radius**2
  return temp
print(area(4))
```

(or)

```
def area(radius):
  return 3.14 * radius**2
print(area(2))
```

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Since these return statements are in an alternative conditional, only one will be executed.

As soon as a return statement executes, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called dead code.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

This function is incorrect because if x happens to be 0, both conditions is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is None, which is not the absolute value of 0.

```
>>> print absolute_value(0)
None
```

By the way, Python provides a built-in function called abs that computes absolute values.

 # Write a Python function that takes two lists and returns True if they have at least one common member.

```
def common_data(list1, list2):
    for x in list1:
        for y in list2:
            if x == y:
                result = True
                return result
print(common_data([1,2,3,4,5], [1,2,3,4,5]))
print(common_data([1,2,3,4,5], [1,7,8,9,510]))
print(common_data([1,2,3,4,5], [6,7,8,9,10]))
```

Output:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py
True
True
None
```

SCOPE OF THE VARIABLES IN A FUNCTION - GLOBAL AND LOCAL VARIABLES:

The scope of a variable determines its accessibility and availability in different portions of a program. Their availability depends on where they are defined. Similarly, life is a period in which the variable is stored in the memory.

Depending on the scope and the lifetime, there are two kinds of variables in Python.

- Local Variables
- Global Variables

Local Variables vs. Global Variables

Here are some of the points to list out the difference between global and local variable for their proper understanding.

- Variables or parameters defined inside a function are called local variables as their scope is limited to the function only. On the contrary, Global variables are defined outside of the function.
- Local variables can't be used outside the function whereas a global variable can be used throughout the program anywhere as per requirement.
- The lifetime of a local variable ends with the termination or the execution of a function, whereas the lifetime of a global variable ends with the termination of the entire program.
- The variable defined inside a function can also be made global by using the global statement.

```
def function_name(args):
  .....
  global x      #declaring global variable inside a function
  .....
```

 # create a global variable

```
x = "global"

def f():
    print("x inside :", x)

f()
print("x outside:", x)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

x inside : global

x outside: global

 # create a local variable

```
def f1():

    y = "local"

    print(y)

f1()
```

Output:
local

- If we try to access the local variable outside the scope for example,

```
def f2():
    y = "local"

f2()
print(y)
```

Then when we try to run it shows an error,

```
Traceback (most recent call last):
  File "C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py", line 6, in <module>
    print(y)
NameError: name 'y' is not defined
```

The output shows an error, because we are trying to access a local variable y in a global scope whereas the local variable only works inside f2() or local scope.

 # use local and global variables in same code

```
x = "global"

def f3():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)

f3()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
globalglobal
local
```

- In the above code, we declare x as a global and y as a local variable in the f3(). Then, we use multiplication operator * to modify the global variable x and we print both x and y.
- After calling the f3(), the value of x becomes global global because we used the x * 2 to print two times global. After that, we print the value of local variable y i.e local.

 # use Global variable and Local variable with same name

```
x = 5

def f4():
    x = 10
    print("local x:", x)

f4()
print("global x:", x)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
local x: 10
global x: 5
```

POWERFUL LAMDA FUNCTION IN PYTHON:

Lambda functions are used along with built-in functions like filter(), map() and reduce()etc….

Filter():

- The filter functions takes list as argument.

- The filter() is called when new list is returned which contains items for which the function evaluates to true.
- Filter:The filter() function returns an iterator were the items are filtered through a function to test if the item is accepted or not.

  Syntax: filter(function, iterable)

#Write a program to filter() function to filter out only even numbers from the given list

```
myList =[1,2,3,4,5,6]

newList = list(filter(lambda x: x%2 ==0,myList ))
print(newList)
```

Output:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py

[2, 4, 6]
```

#Write a program for filter() function to print the items greater than 4

```
list1 = [10,2,8,7,5,4,3,11,0, 1]

result = filter (lambda x: x > 4, list1)

print(list(result))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/m1.py =

[10, 8, 7, 5, 11]
```

Map() :

- Map() function in python takes a function & list.
- The function is called with all items in the list and a new list is returned which contains items returned by that function for each item.
- Map applies a function to all the items in an list.
- The advantage of the lambda operator can be seen when it is used in combination with the map() function.
- map() is a function with two arguments:

  Syntax: r = map(func, seq)

#Write a program for map() function to double all the items in the list

```
myList =[1,2,3,4,5,6,7,8,9,10]
newList = list(map(lambda x: x*2,myList))
print(newList)
```

Output:

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

 # Write a program to separate the letters of the word "hello" and add the letters as items of the list.

```
letters = []
letters = list(map(lambda x:x,"hello"))
print(letters)
```

Output:

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py

['h', 'e', 'l', 'l', 'o']

#Write a program for map() function to double all the items in the list?

```
def addition(n):
    return n + n

numbers = (1, 2, 3, 4)

result = map(addition, numbers)

print(list(result))
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/m1.py =

[2, 4, 6, 8]

Reduce():

- Applies the same operation to items of sequence.
- Use the result of the first operation for the next operation
- Returns an item, not a list.

- Reduce: The reduce(fun, seq)function is used to apply a particular
- function passed in its argument to all of the list elementsmentioned in the sequence passed along. This function is defined in "functools" module.

#Write a program to find some of the numbers for the elements of the list by using reduce()

```
import functools
myList =[1,2,3,4,5,6,7,8,9,10]
print(functools.reduce(lambda x,y: x+y,myList))
```

Output:

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py

55

#Write a program for reduce() function to print the product of items in a list

```
from functools import reduce

list1 = [1,2,3,4,5]

product = reduce (lambda x, y: x*y, list1)

print(product)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/m1.py =

120

UNIT – IV

DATA STRUCTURES:

Data Structures in Python provides / include Python list, Python Tuple, Python set, and Python dictionaries with their syntax and examples.

Here in this data structure we will come to know as a way of organizing and storing data such that we can access and modify it efficiently

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
>>> print(list1)
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
----------
>>> x=list()
>>> x
[]
----------
>>> tuple1=(1,2,3,4)
>>> x=list(tuple1)
>>> x
[1, 2, 3, 4]
```

The list data type has some more methods. Here are all of the methods of list objects:

List Operations:

- Del()

- Append()
- Extend()
- Insert()
- Pop()
- Remove()
- Reverse()
- Sort()

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1])           #deletes the index position 1 in a list
>>> x
[5, 8, 6]
------------
>>> del(x)
>>> x                   # complete list gets deleted
```

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
>>> x
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
>>> y=[3,6,9,1]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 3, 6, 9, 1]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
>>> x.insert(2,10)   #insert(index no, item to be inserted)
>>> x
```

[1, 2, 10, 4, 6, 7]

>>> x.insert(4,['a',11])

>>> x

[1, 2, 10, 4, ['a', 11], 6, 7]

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

>>> x=[1, 2, 10, 4, 6, 7]

>>> x.pop()

7

>>> x

[1, 2, 10, 4, 6]

>>> x=[1, 2, 10, 4, 6]

>>> x.pop(2)

10

>>> x

[1, 2, 4, 6]

Remove: The **remove()** method removes the specified item from a given list.

>>> x=[1,33,2,10,4,6]

>>> x.remove(33)

>>> x

[1, 2, 10, 4, 6]

>>> x.remove(4)

>>> x

[1, 2, 10, 6]

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
>>> x.reverse()
>>> x
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
>>> x.sort()
>>> x
[1, 2, 3, 4, 5, 6, 7]
----------
>>> x=[10,1,5,3,8,7]
>>> x.sort()
>>> x
[1, 3, 5, 7, 8, 10]
```

Slicing: Slice out substrings, sub lists, sub Tuples using index.

[Start: stop: steps]

- Slicing will start from index and will go up to **stop** in **step** of steps.
- Default value of start is 0,
- Stop is last index of list
- And for step default is 1

Example:

```
>>> x='computer'
>>> x[1:4]
'omp'
>>> x[1:6:2]
'opt'
>>> x[3:]
'puter'
```

```
>>> x[:5]
'compu'
>>> x[-1]
'r'
>>> x[-3:]
'ter'
>>> x[:-2]
'comput'
>>> x[::-2]
'rtpo'
>>> x[::-1]
'retupmoc'
```

List:

```
>>> list1=range(1,6)
>>> list1
range(1, 6)
>>> print(list1)
range(1, 6)
>>> list1=[1,2,3,4,5,6,7,8,9,10]
>>> list1[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list1[:1]
[1]
>>> list1[2:5]
[3, 4, 5]
>>> list1[:6]
[1, 2, 3, 4, 5, 6]
>>> list1[1:2:4]
[2]
```

```
>>> list1[1:8:2]
[2, 4, 6, 8]
```

Tuple:

```
>>> list1=(11,12,13,14)
>>> list1[:2]
(11, 12)
```

To create a slice:

```
>>> print(slice(3))
slice(None, 3, None)
>>> print(slice(2))
slice(None, 2, None)
>>> print(slice(1,6,4))
slice(1, 6, 4)
```

To get substring from a given string using slice object:

```
>>> pystr='python'
>>> x=slice(3)
>>> print(pystr[x])
Pyt
```

Using –ve index:

```
>>> pystr='python'
>>> x=slice(1,-3,1)
>>> print(pystr[x])
>>> yt
```

To get sublist and sub-tuple from a given list and tuple respectively:

```
>>> list1=['m','r','c','e','t']
>>> tup1=('c','o','l','l','e','g','e')
>>> x=slice(1,4,1)
>>> print(tup1[x])
('o', 'l', 'l')
```

```
>>> print(list1[x])
['r', 'c', 'e']
>>> x=slice(1,5,2)
>>> print(list1[x])
['r', 'e']
>>> print(tup1[x])
('o', 'l')
>>> x=slice(-1,-4,-1)  #negative index
>>> print(list1[x])
['t', 'e', 'c']
>>> x=slice(-1,-4,-1) #negative index
>>> print(tup1[x])
('e', 'g', 'e')
>>> print(list1[0:3]) #extending indexing syntax
['m', 'r', 'c']
```

Tuples:

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

- Supports all operations for sequences.
- Immutable, but member objects may be mutable.
- If the contents of a list shouldn't change, use a tuple to prevent items from accidently being added, changed, or deleted.
- Tuples are more efficient than list due to python's implementation.

We can construct tuple in many ways:

```
X=()     #no item tuple
X=(1,2,3)
X=tuple(list1)
X=1,2,3,4
```

Example:

```
>>> x=(1,2,3)
>>> print(x)
(1, 2, 3)
>>> x
```

```
(1, 2, 3)
----------------------
>>> x=()
>>> x
()
---------------------------
>>> x=[4,5,66,9]
>>> y=tuple(x)
>>> y
(4, 5, 66, 9)
---------------------------
>>> x=1,2,3,4
>>> x
(1, 2, 3, 4)
```

Some of the operations of tuple are:

- Access tuple items
- Change tuple items
- Loop through a tuple
- Count()
- Index()
- Length()

Access tuple items: Access tuple items by referring to the index number, inside square brackets

```
>>> x=('a','b','c','g')
>>> print(x[2])
c
```

Change tuple items: Once a tuple is created, you cannot change its values. Tuples are unchangeable.

```
>>> x=(2,5,7,'4',8)
>>> x[1]=10
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    x[1]=10
TypeError: 'tuple' object does not support item assignment
>>> x
(2, 5, 7, '4', 8)     # the value is still the same
```

Loop through a tuple: We can loop the values of tuple using for loop

```
>>> x=4,5,6,7,2,'aa'
>>> for i in x:
        print(i)
```

4
5
6
7
2
aa

Count (): Returns the number of times a specified value occurs in a tuple

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.count(2)
4
```

Index (): Searches the tuple for a specified value and returns the position of where it was found

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.index(2)
1
```

(Or)

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=x.index(2)
>>> print(y)
1
```

Length (): To know the number of items or values present in a tuple, we use len().

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=len(x)
>>> print(y)
12
```

Set:

A set is a collection which is unordered and unindexed with no duplicate elements. In Python sets are written with curly brackets.

- To create an empty set we use **set()**
- Curly braces '{}' or the set() function can be used to create sets

We can construct tuple in many ways:

```
X=set()
X={3,5,6,8}
X=set(list1)
```

Example:

```
>>> x={1,3,5,6}
```

```
>>> x
{1, 3, 5, 6}
----------------------
>>> x=set()
>>> x
set()
---------------------
>>> list1=[4,6,"dd",7]
>>> x=set(list1)
>>> x
{4, 'dd', 6, 7}
```

- We cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Some of the basic set operations are:

- Add()
- Remove()
- Len()
- Item in x
- Pop
- Clear

Add (): To add one item to a set use the add () method. To add more than one item to a set use the update () method.

```
>>> x={"mrcet","college","cse","dept"}
>>> x.add("autonomous")
>>> x
{'mrcet', 'dept', 'autonomous', 'cse', 'college'}

----
>>> x={1,2,3}
>>> x.update("a","b")
>>> x
{1, 2, 3, 'a', 'b'}

----------------
>>> x={1,2,3}
>>> x.update([4,5],[6,7,8])
>>> x
{1, 2, 3, 4, 5, 6, 7, 8}
```

Remove (): To remove an item from the set we use remove or discard methods.

```
>>> x={1, 2, 3, 'a', 'b'}
>>> x.remove(3)
```

```
>>> x
{1, 2, 'a', 'b'}
```

Len (): To know the number of items present in a set, we use len().

```
>>> z={'mrcet', 'dept', 'autonomous', 'cse', 'college'}
>>> len(z)
5
```

Item in X: you can loop through the set items using a for loop.

```
>>> x={'a','b','c','d'}
>>> for item in x:
        print(item)

c
d
a
b
```

pop (): This method is used to remove an item, but this method will remove the **last** item. Remember that sets are unordered, so you will not know what item that gets removed.

```
>>> x={1, 2, 3, 4, 5, 6, 7, 8}
>>> x.pop()
1
>>> x
{2, 3, 4, 5, 6, 7, 8}
```

Clear (): This method will the set as empty.

```
>>> x={2, 3, 4, 5, 6, 7, 8}
>>> x.clear()
>>> x
set()
```

The set also consist of some mathematical operations like:

Intersection	AND	&	
Union	OR	|	
Symmetric Diff	XOR	^	
Diff	In set1 but not in set2	set1-set2	
Subset	set2 contains set1	set1<=set2	
Superset	set1 contains set2	set1>=set2	

Some examples:

```
>>> x={1,2,3,4}
>>> y={4,5,6,7}
>>> print(x|y)
{1, 2, 3, 4, 5, 6, 7}
----------------------------
>>> x={1,2,3,4}
>>> y={4,5,6,7}
```

```
>>> print(x&y)
{4}
---------------------------
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> print(A-B)
{1, 2, 3}
---------------------------

>>> B = {4, 5, 6, 7, 8}
>>> A = {1, 2, 3, 4, 5}
>>> print(B^A)
{1, 2, 3, 6, 7, 8}
```

Dictionaries:

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

- Key-value pairs
- Unordered

We can construct or create dictionary like:

```
X={1:'A',2:'B',3:'c'}
X=dict([('a',3) ('b',4)]
X=dict('A'=1,'B' =2)
```

Examples:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2004}
------------------
```

To access specific value of a dictionary, we must pass its key,

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> x=dict1["brand"]
>>> x
'mrcet'
---------------------
```

To access keys and values and items of dictionary:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1.keys()
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['mrcet', 'college', 2004])
>>> dict1.items()
dict_items([('brand', 'mrcet'), ('model', 'college'), ('year', 2004)])
-----------------------------------------------
>>> for items in dict1.values():
        print(items)
```

```
mrcet
college
2004

>>> for items in dict1.keys():
        print(items)


brand
model
year

>>> for i in dict1.items():
        print(i)


('brand', 'mrcet')
('model', 'college')
('year', 2004)
```

Some of the operations are:

- Add/change
- Remove
- Length
- Delete

Add/change values: You can change the value of a specific item by referring to its key name

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1["year"]=2005
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2005}
```

Remove(): It removes or pop the specific item of dictionary.

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> print(dict1.pop("model"))
college
>>> dict1
{'brand': 'mrcet', 'year': 2005}
```

Delete: Deletes a particular item.

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> del x[5]
>>> x
```

Length: we use len() method to get the length of dictionary.

```
>>>{1: 1, 2: 4, 3: 9, 4: 16}
{1: 1, 2: 4, 3: 9, 4: 16}
>>> y=len(x)
>>> y
4
```

Iterating over (key, value) pairs:

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> for key in x:
        print(key, x[key])


1 1
2 4
3 9
4 16
5 25
>>> for k,v in x.items():
        print(k,v)


1 1
2 4
3 9
4 16
5 25
```

List of Dictionaries:

```
>>> customers = [{"uid":1,"name":"John"},
   {"uid":2,"name":"Smith"},
        {"uid":3,"name":"Andersson"},
        ]
>>> >>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'Andersson'}]

## Print the uid and name of each customer
>>> for x in customers:
        print(x["uid"], x["name"])


1 John
2 Smith
3 Andersson

## Modify an entry, This will change the name of customer 2 from Smith to Charlie
>>> customers[2]["name"]="charlie"
```

```
>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'charlie'}]

## Add a new field to each entry

>>> for x in customers:
        x["password"]="123456" # any initial value


>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 2, 'name': 'Smith', 'password': '123456'}, {'uid': 3,
'name': 'charlie', 'password': '123456'}]

## Delete a field
>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]

>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}]

## Delete all fields

>>> for x in customers:
        del x["uid"]


>>> x
{'name': 'John', 'password': '123456'}
```

Sequences:
A sequence is a succession of values bound together by a container that reflects their type. Almost every stream that you put in python is a sequence. Some of them are:

- String
- List
- Tuples
- Range object

String: A string is a group of characters. Since Python has no provision for arrays, we simply use strings. This is how we declare a string. We can use a pair of single or double quotes. Every string object is of the type 'str'.

```
>>> type("name")
<class 'str'>
```

```
>>> name=str()
>>> name
''
>>> a=str('mrcet')
>>> a
'mrcet'
>>> a=str(mrcet)
>>> a[2]
'c'
```

List: A list is an ordered group of items. To declare it, we use square brackets.

```
>>> college=["cse","it","eee","ece","mech","aero"]
>>> college[1]
'it'
>>> college[:2]
['cse', 'it']
>>> college[:3]
['cse', 'it', 'eee']
>>> college[3:]
['ece', 'mech', 'aero']
>>> college[0]="csedept"
>>> college
['csedept', 'it', 'eee', 'ece', 'mech', 'aero']
```

Tuple: It is an immutable group of items. When we say immutable, we mean we cannot change a single value once we declare it.

```
>>> x=[1,2,3]
>>> y=tuple(x)
>>> y
(1, 2, 3)

>>> hello=tuple(["mrcet","college"])
>>> hello
('mrcet', 'college')
```

Range object: A range() object lends us a range to iterate on; it gives us a list of numbers.

```
>>> a=range(4)
>>> type(a)
<class 'range'>

>>> for i in range(1,6,2):
        print(i)


1
3
```

5

Some of the python sequence operations and functions are:

1. Indexing
2. Slicing
3. Adding/Concatenation
4. Multiplying
5. Checking membership
6. Iterating
7. Len()
8. Min()
9. Max()
10. Sum()
11. Sorted()
12. Count()
13. Index()

1. Indexing
   Access any item in the sequence using its index.

string	List

>>> x='mrcet'
>>> print(x[2])
c
	>>> x=['a','b','c']
	>>> print(x[1])
	b

2. Slicing
   Slice out substrings, sub lists, sub tuples using index
   [start : stop : step size]

```
>>> x='computer'
>>> x[1:4]
'omp'
>>> x[1:6:2]
'opt'
>>> x[3:]
'puter'
>>> x[:5]
'compu'
>>> x[-1]
'r'
```

```
>>> x[-3:]
'ter'
>>> x[:-2]
'comput'
>>> x[::-2]
'rtpo'
>>> x[::-1]
'retupmoc'
```

3. Adding/concatenation:
   Combine 2 sequences of same type using +.

string	List

>>> x='mrcet' + 'college'
>>> print(x)
Mrcetcollege
	>>> x=['a','b'] + ['c']
	>>> print(x)
	['a', 'b', 'c']

4. Multiplying:
   Multiply a sequence using *.

string	List

>>> x='mrcet'*3
>>> x
'mrcetmrcetmrcet'
	>>> x=[3,4]*2
	>>> x
	[3, 4, 3, 4]

5. Checking Membership:
   Test whether an item is in or not in a sequence.

string	List

>>> x='mrcet'
>>> print('c' in x)
True
	>>> x=['a','b','c']
	>>> print('a' not in x)
	False

6. Iterating:
   Iterate through the items in asequence

```
>>> x=[1,2,3]
>>> for item in x:
        print(item*2)
```

2
4
6

If we want to display the items of a given list with index then we have to use "enumerate" keyword.

```
>>> x=[5,6,7]
>>> for item,index in enumerate(x):
        print(item,index)


0 5
1 6
2 7
```

7. len():

It will count the number of items in a given sequence.

string	List

>>> x="mrcet"
>>> print(len(x))
5
	>>> x=["aa","b",'c','cc']
	>>> print(len(x))
	4

8. min():

Finds the minimum item in a given sequence lexicographically.

string	List

>>> x="mrcet"
>>> print(min(x))
c
	>>> x=["apple","ant1","ant"]
	>>> print(min(x))
	ant

It is an alpha-numeric type but cannot mix types.

```
>>> x=["apple","ant1","ant",11]
>>> print(min(x))

Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    print(min(x))
TypeError: '<' not supported between instances of 'int' and 'str'
```

9. max():

Finds the maximum item in a given sequence

string	List

>>> x='cognizant'
>>> print(max(x))
z
	>>> x=["hello","yummy","zebra"]
	>>> print(max(x))
	zebra

It is an alpha-numeric type but cannot mix types.

```
>>> x=["hello","yummy1","zebra1",22]
>>> print(max(x))

Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    print(max(x))
TypeError: '>' not supported between instances of 'int' and 'str'
```

10. Sum:
    Finds the sum of items in a sequence

```
>>> x=[1,2,3,4,5]
>>> print(sum(x))
15

>>> print(sum(x[-2:]))
9
```

Entire string must be numeric type.

```
>>> x=[1,2,3,4,5,"mrcet"]
>>> print(sum(x))

Traceback (most recent call last):
  File "<pyshell#83>", line 1, in <module>
    print(sum(x))
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

11. Sorted():
    Returns a new list of items in sorted order but does not change the original list.

string	List

>>> x='college'
>>> print(sorted(x))
['c', 'e', 'e', 'g', 'l', 'l', 'o']
	>>> x=['a','r','g','c','j','z']
	>>> print(sorted(x))
	['a', 'c', 'g', 'j', 'r', 'z']

12. Count():
    It returns the count of an item

string	List

>>> x='college'
>>> print(x.count('l'))
2
>>> 'college'.count('l')
2
	>>> x=['a','b','a','a','c','a']
	>>> print(x.count('a'))
	4

13. Index()
    Returns the index of first occurrence

string	List

>>> x='college'
>>> print(x.index('l'))
2
	>>> x=['a','b','a','a','c','a']
	>>> print(x.index('a'))
	0

COMPREHENSIONS:

List:

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> list1=[]

>>> for x in range(10):

        list1.append(x**2)

>>> list1

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

(or)

This is also equivalent to

```
>>> list1=list(map(lambda x:x**2, range(10)))

>>> list1
```

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

(or)

Which is more concise and redable.

```
>>> list1=[x**2 for x in range(10)]

>>> list1

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

Similarily some examples:

```
>>> x=[m for m in range(8)]
>>> print(x)
[0, 1, 2, 3, 4, 5, 6, 7]

>>> x=[z**2 for z in range(10) if z>4]
>>> print(x)
[25, 36, 49, 64, 81]

>>> x=[x ** 2  for x in range (1, 11)   if  x % 2 == 1]
>>> print(x)
[1, 9, 25, 49, 81]

>>> a=5
>>> table = [[a, b, a * b] for b in range(1, 11)]
>>> for i in table:
        print(i)

[5, 1, 5]
[5, 2, 10]
[5, 3, 15]
[5, 4, 20]
[5, 5, 25]
[5, 6, 30]
[5, 7, 35]
[5, 8, 40]
[5, 9, 45]
[5, 10, 50]
```

Tuple:

Tuple Comprehensions are special: The result of a tuple comprehension is special. You might expect it

to produce a tuple, but what it does is produce a special "generator" object that we can iterate over.

For example:

```
>>> x = (i for i in 'abc')  #tuple comprehension
>>> x
<generator object <genexpr> at 0x033EEC30>

>>> print(x)
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

So, given the code

```
>>> x = (i for i in 'abc')
>>> for i in x:
        print(i)

a
b
c
```

Create a list of 2-tuples like (number, square):

```
>>> z=[(x, x**2) for x in range(6)]
>>> z
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

Set:

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}

>>> x={3*x for x in range(10) if x>5}
>>> x
{24, 18, 27, 21}
```

DICTIONARY:

Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> z={x: x**2 for x in (2,4,6)}
>>> z
```

```
{2: 4, 4: 16, 6: 36}

>>> dict11 = {x: x*x for x in range(6)}
>>> dict11
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

UNIT – V

Sorting:

Bubble Sort:

It is a simple sorting algorithm which sorts 'n' number of elements in the list by comparing the ach pair of adjacent items and swaps them if they are in wrong order.

Algorithm:

1. Starting with the first element (index=0), compare the current element with the next element of a list.
2. If the current element is greater (>) than the next element of the list then swap them.
3. If the current element is less (<) than the next element of the list move to the next element.
4. Repeat step 1 until it correct order is framed.

For ex: list1= [10, 15, 4, 23, 0]

If > --- yes ---- swap

If < --- No ---- Do nothing/remains same

} so here we are comparing values again and again, so we use loops.

#Write a python program to arrange the elements in ascending order using bubble sort:

```
list1=[9,16,6,26,0]
print("unsorted list1 is", list1)
for j in range(len(list1)-1):
    for i in range(len(list1)-1):
        if list1[i]>list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:

```
unsorted list1 is [9, 16, 6, 26, 0]
[9, 16, 6, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 26, 0]
```

[9, 6, 16, 0, 26]

[6, 9, 16, 0, 26]
[6, 9, 16, 0, 26]
[6, 9, 0, 16, 26]
[6, 9, 0, 16, 26]

[6, 9, 0, 16, 26]
[6, 0, 9, 16, 26]
[6, 0, 9, 16, 26]
[6, 0, 9, 16, 26]

[0, 6, 9, 16, 26]
[0, 6, 9, 16, 26]
[0, 6, 9, 16, 26]
[0, 6, 9, 16, 26]

sorted list is [0, 6, 9, 16, 26]

#If we want to reduce no of iterations/steps in output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/bubb.py

```
list1=[9,16,6,26,0]
print("unsorted list1 is", list1)
for j in range(len(list1)-1,0,-1):
    for i in range(j):
        if list1[i]>list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/bubb2.py
unsorted list1 is [9, 16, 6, 26, 0]
[9, 16, 6, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 0, 26]

```
[6, 9, 16, 0, 26]
[6, 9, 16, 0, 26]
[6, 9, 0, 16, 26]

[6, 9, 0, 16, 26]
[6, 0, 9, 16, 26]

[0, 6, 9, 16, 26]

sorted list is [0, 6, 9, 16, 26]
```

 # In a different way:

```
list1=[9,16,6,26,0]
print("unsorted list1 is", list1)
for j in range(len(list1)-1):
    for i in range(len(list1)-1-j):
        if list1[i]>list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/bubb3.py

unsorted list1 is [9, 16, 6, 26, 0]
[9, 16, 6, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 0, 26]

[6, 9, 16, 0, 26]
[6, 9, 16, 0, 26]
[6, 9, 0, 16, 26]

[6, 9, 0, 16, 26]
[6, 0, 9, 16, 26]

[0, 6, 9, 16, 26]

sorted list is [0, 6, 9, 16, 26]
```

 # Program to give input from the user to sort the elements

```
list1=[]
num=int(input("enter how many numbers:"))
print("enter values")
for k in range(num):
    list1.append(int(input()))
print("unsorted list1 is", list1)
for j in range(len(list1)-1):
    for i in range(len(list1)-1):
        if list1[i]>list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/bubb4.py

enter how many numbers:5
enter values
5
77
4
66
30
unsorted list1 is [5, 77, 4, 66, 30]
[5, 77, 4, 66, 30]
[5, 4, 77, 66, 30]
[5, 4, 66, 77, 30]
[5, 4, 66, 30, 77]

[4, 5, 66, 30, 77]
[4, 5, 66, 30, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]

[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]

[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]
```

[4, 5, 30, 66, 77]

sorted list is [4, 5, 30, 66, 77]

#bubble sort program for descending order

```
list1=[9,16,6,26,0]
print("unsorted list1 is", list1)
for j in range(len(list1)-1):
    for i in range(len(list1)-1):
        if list1[i]<list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-2/pyyy/bubbdesc.py
unsorted list1 is [9, 16, 6, 26, 0]
[16, 9, 6, 26, 0]
[16, 9, 6, 26, 0]
[16, 9, 26, 6, 0]
[16, 9, 26, 6, 0]

[16, 9, 26, 6, 0]
[16, 26, 9, 6, 0]
[16, 26, 9, 6, 0]
[16, 26, 9, 6, 0]

[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]

[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]

sorted list is [26, 16, 9, 6, 0]
```

Selection Sort:

Sort (): Built-in list method

Sorted (): built in function

- Generally this algorithm is called as in-place comparison based algorithm. We compare numbers and place them in correct position.
- Search the list and find out the min value, this we can do it by min () method.
- We can take min value as the first element of the list and compare with the next element until we find small value.

Algorithm:

1. Starting from the first element search for smallest/biggest element in the list of numbers.
2. Swap min/max number with first element
3. Take the sub-list (ignore sorted part) and repeat step 1 and 2 until all the elements are sorted.

#Write a python program to arrange the elements in ascending order using selection sort:

```
list1=[5,3,7,1,9,6]
print(list1)
for i in range(len(list1)):
    min_val=min(list1[i:])
    min_ind=list1.index(min_val)
    list1[i],list1[min_ind]=list1[min_ind],list1[i]
    print(list1)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/selectasce.py

```
[5, 3, 7, 1, 9, 6]
[1, 3, 7, 5, 9, 6]
[1, 3, 7, 5, 9, 6]
[1, 3, 5, 7, 9, 6]
[1, 3, 5, 6, 9, 7]
[1, 3, 5, 6, 7, 9]
[1, 3, 5, 6, 7, 9]
```

#Write a python program to arrange the elements in descending order using selection sort:

```
list1=[5,3,7,1,9,6]

print(list1)

for i in range(len(list1)):

    min_val=max(list1[i:])

    min_ind=list1.index(min_val)

    list1[i],list1[min_ind]=list1[min_ind],list1[i]

print(list1)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/selecdecs.py

[5, 3, 7, 1, 9, 6]

[9, 7, 6, 5, 3, 1]

Note: If we want the elements to be sorted in descending order use max () method in place of min ().

Insertion Sort:

- Insertion sort is not a fast sorting algorithm. It is useful only for small datasets.
- It is a simple sorting algorithm that builds the final sorted list one item at a time.

Algorithm:

1. Consider the first element to be sorted & the rest to be unsorted.
2. Take the first element in unsorted order (u1) and compare it with sorted part elements(s1)
3. If u1<s1 then insert u1 in the correct order, else leave as it is.
4. Take the next element in the unsorted part and compare with sorted element.
5. Repeat step 3 and step 4 until all the elements get sorted.

 # Write a python program to arrange the elements in ascending order using insertion sort (with functions)

```
def insertionsort(my_list):
#we need to sorrt the unsorted part at a time.
   for index in range(1,len(my_list)):
      current_element=my_list[index]
      pos=index
      while current_element<my_list[pos-1]and pos>0:
         my_list[pos]=my_list[pos-1]
         pos=pos-1
      my_list[pos]=current_element
list1=[3,5,1,0,10,2]                    num=int(input("enter how many elements to be in list"))
insertionsort(list1)                    list1=[int(input()) for i in range (num)]
print(list1)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/inserti.py

[0, 1, 2, 3, 5, 10]

 # Write a python program to arrange the elements in descending order using insertion sort (with functions)

```
def insertionsort(my_list):
#we need to sorrt the unsorted part at a time.
    for index in range(1,len(my_list)):
        current_element=my_list[index]
        pos=index
        while current_element>my_list[pos-1]and pos>0:
            my_list[pos]=my_list[pos-1]
            pos=pos-1
        my_list[pos]=current_element
#list1=[3,5,1,0,10,2]
#insertionsort(list1)
#print(list1)
num=int(input("enter how many elements to be in list"))
list1=[int(input())for i in range(num)]
insertionsort(list1)
print(list1)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/insertdesc.py
enter how many elements to be in list 5
8
1
4
10
2
[10, 8, 4, 2, 1]
```

Merge Sort:

Generally this merge sort works on the basis of divide and conquer algorithm. The three steps need to be followed is divide, conquer and combine. We will be dividing the unsorted list into sub list until the single element in a list is found.

Algorithm:

1. Split the unsorted list.
2. Compare each of the elements and group them
3. Repeat step 2 until whole list is merged and sorted.

 # Write a python program to arrange the elements in ascending order using Merge sort (with functions)

```
def mergesort(list1):
    if len(list1)>1:
        mid=len(list1)//2
        left_list=list1[:mid]
        right_list=list1[mid:]
        mergesort(left_list)
        mergesort(right_list)
        i=0
        j=0
        k=0
        while i<len(left_list) and j<len(right_list):
            if left_list[i]<right_list[j]:
                list1[k]=left_list[i]
                i=i+1
                k=k+1
            else:
                list1[k]=right_list[j]
                j=j+1
                k=k+1
        while i<len(left_list):
            list1[k]=left_list[i]
            i=i+1
            k=k+1
        while j<len(right_list):
            list1[k]=right_list[j]
            j=j+1
            k=k+1
num=int(input("how many numbers in list1"))
list1=[int(input()) for x in range(num)]
mergesort(list1)
print("sorted list1",list1)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/merg.py
```

```
how many numbers in list15
5
9
10
1
66
sorted list1 [1, 5, 9, 10, 66]
```

Quick Sort:

Algorithm:

1. Select the pivot element
2. Find out the correct position of pivot element in the list by rearranging it.
3. Divide the list based on pivot element
4. Sort the sub list recursively

Note: Pivot element can be first, last, random elements or median of three values.

In the following program we are going to write 3 functions. The first function is to find pivot element and its correct position. In second function we divide the list based on pivot element and sort the sub list and third function (main fun) is to print input and output.

 # Write a python program to arrange the elements in ascending order using Quick sort (with functions)

```
#To get the correct position of pivot element:
def pivot_place(list1,first,last):
    pivot=list1[first]
    left=first+1
    right=last
    while True:
        while left<=right and list1[left]<=pivot:
            left=left+1
        while left<=right and list1[right]>=pivot:
            right=right-1
        if right<left:
            break
        else:
            list1[left],list1[right]=list1[right],list1[left]
    list1[first],list1[right]=list1[right],list1[first]
    return right
#second function
def quicksort(list1,first,last):
    if first<last:
        p=pivot_place(list1,first,last)
        quicksort(list1,first,p-1)
        quicksort(list1,p+1,last)
```

```
#main fun
list1=[56,25,93,15,31,44]
n=len(list1)
quicksort(list1,0,n-1)
print(list1)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/qucksort.py
[15, 25, 31, 44, 56, 93]
```

 # Write a python program to arrange the elements in descending order using Quick sort (with functions)

```
#To get the correct position of pivot element:
def pivot_place(list1,first,last):
    pivot=list1[first]
    left=first+1
    right=last
    while True:
        while left<=right and list1[left]>=pivot:
            left=left+1
        while left<=right and list1[right]<=pivot:
            right=right-1
        if right<left:
            break
        else:
            list1[left],list1[right]=list1[right],list1[left]
    list1[first],list1[right]=list1[right],list1[first]
    return right
def quicksort(list1,first,last):
    if first<last:
        p=pivot_place(list1,first,last)
        quicksort(list1,first,p-1)
        quicksort(list1,p+1,last)
#main fun
list1=[56,25,93,15,31,44]
n=len(list1)
quicksort(list1,0,n-1)
print(list1)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/qukdesc.py

[93, 56, 44, 31, 25, 15]

Linked Lists:

Linked lists are one of the most commonly used data structures in any programming language. Linked Lists, on the other hand, are different. Linked lists, do not store data at contiguous memory locations. For each item in the memory location, linked list stores value of the item and the reference or pointer to the next item. One pair of the linked list item and the reference to next item constitutes a node.

The following are different types of linked lists.

- Single Linked List
  A single linked list is the simplest of all the variants of linked lists. Every node in a single linked list contains an item and reference to the next item and that's it.

- Doubly Linked List
- Circular Linked List
- Linked List with Header
- Sorted Linked List

 # Python program to create a linked list and display its elements.

The program creates a linked list using data items input from the user and displays it.

Solution:

1. Create a class Node with instance variables data and next.
2. Create a class Linked List with instance variables head and last_node.
3. The variable head points to the first element in the linked list while last_node points to the last.
4. Define methods append and display inside the class Linked List to append data and display the linked list respectively.
5. Create an instance of Linked List, append data to it and display the list.

Program:

```
class Node:
   def __init__(self, data):
      self.data = data
      self.next = None

class LinkedList:
   def __init__(self):
       self.head = None
       self.last_node = None

   def append(self, data):
```

```
        if self.last_node is None:
            self.head = Node(data)
            self.last_node = self.head
        else:
            self.last_node.next = Node(data)
            self.last_node = self.last_node.next

    def display(self):
        current = self.head
        while current is not None:
            print(current.data, end = ' ')
            current = current.next

a_llist = LinkedList()
n = int(input('How many elements would you like to add? '))
for i in range(n):
    data = int(input('Enter data item: '))
    a_llist.append(data)
print('The linked list: ', end = '')
a_llist.display()
```

Program Explanation

1. An instance of Linked List is created.
2. The user is asked for the number of elements they would like to add. This is stored in n.
3. Using a loop, data from the user is appended to the linked list n times.
4. The linked list is displayed.

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/link1.py
How many elements would you like to add? 5
Enter data item: 4
Enter data item: 4
Enter data item: 6
Enter data item: 8
Enter data item: 9
The linked list: 4 4 6 8 9
```

Stacks:

Stack works on the principle of "Last-in, first-out". Also, the inbuilt functions in Python make the code short and simple. To add an item to the top of the list, i.e., to push an item, we use append() function and to pop out an element we use pop() function.

 # Python code to demonstrate Implementing stack using list

```
stack = ["Amar", "Akbar", "Anthony"]
stack.append("Ram")
stack.append("Iqbal")
print(stack)
print(stack.pop())
print(stack)
print(stack.pop())
print(stack)
```

Output:

['Amar', 'Akbar', 'Anthony', 'Ram', 'Iqbal']

Iqbal

['Amar', 'Akbar', 'Anthony', 'Ram']

Ram

['Amar', 'Akbar', 'Anthony']

Queues:

Queue works on the principle of "First-in, first-out". Time plays an important factor here. We saw that during the implementation of stack we used append() and pop() function which was efficient and fast because we inserted and popped elements from the end of the list, but in queue when insertion and pops are made from the beginning of the list, it is slow. This occurs due to the properties of list, which is fast at the end operations but slow at the beginning operations, as all other elements have to be shifted one by one. So, we prefer the use of collections. Deque over list, which was specially designed to have fast appends and pops from both the front and back end.

#Python code to demonstrate Implementing Queue using deque and list

```
from collections import deque
queue = deque(["Ram", "Tarun", "Asif", "John"])
print(queue)
queue.append("Akbar")
print(queue)
queue.append("Birbal")
print(queue)
print(queue.popleft())
print(queue.popleft())
print(queue)
```

Output:

deque(['Ram', 'Tarun', 'Asif', 'John'])
deque(['Ram', 'Tarun', 'Asif', 'John', 'Akbar'])
deque(['Ram', 'Tarun', 'Asif', 'John', 'Akbar', 'Birbal'])

```
Ram
Tarun
deque(['Asif', 'John', 'Akbar', 'Birbal'])
```